# Bro Documentation

*Release 2.4.1*

The Bro Project

October 01, 2016

CHAPTER

ONE

# INTRODUCTION SECTION

## 1.1 Introduction

**Contents**

### 1.1.1 Overview

Bro is a passive, open-source network traffic analyzer. It is primarily a security monitor that inspects all traffic on a link in depth for signs of suspicious activity. More generally, however, Bro supports a wide range of traffic analysis tasks even outside of the security domain, including performance measurements and helping with trouble-shooting.

The most immediate benefit that a site gains from deploying Bro is an extensive set of *log files* that record a network's activity in high-level terms. These logs include not only a comprehensive record of every connection seen on the wire, but also application-layer transcripts such as, e.g., all HTTP sessions with their requested URIs, key headers, MIME types, and server responses; DNS requests with replies; SSL certificates; key content of SMTP sessions; and much more. By default, Bro writes all this information into well-structured tab-separated log files suitable for post-processing with external software. Users can however also chose from a set of alternative output formats and backends to interface directly with, e.g., external databases.

In addition to the logs, Bro comes with built-in functionality for a range of analysis and detection tasks, including extracting files from HTTP sessions, detecting malware by interfacing to external registries, reporting vulnerable versions of software seen on the network, identifying popular web applications, detecting SSH brute-forcing, validating SSL certificate chains, and much more.

However, the key to understanding Bro lies in realizing that even though the system comes with such powerful functionality out of the box, fundamentally it represents a *platform* for traffic analyses that's fully customizable and extensible: Bro provides users with a domain-specific, Turing-complete *scripting language* for expressing arbitrary analysis tasks. Conceptually, you can think of Bro as a "domain-specific Python" (or Perl): just like Python, the system comes with a large set of pre-built functionality (the "standard library"), yet you are not limited to what the system ships with but can put Bro to use in novel ways by writing your own code. Indeed, all of Bro's default analyses, including all the logging, is the result of such scripts; there's no specific analysis hard-coded into the core of system.

Bro runs on commodity hardware and hence provides a low-cost alternative to expensive proprietary solutions. Despite the price tag, however, Bro actually goes far beyond the capabilities of other network monitoring tools, which typically remain limited to a small set of hard-coded analysis tasks. We emphasize in particular that Bro is *not* a classic signature-based intrusion detection system (IDS). While it supports such standard functionality as well, Bro's scripting language indeed facilitates a much broader spectrum of very different approaches to finding malicious activity, including semantic misuse detection, anomaly detection, and behavioral analysis.

A large variety of sites deploy Bro operationally for protecting their cyberinfrastructure, including many universities, research labs, supercomputing centers, open-science communities, and major corporations. Bro specifically targets high-speed, high-volume network monitoring, and an increasing number of sites are now using the system to monitor their 10GE networks, with some already moving on to 100GE links. Bro accommodates such high-performance settings by supporting scalable load-balancing: large sites typically run "Bro Clusters" in which a high-speed frontend load-balancer distributes the traffic across an appropriate number of backend PCs, all running dedicated Bro instances on their individual traffic slices. A central manager system coordinates the process, synchronizing state across the backends and providing the operators with a central management interface for configuration and access to aggregated logs. Bro's integrated management framework, BroControl, supports such cluster setups out-of-the-box.

### 1.1.2 Features

Bro supports a wide range of analyses through its scripting language. Yet even without further customization it comes with a powerful set of features.

- Deployment
  - Runs on commodity hardware on standard UNIX-style systems (including Linux, FreeBSD, and MacOS).
  - Fully passive traffic analysis off a network tap or monitoring port.
  - Standard libpcap interface for capturing packets.
  - Real-time and offline analysis.
  - Cluster-support for large-scale deployments.
  - Unified management framework for operating both standalone and cluster setups.
  - Open-source under a BSD license.
- Analysis
  - Comprehensive logging of activity for offline analysis and forensics.
  - Port-independent analysis of application-layer protocols.
  - Support for many application-layer protocols (including DNS, FTP, HTTP, IRC, SMTP, SSH, SSL).
  - Analysis of file content exchanged over application-layer protocols, including MD5/SHA1 computation for fingerprinting.
  - Comprehensive IPv6 support.
  - Tunnel detection and analysis (including Ayiya, Teredo, GTPv1). Bro decapsulates the tunnels and then proceeds to analyze their content as if no tunnel was in place.
  - Extensive sanity checks during protocol analysis.
  - Support for IDS-style pattern matching.
- Scripting Language
  - Turing-complete language for expression arbitrary analysis tasks.
  - Event-based programming model.

