The *Bro* 0.8 User Manual

Vern Paxson
Lawrence Berkeley National Laboratory
and
ICSI Center for Internet Research
International Computer Science Institute
vern@icir.org

March 21, 2004

Contents

1 Introduction **9**

2 Getting Started **12**

2.1 Running Bro . . . 12

2.1.1 Building and installing Bro . . . 12

2.1.2 Using Bro interactively . . . 14

2.1.3 Specifying policy scripts . . . 15

2.1.4 Running Bro on network traffic . . . 15

2.1.5 Modifying Bro policy . . . 16

2.1.6 Bro flags and run-time environment . . . 17

2.2 Helper utilities . . . 19

2.2.1 Scripts . . . 19

2.2.2 The *hf* utility . . . 20

2.2.3 The *cf* utility . . . 20

3 Values, Types, and Constants **21**

3.1 Overview . . . 21

3.1.1 Bro Types . . . 21

3.1.2 Type Conversions . . . 22

3.2 Booleans . . . 22

3.2.1 Boolean Constants . . . 22

3.2.2 Logical Operators . . . 22

3.3 Numeric Types . . . 23

3.3.1 Numeric Constants . . . 23

3.3.2 Mixing Numeric Types . . . 23

3.3.3 Arithmetic Operators . . . 23

3.3.4 Comparison Operators . . . 24

3.4 Enumerations . . . 24

3.5 Strings . . . 24

3.5.1 String Constants . . . 24

3.5.2 String Operators . . . 25

3.6 Patterns . . . 25

3.6.1 Pattern Constants . . . 25

3.6.2 Pattern Operators . . . 26

3.7 Temporal Types . . . 27
3.7.1 Temporal Constants . . . 27
3.7.2 Temporal Operators . . . 27
3.8 Port Type . . . 28
3.8.1 Port Constants . . . 28
3.8.2 Port Operators . . . 29
3.9 Address Type . . . 29
3.9.1 Address Constants . . . 29
3.9.2 Address Operators . . . 29
3.10 Net Type . . . 29
3.10.1 Net Constants . . . 30
3.10.2 Net Operators . . . 30
3.11 Records . . . 30
3.11.1 Defining records . . . 30
3.11.2 Record Constants . . . 31
3.11.3 Accessing Fields Using “$” . . . 32
3.11.4 Record Assignment . . . 32
3.12 Tables . . . 33
3.12.1 Declaring Tables . . . 33
3.12.2 Initializing Tables . . . 34
3.12.3 Table Attributes . . . 35
3.12.4 Accessing Tables . . . 37
3.12.5 Table Assignment . . . 37
3.12.6 Deleting Table Elements . . . 38
3.13 Sets . . . 38
3.14 Files . . . 39
3.15 Functions . . . 39
3.16 Event handlers . . . 41
3.17 The any type . . . 42

4 Statements and Expressions 43
4.1 Statements . . . 43
4.2 Expressions . . . 46

5 Global and Local Variables 53
5.1 Overview . . . 53
5.1.1 Scope . . . 53
5.1.2 Modifiability . . . 54
5.1.3 Typing . . . 54
5.1.4 Initialization . . . 55
5.1.5 Attributes . . . 55
5.1.6 Refinement . . . 56

6 Predefined Variables and Functions 57
6.1 Predefined Variables . . . 57
6.1.1 active.bro . . . 57
6.1.2 alert.bro . . . 57
6.1.3 anon.bro . . . 57
6.1.4 backdoor.bro . . . 58
6.1.5 bro.init . . . 60
6.1.6 code-red.bro . . . 63
6.1.7 conn.bro . . . 63
6.1.8 demux.bro . . . 64
6.1.9 dns.bro . . . 64
6.1.10 dns-mapping.bro . . . 65
6.1.11 finger.bro . . . 65
6.1.12 ftp.bro . . . 65
6.1.13 hot.bro . . . 67
6.1.14 hot-ids.bro . . . 68
6.1.15 http.bro . . . 68
6.1.16 http-abstract.bro . . . 69
6.1.17 http-request.bro . . . 69
6.1.18 icmp.bro . . . 70
6.1.19 ident.bro . . . 70
6.1.20 interconn.bro . . . 70
6.1.21 login.bro . . . 72
6.1.22 mime.bro . . . 74
6.1.23 ntp.bro . . . 74
6.1.24 port-names.bro . . . 74
6.1.25 portmapper.bro . . . 75
6.1.26 rules.bro . . . 76
6.1.27 scan.bro . . . 76
6.1.28 site.bro . . . 79
6.1.29 smtp.bro . . . 79
6.1.30 smtp-relay.bro . . . 80
6.1.31 software.bro . . . 81
6.1.32 ssh.bro . . . 81
6.1.33 stepping.bro . . . 81
6.1.34 tftp.bro . . . 83
6.1.35 udp.bro . . . 83
6.1.36 weird.bro . . . 83
6.1.37 worm.bro . . . 83
6.1.38 Uncategorized . . . 84
6.2 Predefined Functions . . . 85
6.2.1 Run-time errors for non-existing connections . . . 91
6.2.2 Run-time errors for strings with NULs . . . 91
6.2.3 Functions for manipulating strings . . . 91
6.2.4 Functions for manipulating time . . . 91

7 Analyzers and Events **92**
7.1 Activating an Analyzer . . . 92
7.1.1 Loading Analyzers . . . 92
7.1.2 Filtering . . . 93
7.2 General Processing Events . . . 94
7.3 Generic Connection Analysis . . . 95
7.3.1 The `connection` record . . . 96
7.3.2 Definitions of connections . . . 98
7.3.3 Generic TCP connection events . . . 99
7.3.4 The `tcp` analyzer . . . 100
7.3.5 The `udp` analyzer . . . 100
7.3.6 Connection summaries . . . 101
7.3.7 Connection functions . . . 103
7.4 Site-specific information . . . 105
7.4.1 Site variables . . . 105
7.4.2 Site-specific functions . . . 106
7.5 The `hot` Analyzer . . . 106
7.5.1 `hot` variables . . . 106
7.5.2 `hot` functions . . . 110
7.6 The `scan` Analyzer . . . 112
7.6.1 `scan` variables . . . 112
7.6.2 `scan` functions . . . 114
7.6.3 `scan` event handlers . . . 114
7.7 The `port-name` Module . . . 115
7.8 The `mt` Module . . . 115
7.9 The `log` Module . . . 115
7.10 The `active` Module . . . 117
7.11 The `demux` Module . . . 117
7.12 The `dns` Module . . . 118
7.12.1 The `dns_mapping` record . . . 118
7.12.2 `dns` variables . . . 119
7.12.3 `dns` event handlers . . . 119
7.13 The `finger` Analyzer . . . 120
7.13.1 `finger` variables . . . 120
7.13.2 `finger` event handlers . . . 120
7.14 The `frag` Module . . . 121
7.15 The `hot-ids` Module . . . 121
7.16 The `ftp` Analyzer . . . 122
7.16.1 The `ftp_session_info` record . . . 122
7.16.2 `ftp` variables . . . 123
7.16.3 `ftp` functions . . . 125
7.16.4 `ftp` event handlers . . . 126
7.17 The `http` Analyzer . . . 127
7.17.1 `http` variables . . . 127
7.17.2 `http` event handlers . . . 128
7.18 The `ident` Analyzer . . . 128

7.18.1 ident variables 129
7.18.2 ident event handlers 129
7.19 The login Analyzer 130
7.19.1 login analyzer confusion 131
7.19.2 login variables 131
7.19.3 login functions 137
7.19.4 login event handlers 138
7.20 The portmapper Analyzer 142
7.20.1 portmapper variables 143
7.20.2 portmapper functions 144
7.20.3 portmapper event handlers 146
7.21 The analy Analyzer 148
7.22 The signature Module 148
7.23 The SSL Analyzer 149
7.23.1 The x509 record 150
7.23.2 The ssl_connection_info record 150
7.23.3 SSL variables 151
7.23.4 SSL event handlers 152
7.24 The weird Module 154
7.24.1 Actions for “weird” events 154
7.24.2 weird variables 154
7.24.3 weird functions 156
7.24.4 Events handled by conn_weird 156
7.24.5 Events handled by conn_weird_addl 160
7.24.6 Events handled by flow_weird 161
7.24.7 Events handled by net_weird 162
7.24.8 Events generated by the standard scripts 163
7.24.9 Additional handlers for “weird” events 163
7.25 The icmp Analyzer 164
7.26 The stepping Analyzer 164
7.27 The ssh-stepping Module 164
7.28 The backdoor Analyzer 164
7.29 The interconn Analyzer 164

8 Signatures 165
8.1 Overview 165
8.2 Signature language 165
8.2.1 Conditions 166
8.2.2 Actions 168
8.3 snort2bro 168

9 Interactive Debugger 170
9.1 Overview 170
9.2 A Sample Session 170
9.3 Usage 172
9.4 Notes and Limitations 172

9.5 Reference 172

10 Missing Documentation 176

10.1 The use of *prefixes* 176
10.2 The tcpdump save file that Bro writes 176
10.3 The `bro.init` initialization file 176
10.4 Assignment operators such as `+=` 176
10.5 The notion of redefinition/refinement 176
10.6 The logging model 176
10.7 Timer management 177
10.8 SYN-FIN filtering 177
10.9 Split routing 177
10.10 Scan dropping 177
10.11 Operator precedence 177
10.12 Partial connections 177
10.13 Packet drops 177
10.14 The `@load` directive 177
10.15 Global statements 177
10.16 Inserting tables into tables 177
10.17 Demultiplexing 178
10.18 Bro init file 178
10.19 Hostnames vs. addresses 178
10.20 The hot-report script 178
10.21 Use of libpcap/BPF 178
10.22 The problem of evasion 178
10.23 Backscatter 178
10.24 Playing back traces 178
10.25 Discarders 178
10.26 Differences between this release and the previous one 178
10.27 Alert cascade 179
10.28 The need for subtyping 179
10.29 The need for CIDR masks 179
10.30 The wish list 179
10.31 Known bugs 179

List of Figures

7.1 print-filter prints out the tcpdump filter your Bro script would use and then exits. 94
7.2 Definition of the net_stats record. 96
7.3 Definition of conn_id and connection records. 97
7.4 Sample definition of log_hook 116
7.5 Definition of the dns_mapping record. 118
7.6 Definition of the ftp_session_info record 122
7.7 Example of FTP log file entries for a single FTP session. 124
7.8 Example of HTTP log file entries for a single HTTP session. 127
7.9 Definition of the x509 record 150
7.10 Definition of the ssl_connection_info record 150
7.11 Example of SSL log file with a single SSL session. 152

8.1 Definition of the signature_state record. 168

List of Tables

6.1 Different types of directions for set_contents_file 89

7.1 TCP and UDP connection states, as stored in an endpoint record. 97
7.2 Summaries of connection states, as reported in red files. 102
7.3 Different connection states to use when calling check_hot. 111
7.4 Different types of confusion that login analyzer can report. 132
7.5 Types of calls to the RPC portmapper service. 142
7.6 Types of RPC status codes. 146
7.7 endpoint_stats fields for summarizing connection endpoint statistics 148
7.8 Possible actions to take for signatures matches. *signatures-log* defaults to open_log_file("signatures"). 149
7.9 Different types of possible actions to take for “weird” events. 155

Chapter 1

Introduction

Bro is an intrusion detection system that works by passively watching traffic seen on a network link. It is built around an *event engine* that pieces network packets into events that reflect different types of activity. Some events are quite low-level, such as the monitor seeing a connection attempt; some are specific to a particular network protocol, such as an FTP request or reply; and some reflect fairly high-level notions, such as a user having successfully authenticated during a login session.

Bro runs the events produced by the event engine through a *policy script*, which you (the Bro administrator) supply, though in general you will do so by using large portions of the scripts ("*analyzers*"; see below) that come with the Bro distribution.

You write policy scripts in "Bro", a specialized language geared towards network analysis in general and security analysis in particular. Bro scripts are made up of *event handlers* that specify what to do whenever a given event occurs. Event handlers can maintain and update global state information, write arbitrary information to disk files, generate new events, call functions (either user-defined or predefined), generate *alerts* that produce *syslog* messages, and invoke arbitrary shell commands. These latter might terminate a running connection or talk to your border router to install an ACL prohibiting traffic from a particular host, for example.

The Bro language is strongly typed and includes a bunch of types designed to aid analyzing network traffic. It also supports *implicit typing*, meaning that often you don't need to explicitly indicate a variable's type because Bro can figure it out from context. This feature makes the strong typing a bit less of a pain, while retaining its bug-finding benefits.

For high performance, Bro relies on use of an efficient *packet filter* to capture only a (hopefully small) subset of the traffic that transits the link it monitors. Related to this, Bro comes with a set of *analyzers*, that is, scripts for analyzing different protocols and different types of activity. In general you can pick and choose among these for which types of analysis you want to enable, and Bro will only capture traffic relating to the analyzers you choose. Thus, you can control how much work Bro has to do by the analyzers you designate, a potentially major consideration if the monitored link has a high volume of traffic.

Experience has shown that the policy scripts often require tailoring to each environment in which they're used; but if the tailoring is done by editing the analyzers supplied with the Bro distribution, you wind up with multiple copies of the analyzers, all slightly different, such that when you want to make a general change to all of them, it takes careful (and tedious) editing to correctly apply the change to all of the copies.

Consequently, Bro emphasizes the use of tables and sets of values as ways to codify policy particulars such as which hosts should generate alerts if seen engaged in various types of connections, which usernames are sensitive and should trigger alerts when used, and so on. The various analyzers are written such that you can (often) customize them by simply changing variables associated with the analyzer. Furthermore, Bro supports a notion of *refining* the initialization of a variable, so that, in a *separate* file from the one defining an analyzer, you can either *(i) redefine* the variable's initial value, *(ii) add* new elements to a given table, set or pattern, or *(iii) remove* elements from a given table or set. In a nutshell, refinement allows you to specify particular policies in terms of their *differences* from existing policies, rather than in their entirety.

You can find an overview of Bro in the paper "Bro: A System for Detecting Network Intruders in Real-Time," Proceedings of the 1998 USENIX Security Symposium [Pa98], and a revised version in *Computer Networks* [Pa99]. A copy of the latter is included in the Bro distribution.

Using this manual. This manual is intended to provide full documentation for users of Bro, both those who wish to write Bro scripts to use Bro's existing analyzers, and those who wish to implement event engine support for new Bro analyzers. The current version of the manual is *incomplete*; in particular, it does not discuss the internals of the event engines, and a number of other topics have only placeholders.

The manual is organized *not* as a tutorial, but rather closer to a reference manual. In particular, the intent is for the *index* to be highly comprehensive, and to serve as one of the main tools to help you navigate through Bro's numerous features and capabilities. Accordingly, the index contains many "redundant" entries, that is, the same information indexed in multiple ways, to try to make it particularly easy to look up information. For example, you'll find a list of all of the predefined functions under "predefined functions", and also under "functions". There are similar entries for "events" and "variables".

The manual also includes *Note:*'s and *Deficiency:*'s that emphasize points that may be subtle or counter-intuitive, or that reflect bugs of some form. The general delineation between the two is that *Note:*'s discuss facets of Bro not likely to change, while *Deficiency:*'s will (should) eventually get fixed.

I'm very interested in feedback on whether the manual in general and the index in particular is effective, what should be added or removed from it to improve it, any errors found in the index or (of course) elsewhere in the manual, and what topics you would give the highest priority for the next revision of the manual. In addition, *any contributions to the manual* will be highly welcome! You'll find the source for the manual in *doc/manual-src/*.

The current version of the manual is organized as follows. We begin with an overview of how to get started using Bro: building and installing it, running it interactively and on live and prerecorded network traffic, and the helper utilities (scripts and programs) included in the distribution (Chapter 2).

Chapter 3 then discusses the different types, values, and constants that Bro supports. The intent is to provide you with some of the flavor of the language. In addition, later chapters use these concepts to explain things like the types associated with the arguments passed to different event handlers.

Chapter 6 lists the different variables and functions that Bro predefines. The variables generally reflect particular values that control the behavior of the event engine or reflect its status, and the functions are for the most part utilities to aid in the writing of Bro scripts.

Chapter 7 discusses the different analyzers that Bro provides. It is far and away the longest chapter, since there are a good number of analyzers, and some of them are quite rich in their analysis.

Chapter 8 describes how to use Bro's *signature engine*. The signature engine provides a general mechanism for searching for regular expressions in packet payloads or reassembled TCP byte streams. Successful matches can then be fed as events into your policy script for further analysis, including the opportunity to assess the match in terms of surrounding context, which can greatly reduce the problem of "false positives" from which signature-matching can suffer. The chapter also discusses how to incorporate rules from the popular *Snort* intrusion detection system.

Chapter 9 gives an overview of Bro's *interactive debugger*. The debugger allows you to breakpoint your policy script and inspect and change the values of script variables. The chapter also describes the generation of *traces* of all of the events generated during execution.

Finally, Chapter 10 briefly lists different aspects of Bro that have not yet been documented (in addition to the event engine and the Bro language itself).

Acknowledgments:

Major components of Bro's functionality were contributed by Ruoming Pang, Umesh Shankar, Robin Sommer, and Chema Gonzalez. Robin also wrote Chapter 8 of this manual; Umesh wrote Chapter 9; and Michael Kuhn and Benedikt Ostermaier contributed the SSL analyzer (with additional development by Scott Campbell) and the associated documentation.

Many thanks, too, to Craig Leres, Craig Lant, Jim Mellander, Anne Hutton, David Johnston, Mark Handley, and Partha Banerjee for their contributions and operational feedback.

Finally, a number of people were instrumental to supporting Bro's development: Jim Rothfuss, Mark Rosenberg, Stu Loken, Van Jacobson, Dave Stevens, and Jeff Mogul. Again, many thanks!

Chapter 2

Getting Started

This chapter gives an overview of how to get started with operating Bro: *(i)* compiling it, *(ii)* running it interactively, on live network traffic, and on recorded traces, *(iii)* how Bro locates the policy files it should evaluate and how to modify them, *(iv)* the arguments you can give it to control its operation, and *(v)* some helper utilities also distributed with Bro that you'll often find handy.

2.1 Running Bro

This section discusses how to build and install Bro, running it interactively (mostly useful for building up familiarity with the policy language, not for traffic analysis), running it on live and recorded network traffic, modifying Bro policy scripts, and the different run-time flags.

2.1.1 Building and installing Bro

Supported platforms

Bro builds on a number of types of Unix: FreeBSD, Solaris, Linux, SunOS, and Digital Unix, though not all versions. It does *not* build under non-Unix operating systems such as Windows NT.

The Bro source code distribution

You can get the latest public release of Bro from the Bro web page, `http://www.icir.org/vern/bro.html`. Bro is distributed as a *gzip*'d Unix *tar* archive, which you can unpack using:

```
gzcat tar-file | tar xf -
```

or, on some Unix systems:

```
tar zxf tar-file
```

This creates a subdirectory `bro-`*XXX*`-`*version*, where *XXX* is a tag such as `pub` for public releases and `priv` for private releases, and *version* reflects a version and possibly a subversion, such as `0.8a20` for version *0.8* and subversion *a20*.

To build Bro, change to the Bro directory and enter:

```
./configure
make
```

Fix me: Need to discuss configuration options here. `--enable-brov6`

This will compile all of the Bro sources, including a version of the BIND DNS library, version 8, which Bro uses for its non-blocking DNS lookups.

Note: For Linux systems, you may need to use the header files in the linux-include/ *subdirectory included in the Bro distribution to successfully compile Bro.*

Installing Bro

You install Bro by issuing:

```
make install
```

Note: I don't actually use this functionality myself, so it does not tend to be well tested and may have bugs.

Tuning BPF

Bro is written using *libpcap* the portable packet-capture library available from `ftp://ftp.ee.lbl.gov/libpcap.tar.Z`. While *libpcap* knows how to use a wide range of Unix packet filters, it far and away performs most efficiently used in conjunction with the Berkeley Packet Filter (BPF) and with BPF descendants (such as the Digital Unix packet filter). Althought BPF is available from `ftp://ftp.ee.lbl.gov/bpf.tar.Z`, installing it involves modifying your kernel, and perhaps requires significant porting work. However, it comes as part of several operating systems, such as FreeBSD, NetBSD, and OpenBSD.

For BPF systems, you should be aware of the follwoing tuning and configuration issues:

BPF kernel support You need to make sure that kernel support for BPF is enabled. In addition, some systems default to configuring kernel support for only one BPF device. This often proves to be a headache because it means you cannot run more than one Bro at a time, nor can you run it at the same time as `tcpdump`.

/dev/bpf devices Related to the previous item, on BPF systems access to the packet filter is via special */dev/bpf* devices, such as */dev/bpf0*. Just as you need to make sure that the kernel's configuration supports multiple BPF devices, so to must you make sure that an equal number of device files reside in */dev/*.

packet filter permissions On systems for which access to the packet filter is via the file system, you should consider whether you want to only allow root access, or instead create a Unix *group* for which you enable read access to the device file(s). The latter allows you to run Bro as a user other than root, which is *strongly recommended*!

large BPF buffers While running with BPF is often necessary for high performance, it's not necessarily sufficient. By default, BPF programs use very modest kernel buffers for storing packets, which leads to high context switch overhead as the kernel very often has to deliver packets to the user-level Bro process. Minimizing the overhead requires increasing the buffer sizes. This can make a *large* difference!

Under FreeBSD, the configuration variable to increase is `debug.bpf_bufsize`, which you can set via *sysctl*. We recommend creating a script run at boot-up time that increases it from its small default value to something on the order of 100 KB–2 MB, depending on how fast (heavily loaded) is the link being monitored, and how much physical memory the monitor machine has at its disposal.

Important note: some versions of *libpcap*ave internal code that limits the maximum buffer size to 32 KB. For these systems, you should apply the patch included in the Bro distribution in the file `libpcap.patch` .

Finally, once you have increased the buffer sizes, you should *check* that running Bro does indeed consume the amount of kernel memory you expect. You can do this under FreeBSD using *vmstat -m* and searching in the output for the summary of BPF memory. You should find that the *MemUse* statistic goes up by twice the buffer size for every concurrent Bro or `tcpdump` you run.[1] The reason the increase is by twice the buffer size is because Bro uses double-buffering to avoid dropping packets when the buffer fills up.

2.1.2 Using Bro interactively

Once you've built Bro, you can run it interactively to try out simple facets of the policy language. Note that in this mode, Bro is *not* reading network traffic, so you cannot do any traffic analysis; this mode is simply to try out Bro language features.

You run Bro interactively by simply executing "`bro`" without any arguments. It then reads from *stdin* and writes to *stdout*.

Try typing the following to it:

```
print "hello, world";
^D (i.e., end of file)
```

(The end-of-file is critical to remember. It's also a bit annoying for interactive evaluation, but reflects the fact that Bro is not actually meant for interactive use, it simply works as a side-effect of Bro's structure.)

Bro will respond by printing:

```
hello, world
```

to *stdout* and exiting.

You can further declare variables and print expressions, for example:

```
global a = telnet;
print a, a > ftp;
print www.microsoft.com;
```

will print

```
23/tcp, T
207.46.230.229, 207.46.230.219, 207.46.230.218
```

where `23/tcp` reflects the fact that `telnet` is a predefined variable whose value is TCP port 23, which is larger than TCP port 21 (i.e., `ftp`); and the DNS name *www.microsoft.com* currently returns the above three addresses.

You can also define functions:

```
function top18bits(a: addr): addr
    {
    return mask_addr(a, 18);
    }

print top18bits(128.3.211.7);
```

[1] Providing that these programs have been recompiled with the corrected *libpcap* noted above.

prints

```
128.3.192.0
```

and even event handlers:

```
event bro_done()
    {
    print "all done!";
    }
```

which prints "`all done!`" when Bro exits.

2.1.3 Specifying policy scripts

Usually, rather than running Bro interactively you want it to execute a policy script or a set of policy scripts. You do so by specifying the names of the scripts as command-line arguments, such as:

```
bro ~/my-policy.bro ~/my-additional-policy.bro
```

Bro provides several mechanisms for simplifying how you specify which policies to run.

First, if a policy file doesn't exist then it will try again using `.bro` as a suffix, so the above could be specified as:

```
bro ~/my-policy ~/my-additional-policy
```

Second, Bro consults the colon-separated search path `$BROPATH` to locate policy scripts. If your home directory was listed in `$BROPATH`, then you could have invoked it above using:

```
bro my-policy my-additional-policy
```

Note: If you define `$BROPATH`*, you* must *include* bro-dir`/policy`*, where* bro-dir *is where you have built or installed Bro, because it has to be able to locate* bro-dir`/policy/bro.init` *to initialize itself at run-time.*

Third, the `@load` directive can be used in a policy script to indicate the Bro should at that point process another policy script (like C's `include` directive; see § 7.1.1, page 92). So you could have in *my-policy*:

```
@load my-additional-policy
```

and then just invoke Bro using:

```
bro my-policy
```

providing you *always* want to load *my-additional-policy* whenever you load *my-policy*.

Note that the predefined Bro module `mt` loads almost all of the other standard Bro analyzers, so you can pull them in with simply:

```
@load mt
```

or by invoking Bro using "`bro mt` *`my-policy`*".

2.1.4 Running Bro on network traffic

There are two ways to run Bro on network traffic: on traffic captured live by the network interface(s), and on traffic previously recorded using the `-w` flag of `tcpdump` or Bro itself.

Live traffic

Bro reads live traffic from the local network interface whenever you specify the `-i` flag. As mentioned below, you can specify multiple instances to read from multiple interfaces simultaneously, however the interfaces must all be of the same link type (e.g., you can't mix reading from a Fast Ethernet with reading from an FDDI link, though you can mix a 10 Mbps Ethernet interface with a 100 Mbps Ethernet).

In addition, Bro will read live traffic from the interface(s) listed in the `interfaces` variable, *unless* you specify the `-r` flag (and do not specify `-i`). So, for example, if your policy script contains:

```
const interfaces += "sk0";
const interfaces += "sk1";
```

then Bro will read from the *sk0* and *sk1* interfaces, and you don't need to specify `-i`.

Traffic traces

To run on recorded traffic, you use the `-r` flag to indicate the trace file Bro should read. As with `-i`, you can use the flag multiple times to read from multiple files; Bro will merge the packets from the files into a single packet stream based on their timestamps.

The Bro distribution includes an example trace that you can try out, *example.ftp-attack.trace*. If you invoke Bro using:

```
setenv BRO_ID example
bro -r example.ftp-attack.trace mt
```

you'll see that it generates a connection summary to *stdout*, a summary of the FTP sessions to `ftp.example`, a copy of what would have been real-time alerts had Bro been running on live traffic to `log.example`, and a summary of unusual traffic anomalies (none in this trace) to `weird.example`.

2.1.5 Modifying Bro policy

One way to alter the policy Bro executes is of course to directly edit the scripts. When this can be avoided, however, that is preferred, and Bro provides a pair of related mechanisms to help you specify *refinements* to existing policies in separate files.

The first such mechanism is that you can define *multiple* handlers for a given event. So, for example, even though the standard `ftp` analyzer (*bro-dir*`/policy/ftp.bro`) defines a handler for `ftp_request`, you can define another handler if you wish in your own policy script, even if that policy script loads (perhaps indirectly, via the `mt` module) the `ftp` analyzer. When the event engine generates an `ftp_request` event, *both* handlers will be invoked. *Deficiency: Presently, you do not have control over the order in which they are invoked; you also cannot completely override one handler with another, preventing the first from being invoked.*

Second, the standard policy scripts are often written in terms of *redefinable* variables. For example, `ftp.bro` contains a variable `ftp_guest_ids` that defines a list of usernames the analyzer will consider to reflect guest accounts:

```
const ftp_guest_ids = { "anonymous", "ftp", "guest", } &redef;
```

While "`const`" marks this variables as constant at run-time, the attribute "`&redef`" specifies that its value can be redefined.

For example, in your own script you could have:

```
    redef ftp_guest_ids = { "anonymous", "guest", "visitor", "student" };
```

instead. (Note the use of "redef" rather than "const", to indicate that you realize you are redefining an existing variable.)

In addition, for most types of variables you can specify *incremental* changes to the variable, either new elements to add or old ones to subtract. For example, you could instead express the above as:

```
    redef ftp_guest_ids += { "visitor", "student" };
    redef ftp_guest_ids -= "ftp";
```

The potential advantage of incremental refinements such as these are that if any *other* changes are made to ftp.bro's original definition, your script will automatically inherit them, rather than replacing them if you used the first definition above (which explicitly lists all four names to include in the variable). Sometimes, however, you don't want this form of inheriting later changes; you need to decide on a case-by-case basis, though our experience is that generally the incremental approach works best.

Finally, the use of *prefixes* provides a way to specify a whole set of policy scripts to load in a particular context. For example, if you set $BRO_PREFIXES to "dialup", then a load of ftp.bro will *also* load dialup.ftp.bro automatically (if it exists). See § 10.1, page 176 for further discussion.

2.1.6 Bro flags and run-time environment

Flags

When invoking Bro, you can control its behavior using the following flags:

-f *filter*
: Use *filter* as the tcpdump filter for capturing packets, rather than the combination of capture_filter and restrict_filter, or the default of "tcp or udp" (§ 7.1.2, page 93).

-h
: Generate a help message summarizing Bro's options and environment variables, and exit.

-i *interface*
: Add *interface* to the list of interfaces from which Bro should read network traffic (§ 2.1.4, page 16). You can use this flag multiple times to direct Bro to read from multiple interfaces. You can also, or in addition, use refinements of the interfaces variable to specify interfaces.

Note that if no interfaces are specified, then Bro will not read any network traffic. It does *not* have a notion of a "default" interface from which to read.

-p *prefix*
: Add *prefix* to the list of prefixes searched by Bro when @load'ing a script. You can also, or in addition, use @prefix to specify search prefixes. See § 10.1, page 176 for discussion.

-r *readfile*
: Add *readfile* to the list of tcpdump save files that Bro should read. You can use this flag multiple times to direct Bro to read from multiple save files; it will merge the packets read from the different files based on their timestamps. Note that if the save files contain only packet headers and not contents, then of course Bro's analysis of them will be limited.

Note that use of -r is *mutually exclusive* with use of -i. However, you can use -r when running scripts that refine interfaces, in which case the -r option takes precedence and Bro performs off-line analysis.

`-s` *signaturefile*

Add *signaturefile* to the list of files containing signatures to match against the network traffic. See `signatures` for more information about signatures.

`-w` *writefile*

Write a `tcpdump` save file to the file *writefile*. Bro will record all of the packets it captures, including their contents, except as controlled by calls to `set_record_packets`.

Note: One exception is that unless you are analyzing HTTP events (for example, by `@load`*'ing the* `http` *analyzer), Bro does* not *record the* contents *of HTTP SYN/FIN/RST packets to the trace file. The reason for this is that HTTP FIN packets often contain a large amount of data, which is not of any interest if you are not using HTTP analysis, and due to the very high volume of HTTP traffic at many sites, removing this data can significantly reduce the size of the save file. Deficiency: Clearly, this should not be hardwired into Bro but under user control.*

Save files written using `-w` are of course readable using `-r`. Accordingly, you will generally want to use `-w` when running Bro on live network traffic so you can rerun it off-line later to understand any problems that arise, and also to experiment with the effects of changes to the policy scripts.

You can also combine `-r` with `-w` to both read a save file(s) and write another. This is of interest when using multiple instances of `-r`, as it provides a way to merge `tcpdump` save files.

`-v`

Print the version of Bro and exit.

`-F`

Instructs Bro that it *must* resolve all hostnames out of its private DNS cache (§ 7.12, page 118). If the script refers to a hostname not in the cache, then Bro *exits* with a fatal error.

The point behind this option is to ensure that Bro starts quickly, rather than possibly stalling for an indeterminant amount of time resolving a hostname. Fast startup simplifies checkpointing a running Bro—you can start up a new Bro and then killing off the old one shortly after. You'd like this to occur in a manner such that there's no period during which neither Bro is watching the network (the older because you killed it off too early, the newer because it's stuck resolving hostnames).

`-O`

Turns on Bro's optimizer for improving its internal representation of the policy script. *Note: Currently, the amount of improvement is modest, and there's (as always) a risk of an optimizer bug introducing errors into the execution of the script, so the optimizer is not enabled by default.*

`-P`

Instructs Bro to *prime* its private DNS cache (§ 7.12, page 118). It does so by parsing the policy scripts, but not executing them. Bro looks up each hostname's address(es) and records them in the private cache. The idea is that once `bro -P` finishes, you can then use `bro -F` to start up Bro quickly because it will read all the information it needs from the cache.

`-W`

Instructs Bro to activate its internal *watchdog*. The watchdog provides self-monitoring to enable Bro to detect if its processing is wedged.

Bro only activates the watchdog if it is reading live network traffic. The watchdog consists of a periodic timer that fires every `WATCHDOG_INTERVAL` seconds. (*Deficiency: clearly this should be a user-definable value.*)

At that point, the watchdog checks to see whether Bro is still working on the same packet as it was the last time the watchdog expired. If so, then the watchdog logs this fact along with some information regarding when Bro began processing the current packet and how many events it processed after handling the packet. Finally, it prints the packet drop information for the different interfaces Bro was reading from, and aborts execution.

Run-time environment

Bro is also sensitive to the following environment variables:

`$BROPATH`

A colon-separated list of directories that Bro searches whenever you `@load` a policy file. It loads the first instance it finds (though see `$BRO_PREFIXES` for how a single `@load` can lead to Bro loading multiple files).

Default: if you don't set this variable, then Bro uses the path

```
.:policy:policy/local:/usr/local/lib/bro
```

That is, the current directory, any *policy/* and *policy/local/* subdirectories, and */usr/local/lib/bro/*.

`$BRO_PREFIXES`

A colon-separate lists of *prefixes* that Bro should apply to each name in a `@load` directive. For a given prefix and load-name, Bro constructs the filename:

prefix`.`*load-name*`.bro`

(where it doesn't add `.bro` if *load-name* already ends in `.bro`). It then searches for the filename using `$BROPATH` and loads it if its found. Furthermore, it *repeats* this process for all of the other prefixes (left-to-right), and loads *each* file it finds for the different prefixes. *Note: Bro* also *first attempts to load the filename without any prefix at all. If this load fails, then Bro exits with an error complaining that it can't open the given* `@load` *file.*

For example, if you set `$BRO_PREFIXES` to:

```
mysite:mysite.wan
```

and then issue "`@load ftp`", Bro will attempt to load each of the following scripts in the following order:

```
ftp.bro
mysite.ftp.bro
mysite.wan.ftp.bro
```

Default: if you don't specify a value for `$BRO_PREFIXES`, it defaults to empty, and for the example above Bro would only attempt to `@load ftp.bro`.

2.2 Helper utilities

2.2.1 Scripts

Documentation missing.

2.2.2 The *hf* utility

The *hf* utility reads text on *stdin* and attempts to convert any “dotted quads” it sees to hostnames. It is very convenient for running on Bro log files to generate human-readable forms. See the manual page included with the distribution for details.

2.2.3 The *cf* utility

The *cf* utility reads Unix timestamps at the beginning of lines on *stdin* and converts them to human-readable form. For example, for the input line:

```
972499885.784104 #26 131.243.70.68/1899 > 64.55.26.206/ftp start
```

it will generate:

```
Oct 25 11:51:25 #26 131.243.70.68/1899 > 64.55.26.206/ftp start
```

It takes two flags:

`-l`

specifies the *long* human-readable form, which includes the year. For example, on the above input, the output would instead be:

```
Oct 25 11:51:25 2000 #26 131.243.70.68/1899 > 64.55.26.206/ftp start
```

`-s`

specifies *strict* checking to ensure that the number at the beginning of a line is a plausible timestamp: it must have at least 9 digits, at most one decimal, and must have a decimal if there are 10 or more digits.

Without `-s`, an input like:

```
131.243.70.68 > 64.55.26.206
```

generates the output:

```
Dec 31 16:02:11 > 64.55.26.206
```

which, needless to say, is not very helpful. *Deficiency: It seems clear that* `-s` *should be the default behavior.*

Chapter 3

Values, Types, and Constants

3.1 Overview

We begin with an overview of the types of values supported by Bro, giving a brief description of each type and introducing the notions of type conversion and type inference. We discuss each type in detail in § 3.2–§ 3.17 below.

3.1.1 Bro Types

There are 18 types of values in the Bro type system:

- `bool` for Booleans;
- `count`, `int`, and `double` types, collectively called *numeric*, for arithmetic and logical operations, and comparisons;
- `enum` for enumerated types similar to those in C;
- `string`, character strings that can be used for comparisons and to index tables and sets;
- `pattern`, regular expressions that can be used for pattern matching;
- `time` and `interval`, for absolute and relative times, collectively termed *temporal*;
- `port`, a TCP or UDP port number;
- `addr`, an IP address;
- `net`, a network prefix;
- `record`, a collection of values (of possibly different types), each of which has a name;
- `table`, an associative array, indexed by tuples of scalars and yielding values of a particular type;
- `set`, a collection of tuples-of-scalars, for which a particular tuple's membership can be tested;
- `file`, a disk file to write or append to;

- function, a function that when called with a list of values (arguments) returns a value;
- event, an event handler that is invoked with a list of values (arguments) any time an event occurs.

Every value in a Bro script has one of these types. For most types there are ways of specifying *constants* representing values of the type. For example, 2.71828 is a constant of type double, and 80/tcp is a constant of type port. The discussion of types in § 3.2–§ 3.17 below includes a description of how to specify constants for the types.

Finally, even though Bro variables have *static* types, meaning that their type is fixed, often their type is *inferred* from the value to which they are initially assigned when the variable is declared. For example,

```
local a = "hi there";
```

fixes a's type as string, and

```
local b = 6;
```

sets b's type to count. See § 5.1.3, page 55 for further discussion.

3.1.2 Type Conversions

Some types will be automatically converted to other types as needed. For example, a count value can always be used where a double value is expected. The following:

```
local a = 5;
local b = a * .2;
```

creates a local variable a of type count and assigns the double value 1.0 to b, which will also be of type double. Automatic conversions are limited to converting between *numeric* types. The rules for how types are converted are given below.

3.2 Booleans

The bool type reflects a value with one of two possible meanings: *true* or *false*.

3.2.1 Boolean Constants

There are two bool constants: T and F. They represent the values of "true" and "false", respectively.

3.2.2 Logical Operators

Bro supports three logical operators: &&, ||, and ! are Boolean "and," "or," and "not," respectively. && and || are "short circuit" operators, as in C: they evaluate their right-hand operand only if needed.

The && operator returns F if its first operand evaluates to *false*, otherwise it evaluates its second operand and returns T if it evaluates to *true*. The || operator evaluates its first operand and returns T if the operand evaluates to *true*. Otherwise it evaluates its second operand, and returns T if it is *true*, F if *false*.

The unary ! operator returns the boolean negation of its argument. So, ! T yields F, and ! F yields T.

The logical operators are left-associative. The ! operator has very high precedence, the same as unary + and -; see § 3.3.3 and § 10.11. The || operator has precedence just below &&, which in turn is just below that of the comparison operators (see § 3.3.4, page 24).

3.3 Numeric Types

`int`, `count`, and `double` types should be familiar to most programmers as integer, unsigned integer, and double-precision floating-point types.

These types are referred to collectively as *numeric*. *Numeric* types can be used in arithmetic operations (see § 3.3.3 below) as well as in comparisons (§ 3.3.4, page 24).

3.3.1 Numeric Constants

`count` constants are just strings of digits: `1234` and `0` are examples.

`integer` constants are strings of digits preceded by a `+` or `-` sign: `-42` and `+5` for example. Because digit strings without a sign are of type `count`, occasionally you need to take care when defining a variable if it really needs to be of type `int` rather than `count`. Because of type inferencing (§ 5.1.3, page 55), a definition like:

```
local size_difference = 0;
```

will result in `size_difference` having type `count` when `int` is what's instead needed (because, say, the size difference can be negative). This can be resolved either by using an `int` constant in the initialization:

```
local size_difference = +0;
```

or explicitly indicating the type:

```
local size_difference: int = 0;
```

You write floating-point constants in the usual ways, a string of digits with perhaps a decimal point and perhaps a scale-factor written in scientific notation. Optional `+` or `-` signs may be given before the digits or before the scientific notation exponent. Examples are `-1234.`, `-1234e0`, `3.14159`, and `.003e-23`. All floating-point constants are of type `double`.

3.3.2 Mixing Numeric Types

You can freely intermix *numeric* types in expressions. When intermixed, values are promoted to the "highest" type in the expression. In general, this promotion follows a simple hierarchy: `double` is highest, `int` comes next, and `count` is lowest. (Note that `bool` is not a numeric type.)

3.3.3 Arithmetic Operators

For doing arithmetic, Bro supports `+`, `-`, `*`, `/`, and `%`. In general, binary operators evaluate their operands after converting them to the higher type of the two and return a result of that type. However, subtraction of two `count` values yields an `int` value. Division is integral if its operands are `count` and/or `int`.

`+` and `-` can also be used as unary operators. If applied to a `count` type, they yield an `int` type.

`%` computes a *modulus*, defined in the same way as in the C language. It can only be applied to `count` or `int` types, and yields `count` if both operands are `count` types, otherwise `int`.

Binary `+` and `-` have the lowest precedence, `*`, `/`, and `%` have equal and next highest precedence. The unary `+` and `-` operators have the same precedence as the `!` operator (§ 3.2.2, page 22). See § 10.11, page 177, for a table of the precedence of all Bro operators.

All arithmetic operators associate from left-to-right.

3.3.4 Comparison Operators

Bro provides the usual comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. They each take two operands, which they convert to the higher of the two types (see § 3.3.2, page 23). They return a `bool` corresponding to the comparison of the operands. For example,

```
3 < 3.000001
```

yields true.

The comparison operators are all non-associative and have equal precedence, just below that of the `in` operator and just above that of the logical && operator. See § 10.11, page 177, for a general discussion of precedence.

3.4 Enumerations

Enumerations allow you to specify a set of related values that have no further structure, similar to `enum` types in C. For example:

```
type color: enum { Red, White, Blue, };
```

defines the values `Red`, `White`, and `Blue`. A variable of type `color` holds one of these values. Note that `Red` et al have *global scope*. You *cannot* define a variable or type with those names. (Also note that, as usual, the comma after `Blue` is optional.)

The only operations allowed on enumerations are comparisons for equality. Unlike C enuemrations, they do not have values or an ordering associated with them.

You can extend the set of values in an enumeration using `redef enum` *`identifier`* `+=` { *`name-list`* }:

```
redef enum color += { Black, Yellow };
```

3.5 Strings

The `string` type holds character-string values, used to represent and manipulate text.

3.5.1 String Constants

You create string constants by enclosing text within double (`"`) quotes. A backslash character (`\`) introduces an *escape sequence*. The following ANSI C escape sequences are recognized: `\a` yields an alert (bell) character, `\b` yields a backspace character, `\f` yields a formfeed character, `\n` yields a newline character, `\r` yields a carriage return character, `\t` a tab character, \octal-digits the 8-bit ASCII character with code *octal-digits*, and `\x`*hex-digits* the 8-bit ASCII character with code *hex-digits*. Bro string constants currently *cannot* be continued across multiple lines by escaping newlines in the input. This may change in the future. Any other character following a `\` is passed along literally.

Unlike with C, strings are represented internally as a count and a vector of bytes, rather than a NUL-terminated series of bytes. This difference is important because NULs can easily be introduced into strings derived from network traffic, either by the nature of the application, inadvertently, or maliciously by an attacker attempting to subvert the monitor. An example of the latter is sending the following to an FTP server:

```
USER nice\0USER root
```

where "\0" represents a NUL. Depending on how it is written, the FTP application receiving this text might well interpret it as two separate commands, "USER nice" followed by "USER root". But if the monitoring program uses NUL-terminated strings, then it will effectively see only "USER nice" and have no opportunity to detect the subversive action.

Note that Bro string constants are automatically NUL-terminated.

Note: While Bro itself allows NULs in strings, their presence in arguments to many Bro functions results in a run-time error, as often their presence (or, conversely, lack of a NUL terminator) indicates some sort of problem (particularly for arguments that will be passed to C functions). See § 6.2.1, page 91 for discussion.

3.5.2 String Operators

Currently the only string operators provided are the comparison operators discussed in § 3.3.4, page 24 and pattern-matching as discussed in § 3.6.2, page 26. These operators perform character by character comparisons based on the native character set, usually ASCII.

Some functions for manipulating strings are also available. See § 6.2.3, page 91.

3.6 Patterns

The `pattern` type holds regular-expression patterns, which can be used for fast text searching operations.

3.6.1 Pattern Constants

You create pattern constants by enclosing text within forward slashes (/). The syntax is the same as for the *flex* version of the *lex* utility. For example,

```
/foo|bar/
```

specifies a pattern that matches either the text "foo" or the text "bar";

```
/[a-zA-Z0-9]+/
```

matches one or more letters or digits, as will

```
/[[:alpha:][:digit:]]+/
```

or

```
/[[:alnum:]]+/
```

and the pattern

```
/^rewt.*login/
```

matches any string with the text "rewt" at the beginning of a line followed somewhere later in the line by the text "login".

You can create disjunctions (patterns the match any of a number of alternatives) both using the "|" regular expression operator directly, as in the first example above, or by using it to join multiple patterns. So the first example above could instead be written:

```
/foo/ | /bar/
```

This form is convenient when constructing large disjunctions because it's easier to see what's going on.

Note that the speed of the regular expression matching does *not* depend on the complexity or size of the patterns, so you should feel free to make full use of the expressive power they afford.

You can assign `pattern` values to variables, hold them in tables, and so on. So for example you could have:

```
global address_filters: table[addr] of pattern = {
    [128.3.4.4] = /failed login/ | /access denied/,
    [128.3.5.1] = /access timeout/
};
```

and then could test, for example:

```
if ( address_filters[c$id$orig_h] in msg )
    skip_the_activity();
```

Note though that you cannot use create patterns dynamically. this form (or any other) to create dynamic

3.6.2 Pattern Operators

There are two types of pattern-matching operators: *exact* matching and *embedded* matching.

Exact Pattern Matching

Exact matching tests for a string entirely matching a given pattern. You specify exact matching by using the `==` equality relational with one `pattern` operand and one `string` operand (order irrelevant). For example,

```
"foo" == /foo|bar/
```

yields true, while

```
/foo|bar/ == "foobar"
```

yields false. The `!=` operator is the negation of the `==` operator, just as when comparing strings or numerics.

Note that for exact matching, the `^` (anchor to beginning-of-line) and `$` (anchor to end-of-line) regular expression operators are redundant: since the match is *exact*, every pattern is implicitly anchored to the beginning and end of the line.

Embedded Pattern Matching

Embedded matching tests whether a given pattern appears anywhere within a given string. You specify embedded pattern matching using the `in` operator. It takes two operands, the first (which must appear on the left-hand side) of type `pattern`, the second of type `string`. For example,

```
/foo|bar/ in "foobar"
```

yields true, as does

```
/oob/ in "foobar"
```

but

```
/ˆoob/ in "foobar"
```

does not, since the text “oob” does not appear the beginning of the string “foobar”. Note, though, that the `$` regular expression operator (anchor to end-of-line) is not currently supported, so:

```
/oob$/ in "foobar"
```

currently yields true. This is likely to change in the future.

Finally, the `!in` operator yields the negation of the `in` operator.

3.7 Temporal Types

Bro supports types representing *absolute* and *relative* times with the `time` and `interval` types, respectively.

3.7.1 Temporal Constants

There is currently no way to specify an absolute time as a constant (though see the `current_time` and `network_time` functions in § 6.2.4, page 91). You can specify `interval` constants, however, by appending a *time unit* after a numeric constant. For example,

```
3.5 min
```

denotes 210 seconds. The different time units are `usec`, `sec`, `min`, `hr`, and `day`, representing microseconds, seconds, minutes, hours, and days, respectively. The whitespace between the numeric constant and the unit is optional, and the letter “s” may be added to pluralize the unit (this has no semantic effect). So the above example could also be written:

```
3.5mins
```

or

```
150 secs
```

3.7.2 Temporal Operators

You can apply arithmetic and relational operators to temporal values, as follows.

Temporal Negation

The unary `-` operator can be applied to an `interval` value to yield another `interval` value. For example,

```
- 12 hr
```

represents “twelve hours in the past.”

Temporal Addition

Adding two interval values yields another interval value. For example,

```
    5 sec + 2 min
```

yields 125 seconds. Adding a time value to an interval yields another time value.

Temporal Subtraction

Subtracting a time value from another time value yields an interval value, as does subtracting an interval value from another interval, while subtracting an interval from a time yields a time.

Temporal Multiplication

You can multiply an interval value by a *numeric* value to yield another interval value. For example,

```
   5 min * 6.5
```

yields 1,950 seconds. time values cannot be scaled by multiplication or division.

Temporal Division

You can also divide an interval value by a *numeric* value to yield another interval value. For example,

```
   5 min / 2
```

yields 150 seconds. Furthermore, you can divide one interval value by another to yield a double. For example,

```
   5 min / 30 sec
```

yields 10.

Temporal Relationals

You may compare two time values or two interval values for equality, and also for ordering, where times or intervals further in the future are considered larger than times or intervals nearer in the future, or in the past.

3.8 Port Type

The port type corresponds to a TCP or UDP port number. TCP and UDP ports are distinct. Thus, a value of type port can hold either a TCP or a UDP port, but at any given time it is holding exactly one of these.

3.8.1 Port Constants

There are two forms of port constants. The first consists of an unsigned integer followed by either "/tcp" or "/udp." So, for example, "80/tcp" corresponds to TCP port 80 (the HTTP protocol used by the World Wide Web). The second form of constant is specified using a predefined identifier, such as "http", equivalent to "80/tcp." These predefined identifiers are simply const variables defined in the Bro initialization file (§ 10.18, page 178), such as:

```
    const http = 80/tcp;
```

3.8.2 Port Operators

The only operations that can be applied to port values are relationals. You may compare them for equality, and also for ordering. For example,

```
    20/tcp < telnet
```

yields true because telnet is a predefined constant set to 23/tcp.

UDP ports are considered larger than TCP ports, i.e., "0/udp" is larger than "65535/tcp".

3.9 Address Type

Another networking type provided by Bro is addr, corresponding to an IP address. The only operations that can be performed on them are comparisons for equality or inequality (also, a built-in function provides masking, as discussed below).

When configuring the Bro distribution, if you specify --enable-brov6 then Bro will be built to support both IPv4 and IPv6 addresses, and an addr can hold either. Otherwise, addresses are restricted to IPv4.

3.9.1 Address Constants

Constants of type addr have the familiar "dotted quad" format, $A\_1.A\_2.A\_3.A\_4$, where the A_i all lie between 0 and 255. If you have configured for IPv6 support as discussed above, then you can also use the colon-separated hexadecimal form described in [RFC2373].

Often more useful are *hostname* constants. There is no Bro type corresponding to Internet hostnames. Because hostnames can correspond to multiple IP addresses, you quickly run into ambiguities if comparing one hostname with another. Bro does, however, support hostnames as constants. Any series of two or more identifiers delimited by dots forms a hostname constant, so, for example, "lbl.gov" and "www.microsoft.com" are both hostname constants (the latter, as of this writing, corresponds to 5 distinct IP addresses). The value of a hostname constant is a list of addr containing one or more elements. These lists (as with the lists associated with certain port constants, discussed above) cannot be used in Bro expressions; but they play a central role in initializing Bro table's and set's.

3.9.2 Address Operators

The only operations that can be applied to addr values are comparisons for equality or inequality, using == and !=. However, you can also operate on addr values using mask_addr to mask off lower address bits, and to_net to convert an addr to a net (see below).

3.10 Net Type

Related to the addr type is net. net values hold address prefixes. Historically, the IP address space was divided into different *classes* of addresses, based on the uppermost components of a given address: class A spanned the range 0.0.0.0 to 127.255.255.255; class B from 128.0.0.0 to 191.255.255.255; class C from 192.0.0.0 to 223.255.255.255; class D from 224.0.0.0 to 239.255.255.255; and class E from 240.0.0.0 to 255.255.255.255. Addresses were allocated to different networks out of either class A, B, or C, in blocks of 2^{24}, 2^{16}, and 2^{8} addresses, respectively.

Accordingly, net values hold either an 8-bit class A prefix, a 16-bit class B prefix, a 24-bit class C prefix, or a 32-bit class D "prefix" (an entire address). Values for class E prefixes are not defined (because no such addresses are currently allocated, and so shouldn't appear in other than clearly-bogus packets).

Today, address allocations come not from class A, B or C, but instead from *CIDR* blocks (CIDR = Classless Inter-Domain Routing), which are prefixes between 1 and 32 bits long in the range 0.0.0.0 to 223.255.255.255. *Deficiency: Bro* should *deal just with CIDR prefixes, rather than old-style network prefixes. However, these are more difficult to implement efficiently for table searching and the like; hence currently Bro only supports the easier-to-implement old-style prefixes. Since these don't match current allocation policies, often they don't really fit an address range you'll want to describe. But for sites with older allocations, they do, which gives them some basic utility.*

In addition, *Deficiency: IPv6 has no notion of old-style network prefixes, only CIDR prefixes, so the lack of support of CIDR prefixes impairs use of Bro to analyze IPv6 traffic.*

3.10.1 Net Constants

You express constants of type net in one of two forms, either:

$N\_1.N\_2.$

or

$N\_1.N\_2.N\_3$

where the N_i all lie between 0 and 255. The first of these corresponds to class B prefixes (note the trailing "." that's required to distinguish the constant from a floating-point number), and the second to class C prefixes. *Deficiency: There's currently no way to specify a class A prefix.*

3.10.2 Net Operators

The only operations that can be applied to net values are comparisons for equality or inequality, using == and !=.