  - Domain-specific data types such as IP addresses (transparently handling both IPv4 and IPv6), port numbers, and timers.
  - Extensive support for tracking and managing network state over time.
- Interfacing
  - Default output to well-structured ASCII logs.
  - Alternative backends for ElasticSearch and DataSeries. Further database interfaces in preparation.
  - Real-time integration of external input into analyses. Live database input in preparation.
  - External C library for exchanging Bro events with external programs. Comes with Perl, Python, and Ruby bindings.
  - Ability to trigger arbitrary external processes from within the scripting language.

### 1.1.3 History

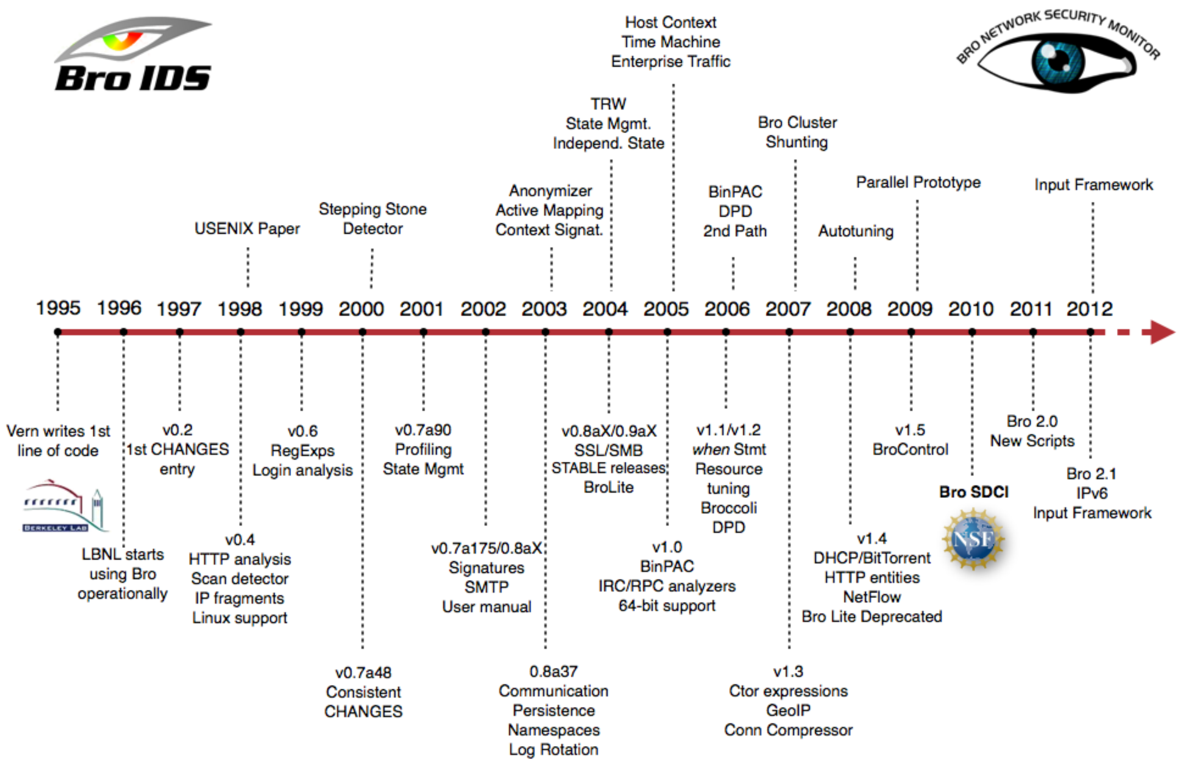

Bro IDS
BRO NETWORK SECURITY MONITOR
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
USENIX Paper
Stepping Stone Detector
Anonymizer Active Mapping Context Signat.
TRW State Mgmt. Independ. State
Host Context Time Machine Enterprise Traffic
BinPAC DPD 2nd Path
Bro Cluster Shunting
Autotuning
Parallel Prototype
Input Framework
Vern writes 1st line of code
LBNL starts using Bro operationally
v0.2 1st CHANGES entry
v0.4 HTTP analysis Scan detector IP fragments Linux support
v0.6 RegExps Login analysis
v0.7a48 Consistent CHANGES
v0.7a90 Profiling State Mgmt
v0.7a175/0.8aX Signatures SMTP User manual
0.8a37 Communication Persistence Namespaces Log Rotation
v0.8aX/0.9aX SSL/SMB STABLE releases BroLite
v1.0 BinPAC IRC/RPC analyzers 64-bit support
v1.1/v1.2 when Stmt Resource tuning Broccoli DPD
v1.3 Ctor expressions GeoIP Conn Compressor
v1.4 DHCP/BitTorrent HTTP entities NetFlow Bro Lite Deprecated
v1.5 BroControl
Bro SDCI
Bro 2.0 New Scripts
Bro 2.1 IPv6 Input Framework

Fig. 1.1: Timeline of Bro's History (click to enlarge).