3.11 Records

A record is a collection of values. Each value has a name, referred to as one of the record's *fields*, and a type. The values do not need to have the same type, and there is no restriction on the allowed types (i.e., each field can be *any* type).

3.11.1 Defining records

A definition of a record type has the following syntax:

record { *field*$^+$ }

(that is, the keyword record followed by one-or-more *field*'s enclosed in braces), where a *field* has the syntax:

identifier : *type field-attributes** ; *identifier* : *type field-attributes** , Each field has a name given by the identifier (which can be the same as the identifier of an existing variable or a field in another record). Field names must follow the same syntax as that for Bro variable names (see § 5, page 53), namely they must begin with a letter or an underscore ("_") followed by zero or more letters, underscores, or digits. Bro reserved words such as `if` or `event` cannot be used for field names. Field names are case-sensitive.

Each field holds a value of the given type. We discuss the optional *field-attributes* below. Finally, you can use either a semicolon or a comma to terminate the definition of a record field.

For example, the following record type:

```
type conn_id: record {
    orig_h: addr;  # Address of originating host.
    orig_p: port;  # Port used by originator.
    resp_h: addr;  # Address of responding host.
    resp_p: port;  # Port used by responder.
};
```

is used throughout Bro scripts to denote a connection identifier by specifying the connections originating and responding addresses and ports. It has four fields: `orig_h` and `resp_h` of type `addr`, and `orig_p` of `resp_p` of type `port`.

3.11.2 Record Constants

You can initialize values of type `record` using either assignment from another, already existing `record` value; or element-by-element; or using a *record constructor*.

In a Bro function or event handler, we could declare a local variable the `conn_id` type given above:

```
local id: conn_id;
```

and then explicitly assign each of its fields:

```
id$orig_h = 207.46.138.11;
id$orig_p = 31337/tcp;
id$resp_h = 207.110.0.15;
id$resp_p = 22/tcp;
```

Deficiency: One danger with this initialization method is that if you forget to initialize a field, and then later access it, you will crash *Bro.*

Or we could use:

```
id = [$orig_h = 207.46.138.11, $orig_p = 31337/tcp,
      $resp_h = 207.110.0.15, $resp_p = 22/tcp];
```

This second form is no different from assigning a `record` value computed in some other fashion, such as the value of another variable, a table element, or the value returned by a function call. Such assignments must specify *all* of the fields in the target (i.e., in `id` in this example), unless the missing field has the `&optional` or `&default` attribute.

3.11.3 Accessing Fields Using “$”

You access and assign record fields using the “$” (dollar-sign) operator. As indicated in the example above, for the record id we can access its orig_h field using:

```
id$orig_h
```

which will yield the addr value 207.46.138.11.

3.11.4 Record Assignment

You can assign one record value to another using simple assignment:

```
local a: conn_id;
...
local b: conn_id;
...
b = a;
```

Doing so produces a *shallow* copy. That is, after the assignment, b refers to the same record as does a, and an assignment to one of b’s fields will alter the field in a’s value (and vice versa for an assignment to one of a’s fields). However, assigning again to b itself, or assigning to a itself, will break the connection.

Deficiency: Bro lacks a mechanism for specifying a deep *copy, in which no linkage is connected between* b *and* a. Consequently, you must be careful when assigning records to ensure you account for the shallow-copy semantics.

You can also assign to a record another record that has fields with the same names and types, even if they come in a different order. For example, if you have:

```
local b: conn_id;
local c: record {
    resp_h: addr, orig_h: addr;
    resp_p: port, orig_p: port;
};
```

then you can assign either b to c or vice versa.

You could *not*, however, make the assignment (in either direction) if you had:

```
local b: conn_id;
local c: record {
    resp_h: addr, orig_h: addr;
    resp_p: port, orig_p: port;
    num_alerts: count;
};
```

because the field num_alerts would either be missing or excess.

However, when declaring a record you can associate attributes with the fields. The relevant ones are &optional, which indicates that when assigning to the record you can omit the field, and &default = *expr*, which indicates that if the field is missing, then a reference to it returns the value of the expression *expr*. So if instead you had:

```
local b: conn_id;
local c: record {
    resp_h: addr, orig_h: addr;
    resp_p: port, orig_p: port;
    num_alerts: count &optional;
};
```

then you could execute `c = b` even though `num_alerts` is missing from `b`. You still could not execute `b = c`, though, since in that direction, `num_alerts` is an extra field (regardless of whether it has been assigned to or not—the error is a type-checking error, not a run-time error).

The same holds for:

```
local b: conn_id;
local c: record {
    resp_h: addr, orig_h: addr;
    resp_p: port, orig_p: port;
    num_alerts: count &default = 0;
};
```

I.e., you could execute `c = b` but not `b = c`. The only difference between this example and the previous one is that for the previous one, access to `c$num_alerts` without having first assigned to it results in a run-time error, while in the second, it yields `0`.

You can test for whether a record field exists using the `?$` operator.

Finally, all of the rules for assigning records also apply when passing a record value as an argument in a function call or an event handler invocation.

3.12 Tables

`table`'s provide *associative arrays*: mappings from one set of values to another. The values being mapped are termed the *index* (or *indices*, if they come in groups of more than one) and the results of the mapping the *yield*.

Tables are quite powerful, and indexing them is very efficient, boiling down to a single hash table lookup. So you should take advantage of them whenever appropriate.

3.12.1 Declaring Tables

You declare tables using the following syntax:

`table [` *type*$^+$ `] of` *type*

where *type*$^+$ is one or more types, separated by commas.

The indices can be of the following *scalar* types: *numeric*, *temporal*, *enumerations*, `string`, `port`, `addr`, or `net`. The yield can be of any type. So, for example:

```
global a: table[count] of string;
```

declares `a` to be a table indexed by a `count` value and yielding a `string` value, similar to a regular array in a language like C. The yield type can also be more complex:

```
global a: table[count] of table[addr, port] of conn_id;
```

declares a to be a table indexed by count and yielding another table, which itself is indexed by an addr and a port to yield a conn_id record.

This second example illustrates a *multi-dimensional* table, one indexed not by a single value but by a *tuple* of values.

3.12.2 Initializing Tables

You initialize tables by enclosing a set of initializers within braces. Each initializer looks like:

[*expr-list*] = *expr*

where *expr-list* is a comma-separated list of expressions corresponding to an index of the table (so, for a table indexed by count, for example, this would be a single expression of type count) and *expr* is the yield value to assign to that index.

For example,

```
global a: table[count] of string = {
    [11] = "eleven",
    [5] = "five",
};
```

initializes the table a to have two elements, one indexed by 11 and yielding the string "eleven" and the other indexed by 5 and yielding the string "five". (Note the comma after the last list element; it is optional, similar to how C allows final commas in declarations.)

You can also group together a set of indices together to initialize them to the same value:

```
type HostType: enum { DeskTop, Server, Router };
global a: table[addr] of HostType = {
    [[155.26.27.2, 155.26.27.8, 155.26.27.44]] = Server,
};
```

is equivalent to:

```
type HostType: enum { DeskTop, Server, Router };
global a: table[addr] of HostType = {
    [155.26.27.2] = Server,
    [155.26.27.8] = Server,
    [155.26.27.44] = Server,
};
```

This mechanism also applies to *hostnames*, which can be used in table initializations for any indices of type addr. For example, if www.my-server.com corresponded to the addresses 155.26.27.2 and 155.26.27.44, then the above could be written:

```
global a: table[addr] of HostType = {
    [[www.my-server.com, 155.26.27.8]] = Server,
};
```

and if it corresponded to all there, then:

```
    global a: table[addr] of HostType = {
        [www.my-server.com] = Server,
    };
```

You can also use multiple index groupings across different indices:

```
    global access_allowed: table[addr, port] of bool = {
        [www.my-server.com, [21/tcp, 80/tcp]] = T,
    };
```

is equivalent to:

```
    global access_allowed: table[addr, port] of bool = {
        [155.26.27.2, 21/tcp] = T,
        [155.26.27.2, 80/tcp] = T,
        [155.26.27.8, 21/tcp] = T,
        [155.26.27.8, 80/tcp] = T,
        [155.26.27.44, 21/tcp] = T,
        [155.26.27.44, 80/tcp] = T,
    };
```

Fix me: add example of cross-product initialization of sets

3.12.3 Table Attributes

When declaring a table, you can specify a number of attributes that affect its operation:

`&default`

Specifies a value to yield when an index does not appear in the table. Syntax:

```
&default = expr
```

expr can have one of two forms. If it's type is the same as the table's yield type, then *expr* is evaluated and returned. If it's type is a `function` with arguments whose types correspond left-to-right with the index types of the table, and which returns a type the same as the yield type, then that function is called with the indices that yielded the missing value to compute the default value.

For example:

```
global a: table[count] of string &default = "nothing special";
```

will return the string `"nothing special"` anytime `a` is indexed with a `count` value that does not appear in `a`.

A more dynamic example:

```
function nothing_special(): string
    {
    if ( panic_mode )
        return "look out!";
    else
        return "nothing special";
    }

global a: table[count] of string &default = nothing_special;
```

An example of using a function that computes using the index:

```
function make_pretty(c: count): string
    {
    return fmt("**%d**", c);
    }

global a: table[count] of string &default = make_pretty;
```

`&create_expire`
Specifies that elements in the table should be *automatically deleted* after a given amount of time has elapsed since they were first entered into the table. Syntax:

```
&create_expire = expr
```

where *expr* is of type `interval`.

`&read_expire`
The same as `&create_expire` except the element is deleted when the given amount of time has lapsed since the last time the element was accessed from the table.

`&write_expire`
The same as `&create_expire` except the element is deleted when the given amount of time has lapsed since the last time the element was entered or modified in the table.

`&expire_func`
Specifies a function to call when an element is due for expression because of `&create_expire`, `&read_expire`, or `&write_expire`. Syntax:

```
&expire_func = expr
```

expr must be a function that takes two arguments: the first one is a table with the same index and yield types as the associated table. The second one is of type `any` and corresponds to the index(es) of the element being expired. The function must return an `interval` value. The `interval` indicates for how much longer the element should remain in the table; returning `0 secs` or a negative value instructs Bro to go ahead and delete the element.

Deficiency: The use of an `any` *type here is* temporary *and will be changing in the future to a general* tuple *notion.*

You specify multiple attributes by listing one after the other, *without* commas between them:

```
global a: table[count] of string &default="foo" &write_expire=5sec;
```

Note that you can specify each type of attribute only once. You can, however, specify more than one of `&create_expire`, `&read_expire`, and `&write_expire`. In that case, whenever any of the corresponding timers expires, the element will be deleted.

3.12.4 Accessing Tables

As usual, you access the values in tables by indexing them with a value (for a single index) or list of values (multiple indices) enclosed in `[]`'s. *Deficiency: Presently, when indexing a multi-dimensional table you must provide* all *of the relevant indices; you can't leave one out in order to extract a sub-table.*

You can also index arrays using `record`'s, providing the record is comprised of values whose types match that of the table's indices. (Any record fields whose types are themselves records are recursively unpacked to effect this matching.) For example, if we have:

```
local b: table[addr, port] of conn_id;
local c = 131.243.1.10;
local d = 80/tcp;
```

then we could index `b` using `b[c, d]`, but if we had:

```
local e = [$field1 = c, $field2 = d];
```

we could also index it using `a[d]`

You can test whether a table holds a given index using the `in` operator. For example:

```
[131.243.1.10, 80/tcp] in b
```

or

```
e in b
```

per the examples above. In addition, if the table has only a single index (not multi-dimensional), then you can omit the `[]`'s:

```
local active_connections: table[addr] of conn_id;
...
if ( 131.243.1.10 in active_connections )
    ...
```

3.12.5 Table Assignment

An indexed can be the target of an assignment:

```
b[131.243.1.10, 80/tcp] = c$id;
```

You can also assign to an entire table. For example, suppose we have the global:

```
global active_conn_count: table[addr, port] of count;
```

then we could later clear the contents of the table using:

```
local empty_table: table[addr, port] of count;
active_conn_count = empty_table;
```

Here the first statement declares a local variable `empty_table` with the same type as `active_conn_count`. Since we don't initialize the table, it starts out empty. Assigning it to `active_conn_count` then replaces the value of `active_conn_count` with an empty table. *Note: As with* `record`*'s, assigning* `table` *values results in a* shallow *copy.*

In addition to directly accessing an element of a table by specifying its index, you can also loop over all of the indices in a table using the `for` statement.

3.12.6 Deleting Table Elements

You can remove an individual element from a table using the `delete` statement:

```
delete active_host[c$id];
```

will remove the element in `active_host` corresponding to the connection identifier `c$id` (which is a `conn_id` record). If the element isn't present, nothing happens.

3.13 Sets

Sets are very similar to tables. The principle difference is that they are simply a collection of indices; they don't yield any values. You declare tables using the following syntax:

`set [` *type*$^+$ `]`

where, as with `table`s, *type*$^+$ is one or more scalar types (or records), separated by commas.

You initialize sets listing their elements in braces:

```
global a = { 21/tcp, 23/tcp, 80/tcp, 443/tcp };
```

which implicitly types `a` as a `set[port]` and then initializes it to contain the given 4 `port` values.

For multiple indices, you enclose each set of indices in brackets:

```
global b = { [21/tcp, "ftp"], [23/tcp, "telnet"], };
```

which implicitly `b` as `set[port, string]` and then initializes it to contain the given two elements. (As with tables, the comma after the last element is optional.)

As with tables, you can group together sets of indices:

```
global c = { [21/tcp, "ftp"], [[80/tcp, 8000/tcp, 8080/tcp], "http"], };
```

initializes `c` to contain 4 elements.

Also as with tables, you can use the `&create_expire`, `&read_expire`, and `&write_expire` attributes to control the automatic expiration of elements in a set. *Deficiency: However, the* `&expire_func` *attribute is not currently supported.*

You can test for whether a particular member is in a set using the `in` operator, as with tables. You add elements using the `add` statement:

```
add c[443/tcp, "https"];
```

and can remove them using the `delete` statement:

```
add d[21/tcp, "ftp"];
```

Also, as with tables, you can assign to the entire set, which assigns a *shallow* copy.

Finally, as with tables, you can loop over all of the indices in a set using the `for` statement.

3.14 Files

Deficiency: Bro currently supports only a very simple notion of files. You can only write to files, you can't read from them: and files are essentially untyped—the only values you can write to them are `string`*'s or values that can be converted to* `string`.

You declare `file` variables simply as type `file`:

```
global f: file;
```

You can create values of type `file` by using the `open` function:

```
f = open("suspicious_info.log");
```

will create (or recreate, if it already exists) the file *suspicious_info.log* and open it for writing. You can also use `open_for_append` to append to an existing file (or create a new one, if it doesn't exist).

You write to files using the `print` statement:

```
print f, 5 * 6;
```

will print the text `30` to the file corresponding to the value of `f`.

There is no restriction regarding how many files you can have open at a given time. In particular, even if your system has a limit imposed by `RLIMIT_NOFILE`s set by the system call `setrlimit`. If, however, you want to to close a file, you can do so using `close`, and you can test whether a file is open using `active_file`.

Finally, you can control whether a file is buffered using `set_buf`, and can flush the buffers of all open files using `flush_all`.

3.15 Functions

You declare a Bro `function` type using:

`function (` *argument** `) [ :` *`type`* `]`

where *argument*$^+$ is a (possibly empty) comma-separated list of arguments, and the final "`:` *type*" declares the return type of the function. It is optional; if missing, then the function does not return a value.

Each argument is declared using:

param-name `:` *type*

So, for example:

```
function(a: addr, p: port): string
```

corresponds to a function that takes two parameters, `a` of type `addr` and `p` of type `port`, and returns a value of type `string`.

You could furthermore declare:

```
global generate_id: function(a: addr, p: port): string;
```

to define `generate_id` as a variable of this type. Note that the declaration does *not* define the body of the function, and, indeed, `generate_id` could have different function bodies at different times, by assigning different function values to it.

When defining a function including its body, the syntax is slightly different:

`function` *func-name* `(` *argument** `)` [`:` *`type`*] { *statement** }

That is, you introduce *func-name*, the name of the function, between the keywork `function` and the opening parenthesis of the argument list, and you list the statements of the function within braces at the end.

For the previous example, we could define its body using:

```
function generate_id(a: addr, p: port): string
    {
    if ( a in local_servers )
        # Ignore port, they’re always the same.
        return fmt("server %s", a);

    if ( p < 1024/tcp )
        # Privileged port, flag it.
        return fmt("%s/priv-%s", a, p);

    # Nothing special - default formatting.
    return fmt("%s/%s", a, p);
    }
```

We also could have omitted the first definition; a function definition like the one immediately above automatically defines `generate_id` as a function of type `function(a: addr, p: port): string`. Note though that if *func-name* was indeed already declared, then the argument list much match *exactly* that of the previous definition. This includes the names of the arguments; *Unlike with C*, you cannot change the argument names between their first (forward) definition and the full definition of the function.

You can also define functions without using any name. These are referred to as anonymous functions, and are a type of expression.

You can only do two things with functions: call them or assign them. As an example of the latter, suppose we have:

```
local id_funcs: table[conn_id] of function(p: port, a: addr): string;
```

would declare a local variable indexed by a `conn_id` record value to yield a function of the same type as in the previous example. You could then execute:

```
id_funcs[c$id] = generate_id
```

or call whatever function is associated with a given `conn_id`:

```
print fmt("id is: %s", id_funcs[c$id](80/tcp, 1.2.3.4));
```

3.16 Event handlers

Event handlers are nearly identical in both syntax and semantics to functions, with the two differences being that event handlers have no return type since they never return a value, and you cannot call an event handler. You declare an event handler using:

`event (` *argument** `)`

So, or example,

```
local eh: event(attack_source: addr, severity: count)
```

declares the local variable `eh` to have a type corresponding to an event handler that takes two arguments, `attack_source` of type `addr`, and `severity` of type `count`.

To declare an event handler along with its body, the syntax is:

`event` *handler* `(` *argument** `)` { *statement** }

As with functions, you can assign event handlers to variables of the same type. Instead of calling event handlers like functions, though, instead they are *invoked*. This can happen in one of three ways:

From the event engine When the event engine detects an event for which you have defined a corresponding event handler, it queues an event for that handler. The handler is invoked as soon as the event engine finishes processing the current packet (and invoking any other event handlers that were queued first). The various event handlers known to the event engine are discussed in Chapter § 7, page 92.

Via the `event` statement The `event` statement queues an event for the given event handler for immediate processing. For example:

```
event password_exposed(c, user, password);
```

queues an inovocation of the event handler `password_exposed` with the arguments `c`, `user`, and `password`. Note that `password_exposed` must have been previously declared as an event handler with a compatible set of arguments.

Or, if we had a local variable `eh` as defined above, we could execute:

```
event eh(src, how_severe);
```

if `src` is of type `addr` and `how_severe` of type `count`.

Via the `schedule` expression The `schedule`expression queues an event for future invocation. For example:

```
schedule 5 secs { password_exposed(c, user, password) };
```

would cause `password_exposed` to be invoked 5 seconds in the future.

3.17 The any type

The any type is a type used internally by Bro to bypass strong typing. For example, the fmt function takes arguments of type any, because its arguments can be of different types, and of variable length. However, the any type is not supported for use by the user; while Bro lets you declare variables of type any, it does not allow assignment to them. This may change in the future. Note, though, that you can achieve some of the same effect using record values with &optional fields.

Chapter 4

Statements and Expressions

You express Bro's analysis of network traffic using *event handlers*, which, as discussed in § 3.16, page 41, are essentially subroutines written in Bro's policy scripting language. In this chapter we discuss the different types of statements and expressions available for expressing event handlers and the auxiliary functions they use.

4.1 Statements

Bro functions and event handlers are written in an imperative style, and the statements available for doing so are similar to those provided in C. As in C, statements are terminated with a semi-colon. There are no restrictions on how many lines a statement can span. Whitespace can appear between any of the syntatic components in a statement, and its presence always serves as a separator (that is, a single syntactic component cannot in general contain embedded whitespace, unless it is escaped in some form, such as appearing inside a string literal).

Bro provides the following types of statements:

expression

Syntax:

```
expr ;
```

As in C, an expression by itself can also be used as a statement. For example, assignments, calling functions, and scheduling timers are all expressions; they also are often used as statements.

`print`

Syntax:

```
print [ file ] expr-list ;
```

The expressions are converted to a list of strings, which are then printed as a comma-separated list. If the first expression is of type `file`, then the other expressions are printed to the corresponding file; otherwise they're written to *stdout*.

For control over how the strings are formatted, see `fmt` function.

`log`

Syntax:

```
log expr-list ;
```

The expressions are converted to a list of strings, which are then logged as a comma-separated list. “Logging” means recording the values to `bro_log_file`. In addition, if Bro is reading *live* network traffic (as opposed to from a trace file), then the messages are also reported via *syslog(3)* at level `LOG_NOTICE`. If the message does not already include a timestamp, one is added.

See the `log` module for a discussion of controlling logging behavior from your policy script. In particular, an important feature of the `log` statement is that prior to logging the giving string(s), Bro first invokes `log_hook` to determine whether to suppress the logging.

`event`

Syntax:

```
event expr ( expr-list* ) ;
```

Evaluates *expr* to obtain an event handler and queues an event for it with the value corresponding to the optional comma-separated list of values given by *expr-list*.

Note: `event` *statements look syntactically just like function calls, other than the keyword “*`event`*”. However, function calls are expressions, while queueing an event is not, since it does not return a value.*

`if`

Syntax:

```
if ( expr ) stmt
if ( expr ) stmt else stmt_2
```

Evaluates *expr*, which must yield a `bool` value. If true, executes *stmt*. For the second form, if false, executes *stmt*$_2$.

`for`

Syntax:

```
for ( var in expr ) stmt
```

Iterates over the indices of *expr*, which must evaluate to either a `set` or a `table`. For each iteration, *var* is set to one of the indices and *stmt* is executed. *var* needn’t have been previously declared (in which case its type is implicitly inferred from that of the indices of *expr*), and must not be a global variable.

If *expr* is a `set`, then the indices correspond to the members of the set. If *expr* is a `table`, then they correspond to the indices of the table.

Deficiency: You can only use `for` *statements to iterate over sets and tables with a single, non-compound index type. You can’t iterate over multi-dimensional or compound indices.*

Deficiency: Bro lacks ways of controlling the order in which it iterates over the indices.

`next`

Syntax:

```
next ;
```

Only valid within a `for` statement. When executed, causes the loop to proceed to the next iteration value (i.e., the next index value).

`break`

Syntax:

```
break ;
```

Only valid within a `for` statement. When executed, causes the loop to immediately exit.

`return`

Syntax:

return [*expr*] ;

Immediately exits the current function or event handler. For a function, returns the value *expr* (which is omitted if the function does not return a value, or for event handlers).

`add`

Syntax:

add $expr_1$ [$expr_2$] ;

Adds the element specified by $expr_2$ to the `set` given by $expr_1$. For example,

```
global active_hosts: set[addr, port];
...
add active_hosts[1.44.33.7, 80/tcp];
```

addes an element corresponding to the pair `1.44.33.7` and `80/tcp` to the set `active_hosts`.

`delete`

Syntax:

delete $expr_1$ [$expr_2$] ;

Deletes the corresponding value, where $expr_1$ corresponds to a set or table, and $expr_2$ an element/index of the set/table. If the element is not in the set/table, does nothing.

compound

Compound statements are formed from a list of (zero or more) statements enclosed in `{}`'s:

{ *statement** }

null

A lone:

```
;
```

denotes an empty, do-nothing statement.

`local`, `const`

Syntax:

```
local var [ : type ] [ = initialization ] [ attributes ] ;
const var [ : type ] [ = initialization ] [ attributes ] ;
```

Declares a local variable with the given type, initialization, and attributes, all of which are optional. The syntax of these fields is the same as for global variable declarations. The second form likewise declares a local variable, but one which is *constant*: trying to assign a new value to it results in an error. *Deficiency: Currently, this* `const` *restriction isn't detected/enforce.*

Unlike with C, the scope of a local variable is from the point of declaration to the end of the encompassing function or event handler.

4.2 Expressions

Expressions in Bro are very similar to those in C, with similar precedence:

parenthesized

Syntax:

```
( expr )
```

Parentheses are used as usual to override precedence.

constant

Any constant value (§ 3, page 21) is an expression.

variable

The name of a *variable*variables is an expression.

increment, decrement

Syntax:

```
++ expr
-- expr
```

Increments or decrements the given expression, which must correspond to an assignable value (variable, table element, or record element) and of a number type.

Yields the value of the expression after the increment.

Unlike with C, these operators only are defined for "pre"-increment/decrement; there is no post-increment/decrement.

negation

Syntax:

```
! expr
- expr
```

Yields the boolean § 3.2.2, page 22 or arithmetic negation for values of boolean or *numeric* (or `interval`) types, respectively.

positivation

Syntax:

```
+ expr
```

Yields the value of *expr*, which must be of type *numeric* or `interval`.

The point of this operator is to explicitly convert a value of type `count` to `int`. For example, suppose you want to declare a local variable `code` to be of type `int`, but initialized to the value `2`. If you used:

```
local code = 2;
```

then Bro's implicit typing would make it of type `count`, because that's the type of a constant specified by a string of digits. You could instead use:

```
local code = +2;
```

to direct the type inferencing to instead assign a type of `int` to `code`. Or, of course, you could specify the type explicitly:

```
local code:int = 2;
```

arithmetic

Syntax:

```
$expr_1$ + $expr_2$
$expr_1$ - $expr_2$
$expr_1$ * $expr_2$
$expr_1$ / $expr_2$
$expr_1$ % $expr_2$
```

The usual C arithmetic operators, defined for numeric types, except modulus (`%`) is only defined for integral types.

logical

Syntax:

```
$expr_1$ && $expr_2$
$expr_1$ || $expr_2$
```

The usual C logical operators, defined for boolean types.

equality

Syntax:

```
$expr_1$ == $expr_2$
$expr_1$ "!= $expr_2$
```

Compares two values for equality or inequality, yielding a `bool` value. Defined for all non-compound types except `pattern`.

relational

Syntax:

$expr_1$ `<` $expr_2$
$expr_1$ `<=` $expr_2$
$expr_1$ `>` $expr_2$
$expr_1$ `>=` $expr_2$

Compares two values for magnitude ordering, yielding a `bool` value. Defined for values of type *numeric*, `time`, `interval`, `port`, or `addr`.

Note: TCP `port` *values are considered less than UDP* `port` *values.*

Note: IPv4 `addr` *values less than IPv6* `addr` *values.*

Deficiency: Should also be defined at for `string` *values.*

conditional

Syntax:

$expr_1$ `?` $expr_2$ `:` $expr_3$

Evaluates $expr_1$ and, if true, evaluates and yields $expr_2$, otherwise evaluates and yields $expr_3$. $expr_2$ and $expr_3$ must have compatible types.

assignment

Syntax:

$expr_1$ `=` $expr_2$

Assigns the value of $expr_2$ to the storage defined by $expr_1$, which must be an assignable value (variable, table element, or record element). Yields the assigned value.

function call

Syntax:

$expr_1$ `(` *expr-list*$_2$ `)`

Evaluates $expr_1$ to obtain a value of type `function`, which is then invoked with its arguments bound left-to-right to the values obtained from the comma-separated list of expressions *expr-list*$_2$. Each element of *expr-list*$_2$ must be assignment-compatible with the corresponding formal argument in the type of $expr_1$. The list may (and must) be empty if the function does not take any parameters.

anonymous function

Syntax:

`function (` *parameters* `)` *body*

Defines an *anonymous function*, which, in abstract terms, is how you specify a constant of type `function`. *parameters* has the syntax of parameter declarations for regular function definitions, as does *body*, which is just a list of statements enclosed in braces.

Anonymous functions can be used anywhere you'd usually instead use a function declared in the usual direct fashion. For example, consider the function:

```
    function demo(msg: string): bool
        {
        if ( msg == "do the demo" )
            {
            print "got it";
            return T;
            }
        else
            return F;
        }
```

You could instead declare `demo` as a global variable of type function:

```
global demo: function(msg: string): bool;
```

and then later assign to it an anonymous function:

```
    demo = function (msg: string): bool
        {
        if ( msg == "do the demo" )
            {
            print "got it";
            return T;
            }
        else
            return F;
        };
```

You can even call the anonymous function directly:

```
    (function (msg: string): bool
        {
        if ( msg == "do the demo" )
            {
            print "got it";
            return T;
            }
        else
            return F;
        })("do the demo")
```

though to do so you need to enclose the function in parentheses to avoid confusing Bro's parser.

One particularly handy form of anonymous function is that used for `&default`.

event scheduling

Syntax:

```
schedule expr₁ { expr₂ ( expr-list₃ ) }
```

Evaluates *expr*$_1$ to obtain a value of type `interval`, and schedules the event given by *expr*$_2$ with parameters *expr-list*$_3$ for that time. Note that the expressions are all evaluated and bound at the time of execution of the `schedule` expression; evaluation is *not* deferred until the future execution of the event handler.

For example, we could define the following event handler:

```
event once_in_a_blue_moon(moon_phase: interval)
    {
    print fmt("wow, a blue moon - phase %s", moon_phase);
    }
```

and then we could schedule delivery of the event for 6 hours from the present, with a `moon_phase` of 12 days, using:

```
schedule +6 hr { once_in_a_blue_moon(12 days) };
```

Note: The syntax is admittedly a bit clunky. In particular, it's easy to (i) *forget to include the braces (which are needed to avoid confusing Bro's parser),* (ii) *forget the final semi-colon if the* `schedule` *expression is being used as an expression-statement, or* (iii) *erroneously place a semi-colon after the event specification but before the closing brace.*

Timer invocation is inexact. In general, Bro uses arriving packets to serve as its clock (when reading a trace file off-line, this is still the case—the timestamp of the latest packet read from the trace is used as the notion of "now"). Once this clock reaches or passes the time associated with a queued event, Bro will invoke the event handler, which is termed "expiring" the timer. (However, Bro will only invoke `max_timer_expires` timers per packet, and these include its own internal timers for managing connection state, so this can also delay invocation.)

It will also expire all pending timers (whose time has not yet arrived) when Bro terminates; if you don't want those event handlers to activate in this instance, you need to test `done_with_network`.

You would think that `schedule` should just be a statement like `event` invocation is, rather than an expression. But it actually does return a value, of the undocumented type `timer`. In the future, Bro may provide mechanisms for manipulating such timers; for example, to cancel them if you no longer want them to expire.

index

Syntax:

```
expr₁ [ expr-list₂ ]
```

Returns the sub-value of $expr_1$ indexed by the value of $expr\text{-}list_2$, which must be compatible with the index type of $expr_1$.

$expr\text{-}list_2$ is a comma-separated list of expressions (with at least one expression listed) whose values are matched left-to-right against the index types of $expr_1$.

The only type of value that can be indexed in this fashion is a `table`. *Note:* `set`*'s cannot be indexed because they do not yield any value. Use* `in` *to test for set membership.*

membership

Syntax:

$expr_1$ `in` $expr_2$

$expr_1$ `!in` $expr_2$

Yields true (false, respectively) if the index $expr_1$ is present in the `table` or `set` $expr_2$.

For example, if `alert_level` is a `table` index by an address and yielding a count:

```
global alert_level: table[addr] of count;
```

then we could test whether the address `127.0.0.1` is present using:

```
127.0.0.1 in alert_level
```

For `table`'s and `set`'s indexed by multiple dimensions, you enclose $expr_1$ in brackets. For example, if we have:

```
global connection_seen: set[addr, addr];
```

then we could test for the presence of the element indexed by `8.1.14.2` and `129.186.0.77` using:

```
[8.1.14.2, 129.186.0.77] in connection_seen
```

We can also instead use a corresponding `record` type. If we had

```
local t = [$x = 8.1.14.2, $y = 129.186.0.77]
```

then we could test:

```
t in connection_seen
```

pattern matching

Syntax:

$expr_1$ `==` $expr_2$

$expr_1$ `"!=` $expr_2$

$expr_1$ `in` $expr_2$

$expr_1$ `"!in` $expr_2$

As discussed for pattern values, the first two forms yield true (false) if the `pattern` *expr*$_1$ exactly matches the `string` *expr*$_2$. (You can also list the `string` value on the left-hand side of the operator and the `pattern` on the right.)

The second two forms yield true (false) if the `pattern` *expr*$_1$ is present within the `string` *expr*$_2$. (For these, you *must* list the pattern as the left-hand operand.)

record field access

Syntax:

```
expr $ field-name
```

Returns the given field *field-name* of the record *expr*. If the record does not contain the given field, a compile-time error results.

record constructor

Syntax:

```
[ field-constructor-list ]
```

Constructs a `record` value. The *field-constructor-list* is a comma-separated list of individual field constructors, which have the syntax:

```
$ field-name = expr
```

For example,

```
[$foo = 3, $bar = 23/tcp]
```

yields a `record` with two fields, `foo` of type `count` and `bar` of type `port`. The values used in the constructor needn't be constants, however; they can be any expression of an assignable type.

record field test

Syntax:

```
expr ?$ field-name
```

Returns true if the given field has been set in the record yielded by *expr*. Note that *field-name must* correspond to one of the fields in the record type of *expr* (otherwise, the expression would always be false). The point of this operator is to test whether an `&optional` field of a record has been assigned to.

For example, suppose we have:

```
type rap_sheet: record {
    num_scans: count &optional;
    first_activity: time;
};
global the_goods: table[addr] of rap_sheet;
```

and we want to test whether the address held in the variable `perp` exists in `the_goods` and, if so, whether `num_scans` has been assigned to, then we could use:

```
perp in the_goods && the_goods[perp]?$num_scans
```

Chapter 5

Global and Local Variables

5.1 Overview

Bro variables can be complicated to understand because they have a number of possibilities and features. They can be global or local in scope; modifiable or constant (unchangeable); explicitly or implicitly typed; optionally initialized; defined to have additional *attributes*; and, for global variables, *redefined* to have a different initialization or different attributes from their first declaration.

Rather than giving the full syntax for variable declarations, which is messy, in the following sections we discuss each of these facets of variables in turn, illustrating them with the minimal necessary syntax. However, keep in mind that the features can be combined as needed in a variable declaration.

5.1.1 Scope

`global` `local` *Global* variables are available throughout your policy script (once declared), while the scope of *local* variables is confined to the function or event handler in which they're declared. You indicate the variable's type using a corresponding keyword:

> `global` *name* `:` *type* `;`

or

> `local` *name* `:` *type* `;`

which declares *name* to have the given type and the corresponding scope.

You can intermix function/event handler definitions with declarations of global variables, and, indeed, they're in fact the same thing (that is, a function or event handler definition is equivalent to defining a global variable of type `function` or `event` and associating its initial value with that of the function or event handler). So the following is fine:

```
global a: count;

function b(p: port): string
    {
    if ( p < 1024/tcp )
```

```
            return "privileged";
        else
            return "ephemeral";
        }

    global c: addr;
```

However, you cannot mix declarations of global variables with global statements; the following is not allowed:

```
    print "hello, world";
    global a: count;
```

Local variables, on the other hand, can *only* be declared within a function or event handler. (Unlike for global statements, these declarations *can* come after statements.) Their scope persists to the end of the function. For example:

```
    function b(p: port): string
        {
        if ( p < 1024/tcp )
            local port_type = "privileged";
        else
            port_type = "ephemeral";

        return port_type;
        }
```

5.1.2 Modifiability

`const` For both global and local variables, you can declare that the variable *cannot be modified* by declaring it using the `const` keyword rather than `global` or `local`:

```
    const response_script = "./scripts/nuke-em";
```

Note that `const` variables *must* be initialized (otherwise, of course, there's no way for them to ever hold a useful value).

The utility of marking a variable as unmodifiable is for clarity in expressing your script—making it explicit that a particular value will never change—and also allows Bro to possibly optimize accesses to the variable (though it does little of this currently).

Note that `const` variables *can* be redefined via `redef`.

5.1.3 Typing

When you define a variable, you can *explicitly* type it by specifying its type after a colon. For example,

```
    global a: count;
```

directly indicates that `a`'s type is `count`.

However, Bro can also *implicitly* type the variable by looking at the type of the expression you use to initialize the variable:

```
global a = 5;
```

also declares `a`'s type to be `count`, since that's the type of the initialization expression (the constant `5`). There is no difference between this declaration and:

```
global a: count = 5;
```

except that it is more concise both to write and to read. In particular, Bro remains *strongly* typed, even though it also supports *implicit* typing; the key is that once the type is implicitly inferred, it is thereafter strongly enforced.

Bro's *type inference* is fairly powerful: it can generally figure out the type whatever initialization expression you use. For example, it correctly infers that:

```
global c = { [21/tcp, "ftp"], [[80/tcp, 8000/tcp, 8080/tcp], "http"], };
```

specifies that `c`'s type is `set[port, string]`. But for still more complicated expressions, it is not always able to infer the correct type. When this occurs, you need to explicitly specify the type.

5.1.4 Initialization

When defining a variable, you can optionally specify an initial value for the variable:

```
global a = 5;
```

indicates that the initial value of `a` is the value `5` (and also implicitly types `a` as type `count`, per § 5.1.3, page 54).

The syntax of an initialization is "= *expression*", where the given expression must be assignment-compatible with the variable's type (if explicitly given). Tables and sets also have special initializer forms, which are discussed in § 3.12.2, page 34 and § 3.13, page 38.

5.1.5 Attributes

When defining a variable, you can optionally specify a set of *attributes* associated with the variable, which specify additional properties associated with it. Attributes have two forms:

& *attr*

for attributes that are specified simply using their name, and

& *attr* = *expr*

for attributes that have a value associated with them.

The attributes `&redef` `&add_func` and `&delete_func`, pertain to redefining variables; they are discussed in § 5.1.6, page 56.

The attributes `&default`, `&create_expire`, `&read_expire`, `&write_expire`, and `&expire_func` are for use with `table`'s and `set`'s. See § 3.12.3, page 35 for discussion.

The attribute `&optional` specifies that a `record` field is optional. See § 3.11.4, page 32 for discussion.

Finall, to specify multiple attributes, you do *not* separate them with commas (doing so would actually make Bro's grammar ambiguous), but just list them one after another. For example:

```
global a: table[port] of string &redef &default="missing";
```

5.1.6 Refinement

To do.

```
   &redef
   &add_func
   &delete_func
```

Chapter 6

Predefined Variables and Functions

6.1 Predefined Variables

Bro predefines and responds to the following variables, organized by the policy file in which they are contained. Note that you will only be able to access the variables in a policy file if you `@load` it or a policy file which `@load` s it.

6.1.1 active.bro

`active_conn : table[conn_id] of connection`
A table of `connection` records corresponding to all active connections.

6.1.2 alert.bro

`alert_action_filters : table[Alert] of function(a: alert_info): AlertAction`
A table that maps each `Alert` into a function that should be called to determine the action.

`alert_file : file`
The file into which alerts are written.

6.1.3 anon.bro

`anon_log : file`
The file into which anonymization *Fix me: Add a reference to doc on anonymization when it is available.* IP address mappings are written.

`preserved_subnet : set[subnet]`
Addresses in these subnet are preserved when anonymization is being performed. See also `preserved_net`. NOTE: This variable is `const`, so may only be changed via `redef`.

`preserved_net : set[net]`
These Class A/B/C nets are preserved when anonymization is being performed. See also `preserved_subnet`.

6.1.4 backdoor.bro

`backdoor_log : file`
The file into which alerts about backdoor servers (`backdoor`) are written.

`backdoor_min_num_lines : count`
The number of lines of *Fix me: must be telnet?* input and output must be more than this amount to trigger backdoor checking.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_min_normal_line_ratio : double`
If the fraction of "normal" (less than a certain length) lines is below this value, then backdoor checking is not performed.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_min_bytes : count`
The total number of bytes transferred on the connection must be at least this large in order for backdoor checking to be performed.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_min_7bit_ascii_ratio : double`
The fraction of 7-bit ASCII characters out of all bytes transferred must be at least this large in order for backdoor checking to be performed.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_demux_disabled : bool`
If T (the default), then suspected backdoor connections are not demuxed into sender and receiver streams.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_demux_skip_tags : set[string]`
If the type of backdoor (the tag) is in this set, the connection will not be demuxed.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_ignore_src_addrs : table[string, addr] of bool`
If the suspected backdoor name ("*" for any) and source address (or its /16 or /24) subnet are in this table as a pair, then the backdoor will not be logged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_ignore_dst_addrs : table[string, addr] of bool`
If the suspected backdoor name ("*" for any) and destination address (or its /16 or /24) subnet are in this table as a pair, then the backdoor will not be logged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_ignore_ports : table[string, port] of bool`
The following (signature, well-known port) paits should not generated a backdoor alert.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_standard_ports : set[port]`
See `backdoor_annotate_standard_ports`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_stat_period : interval`

A report on backdoor stats is generated at this interval.
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_stat_backoff : interval`

*Fix me: N*ot sure about the exact definition here The backdoor report interval (`backdoor_stat_period`) is increased by this factor each time it is generated [, except if the timers are artificially expired].
NOTE: This variable is `const`, so may only be changed via `redef`.

`backdoor_annotate_standard_ports : bool`

If T (the default), backdoors alerts for those on `backdoor_standard_ports` should be annotated with the backdoor tag name.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ssh_sig_disabled : bool`

If T (default = F), then matches against the SSH signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`telnet_sig_disabled : bool`

If T (default = F), then matches against the telnet signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`telnet_sig_3byte_disabled : bool`

If T (default = F), then matches against the 3-byte telnet signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`rlogin_sig_disabled : bool`

If T (default = F), then matches against the rlogin signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`rlogin_sig_1byte_disabled : bool`

If T (default = F), then matches against the 1-byte rlogin signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`root_backdoor_sig_disabled : bool`

If T (default = F), then matches against the root backdoor signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ftp_sig_disabled : bool`

If T (default = F), then matches against the FTP signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`napster_sig_disabled : bool`

If T (default = F), then matches against the Napster signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`gnutella_sig_disabled : bool`

If T (default = F), then matches against the Gnutella signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`kazaa_sig_disabled : bool`

If T (default = F), then matches against the KaZaA signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`http_sig_disabled : bool`

If T (default = F), then matches against the HTTP signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`http_proxy_sig_disabled : bool`

If T (default = F), then matches against the HTTP proxy signature are ignored.
NOTE: This variable is `const`, so may only be changed via `redef`.

`did_sigconns : table[conn_id] of set[string]`

A table which indicates, for each connection, which backdoor server signatures were found in the connection's traffic, e.g., "ftp-sig" or "napster-sig".

`rlogin_conns : table[conn_id] of rlogin_conn_info`

A table that holds relevant state variables (an `rlogin_conn_info` record) for `rsh` connections.

`root_backdoor_sig_conns : set[conn_id]`

The set of connections for which a root backdoor signature ("root-bd-sig") has been detected.

`ssh_len_conns : set[conn_id]`

The set of connections that are predicted to contain SSH traffic, based on the proportion of packets that meet the expected packet size distribution. Relevant parameters are `ssh_min_num_pkts` and `ssh_min_ssh_pkts_ratio`, which are local to `backdoor`.

`ssh_min_num_pkts : count`

The minimum number of packets that look like SSH packets that allow a stream to be classified as such.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ssh_min_ssh_pkts_ratio : double`

The minimum fraction of packets in a stream that look like SSH packets that allow a stream to be classified as such.
NOTE: This variable is `const`, so may only be changed via `redef`.

`telnet_sig_conns : table[conn_id] of count`

The set of connections that are predicted to be Telnet connections, based on observation of the Telnet signature, the IAC byte (0xff).

`telnet_sig_3byte_conns : table[conn_id] of count`

Similar to `telnet_sig_conns`, but the signature matched is a whole 3-byte Telnet command sequence.

6.1.5 bro.init

`ignore_checksums : bool`

If T (default = F), packet checksums are not verified.
NOTE: This variable is `const`, so may only be changed via `redef`.

`partial_connection_ok : bool`

If T (the default), instantiate connection state when a partial connection (one missing its initial establishment negotiation) is seen.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_SYN_ack_ok : bool`

If T (the default), instantiate connection state when a SYN ack is seen but not the initial SYN (even if partial_connection_ok is false).

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_match_undelivered : bool`

If a connection state is removed there may still be some undelivered data waiting in the reassembler. If T (the default), pass this to the signature engine before flushing the state.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_SYN_timeout : interval`

Check up on the result of an initial SYN after this much time. *Fix me: What exactly does this mean? Check that the connection is active?*

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_session_timer : interval`

After a connection has closed, wait this long for further activity before checking whether to time out its state.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_connection_linger : interval`

When checking a closed connection for further activity, consider it inactive if there hasn't been any for this long. Complain if the connection is reused before this much time has elapsed.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_attempt_delayv : interval`

Wait this long upon seeing an initial SYN before timing out the connection attempt.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_close_delay : interval`

Upon seeing a normal connection close, flush state after this much time.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_reset_delay : interval`

Upon seeing a RST, flush state after this much time.

NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_partial_close_delay : interval`

Generate a connection_partial_close event this much time after one half of a partial connection closes, assuming there has been no subsequent activity.

NOTE: This variable is `const`, so may only be changed via `redef`.

`non_analyzed_lifetime : interval`

If a connection belongs to an application that we don't analyze, time it out after this interval. If 0 secs, then don't time it out.

NOTE: This variable is `const`, so may only be changed via `redef`.

`inactivity_timeout : interval`

If a connection is inactive, time it out after this interval. If 0 secs, then don't time it out.
NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_storm_thresh : count`

This many FINs/RSTs in a row constitutes a "storm". See also `tcp_storm_interarrival_thresh`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_storm_interarrival_thresh : interval`

The FINs/RSTs must come with this much time or less between them to be considered a storm. See also `tcp_storm_thresh`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_reassembler_ports_orig : set[port]`

For services without a handler, these sets define which side of a connection is to be reassembled. *Fix me: What is the point of this exactly? What are you analyzing?*
NOTE: This variable is `const`, so may only be changed via `redef`.

`tcp_reassembler_ports_resp : set[port]`

For services without a handler, these sets define which side of a connection is to be reassembled. *Fix me: What is the point of this exactly? What are you analyzing?*
NOTE: This variable is `const`, so may only be changed via `redef`.

`table_expire_interval : interval`

Check for expired table entries after this amount of time *Fix me: Which tables?*
NOTE: This variable is `const`, so may only be changed via `redef`.

`dns_session_timeout : interval`

Time to wait before timing out a DNS request.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ntp_session_timeout : interval`

Time to wait before timing out an NTP request.
NOTE: This variable is `const`, so may only be changed via `redef`.

`rpc_timeout : interval`

Time to wait before timing out an RPC request.
NOTE: This variable is `const`, so may only be changed via `redef`.

`watchdog_interval : interval`

A SIGALRM is set for this interval to make sure that Bro does not get caught up doing something for too long. *Fix me: True?* If this happens, Bro is termination after doing a dump of all remaining packets.
NOTE: This variable is `const`, so may only be changed via `redef`.

`heartbeat_interval : interval`

After each interval of this length, update the `net_stats` variable.
NOTE: This variable is `const`, so may only be changed via `redef`.

`anonymize_ip_addr : bool`

If true (default = false), then IP addresses are anonymized in alert and log generation.
NOTE: This variable is `const`, so may only be changed via `redef`.

`omit_rewrite_place_holder : bool`

If true, omit place holder packets when rewriting. *Fix me: Should this go somewhere else?*
NOTE: This variable is `const`, so may only be changed via `redef`.

`rewriting_http_trace : bool`

If true (default = F), HTTP traces are rewritten.
NOTE: This variable is `const`, so may only be changed via `redef`.

`rewriting_smtp_trace : bool`

If true (default = F), SMTP traces are rewritten.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.6 code-red.bro

`code_red_log : file`

The file into which Code Red-related alerts are written.

`code_red_list1 : table[addr] of count`

A table which contains, for each IP address, how many Code Red I attacks were observed (based on a signature) by the machine at that address.

`code_red_list2 : table[addr] of count`

A table which contains, for each IP address, how many Code Red II attacks were observed (based on a signature) by the machine at that address.

`local_code_red_response_pgm : string`

By default, an empty string; if `&redef`ed, the specified program will be invoked with the attack source IP as the argument the first time an attack from that IP is observed.

`remote_code_red_response_pgm : string`

By default, an empty string; if `&redef`ed, the specified program will be invoked with the attack destination IP as the argument the first time an attack on that IP is observed.

6.1.7 conn.bro

`have_FTP : bool`

If true, `ftp.bro` has been loaded.

`have_SMTP : bool`

If true, `smtp.bro` has been loaded.

`have_stats : bool`

True if `net_stats` was ever updated with packet capture statistics.

`hot_conns_reported : set[string]`

The set of connections (indexed by the entire 'hot' message) that have previously been flagged as `hot`.

`last_stat : net_stats`

The last recorded snapshot of packet capture statistics, in a `net_stats` record.

`last_stat_time : time`

The last time that network statistics were read into `net_stats`.

`RPC_server_map : table[addr, port] of string`

Maps a given port on a given server's address to an RPC service. If we haven't loaded `portmapper.bro`, then it will be empty; see `portmapper.bro` and the `portmapper` module documentation for more information.

6.1.8 demux.bro

For more information on demultiplexing of connections, see the demux module(§ 10.17, page 178).

`demux_dir : string`

The name of the directory which will contain the files with demultiplexed connection data.

`demuxed_conn : set[conn_id]`

The set of connections that are currently being demultiplexed.

6.1.9 dns.bro

`actually_rejected_PTR_anno : set[string]`

Annotations that if returned for a PTR lookup actually indicate a rejected query; for example, ”illegal-address.lbl.gov”.
NOTE: This variable is `const`, so may only be changed via `redef`.

`sensitive_lookup_hosts : set[addr]`

Hosts in this set generate an alert when they are returned in PTR queries, unless the originating host is in `sensitive_lookup_hosts`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`okay_to_lookup_sensitive_hosts : set[addr]`

If the DNS request originator is in this set, then it is allowed to look up “sensitive” hosts (see also `sensitive_lookup_hosts` without causing an alert.

`dns_log : file`

The file into which DNS-related alerts are written.

`dns_sessions : table[addr, addr] of dns_session_info`

A table of outstanding DNS sessions indexed by [client IP, server IP]. *Fix me: Need to illustrate dns_sessions_info.*

`num_dns_sessions : count`

The total number of entries that have ever been in the `dns_sessions` table.

`distinct_PTR_requests : table[addr, string] of count`

The number of DNS PTR requests obseverd with the given source address and request string.

`distinct_rejected_PTR_requests : table[addr] of count`
How many DNS PTR requests from the given source address were rejected. A report is generated if this number crosses a threshold, namely, `report_rejected_PTR_thresh`.

`distinct_answered_PTR_requests : table[addr] of count`
How many DNS PTR requests from the given source address were rejected.

`report_rejected_PTR_thresh : count`
If this many DNS requests from a host are rejected, generate a possible PTR scan event.

`report_rejected_PTR_factor : double`
If DNS requests from a host are rejected more than accepted by this factor, generate a `PTR_scan` event.

`allow_PTR_scans : set[addr]`
The set of hosts for which a `PTR_scan` event does not generate a report (that is, the scan is allowed).

`did_PTR_scan_event : table[addr] of count`
A table of hosts for which a `PTR_scan` event has been generated.

6.1.10 dns-mapping.bro

`dns_interesting_changes : h`
e set of DNS mapping changes (according to lookups by Bro itself) that is interesting enough to alert on.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.11 finger.bro

`hot_names : set[string]`
If a finger request for any of the names in this set is observed, the associated connection is marked "hot".
NOTE: This variable is `const`, so may only be changed via `redef`.

`max_finger_request_len : count`
If a finger request is longer than this length, then it is marked as "hot".
NOTE: This variable is `const`, so may only be changed via `redef`.

`rewrite_finger_trace : bool`
Indicates whether or not finger requests are rewritten for anonymity.

6.1.12 ftp.bro

`ftp_log : file`
The file into which FTP-related alerts are written.

`ftp_sessions : table[conn_id] of ftp_session_info`

`ftp_guest_ids : set[string]`
The set of login IDs which are guest logins, e.g., "anonymous" and "ftp".
NOTE: This variable is `const`, so may only be changed via `redef`.

`ftp_skip_hot : set[addr, addr, string]`

Indexed by source and destination addresses and the id, these connections are not marked as “hot” even if its data would to cause it to be otherwise.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ftp_hot_files : pattern`

If a filename matching this pattern is requested, the `ftp_sensitive_file` event is generated. The default behavior is to log the connection.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ftp_hot_guest_files : pattern`

If a user is logged in under a guest ID and attempts to retrieve a file matching this pattern, the `ftp_sensitive_file` event is generated. The default behavior is to log the connection.
NOTE: This variable is `const`, so may only be changed via `redef`.

`ftp_hot_cmds : table[string] of pattern`

If an FTP command matches an index into the table and its argument matches the associated pattern, the connection is logged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_unexpected : set[addr]`

Pairs of IP addresses for which we shouldn’t bother logging if one of them is used in lieu of the other in a PORT or PASV directive.

`skip_unexpected_net : set[addr]`

Similar to `skip_unexpected`, but matches a /24 subnet.

`ftp_data_expected : table[addr, port] of addr`

Indexed by the server’s responder pair, yields the address expected to make an FTP data connection to it.

`ftp_data_expected_session : table[addr, port] of ftp_session_info`

Indexed by the server’s responder pair, yields the associated `ftp_session_info` record for the expected incoming FTP data connection.

`ftp_excessive_filename_len : count`

If an FTP request filename meets or exceeds this length, an `FTP_ExcessiveFilename` alert is generated.

`ftp_excessive_filename_trunc_len : count`

How much of the excessively long filename is printed in the alert message.

`ftp_ignore_invalid_PORT : pattern`

Invalid PORT/PASV directives that exactly match this pattern don’t generate alerts.

`ftp_ignore_privileged_PASVs : set[port]`

If an FTP PASV port is specified to be a privileged port (¡ 1024/tcp) then an `FTP_PrivPort` event is generated, EXCEPT if the port is in this set.

6.1.13 hot.bro

`same_local_net_is_spoof : bool`

If true (default = F), it should be considered a spoofing attack if a connection has the same local net for source and destination.
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_spoof_services : set[port]`

The services in this set are not counted as spoofed even if they pass the test from `same_local_net_is_spoof`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_pairs : set[addr, addr]`

Connections between these (source address, destination address) pairs are never marked as “hot”.
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_16_net_pairs : set[addr, addr]`

Connections between these (/16 network, /32 destination host) pairs are never marked as “hot”.
NOTE: This variable is `const`, so may only be changed via `redef`.

`hot_srcs : table[addr] of string`

Connections from any of these sources are automatically marked “hot” with the associated message in the table.
NOTE: This variable is `const`, so may only be changed via `redef`.

`hot_dsts : table[addr] of string`

Connections to any of these destinations are automatically marked “hot” with the associated message in the table.
NOTE: This variable is `const`, so may only be changed via `redef`.

`hot_src_24nets : table[addr] of string`

Connections from any of these source /24 nets are automatically marked “hot” with the associated message in the table.
NOTE: This variable is `const`, so may only be changed via `redef`.

`hot_dst_24nets : table[addr] of string`

Connections to any of these destination /24 nets are automatically marked “hot” with the associated message in the table.
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_services : set[port]`

Connections to this set of services are never marked “hot” (based on port number).
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_services_to : set[addr, port]`

Connections to the specified host and port are never marked “hot”.
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_service_pairs : set[addr, addr, port]`

Connections from the first address to the second on the specified destination port are never marked “hot”.
NOTE: This variable is `const`, so may only be changed via `redef`.

`flag_successful_service : table[port] of string`

Successful connections to any of the specified ports are flagged with the accompanying message. Examples are popular backdoor ports.
NOTE: This variable is `const`, so may only be changed via `redef`.

`flag_successful_inbound_service : table[port] of string`

Incoming connections to the specified ports are flagged with the accompanying message. This is similar to `flag_successful_service`, but may be used when the port gives to many false positives for outgoing connections.
NOTE: This variable is `const`, so may only be changed via `redef`.

`terminate_successful_inbound_service : table[port] of string`

Connections to this port, if previously flagged by `flag_successful_service` or `flag_successful_incoming_service`, are terminated.
NOTE: This variable is `const`, so may only be changed via `redef`.

`flag_rejected_service : table[port] of string`

Failed connection attempts to the specified ports are marked as “hot”.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.14 hot-ids.bro

`forbidden_ids : set[string]`

If any of these usernames/login IDs are used, the corresponding connection is terminated.
NOTE: This variable is `const`, so may only be changed via `redef`.

`forbidden_ids_if_no_password : set[string]`

If any of these usernames/login IDs are used with no password, the corresponding connection is terminated.
NOTE: This variable is `const`, so may only be changed via `redef`.

`forbidden_id_patterns : pattern`

If a username/login ID matches this pattern, the corresponding connection is terminated.
NOTE: This variable is `const`, so may only be changed via `redef`.

`always_hot_ids : set[string]`

Connections that attempt to login with these IDs are always marked “hot”, whether or not they succeed. See also `hot_ids`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`hot_ids : set[string]`

Similar to `always_hot_ids`, except that only successful connections are marked “hot”.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.15 http.bro

`http_log : file`

The file into which HTTP-related alerts are written.

`http_sessions : table[addr, addr] of http_session_info`
A [source, destination] indexed table of `http_session_info` records.

`include_HTTP_abstract : bool`
Currently used to indicate whether or not an abstract of the HTTP request data will be included in a rewritten connection.

`log_HTTP_data : bool`
If true, an abstract of the HTTP request data is included in a log message.

`maintain_http_sessions : bool`
If true, HTTP sessions are maintained across multiple connections, otherwise we not (which saves some memory).

`process_HTTP_replies : bool`
If true, HTTP replies (not just requests) are processed.

`process_HTTP_data : bool`
If true, HTTP data is examined as needed (e.g., for making HTTP abstracts, as discussed below).

6.1.16 http-abstract.bro

`http_abstract_max_length : count`
The maximum number of bytes used to store an abstract for an HTTP connection.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.17 http-request.bro

`skip_remote_sensitive_URIs : pattern`
URIs matching this pattern should not be considered sensitive if accessed remotely, i.e., by a local client.

`have_skip_remote_sensitive_URIs : bool`
Due to a quirk in Bro, this must be redef'ed to T if you want to use `skip_remote_sensitive_URIs`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`sensitive_URIs : pattern`
URIs matching this pattern, but not matching `worm_URIs`, are logged. See also `skip_remote_sensitive_URIs`, `sensitive_post_URIs`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`worm_URIs : pattern`
URIs matching this pattern are not logged even if they match `sensitive_URIs`, since worms are so common they would clutter the logs.
NOTE: This variable is `const`, so may only be changed via `redef`.

`sensitive_post_URIs : pattern`
URIs matching this pattern are logged if they are used with the HTTP “POST” method (rather than “GET”).
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.18 icmp.bro

`icmp_flows : table[icmp_flow_id] of icmp_flow_info`

A table tracking all ICMP "flows" by `icmp_flow_id`. "Flows", which are simply inferred related sequences of packets between two machines, based on ICMP ID, are timed out after (currently) 30 seconds of inactivity.

6.1.19 ident.bro

`hot_ident_ids : set[string]`

If any of the User IDs in this set are returned in an `ident` response, an *IdentSensitiveID* alert is generated.

`hot_ident_exceptions : set[string]`

Exceptions to the `hot_ident_ids` set.

`public_ident_user_ids : set[string]`

User IDs in this set are described as "public" in a rewritten `ident` trace.

`public_ident_systems : set[string]`

Operating system names in this set (e.g., "UNIX") are reported directly in a rewritten `ident` trace; other OSes will be reported as "OTHER".

`rewrite_ident_trace : bool`

If true, traces will be rewritten (partially anonymized).

6.1.20 interconn.bro

`interconn_conns : table [conn_id] of conn_info`

A `conn_id`-indexed table of all currently-tracked interactive connections. The table entries are `conn_info` records containing some very basic information about the connection.

`interconn_log : file`

The file into which generic interactive-connection-related alerts are written.

`interconn_min_interarrival : interval`

Used in computing the "alpha" parameter, which is used to determine which connections are interactive, based on the distribution of interarrival times. See also `interconn_max_interarrival`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_max_interarrival : interval`

Used in computing the "alpha" parameter, which is used to determine which connections are interactive, based on the distribution of interarrival times. See also `interconn_min_interarrival`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_max_keystroke_pkt_size : count`

The maximum packet size used to classify keystroke-containing packets.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_default_pkt_size : count`

The estimated packet size used to calculate the number of packets missed when we see an ack above a hole. *Fix me: Please verify.*
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_stat_period : interval`

How often to generate a report of interconn stats.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_stat_backoff : double`

Fix me: I don't fully understand is_expire in timers. The stat report generation interval (`interconn_stat_period`) is increased by this factor each time the report is generated [, unless the report is generated because all timers are artifically expired].
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_num_pkts : count`

A connection must have this number of packets transferred before it may be classified as interactive.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_duration : interval`

A connection must last least this long before it may be classified as interactive.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_ssh_len_disabled : bool`

If false (default = T), and at least one side of the connection has partial state (the initial negotiation was missed), then packets are examined to see if they fit the size distribution associated with interactive SSH connections.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_ssh_pkts_ratio : double`

Analogous to `ssh_min_ssh_pkts_ratio`, except used in the context described in `interconn_ssh_len_disabled`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_bytes : count`

The number of bytes transferred on a connection must be at least this high before the connection may be classified as interactive.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_7bit_ascii_ratio : double`

The ratio of 7-bit ASCII characters to total bytes must be at least this high before the connection may be classified as interactive.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_num_lines : count`

The number of lines transferred on a connection must be at least this high before the connection may be classified as interactive.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_normal_line_ratio : double`
The ratio of “normal” lines to total lines must be at least this high before the connection may be classified as interactive. A normal line, roughly speaking, is one whose length is within a certain bound. *Fix me: Please verify this.*
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_alpha : double`
The “alpha” parameter computed on connection must be at least this high before the connection may be classified as interactive. This parameter measures certain properties of packet interarrival times. See `interconn`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_min_gamma : double`
The “gamme” parameter computed on connection must be at least this high before the connection may be classified as interactive.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_standard_ports : set[port]`
Connections to or from these ports are marked as interactive automatically, unless `interconn_ignore_standard_ports` is set to true.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_ignore_standard_ports : bool`
If true (default = F), then all connections are analyzed for interactive patterns, regardless of port. See `interconn_standard_ports`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`interconn_demux_disabled : bool`
If false (default = T), then interactive connections are demuxed when being logged.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.21 login.bro

`input_trouble : pattern`
If a user’s keystroke input matches this pattern, then an alert is generated.

`edited_input_trouble : pattern`
If a user’s keystroke input matches this pattern, taking into account backspace and delete characters, then an alert is generated.

`full_input_trouble : pattern`
If this pattern is matched in a full line of input, an alert is generated.

`input_wait_for_output : pattern`
The same as `edited_input_trouble`, except that the alert is delayed until the corresponding output is seen, so that both may be logged together.

`output_trouble : pattern`
If the login output matches this pattern, an alert is generated.

`full_output_trouble : pattern`
Similar to `output_trouble`, but the pattern must match the entire output.

`backdoor_prompts : pattern`
If the login output matches this text, but not `non_backdoor_prompts`, generate a possible-backdoor alert.

`non_backdoor_prompts : pattern`
See `backdoor_prompts`.

`hot_terminal_types : pattern`
If the terminal type used matches this pattern, generate an alert.

`hot_telnet_orig_ports : set[port]`
If the source port of a telnet connection is in this set, generate an alert.

`skip_authentication : set[string]`
If a string in this set appears where an authentication prompt would normally, skip processing of authentication (typically for an unauthenticated system). *Fix me: Please verify.*
NOTE: This variable is `const`, so may only be changed via `redef`.

`login_prompts : set[string]`
The set of strings that are recognized as login prompts anywhere on a line, e.g., "Login:".
NOTE: This variable is `const`, so may only be changed via `redef`.

`login_failure_msgs : set[string]`
If any of these strings appear on a line following an authentication attempt, the attempt is considered to have failed, unless a string from `login_non_failure_msgs` also appears on the line. This set has higher precedence than `login_success_msgs`, and the same precedence as `login_timeouts`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`login_non_failure_msgs : set[string]`
If any of these strings appear on a line following an authentication attempt, the connection is not considered to have failed even if `login_failure_msgs` indicates otherwise.
NOTE: This variable is `const`, so may only be changed via `redef`.

`login_success_msgs : set[string]`
If any of these messages is seen, the connection attempt is assumed to have succeeded. This set has lower precedence than `login_failure_msgs` and `login_timeouts`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`login_timeouts : set[string]`
If any of these messages is seen during the login phase, the connection attempt is assumed to have timed out. This set has higher precedence than `login_success_msgs`, and the same precedence as `login_failure_msgs`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`router_prompts : pattern`
Fix me: Don't know what this is

`non_ASCII_hosts : set[addr]`
The set of hosts that do not use ASCII (and to whom logins are thus not processed).

`skip_logins_to : set[addr]`
Do not process logins to this set of hosts.

`always_hot_login_ids : pattern`
Login names which generate an alert even if the login is not successful.

`hot_login_ids : pattern`
Login names which generate an alert, if the login is successful.

`rlogin_id_okay_if_no_password_exposed : set[string]`
Login names in this set are those which are normally considered sensitive, but are allowed if the associated password is not exposed.

`login_sessions : table[conn_id] of login_session_info`
A table, indexed by connection ID, of `login_session_info` records, characterizing each login session.

6.1.22 mime.bro

`mime_log : file`
MIME message-related alerts are logged to this file.

`mime_sessions : table[conn_id] of mime_session_info`
A table, indexed by connection ID, of `mime_session_info` records, characterizing each MIME session.

`check_relay_3 : function(session: mime_session_info, msg_id: string): bool`
Fix me: Don't know about this

`check_relay_4 : function(session: mime_session_info, content_hash: string): bool`
Fix me: Don't know about this

6.1.23 ntp.bro

`excessive_ntp_request : count`
NTP requests over this length are considered "excessive" and will be flagged (marked "hot").
NOTE: This variable is `const`, so may only be changed via `redef`.

`allow_excessive_ntp_requests : set[addr]`
NTP requests from an address in this set are never considered excessively long (see `excessive_ntp_request`).
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.24 port-names.bro

`port_names : table[port] of string`
A mapping of well-known port numbers to the associated service names.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.25 portmapper.bro

`rpc_programs : table[count] of string`

A table correlating numeric RPC service IDs to string names of the services, e.g., `[1000000] = ‘‘portmapper’’`.

`NFS_services : set[string]`

A set of string names of NFS-related RPC services.
NOTE: This variable is `const`, so may only be changed via `redef`.

`RPC_okay : set[addr, addr, string]`

Indexed by the host providing the service, the host requesting it, and the service; do not log Sun portmapper requests from the specified requestor to the specified provider for the specified service.
NOTE: This variable is `const`, so may only be changed via `redef`.

`RPC_okay_nets : set[net]`

Hosts in any of the networks in this set may make portmapper requests without being flagged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`RPC_okay_services : set[string]`

Requests for services in this set will not be flagged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`NFS_world_servers : set[addr]`

Any host may request NFS services from any of the machines in this set without being flagged..
NOTE: This variable is `const`, so may only be changed via `redef`.

`any_RPC_okay : set[addr, string]`

Indexed by the service provider and the service (in string form); any host may access these services without being flagged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`RPC_dump_okay : set[addr, addr]`

Indexed by requesting host and providing host, respectively; dumps of RPC portmaps are allowed between these pairs.
NOTE: This variable is `const`, so may only be changed via `redef`.

`RPC_do_not_complain : set[string, bool]`

Indexed by the portmapper request and a boolean that’s T if the request was answered, F it was attempted but not answered. If there’s an entry in the set matching the current request/attempt, then the access won’t be logged (unless the connection is hot for some other reason).

`suppress_pm_log : set[addr, string]`

Indexed by source and portmapper service. If set, we already logged and shouldn’t do so again. *Fix me: Presumably this can be preloaded with stuff, or we wouldn’t need to document it.*

6.1.26 rules.bro

`rule_actions : table[string] of count`

Decide what to do when each rule (the index into the table) triggers: Ignore the rule (RULE_IGNORE); Process the rule but don't report it individually (RULE_QUIET); Log the match into `rule_file` (RULE_FILE); Log the match into both `rule_file` and the overall log file (generate an alert) (RULE_LOG). The default is RULE_FILE.
NOTE: This variable is `const`, so may only be changed via `redef`.

`rule_file : file`

The file into which rule-based alerts are logged.

`horiz_scan_thresholds : set[count]`

Log if for a pair (orig, rule) the number of different responders has reached one of the thresholds in this set.
NOTE: This variable is `const`, so may only be changed via `redef`.

`vert_scan_thresholds : set[count]`

Log if for a pair (orig, resp) the number of different rule matches has reached one of the thresholds in this set.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.27 scan.bro

`suppress_scan_checks : bool`

If true, we suppress scan checking (we still do account-tried accounting). This is provided because scan checking can consume a lot of memory.

`report_peer_scan : set[count]`

When the number of distinct machines connected to by a given external host reaches each of the levels in the set, an alert is generated.

NOTE: This variable is `const`, so may only be changed via `redef`.

`report_outbound_peer_scan : set[count]`

When the number of distinct machines connected to by a given internal host reaches each of the levels in the set, an alert is generated.

NOTE: This variable is `const`, so may only be changed via `redef`.

`num_distinct_peers : table[addr] of count`

A table indexed by a host's address which indicates how many distinct machines that host has connected to.

`distinct_peers : set[addr,addr]`

A table indexed by source host and target machine that tracks which machines have been scanned by each host.

`num_distinct_ports : table[addr] of count`

A table indexed by a host's address which indicates how many distinct ports that host has connected to.

`distinct_ports : set[addr, port]`

A table indexed by source host and target port that tracks which ports have been scanned by each host.

`report_port_scan : set[count]`

When the number of distinct ports connected to by a given external host reaches each of the levels in the set, an alert is generated.

NOTE: This variable is `const`, so may only be changed via `redef`.

`possible_port_scan_thresh : count`

If a host tries to connect to more than this number of ports, it is considered a possible scanner.

NOTE: This variable is `const`, so may only be changed via `redef`.

`possible_scan_sources : set[addr]`

Hosts are put in this set once they have scanned more than `possible_port_scan_thresh` ports.

`num_scan_triples : table[addr, addr] of count`

Indexed by source address and destination address, the number of services scanned for on the latter by the former. This is only tracked for `possible_scan_sources`.

`scan_triples : set[addr, addr, port]`

For `possible_scan_sources` as a source address, the triples of (source address, destination address, and service/port) scanned.

`accounts_tried : set[addr, string, string]`

Which account names were tried, indexed by source address, user name tried, password tried.

`num_accounts_tried : table[addr] of count`

How many accounts, as defined by a (user name, password) pair, were tried by the host with the given address.

`report_accounts_tried : set[count]`

When the number of distinct accounts (username, password) tried by a given external host reaches each of the levels in the set, an alert is generated.
NOTE: This variable is `const`, so may only be changed via `redef`.

`report_remote_accounts_tried : set[count]`

When the number of distinct remote accounts (username, password) tried by a given internal host reaches each of the levels in the set, an alert is generated.
NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_accounts_tried : set[addr]`

Hosts in this set are not subject to alerts based on `report_accounts_tried` and `report_remote_accounts_tried`.
NOTE: This variable is `const`, so may only be changed via `redef`.

`addl_web : set[port]`

Ports in this set are treated as HTTP services.
NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_services : set[port]`

Connections to ports in this set are ignored for the purposes of scan detection.
NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_outbound_services : set[port]`

Connections to external machines on ports in this set are ignored for the purposes of scan detection.

NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_scan_sources : set[addr]`

Hosts in this set are ignored as possible sources of scans.

NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_scan_nets_16 : set[addr,port]`

Connections matching the specified (source host /16 subnet, port) pairs are ignored for the purpose of scan detection.

NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_scan_nets_24 : set[addr,port]`

Connections matching the specified (source host /24 subnet, port) pairs are ignored for the purpose of scan detection.

NOTE: This variable is `const`, so may only be changed via `redef`.

`backscatter_ports : set[port]`

Reverse (SYN-ack) scans seen from these ports are considered to reflect possible SYN flooding backscatter and not true (stealth) scans.

NOTE: This variable is `const`, so may only be changed via `redef`.

`num_backscatter_peers : table[addr] of count`

Indexed by a host, how many other hosts it connected to with a possible backscatter signature.

`distinct_backscatter_peers : table[addr, addr] of count`

A table of [source, destination] observed backscatter activity; the table entry is a count of backscatter packets from the source to the destination.

`report_backscatter : set[count]`

When the number of machines that a host has sent backscatter packets to reaches each of the levels in the set, an alert is generated.

Fix me: Need to document connection-dropping related variables.

```
global can_drop_connectivity = F &redef;
global drop_connectivity_script = "drop-connectivity" &redef;
global connectivity_dropped: set[addr];
const shut_down_scans: set[port] &redef;
const shut_down_all_scans = F &redef;
const shut_down_thresh = 100 &redef;
\indpredefvar{never_shut_down}{set[addr]}
\indpredefvar{never_drop_nets}{set[net]}
\indpredefvar{never_drop_16_nets}{set[net]}
\indpredefvar{did_drop_address}{table[addr] of count}
```

`root_servers : set[host]`

The set of root DNS servers.

NOTE: This variable is `const`, so may only be changed via `redef`.

`gtld_servers : set[host]`
The set of Generic Top-Level Domain servers (.com, .net, .org, etc.).
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.28 site.bro

`local_nets : set[net]`
Class A/B/C networks that are considered “local”.
NOTE: This variable is `const`, so may only be changed via `redef`.

`local_16_nets : set[addr]`
/16 address blocks that are considered “local”. These are derived directly from `local_nets`. *Fix me: Please verify this*.
NOTE: This variable is `const`, so may only be changed via `redef`.

`local_24_nets : set[addr]`
/24 address blocks that are considered “local”. These are derived directly from `local_nets`. *Fix me: Please verify this*.
NOTE: This variable is `const`, so may only be changed via `redef`.

`neighbor_nets : set[net]`
Class A/B/C networks that are considered “neighbors”. Note that unlike for local_nets, `neighbor_16_nets` is *not* merely a /16 addr version of neighbor_nets, but instead is consulted *in addition* to neighbor_nets.
NOTE: This variable is `const`, so may only be changed via `redef`.

`neighbor_16_nets : set[addr]`
/16 address blocks that are considered “neighbors”. Note that unlike for local_nets, neighbor_16_nets is *not* merely a /16 addr version of `neighbor_nets`, but instead is consulted *in addition* to `neighbor_nets`.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.29 smtp.bro

`local_mail_addr : pattern`
Email addresses matching this pattern are considered to be local. This is used to detect relaying.

`smtp_log : file`
The file into which SMTP-related alerts are written.

`smtp_sessions : table[conn_id] of smtp_session_info`
A table of `smtp_session_info` records tracking SMTP-related state for a given connection.

`process_smtp_relay : bool`
If true (default = F), processing is done to check for mail relaying.
NOTE: This variable is `const`, so may only be changed via `redef`.

```
type smtp_session_info: record {
id: count;
connection_id: conn_id;
external_orig: bool;
in_data: bool;
num_cmds: count;
num_replies: count;
cmds: smtp_cmd_info_list;
in_header: bool;
keep_current_header: bool; # a hack till MIME rewriter is ready
recipients: string;
subject: string;
content_hash: string;
num_lines_in_body: count; # lines in RFC 822 body before MIME decoding
num_bytes_in_body: count; # bytes in entity bodies after MIME decoding
content_gap: bool; # whether there is content gap in conversation

relay_1_rcpt: string; # external recipients
relay_2_from: count;  # session id of same recipient
relay_2_to: count;
relay_3_from: count;  # session id of same msg id
relay_3_to: count;
relay_4_from: count;  # session id of same content hash
relay_4_to: count;
};
```

`smtp_legal_cmds : set[string]`

The set of allowed SMTP commands (not currently used). *Fix me: Is it used somewhere?*
NOTE: This variable is `const`, so may only be changed via `redef`.

`smtp_hot_cmds : table[string] of pattern`

If an SMTP command matching an index into the table has an argument matching the associated pattern, then the request and its reply are logged.
NOTE: This variable is `const`, so may only be changed via `redef`.

`smtp_sensitive_cmds : set[string]`

If an SMTP command is in this set, the request and its reply are logged.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.30 smtp-relay.bro

`relay_log : file`

Alerts related to email relaying go in this file.

`smtp_relay_table : table[count] of smtp_session_info`

A table indexed by SMTP session ID (session$id) that keeps track of each session in an `smtp_session_info` record.

`smtp_session_by_recipient : table[string] of smtp_session_info`

A table indexed by the recipient that holds the corresponding `smtp_session_info` record.

`smtp_session_by_message_id : table[string] of smtp_session_info`

A table indexed by the email message ID that holds the corresponding `smtp_session_info` record.

`smtp_session_by_content_hash : table[string] of smtp_session_info`

A table indexed by the MD5 hash of the message that holds the corresponding `smtp_session_info` record. *Fix me: Currently unimplemented?*

6.1.31 software.bro

`software_file : file`

Alerts related to host software detection go in this file.

`software_table : table[addr] of software_set`

A table of the software running on each host. A `software_set` is itself a table, indexed by the name of the software, of `software` records.

`software_ident_by_major : set[string]`

Software names in this set could be installed twice on the same machine with different major version numbers. Such software is identified as “Software-N” where N is the major version number, to disambiguate the two.

6.1.32 ssh.bro

`ssh_log : file`

Alerts related to ssh connections go in this file.

`did_ssh_version : table[addr, bool] of count`

Indexed by host IP and (T for client, F for server), the table tracks if we have recorded the SSH version. Values of one and greater are essentially equivalent.

6.1.33 stepping.bro

`step_log : file`

Alerts related to stepping-stone detection go in this file.

`display_pairs : table[addr, string] of connection`

If ¡conn¿ was a login to ¡dst¿ propagating a $DISPLAY of ¡display¿, then we make an entry of [¡dst¿, ¡display¿] = ¡conn¿.

`tag_to_conn_map : table[string] of connection`

Maps login tags like ”Last login ...” to connections.

`conn_tag_info : table[conn_id] of tag_info`

A table, indexed by connection ID, of the `tag_info` related to it. Roughly, “tag info” consists of login strings like “Last login” and `$DISPLAY` variables. Since this information can stay constant across stepping stones, it is used to detect them.

`detected_stones : table[addr, port, addr, port, addr, port, addr, port] of count`

Indexed by two pairs of connections: (addr,port)-¿(addr,port) and (addr,port)-¿(addr,port) that have been detected to be multiple links in a stepping stone chain. The table value is the “score” of the pair of connections; the higher the score, the more likely it is to be a real stepping stone pair. More points are assigned for a timing-based correlation than, say, a `$DISPLAY`-based correlation.

`did_stone_summary : table[addr, port, addr, port, addr, port, addr, port] of count`

Basically tracks which suspected stepping stone connection pairs have had alerts generated for them. See `detected_stones` for the indexing scheme.

`stp_delta : interval`

NOTE: This variable is `const`, so may only be changed via `redef`.

`stp_idle_min : interval`

NOTE: This variable is `const`, so may only be changed via `redef`.

`stp_ratio_thresh : double`

For timing correlations, the proportion of idle times that must match up for the correlation to be considered significant.

NOTE: This variable is `const`, so may only be changed via `redef`.

`stp_scale : double`

NOTE: This variable is `const`, so may only be changed via `redef`.

`stp_common_host_thresh : count`

NOTE: This variable is `const`, so may only be changed via `redef`.

`stp_random_pair_thresh : count`

NOTE: This variable is `const`, so may only be changed via `redef`.

`stp_demux_disabled : count`

NOTE: This variable is `const`, so may only be changed via `redef`.

`skip_clear_ssh_reports : set[addr, string]`

NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.34 tftp.bro

`tftp_alert_count : table[addr] of count`
Keeps track of the number of observed outbound TFTP connections from each host.

6.1.35 udp.bro

`udp_req_count : table[conn_id] of count`
Keeps track of the number of UDP requests sent over each connection.

`udp_rep_count : table[conn_id] of count`
Fix me: not really sure

`udp_did_summary : table[conn_id] of count`
Keeps track of which connections have been summarized/recorded *Fix me: what is it really? do people use this?*

6.1.36 weird.bro

`weird_log : file`
Alerts related to `weird` (unexpected or inconsistent) traffic go in this file.

`weird_action : table[string] of WeirdAction`
A table of what to do (a `WeirdAction`) when faced with a particular "weird" scenario (the index). Example include logging to the special "weird" file or ignoring the condition.

`weird_action_filters : table[string] of function(c: connection): WeirdAction`
If an entry exists in this table for a given weird situation, then the corresponding entry is used to determine what action to take; the default is to look in `weird_action`.

`weird_ignore_host : set[addr, string]`
(host, weird condition) pairs in this set are ignored for the purposes of reporting.
NOTE: This variable is `const`, so may only be changed via `redef`.

`weird_do_not_ignore_repeats : set[string]`
The included alert conditions are reported even if they are repeated.
NOTE: This variable is `const`, so may only be changed via `redef`.

6.1.37 worm.bro

`worm_log : file`
The file into which worm-detection-related alerts are written.

`worm_list : table[addr] of count`
A table of infected hosts, indexed by the infected hosts' addresses. The value is how many times the instance has been seen sending packets.

`worm_type_list : table[addr, string] of count`
A table of infected hosts, indexed by host address and type of worm. The value is how many times that particular worm has been seen on the host.

6.1.38 Uncategorized

Fix me: These need categorization.

`bro_log_file : file`

Used to record the messages logged by `log` statements.

Default: *stderr*, unless you `@load` the `log` analyzer; see § 7.9, page 115 for further discussion.

`capture_filter : string`

Specifies what packets Bro's filter should record (§ 7.1.2, page 93).

`direct_login_prompts : set[string]`

Strings that when seen in a login dialog indicate that the user will be directly logged in after entering their username, without requiring a password (§ 7.19.2, page 134).

`discarder_maxlen : int`

The maximum amount of data that Bro should pass to a TCP or UDP *discarder* (§ 10.25, page 178).

Default: 128 bytes.

`done_with_network : bool`

Set to true when Bro is done reading from the network (or from the save files being played back, per § 10.24, page 178). The variable is set by a handler for `net_done`.

Default: initially set to false.

`interfaces : string`

A blank-separated list of network interfaces from which Bro should read network traffic. Bro merges packets from the interfaces according to their timestamps. *Deficiency: All interfaces* must *have the same link layer type.*

If empty, then Bro does not read any network traffic, unless one or more interfaces are specified using the `-i` flag.

Note: `interfaces` *has an* `&add_func` *that allows you to add interfaces to the list simply using a* `+=` *initialization (§ 10.5, page 176).*

Default: empty.

`max_timer_expires : count`

Sets an upper limit on how many pending timers Bro will expire per newly arriving packet. If set to 0, then Bro expires all pending timers whose time has come or past. This variable trades off timer accuracy and memory requirements (because a number of Bro's internal timers relate to expiring state) with potentially bursty load spikes due to a lot of timers expiring at the same time, which can trigger the watchdog, if active.

`restrict_filter : string`

Restricts what packets Bro's filter should record (§ 7.1.2, page 93).

6.2 Predefined Functions

Bro provides a number of built-in functions:

`active_connection (id: conn_id): bool`

Returns true if the given connection identifier (originator/responder addresses and ports) corresponds to a currently-active connection.

`active_file (f: file): bool`

Returns true if the given `file` is open.

`add_interface (iold: string, inew: string): string`

Used to refine the initialization of `interfaces`. Meant for internal use, and as an example of refinement (§ 10.5, page 176).

`add_tcpdump_filter (fold: string, fnew: string): string`

Used to refine the initializations of `capture_filter` and `restrict_filter`. Meant for internal use, and as an example of refinement (§ 10.5, page 176).

`log_hook (msg: string): bool`

If you define this function, then Bro will call it with each string it is about to log. The function should return true if Bro should go ahead and log the message, false otherwise. See § 7.9, page 115 for further discussion and an example.

`byte_len (s: string): count`

Returns the number of bytes in the given string. This includes any embedded NULs, and also a trailing NUL, if any (which is why the function isn't called *strlen*; to remind the user that Bro strings can include NULs).

`cat (args: any): string`

Returns the concatenation of the string representation of its arguments, which can be of any type. For example, `cat("foo", 3, T)` returns `"foo3T"`.

`clean (s: string): string`

Returns a cleaned up version of `s`, meaning that:

- embedded NULs become the text "`\0`"
- embedded DELs (delete characters) become the text "`^?`"
- ASCII "control" characters with code $\leq$ 26 become the text "`^`*Letter*", where *Letter* is the corresponding (upper case) control character; for example, ASCII 2 becomes "`^B`"
- ASCII "control" characters with codes between 26 and 32 (non-inclusive) become the text "`\x`*hex-code*"; for example, ASCII 31 becomes "`\x1f`"
- if the string does not yet have a trailing NUL, one is added.

`close (f: file): bool`

Flushes any buffered output for the given file and closes it. Returns true if the file was open, false if already closed or never opened.

`connection_record (id: conn_id): connection`

Returns the `connection` record corresponding to the given connection identifier. *Note: If the connection does not exist, then exits with a fatal run-time error.*

Deficiency: If Bro had an exception mechanism, then we could avoid the fatal run-time error, and likewise could get rid of `active_connection`.

`contains_string (big: string, little: string): bool`

Returns true if the string `little` occurs somewhere within `big`, false otherwise.

`current_time (): time`

Returns the current clock time. You will usually instead want to use `network_time`.

`discarder_check_icmp (i: ip_hdr, ih: icmp_hdr): bool`

Not documented.

`discarder_check_ip (i: ip_hdr): bool`

Not documented.

`discarder_check_tcp (i: ip_hdr, t: tcp_hdr, d: string): bool`

Not documented.

`discarder_check_udp (i: ip_hdr, u: udp_hdr, d: string): bool`

Not documented.

`edit (s: string, edit_char: string): string`

Returns a version of `s` assuming that `edit_char` is the "backspace" character (usually `"\x08"` for backspace or `"\x7f"` for DEL). For example, `edit("hello there", "e")` returns `"llo t"`.

`edit_char` must be a string of exactly one character, or Bro generates a run-time error and uses the first character in the string.

Deficiency: To do a proper job, `edit` *should also know about delete-word and delete-line editing; and it would be very convenient if it could do multiple types of edits all in one shot, rather than requiring separate invocations.*

`exit (): int`

Exits Bro with a status of 0.

Deficiency: This function should probably allow you to specify the exit status.

Note: If you invoke this function, then the usual cleanup functions `net_done` *and* `bro_done` *are* not *invoked. There probably should be an additional "*`shutdown`*" function that provides for cleaner termination.*

`flush_all (): bool`

Flushes all open files to disk.

`fmt (args: any): string`

Performs *sprintf*-style formatting. The first argument gives the format specifier to which the remaining arguments are formatted, left-to-right. As with *sprintf*, the format for each argument is introduced using "%", and formats beginning with a positive integer m specify that the given field should have a width of m characters. Fields with fewer characters are right-padded with blanks up to this width.

A format specifier of "`.`n" (coming after m, if present) instructs `fmt` to use a precision of n digits. You can only specify a precision for the `e`, `f` or `g` formats. (`fmt` generates a run-time error if either m or n exceeds 127.)

The different format specifiers are:

`%` A literal percent-sign character.

D

Format as a date. Valid only for values of type `time`.

The exact format is *yy–mm–dd–hh*:*mm*:*ss* for the local time zone, per *strftime*.

d

Format as an integer. Valid for types `bool`, `count`, `int`, `port`, `addr`, and `net`, with the latter three being converted from network order to host order prior to formatting. `bool` values of true format as the number 1, and false as 0.

e, f, g Format as a floating point value. Valid for types `double`, `time`, and `interval`. The formatting is the same as for *printf*, including the field width m and precision n.

Given no arguments, `fmt` returns an empty string.

Given a non-string first argument, `fmt` returns the concatenation of all its arguments, per `cat`.

Finally, given the wrong number of additional arguments for the given format specifier, `fmt` generates a run-time error.

`get_login_state (c: conn_id): count`

Returns the state of the given login (Telnet or Rlogin) connection, one of:

`LOGIN_STATE_AUTHENTICATE`

The connection is in its initial authentication dialog.

`LOGIN_STATE_LOGGED_IN`

The analyzer believes the user has successfully authenticated.

`LOGIN_STATE_SKIP`

The analyzer has skipped any further processing of the connection.

`LOGIN_STATE_CONFUSED`

The analyzer has concluded that it does not correctly know the state of the connection, and/or the username associated with it (§ 7.19.1, page 131).

or a run-time error and a value of `LOGIN_STATE_AUTHENTICATE` if the connection is not a login connection.

`get_orig_seq (c: conn_id): count`

Returns the highest sequence number sent by a connection's originator, or 0 if there's no such TCP connection. Sequence numbers are absolute (i.e., they reflect the values seen directly in packet headers; they are not relative to the beginning of the connection).

`get_resp_seq (c: conn_id): count`

Returns the highest sequence number sent by a connection's responder, or 0 if there's no such TCP connection.

`getenv (var: string): string`

Looks up the given environment variable and returns its value, or an empty string if it is not defined.

`is_tcp_port (p: port): bool`

Returns true if the given `port` value corresponds to a TCP port, false otherwise (i.e., it belongs to a UDP port).

`length (args: any): count`

Returns the number of elements in its argument, which must be of type `table` or `set`. If not exactly one argument is specified, or if the argument is not a table or a set, then generates a run-time message and returns 0.

Deficiency: If Bro had a union type, then we could get rid of the magic “`args: any`*” specification and catch parameter mismatches at compile-time instead of run-time.*

`log_file_name (tag: string): string`

Returns a name for a log file (such as `weird` or `red`) in a standard form. The form depends on whether `$BRO_ID` is set. If so, then the format is “<*tag*>`.<\$BRO_ID>`”. Otherwise, it is simply `tag`.

`mask_addr (a: addr, top_bits_to_keep: count): addr`

Returns the address `a` masked down to the number of upper bits indicated by `top_bits_to_keep`, which must be greater than 0 and less than 33. For example, `mask_addr(1.2.3.4, 18)` returns `1.2.0.0`, and `mask_addr(1.2.255.4, 18)` returns `1.2.192.0`.

Compare with `to_net`.

`max_count (a: count, b: count): count`

Returns the larger of `a` or `b`.

`max_double (a: double, b: double): double`

Returns the larger of `a` or `b`.

`max_interval (a: interval, b: interval): interval`

Returns the larger of `a` or `b`.

Deficiency: If Bro supported polymorphic functions, then this function could be merged with its predecessors, gaining simplicity and clarity.

`min_count (a: count, b: count): count`

Returns the smaller of `a` or `b`.

`min_double (a: double, b: double): double`

Returns the smaller of `a` or `b`.

`min_interval (a: interval, b: interval): interval`

Returns the smaller of `a` or `b`.

Deficiency: If Bro supported polymorphic functions, then this function could be merged with its predecessors, gaining simplicity and clarity.

`mkdir (f: string): bool`

Creates a directory with the given name, if it does not already exist. Returns true upon success, false (with a run-time message) if unsuccessful.

`network_time (): time`

Returns the timestamp of the most recently read packet, whether read from a live network interface or from a save file (§ 10.24, page 178). Compare against `current_time`. In general, you should use `network_time`

Direction	Meaning
`CONTENTS_NONE`	Stop recording the connection's contents.
`CONTENTS_ORIG`	Record the data sent by the connection originator (often the client).
`CONTENTS_RESP`	Record the data sent by the connection responder (often the server).
`CONTENTS_BOTH`	Record the data sent in both directions.

Table 6.1: Different types of directions for `set_contents_file`.

unless you're using Bro for non-networking uses (such as general scripting; not particularly recommended), because otherwise your script may behave very differently on live traffic versus played-back traffic from a save file.

`open (f: string): file`
Opens the given filename for write access. Creates the file if it does not already exist. Generates a run-time error if the file cannot be opened/created.

`open_for_append (f: string): file`
Opens the given filename for append access. Creates the file if it does not already exist. Generates a run-time error if the file cannot be opened/created.

`open_log_file (tag: string): file`
Opens a log file associated with the given tag, using a filename format as returned by `log_file_name`.

`parse_ftp_pasv (s: string): ftp_port`
Parses the server's reply to an FTP `PASV` command to extract the IP address and port number indicated by the server. The values are returned in an `ftp_port` record, which has three fields: `h`, the address (h is mnemonic for *host*); `p`, the (TCP) port; and `valid`, a boolean that is true if the server's reply was in the required format, false if not, or if any of the individual values (or the indicated port number) are out of range.

`parse_ftp_port (s: string): ftp_port`
Parses the argument included in a client's FTP `PORT` request to extract the IP address and port number indicated by the server. The values are returned in an `ftp_port` record, which has three fields, as indicated in the discussion of `parse_ftp_pasv`.

`reading_live_traffic (): bool`
Returns true if Bro was invoked to read live network traffic (from one or more network interfaces, per § 2.1.4, page 16), false if it's reading from save files being played back (§ 10.24, page 178).

Note: This function returns true even after Bro has stopped reading network traffic, for example due to receiving a termination signal (§ 7.2, page 95).

`set_buf (f: file, buffered: bool)`
Specifies that writing to the given file should either be fully buffered (if `buffered` is true), or line-buffered (if false). Does not return a value.

`set_contents_file (c: conn_id, direction: count, f: file): bool`
Specifies that the traffic stream of the given connection in the given direction should be recorded to the given file. `direction` is one of the values given in Table 6.2.

Note: `CONTENTS_BOTH` *results in the two directions being intermixed in the file, in the order the data was seen by Bro.*

Note: The data recorded to the file reflects the byte stream, not the contents of individual packets. Reordering and duplicates are removed. If any data is missing, the recording stops at the missing data; see `ack_above_hole` *for how this can happen.*

Deficiency: Bro begins recording the traffic stream starting with new traffic it sees. Experience has shown it would be highly handy if Bro could record the entire connection to the file, including previously seen traffic. In principle, this is possible if Bro is recording the traffic to a save file (§ 10.2, page 176), which a separate utility program could then read to extract the stream.

Returns true upon success, false upon an error.

`set_login_state (c: conn_id, new_state: count): bool`

Manually sets the state of the given login (Telnet or Rlogin) connection to `new_state`, which should be one of the values described in `get_login_state`.

Generates a run-time error and returns false if the connection is not a login connection. Otherwise, returns true.

`set_record_packets (c: conn_id, do_record: bool): bool`

Controls whether Bro should or should not record the packets corresponding to the given connection to the save file (§ 10.2, page 176), if any.

Returns true upon success, false upon an error.

`skip_further_processing (c: conn_id): bool`

Informs bro that it should skip any further processing of the contents of the given connection. In particular, Bro will refrain from reassembling the TCP byte stream and from generating events relating to any analyzers that have been processing the connection. Bro will still generate connection-oriented events such as `connection_finished`.

This function provides a way to shed some load in order to reduce the computational burden placed on the monitor.

Returns true upon success, false upon an error.

`sub_bytes (s: string, start: count, n: count): string`

Returns a copy of `n` bytes from the given string, starting at position `start`. The beginning of a string corresponds to position 1.

If `start` is 0 or exceeds the length of the string, returns an empty string.

If the string does not have `n` characters from `start` to its end, then returns the characters from `start` to the end.

`system (s: string): int`

Runs the given string as a *sh* command (via C's *system* call).

Note: The command is run in the background *with* stdout *redirected to* stderr.

Returns the return value from the *system* call. *Note: This corresponds to the status of backgrounding the given command,* not *to the exit status of the command itself.* A value of 127 corresponds to a failure to execute *sh*, and -1 to an internal system failure.

`to_lower (s: string): string`

Returns a copy of the given string with the uppercase letters (as indicated by *isascii* and *isupper*) folded to lowercase (via *tolower*).

`to_net (a: addr): net`

Returns the network prefix historically associated with the given address. That is, if `a`'s leading octet is less than 128, then returns <*a*>*/8*; if between 128 and 191, inclusive, then <*a*>*/16*; if between 192 and 223, then <*a*>*/24*; and, otherwise, <*a*>*/32*. See the discussion of the `net` type for more about network prefixes.

Generates a run-time error and returns `0.0.0.0` if the address is IPv6.

Note: Such network prefixes have become obsolete with the advent of CIDR; still, for some sites they prove useful because they correspond to existing address allocations.

Compare with `mask_addr`.

`to_upper (s: string): string`

Returns a copy of the given string with the lowercase letters (as indicated by *isascii* and *islower*) folded to uppercase (via *toupper*).

6.2.1 Run-time errors for non-existing connections

Note that for all functions that take a `conn_id` argument except `active_connection`, Bro generates a run-time error and returns false if the given connection does not exist.

6.2.2 Run-time errors for strings with NULs

While Bro allows NULs embedded within strings (§ 3.5.1, page 24), for many of the predefined functions, their presence spells trouble, particularly when the string is being passed to a C run-time function. The same holds for strings that are *not* NUL-terminated. Because Bro string constants and values returned by Bro functions that construct strings such as `fmt` and `cat` are all NUL-terminated, such strings will not ordinarily arise; but their presence could indicate an attacker attempting to manipulate either a TCP endpoint, or the monitor itself, into misinterpreting a string they're sending.

In general, any of the functions above that are passed a string argument will check for the presence of an embedded NUL or the lack of a terminating NUL. If either occurs, they generate a run-time message, and the string is transformed into the value `"<string-with-NUL>"`.

There are three exceptions: `clean`, `byte_len`, and `sub_bytes`. These functions do not complain about embedded NULs or lack of trailing NULs.

6.2.3 Functions for manipulating strings

Fix me: Missing

6.2.4 Functions for manipulating time

Fix me: Missing

Chapter 7

Analyzers and Events

In this chapter we detail the different analyzers that Bro provides. Some analyzers look at traffic in fairly generic terms, such as at the level of TCP or UDP connections. Others delve into the specifics of a particular application that is carried on top of TCP or UDP.

As we use the term here, *analyzer* primarily refers to Bro's event engine. We use the term *script* to refer to a set of event handlers (and related functions and variables) written in the Bro language; *module* to refer to a script that serves primarily to provide utility (helper) functions and variables, rather than event handlers; and *handler* to denote an event handler written in the Bro language. Furthermore, the *standard script* is the script that comes with the Bro distribution for handling the events generated by a particular analyzer.

Note: However, we also sometimes use analyzer *to refer to the event handler that processes events generated by the event engine.*

We characterize the analyzers in terms of *what* events they generate, but don't here go into the details of *how* they generate the events (i.e., the nitty gritty C++ implementations of the analyzers).

7.1 Activating an Analyzer

In general, Bro will only do the work associated with a particular analyzer if your policy script defines one or more event handlers associated with the analyzer. For example, Bro will instantiate an FTP analyzer only if your script defines an `ftp_request` or `ftp_reply` handler. If it doesn't, then when a new FTP connection begins, Bro will only instantiate a generic TCP analyzer for it. This is an important point, because some analyzers can require Bro to capture a large volume of traffic (§ 7.1.2, page 93) and perform a lot of computation; therefore, you need to have a way to trade off between the type of analysis you do and the performance requirements it entails, so you can strike the best balance for your particular monitoring needs.

Deficiency: While Bro attempts to instantiate an analyzer if you define a handler for any *of the events the analyzer generates, its method for doing so is incomplete: if you only define an analyzer's less mainstream handlers, Bro may fail to instantiate the analyzer.*

7.1.1 Loading Analyzers

The simplest way to use an analyzer is to `@load` the standard script associated with the analyzer. (See § 10.14, page 177 for a discussion of `@load`.) However, there's nothing magic about these scripts; you can freely modify or write your own. The only caveat is that some scripts @load other scripts, so the original version may wind up being loaded even though you've also written your own version. *Deficiency: It would be useful to have a mechanism to fully override one script with another.*

In this chapter we discuss each of the standard scripts as we discuss their associated analyzers.

7.1.2 Filtering

Most analyzers require Bro to capture a particular type of network traffic. These traffic flows can vary immensely in volume, so different analyzers can cost greatly differing amounts in terms of performance.

Bro predefines two redefinable string variables that have special interpretations with regard to filtering. (See § 10.5, page 176 for a discussion of redefinable variables.) capture_filter is a tcpdump filter that tells Bro what traffic it should capture. restrict_filter *limits* what traffic Bro captures. The tcpdump filter Bro uses is:

```
(capture_filter) and (restrict_filter)
```

So, for example, if you specify:

```
redef capture_filter = "port http";
redef restrict_filter = "net 128.3";
```

then the corresponding tcpdump filter will be:

```
(port http) and (net 128.3)
```

which will capture all TCP port 80 traffic that has either a source or destination address belonging to the 128.3 network (i.e., 128.3/16).

If you do not define capture_filter, then its value is set to "tcp or udp"; if you do not define restrict_filter, then no restriction is in effect.

You may have noticed that other than their default values, the definitions of capture_filter and restrict_filter are symmetric. They differ only in the convention of how they are used. Usually, you either don't define a value for restrict_filter at all, or define it just once, using it to specify a restriction that holds across your environment. For example, either to confine packet capture to a subset of the traffic (like the "net 128.3" example above), or to exclude a particular traffic source ("not host syn-flood.magnet.com") or both of these ("net 128.3 and not host syn-flood.magnet.com").

For capture_filter, on the other hand, you usually don't define a single value, but instead *refine* it one or more times using the += initializer. (See § 10.5, page 176 for a discussion of refining a variable's initial value.) The way capture_filter's refinement is defined, it constructs a filter that is the "or" of each of its refinements. So, for example, if at one point in your script you have:

```
redef capture_filter += "port ftp";
```

and at another:

```
redef capture_filter += "udp port 53";
```

and at a third:

```
redef capture_filter += "len >= 512 and len <= 1024";
```

then the resulting capture_filter will be:

```
event bro_init()
    {
    if ( restrict_filter == "" && capture_filter == "" )
        print "tcp or not tcp";  # Capture everything.

    else if ( restrict_filter == "" )
        print capture_filter;

    else if ( capture_filter == "" )
        print restrict_filter;

    else
        print fmt("(%s) and (%s)", capture_filter, restrict_filter);

    exit();
    }
```

Figure 7.1: `print-filter` prints out the `tcpdump` filter your Bro script would use and then exits.

```
    (port ftp) or (udp port 53) or (len >= 512 and len <= 1024)
```

(except there will be more parentheses, which don't actually affect the interpretation of the filter; see § 5.1.6, page 56 for the details of how the refinement is done, and why it leads to the extra parentheses).

`restrict_filter` has the same refinement mechanism, the "or"ing together of the different refinement additions, though, as mentioned above, it is not usually refined.

As you add analyzers, the final `tcpdump` filter can become quite complicated. You can use the predefined `print-filter` script shown in Figure 7.1.2 to print out the filter. This script handles the `bro_init` event and exits after printing the filter. Its intended use is that you can add it to the Bro command line ("`bro` *`my-own-script`* `print-filter`") when you want to see what filter the script *my-own-script* winds up using.

There are two particular uses for `print-filter`. The first is to debug filtering problems. Unfortunately, Bro sometimes uses sufficiently complicated expressions that they tickle bugs in `tcpdump`'s optimizer. You can take the filter printed out for your script and try running it through `tcpdump` by hand, and then also try using `tcpdump`'s `-O` option to see if turning off the optimizer fixes the problem.

The second use is to provide a *shadow* backup to Bro: that is, a version of `tcpdump` running either on the same machine or a separate machine that uses the same network filter as Bro. While `tcpdump` can't perform any analysis of the traffic, the shadow guards against the possibility of Bro crashing, because if it does, you will still have a record of the subsequent network traffic which you can run through Bro for post-analysis.

7.2 General Processing Events

Bro provides the following events relating to its overall processing:

`bro_init ()`
generated when Bro first starts up. In particular, after Bro has initialized the network (or initialized to read from

a save file) and executed any initializations and global statements (§ 10.15, page 177), and just before Bro begins to read packets from the network input source(s).

`net_done (t: time)`
generated when Bro has finished reading from the network, due to either having exhausted reading the save file(s), or having received a terminating signal (§ 7.2, page 95). *Deficiency: This event is generated on a terminating signal even if Bro is not reading network traffic.* `t` gives the time at which network processing finished.

This event is generated *before* `bro_done`. *Note: If Bro terminates due to an invocation of* `exit`*, then this event is* not *generated.*

`bro_done ()`
generated when Bro is about to terminate, either due to having exhausted reading the save file(s), receiving a terminating signal (§ 7.2, page 95), or because Bro was run without the network input source and has finished executing any global statements (§ 10.15, page 177).

This event is generated *after* `net_done`. If you have cleanup that only needs to be done when processing network traffic, it likely is better done using `net_done`. *Note: If Bro terminates due to an invocation of* `exit`*, then this event is* not *generated.*

`bro_signal (signal: count)`
generated when Bro receives a signal. Currently, the signals Bro handles are *SIGTERM*, *SIGINT*, and *SIGHUP*.

Receiving either of the first two terminates Bro, though if Bro is in the middle of processing a set of events, it first finishes with them before shutting down. The shutdown leads to invocations of `net_done` and `bro_done`, in that order. *Deficiency: In this case, Bro fails to invoke* `bro_signal`*, clearly a bug.*

Upon receiving *SIGHUP*, Bro invokes `flush_all` (in addition to your handler, if any).

`net_stats_update (t: time, ns: net_stats)`
This event includes two arguments, `t`, the `time` at which the event was generated, and `ns`, a `net_stats` record, as defined in Figure 7.2. Regarding this second parameter, the `pkts_recvd` field gives the total number of packets accepted by the packet filter so far during this execution of Bro; `pkts_dropped` gives the total number of packets reported *dropped* by the kernel; and `interface_drops` gives the total number of packets reported by the kernel as having been dropped by the network interface.

Note: An important consideration is that, as shown by experience, the kernel's reporting of these statistics is not always accurate. In particular, the `$pkts_dropped` *statistic is sometimes missing actual packet drops, and some operating systems do not support the* `interface_drops` *statistic at all. See the* `ack_above_hole` *event for an alternate way to detect if packets are being dropped.*

7.3 Generic Connection Analysis

The `conn` analyzer performs generic connection analysis: connection start time, duration, sizes, hosts, and the like. You don't in general load `conn` directly, but instead do so implicitly by loading the `tcp`, `udp`, or `icmp` analyzers. Consequently, `conn` doesn't load a `capture_filter` value by itself, but instead uses whatever is set up by these more specific analyzers.

`conn` analyzes a number of events related to connections beginning or ending. We first describe the `connection` record data type that keeps track of the state associated with each connection (§ 7.3.1, page 96), and then we detail the

```
type net_stats: record {
    # All counts are cumulative.
    pkts_recvd: count;       # Number of packets received so far.
    pkts_dropped: count;     # Number of packets *reported* dropped.
    interface_drops: count;  # Number of drops reported by interface(s).
};
```

Figure 7.2: Definition of the `net_stats` record.

events in § 7.3.3, page 99. The main output of its analysis are one-line connection summaries, which we describe in § 7.3.6, page 101, and in § 7.3.7, page 103 we give an overview of the different callable functions provided by `conn`.

`conn` also loads three other Bro modules: the `hot` and `scan` analyzers, and the `port-name` utility module.

7.3.1 The `connection` record

A `connection` record holds the state associated with a connection, as shown in Figure 7.3.1. Its first field, `id`, is defined in terms of the `conn_id` record, which has the following fields:

`orig_h`
: The IP address of the host that originated (initiated) the connection. In "client/server" terminology, this is the "client."

`orig_p`
: The TCP or UDP port used by the connection originator (client). For ICMP "connections", it is set to 0 (§ 7.25, page 164).

`resp_h`
: The IP address of the host that responded (received) the connection. In "client/server" terminology, this is the "server."

`resp_p`
: The TCP or UDP port used by the connection responder (server). For ICMP "connections", it is set to 0 (§ 7.25, page 164).

The `orig` and `resp` fields of a `connection` record both hold `endpoint` record values, which consist of the following fields:

`size`
: How many bytes the given endpoint has transmitted so far. Note that for some types of filtering, the size will be zero until the connection terminates, because the nature of the filtering is to discard the connection's intermediary packets and only capture its start/stop packets (§ 10.8, page 177).

`state`
: The current state the endpoint is in with respect to the connection. Table 7.3.1 defines the different possible states for TCP and UDP connections. *Deficiency: The states are currently defined as* `count`*, but should instead be an enumerated type; but Bro does not yet support enumerated types.*

: *Note: UDP "connections" do not have a well-defined structure, so the states for them are quite simplistic. See § 7.3.2, page 98 for further discussion.*

```
type conn_id: record {
    orig_h: addr;  # Address of originating host.
    orig_p: port;  # Port used by originator.
    resp_h: addr;  # Address of responding host.
    resp_p: port;  # Port used by responder.
};

type endpoint: record {
    size: count;  # Bytes sent by this endpoint so far.
    state: count; # The endpoint’s current state.
};

type connection: record {
    id: conn_id;         # Originator/responder addresses/ports.
    orig: endpoint;      # Endpoint info for originator.
    resp: endpoint;      # Endpoint info for responder.
    start_time: time;    # When the connection began.
    duration: interval;  # How long it was active (or has been so far).
    service: string;     # The service we associate with it (e.g., "http").
    addl: string;        # Additional information associated with it.
    hot: count;          # How many times we’ve marked it as sensitive.
};
```

Figure 7.3: Definition of `conn_id` and `connection` records.

State	**Meaning**
`TCP_INACTIVE`	The endpoint has not sent any traffic.
`TCP_SYN_SENT`	It has sent a SYN to initiated a connection.
`TCP_SYN_ACK_SENT`	It has sent a SYN ACK to respond to a connection request.
`TCP_PARTIAL`	The endpoint has been active, but we did not see the beginning of the connection.
`TCP_ESTABLISHED`	The two endpoints have established a connection.
`TCP_CLOSED`	The endpoint has sent a FIN in order to close its end of the connection.
`TCP_RESET`	The endpoint has sent a RST to abruptly terminate the connection.
`UDP_INACTIVE`	The endpoint has not sent any traffic.
`UDP_ACTIVE`	The endpoint has sent some traffic.

Table 7.1: TCP and UDP connection states, as stored in an `endpoint` record.

The remaining fields in a `connection` record are:

`start_time`
The time at which the first packet associated with this connection was seen.

`duration`
How long the connection lasted, or, if it is still active, how long since it began.

`service`
The name of the service associated with the connection. For example, if `$id$resp_p` is `tcp/80`, then the service will be `"http"`. Usually, this mapping is provided by the `port_names` global variable, perhaps via the `endpoint_id` function; but the service does not always directly correspond to `$id$resp_p`, which is why it's a separate field. In particular, an FTP data connection can have a `service` of `"ftp-data"` even though its `$id$resp_p` is something other than `tcp/20` (which is not consistently used by FTP servers).

If the name of the service has not yet been determined, then this field is set to an empty string.

`addl`
Additional information associated with the connection. For example, for a *login* connection, this is the username associated with the login.

Deficiency: A significant deficiency *associated with the* `addl` *field is that it is simply a* `string` *without any further structure. In practice, this has proven too restrictive. For example, we may well want to associate an unambiguous username with a* login *session,* and *also keep track of the names associated with failed login attempts. (See the* `login` *analyzer for an example of how this is implemented presently.) What's needed is a notion of* `union` *types which can then take on a variety of values in a type-safe manner.*

If no additional information is yet associated with this connection, then this field is set to an empty string.

`hot`
How many times this connection has been marked as potentially sensitive or reflecting a break-in. The default value of 0 means that so far the connection has not been regarded as "hot".

Note: Bro does not presently make fine-grained use of this field; the standard scripts log connections with a non-zero `hot` *field, and do not in general log those that do not, though there are exceptions. In particular, the* `hot` *field is* not *rigorously maintained as an indicator of trouble; it instead is used loosely as an indicator of particular types of trouble (access to sensitive hosts or usernames).*

7.3.2 Definitions of connections

Connections for TCP are well-defined, because establishing and terminating a connection plays a central part of the TCP protocol. For UDP and ICMP, however, the notion is much looser.

For UDP, a connection begins when host A sends a packet to host B for the first time, B never having sent anything to A. This transmission is termed a *request*, even if in fact the application protocol being used is not based on requests and replies. If B sends a packet back, then that packet is termed a *reply*. Each packet A or B sends is another request or reply. *Deficiency: There is presently no mechanism by which generic (non-RPC) UDP connections are terminated; Bro holds the state indefinitely. There should probably be a generic timeout for UDP connections that don't correspond to some higher-level protocol (such as RPC), and a user-accessible function to mark connections with particular timeouts.*

For ICMP, Bro likewise creates a connection the first time it sees an ICMP packet from A to B, even if B previously sent a packet to A, because that earlier packet would have been for a different *transport* connection than the ICMP itself—the ICMP will likely *refer* to that connection, but it itself is not part of the connection. For simplicity, this holds even for ICMP ECHOs and ECHO_REPLYs; if you want to pair them up, you need to do so explicitly in the policy script. *Deficiency: As with UDP, Bro does not time out ICMP connections.*

7.3.3 Generic TCP connection events

There are a number of generic events associated with TCP connections, all of which have a single `connection` record as their argument:

`new_connection`
Generated whenever state for a new (TCP) connection is instantiated.

Note: Handling this event is potentially expensive. For example, during a SYN flooding attack, every spoofed SYN packet will lead to a new `new_connection` *event.*

`connection_established`
Generated when a connection has become established, i.e., both participating endpoints have agreed to open the connection.

`connection_attempt`
Generated when the originator (client) has unsuccessfully attempted to establish a connection. "Unsuccessful" is defined as at least `ATTEMPT_INTERVAL` seconds having elapsed since the client first sent a connection establishment packet to the responder (server), where `ATTEMPT_INTERVAL` is an internal Bro variable which is presently set to 300 seconds. *Deficiency: This variable should be user-settable.* If you want to *immediately* detect that a client is attempting to connect to a server, regardless of whether it may soon succeed, then you want to handle the `new_connection` event instead.

Note: Handling this event is potentially expensive. For example, during a SYN flooding attack, every spoofed SYN packet will lead to a new `connection_attempt` *event, albeit delayed by* `ATTEMPT_INTERVAL.`

`partial_connection`
Generated when both connection endpoints enter the `TCP_PARTIAL` state (Table 7.3.1). This means that we have seen traffic generated by each endpoint, but the activity did not begin with the usual connection establishment. *Deficiency: For completeness, Bro's event engine should generate another form of* `partial_connection` *event when a single endpoint becomes active (see* `new_connection` *below). This hasn't been implemented because our experience is network traffic often contains a great deal of "crud", which would lead to a large number of these really-partial events. However, by not providing the event handler, we miss an opportunity to detect certain forms of stealth scans until they begin to elicit some form of reply.*

`connection_finished`
Generated when a connection has gracefully closed.

`connection_rejected`
Generated when a server rejects a connection attempt by a client.

Note: This event is only generated as the client attempts to establish a connection. If the server instead accepts the connection and then later aborts it, a `connection_reset` *event is generated (see below). This can happen, for example, due to use of TCP Wrappers.*

Note: Per the discussion above, a client attempting to connect to a server will result in one *of* connection_attempt, connection_established, *or* connection_rejected*; they are mutually exclusive.*

connection_half_finished

Generated when Bro sees one endpoint of a connection attempt to gracefully close the connection, but the other endpoint is in the TCP_INACTIVE state. This can happen due to *split routing* (§ 10.9, page 177), in which Bro only sees one side of a connection.

connection_reset

Generated when one endpoint of an established connection terminates the connection abruptly by sending a TCP RST packet.

connection_partial_close

Generated when a previously inactive endpoint attempts to close a connection via a normal FIN handshake or an abort RST sequence. When it sends one of these packets, Bro waits PARTIAL_CLOSE_INTERVAL (an internal Bro variable set to 10 seconds) prior to generating the event, to give the other endpoint a chance to close the connection normally.

connection_pending

Generated for each still-open connection when Bro terminates.

7.3.4 The tcp analyzer

The general tcp analyzer lets you specify that you're interested in generic connection analysis for TCP. It simply @load's conn and adds the following to capture_filter:

```
tcp[13] & 0x7 != 0
```

which instructs Bro to capture all TCP SYN, FIN and RST packets; that is, the control packets that delineate the beginning (SYN) and end (FIN) or abnormal termination (RST) of a connection.

7.3.5 The udp analyzer

The general udp analyzer lets you specify that you're interested in generic connection analysis for UDP. It @load's both hot and conn, and defines two event handlers:

udp_request (u: connection)

Invoked whenever a UDP packet is seen on the forward (request) direction of a UDP connection. See § 7.3.2, page 98 for a discussion of how Bro defines UDP connections.

The analyzer invokes check_hot with a mode of CONN_ATTEMPTED and then record_connection to generate a connection summary (necessary because Bro does not time out UDP connections, and hence cannot generate a connection-attempt-failed event).

udp_reply (u: connection)

Invoked whenever a UDP packet is seen on the reverse (reply) direction of a UDP connection. See § 7.3.2, page 98 for a discussion of how Bro defines UDP connections.

The analyzer invokes `check_hot` with a mode of `CONN_ESTABLISHED` and then again with a mode of `CONN_FINISHED` to cover the general case that the reply reflects that the connection was both established and is now complete. Finally, it invokes `record_connection` to generate a connection summary.

Note: The standard script does not *update* `capture_filter` *to capture UDP traffic. Unlike for TCP, where there is a natural generic filter that captures only a subset of the traffic, the only natural UDP filter would be simply to capture all UDP traffic, and that can often be a huge load.*

7.3.6 Connection summaries

The main output of `conn` is a one-line ASCII summary of each connection. By tradition, these summaries are written to a file with the name `red.`*tag*, where *tag* uniquely identifies the Bro session generating the logs. ("`red`" is mnemonic for "reduced," from Bro's roots in performing protocol analysis for Internet traffic studies.)

The summaries are produced by the `record_connection` function, and have the following format:

<start> <duration> <service> B_o B_r A_l A_r *<state> <flags> <addl>*

start corresponds to the connection's start time, as defined by `start_time`.

duration gives the connection's duration, as defined by `duration`.

service is the connection's service, as defined by `service`.

B_o**,** B_r give the number of bytes sent by the *originator* and *responder*, respectively. These correspond to the `size` fields of the corresponding `endpoint` records.

A_l**,** A_r correspond to the *local* and *remote* addresses that participated in the connection, respectively. The notion of which addresses are local is controlled by the `local_nets` global variable, if refined from its default value of empty. If `local_nets` has *not* been refined, then A_l is the connection *responder* and A_r is the connection *originator*.

Note: The format and defaults for A_l *and* A_r *are unintuitive; they reflect the use of Bro's predecessor for analyzing Internet traffic patterns, and have not been changed so as to maintain compatibility with old, archived connection summaries.*

state reflects the state of the connection at the time the summary was written (which is usually either when the connection terminated, or when Bro terminated). The different states are summarized in Table 7.3.6. The ASCII **Name** given in the Table is what appears in the `red` file; it is returned by the `conn_state` function. The **Symbol** is used when generating human-readable versions of the file—see `hot-report`.

For UDP connections, the analyzer reports connections for which both endpoints have been active as `SF`; those for which just the originator was active as `S0`; those for which just the responder was active as `SHR`; and those for which neither was active as `OTH` (this latter shouldn't happen!).

flags reports a set of additional binary state associated with the connection:

`L` indicates that the connection was initiated *locally*, i.e., the host corresponding to A_l initiated the connection. If `L` is missing, then the host corresponding to A_r initiated the connection.

Symbol	Name	Meaning
}	S0	Connection attempt seen, no reply.
>	S1	Connection established, not terminated.
>	SF	Normal establishment and termination. Note that this is the same symbol as for state S1. You can tell the two apart because for S1 there will not be any byte counts in the summary, while for SF there will be.
[	REJ	Connection attempt rejected.
}2	S2	Connection established and close attempt by originator seen (but no reply from responder).
}3	S3	Connection established and close attempt by responder seen (but no reply from originator).
>]	RSTO	Connection established, originator aborted (sent a RST).
>[	RSTR	Established, responder aborted.
}]	RSTOS0	Originator sent a SYN followed by a RST, we never saw a SYN ACK from the responder.
<[	RSTRH	Responder sent a SYN ACK followed by a RST, we never saw a SYN from the (purported) originator.
>h	SH	Originator sent a SYN followed by a FIN, we never saw a SYN ACK from the responder (hence the connection was “half” open).
<h	SHR	Responder sent a SYN ACK followed by a FIN, we never saw a SYN from the originator.
?>?	OTH	No SYN seen, just midstream traffic (a “partial connection” that was not later closed).

Table 7.2: Summaries of connection states, as reported in red files.

`U` indicates the connection involved one of the networks listed in the `neighbor_nets` variable. The use of "`U`" for this indication (rather than "`N`", say) is historical, as for the most part is the whole notion of "neighbor network."

Note that connection can have both `L` and `U` set (see next item).

`X` is used to indicate that *neither* the "`L`" or "`U`" flags is associated with this connection. An explicit negative indication is needed to disambiguate the *flags* field from the subsequent *addl* field.

addl lists additional information associated with the connection, i.e., as defined by `addl`.

Putting all of this together, here is an example of a `red` connection summary:

```
931803523.006848 54.3776 http 7320 38891 206.132.179.35 128.32.162.134 RSTO X %103
```

The connection began at timestamp 931803523.006848 (18:18:43 hours GMT on July 12, 1999; see the *cf* utility for how to determine this) and lasted 54.3776 seconds. The service was HTTP (presuambly; this conclusion is based just on the responder's use of port `80/tcp`). The originator sent 7,320 bytes, and the responder sent 38,891 bytes. Because the "`L`" flag is absent, the connection was initiated by host 128.32.162.134, and the responding host was 206.132.179.35. When the summary was written, the connection was in the "`RSTO`" state, i.e., after establishing the connection and transferring data, the originator had terminated it with a RST (this is unfortunately common for Web clients). The connection had neither the `L` or `U` flags associated with it, and there was additional information, summarized by the string "`%103`" (see the `http` analyzer for an explanation of this information).

7.3.7 Connection functions

We finish our discussion of generic connection analysis with a brief summary of the different Bro functions provided by the `conn` analyzer:

`conn_size (e: endpoint, is_tcp: bool): string`
returns a string giving either the number of bytes the endpoint sent during the given connection, or `"?"` if from the connection state this can't be determined. The `is_tcp` parameter is needed so that the function can inspect the endpoint's state to determine whether the connection was closed.

`conn_state (c: connection, is_tcp: bool): string`
returns the name associated with the connection's state, as given in Table 7.3.6.

`determine_service (c: connection): bool`
sets the `service` field of the given connection, using `port_names`. If you are using the `ftp` analyzer, then it knows about FTP data connections and maps them to `port_names[20/tcp]`, i.e., `"ftp-data"`.

`full_id_string (c: connection): string`
returns a string identifying the connection in one of the two following forms. If the connection is in state `S0`, `S1`, or `REJ`, then no data has been transferred,[1] and the format is:

A_o *<state>* A_r/*<service>* *<addl>*

[1] *Deficiency: Actually, for state* `S1` *data may have been transferred, and so this assumption should be corrected in that case.*

where A_o is the IP address of the originator (`$id$orig_h`), *state* is as given in the **Symbol** column of Table 7.3.6, A_r is the IP address of the responder (`$id$resp_h`), *service* gives the application service (`$service`) as set by `determine_service`, and *addl* is the contents of the `$addl` field (which may be an empty string).

Note that the ephemeral port used by the originator is not reported. If you want to display it, use `id_string`.

So, for example:

```
128.3.6.55 > 131.243.88.10/telnet "luser"
```

identifies a connection originated by `128.3.6.55` to `131.243.88.10`'s Telnet server, for which the additional associated information is `"luser"`, the username successfully used during the authentication dialog as determined by the `login` analyzer. From Table 7.3.6 we see that the connection must be in state `S1`, as that's the only state of `S0`, `S1`, or `REJ` that has a `>` symbol. (We can tell it's *not* in state `SF` because the format used for that state differs—see below.)

For connections in other states, Bro has size and duration information available, and the format returned by `full_id_string` is:

A_o S_o`b` <*state*> A_r`/`<*service*> S_r`b` D`s` <*addl*>

where A_o, A_r, *state*, *service*, and *addl* are as before, S_o and S_r give the number of bytes transmitted so far by the originator to the responder and vice versa, and D gives the duration of the connection in seconds (reported with one decimal place) so far.

An example of this second format is:

```
128.3.6.55 63b > 131.243.88.10/telnet 391b 39.1s "luser"
```

which reflects the same connection as before, but now `128.3.6.55` has transmitted 63 bytes to `131.243.88.10`, which has transmitted 391 bytes in response, and the connection has been active for 39.1 seconds. The "`>`" indicates that the connection is in state `SF`.

`id_string (id: conn_id): string`

returns a string identifying the connection by its address/port quadruple. Regardless of the connection's state, the format is:

A_o`/`P_o `>` A_r`/`P_r

where A_o and A_r are the originator and responder addresses, respectively, and P_o and P_r are representations of the originator and responder ports as returned by the `port-name` module, i.e., either "<*number*>`/<tcp` *or* `udp>`" or a string like "`http`" for a well-known port such as `80/tcp`.

An example:

```
128.3.6.55/2244 > 131.243.88.10/telnet
```

Note, `id_string` is implemented using a pair of calls to `endpoint_id`.

Deficiency: It would be convenient to have a form of `id_string` *that can incorporate a notion of directionality, for example* `128.3.6.55/2244 < 131.243.88.10/telnet` *to indicate the same connection as before, but referring specifically to the flow from responder to originator in that connection (indicated by using "*`<`*" instead of "*`>`*").*

`log_hot_conn (c: connection)`

logs a real-time alert of the form:

> hot: <*connection-id*>

where *connection-id* is the format returned by `full_id_string`. `log_hot_conn` keeps track of which connections it has logged and will not log the same connection more than once.

`record_connection (c: connection, disposition: string)`

Generates a connection summary to the `red` file in the format described in § 7.3.6, page 101. If the connection's `$hot` field is positive, then also logs the connection using `log_hot_conn`. The `disposition` is a text description of the connection's state, such as `"attempt"` or `"half_finished"`; it is not presently used.

`service_name (c: connection): string`

returns a string describing the service associated with the connection, computed as follows. If the responder port (`$id$resp_p`), p, is well-known, that is, in the `port_names` table, then p's entry in the table is returned (such as `"http"` for TCP port 80). Otherwise, for TCP connections, if the responder port is less than 1024, then `priv-`p is returned, otherwise `other-`p. For UDP connections, the corresponding service names are `upriv-`p and `uother-`p.

`terminate_connection (c: connection)`

Attempts to terminate the given connection using the `rst` utility in the current directory. It does not check to see whether the utility is actually present, so an unaesthetic shell error will appear if the utility is not available.

`rst` terminates connections by forging RST packets. It is not presently distributed with Bro, due to its potential for disruptive use.

If Bro is reading a trace file rather than live network traffic, then `terminate_connection` logs the `rst` invocation but does not actually invoke the utility. In either case, it finishes by logging that the connection is being terminated.

7.4 Site-specific information

The `site` analyzer is not actually an analyzer but simply a set of global variables (and one function) used to define a site's basic topological information.

7.4.1 Site variables

The `site` module defines the following variables, all redefinable:

`local_nets : set[net]`

Defines which `net`'s Bro should consider as reflecting a local address.

Default: empty.

`local_16_nets : set[net]`

Defines which /16 prefixes Bro should consider as reflecting a local address. *Deficiency: Bro currently is inconsistent regarding when it consults* `local_nets` *versus* `local_16_nets`*, so you should ensure that this variable and the previous one are always consistent.*

Default: empty.

`local_24_nets : set[net]`
The same, but for /24 addresses.

Default: empty.

`neighbor_nets : set[net]`
Defines which `net`'s Bro should consider as reflecting a "neighbor." Neighbors networks can be treated specially in some policies, distinct from other non-local addresses. In particular, `drop_address` will not drop connectivity to an address belonging to a neighbor.

The notion is somewhat historical, as is the use of "`U`" to mark neighbors in connection summaries (§ 7.3.6, page 101).

Default: empty.

`neighbor_16_nets : set[addr]`
Defines which /16 addresses Bro should consider as reflecting a neighbor; the only use of this variable in the standard scripts is that a scan originating from an address with one of these prefixes will not be dropped (§ 10.10, page 177). *Deficiency: The name is poorly chosen and should be changed to better reflect this use. Deficiency: In addition, this variable should be kept consistent with* `neighbor_nets`*, until the fine day when the processing is rectified to only use one variable.*

Default: empty.

`neighbor_24_nets : set[net]`
The same, but for /24 addresses.

Default: empty.

7.4.2 Site-specific functions

Currently, the `site` module only defines one function:

`is_local_addr (a: addr): bool`
returns true if the given address belongs to one of the "local" networks, false otherwise. Currently, the test is made by masking the address to /16 and /24 and comparing it to `local_16_nets` and `local_24_nets`.

7.5 The `hot` Analyzer

The standard `hot` script defines policy relating to fairly generic notions of allowed and prohibited connections. It defines a number of variables that you will need to refine to customize your site's policies. It also provides two functions for checking connections against the policies, which can be used by other of the standard scripts.

7.5.1 `hot` variables

The standard `hot` script defines the following variables, all redefinable:

`same_local_net_is_spoof : bool`

If true, then a connection with a local originator address and a local responder address is considered by `check_spoof` to have been spoofed. *Deficiency: The name is poorly chosen (and may be changed in the future) to something more accurate like* `both_local_nets_is_spoof`.

In general, you want to use true for a Bro that is monitoring Internet access links (DMZs) and false for internal monitors.

Default: `F`.

`allow_spoof_services : set[port]`

Defines a set of services (responder ports) for which Bro should not generate alerts if it sees apparent spoofed traffic.

Default: `110/tcp` (POP version 3; [RFC1939]). This default was chosen because in our experience one common form of benign spoof is an off-site laptop attempting to read mail while still configured to use its on-site address.

`allow_pairs : set[addr, addr]`

Defines pairs of source and destination addresses for which the source is allowed to connect to the destination. The intent with this variable is that the source or destination address will be a sensitive host (such as defined with `hot_srcs` or `hot_dsts`), for which this particular access should be allowed.

Default: empty.

`allow_16_net_pairs : set[addr, addr]`

Defines pairs of source and destination /16 networks for which the source is allowed to connect to the destination, similar to `allow_pairs`. *Note: The set is defined in terms of* `addr`*'s and not* `net`*'s. So, for example, rather than specifying* `128.32.`*, which is a* `net` *constant, you'd use* `128.32.0.0` *(an* `addr` *constant).*

Default: empty.

`hot_srcs : table[addr] of string`

Defines source addresses that should be considered "hot". A successfully established connection from such a source address is logged, unless one of the access exception variables such as `allow_pairs` also matches the connection. The value of the table gives an explanatory message as to why the source is hot; for example, `"known attacker site"`. *Note: This value is not currently used, though it aids in documenting the policy script.*

Default: empty.

Example: redefining `hot_srcs` using

```
redef hot_srcs: table[addr] of string = {
    [ph33r.the.eleet.com] = "script kideez",
};
```

would result in Bro alerting on any traffic coming `ph33r.the.eleet.com`.

`hot_dsts : table[addr] of string`

Same as `hot_srcs`, except for destination addresses.

Default: empty.

hot_src_24nets : table[addr] of string

Defines /24 source networks should be considered “hot,” similar to hot_srcs. *Deficiency: Other network masks, particularly /16, should be provided.*

Default: empty.

Example: redefining hot_src_24nets using

```
redef hot_src_24nets: table[addr] of string = {
    [198.81.129.0] = "CIA incoming!",
};
```

would result in Bro alerting on any traffic coming from the 198.81.129/24 network.

hot_dst_24nets : table[addr] of string

same as hot_src_24nets, except for destination networks.

Default: empty.

allow_services : set[port]

Defines a set of services that are always allowed, regardless of whether the source or destination address is “hot.”

Default: ssh, http, gopher ident, smtp, 20/tcp (FTP data).

Note: The defaults are a bit unusual. They are intended for a quite open site with many services.

allow_services_to : set[addr, port]

Defines a set of services that are always allowed if the server is the given host, regardless of whether the source or destination address is “hot.”

Default: empty.

Example: redefining allow_services_to using

```
redef allow_services_to: set[addr, port] += {
    [ns.mydomain.com, [domain, 123/tcp]],
} &redef;
```

would result in Bro not alerting on any TCP DNS or NTP traffic heading to ns.mydomain.com. You might add this if ns.mydomain.com is also in hot_dsts, because in general you want to consider any access (other than DNS or NTP) as sensitive.

allow_services_pairs : set[addr, addr, port]

Defines a set of services that are always allowed if the connection originator is the first address and the responder (server) the second address.

Default: empty.

Example: redefining allow_services_pairs using

```
redef allow_services_pairs: set[addr, addr, port] += {
    [ns2.mydomain.com, ns.mydomain.com, [domain, 123/tcp]],
} &redef;
```

would result in Bro not alerting on any TCP DNS or NTP traffic initiated from ns2.mydomain.com to ns.mydomain.com.

flag_successful_service : table[port] of string

The opposite of allow_services. Defines a set of services that should always be flagged as sensitive, even if neither the source nor the destination address is "hot." The string value in the table gives the reason for why the service is considered hot. *Note: Bro currently does not use these explanatory messages.*

Default: 31337/tcp (a popular backdoor because in stylized lettering it spells ELEET) and 2766/tcp (the Solaris listen service, in our experience rarely used legitimately in wide-area traffic).

Note: Bro can flag these services erroneously when a server happens to run a different service on the same port. For example, if you're not running the FTP analyzer, then Bro won't know that FTP data connections using ephemeral ports in fact belong to legitimate FTP traffic, and will flag any that coincide with these services. A related problem arises when a user has configured their SSH access to tunnel FTP control channels through the FTP connection, but not the corresponding data connections (so they don't pay the expense of encrypting the data transfers), so again Bro can't recognize that the ephemeral ports used for the data connections does not reflect the presumed sensitive service.

Example: redefining flag_successful_service using

```
redef flag_successful_service: table[port] of string += {
        [1524/tcp] = "popular backdoor",
};
```

would result in Bro also alerting on any successful connection to a server running on TCP port 1524.

flag_successful_inbound_service : table[port] of string

The same as flag_successful_service, except only applies to connections with a remote initiator and a local responder (determined by finding the responder address in local_nets).

Default: 1524/tcp (ingreslock, a popular backdoor because an attacker can place an entry for the backdoor in */etc/inetd.conf* using a service name rather than a raw port number, and hence more likely to appear legitimate to casual inspection). *Note: There's no compelling reason why* ingreslock *is in this table rather than the more general* flag_successful_service*, though it does tend to result in a few more false hits than the others, presumably because it's a lower port number, and hence more likely on some systems to be chosen for an ephemeral port.* ingreslock

Note: Symmetry would call for flag_successful_outbound_service*. This hasn't been implemented in Bro yet simply because the Bro development site has a threat model structured primarily around external threats.*

terminate_successful_inbound_service : table[port] of string

The same as flag_successful_inbound_service, except invokes terminate_connection in an attempt to terminate the connection.

Default: empty.

Note: As for flag_successful_inbound_service*, it would be symmetric to have* terminate_successful_outbound_service*, and also to have a more general* terminate_successful_service.

`flag_rejected_service : table[port] of string`

Similar to `flag_successful_service`, except applies to connections that a server rejects. For example, you could detect a particular, failed Linux *mountd* attack by adding `10752/tcp` to this table, since that happens to be the port used by the commonly available version of the exploit for its backdoor if the attack succeeds. *Note: You would of course likely also want to put* `10752/tcp` *in* `flag_successful_service`*; or put the entire* `flag_rejected_service` *table into* `flag_successful_service`*, as discussed in § 10.16, page 177.*

Default: none.

Deficiency: It might make sense to have `flag_attempted_service`*, which doesn't require that a server actively reject the connection, but Bro doesn't currently have this.*

7.5.2 `hot` functions

The `hot` module defines two functions for external use:

`check_spoof (c: connection): bool`

checks the originator and responder addresses of the given connection to determine if they are both local (and the connection is not explicitly allowed in `allow_spoof_services`). If so, and if `same_local_net_is_spoof` is true, then marks the connection as "hot".

The function also checks for a specific denial of service attack, the "Land" attack, in which the addresses are the same and so are the ports. If so, then it generates a `conn_weird` event with a name of `"Land_attack"`. It makes this check even if `same_local_net_is_spoof` is false.

Returns: true if the connection is now hot (or was upon entry), false otherwise.

`check_hot (c: connection, state: count): bool`

checks the given connection against the various policy variables discussed above, and bumps the connection's `hot` field if it matches the policies for being sensitive, and does not match the various exceptions. It also uses `check_spoof` to see if the connection reflects a possible spoofing attack; and terminates the connection if `terminate_successful_service` indicates so.

The caller indicates the connection's state in the second parameter to the function, using one of the values given in Table 7.5.2. As noted in the Table, the processing differs depending on the state.

In general, the pattern is to make one call when the connection is first seen, either `CONN_ATTEMPTED`, `CONN_ESTABLISHED`, or `CONN_REJECTED`. If the application is one for which connections should only be considered "established" after a successful pre-exchange between originator and responder, then a subsequent call is made with a state of `APPL_ESTABLISHED`. The idea here is to provide a way to filter out what are in fact not really successful connections so that they are not analyzed in terms of successful service. Finally, for services that don't use `APPL_ESTABLISHED`, a call is made instead when the connection finishes for some reason, using state `CONN_FINISHED`. *Note: This approach delays alerting until the connection is over, which might be later than you want, in which case you may need to edit* `check_hot` *to provide the desired functionality.*

Returns: true if the connection is now hot (or was upon entry), false otherwise.

State	Meaning	Tests
CONN_ATTEMPTED	Connection attempted, no reply seen. Note that you should also use this value for scans with indeterminant state, such as possible stealth scans. For example, connection_half_finished does this.	check_spoof.
CONN_ESTABLISHED	Connection established. Also used for connections apparently established, per partial_connection.	check_spoof, flag_successful_service, flag_successful_inbound_service, allow_services_to, terminate_successful_inbound_service.
APPL_ESTABLISHED	The connection has reached application-layer establishment. For example, for Telnet or Rlogin, this is after the user has authenticated.	allow_services_to, allow_service_pairs, allow_pairs, allow_16_net_pairs, hot_srcs, hot_dsts, hot_src_24nets, hot_dst_24nets.
CONN_FINISHED	The connection has finished, either cleanly or abnormally (for example, connection_reset).	Same as APPL_ESTABLISHED, if the connection exchanged non-zero amounts of data in both directions, and if the service wasn't one of the ones that generates APPL_ESTABLISHED.
CONN_REJECTED	The connection attempt was rejected by the server.	check_spoof, flag_rejected_service.

Table 7.3: Different connection states to use when calling check_hot.

7.6 The `scan` Analyzer

The `scan` analyzer detects connection attempts to numerous machines (address scanning), connection attempts to many different services on the same machine (port scanning), and attempts to access many different accounts (password guessing). The basic methodology is to use tables to keep track of the distinct addresses and ports to which a given host attempts to connect, and to trigger alerts when either of these reaches a specified size. *Deficiency: As currently written, the analyzer will not detect distributed scans, i.e., when many sites are used to probe individually just a few, but together a large number, of ports or addresses.*

A powerful technique that Bro potentially provides is dropping border connectivity with remote scanning sites, though you must supply the magic script to talk with your router and effect the block. See `drop_address` below for a discussion of the interface provided. *Note: Naturally, providing this capability means you might become vulnerable to denial-of-service attacks in which spoofed packets are used in an attempt to trigger a block of a site to which you want to have access.*

7.6.1 `scan` variables

In addition to internal variables for its bookkeeping, the analyzer provides the following redefinable variables:

`report_peer_scan : set[count]`

Generate a log message whenever a remote host (as determined by `is_local_addr`) has attempted to connect to the given number of distinct hosts.

Default: { `100, 1000, 10000,` }. So, for example, if a remote host attempts to connect to 3,500 different local hosts, a report will be generated when it makes the 100th attempt, and another when it makes the 1,000th attempt.

`report_outbound_peer_scan : set[count]`

The same as `report_peer_scan`, except for connections initiated locally.

Default: { `1000, 10000,` }.

`possible_port_scan_thresh : count`

Initially, port scan detection is done based on how many different ports a given host connects to, regardless of on which hosts. Once this threshold is reached, however, then the analyzer begins tracking ports accessed per-server, which is important for reducing false positives. *Note: The reason this variable exists is because it is very expensive to track per-server ports accessed for every active host; this variable limits such tracking to only active hosts contacting a significant number of different ports.*

Default: `25`.

`report_accounts_tried : set[count]`

Whenever a remote host has attempted to access a number of local accounts present in this set, generate a log message. Each distinct username/password pair is considered a different access.

Default: { `25, 100, 500,` }.

`report_remote_accounts_tried : set[count]`

The same, except for access to remote accounts rather than local ones.

Default: { `100, 500,` }.

`skip_accounts_tried : set[addr]`

Do not do bookkeeping for account attempts for the given hosts.

Default: empty.

`skip_outbound_services : set[port]`

Do not do outbound-scanning bookkeeping for connections involving the given services.

Default: `allow_services`, `ftp`, `addl_web` (see next item).

`addl_web : set[port]`

Additional ports that should be considered as Web traffic (and hence skipped for outbound-scan bookkeeping).

Default: { `81/tcp, 443/tcp, 8000/tcp, 8001/tcp, 8080/tcp,` }.

`skip_scan_sources : set[addr]`

Hosts that are allowed to address-scan without complaint.

Default: `scooter.pa-x.dec.com`, `scooter2.av.pa-x.dec.com` (AltaVista crawlers; you get the idea.)

`skip_scan_nets_24 : set[addr, port]`

/24 networks that are allowed to address scan for the given port without complaint.

Default: empty.

`can_drop_connectivity : bool`

True if the Bro has the capability of dropping connectivity, per `drop_address`.

Default: false.

`shut_down_scans : set[port]`

Scans of these ports trigger connectivity-dropping (if the Bro is capable of dropping connectivity), unless `shut_down_all_scans` is defined (next item).

Default: empty.

`shut_down_all_scans : bool`

Ignore `shut_down_scans` and simply drop all scans regardless of service.

Default: false.

`shut_down_thresh : count`

Shut down connectivity after a host has scanned this many addresses.‘

Default: `100`.

`never_shut_down : set[addr]`

Purported scans from these addresses are never shut down.

Default: the root name servers (`a.root-servers.net` through `m.root-servers.net`).

7.6.2 `scan` functions

The standard `scan` script provides the following functions:

`drop_address (a: addr, msg: string)`
Drops external connectivity to the given address and logs a notification using the given message.

Dropping connectivity requires all of the following to be true:

- `can_drop_connectivity` is true.
- The address is neither local nor a neighbor (§ 7.4.1, page 106).
- The address is not in `never_shut_down`.

If these checks succeed, then the script simply attempts to invoke a shell script *drop-connectivity* with a single argument, the IP address to block. It is up to you to provide the script, using whatever interface to your router/firewall you have available.

The function does not return a value.

`check_scan (c: connection, established: bool, reverse: bool): bool`
Updates the analyzer's internal bookkeeping on the basis of the new connection `c`. If `established` is true, then the connection was successfully established, otherwise not. If `reverse` is true, then the function should consider the originator/responder fields in the connection's record as reversed. *Note: This last is needed for some unusual new connections that may reflect stealth scanning. For example, when the event engine sees a SYN-ack without a corresponding SYN, it instantiates a new connection with an assumption that the SYN-ack came from the responder (and it missed the initial SYN either due to split routing (§ 10.9, page 177), a packet drop (§ 10.13, page 177), or Bro having started running after the initial SYN was sent).*

If the originating host's activity matches the policy defined by the variables above, then the analyzer logs this fact, and possibly attempts to drop connectivity to the originating host. The function also schedules an event for 24 hours in the future (or when Bro terminates) to generate a summary of the scanning activity (so if the host continues scanning, you get a report on how many hosts it wound up scanning). *Deficiency: This time interval should be selectable.*

Note: Purported scans of the FTP data port (`20/tcp`) or the `ident` service (`113/tcp`) are never reported or dropped, as experience has shown they yield too many false hits.

The function does not return a value.

7.6.3 `scan` event handlers

The standard `scan` script defines one event handler:

`account_tried (c: connection, user: string, passwd: string)`
The given connection made an attempt to access the given username and password. Each distinct username/password pair is considered a new access. The event handler generates a log message if the access matches the logging policy outlined above.

Note: `account_tried` *events are generated by* `login` *and* `ftp` *analyzers.*

7.7 The port-name Module

The port-name utility module provides one redefinable variable and one callable function:

port_names : table[port] of string
: Maps TCP/UDP ports to names for the services associated with those ports. For example, 80/tcp maps to "http". These names are used by the conn analyzer when generating connection logs (§ 7.3, page 95).

endpoint_id (h: addr, p: port): string
: Returns a printable form of the given address/port connection endpoint. The format is either <*address*>/<*service-name*> or <*address*>/<*port-number*> depending on whether the port appears in port_names.

7.8 The mt Module

The mt module is intended to provide a convenient way to run (almost) all of the analyzers. It @load's the following other modules and analyzers: log, dns, hot, port-name, frag, tcp, scan, weird, finger, ident, ftp, login and portmapper. So you can run Bro using *bro -i in0 mt* to have it analyze traffic on interface *in0* using the above analyzers (§ 2.1.4, page 16); or you can @load mt to load in the above analyzers.

Note: The mt *analyzer doesn't load* http *(because it can prove a very high load for many sites) nor experimental analyzers such as* stepping *or* backdoor.

7.9 The log Module

The log utility module redefines a single variable:

bro_log_file : file
: A special Bro variable used internally to specify a file where Bro should record messages logged by log statements (as well as generating real-time alerts via *syslog*).

  Default: if the $BRO_ID environment variable is defined, then log.<*$BRO_ID*>, otherwise bro.log.

  Note: This value is slightly different than that returned by open_log_file*, because the latter would return* log *if* $BRO_ID *wasn't defined, and that name seems too easy to confuse with other uses.*

  See § 6.1.38, page 84 for further discussion.

If you do not include this module, then Bro records log messages to *stderr*.

You can also control Bro's log processing by defining the special function log_hook. It takes a single argument, msg: string, the message in a just-executed log statement, and returns a boolean value: true if Bro should indeed log the message, false if not. For example, Figure 7.9 shows a definition of log_hook that checks each log message to see whether the same text has been logged before. It only logs the first instance of a message. If a message appears at least five times, then it schedules a future log_summary event for 5 minutes in the future; the purpose of this event is to summarize the total number of times the message has appeared at that point in time.

```
global msg_count: table[string] of count &default = 0;

event log_summary(msg: string)
    {
    log fmt("(%s) %d times", msg, msg_count[msg]);
    }

function log_hook(msg: string): bool
    {
    if ( ++msg_count[msg] == 1 )
        # First time we’ve seen this message - log it.
        return T;

    if ( msg_count[msg] == 5 )
        # We’ve seen it five times, enough to be worth
        # summarizing.  Do so five minutes from now,
        # for whatever total we’ve seen by then.
        schedule +5 min { log_summary(msg) };

    return F;
    }
```

Figure 7.4: Sample definition of log_hook

7.10 The `active` Module

The `active` utility module provides a single, non-redefinable variable that holds information about active connections:

`active_conn : table[conn_id] of connection`

Indexed by a `conn_id` (Figure 7.3.1) giving the originator/responder addresses/ports, returns the connection's `connection` record. As usual, accessing the table with a non-existing index results in a run-time error, so you should first test for the presence of the index using the `in` operator.

Default: empty.

This functionality is quite similar to that of the `active_connection` function, and *Deficiency: arguably this module should be removed in favor of the function*. It does, however, provide a useful example of maintaining bookkeeping by defining additional handlers for events that already have handlers elsewhere.

7.11 The `demux` Module

The `demux` utility module provides a single function:

`demux_conn (id: conn_id, tag: string, otag: string, rtag: string): bool`

Instructs Bro to write ("demultiplex") the contents of the connection with the given `id` to a pair of files whose names are constructed out of `tag`, `otag`, and `rtag`, as follows.

The originator-to-responder direction of the connection goes into a file named:

<*otag*>`.`< *tag*>`.`< *orig-addr*>`.`< *orig-port*>`-`< *resp-addr*>`.`< *resp-port*>

and the other direction in:

<*rtag*>`.`< *tag*>`.`< *resp-addr*>`.`< *resp-port*>`-`< *orig-addr*>`.`< *orig-port*>

Accordingly, *tag* can be used to associate a unique label with the pair of files, while *otag* and *rtag* provide distinct labels for the two directions.

If Bro is already demuxing the connection, or if the connection is not active, then nothing happens, and the function returns false. Otherwise, it returns true.

Bro places demuxed streams in a directory defined by the redefinable global `demux_dir`, which defaults in the usual fashion to `open_log_file("xscript")`.

Deficiency: Experience has shown that it would be highly convenient if Bro would demultiplex the entire *connection contents into the files, instead of just the part of the connection seen subsequently after the call to* `demux_conn`*. One way to do this would be for* `demux_conn` *to offset the contents in the file by the current stream position, and then to invoke a utility tool that goes through the Bro output trace file (§ 10.2, page 176) and copies the contents up to the current stream position to the front of the file. This utility tool might even be another instance of Bro running with suitable arguments.*

```
type dns_mapping: record {
    creation_time: time;  # When the mapping was created.

    req_host: string;     # The hostname in the request, if any.
    req_addr: addr;       # The address in the request, if any.

    valid: bool;          # Whether we received an answer.
    hostname: string;     # The hostname in the answer, or "<none>".
    addrs: set[addr];     # The addresses in the answer, if any.
};
```

Figure 7.5: Definition of the `dns_mapping` record.

7.12 The `dns` Module

The `dns` module deals with Bro's internal mapping of hostnames to/from IP addresses. *Deficiency: There is no DNS protocol analyzer available at present.* Furthermore, *Deficiency: the lookup mechanisms discussed here are not available to the Bro script writer, other than implicitly by using hostnames in lieu of addresses in variable initializations (§ 10.19, page 178).*

The module's function is to handle different events that can occur when Bro resolves hostnames upon startup. Bro maintains its own cache of DNS information which persists across invocations of Bro on the same machine and by the same user. The role of the cache is to allow Bro to resolve hostnames even in the face of DNS outages; the philosophy is that it's better to use old addresses than none at all, and this helps harden Bro against attacks in which the attacker causes DNS outages in order to prevent Bro from resolving particular sensitive hostnames (e.g., `hot_srcs`). The cache is stored in the file "`.bro-dns-cache`" in the user's home directory. You can delete this file whenever you want, for example to purge out old entries no longer needed, and Bro will recreate it next time it's invoked using `-P`.

Currently, all of the event handlers are invoked upon *comparing* the results of a new attempt to look up a name or an address versus the results obtained the *last time* Bro did the lookup. When Bro looks up a name for the first time, no events are generated.

Also, Bro currently only looks up hostnames to map them to addresses. It does not perform inverse lookups.

7.12.1 The `dns_mapping` record

All of the events handled by the module include at least one record of DNS mapping information, defined by the `dns_mapping` type shown in Figure 7.12.1. The corresponding fields are:

`creation_time`
When the mapping was created.

`req_host`
The hostname looked up, or an empty string if this was not a hostname lookup.

`req_addr`
The address looked up (reverse lookup), or `0.0.0.0` if this was not an address lookup.

`valid`

True if an answer was received for a lookup (even if the answer was that the request name or address does not exist in the DNS).

`hostname`

The hostname answer in response to an address lookup, or the string `"<none>"` if an answer was received but it indicated there was no PTR record for the given address.

`addrs`

A set of addresses in response to a hostname lookup. Empty if an answer was received but it indicated that there was no A record for the given hostname.

7.12.2 `dns` variables

The modules provides one redefinable variable:

`dns_interesting_changes : set[string]`

The different DNS events have names associated with them. If the name is present in this set, then the event will be logged, otherwise not.

One exception to this list is that DNS changes involving the loopback address `127.0.0.1` are always considered log-worthy, since they may reflect DNS corruption.

Default: { `"unverified", "old name", "new name", "mapping",` }.

7.12.3 `dns` event handlers

The DNS module supplies the following event handlers:

`dns_mapping_valid (dm: dns_mapping)`

The given request was looked up and it was identical to its previous mapping.

`dns_mapping_unverified (dm: dns_mapping)`

The given request was looked up but no answer came back.

`dns_mapping_new_name (dm: dns_mapping)`

In the past, the given address did not resolve to a hostname; this time, it did.

`dns_mapping_lost_name (dm: dns_mapping)`

In the past, the given address resolved to a hostname; now, that name has gone away. (An answer was received, but it stated that there is no hostname corresponding to the given address.)

`dns_mapping_name_changed (old_dm: dns_mapping, new_dm: dns_mapping)`

The name returned this time for the given address differs from the name returned in the past.

`dns_mapping_altered (dm: dns_mapping, old_addrs: set[addr], new_addrs: set[addr])`

The addresses associated with the given hostname have changed. Those in `old_addrs` used to be part of the set returned for the name, but aren't any more; while those in `new_addrs` didn't used to be, but now are. There may also be some unchanged addresses, which are those in `dm$addrs` but not in `new_addrs`.

7.13 The finger Analyzer

The finger analyzer processes traffic associated with the Finger service [RFC1288]. Bro instantiates a finger analyzer for any connection with service port 79/tcp (if you @load the finger analyzer in your script, or define your own finger_request or finger_reply handlers, of course).

The analyzer uses a capture filter of "port finger" (§ 7.1.2, page 93).

In the past, attackers often used Finger requests to obtain information about a site's users, and sometimes to launch attacks of various forms (buffer overflows, in particular). In our experience, exploitation of the service has greatly diminished over the past years (no doubt in part to the service being increasingly turned off, or prohibited by firewalls). Now it is only rarely associated with an attack.

7.13.1 finger variables

The standard script defines two redefinable variables:

hot_names : set[string]

A list of usernames that should be considered sensitive (log-worthy) if included in a Finger request.

Default: { "root", "lp", "uucp", "nuucp", "demos", "operator", "sync", "guest", "visitor", }.

max_request_length : count

The largest reasonable request size (used to flag possible buffer overflow attacks). Bro marks a connection as "hot" if its request exceeds this length, and truncates its logging of the request to this many bytes, followed by "...".

Default: 80.

7.13.2 finger event handlers

The standard script defines one event handler:

finger_request (c: connection, request: string, full: bool)

Invoked upon connection c having made the request request. The full flag is true if the request included the "long format" option (which the event engine will have removed from the request).

The standard script flags long requests and truncates them as noted above, and then checks whether the request is for a name in hot_names. It then formats the request either by placing double quotation marks around it, or, if the request was empty—indicating a request for information on all users—the request is changed to the string ALL with no quotes around it.

If the originator already made a request, then this additional request is placed in parentheses (though multiple requests violate the Finger protocol). If the request was for the full format, then the text "(/W)" is appended to the request. Finally, the request is appended to the connection's addl field.

The event engine generates an additional event that the predefined finger script does not handle:

finger_reply (c: connection, reply_line: string)

Generated for each line of text sent in response to the originator's request.

7.14 The frag Module

The frag utility module simply refines the capture filter (§ 7.1.2, page 93) so that Bro will capture and reassemble IP fragments. Bro reassembles any fragments it receives; but normally it doesn't receive any, except the beginnings of TCP fragments (see the tcp module), and UDP port 111 (per the portmapper module).

So, to make Bro do fragment reassembly, you simply use "@load frag". It effects this by adding:

```
(ip[6:2] & 0x3fff != 0) and tcp
```

to the filter. The first part of this expression matches all IP fragments, while the second restricts those matched to TCP traffic. We would *like* to use:

```
(ip[6:2] & 0x3fff != 0) and (tcp or udp port 111)
```

to also include portmapper fragments, but that won't work—the port numbers will only be present in the first fragment, so the packet filter won't recognize the subsequent fragments as belonging to a UDP port 111 packet, and will fail to capture them.

Note: Alternatively, we might be tempted to use "(tcp or udp)*" and so capture* all *UDP fragments, including port 111. This would work in principle, but in practice can capture very high volumes of traffic due to NFS traffic, which can send all of its file data in UDP fragments.*

7.15 The hot-ids Module

The hot-ids module defines a number of redefinable variables that specify usernames Bro should consider sensitive:

forbidden_ids : set[string]

lists usernames that should never be used. If Bro detects use of one, it will attempt to terminate the corresponding connection.

Default: { "uucp", "daemon", "rewt", "nuucp", "EZsetup", "OutOfBox", "4Dgifts", "ezsetup", "outofbox", "4dgifts", "sgiweb", }. All of these correspond to accounts that some systems have enabled by default (with well-known passwords), except for "rewt", which corresponds to a username often used by (weenie) attackers.

Deficiency: The repeated definitions such as "EZsetup" *and* "ezsetup" *reflect that this variable is a* set *and not a* pattern. *Consequently, the exact username must appear in it (with a pattern, we could use character classes to match both upper and lower case).*

forbidden_ids_if_no_password : set[string]

Same as forbidden_ids except only considered forbidden if the login succeeded with an empty password.

Default: "lp", a default passwordless IRIX account.

forbidden_id_patterns : pattern

A pattern giving user ids that should be considered forbidden. *Deficiency: This pattern is currently only used to check Telnet/Rlogin user ids, not ids seen in other contexts, such as FTP sessions.*

Default: /(y[o0]u)(r|ar[e3])([o0]wn.*)/, a particularly egregious style of username of which we've observed variants in different break-ins.

```
type ftp_session_info: record {
    id: count;              # unique number associated w/ session
    user: string;           # username, if determined
    request: string;        # pending request or requests
    num_requests: count;    # count of pending requests
    request_t: time;        # time of request
    log_if_not_denied: bool;        # unless code 530 on reply, log it
    log_if_not_unavail: bool;       # unless code 550 on reply, log it
    log_it: bool;           # if true, log the request(s)
};
```

Figure 7.6: Definition of the `ftp_session_info` record.

`always_hot_ids : set[string]`

A list of usernames that should always be considered sensitive, though not necessarily so sensitive that they should be terminated whenever used.

Default: { `"lp", "warez", "demos", forbidden_ids,` }. The `"lp"` and `"demos"` accounts are specified here rather than `forbidden_ids` because it's possible that they might be used for legitimate accounts. `"warez"` (for "wares", i.e., bootlegged software) is listed because its use likely constitutes a policy violation, not a security violation.

Note: `forbidden_ids` *is incorporated into* `always_hot_ids` *to avoid replicating the list of particularly sensitive ids by listing it twice and risking inconsistencies.*

`hot_ids : set[string]`

User ids that generate alerts if the user logs in successfully.

Default: { `"root", "system", always_hot_ids,` }. The ones included in addition to `always_hot_ids` are only considered sensitive if the user logs in successfully.

7.16 The `ftp` Analyzer

The `ftp` analyzer processes traffic associated with the FTP file transfer service [RFC959]. Bro instantiates an `ftp` analyzer for any connection with service port `21/tcp`, providing you have loaded the `ftp` analyzer, or defined a handler for `ftp_request` or `ftp_reply`.

The analyzer uses a capture filter of "`port ftp`" (§ 7.1.2, page 93). It generates summaries of FTP sessions; looks for sensitive usernames, access to sensitive files, and possible FTP "bounce" attacks, in which the host specified in a "`PORT`" or "`PASV`" directive does not correspond to the host sending the directive; or in which a different host than the server (client) connects to the endpoint specified in a `PORT` (`PASV`) directive.

7.16.1 The `ftp_session_info` record

The main data structure managed by the `ftp` analyzer is a collection of `ftp_session_info` records, where the record type is shown in Figure 7.16.1. The corresponding fields are:

`id`
: The unique session identifier assigned to this session. Sessions are numbered starting at `1` and incrementing with each new session.

`user`
: The username associated with this session (from the initial FTP authentication dialog), or an empty string if not yet determined.

`request`
: The pending request, if the client has issued any. Ordinarily there would be at most one pending request, but a client can in fact send multiple requests to the server all at once, and an attacker could do so attempting to confuse the analyzer into mismatching responses with requests, or simply forgetting about previous requests.

`num_requests`
: A count of how many requests are currently pending.

`request_t`
: The time at which the pending request was issued.

`log_if_not_denied`
: If true, then when the reply to the current request comes in, Bro should log it, unless the reply code is `530` (“`denied`”).

`log_if_not_unavail`
: If true, then when the reply to the current request comes in, Bro should log it, unless the reply code is `550` (“`unavail`”).

`log_it`
: If true, then when the reply to the current request comes in, Bro should log it.

7.16.2 `ftp` variables

The standard script defines the following redefinable variables:

`ftp_guest_ids : set[string]`
: A set of usernames associated with publicly accessible “guest” services. Bro interprets guest usernames as indicating Bro should use the authentication *password* as the effective username.

Default: { `"anonymous", "ftp", "guest",` }.

`ftp_skip_hot : set[addr, addr, string]`
: Entries indicate that a connection from the first given address to the second given address, using the given string username, should not be treated as hot even if the username is sensitive.

Default: empty.

Example: redefining `ftp_skip_hot` using

```
redef ftp_skip_hot: set[addr, addr, string] += {
    [[bob1.dsl.home.net, bob2.dsl.home.net], bob.work.com, "root"],
};
```

```
972499885.784104 #26 131.243.70.68/1899 > 64.55.26.206/ftp start
972499886.685046 #26 response (220 tuvok.ooc.com FTP server
    (Version wu-2.6.0(1) Fri Jun 23 09:17:44 EDT 2000) ready.)
972499886.686025 #26 USER anonymous/IEUser@ (logged in)
972499887.850621 #26 TYPE I (ok)
972499888.421741 #26 PASV (227 64.55.26.206/2427)
972499889.493020 #26 SIZE /pub/OB/4.0/JOB-4.0.3.zip (213 1675597)
972499890.135706 #26 *RETR /pub/OB/4.0/JOB-4.0.3.zip, ABOR (complete)
972500055.491045 #26 response (225 ABOR command successful.)
```

Figure 7.7: Example of FTP log file entries for a single FTP session.

would result in Bro not alerting on FTP connections as user `"root"` from either `bob1.dsl.home.net` or `bob2.dsl.home.net` to the server running on `bob.work.com`.

`ftp_hot_files : pattern`

Bro matches the argument given in each FTP file manipulation request (RETR, STOR, etc.) against this pattern to see if the file is sensitive. If so, and if the request succeeds, then the access is logged.

`eggdrop` Default: a pattern that matches various flavors of password files, plus any string with `eggdrop` in it. *Note: Eggdrop is an IRC management tool often installed by certain attackers upon a successful break-in.*

`ftp_not_actually_hot_files : pattern`

A pattern giving exceptions to `ftp_hot_files`. It turns out that a pattern like `/passwd/` generates a lot of false hits, such as from `passwd.c` (source for the *passwd* utility; this can turn up in FTP sessions that fetch entire sets of utility sources using `MGET`) or `passwd.html` (a Web page explaining how to enter a password for accessing a particular page).

Default: `/(passwd|shadow).*\.(c|gif|htm|pl|rpm|tar|zip)/`.

`ftp_hot_guest_files : pattern`

Files that guests should not attempt to access.

Default: `.rhosts` and `.forward` .

`skip_unexpected : set[addr]`

If a new host (address) unexpectedly connects to the endpoint specified in a `PORT` or `PASV` directive, then if either the original host or the new host is in this set, no message is generated. The idea is that you can specify multi-homed hosts that frequently show up in your FTP traffic, as these can generate innocuous warnings about connections from unexpected hosts.

Default: some `hp.com` hosts, as an example. Most are specified as raw IP addresses rather than hostnames, since the hostnames don't always consistently resolve.

`skip_unexpected_net : set[addr]`

The same as `skip_unexpected`, except addresses are masked to /24 and /16 before looked up in this set.

Default: empty.

In addition, ftp_log holds the name of the FTP log file to which Bro writes FTP session summaries. It defaults to open_log_file("ftp").

Figure 7.16.2 shows an example of what entries in this file look like. Here we see a transcript of the 26th FTP session seen since Bro started running. The first line gives its start time and the participating hosts and ports. The next line (split across two lines above for clarity) gives the server's welcome banner. The client then logged in as user "anonymous", and because this is one of the guest usernames, Bro recorded their password too, which in this case was "IEUser@" (a useless string supplied by their Web browser). The server accepted this authentication, so the status on the line is "(logged in)".

The client then issues a request for the Image file type, to which the server agreed. Next they issued a PASV directive, and received a response instructing them to connect to the server on port 2427/tcp for the next transfer. At this point, after issuing a SIZE directive (to which the server returned 1,675,597 bytes), they send RETR to fetch the file */pub/OB/4.0/JOB-4.0.3.zip*. However, before the transfer completed, they issued ABOR, but the transfer finished before the server processed the abort, so the log shows a status of (completed). Furthermore, because the client issued two commands without waiting for an intervening response, these are shown together in the log file, and the line marked with a "*" so it draws the eye. Finally, because Bro paired up the (completed) with the multi-request line, it then treats the response to the ABOR command as a reply by itself, showing in the last line that the server reported it successfully carried out the abort.

The corresponding lines in the red file look like:

```
972499885.784104 565.836 ftp 118 427 131.243.70.68 64.55.26.206
    RSTO L #26 anonymous/IEUser@
972499888.984116 165.098 ftp-data ? 1675597 131.243.70.68 64.55.26.206
    RSTO L
```

The first line summarizes the FTP control session (over which the client sends its requests and receives the server's responses). It includes an addl annotation of "#26 anonymous/IEUser@", summarizing the session number (so you can find the corresponding records in the ftp log file) and the authentication information.

The second line summarizes the single FTP data transfer, of 1,675,597 bytes. The amount of data sent by the client for this connection is shown as unknown because the client aborted the connection with a RST (hence the state RSTO). For connections that Bro does not look inside (such as FTP data transfers), it learns the amount of data transferred from the sequence numbers of the SYN and FIN connection control packets, and can't (reliably) learn them for the sender of a RST. (It can for the receiver of the RST.)

They also aborted the control session (again, state RSTO), but in this case, Bro captured all of the packets of the session, so it could still assign sizes to both directions.

7.16.3 ftp functions

The standard ftp script provides one function for external use:

is_ftp_data_conn (c: connection): bool

Returns true if the given connection matches one we're expecting as the data connection half of an FTP session. *Note: This function is not idempotent: if the connection matches an expected one, then Bro updates its state such that that connection is no longer expected. It also logs a discrepancy if the connection appears to be usurping another one that generated either a "PORT" or a "PASV" directive.*

Also returns true if the source port is 20/tcp and there's currently an FTP session active between the originator and responder, in case for some reason Bro's bookkeeping is inconsistent.

7.16.4 `ftp` event handlers

The standard script handles the following events:

`ftp_request (c: connection, command: string, arg: string)`
Invoked upon the client side of connection `c` having made the request `command` with the argument `arg`.

The processing depends on the particular command:

`USER`
Specifies the username that the client wishes to use for authentication. If it is sensitive—in `hot_ids` (which the `ftp` analyzer accesses via a `@load` of `hot-ids`)—then the analyzer flags the FTP session as log-worthy. In addition, if the username is in `forbidden_ids`, then the analyzer terminates the session.

The analyzer also updates the connection's `addl` field with the username.

`PASS`
Specifies the password to use for authentication.

If the password is empty and the username appears in `forbidden_ids_if_no_password` (also from the `hot-ids` analyzer), then the analyzer terminates the connection.

If the username corresponds to a guest account (`ftp_guest_ids`), then the analyzer updates the connection's `addl` field with the password as additional account information. Otherwise, it generates an `account_tried` event to facilitate detection of password guessing.

`PORT`
Instructs the FTP server to connect to the given IP address and port for delivery of the next FTP data item. The analyzer first checks the address/port specifier for validity. If valid, it will generate an alert if either the address specified in the directive does not match that of the client, or if the port corresponds to a "privileged" port, i.e., one in the range 0–1023. Finally, it establishes state so that `is_ftp_data_conn` can identify a subsequent connection corresponding to this directive as belonging to this FTP session.

`ACCT`
Specifies additional accounting information associated with a session, which the analyzer simply adds to the connection's `addl` field.

`APPE`, `CWD`, `DELE`, `MKD`, `RETR`, `RMD`, `RNFR`, `RNTO`, `STOR`, `STOU`
All of these manipulate files (and directories). The analyzer checks the filename against the policies to see if it is sensitive in the context of the given username (i.e., guest or non-guest), and, if so, marks the connection to generate an alert unless the operation fails. The analyzer also checks for an excessively long filename, currently by checking its length against a *Deficiency: hardwired maximum of 250 bytes*.

`ftp_reply (c: connection, code: count, msg: string, cont_resp: bool)`
Invoked upon the server side of connection `c` having replied to a request using the given status code and text message. `cont_resp` is true if the reply line is tagged as being continued to the next line. The analyzer only processes requests when the last line of a continued reply is received.

The analyzer checks the reply against any expected for the connection (for example, "`log_if_not_denied`") and generates alerts accordingly. If the reply corresponds to a `PASV` directive, then it parses the address/port specification in the reply and generates alerts in an analogous fashion as done by the `ftp_request` handler for `PORT` directives.

Finally, if the reply is not one that the analyzer is hardwired to skip (code `150`, used at the beginning of a data transfer, and code `331`, used to prompt for a password), then it writes a summary of the request and reply to the

```
972482763.371224 %1596 start 200.241.229.80 > 131.243.2.12
%1596 GET /ITG.hm.pg.docs/dissect/portuguese/dissect.html
%1596 GET /vfrog/bottom.icon.gif
%1596 GET /vfrog/top.icon.gif
%1596 GET /vfrog/movies/off.gif
%1596 GET /vfrog/new.frog.small.gif
```

Figure 7.8: Example of HTTP log file entries for a single HTTP session.

FTP log file (§ 7.16.2, page 125). Also, if the reply is an “orphan” (there was no corresponding request, perhaps because Bro started up after the request was made), then the reply is summarized in the log file by itself.

The standard `ftp` script defines one other handler, an instance of `connection_finished` used to flush FTP session information in case the session terminates abnormally and no reply is seen to the pending request(s).

7.17 The `http` Analyzer

The `http` analyzer processes traffic associated with the Hyper Text Transfer Protocol (HTTP) [RFC1945, RFC2616], the main protocol used by the Web. Bro instantiates an `http` analyzer for any connection with service port `80/tcp`, providing you have loaded the `http` analyzer, or defined a handler for `http_request`. It also instantiates an analyzer for service ports `8080/tcp` and `8000/tcp`, as these are often also used for Web servers.

The analyzer uses a capture filter of “`tcp dst port 80 or tcp dst port 8080 or tcp dst port 8000`” (§ 7.1.2, page 93). *Note: This filter excludes traffic sent by an HTTP server (that would be matched by* `tcp src port 80`*, etc.), because Deficiency: Bro doesn’t yet have an analyzer for HTTP replies.* It generates summaries of HTTP sessions (connections between the same client and server) and looks for access to sensitive URIs (effectively, URLs).

7.17.1 `http` variables

`sensitive_URIs : pattern`

Any HTTP method (e.g., `GET`, `HEAD`, `POST`) specifying a URI that matches this pattern is flagged as sensitive.

Default: URIs with `/etc/passwd` or `/etc/shadow` embedded in them, or `/cfdocs/expeval` (used in some Cold Fusion exploits). *Note: This latter generates some false hits; it’s mainly included just to convey the notion of looking for direct attacks rather than attacks used to exploit sensitive files like the first ones.*

Deficiency: It would be very handy to have variables providing hooks for more context when considering whether a particular access is sensitive, such as whether the request was inbound or outbound.

`sensitive_post_URIs : pattern`

Any `POST` method specifying a URI that matches this pattern is flagged as sensitive.

Default: URIs with `wwwroot` embedded in them.

In addition, `http_log` holds the name of the HTTP log file to which Bro writes HTTP session summaries. It defaults to `open_log_file("http")`.

Figure 7.17.1 shows an example of what entries in this file look like. Here we see a transcript of the 1596th HTTP session seen since Bro started running. The first line gives its start time and the participating hosts. The next five lines all correspond to GET methods retrieving different items from the Web server. *Deficiency: Bro can't log whether the retrievals succeeded or failed because it doesn't yet have an HTTP reply analyzer.*

The corresponding lines in the red file look like:

```
972482762.872695 481.551 http 441 5040 131.243.2.12 200.241.229.80
    S3 X %10596
972482764.686470 18.7611 http 596 7712 131.243.2.12 200.241.229.80
    S3 X %10596
972482764.685047 ? http 603 2959 131.243.2.12 200.241.229.80
    S1 X %10596
```

That there are three rather than five reflects *(i)* that the client used persistent HTTP, and so didn't need one connection per item, but also *(ii)* the client used three parallel connections (the maximum the standard allows is only two) to fetch the items more quickly. As with FTP sessions, the %10596 addl annotation lets you correlate the red entries with the http log entries.

Note: All three of the connections wound up in unusual states. The first two are in state S3*, which, as indicated by Table 7.3.6 means that the responder (in this case, the Web server) attempted to close the connection, but their was no reply from the originator. The last is in state* S1*, indicating that neither side attempted to close the connection (which is why no duration is listed for the connection).*

7.17.2 http event handlers

The standard HTTP script defines one event handler:

http_request (c: connection, request: string, URI: string)

Invoked whenever the client side of the given connection generates an HTTP request. request gives the HTTP method and URI the associated resource. The analyzer matches the URI against the ones defined as sensitive, as given above.

Deficiency: As mentioned above, the event engine does not currently generate an http_reply *event. This is for two reasons: first, the HTTP request stream is much lower volume than the HTTP reply stream, and I was interested in the degree to which Bro could get away without analyzing the higher volume stream. (Of course, this argument is shallow, since one could control whether or not Bro should analyze HTTP replies by deciding whether or not to define an* http_reply *handler.) Second, matching HTTP replies in their full generality involves a lot of work, because the HTTP standard allows replies to be delimited in a number of ways. That said, most of the work for implementing* http_reply *is already done in the event engine, but it is missing testing and debugging.*

7.18 The ident Analyzer

The ident analyzer processes traffic associated with the Identification Protocol [RFC1413], which provides a simple service whereby clients can query Ident servers to discover user information associated with an existing connection between the server's host and the client's host. Bro instantiates an ident analyzer for any connection with service port 113/tcp, providing you have loaded the ident analyzer, or defined a handler for ident_request, ident_reply, or ident_error.

The analyzer uses a capture filter of "`tcp port 113`" (§ 7.1.2, page 93). The `ident_reply` handler annotates the `addl` field of the connection for which the Ident client made its query with the user information returned in the reply. It also checks the user information against sensitive usernames, because a match indicates that the connection in the Ident query was initiated by a possibly-compromised account.

7.18.1 `ident` variables

The standard script defines the following pair of redefinable variables:

`hot_ident_ids : set[string]`
usernames to flag as sensitive if they appear in an Ident reply.

Default: `always_hot_ids` (§ 7.15, page 122).

`hot_ident_exceptions : set[string]`
usernames not to consider sensitive even if they appear in `hot_ident_ids`.

Default: { `"uucp", "nuucp", "daemon",` }. These usernames are exceptions because daemons sometimes run with the given user ids and their use is often innocuous.

7.18.2 `ident` event handlers

The standard script handles the following events:

`ident_request (c: connection, lport: port, rport: port)`
Invoked when a client request arrives on connection `c`, querying about the connection from local port `lport` to remote port `rport`, where local and remote are relative to the client.

`ident_reply (c: connection, lport: port, rport: port, user_id: string, system: string)`
Invoked when a server replies to an Ident request. `lport` and `rport` are again the local and remote ports (relative to the client) of the connection being asked about. `user_id` is the user information returned in the Ident server's reply, and `system` is information regarding the operating system (the Ident specification [RFC1413] does not further standardize this information).

The handler annotates the queried connection with the user information, which it also checks against `hot_ident_ids` and `hot_ident_exceptions` as discussed above. At present, it does nothing with the `system` information.

`ident_error (c: connection, lport: port, rport: port, line: string)`
Invoked when the given request yielded an error reply from the Ident server. The handler annotates the connection with `ident/`<*error*>, where *error* is the text given in `line`.

7.19 The login Analyzer

The login analyzer inspects interactive login sessions to extract username and password information, and monitors user keystrokes and the text returned by the login server. It is one of the most powerful Bro modules for detecting break-ins to Unix systems because of the ability to look for particular commands that attackers often execute once they have penetrated a Unix machine.

The analyzer is generic in the sense that it applies to more than one protocol. Currently, Bro instantiates a login analyzer for both Telnet [RFC854] and Rlogin [RFC1282] traffic. In principle, it could do the same for other protocols such as SSH [YKSRL00] or perhaps X11 [RFC1013], if one could write the corresponding elements of the event engine to decrypt the SSH session (naturally, this would require access to the encryption keys) or extract authentication information and keystrokes from the X11 event stream. *Note: The analyzer does an exceedingly limited form of SSH analysis; see* hot_ssh_orig_ports.

For Telnet, the event engine knows how to remove in-band Telnet option sequences [RFC855] from the text stream, and does not deliver these to the event handlers, except for a few options that the engine analyzes in detail (such as attempts to negotiate authentication). Unfortunately, the Telnet protocol does not include any explicit marking of username or password information (unlike the FTP protocol, as discussed in § 7.16, page 122). Consequently, Bro employs a series of heuristics that attempt to extract the username and password from the authentication dialog the session is presumed to begin with. The analysis becomes quite complicated due to the possible use of type-ahead and editing sequences by the user, plus the possibility that the user may be an attacker who attempts to mislead the heuristics in order to disguise the username they are accessing.

Analyzing Rlogin is nominally easier than analyzing Telnet because Rlogin has a simpler in-band option scheme, and because the Rlogin protocol explicitly indicates the username in the initial connection dialog. However, this last is not actually a help to the analyzer, because for most Rlogin servers, if the initial username fails authentication (for example, is not present in the .rhosts file local to the server), then the server falls back on the same authentication dialog as with Telnet (prompting for username and then password, or perhaps just for a password to go with the transmitted username). Consequently, the event engine employs the same set of heuristics as for Telnet.

Each connection processed by the analyzer is in a distinct state: user attempting to authenticate, user has successfully authenticated, analyzer is skipping any further processing, or the analyzer is confused (§ 7.19.1, page 131). You can find out the state of a given connection using get_login_state.

The analyzer uses a capture filter of "tcp port 23 or tcp port 513" (§ 7.1.2, page 93). It annotates each connection with the username(s) present in the authentication dialog. If the username was authenticated successfully, then it encloses the annotation in quotes. If the authentication failed, then the name is marked as failed/<*username*>. So, for example, if user "smith" successfully authenticates, then the connection's addl field will have "smith" appended to it:

```
931803523.006848 254.377 telnet 324 8891 1.2.3.4 5.6.7.8 SF L "smith"
```

while if "smith" failed to authenticate, the report will look like:

```
931803523.006848 254.377 telnet 324 8891 1.2.3.4 5.6.7.8 SF L fail/smith
```

and if they first tried as "smith" and failed, and then succeeded as "jones", the record would look like:

```
931803523.006848 254.377 telnet 324 8891 1.2.3.4 5.6.7.8 SF L fail/smith "jones"
```

Note: The event engine's heuristics can sometimes get out of synch such that it interprets a password as a username; in addition, users sometimes type their password when they should instead enter their username. Consequently, the connection logs sometimes include passwords in the annotations, and so should be treated as very sensitive information (e.g., not readable by any user other than the one running Bro).

7.19.1 `login` analyzer confusion

Because there is no well-defined protocol for Telnet authentication (or Rlogin, if the initial `.rhosts` authentication fails), the `login` analyzer employs a set of heuristics to detect the username, password, and whether the authentication attempt succeeded. All in all, these heuristics work quite well, but it is possible for them to become confused and reach incorrect conclusions.

Bro attempts to detect such confusion. If it does, then it generates a `login_confused` event, after which the event engine will no longer attempt to follow the authentication dialog. In particular, it will *not* generate subsequent `login_failure` or `login_success` events. The `login_confused` event includes a string describing the type of confusion, using one of the values given in Table 7.19.1.

7.19.2 `login` variables

The standard script defines a large number of variables for refining the analysis policy:

`input_trouble : pattern`
lists patterns that the analyzer should flag if they appear in the user's input (keystroke) stream.

The analyzer searches for these patterns both in the raw text typed by the user and the same lines after applying *editing* using the `edit` function twice: once with interpreting *BS* (ctrl-H) as delete-one-character, and once with *DEL* as the edit character. If any of these matches, then the analyzer considers the pattern to have matched.

`eggdrop` Default: a pattern matching occurrences of the strings "`rewt`", "`eggdrop`", "`loadmodule`", or "`/bin/eject`". The first of these is a popular username attackers use for root backdoor accounts. The second reflects that one prevalent class of attackers are devotees of Internet Relay Chat (IRC), who frequently upon breaking into an account install the IRC `eggdrop` utility.

`edited_input_trouble : pattern`
is the same as `input_trouble` except the analyzer only checks the edited user input against the pattern, not the raw input (see above).

This variable is provided so you can specify patterns that can occur innocuously as typos; whenever the user corrects the typo before terminating the line, the pattern won't match, because it won't be present in the edited version of the line. In addition, for matches to these patterns, the analyzer *delays* reporting the match until it sees the next line of output from the server. It then includes both the line that triggered the match and the corresponding response from the server, which makes it easy for a human inspecting the logs to tell if the occurrence of the pattern was in fact innocuous.

Here's an example of an innocuous report:

```
936723303.760483 1.2.3.4/21550 > 5.6.7.8/telnet
    input "cd ..." yielded output "ksh: ...:  not found."
```

It was flagged because the user's input included "`...`", a name commonly used by attackers to surreptitiously hide a directory containing their tools and the like. However, we see from the Telnet server's response that this was not actual access to such a directory, but merely a typing mistake.

On the other hand:

```
937528764.579039 1.2.3.4/3834 > 5.6.7.8/telnet
    input "cd ..." yielded output "maroon# ftp sunspot.sunspot.noao.edu "
```

Type of confusion	**Meaning**
"excessive_typeahead"	The user has typed ahead 12 or more lines. *Deficiency: The upper bound should be adjustable.*
"extra_repeat_text"	The user has entered more than one VMS repeat sequence (an escape followed by "[A") on the same line. *Note: Bro determines that a login session involves a VMS server if the server prompts with ""Username:"". It then interprets VMS repeat sequences as indicating it should replace the current line with the previous line.*
"multiple_USERs"	The user has specified more than one username using the $USER environment variable.
"multiple_login_prompts"	The analyzer has seen several login prompts on the same line, and has not seen a corresponding number of lines typed ahead previously by the user.
"no_login_prompt"	The analyzer has seen 50 lines sent by the server without any of them matching login_prompts. *Deficiency: The value of 50 should be adjustable.*
"no_username"	The analyzer is generating an event after having already seen a login failure, but the user's input has not provided another username to include with the event. *Note: If the analyzer's heuristics indicate it's okay that no new username has been given, such as when the event is generated due to one connection endpoint closing the connection, then it instead uses the username* <none>.
"no_username2"	The analyzer saw an additional password prompt without seeing an intervening username, and it has no previous username to reuse.
"non_empty_multi_login"	The analyzer saw multiple adjacent login prompts, with an apparently ignored intervening username typed-ahead between them.
"possible_login_ploy"	The client sent text that matches one of the patterns reflecting text usually sent by the server. This form of confusion can reflect an attacker attempting to evade the monitor. For example, the client may have sent the text "login:" as a username so that when echoed back by the server, the analyzer would misinterpret it as reflecting another login prompt from the server.
"repeat_without_username"	The user entered a VMS repeat sequence but there is no username to repeat. (See extra_repeat_text for a discussion of the analyzer's heuristics for dealing with VMS servers.)
"responder_environment"	The responder (login server) has signaled a set of environment variables to the originator (login client). This is in the opposite direction as to what makes sense.
"username_with_embedded_repeat"	The line repeated by a VMS server in response to a repeat sequence itself contains a repeat sequence.

Table 7.4: Different types of confusion that login analyzer can report.

shows a problem—the lines returned by the server was a root prompt (“`maroon#`”), to which the user issued a command to access a remote FTP server.

Deficiency: The analyzer should decouple the notion of waiting to receive the server’s reply from the notion of matching only the edited form of the line; there might be raw inputs for which it is useful to see the server’s response, and edited inputs for which the server’s response is unimportant in terms of knowing that the input spells trouble.

Default: the pattern

```
/[ \t]*cd[ \t]+(([′"]?\.\.\.)|(["′](\.[ˆ"′]*)[ \t]))/
```

which looks for a “`cd`” command to either a directory beginning with “`...`” (optionally quoted by the user) or a directory name beginning with “`.`” that is quoted and includes an embedded blank or tab.

`output_trouble : pattern`

lists patterns that the analyzer should flag if they occur in the output sent by the login server back to the user.

`PATH_UTMP` `smashdu.c`   Default: the pattern

```
      /ˆ-r.s.*root.*\/bin\/(sh|csh|tcsh)/
    | /Jumping to address/
    | /smashdu\.c/
    | /PATH_UTMP/
    | /Log started at =/
    | /www\.anticode\.com/
    | /smurf\.c by TFreak/
    | /Trojaning in progress/
    | /Super Linux Xploit/
```

The first of these triggers any time the user inspects with the *ls* utility an executable whose pathname ends in `/bin/` followed by one of the popular command shells, and the *ls* output shows that the command shell has been altered to be setuid to root. The remainder match either the output generated by some popular exploit tools (for example, “`Jumping to address`”, present in many buffer overflow exploit tools), exploit tool names (“`smashdu.c`”), text found within the tool source code (“`smurf.c by TFreak`”), or URLs accessed (say via the *lynx* or *fetch* utilities) to retrieve attack software (“`www.anticode.com`”).

`backdoor_prompts : pattern`

lists patterns that the analyzer should flag if they are seen as the first line sent by the server to the user, because they often correspond with backdoors that offer a remote user immediate command shell access without having to first authenticate.

Default: the pattern “`/ˆ[!-˜]*( ?)[#%$] /`”, which matches a line that begins with a series of printable, non-blank characters and ends with a likely prompt character, with a blank just after the prompt character and perhaps before it.

`non_backdoor_prompts : pattern`

lists patterns that if a possible backdoor prompt also matches, then the analyzer should not consider the server output as indicating a backdoor prompt. Used to limit false positives for `backdoor_prompts`.

Default: the pattern “`/ˆ *#.*#/`”, which catches lines with more than one occurrence of a `#`. Some servers generate such lines as part of their welcome banner.

`hot_terminal_types : pattern`

lists “magic” terminal types sometimes used by attackers to access backdoors. Both Telnet and Rlogin have mechanisms for negotiating a terminal type (name; e.g., “`xterm`”); these backdoors trigger and skip authentication if the name has a particular value.

`VT666` Default: the name “`VT666`”, one of the trigger terminal types we’ve observed in practice.

`hot_telnet_orig_ports : set[port]`

Some Telnet backdoors trigger if the ephemeral port used by the client side of the connection happens to be a particular value. This variable is used to list the port values whose use should be considered as possibly indicating a backdoor. *Note: Clearly, this mechanism can generate false positives when the client by chance happens to choose one of the listed ports.*

Default: `53982/tcp`, one of the trigger ports we have observed in practice.

Deficiency: There should be a corresponding variable for Rlogin backdoors triggered by a similar mechanism.

`hot_ssh_orig_ports : set[port]`

Similar to `hot_telnet_orig_ports`, only for SSH.

Default: `31337/tcp`, a trigger port that we’ve observed in practice.

`skip_authentication : set[string]`

A set of strings that, if present in the server’s initial output (i.e., its welcome banner), indicates the analyzer should not attempt to analyze the session for an authentication dialog. This is used for servers that provide public access and don’t bother authenticating the user.

Default: the string `"WELCOME TO THE BERKELEY PUBLIC LIBRARY"`, which corresponds to a frequently accessed public server in the Berkeley area. (Obviously, we include this default as an example, and not because it will be appropriate for most Bro users! But it does little harm to include it.)

Deficiency: It would be more natural if this variable and a number of others listed below were of type `pattern` *rather than* `set[string]`. *They are actually converted internally by the event engine into regular expressions.*

`direct_login_prompts : set[string]`

A set of strings that if seen during the authentication dialog mean that the user will be logged in as soon as they answer the prompt.

Default: `"TERMINAL?"`, a prompt used by some terminal servers.

`login_prompts : set[string]`

A set of strings corresponding to login username prompts during an authentication dialog.

Default: the strings

```
Login:
login:
Name:
Username:
User:
Member Name
```

and the default contents of `direct_login_prompts`.

`login_failure_msgs : set[string]`

A set of strings that if seen in text sent by the server during the authentication dialog correspond to a failed login attempt.

Default: the strings

```
invalid
Invalid
incorrect
Incorrect
failure
Failure,
User authorization failure,
Login failed,
INVALID
Sorry,
Sorry.
```

`login_non_failure_msgs : set[string]`

A set of strings similar to `login_failure_msgs` that if present mean that the server text does not actually correspond to an authentication failure (i.e., if `login_failure_msgs` also matches, it's a false positive).

Default: the strings

```
Failures
failures
failure since last successful login
failures since last successful login
```

`router_prompts : set[string]`

A set of strings corresponding to prompts returned by the local routers when a user successfully authenticates to the router. For the purpose of this variable, see the next variable.

Default: empty.

`login_success_msgs : set[string]`

A set of strings that if seen in text sent by the server during the authentication dialog correspond to a successful authentication attempt.

Default: the strings

```
Last login
Last successful login
Last   successful login
checking for disk quotas
unsuccessful login attempts
failure since last successful login
failures since last successful login
```

and the default contents of the `router_prompts` variable.

Deficiency: Since by default `router_prompts` *is empty, this last inclusion does nothing. In particular, if you redefine* `router_prompts` *then* `login_success_msgs` *will* not *pick up the change; you will need to redefine it to (again) include* `router_prompts`*, using:* `redef login_success_msgs += router_prompts`*. This is clearly a misfeature of Bro and will be fixed one fine day.*

`login_timeouts : set[string]`
A set of strings that if seen in text sent by the server during the authentication dialog correspond to the server having timed out the authentication attempt.

Default: the strings

```
timeout
timed out
Timeout
Timed out
Error reading command input
```

(This last is returned by the VMS operating system.)

`non_ASCII_hosts : set[addr]`
A set of addresses corresponding to hosts whose login servers do not (primarily) use 7-bit ASCII. The analyzer will not attempt to analyze authentication dialogs to such hosts, and will not complain about huge lines generated by either the sender or receiver (per `excessive_line`).

Default: empty.

`skip_logins_to : set[addr]`
A set of addresses corresponding to hosts for which the analyzer should not attempt to analyze authentication dialogs.

Default: the (empty) contents of `non_ASCII_hosts`.

`always_hot_login_ids : set[string]`
A set of usernames that the analyzer should always flag as sensitive, even if they're seen in a session for which the analyzer is *confused* (§ 7.19.1, page 131).

Default: the value of `always_hot_ids` defined by the `hot` analyzer.

`hot_login_ids : set[string]`
A set of usernames that the analyzer should flag as sensitive, unless it sees them in a session for which the analyzer is *confused* (§ 7.19.1, page 131).

Default: the value of `hot_ids` defined by the `hot-ids` analyzer.

`rlogin_id_okay_if_no_password_exposed : set[string]`
A set of username exceptions to `hot_login_ids` which the analyzer should not flag as sensitive if the user authenticated without exposing a password (so, for example, via `.rhosts`).

Default: the username `"root"`.

7.19.3 `login` functions

The standard `login` script provides the following functions for external use:

`is_login_conn (c: connection): bool`
: Returns true if the given connection is one analyzed by `login` (currently, Telnet or Rlogin), false otherwise.

`hot_login (c: connection, msg: string, tag: string)`
: Marks the given connection as hot, logs the given message, and demultiplexes (§ 10.17, page 178) the subsequent server-side contents of the connection to a filename based on `tag` and the client-side to a filename based on the name `"keys"`. No return value.

`is_hot_id (id: string, successful: bool, confused: bool): bool`
: Returns true if the username id should be considered sensitive, given that the user either did or did not successfully authenticate, and that the analyze was or was not in a *confused* state (§ 7.19.1, page 131).

`is_forbidden_id (id: string): bool`
: Returns true if the username id is present in `forbidden_ids` or `forbidden_id_patterns`.

`edit_and_check_line (c: connection, line: string, successful: bool): check_info`
: Tests whether the given line of text seen on connection `c` includes a sensitive username, after first applying *BS* and *DEL* keystroke editing (§ 7.19.2, page 131). `successful` should be true if the user has successfully authenticated, false otherwise.

The return value is a `check_info` record, which contains four `check_info` fields:

`expanded_line`
: All of the different editing interpretations of the line, separated by commas. For example, if the original line is

`"rob<`*`DEL`*`><`*`BS`*`><`*`BS`*`>ot"`

then the different editing interpretations are `"ro<`*`BS`*`><`*`BS`*`>ot"` and `"root"`, so the return value will be:

`"rob<`*`DEL`*`><`*`BS`*`><`*`BS`*`>ot,ro<`*`BS`*`><`*`BS`*`>ot,root"`

Deficiency: Ideally, these values would be returned in a list of some form, so that they can be accessed separately and unambiguously. The current form is really suitable only for display to a person, and even that can be quite confusing if `line` *happens to contain commas already.* Or, *perhaps an algorithm of "simply pick the shortest" would find the correct editing every time anyway.*

`hot: bool` True if any editing sequence resulted in a match against a sensitive username.

`hot_id: string` The version of the input line (with or without editing) that was considered hot, or an empty string if none.

`forbidden: bool` True if any editing sequence resulted in a match against a username considered "forbidden", per `is_forbidden_id`.

`edit_and_check_user (c: connection, user: string, successful: bool, fmt_s: string): bool`
: Tests whether the given username used for authentication on connection `c` is sensitive, after first applying *BS*

and *DEL* keystroke editing (§ 7.19.2, page 131). `successful` should be true if the user has successfully authenticated, false otherwise.

`fmt_s` is a `fmt` format specifying how the username information should be included in the connection's `addl` field. It takes two `string` parameters, the current value of the field and the expanded version of the username as described in `expanded_line`.

If `edit_and_check_line` indicates that the username is sensitive, then `edit_and_check_user` records the connection into its own demultiplexing files (§ 10.17, page 178). If the username is *forbidden*, then unless the analyzer is confused, we attempt to terminate the connection using `terminate_connection`.

Returns true if the connection is now considered "hot," either due to having a sensitive username, or because it was hot upon entry to the function.

`edit_and_check_password (c: connection, password: string): bool`

Checks the given password to see whether it contains a sensitive username. If so, then marks the connection as hot and logs the sensitive password. No return value.

Note: The purpose of this function is to catch instances in which the event engine becomes out of synch with the authentication dialog and mistakes what is, in fact, a username being entered, for a password being entered. Such confusion can come about either due to a failure of the event engine's heuristics, or due to deliberate manipulation of the event engine by an attacker.

7.19.4 `login` event handlers

The standard `login` script handles the following events:

`login_failure (c: connection, user: string, client_user: string, password: string, line: string)`

Invoked when the event engine has seen a failed attempt to authenticate as `user` with `password` on the given connection `c`. `client_user` is the user's username on the client side of the connection. For Telnet connections, this is an empty string, but for Rlogin connections, it is the client name passed in the initial authentication information (to check against `.rhosts`). `line` is the line of text that led the analyzer to conclude that the authentication had failed.

The analyzer first generates an `account_tried` event to facilitate detection of password guessing, and then checks for a sensitive username or password. If the username was not sensitive and the password is empty, then no further analysis is applied, since clearly the attempt was half-hearted and aborted. Otherwise, the analyzer annotates the connection's `addl` field with `fail/<username>` to mark the authentication failure, and also checks the `client_user` to see if it is sensitive. If we then find that the connection is hot, the analyzer logs a message to that effect.

`login_success (c: connection, user: string, client_user: string, password: string, line: string)`

Invoked when the event engine has seen a successful attempt to authenticate. The parameters are the same as for `login_failure`.

The analyzer invokes `check_hot` with mode `APPL_ESTABLISHED` since the application session has now been established. It generates an `account_tried` event to facilitate detection of password guessing, and then checks for a sensitive username or password. The event engine uses the special password `"<none>"` to indicate that no password was exposed, and this mitigates the sensitivity of logins using particular usernames per `rlogin_id_okay_if_no_password_exposed`.

The analyzer annotates the connection's `addl` field with `"<`*`username`*`>"` to mark the successful authentication. Finally, if we then find that the connection is hot, the analyzer logs a message to that effect.

`login_input_line (c: connection, line: string)`
Invoked for every line of text sent by the client side of the login session to the server side. The analyzer matches the text against `input_trouble` and `edited_input_trouble` and invokes `hot_login` with a tag of `"trb"` if it sees a match, which will log an alert concerning the connection. However, this invocation is only done while the connection's `hot` field count is ≤ 2, to avoid cascaded alerts when an attacker gets really busy and steps on a lot of sensitive patterns.

`login_output_line (c: connection, line: string)`
Invoked for every line of text sent by the server side of the login session to the client side. The analyzer checks `backdoor_prompts` and any pending input alerts that were waiting on the server output, per `edited_input_trouble`. These last are then logged unless the output matched the pattern:

```
/No such file or directory/
```

Deficiency: Clearly, this pattern should not be hardwired but instead specified by a redefinable variable.

Finally, if the line is not too long and the text matches `output_trouble` and the connection's `hot` field count is ≤ 2 (to avoid cascaded alerts), the analyzer invokes `hot_login` with a tag of `"trb"`. *Deficiency: "Too long" is hardwired to be a length ≥ 256 bytes. It, too, should be specifiable via a redefinable variable. Note: We might wonder if not checking overly long lines presents an evasion threat: the attacker can bury their access to a sensitive string in an excessive line and thus avoid detection. While this is true, it doesn't appear to cost much. First, some of the sensitive patterns are generated in server output that will be hard to manipulate into being overly long. Second, if the attacker is trying to avoid detection, there are easier ways, such as passing their output through a filter that alters it a good deal.*

`login_confused (c: connection, msg: string, line: string)`
Invoked when the event engine's heuristics have concluded that they have become confused and can no longer correctly track the authentication dialog (§ 7.19.1, page 131). `msg` gives the particular problem the heuristics detected (for example, `multiple_login_prompts` means that the engine saw several login prompts in a row, without the type-ahead from the client side presumed necessary to cause them) and `line` the line of text that caused the heuristics to conclude they were confused.

Once declaring that it's confused, the event engine will no longer attempt to follow the authentication dialog. In particular, it will *not* generate subsequent `login_failure` or `login_success` events.

Upon this event, the standard `login` script invokes `check_hot` with mode `APPL_ESTABLISHED` since it could well be that the application session is now established (it can't know for sure, of course, because the event engine has given up). It annotates the connection's `addl` field with `confused/<`*line*`>` to mark the confused state, and then logs to the `weird` file the particulars of the connection and the type of confusion (`msg`). *Deficiency: This should be done by generating a* weird*-related event instead.*

Finally, the analyzer invokes `set_record_packets` to specify that all of the packets associated with this connection should be recorded to the `trace` file. *Note: For the current* `login` *analyzer, this call is not needed—it records every packet of every login session anyway, because the generally philosophy is that Bro should record* *whatever it analyzes, so that the analysis may be repeated or examined in detail. Since the current analyzer looks at every input and output line via* `login_input` *and* `login_output`*, it records all of the packets of every such analyzed session. There is commented-out text in* `login_success` *to be used if* `login_input` *and* `login_output` *are not being used; it turns off recording of a session's packets after the user has successfully logged in (assuming the connection is not considered hot).*

`login_confused_text (c: connection, line: string)`

Invoked for every line the user types after the event engine has entered the *confused* state. If the connection is not already considered hot, then the analyzer checks for the presence of sensitive usernames in the line using `edit_and_check_line`, and, if present, annotates the connection's `addl` field with `confused/`<*line*>, logs that the connection has become hot, and invokes `set_record_packets` to record to the `trace` file all of the packets associated with the connection.

`login_terminal (c: connection, terminal: string)`

Invoked when the client transmits a terminal type to the server. The mechanism by which the client transmits the type depends on the underlying protocol (Rlogin or Telnet).

The handler checks the terminal type against `hot_terminal_types` and if it finds a match invokes `hot_login` with a tag of `"trb"`.

`excessive_line (c: connection)`

Invoked when the event engine observes a very long line sent by either the client or the server. Such long lines are seen as potential attempts by an attacker to evade the `login` analyzer; or, possibly, as a Login session carrying an unusual application. *Note: One example we have observed occurs when a high-bandwidth binary payload protocol such as Napster is sent over the Telnet or Rlogin well-known port in an attempt to either evade detection or tunnel through a firewall.*

This event is actually generic to any TCP connection carrying an application that uses the "Network Virtual Terminal" (NVT) abstraction, which presently comprises Telnet and FTP. But the only handler defined in the demonstration Bro policy is for Telnet, hence we discuss it here. For this reason, the handler first invokes `is_login_conn` to check whether the connection is in fact a login session. If so, then if the connection is not hot, and if the analyzer finds the server listed in `non_ASCII_hosts`, then it presumes the long line is due to use of a non-ASCII character set; the analyzer invokes `set_login_state` and `set_record_packets` to avoid further analysis or recording of the connection.

Otherwise, if the connection is still in the authentication dialog, then the handler generates a `login_confused` event with a confusion-type of `"excessive_line"`, and changes the connection's state to *confused*.

Deficiency: The event engine is currently hardwired to consider a line of ≥ 1024 *bytes as "excessive"; clearly this should be user-redefinable.*

`inconsistent_option (c: connection)`

NVT options are specified by the client and server stating which options they are willing to support vs. which not, and then instructing one another which in fact they should or should not use for the current connection. If the event engine sees a peer violate either what the other peer has instructed it to do, or what it itself offered in terms of options in the past, then the engine generates an `inconsistent_option` event.

The handler for this event simply records an entry about it to the `weird` file. *Deficiency: The event handler invocation does not include enough information to determine what option was inconsistently specified; in addition, it would be convenient to integrate the handling of problems like this within the general "weird" framework.*

Note: As for `excessive_line` *above, this event is actually a generic one applicable to any NVT-based protocol. It is handled here because the problem most often crops up for Telnet sessions. Note: Also, the handler does not check to see whether the connection is a login session (as it does for* `excessive_line`*); it serves as the handler for any NVT session with an excessive line.*

Note: Finally, note that this event can be generated if the session contains a stream of binary data. One way this can occur is when the session is encrypted but Bro fails to recognize this fact.

`bad_option (c: connection)`

If an NVT option is either ill-formed (e.g., a bad length field) or unrecognized, then the analyzer generates this event.

The processing of this event (recording information to the `weird` file) and the various notes and deficiencies associated with it are the same as those for `inconsistent_option` above.

`bad_option_termination (c: connection)`

If an NVT option fails to be terminated correctly (for example, a character is seen within the option that is disallowed for use in the option), then the analyzer generates this event.

The processing of this event (recording information to the `weird` file) and the various notes and deficiencies associated with it are the same as those for `inconsistent_option` above.

`authentication_accepted (name: string, c: connection)`

The NVT framework includes options for negotiating authentication. When such an option is sent from client to server and the server replies that it accepts the authentication, then the event engine generates this event.

The handler annotates the connection's `addl` field with `auth/`<*name*>, unless that annotation is already present.

`authentication_rejected (name: string, c: connection)`

The same as `authentication_accepted`, except invoked when the server replies that it rejects the attempted authentication.

The handler annotates the connection's `addl` field with `auth-failed/`<*name*>.

`authentication_skipped (c: connection)`

Invoked when the event engine sees a line in the authentication dialog that matches `skip_authentication`.

The handler annotates the connection's `addl` field with "`(skipped)`" to mark that authentication was skipped, and then invokes `skip_further_processing` and (unless the connection is hot) `set_record_packets` to skip any further analysis of the connection, and to stop recording its packets to the `trace` file.

`connection_established (c: connection)`

`connection_established` is a generic event generated for all TCP connections; however, the `login` analyzer defines an additional handler for it.

The handler first checks (via `is_login_conn`) whether this is a Telnet or Rlogin connection. If so, it generates an `authentication_skipped` event if the server's address occurs in `skip_logins_to`, and also (for Telnet) checks whether the client's port occurs in `hot_telnet_orig_ports`, invoking `hot_login` with the tag `"orig"` if it does.

For SSH connections, it likewise checks the client's port, but in `hot_ssh_orig_ports`, marking the connection as hot and logging a real-time alert if it is.

Call	**Meaning**
NULL	A do-nothing call typically provided by all RPC services.
GETPORT	Look up the port associated with a given RPC program.
SET	Add a new port mapping (or replace an existing mapping) for an RPC program.
UNSET	Remove a port mapping.
DUMP	Retrieve all of the RPC program mappings.
CALLIT	Both look up a program and then directly call it.

Table 7.5: Types of calls to the RPC portmapper service.

`partial_connection (c: connection)`

As noted earlier, `partial_connection` is a generic event generated for all TCP connections. The `login` analyzer also defines a handler for it, one which (if it's a Telnet/Rlogin connection) sets the connection's state to *confused* and checks for `hot_telnet_orig_ports`.

`activating_encryption (c: connection)`

The NVT framework includes options for negotiating encryption. When such a series of options is successfully negotiated, the event engine generates this event. *Note: The negotiation sequence is complex and can fail at a number of points. The event engine does not attempt to generate events for each possible failure, but instead only looks for the option sent after a successful negotiation sequence.*

The handler annotates the connection's `addl` field with "`(encrypted)`" to mark that authentication was encrypted. *Note: The event engine itself marks the connection as requiring no further processing. This is done by the event engine rather than the handler because the event engine cannot do its job (regardless of the policy the handler might desire) in the face of encryption.*

7.20 The `portmapper` Analyzer

The `portmapper` analyzer monitors one particularly important form of remote procedure call (RPC) [RFC1831, RFC1832] traffic: the portmapper service, used to map between RPC program (and version) numbers and the TCP or UDP port on which the service runs for a particular host. For example, *rstatd* is an RPC service that provides "remote host status monitoring" so that a set of hosts can be informed when any of them reboots. *rstatd* has been assigned a standard RPC program number of 100002. To find out the corresponding TCP or UDP port on a given host, a remote host would usually first contact the portmapper RPC service running on the host and request the port corresponding to program 100002.

All in all, clients can make six different types of calls to the portmapper, as summarized in Table 7.20. Attackers often use GETPORT and DUMP to see whether a host may be running an RPC service vulnerable to a known exploit.

The analyzer uses a capture filter of "`port 111`" (§ 7.1.2, page 93), equivalent to "`tcp port 111 or udp port 111`" (since the portmapper service ordinarily accepts calls using either TCP or UDP, both on port 111). It checks the different types of portmapper calls against policies expressed using a number of different variables.

Note: An important point not to overlook is that an attacker does not *have to first call the portmapper service in order to call an RPC program. They might instead happen to know the port on which the service runs* a priori, *since for example it may generally run on the same port for a particular operating system; or they might scan the host's* *different TCP or UDP ports directly looking for a reply from the service. Thus, while portmapper monitoring proves very useful in detecting attacks, it does* not *provide comprehensive monitoring of attempts to exploit RPC services.*

7.20.1 `portmapper` variables

The standard script provides the following redefinable variables:

`rpc_programs : table[count] of string`

Maps RPC program numbers to a string used to name the service. For example, the `[100002]` entry is mapped to `"rstatd"`.

Default: a large list of RPC services.

`NFS_services : set of string`

Lists the names of those RPC services that correspond to Network File System (NFS) [RFC1094, RFC1813] services. This variable is provided because it is convenient to express policies specific to accessing NFS file systems.

Default: the services *mountd*, *nfs*, *pcnfsd*, *nlockmgr*, *rquotad*, *status*.

Deficiency: Bro's notion of NFS is currently confined to just knowledge of the existence of these services. It does not analyze the particulars of different NFS operations.

`RPC_okay : set[addr, addr, string]`

Indexed by the host providing a given service and then by the host accessing the service. If an entry is present, it means that the given access is allowed. For example, an entry of:

```
[1.2.3.4, 5.6.7.8, "rstatd"]
```

means that host `5.6.7.8` is allowed to access the *rstatd* service on host `1.2.3.4`.

Default: empty.

`RPC_okay_nets : set[net]`

A set of networks allowed to make GETPORT requests without complaint. The notion behind providing this variable is that the listed networks are trusted. However, the trust doesn't extend beyond GETPORT to other portmapper requests, because GETPORT is the only portmapper operation used routinely by a set of hosts trusted by another set of hosts (but that don't belong to the same group, and hence are not issuing SET and UNSET calls).

Default: empty.

`RPC_okay_services : set[string]`

A set of services for which GETPORT requests should not generate complaints. These might be services that are widely invoked and believed exploit-free, such as *walld*, though care should be taken with blithely assuming that a given service is indeed exploit-free.

Note that, like for `RPC_okay_nets`, the trust does not extend beyond GETPORT, because it should be the only portmapper operation routinely invoked.

Default: empty.

`NFS_world_servers : set[addr]`

A set of hosts that provide public access to an NFS file system, and thus should not have any of their NFS traffic flagged as possibly sensitive. (The presumption here is that such public servers have been carefully secured against any remote NFS operations.) An example of such a server might be one providing read-only access to a public database.

Default: empty.

`RPC_dump_okay : set[addr, addr]`

Indexed first by the host requesting a portmapper dump, and second by the host from which it's requesting the dump. If an entry is present, then the dump operation is not flagged.

Default: empty.

`any_RPC_okay : set[addr, string]`

Pairs of hosts and services for which any GETPORT access to the given service is allowed.

`sun-rpc.mcast.net` Default:

```
    [NFS_world_servers, NFS_services],
    [sun-rpc.mcast.net, "ypserv"]
```

The first of these allows access to any NFS service of any of the `NFS_world_servers`, using Bro's cross-product initialization feature (§ 3.12.2, page 35). The second allows *ypserv* requests to the multicast address reserved for RPC multicasts.[2]

`suppress_pm_log : table[addr, string] of bool`

Do not generate real-time alerts for access by the given address for the given service. Note that unlike most Bro policy variables, this one is not `const` but is modified at run-time to add to it any host that invokes the *walld* RPC service, so that such access is only reported once for each host.

Default: empty, but dynamic as discussed above.

7.20.2 `portmapper` functions

The standard script provides the following externally accessible functions:

`rpc_prog (p: count): string`

Returns the name of the RPC program with the given number, if it's present in `rpc_programs`; otherwise returns the text `"unknown-<p>"`.

`pm_check_getport (r: connection, prog: string): bool`

Checks a GETPORT request for the given program against the policy expressed by `RPC_okay_services`, `any_RPC_okay`, `RPC_okay`, and `RPC_okay_nets`, returning true if the request violates policy, false if it's allowed.

[2] I don't know how much this type of access is actually used in practice, but experience shows that requests for *ypserv* directed to that address pop up not infrequently.

`pm_activity (r: connection, log_it: bool)`
A bookkeeping function invoked when there's been portmapper activity on the given connection.

The function records the connection via `record_connection`, unless it is a TCP connection (which will instead be recorded by `connection_finished`). If `log_it` is true then the function generates a real-time alert of the form:

rpc: <*connection-id*> <*RPC-service*> <*r$addl*>

For example:

```
    972616255.679799 rpc: 65.174.102.21/832 > 182.7.9.47/portmapper
pm_getport: nfs -> 2049/udp
```

However, it does not generate the alert if either the client host and service are present in `suppress_pm_log`, or if it already generated an alert in the past for the same client, server and service (to prevent alert cascades).

`pm_request (r: connection, proc: string, addl: string, log_it: bool)`
Invoked when the given connection has made a portmapper request of some sort for the given RPC procedure `proc`. `addl` gives an annotation to add to the connection's `addl` field. If `log_it` is true, then connection should be logged; it will also be logged if the function determines that it is hot.

The function first invokes `check_scan` and `check_hot` (with a mode of `CONN_ESTABLISHED`), unless `r` is a TCP connection, in which case these checks have already been made by `connection_established`. The function then adds `addl` to the connection's `addl` field, though if the field's length already exceeds 80 bytes, then it just tacks on `"..."` (unless already present). This last is necessary because Bro will sometimes see zillions of successive portmapper requests that all use the same connection ID, and these will each add to `addl` until it becomes unwieldy in size. *Deficiency: Clearly, the byte limit of 80 should be adjustable.*

Finally, the function invokes `check_hot` with a mode of `CONN_FINISHED`, and `pm_activity` to finish up bookkeeping for the connection.

No return value.

`pm_attempt (r: connection, proc: string, status: count, addl: string, log_it: bool)`
Invoked when the given connection attempted to make a portmapper request of some sort, but the request failed or went unanswered. The arguments are the same as for `pm_request`, with the addition of `status`, which gives the RPC status code corresponding to why the attempt failed (see below).

The function first invokes `check_scan` and `check_hot` (with a mode of `CONN_ATTEMPTED`), unless `r` is a TCP connection, in which case these checks have already been made by `connection_attempt`.

The function then adds `addl` to the connection's `addl` field, along with a text description of the RPC status code, as given in Table 7.20.2.

No return value.

Status description	Meaning
`"ok"`	The call succeeded.
`"prog unavail"`	The call was for an RPC program that has not registered with the portmapper.
`"mismatch"`	The call was for a version of the RPC program that has not registered with the portmapper.
`"garbage args"`	The parameters in the call did not decode correctly.
`"system err"`	A system error (such as out-of-memory) occurred when processing the call.
`"timeout"`	No reply was received within 24 seconds of the request.
`"auth error"`	The caller failed to authenticate to the server, or was not authorized to make the call.
`"unknown"`	An unknown error occurred.

Table 7.6: Types of RPC status codes.

7.20.3 `portmapper` event handlers

The standard script handles the following events:

`pm_request_null (r: connection)`

Invoked upon a successful portmapper request for the "null" procedure. The script invokes `pm_request` with `log_it=F`.

`pm_request_set (r: connection, m: pm_mapping, success: bool)`

Invoked upon a nominally successful portmapper request to set the portmapper binding `m`. The script invokes `pm_request` with `log_it=T`. `success` is true if the server honored the request, false otherwise; the script turns this into an annotation of `"ok"` or `"failed"`.

The `pm_mapping` type (for `m`) has three fields, `program: count`, `version: count` and `p: port`, the port for the mapping of the given program and version. `pm_mapping`

`pm_request_unset (r: connection, m: pm_mapping, success: bool)`

Invoked upon a nominally successful portmapper request to remove a portmapper binding. The script invokes `pm_request` with `log_it=T`. `success` is true if the server honored the request, false otherwise; the script turns this into an annotation of `"ok"` or `"failed"`.

`pm_request_getport (r: connection, pr: pm_port_request, p: port)`

Invoked upon a successful portmapper request to look up a portmapper binding. `pr`, of type `pm_port_request`, has three fields: `program: count`, `version: count`, and `is_tcp: bool`, this last indicating whether the caller is request the TCP or UDP port, if the given program/version has mappings for both. The script invokes `pm_request` with `log_it` set according to the return value of `pm_check_getport` and an annotation of the mapping.

`pm_request_dump (r: connection, m: pm_mappings)`

Invoked upon a successful portmapper request to dump the portmapper bindings. The script invokes `pm_request` with `log_it=T` unless `RPC_dump_okay` indicates that the dump call is allowed. The script ignores `m`, which gives the mappings as a `table[count] of pm_mapping`, where the table index simply reflects the order in which the mappings were returned, starting with an index of 1. *Deficiency: What the script* should *do, instead, is keep track of the mappings so that Bro can identify the service associated with connections for otherwise unknown ports.*

pm_request_callit (r: connection, pm_callit_request, p: port)

Invoked upon a successful portmapper request to look up and call an RPC procedure. The script invokes pm_request with log_it=T unless the combination of the caller and the program are in suppress_pm_log. Finally, if the program called is *walld*, then the script adds the caller to suppress_pm_log.

The pm_callit_request type has four fields: pm_callit_request program: count, version: count, proc: count, and arg_size: count. These reflect the procedure being looked up and called, and the size of the arguments being passed to it, respectively. *Deficiency: Currently, the event engine does not do any analysis or refinement of the arguments passed to the procedure (such as making them available to the event handler) or the return value.* p is the port value returned by the call.

pm_attempt_null (r: connection, status: count)

Invoked upon a failed portmapper request for the "null" procedure. status gives the reason for the failure. The script invokes pm_attempt with log_it=T.

pm_attempt_set (r: connection, status: count, m: pm_mapping)

Invoked upon a failed portmapper request to set the portmapper binding m. The script invokes pm_attempt with log_it=T.

pm_attempt_unset (r: connection, status: count, m: pm_mapping)

Invoked upon a failed portmapper request to remove a portmapper binding. The script invokes pm_attempt with log_it=T.

pm_attempt_getport (r: connection, status: count, pr: pm_port_request)

Invoked upon a failed portmapper request to look up a portmapper binding. pr, of type pm_port_request, has three fields: program: count, version: count, and is_tcp: bool, this last indicating whether the caller requested the TCP or UDP port. The script invokes pm_attempt with log_it set according to the return value of pm_check_getport.

pm_attempt_dump (r: connection, status: count)

Invoked upon a failed portmapper request to dump the portmapper bindings. The script invokes pm_attempt with log_it=T unless RPC_dump_okay indicates that the dump call is allowed.

pm_attempt_callit (r: connection, status: count, pm_callit_request)

Invoked upon a failed portmapper request to look up and call an RPC procedure. The script invokes pm_attempt with log_it=T unless the combination of the caller and the program are in suppress_pm_log. Finally, if the program called is *walld*, then the script adds the caller to suppress_pm_log.

pm_bad_port (r: connection, bad_p: count)

Invoked when a portmapper request or response includes an invalid port number. Since ports are represented by unsigned 4-byte integers, they can stray outside the allowed range of 0–65535 by being $\geq$ 65536. The script invokes conn_weird_addl with a *weird tag* of "bad_pm_port".

Field	Meaning
num_pkts	The number of packets sent by the endpoint, as seen by the monitor. The endpoint may have sent others that the network dropped upstream from the monitor.
num_rxmit	The number of packets retransmitted by the endpoint, as seen by the monitor.
num_rxmit_bytes	The number of bytes retransmitted by the endpoint.
num_in_order	The number of packets sent by the endpoint that arrived at the monitor in order, where "in order" means in the same order as sent by the endpoint, rather than in sequence number. (Thus, a retransmission can arrive in order, by this definition.) Bro determines if the packet arrived in order by applying heuristics to the IP identification (ID) field, which in general will increase by a small amount between successive packets transmitted by an endpoint.
num_OO	The number of packets sent by the endpoint that arrived at the monitor out of order. See the previous entry for the definition of "in order", and hence "out of order."
num_repl	The number of extra copies of packets sent by the endpoint that arrived at the monitor. Bro considers a packet replicated if its IP ID field is the same as for the previous packet it saw from the endpoint. Using this definition, a replication is most likely caused by a network mechanism such as duplication of a packet by a router, rather than a transport mechanism such as retransmission, though some TCPs fully reuse packets when retransmitting them, including their IP ID field.
endian_type	Whether the advance of the IP ID field as seen by the monitor was consistent with big-endian (network order) addition, little-endian, or undetermined. The three values are represented by the Bro constants `ENDIAN_BIG`, `ENDIAN_LITTLE`, and `ENDIAN_UNKNOWN`. In addition, the value can be `ENDIAN_CONFUSED`, meaning that the monitor saw conflicting evidence for little- and big-endian.

Table 7.7: `endpoint_stats` fields for summarizing connection endpoint statistics, all of type `count`.

7.21 The `analy` Analyzer

The `analy` analyzer provides a limited mechanism to use Bro to do statistical analysis on TCP connections. Its primary purpose is to demonstrate that Bro has applications to network traffic analysis beyond intrusion detection. It defines one event handler:

`conn_stats (c: connection, os: endpoint_stats, rs: endpoint_stats)`

Invoked for each connection when it terminates (for whatever reason). `os` and `rs` are the statistics for the originator endpoint and the responder endpoint, respectively; Table 7.21 gives the different record fields.

7.22 The `signature` Module

The `signature` module analyzes *signature matches* (see § 8, page 165). For each signature, you can specify one of the actions defined in Table 7.22. In addition, the module identifies two types of *exploit scans*: *horizontal* (a host triggers a signature for multiple destinations) and *vertical* (a host triggers multiple signature for the same destination).

The module handles one event:

`signature_match (state: signature_state, msg: string, data: string)`

Action	Meaning
SIG_IGNORE	Ignore the signature completely.
SIG_QUIET	Process for scan detection but don't report individually.
SIG_FILE	Write matches to *signatures-log*
SIG_LOG	Log matches and write them to *signatures-log*

Table 7.8: Possible actions to take for signatures matches. *signatures-log* defaults to open_log_file("signatures").

Invoked upon a match of a signature which contains an event action (§ 8.2.2, page 168).

It provides the following redefinable variables:

sig_actions : table[string] of count
Maps signature IDs to actions as defined in Table 7.22.

Default: SIG_FILE.

horiz_scan_thresholds : set[count]
Generate a log message whenever a remote host triggers a signature for the given number of hosts.

Default: { 5, 10, 50, 100, 500, 1000}

vert_scan_thresholds : set[count]
Generate a log message whenever a remote host triggers the given number of signatures for the same destination.

Default: { 5, 10, 50, 100, 500, 1000}

The module defines one function for external use:

has_signature_matched id: string, orig: addr, resp: addr): bool
Returns true if the given signature has already matched for the (originator,responder) pair.

7.23 The SSL Analyzer

The SSL analyzer processes traffic associated with the SSL (Secure Socket Layer) protocol versions 2.0 [SSLv2], 3.0 [SSLv30] and 3.1 [TLSv1]. SSL version 3.1 is also known as TLS (Transport Layer Security) version 1.0 since from that version onward the IETF has taken responsibility for further developement of SSL.

Bro instantiates an SSL analyzer for any connection with service ports 443/tcp (https), 563/tcp (nntps), 585/tcp (imap4-ssl), 614/tcp (sshell), 636/tcp (ldaps), 989/tcp (ftps-data), 990/tcp (ftps), 992/tcp (telnets), 993/tcp (imaps), 994/tcp (ircs), 995/tcp (pop3s), providing you have loaded the SSL analyzer, or defined a handler for one of the SSL events.

By default, the analyzer uses the above set of ports as a capture filter (§ 7.1.2, page 93). It currently checks the SSL handshake process for consistency, tries to verify seen certificates, generates several events, does connection logging, tries to detect security weaknesses, and produces simple statistics. It is also able to store seen certificates on disk. However, it does no decryption, so analysis is limited to clear text SSL records. This means that analysis stops in the

```
type x509: record {
    issuer:  string; # issuer name of the certificate
    subject: string; # subject name of the certificate
};
```

Figure 7.9: Definition of the `x509` record.

```
type ssl_connection_info: record {
    id: count;                      # the log identifier number
    connection_id: conn_id;         # IP connection information
    version: count;                 # version associated with connection
    client_cert: x509;
    server_cert: x509;
    id_index: string;               # index for associated sessionID
    handshake_cipher: count;        # cipher suite client and server agreed upon
};
```

Figure 7.10: Definition of the `ssl_connection_info` record.

middle of the handshaking phase for SSLv2 and at the end of it for SSLv3.0/SSLv3.1 (TLS). For this reason we have not implemented the SSL session caching mechanism (yet).

The analyzer consists of the four files: `ssl.bro`, `ssl-ciphers.bro`, `ssl-errors.bro`, and `ssl-alerts.bro`, which are accessed by `@load ssl`. The analyzer writes to the `weird` and `ssl` log files. The first receives all non-conformant and "weird" activity, while the latter tracks the SSL handshaking phase.

7.23.1 The `x509` record

This record is a very simplified structure for storing X.509 [X509] certificate information. It currently supports only the issuer and subject names.

7.23.2 The `ssl_connection_info` record

The main data structure managed by the `SSL` analyzer is a collection of `ssl_connection_info` records, where the record type is shown in Figure 7.23.2. The corresponding fields are *Fix me: the description here is out of date*:

`id`
: The unique connection identifier assigned to this connection. Connections are numbered starting at `1` and incrementing with each new connection.

`connection_id`
: The TCP connection which this SSL connection is based on.

`version`
: The SSL version number for this connection. Possible values are `SSLv20`, for SSL version 2.0, `SSLv30` for version 3.0, and `SSLv31` for version 3.1.

`client_cert`
: The information from the client certificate, if available.

`server_cert`
: The information from the server certificate, if available.

`id_index`
: Index into associated `SSL_sessionID_record` table.

`handshake_cipher`
: The cipher suite client and server agreed upon. *Note: For SSLv2 cached sessions, this is a placeholder (*`0xABCD`*).*

7.23.3 `SSL` variables

The standard script defines the following redefinable variables:

`ssl_compare_cipherspecs : bool`
: If true, remember the client and server cipher specs and perform additional tests. This costs an extra amount of memory (normally only for a short time) but enables detection of non-intersecting cipher sets, for example.

Default: `T`.

`ssl_analyze_certificates : bool`
: If true, analyze certificates seen in SSL connections, which includes the following steps:

- Generating a hash of the certificate and checking if we already saw it earlier from the current host. If so, we won't verify it, because we already did and verifying is a computational expensive process. If the certificate has changed for the current host, generate a weird event.
- Verify the certificate.
- Store of the certificate on disk in DER format.

Default: `T`.

`ssl_store_certificates : bool`
: If certificates are analyzed, this variable determines they should be stored on disk.

Default: `T`.

`ssl_store_cert_path : string`
: Path where certificates are stored. If empty, use the current directory. *Note: The path must not end with a slash!*

Default: `"../certs"`.

`ssl_verify_certificates : bool`
: If certificates are analyzed, wheter to verify them.

Default: `T`.

`x509_trusted_cert_path : string`
: Path where OpenSSL looks for trusted certificates. If empty, use the default OpenSSL path.

Default: `""`.

```
1046778101.534846 #1 192.168.0.98/32988 > 213.61.126.124/https start
1046778101.534846 #1 connection attempt version: 3.1
1046778101.534846 #1 cipher suites: SSLv3x_RSA_WITH_RC4_128_MD5 (0x4), SSLv3x_RSA_FIPS_WITH_
1046778101.753356 #1 server reply, version: 3.1
1046778101.753356 #1 cipher suite: SSLv3x_RSA_WITH_RC4_128_MD5 (0x4),
1046778101.762601 #1 X.509 server issuer: /C=DE/ST=Hamburg/L=Hamburg/O=TC TrustCenter for Se
1046778101.762601 #1 X.509 server subject: /C=DE/ST=Berlin/O=Lehmanns Fachbuchhandlung GmbH/
1046778101.894567 #1 handshake finished, version 3.1, cipher suite: SSLv3x_RSA_WITH_RC4_128_
1046778104.877207 #1 finish
---
Used cipher-suites statistics:
SSLv3x_RSA_WITH_RC4_128_MD5 (0x4): 1
```

Figure 7.11: Example of SSL log file with a single SSL session.

`ssl_max_cipherspec_size : count`

Maximum size in bytes for an SSL cipherspec. If we see attempted use of larger cipherspecs, warn and skip comparing it.

Default: `45`.

`ssl_store_key_material : bool`

If true, stores key material exchanged in the handshaking phase. *Note: This is mainly for decryption purposes and currently useless.*

Default: `T`.

In addition, `ssl_log` holds the name of the SSL log file to which Bro writes SSL connection summaries. It defaults to `open_log_file("ssl")`.

Figure 7.23.3 shows an example of how entries in the SSL log file look like. We see a transcript of the first SSL connection seen since Bro started running. The first line gives its start and the participating hosts and ports. Next, we see a client trying to attempt a SSL (Version 3.1) connection and the cipher suites offered. The server replies with a SSL 3.1 `SERVER-REPLY` and the desired cipher suite. *Note: In SSL v3.0/v3.1 this determines which cipher suite will be used for the connection*. Following this is the certificate the server sends, including the issuer and subject. Finally, we see that the handshaking phase for this SSL connection is finished now, and that client and server agreed on the cipher suite: `RSA_WITH_RC4_128_MD5`. Due to encryption, the SSL analyzer skips all further data. We only see the end of the connection. When Bro finishes, we get some statistics about the cipher suites used in all monitored SSL connections.

7.23.4 `SSL` event handlers

The standard script handles the following events:

`ssl_conn_attempt (c: connection, version: count, cipherSuites: cipher_suites_list)`

Invoked upon the client side of connection `c` when the analyzer sees a `CLIENT-HELLO` of SSL version `version` including the cipher suites the client offers `cipherSuites`.

The version can be `0x0002`, `0x0300` or `0x0301`. A new entry is generated inside the SSL connection table and the cipher suites are listed. Ciphers, that are known as weak (according to a corresponding table of weak ciphers) are logged inside the `weak.log` file. This also happens to cipher suites that we do not know yet. *Note: S*ee the file `ssl-ciphers.bro` for a list of known cipher suites.

`ssl_conn_server_reply (c: connection, version: count, cipherSuites: cipher_suites_list)`

This event is invoked upon the analyzer receiving a `SERVER-HELLO` of the SSL server. It contains the SSL version the server wishes to use (*Note: This finally determines, which SSL version will be used further*) and the cipher suite he offers. If it is SSL version 3.0 or 3.1, the server determines within this `SERVER-HELLO` the cipher suite for the following connection (so it will only be one). But if it's a SSL version 2.0 connection, the server only announces the cipher suites he supports and it's up to the client to decide which one to use.

Again, the cipher suites are listed and weak and unknown cipher suites are reported inside `weak.log`.

`ssl_certificate_seen ( c: connection, isServer: int )`

Invoked whenever we see a certificate from client or server but before verification of the certificate takes place. This may be useful, if you want to do something before certificate verification (e.g. do not verify certificates of some given servers).

`ssl_certificate ( c: connection, cert: x509, isServer: bool)`

Invoked after the certificate from server or client (`isServer`) has been verified. *Note: We only verify certificates once. If we see them again, we only check if they have changed!* `cert` holds the issuer and subject of the certificate, which gets stored inside this SSL connection's information record inside the SSL connection table and are written to `ssl.log`.

`ssl_conn_reused (c: connection, session_id: string)`

Invoked whenever a former SSL session is reused. `session_id` holds the session ID as string of the reused session and is written to `ssl.log`. Currently we don't do session tracking, because SSL version 2.0 doesn't send the session ID in clear text when it's generated.

`ssl_conn_established (c: connection, version: count, cipher_suite: count)`

Invoked when the handshaking phase of an SSL connection is finished. We see the used SSL version and the cipher suite that will be used for cryptography (written to `ssl.log`) if we have SSL version 3.0 or 3.1. In case of SSL version 2.0 we can only determine the used cipher suite for new sessions, not for reused ones. (*Note: In SSL version 3.0 and 3.1 the cipher suite to be used is already anounced in the* `SERVER-HELLO`.)

`ssl_conn_alert (c: connection, version: count, level: count, description: count)`

Invoked when the analyzer receives an SSL alert. The `level` of the alert (warning or fatal) and the `description` are written into `ssl.log`. (*Note: See* `ssl-alerts.bro`).

`ssl_conn_weak (name: string, c: connection)`

This event is called when the analyzer sees:

- weak ciphers (See: `ssl_conn_attempt`, `ssl_server_reply`, `ssl_conn_established`),
- unknown ciphers (See: `ssl_conn_attempt`, `ssl_server_reply`, `ssl_conn_established`)
- or certificate verification failed.

See `weak.bro`.

7.24 The `weird` Module

The `weird` module processes unusual or exceptional events. A number of these "shouldn't" or even "can't" happen, yet they do. The general design philosophy of Bro is to check for such events whenever possible, because they can reflect incorrect assumptions (either Bro's or the user's), attempts by attackers to confuse the monitor and evade detection, broken hardware, misconfigured networks, and so on.

Weird events are divided into three categories, namely those pertaining to: connections; flows (a pair of hosts, but for which a specific connection cannot be identified); and network behavior (cannot be associated with a pair of hosts). These categories have a total of four event handlers: `conn_weird`, `conn_weird_addl`, `flow_weird`, and `net_weird`, and in the corresponding sections below we catalog the events handled by each. In addition, we separately catalog the events generated by the standard scripts themselves (§ 7.24.8, page 163). Finally, two more weird events have their own handlers, in order to associate detailed information with the event: `rexmit_inconsistency` and `ack_above_hole`.

`weird_file` is the logging file that the module uses to record exceptional events. It defaults to `open_log_file("weird")`.

Note: While these events "shouldn't" happen, in reality they often do. For example, of the 73 listed below, a search of 10 months' worth of logs at LBNL shows that 42 were seen operationally. While some of the instances reflect attacks, the great majority are simply due to (i) *buggy implementations,* (ii) *diverse use of the network, or* (iii) *Bro bugs or limitations. Accordingly, you may initially be inclined to log each instance, but don't be surprised to find that you soon decide to only record many of them in the* `weird` *file, or not record them at all. (For further discussion, see the section on "crud" in [Pa99].)*

7.24.1 Actions for "weird" events

The general approach taken by the module is to categorize for each event the action to take when the event engine generates the event. Table 7.24.1 summarizes the different possible actions.

7.24.2 `weird` variables

The standard `weird` script provides the following redefinable variables:

Action	Meaning
WEIRD_UNSPECIFIED	No action specified.
WEIRD_IGNORE	Ignore the event.
WEIRD_FILE	Record the event to weird_file, if it has not been seen for these hosts before. (But see weird_do_not_ignore_repeats.)
WEIRD_LOG_ALWAYS	Record the event to weird_file and generate a real-time alert each time the event occurs.
WEIRD_LOG_ONCE	Record the event to weird_file; generate a real-time alert the first time the event occurs.
WEIRD_LOG_PER_CONN	Record the event to weird_file; generate a real-time alert the first time it occurs for a given connection.
WEIRD_LOG_PER_ORIG	Record the event to weird_file; generate a real-time alert the first time it occurs for a given originating host.

Table 7.9: Different types of possible actions to take for "weird" events.

weird_action : table[string] of count

Maps different weird events to actions as given in Table 7.24.1.

Default: as specified in conn_weird, conn_weird_addl, flow_weird, net_weird, and § 7.24.8, page 163. As usual, you can change particular values using refinement. For example:

```
redef weird_action: table[string] of count += {
    [["bad_TCP_checksum", "bad_UDP_checksum"]] = WEIRD_IGNORE,
    ["fragment_overlap"] = WEIRD_LOG_PER_CONN,
};
```

would specify to ignore TCP and UDP checksum errors (rather than the default of WEIRD_FILE), and to alert on fragment overlaps once per connection in which they occur, rather than the default of WEIRD_LOG_ALWAYS.

weird_action_filters : table[string] of function(c: connection): count

Indexed by the name of a weird event, yields a function that when called for a given connection exhibiting the event, returns an action from Table 7.24.1. A return value of WEIRD_UNSPECIFIED means "no special action, use the action you normally would." This variable thus allows arbitrary customization of the handling of particular events.

Default: empty, for the weird analyzer itself. The portmapper analyzer redefines this variable as follows:

```
    redef weird_action_filters += {
        [["bad_RPC", "excess_RPC", "multiple_RPCs", "partial_RPC"]] =
                RPC_weird_action_filter,
};
```

where RPC_weird_action_filter is a function internal to the analyzer that returns WEIRD_FILE if the originating host is in RPC_okay_nets, and WEIRD_UNSPECIFIED otherwise.

weird_ignore_host : set[addr, string]

Specifies that the analyzer should ignore the given weird event (named by the second index) if it involves the given address (as either originator or responder host).

Default: empty.

`weird_do_not_ignore_repeats : set[string]`
Gives a set of weird events that, if their action is `WEIRD_FILE`, should still be recorded to the `weird_file` each time they occur.

Default: the events relating to checksum errors, i.e., `"bad_IP_checksum"`, `"bad_TCP_checksum"`, `"bad_UDP_checksum"`, and `"bad_ICMP_checksum"`. These are recorded multiple times because it can prove handy to be able to track clusters of checksum errors.

7.24.3 `weird` functions

The `weird` analyzer includes the following functions:

`report_weird (t: time, name: string, id: string, action: count, no_log: bool)`
Processes an occurrence of the weird event `name` associated with the connection described by the string `id` (which may be empty if no connection is associated with the event). `action` is the action associated with the event. For `report_weird`, the only distinctions made between the different actions are that `WEIRD_IGNORE` causes the function to do nothing; any of `WEIRD_LOG_XXX` cause the function to log a message, unless `no_log` is true; and `WEIRD_UNSPECIFIED` causes the function to look up the action in `weird_action`. If the function does *not* find an action for the event, then it uses `WEIRD_LOG_ALWAYS` and prepends the log message with a pair of asterisks (“`**`”) to flag that this event does not have a specified action.

For `WEIRD_FILE`, `report_weird` only records the event once to the file, unless the given event is present in `weird_do_not_ignore_repeats`. Events with loggable actions are always recorded to `weird_file`.

`report_weird_conn (t: time, name: string, id: string, c: connection)`
Processes an occurrence of the weird event `name` associated with the connection `c`, which is described by the string `id`.

If `report_weird_conn` finds one of the hosts and the given event name in `weird_ignore_host`, then it does nothing. Then, if the event is in `weird_action`, then it looks up the event in `weird_action_filters` and invokes the corresponding function if present, otherwise taking the action from `weird_action`. It then implements the various flavors of `WEIRD_LOG_`*`XXX`* by not logging events more than once per connection, originator host, etc., though the events are still written to `weird_file`. Finally, the function invokes `report_weird` to do the actual recording and/or writing to `weird_file`.

`report_weird_orig (t: time, name: string, id: string, orig: addr)`
Processes an occurrence of the weird event `name` associated with the source address `orig`. `id` textually describes the flow from `orig` to the destination, for example using `endpoint_id`.

The function looks up the event name in `weird_action` and passes it along to `report_weird`.

7.24.4 Events handled by `conn_weird`

`conn_weird (name: string, c: connection)`
Invoked for most “weird” events. `name` is the name of the weird event, and `c` is the connection with which it’s associated.

`conn_weird` handles the following events, all of which have a default action of `WEIRD_FILE`:

`active_connection_reuse`
A new connection attempt (initial SYN) was seen for an already-established connection that has not yet terminated.

`bad_HTTP_reply`
The first line of a reply from an HTTP server did not include `HTTP/`*version*.

`bad_HTTP_version`
The first line of a request from an HTTP client did not include `HTTP/`*version*.

`bad_ICMP_checksum`
The checksum field in an ICMP packet was invalid.

`bad_rlogin_prolog`
The beginning of an Rlogin connection had a syntactical error.

`bad_RPC`
A Remote Procedure Call was ill-formed.

`bad_RPC_program`
A portmapper RPC call did not include the correct portmapper program number.

`bad_SYN_ack`
A TCP SYN acknowledgment (SYN-ack) did not acknowledge the sequence number sent in the initial SYN.

`bad_TCP_checksum`
A TCP packet had a bad checksum.

`bad_UDP_checksum`
A UDP packet had a bad checksum.

`baroque_SYN`
A TCP SYN was seen with an unlikely combination of other flags (the URGent pointer).

`blank_in_HTTP_request`
The URL in an HTTP request includes an embedded blank.

`connection_originator_SYN_ack`
A TCP endpoint that originated a connection by sending a SYN followed this up by sending a SYN-ack.

`data_after_reset`
After a TCP endpoint sent a RST to terminate a connection, it sent some data.

`data_before_established`
Before the connection was fully established, a TCP endpoint sent some data.

`excessive_RPC_len`
An RPC record sent over a TCP connection exceeded 8 KB.

`excess_RPC`
The sender of an RPC request or reply included leftover data beyond what the RPC parameters or result value themselves consumed.

FIN_advanced_last_seq

A TCP endpoint retransmitted a FIN with a higher sequence number than previously.

FIN_after_reset

A TCP endpoint sent a FIN after sending a RST.

FIN_storm

The monitor saw a flurry of FIN packets all sent on the same connection. A “flurry” is defined as 1,000 packets that arrived with less than 1 sec between successive FINs. *Deficiency: Clearly, this numbers should be user-controllable.*

HTTP_unknown_method

The method in an HTTP request was not GET, POST or HEAD.

HTTP_version_mismatch

A persistent HTTP connection sent a different version number for a subsequent item than it did initially.

inappropriate_FIN

A TCP endpoint sent a FIN before the connection was fully established.

multiple_HTTP_request_elements

An HTTP request included multiple methods.

multiple_RPCs

A TCP RPC stream included more than one remote procedure call.

NUL_in_line

A NUL (ASCII 0) was seen in a text stream that is expected to be free of NULs. Currently, the only such stream is that associated with an FTP control connection.

originator_RPC_reply

The originator (and hence presumed client) of an RPC connection sent an RPC reply (either instead of a request, or in addition to a request).

partial_finger_request

When a Finger connection terminated, it included a final line of unanalyzed text because the text was not newline-terminated.

partial_ftp_request

When an FTP connection terminated, it included a final line of unanalyzed text because the text was not newline-terminated.

partial_ident_request

When an IDENT connection terminated, it included a final line of unanalyzed text because the text was not newline-terminated.

partial_portmapper_request

A portmapper connection terminated with an unanalyzed request because the data stream was incomplete.

partial_RPC

An RPC was missing some required header information due to truncation.

`pending_data_when_closed`

A TCP connection closed even though not all of the data in it was analyzed due to a sequence hole.

`possible_split_routing`

Bro appears to be seeing only one direction of some bi-directional connections (§ 10.9, page 177). This can also occur due to certain forms of stealth-scanning.

`premature_connection_reuse`

A TCP connection tuple is being reused less than 30 sec after its previous use. (The standard requires waiting $2 \cdot \text{MSL} = 4$ minutes [RFC793, p. 27].)

`repeated_SYN_reply_wo_ack`

A TCP responder that replied to an initial SYN with a SYN-ack has subsequently sent a SYN *without* an acknowledgment.

`repeated_SYN_with_ack`

A TCP originator that sent an initial SYN has subsequently sent a SYN-ack.

`responder_RPC_call`

The responder (and hence presumed server) of an RPC connection sent an RPC request (either instead of a reply, or in addition to a reply).

`rlogin_text_after_rejected`

An Rlogin client sent additional text to an Rlogin server after the server already presumably rejected the client's service request.

`RPC_rexmit_inconsistency`

An RPC call was retransmitted, and the retransmitted call differed from the original call. This could reflect an attempt by an attacker to evade the monitor. *Note: This type of inconsistency checking is not available for RPC replies because the transmission of the reply in general marks the end of the RPC connection, and the monitor deletes the connection state shortly afterward.*

`RST_storm`

The monitor saw a flurry of RST packets all sent on the same connection. See `FIN_storm` for the definition of "flurry."

`RST_with_data`

A TCP RST packet included data. This actually is allowed by the specification [RFC1122, 4.2.2.12]. *Deficiency: This event should include the data.*

`simultaneous_open`

The monitor saw a TCP simultaneous open, i.e., both endpoints sent initial SYNs to one another at the same time. While the specification allows this [RFC793, p. 30], none of the protocols analyzed by Bro should be using it.

`spontaneous_FIN`

A TCP endpoint sent a FIN packet without sending any previous packets. This event can reflect stealth-scanning, but can also occur when Bro has recently started up and has not seen other traffic on a connection and hence does not know that the connection already exists.

`spontaneous_RST`
A TCP endpoint sent a RST packet without sending any previous packets. As with `spontaneous_FIN`, this event can reflect either stealth scanning or a Bro start-up transient.

`SYN_after_close`
A TCP endpoint sent a SYN (connection initiation) after sending a FIN (connection termination), but before the connection fully closed.

`SYN_after_partial`
A TCP endpoint in a "partial" connection (§ 10.12, page 177) sent a SYN.

`SYN_after_reset`
A TCP endpoint sent a SYN after sending a RST (reset connection).

`SYN_inside_connection`
A TCP endpoint sent a SYN during a connection (or partial connection) on which it had already sent data.

`SYN_seq_jump`
A TCP endpoint retransmitted a SYN or a SYN-ack, but with a different sequence number.

`SYN_with_data`
A TCP endpoint included data in a SYN packet it sent. Note, this can legitimately occur for T/TCP connections [RFC1644].

`TCP_christmas`
A TCP endpoint sent a SYN packet that included the RST flag (a nonsensical combination). The term "Christmas packet" has been used in this context (particularly if other flags are set, too) because the packet's flags are "lit up like a Christmas tree."

`UDP_datagram_length_mismatch`
The length field in a UDP header did not match the length field in the IP header. This could reflect an attempt by an attacker to evade the monitor.

`unpaired_RPC_response`
An RPC reply was seen for which no request was seen. This event could reflect a Bro start-up transient (it started running after the request was sent).

`unsolicited_SYN_response`
A TCP endpoint sent a SYN-ack without first receiving an initial SYN. This event could reflect a Bro start-up transient.

7.24.5 Events handled by `conn_weird_addl`

`conn_weird_addl (name: string, c: connection, addl: string)`
Invoked for a few "weird" events that require an extra (string) argument to help clarify the event. *Deficiency: It would likely be very handy if the general "weird" event handling was more flexible, with the ability to have various parameters associated with the events. Doing so will likely have to wait on general Bro mechanism for dealing with default parameters and/or polymorphic functions and event handlers.*

`conn_weird_addl` handles the following events, all of which have a default action of `WEIRD_FILE`:

`bad_ident_reply`
A reply from an IDENT server was syntactically invalid.

`bad_ident_request`
A request to an IDENT server was syntactically invalid.

`ident_request_addendum`
An IDENT request included additional text beyond that forming the request itself.

7.24.6 Events handled by `flow_weird`

`flow_weird (name: string, src: addr, dst: addr)`
is invoked for "weird" events that cannot be associated with a particular connection, but only with a pair of hosts, corresponding to a flow of packets from `src` to `dst`. Presently, all of these events deal with fragments.

`flow_weird` handles the following events:

`excessively_large_fragment`
A set of IP fragments reassembled to a maximum size exceeding 64,000 bytes. *Note: Sizes between 64,000 and 65,535 bytes are allowed, strictly speaking, but are highly unlikely in legitimate traffic. Sizes above 65,535 bytes generally represent attempted denial-of-service attacks, due to IP implementations that crash upon receiving such impossibly-large fragment sets.*

Default: `WEIRD_LOG_ALWAYS`.

`excessively_small_fragment`
A fragment other than the last fragment in a set was less than 64 bytes in size. *Note: The standard allows such small fragments, but their presence may reflect an attacker attempting to evade the monitor by splitting header information across multiple fragments.*

Default: `WEIRD_LOG_ALWAYS`.

`fragment_inconsistency`
A fragment overlaps with a previously sent fragment, and the two disagree on data they share in common. This event could reflect an attacker attempting to evade the monitor; it can also occur because Bro keeps previous fragments indefinitely (*Deficiency: it needs to provide a means for flushing old fragments, otherwise it becomes vulnerable to a state-holding attack*), and occasionally a fragment will overlap with one sent much earlier and long-since forgotten by the endpoints.

Default: `WEIRD_LOG_ALWAYS`.

`fragment_overlap`
A fragment overlaps with a previously sent fragment. As for `fragment_inconsistency`, this event can occur due to Bro keeping previous fragments indefinitely. This event does not in general reflect a possible attempt at evasion.

Default: `WEIRD_LOG_ALWAYS`.

`fragment_protocol_inconsistency`
Two fragments were seen for the same flow and IP ID which differed in their transport protocol (e.g., UDP, TCP). According to the specification, this is allowed [RFC791, p. 24], but its use appears highly unlikely.

Default: `WEIRD_FILE`, because it is difficult to see how an attacker can exploit this anomaly.

`fragment_size_inconsistency`

A "last fragment" was seen twice, and the two disagree on how large the reassembled datagram should be. This event could reflect an attacker attempting to evade the monitor.

Default: `WEIRD_FILE`, since it is more likely that this occurs due to a high volume flow of fragments wrapping the IP ID space than due to an actual attack.

`fragment_with_DF`

A fragment was seen with the "Don't Fragment" bit set in its header. While strictly speaking this is not illegal, and not impossible (a router could have fragmented a packet and then decided that the fragments should not be further fragmented), its presence is highly unusual.

Default: `WEIRD_FILE`, because it's difficult to see how this could reflect malicious activity.

`incompletely_captured_fragment`

A fragment was seen whose length field is larger than the fragment datagram appearing on the monitored link.

Default: `WEIRD_LOG_ALWAYS`.

7.24.7 Events handled by `net_weird`

`net_weird (name: string)`

is invoked for "weird" events that cannot be associated with a particular connection or set of hosts. Except as noted, the default action for all such events is `WEIRD_FILE`.

`net_weird` handles the following events:

`bad_IP_checksum`

A packet had a bad IP header checksum.

`bad_TCP_header_len`

The length of the TCP header (which is itself specified in the header) was smaller than the minimum allowed size.

`internally_truncated_header`

A captured packet with a valid IP length field was smaller as actually recorded, such that the captured version of the packet was illegally small. This event may reflect an error in Bro's packet capture hardware or software.

Default: `WEIRD_LOG_ALWAYS`, because this event can indicate a basic problem with Bro's packet capture.

`truncated_IP`

A captured packet either was too small to include a minimal IP header, or the full length as recorded by the packet capture library was smaller than the length as indicated by the IP header.

`truncated_header`

An IP datagram's header indicates a length smaller than that required for the indicated transport type (TCP, UDP, ICMP).

7.24.8 Events generated by the standard scripts

The following events are generated by the standard scripts themselves:

`bad_pm_port`

See `pm_bad_port`. Handled by `conn_weird_addl`, where the extra parameter is the text `"port <`*bad-port*`>"`.

`Land_attack`

A TCP connection attempt was seen with identical initiator and responder addresses and ports. This event likely reflects an attempted denial-of-service attack known as a "Land" attack. See `check_spoof`. Handled by `conn_weird`.

7.24.9 Additional handlers for "weird" events

In addition to the above, generalized events, Bro includes two specific events that are defined by themselves so they can include additional parameterization:

`rexmit_inconsistency (c: connection, t1: string, t2: string)`

Invoked when a retransmission associated with connection `c` differed in its data from the contents transmitted previously. `t1` gives the original data and `t2` the different retransmitted data.

This event may reflect an attacker attempting to evade the monitor. Unfortunately, however, experience has shown that *(i)* inconsistent retransmissions do in fact happen due to (appalling) TCP implementation bugs, and *(ii)* once they occur, they tend to cascade, because often the source of the bug is that the two endpoints have become desynchronized.

The handler logs the message in the format `"`*id* `rexmit inconsistency (<t1>) (<t2>)"`. However, the handler only logs the first instance of an inconsistency, due to the cascade problem mentioned above.

Deficiency: The handler is not told which of the two connection endpoints was the faulty transmitter.

`ack_above_hole (c: connection, t1: string, t2: string)`

Invoked when Bro sees a TCP receiver acknowledge data above a sequence hole. In principle, this should never occur. Its presence generally means one of two things: *(i)* a TCP implementation with an appalling bug (these definitely exist), or *(ii)* a packet drop by Bro's packet capture facility, such that it never saw the data now being acknowledged.

Because of the seriousness of this latter possibility, the handler logs a message `"ack above a hole"`. *Note: You can often distinguish between a truly broken TCP acknowledgment and Bro dropping packets by the fact that in the latter case you generally see a cluster of ack-above-a-hole messages among otherwise unrelated connections.*

Deficiency: The handler is not told which of the two connection endpoints sent the acknowledgment.

7.25 The `icmp` Analyzer

7.26 The `stepping` Analyzer

7.27 The `ssh-stepping` Module

7.28 The `backdoor` Analyzer

7.29 The `interconn` Analyzer

Chapter 8

Signatures

8.1 Overview

In addition to the policy language, Bro provides another language which is specifally designed to define *signatures*. Signatures precisly describe how network traffic looks for certain, well-known attacks. As soon as a attack described by a signature is recognized, Bro may generate an event for this *signature match* which can then be analyzed by a policy script. To define signatures, Bro's language provides several powerful constructs like regular expressions§ 3.6, page 25 and dependencies between multiple signatures.

Signatures are independent of Bro's policy scripts and, therefore, are put into their own file(s). There two ways to specify which files contain signatures: By using the `-s` flag when you invoke Bro, or by extending the Bro variable `signatures_files` using the `+=` operator. If a signature file is given without a path, it is searched along `$BROPATH`. The default extension of the file name is `.sig` which Bro appends automatically.

8.2 Signature language

Each individual signature has the format

`signature` *`id`* { *`attribute-set`* }

`id` is an unique label for the signature. There are two types of attributes: *conditions* and *actions*. The conditions define *when* the signature matches, while the actions declare *what to do* in the case of a match. Conditions can be further divided into four types: *header*, *content*, *dependency*, and *context*. We will discuss these in more detail in the following subsections.

This is an example of a signature:

```
signature formmail-cve-1999-0172 {
  ip-proto == tcp
  dst-ip == 1.2.0.0/16
  dst-port = 80
  http /.*formmail.*\?.*recipient=[^&]*[;|]/
  event "formmail shell command"
  }
```

8.2.1 Conditions

Header conditions

Header conditions limit the applicability of the signature to a subset of traffic that contains matching packet headers. For TCP, this match is performed only for the first packet of a connection. For other protocols, it is done on each individual packet. There are pre-defined header conditions for some of the most used header fields:

`dst-ip` *`comp address-list`*
: Destination address of IP packet (may include CIDR masks for specifying networks)

`dst-port` *`comp integer-list`*
: Destination port of TCP or UDP packet

`ip-proto` *`comp protocol-list`*
: IP protocol; *protocol* may be `tcp`, `udp`, or `icmp`.

`src-ip` *`comp address-list`*
: Source address of IP packet (may include CIDR masks for specifying networks)

`src-port` *`comp integer-list`*
: Source port of TCP or UDP packet

comp is one of `==`, `!=`, `<`, `<=`, `>`, `>=`. All lists are comma-separated values of the given type which are sequentially compared against the corresponding header field. If at least one of the comparisions evaluates to true, the whole header condition matches (exception: if *comp* is `!=`, the header condition only matches if *all* values differ). *address* is an dotted IP address optionally followed by a CIDR/mask to define a subnet instead of an individual address. *protocol* is either one of `ip`, `tcp`, `udp` and `icmp`, or an integer.

In addition to this pre-defined short-cuts, a general header condition can be defined either as

```
header proto[offset:size] comp value-list
```

or as

```
header proto[offset:size] & integer comp value-list
```

This compares the value found at the given position of the packet header with a list of values. *offset* defines the position of the value within the header of the protocol defined by *proto* (which can `ip`, `tcp`, `udp` or`icmp`. *size* is either 1, 2, or 4 and specifies the value to have a size of this many bytes. If the optinal `&` *`integer`* is given, the packet's value is first masked with the *integer* before it is compared to the value-list. *comp* is one of `==`, `!=`, `<`, `<=`, `>`, `>=`. *value-list* is a list of comma-separated integers similar to those described above. The integers within the list may be followed by an additional *`/mask`* where *mask* is a value from 0 to 32. This correponds to the CIDR notation for netmasks and is translated into a corresponding bitmask which is applied to the packet's value prior to the comparision (similar to the optional `&` *`integer`*).

Putting all together, this is an example which is aequivalent to `dst-ip == 1.2.3.4/16, 5.6.7.8/24`:

```
header ip[16:4] == 1.2.3.4/16, 5.6.7.8/24
```

Content conditions

Content conditions are defined by regular expressions. We differentiate two kinds of content conditions: first, the expression may be declared with the `payload` statement, in which case it is matched against the raw payload of a connection (for reassembled TCP streams) or of a each packet. Alternatively, it may be prefixed with an analyzer-specific label, in which case the expression is matched against the data as extracted by the corresponding analyzer.

A `payload` condition has the form

`payload` */regular expression/*

Currently, the following analyzer-specific content conditions are defined (note that the corresponding analyzer has to be activated by loading its policy script):

`http-request` */regular expression/*
: The regular expression is matched against decoded URIs of the HTTP requests.

`http-request-header` */regular expression/*
: The regular expression is matched against client-side HTTP headers.

`http-reply-header` */regular expression/*
: The regular expression is matched against server-side HTTP headers.

`ftp` */regular expression/*
: The regular expression is matched against the command line input of FTP sessions.

`finger` */regular expression/*
: The regular expression is matched against the finger requests.

For example, `http /(etc(passwd|shadow)/` matches any URI containing either `etc/passwd` or `etc/shadow`.

Dependency conditions

To define dependencies between different signatures, there are two conditions:

`requires-signature [!` *id*]
: Defines the current signature to match only if the signature given by *id* matches for the same connection. Using '`!`' negates the condition: The current signature only matches if *id* does not match for the same connection (this decision is necessarily deferred until the connection terminates).

`requires-reverse-signature [!` *id*]
: Similar to `requires-signature`, but *id* has to match for the other direction of the same connections than the current signature. This allows to model the notion of requests and replies.

Context conditions

Context conditions pass the match decision on to various other components of Bro. They are only evaluated if all other conditions have already matched. The following context conditions are defined:

```
type signature_state: record {
    id: string;          # ID of the signature
    conn: connection;    # Current connection
    is_orig: bool;       # True if current endpoint is originator
    payload_size: count; # Payload size of the first pkt of curr. endpoint
    };
```

Figure 8.1: Definition of the `signature_state` record.

`eval` *`policy function`*

The given policy function is called and has to return a boolean indicating the match result. The function has to be of the type `function cond(state: signature_state): bool`. See Figure 8.2.1 for the definition of `signature_state`.

`ip-options`

Not implemented currently.

`payload-size` *`comp integer`*

Compares the integer to the size of the payload of a packet. For reassembled TCP streams, the integer is compared to the size of the first in-order payload chunk. Note that the latter is not well defined.

`same-ip`

Evaluates to true if the source address of the IP packets equals its destination address.

`tcp-state` *`state-list`*

Poses restrictions on the current TCP state of the connection. *state-list* is a comma-separated list of `established` (the three-way handshake has already been performed), `originator` (the current data is send by the originator of the connection), and `responder` (the current data is send by the responder of the connection).

8.2.2 Actions

Actions define what to do if a signature matches. Currently, there is only one action defined: `event` *`string`* raises a `signature_match` event. The event handler has the following type:

```
event signature_match(state: signature_state, msg: string, data:
string)
```

See Figure 8.2.1 for a description of `signature_state`. The given string is passed as `msg`, and data is the current part of the payload that has eventually lead to the signature match (this may be empty for signatures without content conditions).

8.3 snort2bro

The open-source IDS Snort provides an extensive library of signatures. The Python script snort2bro converts Snort's signature into Bro signatures. Due to different internal architectures of Bro and Snort, it is not always possible to keep the exact semantics of Snort's signatures, but most of the time it works very well.

To convert Snort signatures into Bro's format, `snort2bro` needs a workable Snort configuration file (`snort.cfg`) which, in particular, defines the variables used in the Snort signatures (usally things like `$EXTERNAL_NET` or `$HTTP_SERVERS`). The conversion is performed by calling `snort2bro [-I` *`dir`*`] snort.cfg` where the directory optionally given by `-I` contains the files imported by Snort's `include` statement. The converted signature set is written to standard output and may be redirected to a file. This file can then be evaluated by Bro using the `-s` flag or the `signatures_files` variable.

Deficiency: `snort2bro` does not know about some of the newer Snort signature options and ignores them (but it gives a warning).

Chapter 9

Interactive Debugger

9.1 Overview

Bro's interactive debugger is intended to aid in the development, testing, and maintenance of policy scripts. The debugger's interface has been modeled after the popular gdb debugger; the command syntax is virtually identical. While at present the Bro debugger supports only a small subset of gdb's features, these were chosen to be the most commonly used commands. Additional features beyond those of gdb, such as wildcarding, have been added to specifically address needs created by Bro policy scripts.

9.2 A Sample Session

The transcript below should look very familiar to those familiar with gdb. The debugger's command prompt accepts debugger commands; before each prompt, the line of policy code that is next to be executed is displayed.

First we activate the debugger with the -d command-line switch.

```
bobcat:˜/bro/bro$ ./bro -d -r slice.trace mt
Policy file debugging ON.
In bro_init() at policy/ftp.bro:437
437             have_FTP = T;
```

Next, we set a breakpoint in the connection_finished event handler [reference this somehow]. A breakpoint causes the script's execution to stop when it reaches the specified function. In this case, there are many event handlers for the connection_finished event, so we are given a choice.

```
(Bro [0]) break connection_finished
Setting breakpoint on connection_finished:

There are multiple definitions of that event handler.
Please choose one of the following options:
[1] policy/conn.bro:268
[2] policy/active.bro:14
[3] policy/ftp.bro:413
```

```
[4] policy/demux.bro:40
[5] policy/login.bro:496
[a] All of the above
[n] None of the above
Enter your choice: 1
Breakpoint 1 set at connection_finished at policy/conn.bro:268
```

Now we resume execution; when the breakpoint is reached, execution stops and the debugger prompt returns.

```
(Bro [1]) continue
Continuing.
Breakpoint 1, connection_finished(c = ’[id=[orig_h=1.0.0.163,
orig_p=2048/tcp, resp_h=1.0.0.6, resp_p=23/tcp], orig=[size=0,
state=5], resp=[size=46, state=5], start_time=929729696.316166,
duration=0.0773319005966187, service=, addl=, hot=0]’) at
policy/conn.bro:268
In connection_finished(c = ’[id=[orig_h=1.0.0.163, orig_p=2048/tcp,
resp_h=1.0.0.6, resp_p=23/tcp], orig=[size=0, state=5], resp=[size=46,
state=5], start_time=929729696.316166, duration=0.0773319005966187,
service=, addl=, hot=0]’) at policy/conn.bro:268
268             if ( c$orig$size == 0 || c$resp$size == 0 )
```

We now step through a few lines of code and into the `record_connection` call.

```
(Bro [2]) step
274             record_connection(c, "finished");
(Bro [3]) step
In record_connection(c = ’[id=[orig_h=1.0.0.163, orig_p=2048/tcp,
resp_h=1.0.0.6, resp_p=23/tcp], orig=[size=0, state=5], resp=[size=46,
state=5], start_time=929729696.316166, duration=0.0773319005966187,
service=, addl=, hot=0]’, disposition = ’finished’) at
policy/conn.bro:162
162             local id = c$id;
(Bro [4]) step
163             local local_init = to_net(id$orig_h) in local_nets;
```

We now print the value of the `id` variable, which was set in the previously executed statement `local id = c$id;`. We follow that with a backtrace (`bt`) call, which prints a trace of the currently-executing functions and event handlers (along with their actual arguments). We then remove the breakpoint and continue execution to its end (the remaining output has been trimmed off).

```
(Bro [5]) print id
[orig_h=1.0.0.163, orig_p=2048/tcp, resp_h=1.0.0.6, resp_p=23/tcp]
(Bro [6]) bt
#0 In record_connection(c = ’[id=[orig_h=1.0.0.163, orig_p=2048/tcp,
 resp_h=1.0.0.6, resp_p=23/tcp], orig=[size=0, state=5],
 resp=[size=46, state=5], start_time=929729696.316166,
 duration=0.0773319005966187, service=, addl=, hot=0]’, disposition =
 'finished') at policy/conn.bro:163
#1 In connection_finished(c = '[id=[orig_h=1.0.0.163, orig_p=2048/tcp,
 resp_h=1.0.0.6, resp_p=23/tcp], orig=[size=0, state=5],
 resp=[size=46, state=5], start_time=929729696.316166,
 duration=0.0773319005966187, service=, addl=, hot=0]') at
 policy/conn.bro:274
(Bro [7]) delete
Breakpoint 1 deleted
(Bro [8]) continue
Continuing.
...
```

9.3 Usage

The Bro debugger is invoked with the `-d` command-line switch. It is strongly recommended that the debugger be used with a tcpdump capture file as input (the `-r` switch) rather than in "live" mode, so that results are repeatable.

Execution tracing is a feature which generates a complete record of which code statements are executed during a given run. It is enabled with the `-t` switch, whose argument specifies a file which will contain the trace.

Debugger commands all are a single word, though many of them take additional arguments. Commands may be abbreviated with a prefix (e.g., `fin` for `finish`); if the same prefix matches multiple commands, the debugger will list all that match. Certain very frequently-used commands, such as `next`, have been given specific one-character shortcuts (in this case, `n`). For more details on all the debugger commands, see the Reference in section 9.5, below.

The debugger's prompt can be activated in three ways. First, when the `-d` switch is supplied, Bro stops in the `bro_init` initialization function (more precisely, after global-scope code has been executed; see section 9.4). It is also activated when a breakpoint is hit. Breakpoints are set with the `break` command (see the Reference). The final way to invoke the debugger's prompt is to interrupt execution by pressing Ctrl-C (sending an Interrupt signal to the process). Execution will be suspended after the currently-executing line is completed.

9.4 Notes and Limitations

- Statements at global scope, i.e., those executed before the `bro_init` function, may not be debugged at present. This is because those statements load declarations for other functions needed for the debugger to function properly.

9.5 Reference

Summary of Commands

Note: all commands may be abbreviated with a unique prefix. Shortcuts below are special exceptions to this rule.

Command	Shortcut	Description
help		Get help with debugger commands
quit		Exit Bro
next	n	Step to the following statement, skipping function calls
step	s	Step to following statements, stepping in to function calls
continue	c	Resume execution of the policy script
finish		Run until the currently-executing function completes
break	b	Set a breakpoint
condition		Set a condition on an existing breakpoint
delete	d	Delete the specified breakpoints; delete all if no arguments
disable		Turn off the specified breakpoint; do not delete permanently
enable		Undo a prior ‘disable’ command
info		Get information about the debugging environment
print	p	Evaluate an expression and print the result
set		Alias for ‘print’
backtrace	bt	Print a stack trace
frame		Select frame number N
up		Select the stack frame one level up from the current one
down		Select the stack frame one level down from the current one
list	l	Print source lines surrounding specified context
trace		Turn on or off execution tracing

Getting Help

help Help for each command may be invoked with the `help` command. Calling the command with no arguments displays a one-line summary of each command.

Command-Line Options

`-d` switch The `-d` switch enables the Bro script debugger.

`-t` switch The `-t` enables execution tracing. There is an argument to the switch, which indicates a file that will contain the result of the trace. Trace output consists of the source code lines executed, indented for each nested function invocation.

Example. The following command invokes Bro, using `tcpdump_file` for the input packets and outputting the result of the trace to `execution_trace`.

```
./bro -t execution_trace -r tcpdump_file policy_script.bro
```

Example. If the argument to `-t` is a single dash character (“`-`”), then the trace output is sent to `stderr`.

```
./bro -t - -r tcpdump_file policy_script.bro
```

Example. Lastly, execution tracing may be combined with the debugger. Here we send output to `stderr`, so it will be intermingled with the debugger's output. Tracing may be turned off and on in the debugger using the `trace` command.

```
./bro -d -t - -r tcpdump_file policy_script.bro
```

Running the Script

quit Exit Bro, aborting execution of the currently executing script.

restart (r) *(Currently Unimplemented)* Restart the execution of the script, rewinding to the beginning of the input file(s), if appropriate. Breakpoints and other debugger state are preserved.

continue (c) Resume execution of the script file. The script will continue running until interrupted by a breakpoint or a signal.

next (n) Execute one statement, without entering any subroutines called in that statement.

step (s) Execute one statement, but stop on entry to any called subroutine.

finish Run until the currently executing function returns.

Breakpoints

break (b) Set a breakpoint. A breakpoint suspend execution when execution reaches a particular location and returns control to the debugger. Breakpoint locations may be specified in a number of ways:

`break`	With no argument, the current line is used.
`break` *[FILE:]LINE*	The specified line in the specified file; if no policy file is specified, the current file is implied.
`break` *FUNCTION*	The first line of the specified function or event handler. If more than one event handler matches the name, a choice will be presented.
`break` *WILDCARD*	Similar to *FUNCTION*, but a POSIX-compliant regular expression (see the `regex(3)` man page)is supplied, which is matched against all functions and event handlers. One exception to the the POSIX syntax is that, as in the shell, the `*` character may be used to match zero or more of any character without a preceding period character (`.`).

condition N ***expression*** The numeric argument N indicates which breakpoint to add a condition to, and the expression is the conditional expression. A breakpoint with a condition will only stop execution when the supplied condition is true. The condition will be evaluated in the context of the breakpoint's location when it is reached.

enable With no arguments, enable all breakpoints previously disabled with the `disable` command. If numeric arguments separated by spaces are provided, the breakpoints with those numbers will be enabled.

disable With no arguments, disable all breakpoints. Disabled breakpoints will not stop execution, but will be retained to be enabled later. If numeric arguments separated by spaces are provided, the breakpoints with those numbers will be disabled.

delete (d) With no arguments, permanently delete all breakpoints. If numeric arguments separated by spaces are provided, the breakpoints with those numbers will be deleted.

Debugger State

info Give information about the current script and debugging environment. A subcommand should follow the `info` command to indicate which information is desired. At present, the following subcommands are available:

`info break`	List all breakpoints and their status

Inspecting Program State

print (p) / set The `print` command and its alias, `set`, are used to evaluate any expression in the policy script language. The result of the evaluation is printed out. Results of the evaluation affect the current execution environment; expressions may include things like assignment. The expression is evaluated in the context of the currently selected stack frame. The `frame`, `up`, and `down` commands (below) are used to change the currently selected frame, which defaults to the innermost one.

backtrace (bt) Print a description of all the stack frames (function invocations) of the currently executing script.
With no arguments, prints out the currently selected stack frame.
With a numeric argument $\pm N$, prints the innermost N frames if the argument is positive, or the outermost N frames if the argument is negative.

frame With no arguments, prints the currently selected frame.
With a numeric argument N, selects frame N. Frame numbers are numbered inside-out from 0; the

up Select the stack frame that called the currently selected one. If a numeric argument N is supplied, go up that many frames.

down Select the stack frame called by the currently selected one. If a numeric argument N is supplied, go down that many frames.

list (l) With no argument, print the ten lines of script source code following the previous listing. If there was no previous listing, print ten lines surrounding the next statement to be executed. If an argument is supplied, ten lines are printed around the location it describes. The argument may take one of the following forms:

[FILE:]LINE	The specified line in the specified file; if no policy file is specified, the current file is implied.
FUNCTION	The first line of the specified function or event handler. If more than one event handler matches the name, a choice will be presented.
$\pm N$	With a numeric argument preceded by a plus or minus sign, the line at the supplied offset from the previously selected line.

Chapter 10

Missing Documentation

This chapter holds stubs for subjects that have yet to be documented. Some of these are actually already somewhat covered elsewhere in the manual. In addition, a major missing piece for the manual is the Bro language itself; below we mention some Bro language topics that come up elsewhere in the current version of the manual.

10.1 The use of *prefixes*

10.2 The tcpdump save file that Bro writes

10.3 The `bro.init` initialization file

10.4 Assignment operators such as +=

10.5 The notion of redefinition/refinement

10.6 The logging model

10.7 Timer management

10.8 SYN-FIN filtering

10.9 Split routing

10.10 Scan dropping

10.11 Operator precedence

10.12 Partial connections

10.13 Packet drops

10.14 The `@load` directive

10.15 Global statements

10.16 Inserting tables into tables

10.17 Demultiplexing

10.18 Bro init file

10.19 Hostnames vs. addresses

10.20 The hot-report script

10.21 Use of libpcap/BPF

[MJ93, MLJ94]

10.22 The problem of evasion

[PN98]

10.23 Backscatter

10.24 Playing back traces

10.25 Discarders

10.26 Differences between this release and the previous one

10.27 Alert cascade

10.28 The need for subtyping

E.g., `src addr` vs. `dst addr`, perhaps using attributes.

10.29 The need for CIDR masks

10.30 The wish list

10.31 Known bugs

Bibliography

[RFC2373] R. Hinden and S. Deering, “IP Version 6 Addressing Architecture,” RFC 2373, Jul. 1998.

[MJ93] S. McCanne and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture,” *Proc. 1993 Winter USENIX Conference*, San Diego, CA.

[MLJ94] S. McCanne, C. Leres and V. Jacobson, `libpcap`, available via anonymous ftp to ftp.ee.lbl.gov, 1994.

[Pa98] V. Paxson, “Bro: A System for Detecting Network Intruders in Real-Time,” Proc. 7th USENIX Security Symposium, Jan. 1998.

[Pa99] V. Paxson, “Bro: A System for Detecting Network Intruders in Real-Time,” *Computer Networks* special issue on intrusion detection, 31(23–24), pp. 2435–2463, Dec. 1999.

[PN98] T. Ptacek and T. Newsham, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection,” Secure Networks, Inc., http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps, Jan. 1998.

[RFC791] J. Postel, “Internet Protocol,” RFC 791, Sep. 1981.

[RFC793] J. Postel, “Transmission Control Protocol,” RFC 793, Sep. 1981.

[RFC854] J. Postel and J. Reynolds, “Telnet Protocol Specification,” RFC 854, May 1983.

[RFC855] J. Postel and J. Reynolds, “Telnet Option Specifications,” RFC 855, May 1983.

[RFC959] J. Postel and J. Reynolds, “File Transfer Protocol (FTP),” RFC 959, Oct. 1985.

[RFC1013] R. Scheifler, “X Window System Protocol, version 11: Alpha update,” RFC 1013, Apr. 1987.

[RFC1094] Sun Microsystems, “NFS: Network File System Protocol specification,” RFC 1094, Mar. 1989.

[RFC1122] B. Braden, “Requirements for Internet hosts - communication layers,” RFC 1122, Oct. 1989.

[RFC1282] B. Kantor, “BSD Rlogin,” RFC 1282, Dec. 1991.

[RFC1288] D. Zimmerman, “The Finger User Information Protocol,” RFC 1288, Dec. 1991.

[RFC1413] M. St. Johns, “Identification Protocol,” RFC 1413, Jan. 1993.

[RFC1644] B. Braden, “T/TCP – TCP Extensions for Transactions Functional Specification,” RFC 1644, Jul. 1994.

[RFC1813] B. Callaghan, B. Pawlowski, P. Staubach, “NFS Version 3 Protocol Specification,” RFC 1813, June 1995.

[RFC1831] R. Srinivasan, “RPC: Remote Procedure Call Protocol Specification Version 2,” RFC 1831, Aug. 1995.

[RFC1832] R. Srinivasan, “XDR: External Data Representation Standard,” RFC 1832, Aug. 1995.

[RFC1939] J. Myers and M. Rose, “Post Office Protocol - Version 3,” RFC 1939, May 1996.

[RFC1945] T. Berners-Lee, R. Fielding and H. Frystyk, “Hypertext Transfer Protocol – HTTP/1.0,” RFC 1945, May 1996.

[RFC2616] J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1,” RFC 2626, Jun. 1999.

[YKSRL00] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne and S. Lehtinen, “SSH Connection Protocol,” Internet Draft *draft-ietf-secsh-connect-07.txt*, May 2000.

[SSLv2] Kipp E.B. Hickman, “The SSL Protocol,” Netscape Communications Corp. *http://wp.netscape.com/eng/security/SSL_2.html*, February 1995.

[SSLv30] Alan O. Freier, Philip Karlton, Paul C. Kocher, “The SSL Protocol Version 3.0,” Internet Draft *draft-freier-ssl-version3-02.txt*, November 1996.

[TLSv1] T. Dierks, C. Allen, “ The TLS Protocol Version 1.0,” RFC 2246, January 1999.

[SSL-FIPS] Nelson Bolyard, Wan-Teh Chang, “FIPS SSL CipherSuites,” *http://www.mozilla.org/projects/security/pki/nss/ssl/fips-ssl-ciphersuites.html*, June 2001.

[SSL-AES] P. Chown, “Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS),” RFC 3268, June 2002.

[TLS-56] John Banes, Richard Harrington, “56-bit Export Cipher Suites For TLS,” Internet Draft *draft-ietf-tls-56-bit-ciphersuites-00.txt*, April 1999.

[X509] R. Housley, W. Polk, W. Ford, D. Solo, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 3280, June 2002.

Index

! operator, 46
!in operator, 51
%modulus operator, 23
|| short-circuit “or”, 22, 47
&& short-circuit “and”, 22, 47
! “not” operator, 22
(operator, 50
) operator, 50
+ operator, 47
++ operator, 46
+ addition operator, 23, 28, 47
+ unary operator, 23
- subtraction operator, 23, 28, 47
- unary operator, 23, 27
* multiplication operator, 23, 28, 47
/ division operator, 23, 28, 47
+= operator, 93
- operator, 46
-- operator, 46
: operator, 48
= operator, 48
== equality operator, 24, 26, 47, 51
!˜ exact match negation, 26
˜ exact pattern match operator, 26
!= inequality operator, 24, 26, 47, 51
< less-than operator, 24, 48
<= less-or-equal operator, 24, 48
> greater-than operator, 24, 48
>= greater-or-equal operator, 24, 48
? operator, 48
?$ record field test, 52
[operator, 50, 52
$ record field access operator, 52
$ record field access operator, 32
% format, 87
\a alert escape, 24
\a bell escape, 24
\b backspace escape, 24
\f formfeed escape, 24
\n newline escape, 24
%modulus operator, 47
\r carriage return escape, 24
\t tab escape, 24
] operator, 50, 52
4Dgifts username, 121

aborted execution, 18
absolute time, 27
access
  allowable /16 network pairs, 107
  allowable address pairs, 107
  allowable services, 108
  fatal inbound services, 109
  forbidden attempted services, 109
  forbidden inbound services, 109
  forbidden services, 109
  sensitive /24 destination networks, 108
  sensitive /24 source networks, 107
  sensitive destination addresses, 107
  sensitive source addresses, 107
  service allowed to a particular host, 108
  service allowed to particular host pairs, 108
account_tried event, 114
accounts_tried variable, 77
ack above a hole (possible packet drop message), 163
ack_above_hole event, 95, 163
acknowledgment holes, 163
actions, 149, 155
  SIG_FILE, 149
  SIG_IGNORE, 149
  SIG_LOG, 149
  SIG_QUIET, 149
  WEIRD_FILE, 155
  WEIRD_IGNORE, 155
  WEIRD_LOG_ALWAYS, 155
  WEIRD_LOG_ONCE, 155
  WEIRD_LOG_PER_CONN, 155
  WEIRD_LOG_PER_ORIG, 155
  WEIRD_UNSPECIFIED, 155
activating_encryption event, 142
active module, 117
active_conn variable, 57, 117
active_connection function, 85
active_connection_reuse (“weird” event), 156
active_file function, 85
actually_rejected_PTR_anno variable, 64
add keyword, 45
add statement, 45
&add_func attribute, 56
add_interface function, 85
add_tcpdump_filter function, 85
addition

  - numeric, 23
  - temporal, 28
- additional information associated with a connection, 98, 103
- `addl`, 98
  - `connection` field, 98
- `addl_web` variable, 77, 113
- `addr`, *see* types, `addr`
- address masking, 29, 88, 91
- address scanning, 112
- address type, 29
  - constants, 29
  - operators, 29
- addresses
  - hot destinations, 107
  - hot sources, 107
  - in a connection, 101
  - local, 105–106
  - mapping to hostnames, 20
  - neighbor, 106
- `addrs`, 119
  - `dns_mapping` field, 119
- `alert_action_filters` variable, 57
- `alert_file` variable, 57
- `allow_16_net_pairs` variable, 67, 107
- `allow_excessive_ntp_requests` variable, 74
- `allow_pairs` variable, 67, 107
- `allow_PTR_scans` variable, 65
- `allow_service_pairs` variable, 67
- `allow_services` variable, 67, 108
- `allow_services_pairs` variable, 108
- `allow_services_to` variable, 67, 108
- `allow_spoof_services` variable, 67, 107
- allowable /16 network pairs, 107
- allowable address pairs, 107
- altering log files, 133
- `always_hot_ids` variable, 68, 121
- `always_hot_login_ids` variable, 74, 136
- `analy` analyzer, 148
- analysis
  - bidirectional vs. unidirectional, 159
  - off-line, 16, 17, 89, 105
  - on-line, 16, 17, 84, 89, 105
- analyzers, 92–164
  - `@load`, 92
  - activating, 92
  - `analy`, 148
  - application-specific, 120–147
  - `conn`, 95
  - filtering, 93–94
  - `finger`, 120
    - event handlers, 120
    - variables, 120
  - `ftp`, 122
    - event handlers, 126–127
    - functions, 125
    - variables, 123–125
  - generic, 95–105
  - `hot`, 106
    - functions, 110
    - variables, 106–110
  - `http`, 127
    - event handlers, 128
    - variables, 127–128
  - `ident`, 128
    - event handlers, 129
    - variables, 129
  - instantiating, 92
  - loading, 92
  - `login`, 130
    - event handlers, 138–142
    - functions, 137–138
    - variables, 131–136
  - `portmapper`, 142
    - event handlers, 146–147
    - functions, 144–145
    - variables, 143–144
  - `print-filter`, 94
  - `scan`, 112
    - event handlers, 114
    - functions, 114
    - variables, 112–113
  - `signature`, 148
  - `site`, 105
  - site-specific information, 105–106
  - `SSL`, 149
    - event handlers, 152–154
    - variables, 151–152
  - `tcp`, 100
  - `udp`, 100
- && "and" operator, 22, 47
- `anon_log` variable, 57
- `anonymize_ip_addr` variable, 62
- *anonymous function* expression, 48
- *anticode.com*, 133
- "any" type, 42
  - replacing with union type, 88
- `any_RPC_okay` variable, 75, 144
- appending to a file, 89
- *arithmetic* expression, 47
- array
  - associative, 33
  - multi-dimensional, 34
- as, 39
  - `RLIMIT_NOFILE`, 39
- ASCII
  - as usual character set, 25
- assigning records, 32–33
- *assignment* expression, 48
- associative array, 33

- attack
  - Land, 110
- attackers
  - weenie, 121
- attacks
  - smurf, 133
- `ATTEMPT_INTERVAL` internal variable, 99
- attempted connections, 99
- attempted services
  - forbidden, 109
- attributes, 55
  - `&add_func`, 56
  - `&create_expire`, 36
  - `&default`, 35
  - `&delete_func`, 56
  - `&expire_func`, 36
  - `&read_expire`, 36
  - record fields, 32
  - `&redef`, 56
  - `&write_expire`, 36
- `auth error` (RPC status code), 146
- `auth-failed/` authentication annotation, 141
- `auth/` authentication annotation, 141
- authentication
  - accepted, 141
  - rejected, 141
  - skipped, 141
- authentication annotations, 129, 139–141
  - `auth-failed/`, 141
  - `auth/`, 141
  - `confused/`, 139, 140
  - `ident/`, 129
  - `(skipped)`, 141
- authentication dialog, 87, 90, 130, 131
  - evasion, 132
- `authentication_accepted` event, 141
- `authentication_rejected` event, 141
- `authentication_skipped` event, 141
- avoiding processing, 90

- backdoor
  - avoiding false positives, 133
  - prompts, 133
  - triggered by ephemeral port, 134
  - triggered by terminal type, 133
- `backdoor_annotate_standard_ports` variable, 59
- `backdoor_demux_disabled` variable, 58
- `backdoor_demux_skip_tags` variable, 58
- `backdoor_ignore_dst_addrs` variable, 58
- `backdoor_ignore_ports` variable, 58
- `backdoor_ignore_src_addrs` variable, 58
- `backdoor_log` variable, 58
- `backdoor_min_7bit_ascii_ratio` variable, 58
- `backdoor_min_bytes` variable, 58
- `backdoor_min_normal_line_ratio` variable, 58
- `backdoor_min_num_lines` variable, 58
- `backdoor_prompts` variable, 73, 133
- `backdoor_standard_ports` variable, 58
- `backdoor_stat_backoff` variable, 59
- `backdoor_stat_period` variable, 58
- `backscatter_ports` variable, 78
- backspace character, 86
- bad address mask
  - run-time error, 88
- bad fmt date argument
  - run-time error, 87
- bad fmt editing character
  - run-time error, 86
- bad fmt field width
  - run-time error, 86
- bad fmt floating-point argument
  - run-time error, 87
- bad fmt format specifier
  - run-time error, 87
- bad fmt integer argument
  - run-time error, 87
- bad fmt precision
  - run-time error, 86
- `bad format`, 87
- bad length argument (not a table or set)
  - run-time error, 88
- `bad second argument to mask_addr()`, 88
- format conversion error
  - `bad time`, 87
- `bad type for Date format`, 87
- `bad type for floating-point format`, 87
- `bad type for integer format`, 87
- `bad_HTTP_reply` (“weird” event), 157
- `bad_HTTP_version` (“weird” event), 157
- `bad_ICMP_checksum` (“weird” event), 157
- `bad_ident_reply` (“weird” event), 160
- `bad_ident_request` (“weird” event), 161
- `bad_IP_checksum` (“weird” event), 162
- `bad_option` event, 141
- `bad_option_termination` event, 141
- `bad_pm_port` (“weird” event), 163
- `bad_rlogin_prolog` (“weird” event), 157
- `bad_RPC` (“weird” event), 157
- `bad_RPC_program` (“weird” event), 157
- `bad_SYN_ack` (“weird” event), 157
- `bad_TCP_checksum` (“weird” event), 157
- `bad_TCP_header_len` (“weird” event), 162
- `bad_UDP_checksum` (“weird” event), 157
- `baroque_SYN` (“weird” event), 157
- beginning time of a connection, 98, 101
- bidirectional vs. unidirectional analysis, 159
- big endian, 87, 148
- `/bin/eject` exploit, 131
- BIND
  - non-blocking DNS lookups, 13
- `blank_in_HTTP_request` (“weird” event), 157

- `bool`, *see* types, `bool`
- booleans, 22
- Bourne shell, 90
- BPF (Berkeley Packet Filter)
  - tuning, 13
- BPF buffers
  - ensuring they are large, 13
- `break` keyword, 45
- `break` statement, 45
- Bro
  - checkpointing, 18
  - execution aborted, 18
  - flags
    - `-F`, 18
    - `-O`, 18
    - `-P`, 18, 118
    - `-W`, 18
    - `-f`, 17
    - `-h`, 17
    - `-i`, 17
    - `-p`, 17
    - `-r`, 17
    - `-s`, 17
    - `-v`, 18
    - `-w`, 18
  - installing, 12
  - interactive use, 14
  - not running as root, 13
  - optimizer, 18
  - private caches, 18
  - references, 10
  - running, 12
  - search path, 19
  - shadow, 94
  - source code, 12
  - system configuration, 13
  - usage, 17
  - version, 18
  - watchdog, 18
  - web page, 12
  - wedging, 18
- Bro bugs/limitations
  - causing "weird" events, 154
- `.bro` suffix, 19
- `.bro-dns-cache`, 118
- `bro_done` event, 95
- `$BRO_ID` environment variable, 115
- `bro_init` event, 94
- `bro_log_file` variable, 84, 115
- `$BRO_PREFIXES` environment variable, 17, 19
- `bro_signal` event, 95
- `$BROPATH` environment variable, 19
- BS, 86
- buffer overflow tools, 133
- buffer size patch for *l*ibpcap, 13
- buffers
  - large for BPF, 13
- buggy implementations
  - causing "weird" events, 154
- bugs
  - `$` pattern operator not supported, 27
  - appalling, 163
  - causing "weird" events, 154
  - `tcpdump`, 94
- building Bro, 12
- `byte_len` function, 85
- bytes in connection, 101, 103

- caches
  - Bro's private ones, 18
- CALLIT portmapper call, 142
- can't open
  - run-time error, 19
- `can_drop_connectivity` variable, 113
- `cannot create directory`, 88
- `capture_filter` global variable, 17
- `capture_filter` variable, 84, 93–94
- casting
  - not provided in Bro, 42
- `cat` function, 85
- Central Intelligence Agency
  - detection, 108
- *cf* utility program, 20
- character set
  - ASCII, 25
- `check_hot` function, 110
- `check_info` record, 137
  - `forbidden`, 137
  - `hot`, 137
  - `hot_id`, 137
- `check_relay_3` variable, 74
- `check_relay_4` variable, 74
- `check_scan` function, 114
- `check_spoof` function, 110
- checkpointing Bro, 18
- checksum error
  - ICMP, 157
  - IP, 162
  - TCP, 157
  - UDP, 157
- Christmas packet, 160
- CIA detection, 108
- CIDR, 29, 88, 91, 105
- `clean` function, 85
- cleanup event, 95
- client port
  - triggering a backdoor, 134
- `client_cert`, 150
  - `ssl_connection_info` field, 150
- clock time, 86, 88
- `close` function, 85

code_red_list1 variable, 63
code_red_list2 variable, 63
code_red_log variable, 63
Cold Fusion exploits, 127
command shell, 90
  setuid root, 133
compiling Bro, 12
completed connections, 99
compound statement, 45
concatenation of strings, 85
conditional expression, 48
configuration options
  --enable-brov6, 13
confused login analysis, 131
confused/ authentication annotation, 139, 140
confusion of heuristics, 131
conn analyzer, 95
conn_id record, 96, 97
conn_size function, 103
conn_state function, 103
conn_stats event, 148
conn_tag_info variable, 81
conn_weird event, 156
conn_weird_addl event, 160
connection
  additional information, 98, 103
  addresses, 96, 101
  analysis, 95, 106, 148
  attempt, 99
  bytes, 96, 101, 103
  completion, 99, 100
  definitions, 98
  detecting sensitive, 110
  duration, 98, 101
  establishment, 99
  events, 99
  finished, 99
  flags, 101
  functions, 103
  generic analysis, 95
  half finished, 100
  hosts, 101
  hot, 98, 104, 138
  hot analysis, 106
  ICMP, 98
  ID, 103, 104
  initiator, 96
  logging, 104
  new, 99
  non-existing, 91
  originator, 96
  partial, 99
  partial close, 100
  pending, 100
  ports, 96
  recording, 105
  rejected, 99
  reset, 100
  reuse, 159
  sensitivity, 98
  sequence numbers, 87
  service, 98, 101, 103, 105
  simultaneous open, 159
  size, 96, 101, 103
  start time, 98, 101
  state, 96, 101, 103
  summaries, 101
  TCP, 98
  terminating with extreme prejudice, 105
  testing for existence, 85
  UDP, 98
connection events
  TCP-specific, 99
connection id is not a known connection, 86, 91
connection id is not a known login connection, 87, 90
connection record, 96–98
connection size
  undetermined for RST termination, 125
connection states, 102
  OTH, 102
  REJ, 102
  RSTO, 102
  RSTOS0, 102
  RSTR, 102
  RSTRH, 102
  S0, 102
  S1, 102
  S2, 102
  S3, 102
  SF, 102
  SH, 102
  SHR, 102
connection summary files, 102
  red, 102
connection_attempt event, 99
connection_established event, 99, 141
connection_finished event, 99
connection_half_finished event, 100
connection_id, 150
  ssl_connection_info field, 150
connection_originator_SYN_ack (“weird” event), 157
connection_partial_close event, 100
connection_pending event, 100
connection_record function, 85
connection_rejected event, 99
connection_reset event, 100
connectivity

  dropping, 113, 114
const statement, 45
const variable declaration, 54
constant expression, 46
constant variables, 45
constants, 148
  address, 29
  boolean, 22
  count, 23
  ENDIAN_BIG, 148
  ENDIAN_CONFUSED, 148
  ENDIAN_LITTLE, 148
  ENDIAN_UNKNOWN, 148
  floating-point, 23
  hostname, 29
  integer, 23
  interval, 27
  net, 30
  pattern, 25–26
  port, 28
  record, 31
  string, 24–25
  temporal, 27
  time, 27
contains_string function, 86
CONTENTS_BOTH direction, 89
CONTENTS_NONE direction, 89
CONTENTS_ORIG direction, 89
CONTENTS_RESP direction, 89
control packets (SYN/FIN/RST), 18, 100
conversion of non-IPv4 address to net, 91
converting an IPv6 address to net
  run-time error, 91
copy
  shallow vs. deep, 32, 38
corrupted packets, 157, 162
count, *see* types, count
count maximum, 88
count minimum, 88
&create_expire attribute, 36
creating directories, 88
creation_time, 118
  dns_mapping field, 118
crud, 99, 154
current_time function, 86

D format, 87
d format, 87
daemon username, 121, 129
daemons
  as innocuous user names, 129
data
  unanalyzed, 158
data_after_reset (“weird” event), 157
data_before_established (“weird” event), 157
day interval unit, 27
debugging
  filtering problems, 94
decrement expressions, 46
deep copy, 32, 38
default
  filtering, 93
&default attribute, 35
default values, 35
DEL, 85–86
delete character, 85
delete keyword, 45
delete statement, 45
&delete_func attribute, 56
demux module, 117
demux_conn function, 117
demux_dir variable, 64
demuxed_conn variable, 64
denial of service
  excessively large fragments, 161
  Land attack, 163
detected_stones variable, 81
detecting scans, 112
detecting sensitive connections, 110
determine_service function, 103
/dev/bpf, 13
did_PTR_scan_event variable, 65
did_sigconns variable, 60
did_ssh_version variable, 81
did_stone_summary variable, 82
direct_login_prompts variable, 84, 134
directions, 89
  CONTENTS_BOTH, 89
  CONTENTS_NONE, 89
  CONTENTS_ORIG, 89
  CONTENTS_RESP, 89
directories
  creating, 88
directory names
  sensitive, 131
discarder_check_icmp function, 86
discarder_check_ip function, 86
discarder_check_tcp function, 86
discarder_check_udp function, 86
discarder_maxlen variable, 84
display_pairs variable, 81
distinct_answered_PTR_requests variable, 65
distinct_backscatter_peers variable, 78
distinct_peers variable, 76
distinct_ports variable, 76
distinct_PTR_requests variable, 64
distinct_rejected_PTR_requests variable, 64
diverse network use
  causing “weird” events, 154
division

  - numeric, 23
  - temporal, 28
- DMZ
  - spoof detection, 107
- DNS
  - Bro’s private cache, 118
    - forcing access to, 18
  - mappings, 118
- DNS lookups
  - non-blocking, 13
- `dns` module, 118
- `dns_interesting_changes` variable, 65, 119
- `dns_log` variable, 64
- `dns_mapping` record, 118, 119
- `dns_mapping_altered` event, 119
- `dns_mapping_lost_name` event, 119
- `dns_mapping_name_changed` event, 119
- `dns_mapping_new_name` event, 119
- `dns_mapping_unverified` event, 119
- `dns_mapping_valid` event, 119
- `dns_session_timeout` variable, 62
- `dns_sessions` variable, 64
- `done_with_network` global variable, 50
- `done_with_network` variable, 84
- dotted quads, 20
- `double`, *see* types, `double`
- `double` maximum, 88
- `double` minimum, 88
- *drop-connectivity* shell script, 114
- `drop_address` function, 114
- dropping connectivity, 113, 114
- DUMP portmapper call, 142
- `duration`, 98
  - `connection` field, 98
- duration of a connection, 98, 101
- dynamic defaults, 35

- `e` format, 87
- `edit` function, 86
- `edit_and_check_line` function, 137
- `edit_and_check_password` function, 138
- `edit_and_check_user` function, 137
- `edited_input_trouble` variable, 72, 131
- editing, 86
- `eggdrop` sensitive filename, 124
- `eggdrop` sensitive login input, 131
- `eject` exploit, 131
- `else` keyword, 44
- embedded NUL
  - run-time error, 91
- `--enable-brov6` configuration option, 13
- encrypted login sessions, 142
- encryption
  - leading to “excessive lines”, 141
- endian issues, 87, 148
- `ENDIAN_BIG` constant, 148
- `ENDIAN_CONFUSED` constant, 148
- `ENDIAN_LITTLE` constant, 148
- `endian_type` statistic, 148
- `ENDIAN_UNKNOWN` constant, 148
- `endpoint` record, 96, 97, 101
- `endpoint_id` function, 115
- `enum`, *see* types, `enum`, *see* types, `enum`
- enumerations, 24
- environment
  - accessing, 87
  - responder, 132
  - Telnet options, 132
- environment variables
  - `$BRO_ID`, 115
  - `$BRO_PREFIXES`, 17, 19
  - `$BROPATH`, 19
  - `$USER`, 132
- ephemeral port, 104
  - triggering a backdoor, 134
- ephemeral ports
  - confused with sensitive services, 109
- *equality* expression, 47
- escape sequences, 24
- established connections, 99
- */etc/inetd.conf*, 109
- */etc/passwd*, 127
- */etc/shadow*, 127
- evasion
  - authentication dialog, 130, 132
  - excessively small fragments, 161
  - inconsistent fragment size, 161
  - inconsistent fragments, 161
  - inconsistent RPC retransmission, 159
  - inconsistent TCP retransmission, 163
  - inserting NULs, 24
  - length mismatch, 160
  - using tunneling, 140
- `event`, *see* types, `event`
- event engine, 41
- event handler
  - invocation, 41
- event handlers, 41
- event handling
  - weird, 156–163
- `event` keyword, 44
- *event scheduling* expression, 50
- `event` statement, 44
- event type, 41
- events
  - `account_tried`, 114
  - `ack_above_hole`, 95, 163
  - `activating_encryption`, 142
  - `authentication_accepted`, 141
  - `authentication_rejected`, 141
  - `authentication_skipped`, 141

  bad_option, 141
  bad_option_termination, 141
  bro_done, 95
  bro_init, 94
  bro_signal, 95
  conn_stats, 148
  conn_weird, 156
  conn_weird_addl, 160
  connection_attempt, 99
  connection_established, 99, 141
  connection_finished, 99
  connection_half_finished, 100
  connection_partial_close, 100
  connection_pending, 100
  connection_rejected, 99
  connection_reset, 100
  dns_mapping_altered, 119
  dns_mapping_lost_name, 119
  dns_mapping_name_changed, 119
  dns_mapping_new_name, 119
  dns_mapping_unverified, 119
  dns_mapping_valid, 119
  exceptional, 154–163
  excessive_line, 140
  finger_reply, 120
  finger_request, 120
  finish, 95
  flow_weird, 161
  ftp_reply, 126
  ftp_request, 126
  general Bro processing, 94
  generic TCP connection, 99
  http_request, 128
  ident_error, 129
  ident_reply, 129
  ident_request, 129
  inconsistent_option, 140
  initialization, 94
  login_confused, 139
  login_confused_text, 140
  login_failure, 138
  login_input_line, 139
  login_output_line, 139
  login_success, 138
  login_terminal, 140
  net_done, 95
  net_stats_update, 95
  net_weird, 162
  new_connection, 99
  partial_connection, 99, 141
  pm_attempt_callit, 147
  pm_attempt_dump, 147
  pm_attempt_getport, 147
  pm_attempt_null, 147
  pm_attempt_set, 147
  pm_attempt_unset, 147
  pm_bad_port, 147
  pm_request_callit, 146
  pm_request_dump, 146
  pm_request_getport, 146
  pm_request_null, 146
  pm_request_set, 146
  pm_request_unset, 146
  PTR_scan, 65
  rexmit_inconsistency, 163
  scheduling, 50
  signature_match, 148
  ssl_certificate, 153
  ssl_certificate_seen, 153
  ssl_conn_alert, 153
  ssl_conn_attempt, 152
  ssl_conn_established, 153
  ssl_conn_reused, 153
  ssl_conn_server_reply, 153
  ssl_conn_weak, 154
  startup, 94
  termination, 95
  udp_reply, 100
  udp_request, 100
exceptional events, 154–163
excess_RPC (“weird” event), 157
excessive_line event, 140
excessive_ntp_request variable, 74
excessive_RPC_len (“weird” event), 157
excessive_typeahead (login confusion state), 132
excessively long lines, 140
excessively_large_fragment (“weird” event), 161
excessively_small_fragment (“weird” event), 161
excluding hosts, 93
executables
  running, 90
exit function, 86
expanded_line, 137
  check_info field, 137
expiration
  timer, 50, 84
&expire_func attribute, 36
explicit typing, 54
exploit scans, 148
exploit tools, 133
  smashdu.c, 133
exploits, 131
  /bin/eject, 131
  buffer overflow, 133
  eject, 131
  loadmodule, 131
  Unix, 131
expression statement, 43
expressions, 46–52
  anonymous function, 48

    arithmetic, 47
    assignment, 48
    conditional, 48
    constant, 46
    equality, 47
    event scheduling, 50
    function call, 48
    index, 50
    logical, 47
    membership, 51
    negation, 46
    parenthesized, 46
    pattern matching, 51
    positivation, 47
    record constructor, 52
    record field access, 52
    record field test, 52
    relational, 48
    variable, 46
    decrement, 46
    increment, 46
`extra_repeat_text` (login confusion state), 132
`EZsetup` username, 121

`F`, 22
`-F` flag, 18
`-f` flag, 17
`f` format, 87
failure of heuristics, 131
fatal run-time error
    non-existing connection, 86
fetch utility, 133
fflush, 86
field attributes, 32
`file`, *see* types, `file`
file type, 39
filenames
    sensitive, 124, 131
files
    appending, 89
    opening, 89
    testing if open, 85
filtering
    default, 93
filters, 93–94
    displaying, 94
    errors, 94
FIN control packet, 18, 100
`FIN_advanced_last_seq` (“weird” event), 157
`FIN_after_reset` (“weird” event), 158
`FIN_storm` (“weird” event), 158
Finger
    analysis, 120
    weird events, 158
`finger` analyzer, 120
`finger_reply` event, 120
`finger_request` event, 120
finish event, 95
firewall
    reactive, 113, 114
`flag_rejected_service` variable, 68, 109
`flag_successful_inbound_service` variable, 68, 109
`flag_successful_service` variable, 67, 109
flags
    `-F`, 18
    `-O`, 18
    `-P`, 18, 118
    `-W`, 18
    `-f`, 17
    `-h`, 17
    `-i`, 17
    `-p`, 17
    `-r`, 17
    `-s`, 17
    `-v`, 18
    `-w`, 18
flags of connection, 101
flex utility, 25
`flow_weird` event, 161
`flush_all` function, 86
`fmt` function, 86
`for` keyword, 44
`for` statement, 44
`forbidden check_info` record, 137
`forbidden_id_patterns` variable, 68, 121
`forbidden_ids` variable, 68, 121
`forbidden_ids_if_no_password` variable, 68, 121
forcing access to Bro’s private DNS cache, 18
format
    `%`, 87
    `D`, 87
    `d`, 87
    `e`, 87
    `f`, 87
    `g`, 87
    precision, 86
    width, 86
formatting text, 86
`.forward`, 124
`frag` module, 121
fragment reassembly, 121
`fragment_inconsistency` (“weird” event), 161
`fragment_overlap` (“weird” event), 161
`fragment_protocol_inconsistency` (“weird” event), 161
`fragment_size_inconsistency` (“weird” event), 161
`fragment_with_DF` (“weird” event), 162
fragments
    excessively large, 161

  excessively small, 161
  inconsistent, 161
  inconsistent protocols, 161
  inconsistent sizes, 161
  overlapping, 161
  TCP vs. UDP, 121
frogs
  dissecting, 127
FTP
  analysis, 122
  ephemeral ports confused with sensitive services, 109
  log file, 125
  session information, 122
  weird events, 158
ftp analyzer, 122
ftp session summary file, 125
ftp_data_expected variable, 66
ftp_data_expected_session variable, 66
ftp_excessive_filename_len variable, 66
ftp_excessive_filename_trunc_len variable, 66
ftp_guest_ids variable, 65, 123
ftp_hot_cmds variable, 66
ftp_hot_files variable, 66, 124
ftp_hot_guest_files variable, 66, 124
ftp_ignore_invalid_PORT variable, 66
ftp_ignore_privileged_PASVs variable, 66
ftp_log variable, 65
ftp_not_actually_hot_files variable, 124
ftp_port record, 89
ftp_reply event, 126
ftp_request event, 126
ftp_session_info record, 122, 123
ftp_sessions variable, 65
ftp_sig_disabled variable, 59
ftp_skip_hot variable, 65, 123
full_id_string function, 103
full_input_trouble variable, 72
full_output_trouble variable, 72
function, *see* types, function
function call expression, 48
function invocation, 48
function keyword, 48
function type, 39–40
functions, 39–40
  active_connection, 85
  active_file, 85
  add_interface, 85
  add_tcpdump_filter, 85
  anonymous, 48
  byte_len, 85
  cat, 85
  check_hot, 110
  check_scan, 114
  check_spoof, 110
  clean, 85
  close, 85
  conn_size, 103
  conn_state, 103
  connection_record, 85
  contains_string, 86
  current_time, 86
  demux_conn, 117
  determine_service, 103
  discarder_check_icmp, 86
  discarder_check_ip, 86
  discarder_check_tcp, 86
  discarder_check_udp, 86
  drop_address, 114
  edit, 86
  edit_and_check_line, 137
  edit_and_check_password, 138
  edit_and_check_user, 137
  endpoint_id, 115
  exit, 86
  flush_all, 86
  fmt, 86
  full_id_string, 103
  get_login_state, 87
  get_orig_seq, 87
  get_resp_seq, 87
  getenv, 87
  has_signature_matched, 149
  hot_login, 137
  id_string, 104
  is_forbidden_id, 137
  is_ftp_data_conn, 125
  is_hot_id, 137
  is_local_addr, 106
  is_login_conn, 137
  is_tcp_port, 87
  length, 88
  log_file_name, 88
  log_hook, 85, 115, 116
  log_hot_conn, 104
  mask_addr, 88
  max_count, 88
  max_double, 88
  max_interval, 88
  min_count, 88
  min_double, 88
  min_interval, 88
  mkdir, 88
  network_time, 88
  open, 39, 89
  open_for_append, 39, 89
  open_log_file, 89
  parse_ftp_pasv, 89
  parse_ftp_port, 89
  pm_activity, 144
  pm_attempt, 145

  pm_check_getport, 144
  pm_request, 145
  reading_live_traffic, 89
  record_connection, 105
  redefining, 40
  report_weird, 156
  report_weird_conn, 156
  report_weird_orig, 156
  rpc_prog, 144
  service_name, 105
  set_buf, 89
  set_contents_file, 89
  set_login_state, 90
  set_record_packets, 90
  site-specific, 106
  skip_further_processing, 90
  sub_bytes, 90
  system, 90
  terminate_connection, 105
  to_lower, 90
  to_net, 91
  to_upper, 91

g format, 87
garbage args (RPC status code), 146
general Bro processing events, 94
general scripting, 88
generic connection analysis, 95
GET HTTP method, 127
get_login_state function, 87
get_orig_seq function, 87
get_resp_seq function, 87
getenv function, 87
GETPORT portmapper call, 142
global scope
  of enumerations, 24
global variable declaration, 53
global variables, 53
  capture_filter, 17
  done_with_network, 50
  interfaces, 16, 17
  restrict_filter, 17
gnutella_sig_disabled variable, 59
gtld_servers variable, 78

-h flag, 17
half-finished connections, 100
handling signals, 95
handshake_cipher, 151
  ssl_connection_info field, 151
has_signature_matched function, 149
have_FTP variable, 63
have_skip_remote_sensitive_URIs variable, 69
have_SMTP variable, 63
have_stats variable, 63
HEAD HTTP method, 127
headers
  truncated, 162
heartbeat_interval variable, 62
help message, 17
heuristics
  attacker-induced confusion, 132
  confusion, 131
  environment, 132
  extracting username information, 130, 131
  missing login prompt, 132
  missing username, 132
  multiple login prompts, 132
  multiple usernames, 132
  type-ahead, 132
  VMS, 132
\x*hex-digits* hexadecimal escape, 24
hf utility program, 20
horiz_scan_thresholds variable, 76, 149
horizontal exploit scans, 148
host order (vs. network order), 87
hostname, 119
  dns_mapping field, 119
hostnames, 29
  mapping addresses to, 20
hosts
  excluding, 93
  in a connection, 101
hot, 98
  connection field, 98
hot /24 destination networks, 108
hot /24 source networks, 107
hot analyzer, 106
hot check_info record, 137
hot connection
  analysis, 106
  logging, 104
hot connections, 138
hot destination addresses, 107
hot detection, 110
hot source addresses, 107
hot usernames, 121
hot-ids module, 121
hot_conns_reported variable, 63
hot_dst_24nets variable, 67, 108
hot_dsts variable, 67, 107
hot_id check_info record, 137
hot_ident_exceptions variable, 70, 129
hot_ident_ids variable, 70, 129
hot_ids variable, 68, 122
hot_login function, 137
hot_login_ids variable, 74, 136
hot_names variable, 65, 120
hot_src_24nets variable, 67, 107
hot_srcs variable, 67, 107
hot_ssh_orig_ports variable, 134

- `hot_telnet_orig_ports` variable, 73, 134
- `hot_terminal_types` variable, 73, 133
- `hr` (hours) interval unit, 27
- HTTP
  - analysis, 127
  - log file, 127
  - weird events, 157
- `http` analyzer, 127
- HTTP methods, 127
  - `GET`, 127
  - `HEAD`, 127
  - `POST`, 127
- HTTP packets
  - contents not being recorded, 18
- `http` session summary file, 127
- `http_abstract_max_length` variable, 69
- `http_log` variable, 68
- `http_proxy_sig_disabled` variable, 60
- `http_request` event, 128
- `http_sessions` variable, 68
- `http_sig_disabled` variable, 60
- `HTTP_unknown_method` (“weird” event), 158
- `HTTP_version_mismatch` (“weird” event), 158
- *HUP* signal, 95

- `-i` flag, 17
- ICMP
  - checksum error, 157
  - connections, 98
  - timeout, 98
  - weird events, 157
- `icmp_flows` variable, 70
- `id`, 96, 122, 150
  - `ftp_session_info` field, 122
  - `ssl_connection_info` field, 150
- ID of connection, 103, 104
- `id_index`, 151
  - `ssl_connection_info` field, 151
- `id_string` function, 104
- IDENT
  - analysis, 128
  - weird events, 160
- `ident` analyzer, 128
- `ident/` authentication annotation, 129
- `ident_error` event, 129
- `ident_reply` event, 129
- `ident_request` event, 129
- `ident_request_addendum` (“weird” event), 161
- IEUser
  - useless FTP username, 125
- `if` keyword, 44
- `if` statement, 44
- `ignore_checksums` variable, 60
- implicit typing, 54
- `in` operator, 26, 51
- in-order delivery, 148
- `!in` negation of `in` operator, 27
- `inactivity_timeout` variable, 61
- `inappropriate_FIN` (“weird” event), 158
- inbound services
  - fatal, 109
  - forbidden, 109
- `include_HTTP_abstract` variable, 69
- `incompletely_captured_fragment` (“weird” event), 162
- inconsistent acknowledgment, 163
- inconsistent retransmission, 159, 163
- `inconsistent_option` event, 140
- *increment* expressions, 46
- index
  - of a table, 33
- *index* expression, 50
- *inetd.conf*, 109
- inferring types, 55
- information associated with a connection, 98, 103
- `ingreslock` popular backdoor, 109
- initialization event, 94
- initialization of variables, 55
- input
  - analysis, 130
  - editing, 131
- `input_trouble` variable, 72, 131
- `input_wait_for_output` variable, 72
- installing Bro, 13
- `int`, *see* types, `int`
- *INT* signal, 95
- integers
  - network vs. host order, 87
- `interconn_conns` variable, 70
- `interconn_default_pkt_size` variable, 70
- `interconn_demux_disabled` variable, 72
- `interconn_ignore_standard_ports` variable, 72
- `interconn_log` variable, 70
- `interconn_max_interarrival` variable, 70
- `interconn_max_keystroke_pkt_size` variable, 70
- `interconn_min_7bit_ascii_ratio` variable, 71
- `interconn_min_alpha` variable, 72
- `interconn_min_bytes` variable, 71
- `interconn_min_duration` variable, 71
- `interconn_min_gamma` variable, 72
- `interconn_min_interarrival` variable, 70
- `interconn_min_normal_line_ratio` variable, 71
- `interconn_min_num_lines` variable, 71
- `interconn_min_num_pkts` variable, 71
- `interconn_min_ssh_pkts_ratio` variable, 71
- `interconn_ssh_len_disabled` variable, 71
- `interconn_standard_ports` variable, 72
- `interconn_stat_backoff` variable, 71
- `interconn_stat_period` variable, 71
- `interfaces` global variable, 16, 17
- `interfaces` variable, 84

- internal networks
  - spoof detection, 107
- internal variables
  - `ATTEMPT_INTERVAL`, 99
  - `PARTIAL_CLOSE_INTERVAL`, 100
  - `WATCHDOG_INTERVAL`, 18
- `internally_truncated_header` (“weird” event), 162
- Internet Relay Chat (IRC)
  - attacker subpopulation, 131
- `interval`, *see* types, `interval`
- `interval` maximum, 88
- `interval` minimum, 88
- interval units
  - `day`, 27
  - `hr`, 27
  - `min`, 27
  - `sec`, 27
  - `usec`, 27
- invocation
  - function, 48
- invoking event handlers, 41
- IP
  - checksum error, 162
  - fragments, 161
  - identification field, 148
  - weird events, 162
- IPv4/IPv6 address constants, 29
- IPv6 and lack of CIDR prefixes, 30
- IPv6 support, 29
- IRC, 131
- `is not a TCP connection`, 87
- `is_forbidden_id` function, 137
- `is_ftp_data_conn` function, 125
- `is_hot_id` function, 137
- `is_local_addr` function, 106
- `is_login_conn` function, 137
- `is_tcp_port` function, 87
- *isascii*, 90, 91
- *islower*, 91
- *isupper*, 90

- `kazaa_sig_disabled` variable, 59
- keystrokes
  - analysis, 130
  - editing, 131
- keywords
  - `add`, 45
  - `break`, 45
  - `delete`, 45
  - `else`, 44
  - `event`, 44
  - `for`, 44
  - `function`, 48
  - `if`, 44
  - `log`, 43
  - `next`, 44
  - `print`, 43
  - `return`, 45
  - `schedule`, 50
- kiddies
  - script, 107

- Land attack, 110, 163
- `Land_attack` (“weird” event), 163
- large BPF buffers, 13
- `last_stat` variable, 63
- `last_stat_time` variable, 64
- `(` operator, 46, 48
- length
  - of strings, 85
  - of table or set, 88
- `length` function, 88
- length mismatch
  - UDP, 160
- `length() requires a table/set argument`, 88
- `length() takes exactly one argument`, 88
- *lex* utility, 25
- *l*ibpcap buffer size patch, 13
- *libpcap* library, 13
- libraries
  - *libpcap*, 13
- line editing, 86
- Linux
  - compiling Bro under, 13
  - super exploit, 133
- little endian, 87, 148
- live traffic, 44, 89
- load
  - shedding, 90
- `loadmodule` exploit, 131
- local addresses, 105, 106
  - spoofing, 106, 107, 110
- `local` statement, 45
- `local` variable declaration, 53
- local variables, 45, 53
- `local_16_nets` variable, 79, 105
- `local_24_nets` variable, 79, 105
- `local_code_red_response_pgm` variable, 63
- `local_mail_addr` variable, 79
- `local_nets` variable, 79, 105
- log file, 84, 88, 115
  - altering, 133
  - connection summary (`red`), 105
  - FTP, 125
  - HTTP, 127
  - signatures, 148
  - SSL, 152
  - weird events, 154
- `log` keyword, 43
- `log` module, 115

log statement, 43
log_file_name function, 88
log_hook function, 115, 116
log_hook predefined function, 85
log_hot_conn function, 104
log_HTTP_data variable, 69
log_if_not_denied, 123
  ftp_session_info field, 123
log_if_not_unavail, 123
  ftp_session_info field, 123
log_it, 123
  ftp_session_info field, 123
LOG_NOTICE syslog level, 44
logging
  connection, 104
  control of, 85
logical expression, 47
logical negation, 22
login analysis
  confusion, 131
login analyzer, 130
login confusion states, 132
  excessive_typeahead, 132
  extra_repeat_text, 132
  multiple_login_prompts, 132
  multiple_USERs, 132
  no_login_prompt, 132
  no_username, 132
  no_username2, 132
  non_empty_multi_login, 132
  possible_login_ploy, 132
  repeat_without_username, 132
  responder_environment, 132
  username_with_embedded_repeat, 132
login prompts
  missing, 132
  repeated, 132
login session, 130
  state, 87, 90
login_confused event, 139
login_confused_text event, 140
login_failure event, 138
login_failure_msgs variable, 73, 134
login_input_line event, 139
login_non_failure_msgs variable, 73, 135
login_output_line event, 139
login_prompts variable, 73, 134
login_sessions variable, 74
LOGIN_STATE_AUTHENTICATE state of login connection, 87
LOGIN_STATE_CONFUSED state of login connection, 87
LOGIN_STATE_LOGGED_IN state of login connection, 87
LOGIN_STATE_SKIP state of login connection, 87
login_success event, 138
login_success_msgs variable, 73, 135
login_terminal event, 140
login_timeouts variable, 73, 136
ls utility, 133
lynx utility, 133

magic terminal types, 133
maintain_http_sessions variable, 69
management
  of state, 36
mask_addr function, 88
masking, 88, 91
max_count function, 88
max_double function, 88
max_finger_request_len variable, 65
max_interval function, 88
max_request_length variable, 120
max_timer_expires variable, 84
maximum, 88
Maximum Segment Lifetime (MSL), 159
maximums, 88
  count, 88
  double, 88
  interval, 88
membership expression, 51
memory management, 36
message
  bad format, 87
  bad second argument to mask_addr(), 88
  bad type for Date format, 87
  bad type for floating-point format, 87
  bad type for integer format, 87
  cannot create directory, 88
  connection id is not a known connection, 86, 91
  connection id is not a known login connection, 87, 90
  conversion of non-IPv4 address to net, 91
  is not a TCP connection, 87
  length() requires a table/set argument, 88
  length() takes exactly one argument, 88
  not exactly one edit character, 86
  precision specified for non-floating point format, 86
  ridiculous field width or precision, 86
  string with embedded NUL, 91
  string without NUL terminator, 91
  too few arguments for format, 87
  too many arguments for format, 87
  wrong number of *fmt* arguments, 87
mime_log variable, 74

- `mime_sessions` variable, 74
- `min` (minutes) interval unit, 27
- `min_count` function, 88
- `min_double` function, 88
- `min_interval` function, 88
- minimum, 88
- minimums, 88
  - `count`, 88
  - `double`, 88
  - `interval`, 88
- `mismatch` (RPC status code), 146
- missing login prompts, 132
- missing username, 132
- mkdir failure
  - run-time error, 88
- `mkdir` function, 88
- modifiability of variables, 54
- modules
  - `active`, 117
  - `demux`, 117
  - `dns`, 118
    - event handlers, 119
    - variables, 119
  - `frag`, 121
  - `hot-ids`, 121
  - `log`, 115
  - `mt`, 115
  - `port-name`, 115
  - `weird`, 154
- MSL (Maximum Segment Lifetime), 159
- `mt` module, 115
- multi-dimensional table, 34
- multiple login prompts, 132
- multiple usernames, 132
- `multiple_HTTP_request_elements` (“weird” event), 158
- `multiple_login_prompts` (login confusion state), 132
- `multiple_RPCs` (“weird” event), 158
- `multiple_USERs` (login confusion state), 132
- multiplication
  - numeric, 23
  - temporal, 28

- name
  - of log file, 88
- names
  - case-sensitive, 31
- Napster
  - tunneled over Telnet or Rlogin, 140
- `napster_sig_disabled` variable, 59
- negation
  - logical, 22
  - temporal, 27
- *negation* expression, 46
- neighbor addresses, 106
- `neighbor_16_nets` variable, 79, 106
- `neighbor_24_nets` variable, 106
- `neighbor_nets` variable, 79, 106
- `net`, *see* types, `net`
  - constants, 30
  - operators, 30
- net type, 29–30
- `net_done` event, 95
- `net_stats`, *see* types, `net_stats`
- `net_stats` record, 96
- `net_stats_update` event, 95
- `net_weird` event, 162
- network cleanup event, 95
- Network File System (NFS), 143
- network interfaces, 16, 17, 84
- network order (vs. host order), 87
- network prefixes, 29, 91, 105
- network statistics, 95
- Network Virtual Terminal (NVT), 140
- `network_time` function, 88
- networks
  - hot destinations, 108
  - hot sources, 107
- `never_shut_down` variable, 113
- new connection, 99
- `new_connection` event, 99
- `next` keyword, 44
- `next` statement, 44
- NFS (Network File System), 143
- NFS traffic
  - high volume fragments, 121
- `NFS_services` variable, 75, 143
- `NFS_world_servers` variable, 75, 143
- no such connection
  - run-time error, 91
- `no_login_prompt` (login confusion state), 132
- `no_username` (login confusion state), 132
- `no_username2` (login confusion state), 132
- non-blocking DNS lookups, 13
- non-existing connection
  - fatal run-time error, 86
- `non_analyzed_lifetime` variable, 61
- `non_ASCII_hosts` variable, 73, 136
- `non_backdoor_prompts` variable, 73, 133
- `non_empty_multi_login` (login confusion state), 132
- `<none>` username, 132
- not a login connection
  - run-time error, 87, 90
- not a TCP connection
  - run-time error, 87
- `not exactly one edit character`, 86
- `!in` negation of `in` operator, 27
- `!` “not” operator, 22
- NT
  - not supported, 12

ntp_session_timeout variable, 62
NUL, 85
NUL_in_line (“weird” event), 158
NULL portmapper call, 142
null statement, 45
NULs, 158
  allowed in strings, 24, 91
  disallowed in certain function calls, 91
  terminating string constants, 25
  termination, 91
  terminator missing
    run-time error, 91
num_accounts_tried variable, 77
num_backscatter_peers variable, 78
num_distinct_peers variable, 76
num_distinct_ports variable, 76
num_dns_sessions variable, 64
num_in_order statistic, 148
num_OO statistic, 148
num_pkts statistic, 148
num_repl statistic, 148
num_requests, 123
  ftp_session_info field, 123
num_rxmit statistic, 148
num_rxmit_bytes statistic, 148
num_scan_triples variable, 77
number of elements
  in table or set, 88
numeric types
  count, 21
  double, 21
  int, 21
nuucp username, 121, 129
NVT (Network Virtual Terminal), 140
NVT options
  authentication, 141
  bad, 141
  bad termination, 141
  encryption, 142
  inconsistent, 140

-O flag, 18
\octal-digits octal escape, 24
off-line analysis, 16, 17, 89, 105
ok (RPC status code), 146
okay_to_lookup_sensitive_hosts variable, 64
omit_rewrite_place_holder variable, 63
on-line analysis, 16, 17, 84, 89, 105
open function, 39, 89
open_for_append function, 39, 89
open_log_file function, 89
opening a file, 89
operator
  && “and”, 22, 47
  (parenthesis, 46, 48
  ! “not”, 22
  || “or”, 22, 47
) parenthesis, 46, 48
operators
  !, 46
  !in, 51
  (, 50
  ), 50
  *, 23, 28, 47
  +, 23, 28, 47
  ++, 46
  +=, 93
  -, 23, 27, 28, 46, 47
  --, 46
  /, 23, 28, 47
  :, 48
  =, 48
  ?, 48
  [, 50, 52
  $, 32
  ], 50, 52
  address, 29
  arithmetic, 23
    associativity, 23
    operand conversion, 23
    precedence, 23
  comparison, 24
    associativity, 24
    operand conversion, 24
    precedence, 24
  in, 51, *see* in operator
  logical, 22
    associativity, 22
    precedence, 22
  net, 30
  pattern, 26
  ports, 29
  string, 25
  temporal, 27
optimizer for policy script interpreter, 18
optimizing your system for Bro, 13
options
  Telnet, 130
|| “or” operator, 22, 47
orig, 96
orig_h, 96
  conn_id field, 96
orig_p, 96
  conn_id field, 96
originator_RPC_reply (“weird” event), 158
OTH connection state, 102
out-of-order delivery, 148
OutOfBox username, 121
output_trouble variable, 72, 133

-P flag, 18, 118
-p flag, 17

packet filter
  access, 13
  permissions, 13
packets
  control (SYN/FIN/RST), 18, 100
  corrupted, 157, 162
  drops, 95, 163
  recording, 90
  replication, 148
  storms, 158
  time, 88
(), 46, 48
parenthesized expression, 46
parse_ftp_pasv function, 89
parse_ftp_port function, 89
partial connections, 99
PARTIAL_CLOSE_INTERVAL internal variable, 100
partial_connection event, 99, 141
partial_connection_ok variable, 60
partial_finger_request ("weird" event), 158
partial_ftp_request ("weird" event), 158
partial_ident_request ("weird" event), 158
partial_portmapper_request ("weird" event), 158
partial_RPC ("weird" event), 158
partially closed connections, 100
passwd, 127
passwords
  guessing, 112
  inadvertently exposed, 130
  sniffing, 130
PATH_UTMP sensitive pattern, 133
pattern, *see* types, pattern
pattern matching, 25
  embedded, 26
  exact, 26
pattern matching expression, 51
patterns, 25–27
pending connections, 100
pending_data_when_closed ("weird" event), 158
performance
  analysis tradeoffs, 92
  filtering, 93
pm_activity function, 144
pm_attempt function, 145
pm_attempt portmapper attempt, 147
pm_attempt_callit event, 147
pm_attempt_dump event, 147
pm_attempt_getport event, 147
pm_attempt_null event, 147
pm_attempt_set event, 147
pm_attempt_unset event, 147
pm_bad_port event, 147
pm_callit_request portmapper call, 147
pm_check_getport function, 144
pm_mapping portmapper mapping record, 146
pm_port_request portmapper request, 146
pm_request function, 145
pm_request_callit event, 146
pm_request_dump event, 146
pm_request_getport event, 146
pm_request_null event, 146
pm_request_set event, 146
pm_request_unset event, 146
policy/ policy directory, 19
policy directories, 19
policy script interpreter
  optimizer, 18
policy/local/ policy directory, 19
polymorphic functions
  need for, 88, 160
popular backdoors, 109
  ingreslock, 109
port, *see* types, port
  ephemeral, 104
port scanning, 112
port type, 28–29
port-name module, 115
port_names variable, 74, 98, 103, 115
portmapper analyzer, 142
portmapper attempts, 147
  pm_attempt, 147
portmapper calls, 147
  CALLIT, 142
  DUMP, 142
  GETPORT, 142
  NULL, 142
  pm_callit_request, 147
  SET, 142
  UNSET, 142
portmapper mapping records, 146
  pm_mapping, 146
portmapper requests, 146
  pm_port_request, 146
ports
  constants, 28
  operators, 29
  TCP, 28
  TCP vs. UDP, 87
  UDP, 28
positivation expression, 47
possible future changes
  breaking string constants across multiple lines, 24
  constants for absolute times, 27
  timer type, 50
  use of any type for bypassing strong typing, 42
possible packet drop messages, 163
  ack above a hole, 163
possible_login_ploy (login confusion state), 132
possible_port_scan_thresh variable, 77, 112
possible_scan_sources variable, 77

possible_split_routing ("weird" event), 159
POST HTTP method, 127
precision
  of formatted strings, 86
precision specified for non-floating point format, 86
predefined functions, 85–91
  active_connection, 85
  active_file, 85
  add_interface, 85
  add_tcpdump_filter, 85
  byte_len, 85
  cat, 85
  clean, 85
  close, 85
  connection_record, 85
  contains_string, 86
  current_time, 86
  discarder_check_icmp, 86
  discarder_check_ip, 86
  discarder_check_tcp, 86
  discarder_check_udp, 86
  edit, 86
  exit, 86
  flush_all, 86
  fmt, 86
  get_login_state, 87
  get_orig_seq, 87
  get_resp_seq, 87
  getenv, 87
  is_tcp_port, 87
  length, 88
  log_file_name, 88
  log_hook, 85
  mask_addr, 88
  max_count, 88
  max_double, 88
  max_interval, 88
  min_count, 88
  min_double, 88
  min_interval, 88
  mkdir, 88
  network_time, 88
  open, 89
  open_for_append, 89
  open_log_file, 89
  parse_ftp_pasv, 89
  parse_ftp_port, 89
  reading_live_traffic, 89
  set_buf, 89
  set_contents_file, 89
  set_login_state, 90
  set_record_packets, 90
  skip_further_processing, 90
  sub_bytes, 90
  system, 90
  to_lower, 90
  to_net, 91
  to_upper, 91
predefined variables, 57–84
  accounts_tried, 77
  active_conn, 57
  actually_rejected_PTR_anno, 64
  addl_web, 77
  alert_action_filters, 57
  alert_file, 57
  allow_16_net_pairs, 67
  allow_excessive_ntp_requests, 74
  allow_pairs, 67
  allow_PTR_scans, 65
  allow_service_pairs, 67
  allow_services, 67
  allow_services_to, 67
  allow_spoof_services, 67
  always_hot_ids, 68
  always_hot_login_ids, 74
  anon_log, 57
  anonymize_ip_addr, 62
  any_RPC_okay, 75
  backdoor_annotate_standard_ports, 59
  backdoor_demux_disabled, 58
  backdoor_demux_skip_tags, 58
  backdoor_ignore_dst_addrs, 58
  backdoor_ignore_ports, 58
  backdoor_ignore_src_addrs, 58
  backdoor_log, 58
  backdoor_min_7bit_ascii_ratio, 58
  backdoor_min_bytes, 58
  backdoor_min_normal_line_ratio, 58
  backdoor_min_num_lines, 58
  backdoor_prompts, 73
  backdoor_standard_ports, 58
  backdoor_stat_backoff, 59
  backdoor_stat_period, 58
  backscatter_ports, 78
  bro_log_file, 84
  capture_filter, 84
  check_relay_3, 74
  check_relay_4, 74
  code_red_list1, 63
  code_red_list2, 63
  code_red_log, 63
  conn_tag_info, 81
  demux_dir, 64
  demuxed_conn, 64
  detected_stones, 81
  did_PTR_scan_event, 65
  did_sigconns, 60
  did_ssh_version, 81
  did_stone_summary, 82

direct_login_prompts, 84
discarder_maxlen, 84
display_pairs, 81
distinct_answered_PTR_requests, 65
distinct_backscatter_peers, 78
distinct_peers, 76
distinct_ports, 76
distinct_PTR_requests, 64
distinct_rejected_PTR_requests, 64
dns_interesting_changes, 65
dns_log, 64
dns_session_timeout, 62
dns_sessions, 64
done_with_network, 84
edited_input_trouble, 72
excessive_ntp_request, 74
flag_rejected_service, 68
flag_successful_inbound_service, 68
flag_successful_service, 67
forbidden_id_patterns, 68
forbidden_ids, 68
forbidden_ids_if_no_password, 68
ftp_data_expected, 66
ftp_data_expected_session, 66
ftp_excessive_filename_len, 66
ftp_excessive_filename_trunc_len, 66
ftp_guest_ids, 65
ftp_hot_cmds, 66
ftp_hot_files, 66
ftp_hot_guest_files, 66
ftp_ignore_invalid_PORT, 66
ftp_ignore_privileged_PASVs, 66
ftp_log, 65
ftp_sessions, 65
ftp_sig_disabled, 59
ftp_skip_hot, 65
full_input_trouble, 72
full_output_trouble, 72
gnutella_sig_disabled, 59
gtld_servers, 78
have_FTP, 63
have_skip_remote_sensitive_URIs, 69
have_SMTP, 63
have_stats, 63
heartbeat_interval, 62
horiz_scan_thresholds, 76
hot_conns_reported, 63
hot_dst_24nets, 67
hot_dsts, 67
hot_ident_exceptions, 70
hot_ident_ids, 70
hot_ids, 68
hot_login_ids, 74
hot_names, 65
hot_src_24nets, 67
hot_srcs, 67
hot_telnet_orig_ports, 73
hot_terminal_types, 73
http_abstract_max_length, 69
http_log, 68
http_proxy_sig_disabled, 60
http_sessions, 68
http_sig_disabled, 60
icmp_flows, 70
ignore_checksums, 60
inactivity_timeout, 61
include_HTTP_abstract, 69
input_trouble, 72
input_wait_for_output, 72
interconn_conns, 70
interconn_default_pkt_size, 70
interconn_demux_disabled, 72
interconn_ignore_standard_ports, 72
interconn_log, 70
interconn_max_interarrival, 70
interconn_max_keystroke_pkt_size, 70
interconn_min_7bit_ascii_ratio, 71
interconn_min_alpha, 72
interconn_min_bytes, 71
interconn_min_duration, 71
interconn_min_gamma, 72
interconn_min_interarrival, 70
interconn_min_normal_line_ratio, 71
interconn_min_num_lines, 71
interconn_min_num_pkts, 71
interconn_min_ssh_pkts_ratio, 71
interconn_ssh_len_disabled, 71
interconn_standard_ports, 72
interconn_stat_backoff, 71
interconn_stat_period, 71
interfaces, 84
kazaa_sig_disabled, 59
last_stat, 63
last_stat_time, 64
local_16_nets, 79
local_24_nets, 79
local_code_red_response_pgm, 63
local_mail_addr, 79
local_nets, 79
log_HTTP_data, 69
login_failure_msgs, 73
login_non_failure_msgs, 73
login_prompts, 73
login_sessions, 74
login_success_msgs, 73
login_timeouts, 73
maintain_http_sessions, 69
max_finger_request_len, 65
max_timer_expires, 84
mime_log, 74

mime_sessions, 74
napster_sig_disabled, 59
neighbor_16_nets, 79
neighbor_nets, 79
NFS_services, 75
NFS_world_servers, 75
non_analyzed_lifetime, 61
non_ASCII_hosts, 73
non_backdoor_prompts, 73
ntp_session_timeout, 62
num_accounts_tried, 77
num_backscatter_peers, 78
num_distinct_peers, 76
num_distinct_ports, 76
num_dns_sessions, 64
num_scan_triples, 77
okay_to_lookup_sensitive_hosts, 64
omit_rewrite_place_holder, 63
output_trouble, 72
partial_connection_ok, 60
port_names, 74
possible_port_scan_thresh, 77
possible_scan_sources, 77
preserved_net, 57
preserved_subnet, 57
process_HTTP_data, 69
process_HTTP_replies, 69
process_smtp_relay, 79
public_ident_systems, 70
public_ident_user_ids, 70
relay_log, 80
remote_code_red_response_pgm, 63
report_accounts_tried, 77
report_backscatter, 78
report_outbound_peer_scan, 76
report_peer_scan, 76
report_port_scan, 76
report_rejected_PTR_factor, 65
report_rejected_PTR_thresh, 65
report_remote_accounts_tried, 77
restrict_filter, 84
rewrite_finger_trace, 65
rewrite_ident_trace, 70
rewriting_http_trace, 63
rewriting_smtp_trace, 63
rlogin_conns, 60
rlogin_id_okay_if_no_password_exposed, 74
rlogin_sig_1byte_disabled, 59
rlogin_sig_disabled, 59
root_backdoor_sig_conns, 60
root_backdoor_sig_disabled, 59
root_servers, 78
router_prompts, 73
RPC_do_not_complain, 75
RPC_dump_okay, 75
RPC_okay, 75
RPC_okay_nets, 75
RPC_okay_services, 75
rpc_programs, 75
RPC_server_map, 64
rpc_timeout, 62
rule_actions, 76
rule_file, 76
same_local_net_is_spoof, 67
scan_triples, 77
sensitive_lookup_hosts, 64
sensitive_post_URIs, 69
sensitive_URIs, 69
skip_accounts_tried, 77
skip_authentication, 73
skip_clear_ssh_reports, 82
skip_logins_to, 74
skip_outbound_services, 77
skip_remote_sensitive_URIs, 69
skip_scan_nets_16, 78
skip_scan_nets_24, 78
skip_scan_sources, 78
skip_services, 77
skip_unexpected, 66
skip_unexpected_net, 66
smtp_hot_cmds, 80
smtp_legal_cmds, 80
smtp_log, 79
smtp_relay_table, 80
smtp_sensitive_cmds, 80
smtp_session_by_content_hash, 81
smtp_session_by_message_id, 80
smtp_session_by_recipient, 80
smtp_sessions, 79
software_file, 81
software_ident_by_major, 81
software_table, 81
ssh_len_conns, 60
ssh_log, 81
ssh_min_num_pkts, 60
ssh_min_ssh_pkts_ratio, 60
ssh_sig_disabled, 59
step_log, 81
stp_common_host_thresh, 82
stp_delta, 82
stp_demux_disabled, 82
stp_idle_min, 82
stp_random_pair_thresh, 82
stp_ratio_thresh, 82
stp_scale, 82
suppress_pm_log, 75
suppress_scan_checks, 76
table_expire_interval, 62
tag_to_conn_map, 81

    tcp_attempt_delayv, 61
    tcp_close_delay, 61
    tcp_connection_linger, 61
    tcp_match_undelivered, 61
    tcp_partial_close_delay, 61
    tcp_reassembler_ports_orig, 62
    tcp_reassembler_ports_resp, 62
    tcp_reset_delay, 61
    tcp_session_timer, 61
    tcp_storm_interarrival_thresh, 62
    tcp_storm_thresh, 62
    tcp_SYN_ack_ok, 61
    tcp_SYN_timeout, 61
    telnet_sig_3byte_conns, 60
    telnet_sig_3byte_disabled, 59
    telnet_sig_conns, 60
    telnet_sig_disabled, 59
    terminate_successful_inbound_service, 68
    tftp_alert_count, 83
    udp_did_summary, 83
    udp_rep_count, 83
    udp_req_count, 83
    vert_scan_thresholds, 76
    watchdog_interval, 62
    weird_action, 83
    weird_action_filters, 83
    weird_do_not_ignore_repeats, 83
    weird_ignore_host, 83
    weird_log, 83
    worm_list, 83
    worm_log, 83
    worm_type_list, 83
    worm_URIs, 69
prefixes, 17, 19
    network, 29, 91, 105
premature_connection_reuse ("weird" event), 159
preserved_net variable, 57
preserved_subnet variable, 57
priming Bro's private DNS cache, 18
print keyword, 43
print statement, 43
print-filter analyzer, 94
printf, 86
process_HTTP_data variable, 69
process_HTTP_replies variable, 69
process_smtp_relay variable, 79
processing
    avoiding, 90
prog unavail (RPC status code), 146
programs
    cf, 20
    hf, 20
PTR_scan event, 65
public_ident_systems variable, 70
public_ident_user_ids variable, 70

-r flag, 17
reactive firewall, 113, 114
&read_expire attribute, 36
reading tcpdump files, 17
reading_live_traffic function, 89
record, *see* types, record
    connection, 96
    ftp_port, 89
record constructor expression, 52
record field access expression, 52
record field test expression, 52
record_connection function, 105
recorded traffic, 89
recording connections, 105
recording packets, 90
records, 30–33, 89, 96–98, 101, 118, 122, 137, 150, 168
    assignment, 32–33
    check_info, 137
    conn_id, 96, 97
    connection, 96–98
    dns_mapping, 118
    endpoint, 96, 97, 101
    field attributes, 32
    fields, 30
        accessing, 32
        legal names, 31
    ftp_port, 89
    ftp_session_info, 122
    net_stats, 96
    signature_state, 168
    ssl_connection_info, 150
    x509, 150
red connection summary file, 102
&redef attribute, 56
redefining functions, 40
redefining variables, 56
refinement, 56
regular expressions, *see* patterns
REJ connection state, 102
rejected connections, 99
relational expression, 48
relationals
    address, 29
    net, 30
    numeric, 24
    string, 25
    temporal, 28
relative time, 27
relay_log variable, 80
remote procedure call (RPC), 142
remote_code_red_response_pgm variable, 63
repeat text, 132
repeat text (VMS), 132

repeat_without_username (login confusion state), 132
repeated_SYN_reply_wo_ack ("weird" event), 159
repeated_SYN_with_ack ("weird" event), 159
replication of packets, 148
report_accounts_tried variable, 77, 112
report_backscatter variable, 78
report_outbound_peer_scan variable, 76, 112
report_peer_scan variable, 76, 112
report_port_scan variable, 76
report_rejected_PTR_factor variable, 65
report_rejected_PTR_thresh variable, 65
report_remote_accounts_tried variable, 77, 112
report_weird function, 156
report_weird_conn function, 156
report_weird_orig function, 156
req_addr, 118
  dns_mapping field, 118
req_host, 118
  dns_mapping field, 118
request, 123
  ftp_session_info field, 123
request_t, 123
  ftp_session_info field, 123
reserved multicast addresss, 144
  sun-rpc.mcast.net, 144
reset connections, 100
resp, 96
resp_h, 96
  conn_id field, 96
resp_p, 96
  conn_id field, 96
responder_environment (login confusion state), 132
responder_RPC_call ("weird" event), 159
restrict_filter global variable, 17
restrict_filter variable, 84, 93–94
restricting traffic, 93
retransmission
  inconsistent, 159, 163
return keyword, 45
return statement, 45
rewrite_finger_trace variable, 65
rewrite_ident_trace variable, 70
rewriting_http_trace variable, 63
rewriting_smtp_trace variable, 63
rewt username, 121, 131
rexmit_inconsistency event, 163
.rhosts, 124, 130, 131, 136, 138
ridiculous field width or precision, 86
) operator, 46, 48
RLIMIT_NOFILE a, 39
Rlogin
  session state, 87, 90
  sessions, 130
  weird events, 157
rlogin_conns variable, 60
rlogin_id_okay_if_no_password_exposed variable, 74, 136
rlogin_sig_1byte_disabled variable, 59
rlogin_sig_disabled variable, 59
rlogin_text_after_rejected ("weird" event), 159
root
  backdoors, 131
  Bro not running as, 13
  setuid, 133
root_backdoor_sig_conns variable, 60
root_backdoor_sig_disabled variable, 59
root_servers variable, 78
router_prompts variable, 73, 135
routing
  split, 159
RPC (Remote Procedure Call), 142
  reserved multicast address, 144
  weird events, 157
RPC status codes, 146
  auth error, 146
  garbage args, 146
  mismatch, 146
  ok, 146
  prog unavail, 146
  system err, 146
  timeout, 146
  unknown, 146
RPC_do_not_complain variable, 75
RPC_dump_okay variable, 75, 144
RPC_okay variable, 75, 143
RPC_okay_nets variable, 75, 143
RPC_okay_services variable, 75, 143
rpc_prog function, 144
rpc_programs variable, 75, 143
RPC_rexmit_inconsistency ("weird" event), 159
RPC_server_map variable, 64
rpc_timeout variable, 62
RST control packet, 18, 100
RST termination
  causing undetermined connection size, 125
RST_storm ("weird" event), 159
RST_with_data ("weird" event), 159
RSTO connection state, 102
RSTOS0 connection state, 102
RSTR connection state, 102
RSTRH connection state, 102
rule_actions variable, 76
rule_file variable, 76
run-time error
  bad address mask, 88
  bad fmt date argument, 87
  bad fmt editing character, 86
  bad fmt field width, 86
  bad fmt floating-point argument, 87

  bad fmt format specifier, 87
  bad fmt integer argument, 87
  bad fmt precision, 86
  bad length argument (not a table or set), 88
  can’t open, 19
  converting an IPv6 address to net, 91
  embedded NUL, 91
  mkdir failure, 88
  no such connection, 91
  non-existing connection, 86
  not a login connection, 87, 90
  not a TCP connection, 87
  NULs
    terminator missing, 91
  watchdog timer expired, 18
  wrong number of fmt arguments, 87
  wrong number of length arguments, 88
running Bro, 12
running outside scripts or executables, 90

`-s` flag, 17
`S0` connection state, 102
`S1` connection state, 102
`S2` connection state, 102
`S3` connection state, 102
`same_local_net_is_spoof` variable, 67, 106
save file
  control over what’s recorded, 90
  reading, 17
  writing, 18
scalars, 33
`scan` analyzer, 112
scan detection, 112–114
`scan_triples` variable, 77
scanning
  address, 112
  port, 112
  shutting down, 113, 114
  stealth, 99, 111, 114, 159
scans
  exploit, 148
`schedule` keyword, 50
scheduling events, 50
scoping of variables, 53
script kiddies, 107
scripting
  general, 88
scripts
  running, 90
  standard, 92–164
search path, 19
searching for strings, 25
`sec` (seconds) interval unit, 27
semi-colon statement termination, 43
sensitive /24 destination networks, 108
sensitive /24 source networks, 107
sensitive destination addresses, 107
sensitive filenames, 124, 131
  `eggdrop`, 124
sensitive information
  inadvertently exposed, 130
sensitive login inputs, 131
  `eggdrop`, 131
sensitive patterns, 133
  `PATH_UTMP`, 133
sensitive POST URIs, 127
  `wwwroot`, 127
sensitive services
  confused with ephemeral ports, 109
sensitive source addresses, 107
sensitive usernames, 121
`sensitive_lookup_hosts` variable, 64
`sensitive_post_URIs` variable, 69, 127
`sensitive_URIs` variable, 69, 127
sensitivity associated with a connection, 98
sequence numbers
  connection originator, 87
  connection responder, 87
`server_cert`, 151
  `ssl_connection_info` field, 151
`service`, 98
  `connection` field, 98
service associated with a connection, 98, 101, 103, 105
`service_name` function, 105
services
  allowable, 108
  allowed to a particular host, 108
  allowed to particular host pairs, 108
  fatal if inbound, 109
  forbidden, 109
  forbidden if attempted, 109
  forbidden if inbound, 109
`set`, *see* types, `set`
SET portmapper call, 142
set size, 88
set type, 38–39
`set_buf` function, 89
`set_contents_file` function, 89
`set_login_state` function, 90
`set_record_packets` function, 90
`setrlimit` system calls, 39
setuid root, 133
`SF` connection state, 102
`sgiweb` username, 121
sh, 90
`SH` connection state, 102
shadow, 127
shadowing, 94
shallow copy, 32, 38
shedding load, 90
shell escape, 90

shell scripts
  drop-connectivity, 114
short-circuit && “and” operator, 22, 47
short-circuit || “or” operator, 22, 47
SHR connection state, 102
shut_down_all_scans variable, 113
shut_down_scans variable, 113
shut_down_thresh variable, 113
shutting down scans, 113, 114
sig_actions variable, 149
SIG_FILE action, 149
SIG_IGNORE action, 149
SIG_LOG action, 149
SIG_QUIET action, 149
SIGHUP, 95
SIGINT, 95
signal handling, 95
signature analysis, 148
signature analyzer, 148
signature_match event, 148
signature_state record, 168
signatures
  log file, 148
SIGTERM, 95
simultaneous open, 159
simultaneous_open (“weird” event), 159
site addresses, 106
site analyzer, 105
site-specific
  functions, 106
  information, 105
  variables, 105–106
size, 96
  endpoint field, 96
  of table or set, 88
size of connection, 101, 103
skip_accounts_tried variable, 77, 112
skip_authentication variable, 73, 134
skip_clear_ssh_reports variable, 82
skip_further_processing function, 90
skip_logins_to variable, 74, 136
skip_outbound_services variable, 77, 113
skip_remote_sensitive_URIs variable, 69
skip_scan_nets_16 variable, 78
skip_scan_nets_24 variable, 78, 113
skip_scan_sources variable, 78, 113
skip_services variable, 77
skip_unexpected variable, 66, 124
skip_unexpected_net variable, 66, 124
(skipped) authentication annotation, 141
smashdu.c exploit tool, 133
smtp_hot_cmds variable, 80
smtp_legal_cmds variable, 80
smtp_log variable, 79
smtp_relay_table variable, 80
smtp_sensitive_cmds variable, 80
smtp_session_by_content_hash variable, 81
smtp_session_by_message_id variable, 80
smtp_session_by_recipient variable, 80
smtp_sessions variable, 79
smurf attacks, 133
sniffer logs, 133
sniffing, 130
software_file variable, 81
software_ident_by_major variable, 81
software_table variable, 81
source code
  for Bro, 12
split routing, 159
spontaneous_FIN (“weird” event), 159
spontaneous_RST (“weird” event), 159
spoofing
  allowable services, 107
  detection, 106, 110
spook detection, 108
sprintf, 86
ssh_len_conns variable, 60
ssh_log variable, 81
ssh_min_num_pkts variable, 60
ssh_min_ssh_pkts_ratio variable, 60
ssh_sig_disabled variable, 59
SSL
  analysis, 149
  connection information, 150
  log file, 152
  x509, 150
SSL analyzer, 149
SSL session summary file, 152
ssl_analyze_certificates variable, 151
ssl_certificate event, 153
ssl_certificate_seen event, 153
ssl_compare_cipherspecs variable, 151
ssl_conn_alert event, 153
ssl_conn_attempt event, 152
ssl_conn_established event, 153
ssl_conn_reused event, 153
ssl_conn_server_reply event, 153
ssl_conn_weak event, 154
ssl_connection_info record, 150, 151
ssl_max_cipherspec_size variable, 151
ssl_store_cert_path variable, 151
ssl_store_certificates variable, 151
ssl_store_key_material variable, 152
ssl_verify_certificates variable, 151
standard scripts, 92–164
start time of a connection, 98, 101
start_time, 98
  connection field, 98
startup
  event, 94

- transients, 159

- `state`, 96
  - `endpoint` field, 96
  - of a Telnet/Rlogin session, 87, 90
- state management, 36
- state of connection, 101, 103
- state of login connections
  - `LOGIN_STATE_AUTHENTICATE`, 87
  - `LOGIN_STATE_CONFUSED`, 87
  - `LOGIN_STATE_LOGGED_IN`, 87
  - `LOGIN_STATE_SKIP`, 87
- statements, 43–46
  - *compound*, 45
  - *expression*, 43
  - *null*, 45
  - `add`, 45
  - `break`, 45
  - `const`, 45
  - `delete`, 45
  - `event`, 44
  - `for`, 44
  - `if`, 44
  - `local`, 45
  - `log`, 43
  - multi-line, 43
  - `next`, 44
  - `print`, 43
  - `return`, 45
  - semi-colon termination, 43
- static typing, 22
- statistical analysis, 148
- statistics, 148
  - `endian_type`, 148
  - `num_in_order`, 148
  - `num_OO`, 148
  - `num_pkts`, 148
  - `num_repl`, 148
  - `num_rxmit`, 148
  - `num_rxmit_bytes`, 148
- *stderr*, 84, 90, 115
- *stdout*, 43, 90
- stealth scans, 99, 111, 114, 159
- `step_log` variable, 81
- storms, 158
- `stp_common_host_thresh` variable, 82
- `stp_delta` variable, 82
- `stp_demux_disabled` variable, 82
- `stp_idle_min` variable, 82
- `stp_random_pair_thresh` variable, 82
- `stp_ratio_thresh` variable, 82
- `stp_scale` variable, 82
- *strftime*, 87
- `string`, *see* types, `string`
  - extraction, 90
  - formatting, 86
- string constants
  - NUL terminated, 25
- `string with embedded NUL`, 91
- `string without NUL terminator`, 91
- `<string-with-NUL>` error value, 91
- strings, 24–25
  - cleaned up, 85
  - concatenation, 85
  - length, 85
  - termination with NULs, 91
- *strlen*, 85
- *strstr*, 86
- sub-tables
  - lack of, 37
- `sub_bytes` function, 90
- subnets, 29, 88, 91, 105
- substrings, 90
- subtraction
  - numeric, 23
  - temporal, 28
- `sun-rpc.mcast.net` reserved multicast address, 144
- `suppress_pm_log` variable, 75, 144
- `suppress_scan_checks` variable, 76
- SYN control packet, 18, 100
- `SYN_after_close` (“weird” event), 160
- `SYN_after_partial` (“weird” event), 160
- `SYN_after_reset` (“weird” event), 160
- `SYN_inside_connection` (“weird” event), 160
- `SYN_seq_jump` (“weird” event), 160
- `SYN_with_data` (“weird” event), 160
- *syslog*, 44
- syslog levels, 44
  - `LOG_NOTICE`, 44
- system callss, 39
  - `setrlimit`, 39
- system configuration, 13
- `system err` (RPC status code), 146
- `system` function, 90

- `T`, 22
- T/TCP, 160
- `table`, *see* types, `table`
- table size, 88
- `table_expire_interval` variable, 62
- tables, 33–38
  - clearing entries, 38
- `tag_to_conn_map` variable, 81
- TCP
  - analysis, 100
  - checksum error, 157
  - Christmas packet, 160
  - connections, 98
  - corrupted header, 162
  - events, 99
  - fragments, 121
  - transaction, 160

  - weird events, 157
- `tcp` analyzer, 100
- TCP control packets (SYN/FIN/RST), 18, 100
- TCP vs. UDP ports, 87
- TCP Wrappers
  - reset vs. rejected connections, 99
- TCP-specific connection events, 99
- `tcp_attempt_delayv` variable, 61
- `TCP_christmas` (“weird” event), 160
- `tcp_close_delay` variable, 61
- `tcp_connection_linger` variable, 61
- `tcp_match_undelivered` variable, 61
- `tcp_partial_close_delay` variable, 61
- `tcp_reassembler_ports_orig` variable, 62
- `tcp_reassembler_ports_resp` variable, 62
- `tcp_reset_delay` variable, 61
- `tcp_session_timer` variable, 61
- `tcp_storm_interarrival_thresh` variable, 62
- `tcp_storm_thresh` variable, 62
- `tcp_SYN_ack_ok` variable, 61
- `tcp_SYN_timeout` variable, 61
- `tcpdump`, 13, 15, 17, 18, 93, 94
  - bugs, 94
  - filters, 17, 93
  - merging save files, 18
  - reading save files, 15, 17
  - running concurrently with Bro, 13
  - shadow, 94
  - turning off optimization, 94
  - writing save files, 18
- Telnet
  - options, 130
    - authentication, 141
    - bad, 141
    - bad termination, 141
    - encryption, 142
    - environment, 132
    - inconsistent, 140
  - session state, 87, 90
  - sessions, 130
- `telnet_sig_3byte_conns` variable, 60
- `telnet_sig_3byte_disabled` variable, 59
- `telnet_sig_conns` variable, 60
- `telnet_sig_disabled` variable, 59
- temporal
  - addition, 28
  - constants, 27
  - division, 28
  - multiplication, 28
  - negation, 27
  - relationals, 28
  - subtraction, 28
  - types, 27
- *TERM* signal, 95
- terminal type backdoors, 134
  - `VT666`, 134
- `terminate_connection` function, 105
- `terminate_successful_inbound_service` variable, 68, 109
- terminating connections forcibly, 105
- termination event, 95
- text
  - formatting, 86
- TFreak, 133
- `tftp_alert_count` variable, 83
- `time`, *see* types, `time`, 27–28
  - clock, 86, 88
  - packet, 88
- `timeout` (RPC status code), 146
- timer expiration, 50, 84
- timers, 50
- timestamps
  - mapping to readable form, 20
- `to_lower` function, 90
- `to_net` function, 91
- `to_upper` function, 91
- *tolower*, 90
- `too few arguments for format`, 87
- `too many arguments for format`, 87
- *toupper*, 91
- trace file
  - control over what’s recorded, 90
  - reading, 17
  - writing, 18
- traffic
  - live vs. recorded, 44, 89
  - restricting, 93
- transaction TCP, 160
- transients
  - startup, 159
- trojaning, 133
- truncated headers, 162
- `truncated_header` (“weird” event), 162
- `truncated_IP` (“weird” event), 162
- tunneling, 140
- type casting
  - not provided in Bro, 42
- type inference, 55
- type-ahead
  - maximum allowed, 132
- types
  - `addr`, 21
  - `bool`, 21, 22
  - conversion, 22
    - automatic, 22
  - `count`, 21, 23
  - `double`, 21, 23
  - `enum`, 21, 24
  - *enumeration*, 21
  - `event`, 22

  - file, 21
  - function, 21
  - int, 21, 23
  - interval, 21, 27
  - net, 21
  - *numeric*, 21, 23–24
    - bool not numeric, 23
    - intermixing, 23
  - overview, 21
  - pattern, 21, 25
  - port, 21
  - record, 21
  - set, 21
  - string, 21, 24
  - table, 21
  - *temporal*, 21
  - time, 21, 27
- types, need fors, 98
  - union, 98
- typing
  - static, 22
- typing of variables, 54

- UDP
  - analysis, 100
  - checksum error, 157
  - “connections”, 98
  - fragments, 121
  - length mismatch, 160
  - timeout, 98
  - weird events, 157
- udp analyzer, 100
- UDP_datagram_length_mismatch (“weird” event), 160
- udp_did_summary variable, 83
- udp_rep_count variable, 83
- udp_reply event, 100
- udp_req_count variable, 83
- udp_request event, 100
- unanalyzed data, 158
- undirectional analysis, 159
- union type
  - need for, 88
- union types, need for, 98
- Unix analysis, 130
- Unix support, 12
- Unix timestamps, 20
- unknown (RPC status code), 146
- unpaired_RPC_response (“weird” event), 160
- UNSET portmapper call, 142
- unsolicited_SYN_response (“weird” event), 160
- unusual events, 154–163
  - prevalence in actual network traffic, 154
- usage message, 17
- usec (microseconds) interval unit, 27
- user, 123
  - ftp_session_info field, 123
- $USER environment variable, 132
- user keystrokes
  - analysis, 130
  - editing, 131
- Username: (VMS login prompt), 132
- username_with_embedded_repeat (login confusion state), 132
- usernames, 121, 129, 131
  - 4Dgifts, 121
  - daemon, 121, 129
  - extracting, 130, 131
  - EZsetup, 121
  - missing, 132
  - <none>, 132
  - nuucp, 121, 129
  - OutOfBox, 121
  - repeated, 132
  - rewt, 121, 131
  - sensitive, 121
  - sgiweb, 121
  - uucp, 121, 129
- */usr/local/lib/bro/* policy directory, 19
- utilities
  - *fetch*, 133
  - *flex*, 25
  - *lex*, 25
  - *ls*, 133
  - *lynx*, 133
- utility programs
  - *cf*, 20
  - *hf*, 20
- uucp username, 121, 129

- -v flag, 18
- valid, 118
  - dns_mapping field, 118
- values
  - overview, 21
- vantage point, 159
- variable declarations, 53, 54
  - const, 54
  - global, 53
  - local, 53
- *variable* expression, 46
- variables
  - accounts_tried, 77
  - active_conn, 57, 117
  - actually_rejected_PTR_anno, 64
  - addl_web, 77, 113
  - alert_action_filters, 57
  - alert_file, 57
  - allow_16_net_pairs, 67, 107
  - allow_excessive_ntp_requests, 74
  - allow_pairs, 67, 107
  - allow_PTR_scans, 65

allow_service_pairs, 67
allow_services, 67, 108
allow_services_pairs, 108
allow_services_to, 67, 108
allow_spoof_services, 67, 107
always_hot_ids, 68, 121
always_hot_login_ids, 74, 136
anon_log, 57
anonymize_ip_addr, 62
any_RPC_okay, 75, 144
attributes, 55
backdoor_annotate_standard_ports, 59
backdoor_demux_disabled, 58
backdoor_demux_skip_tags, 58
backdoor_ignore_dst_addrs, 58
backdoor_ignore_ports, 58
backdoor_ignore_src_addrs, 58
backdoor_log, 58
backdoor_min_7bit_ascii_ratio, 58
backdoor_min_bytes, 58
backdoor_min_normal_line_ratio, 58
backdoor_min_num_lines, 58
backdoor_prompts, 73, 133
backdoor_standard_ports, 58
backdoor_stat_backoff, 59
backdoor_stat_period, 58
backscatter_ports, 78
bro_log_file, 84, 115
can_drop_connectivity, 113
capture_filter, 84, 93–94
check_relay_3, 74
check_relay_4, 74
code_red_list1, 63
code_red_list2, 63
code_red_log, 63
conn_tag_info, 81
constant, 45
demux_dir, 64
demuxed_conn, 64
detected_stones, 81
did_PTR_scan_event, 65
did_sigconns, 60
did_ssh_version, 81
did_stone_summary, 82
direct_login_prompts, 84, 134
discarder_maxlen, 84
display_pairs, 81
distinct_answered_PTR_requests, 65
distinct_backscatter_peers, 78
distinct_peers, 76
distinct_ports, 76
distinct_PTR_requests, 64
distinct_rejected_PTR_requests, 64
dns_interesting_changes, 65, 119
dns_log, 64
dns_session_timeout, 62
dns_sessions, 64
done_with_network, 84
edited_input_trouble, 72, 131
excessive_ntp_request, 74
flag_rejected_service, 68, 109
flag_successful_inbound_service, 68, 109
flag_successful_service, 67, 109
forbidden_id_patterns, 68, 121
forbidden_ids, 68, 121
forbidden_ids_if_no_password, 68, 121
ftp_data_expected, 66
ftp_data_expected_session, 66
ftp_excessive_filename_len, 66
ftp_excessive_filename_trunc_len, 66
ftp_guest_ids, 65, 123
ftp_hot_cmds, 66
ftp_hot_files, 66, 124
ftp_hot_guest_files, 66, 124
ftp_ignore_invalid_PORT, 66
ftp_ignore_privileged_PASVs, 66
ftp_log, 65
ftp_not_actually_hot_files, 124
ftp_sessions, 65
ftp_sig_disabled, 59
ftp_skip_hot, 65, 123
full_input_trouble, 72
full_output_trouble, 72
gnutella_sig_disabled, 59
gtld_servers, 78
have_FTP, 63
have_skip_remote_sensitive_URIs, 69
have_SMTP, 63
have_stats, 63
heartbeat_interval, 62
horiz_scan_thresholds, 76, 149
hot_conns_reported, 63
hot_dst_24nets, 67, 108
hot_dsts, 67, 107
hot_ident_exceptions, 70, 129
hot_ident_ids, 70, 129
hot_ids, 68, 122
hot_login_ids, 74, 136
hot_names, 65, 120
hot_src_24nets, 67, 107
hot_srcs, 67, 107
hot_ssh_orig_ports, 134
hot_telnet_orig_ports, 73, 134
hot_terminal_types, 73, 133
http_abstract_max_length, 69
http_log, 68
http_proxy_sig_disabled, 60
http_sessions, 68
http_sig_disabled, 60
icmp_flows, 70

ignore_checksums, 60
inactivity_timeout, 61
include_HTTP_abstract, 69
initialization, 55
input_trouble, 72, 131
input_wait_for_output, 72
interconn_conns, 70
interconn_default_pkt_size, 70
interconn_demux_disabled, 72
interconn_ignore_standard_ports, 72
interconn_log, 70
interconn_max_interarrival, 70
interconn_max_keystroke_pkt_size, 70
interconn_min_7bit_ascii_ratio, 71
interconn_min_alpha, 72
interconn_min_bytes, 71
interconn_min_duration, 71
interconn_min_gamma, 72
interconn_min_interarrival, 70
interconn_min_normal_line_ratio, 71
interconn_min_num_lines, 71
interconn_min_num_pkts, 71
interconn_min_ssh_pkts_ratio, 71
interconn_ssh_len_disabled, 71
interconn_standard_ports, 72
interconn_stat_backoff, 71
interconn_stat_period, 71
interfaces, 84
kazaa_sig_disabled, 59
last_stat, 63
last_stat_time, 64
local, 45
local_16_nets, 79, 105
local_24_nets, 79, 105
local_code_red_response_pgm, 63
local_mail_addr, 79
local_nets, 79, 105
log_HTTP_data, 69
login_failure_msgs, 73, 134
login_non_failure_msgs, 73, 135
login_prompts, 73, 134
login_sessions, 74
login_success_msgs, 73, 135
login_timeouts, 73, 136
maintain_http_sessions, 69
max_finger_request_len, 65
max_request_length, 120
max_timer_expires, 84
mime_log, 74
mime_sessions, 74
modifiability, 54
napster_sig_disabled, 59
neighbor_16_nets, 79, 106
neighbor_24_nets, 106
neighbor_nets, 79, 106
never_shut_down, 113
NFS_services, 75, 143
NFS_world_servers, 75, 143
non_analyzed_lifetime, 61
non_ASCII_hosts, 73, 136
non_backdoor_prompts, 73, 133
ntp_session_timeout, 62
num_accounts_tried, 77
num_backscatter_peers, 78
num_distinct_peers, 76
num_distinct_ports, 76
num_dns_sessions, 64
num_scan_triples, 77
okay_to_lookup_sensitive_hosts, 64
omit_rewrite_place_holder, 63
output_trouble, 72, 133
overview, 53
partial_connection_ok, 60
port_names, 74, 98, 103, 115
possible_port_scan_thresh, 77, 112
possible_scan_sources, 77
preserved_net, 57
preserved_subnet, 57
process_HTTP_data, 69
process_HTTP_replies, 69
process_smtp_relay, 79
public_ident_systems, 70
public_ident_user_ids, 70
redefining, 56
refinement, 56
relay_log, 80
remote_code_red_response_pgm, 63
report_accounts_tried, 77, 112
report_backscatter, 78
report_outbound_peer_scan, 76, 112
report_peer_scan, 76, 112
report_port_scan, 76
report_rejected_PTR_factor, 65
report_rejected_PTR_thresh, 65
report_remote_accounts_tried, 77, 112
restrict_filter, 84, 93–94
rewrite_finger_trace, 65
rewrite_ident_trace, 70
rewriting_http_trace, 63
rewriting_smtp_trace, 63
rlogin_conns, 60
rlogin_id_okay_if_no_password_exposed, 74, 136
rlogin_sig_1byte_disabled, 59
rlogin_sig_disabled, 59
root_backdoor_sig_conns, 60
root_backdoor_sig_disabled, 59
root_servers, 78
router_prompts, 73, 135
RPC_do_not_complain, 75

RPC_dump_okay, 75, 144
RPC_okay, 75, 143
RPC_okay_nets, 75, 143
RPC_okay_services, 75, 143
rpc_programs, 75, 143
RPC_server_map, 64
rpc_timeout, 62
rule_actions, 76
rule_file, 76
same_local_net_is_spoof, 67, 106
scan_triples, 77
scope, 46
scoping, 53
sensitive_lookup_hosts, 64
sensitive_post_URIs, 69, 127
sensitive_URIs, 69, 127
shut_down_all_scans, 113
shut_down_scans, 113
shut_down_thresh, 113
sig_actions, 149
skip_accounts_tried, 77, 112
skip_authentication, 73, 134
skip_clear_ssh_reports, 82
skip_logins_to, 74, 136
skip_outbound_services, 77, 113
skip_remote_sensitive_URIs, 69
skip_scan_nets_16, 78
skip_scan_nets_24, 78, 113
skip_scan_sources, 78, 113
skip_services, 77
skip_unexpected, 66, 124
skip_unexpected_net, 66, 124
smtp_hot_cmds, 80
smtp_legal_cmds, 80
smtp_log, 79
smtp_relay_table, 80
smtp_sensitive_cmds, 80
smtp_session_by_content_hash, 81
smtp_session_by_message_id, 80
smtp_session_by_recipient, 80
smtp_sessions, 79
software_file, 81
software_ident_by_major, 81
software_table, 81
ssh_len_conns, 60
ssh_log, 81
ssh_min_num_pkts, 60
ssh_min_ssh_pkts_ratio, 60
ssh_sig_disabled, 59
ssl_analyze_certificates, 151
ssl_compare_cipherspecs, 151
ssl_max_cipherspec_size, 151
ssl_store_cert_path, 151
ssl_store_certificates, 151
ssl_store_key_material, 152
ssl_verify_certificates, 151
step_log, 81
stp_common_host_thresh, 82
stp_delta, 82
stp_demux_disabled, 82
stp_idle_min, 82
stp_random_pair_thresh, 82
stp_ratio_thresh, 82
stp_scale, 82
suppress_pm_log, 75, 144
suppress_scan_checks, 76
table_expire_interval, 62
tag_to_conn_map, 81
tcp_attempt_delayv, 61
tcp_close_delay, 61
tcp_connection_linger, 61
tcp_match_undelivered, 61
tcp_partial_close_delay, 61
tcp_reassembler_ports_orig, 62
tcp_reassembler_ports_resp, 62
tcp_reset_delay, 61
tcp_session_timer, 61
tcp_storm_interarrival_thresh, 62
tcp_storm_thresh, 62
tcp_SYN_ack_ok, 61
tcp_SYN_timeout, 61
telnet_sig_3byte_conns, 60
telnet_sig_3byte_disabled, 59
telnet_sig_conns, 60
telnet_sig_disabled, 59
terminate_successful_inbound_service, 68, 109
tftp_alert_count, 83
typing, 54
udp_did_summary, 83
udp_rep_count, 83
udp_req_count, 83
vert_scan_thresholds, 76, 149
watchdog_interval, 62
weird_action, 83, 154
weird_action_filters, 83, 155
weird_do_not_ignore_repeats, 83, 156
weird_ignore_host, 83, 155
weird_log, 83
worm_list, 83
worm_log, 83
worm_type_list, 83
worm_URIs, 69
x509_trusted_cert_path, 151

version, 150
  ssl_connection_info field, 150
version message, 18
vert_scan_thresholds variable, 76, 149
vertical exploit scans, 148
VMS input editing, 132

VMS login prompts, 132
    Username:, 132
VT666 terminal type backdoor, 134

-W flag, 18
-w flag, 18
walld, 143, 144, 146, 147
watchdog, 18
watchdog timer expired
    run-time error, 18
WATCHDOG_INTERVAL internal variable, 18
watchdog_interval variable, 62
“weird” event, 156–163
    active_connection_reuse, 156
    bad_HTTP_reply, 157
    bad_HTTP_version, 157
    bad_ICMP_checksum, 157
    bad_ident_reply, 160
    bad_ident_request, 161
    bad_IP_checksum, 162
    bad_pm_port, 163
    bad_rlogin_prolog, 157
    bad_RPC, 157
    bad_RPC_program, 157
    bad_SYN_ack, 157
    bad_TCP_checksum, 157
    bad_TCP_header_len, 162
    bad_UDP_checksum, 157
    baroque_SYN, 157
    blank_in_HTTP_request, 157
    connection_originator_SYN_ack, 157
    data_after_reset, 157
    data_before_established, 157
    excess_RPC, 157
    excessive_RPC_len, 157
    excessively_large_fragment, 161
    excessively_small_fragment, 161
    FIN_advanced_last_seq, 157
    FIN_after_reset, 158
    FIN_storm, 158
    fragment_inconsistency, 161
    fragment_overlap, 161
    fragment_protocol_inconsistency, 161
    fragment_size_inconsistency, 161
    fragment_with_DF, 162
    HTTP_unknown_method, 158
    HTTP_version_mismatch, 158
    ident_request_addendum, 161
    inappropriate_FIN, 158
    incompletely_captured_fragment, 162
    internally_truncated_header, 162
    Land_attack, 163
    multiple_HTTP_request_elements, 158
    multiple_RPCs, 158
    NUL_in_line, 158
    originator_RPC_reply, 158
    partial_finger_request, 158
    partial_ftp_request, 158
    partial_ident_request, 158
    partial_portmapper_request, 158
    partial_RPC, 158
    pending_data_when_closed, 158
    possible_split_routing, 159
    premature_connection_reuse, 159
    repeated_SYN_reply_wo_ack, 159
    repeated_SYN_with_ack, 159
    responder_RPC_call, 159
    rlogin_text_after_rejected, 159
    RPC_rexmit_inconsistency, 159
    RST_storm, 159
    RST_with_data, 159
    simultaneous_open, 159
    spontaneous_FIN, 159
    spontaneous_RST, 159
    SYN_after_close, 160
    SYN_after_partial, 160
    SYN_after_reset, 160
    SYN_inside_connection, 160
    SYN_seq_jump, 160
    SYN_with_data, 160
    TCP_christmas, 160
    truncated_header, 162
    truncated_IP, 162
    UDP_datagram_length_mismatch, 160
    unpaired_RPC_response, 160
    unsolicited_SYN_response, 160
weird event summary file, 154
weird events, 154–163
    actions, 154
    additional handlers, 163
    generated by standard scripts, 163
    handled by conn_weird, 156
    handled by conn_weird_addl, 160
    handled by flow_weird, 161
    handled by net_weird, 162
    prevalence in actual network traffic, 154
weird module, 154
weird_action variable, 83, 154
weird_action_filters variable, 83, 155
weird_do_not_ignore_repeats variable, 83, 156
WEIRD_FILE action, 155
WEIRD_IGNORE action, 155
weird_ignore_host variable, 83, 155
weird_log variable, 83
WEIRD_LOG_ALWAYS action, 155
WEIRD_LOG_ONCE action, 155
WEIRD_LOG_PER_CONN action, 155
WEIRD_LOG_PER_ORIG action, 155
WEIRD_UNSPECIFIED action, 155
whitespace
    in statements, 43

width
  of formatted strings, 86
Windows
  not supported, 12
`worm_list` variable, 83
`worm_log` variable, 83
`worm_type_list` variable, 83
`worm_URIs` variable, 69
write file
  control over what’s recorded, 90
`&write_expire` attribute, 36
writing `tcpdump` files, 18
`wrong number of` ***`fmt`*** `arguments`, 87
wrong number of fmt arguments
  run-time error, 87
wrong number of length arguments
  run-time error, 88
www.anticode.com, 133
`wwwroot` sensitive POST URI, 127

`x509` record, 150
`x509_trusted_cert_path` variable, 151

yield
  of a table, 33
ypserv, 144