Bro's history goes back much further than many people realize. Vern Paxson designed and implemented the initial version almost two decades ago. Vern began work on the code in 1995 as a researcher at the Lawrence Berkeley National Laboratory (LBNL). Berkeley Lab began operational deployment in 1996, and the USENIX Security Symposium published the original Bro paper in 1998 (later refined in a subsequent journal publication). In 2003, the National Science Foundation (NSF) began supporting research and advanced development on Bro at the International Computer Science Institute (ICSI), where Vern now leads the Networking and Security group. Over the years, a growing team of ICSI researchers and students kept adding novel functionality to Bro, while LBNL continued its support with funding from the Department of Energy (DOE).

Much of Bro's capabilities originate in academic research projects, with results often published at top-tier conferences. However, the key to Bro's success was its ability to bridge the traditional gap between academia and operations from early on, which provided the research with crucial grounding to ensure that developed approaches stand up to the challenges of the real world. Yet, with Bro's operational user community growing over time, the research-centric development model eventually became a bottleneck to the system's evolution: research grants do not tend to support the more mundane parts of software development and maintenance, even though those prove crucial for the end-user experience. While Bro's capabilities always went beyond those of traditional systems, a successful deployment used to require significant technical expertise, typically with a large upfront investment in tackling Bro's steep learning curve. In 2010, NSF set out to address this gap by awarding ICSI a grant dedicated solely to Bro development out of its SDCI program. With that support in place, the National Center for Supercomputing Applications (NCSA) joined the team as a core partner, and the Bro Project began to completely overhaul many of the user-visible parts of the system for the 2.0 release. Since that version came out, Bro has experienced an tremendous growth in new deployments across a diverse range of settings, and the Bro team is now working to build on this success by further advancing the system's capabilities to address the challenges of future networks.

### 1.1.4 Architecture

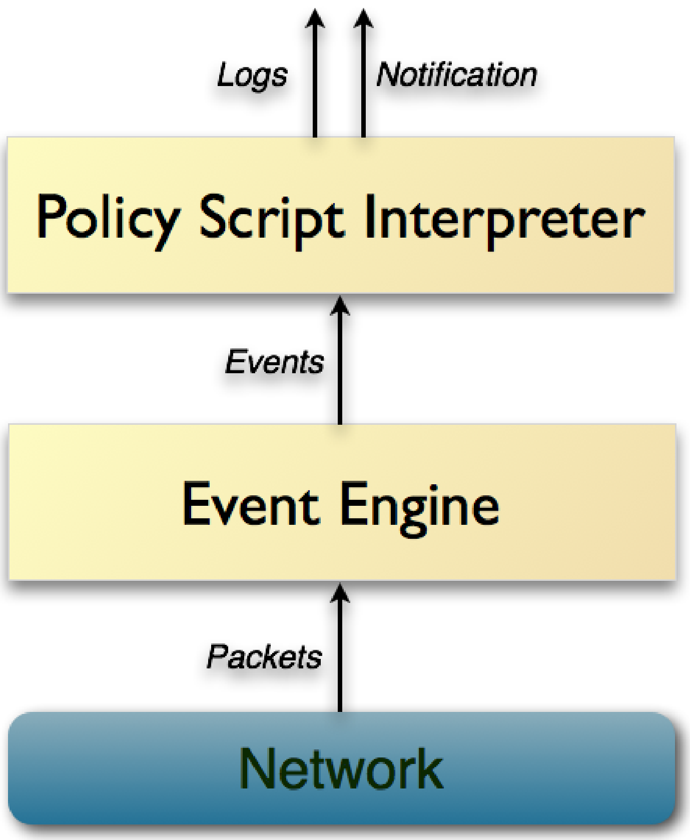


Fig. 1.2: Bro's internal architecture.

Architecturally, Bro is layered into two major components. Its *event engine* (or *core*) reduces the incoming packet stream into a series of higher-level *events*. These events reflect network activity in policy-neutral terms, i.e., they describe *what* has been seen, but not *why*, or whether it is significant. For example, every HTTP request on the wire turns into a corresponding `http_request` event that carries with it the involved IP addresses and ports, the URI being requested, and the HTTP version in use. The event however does not convey any further *interpretation*, e.g., of whether that URI corresponds to a known malware site.

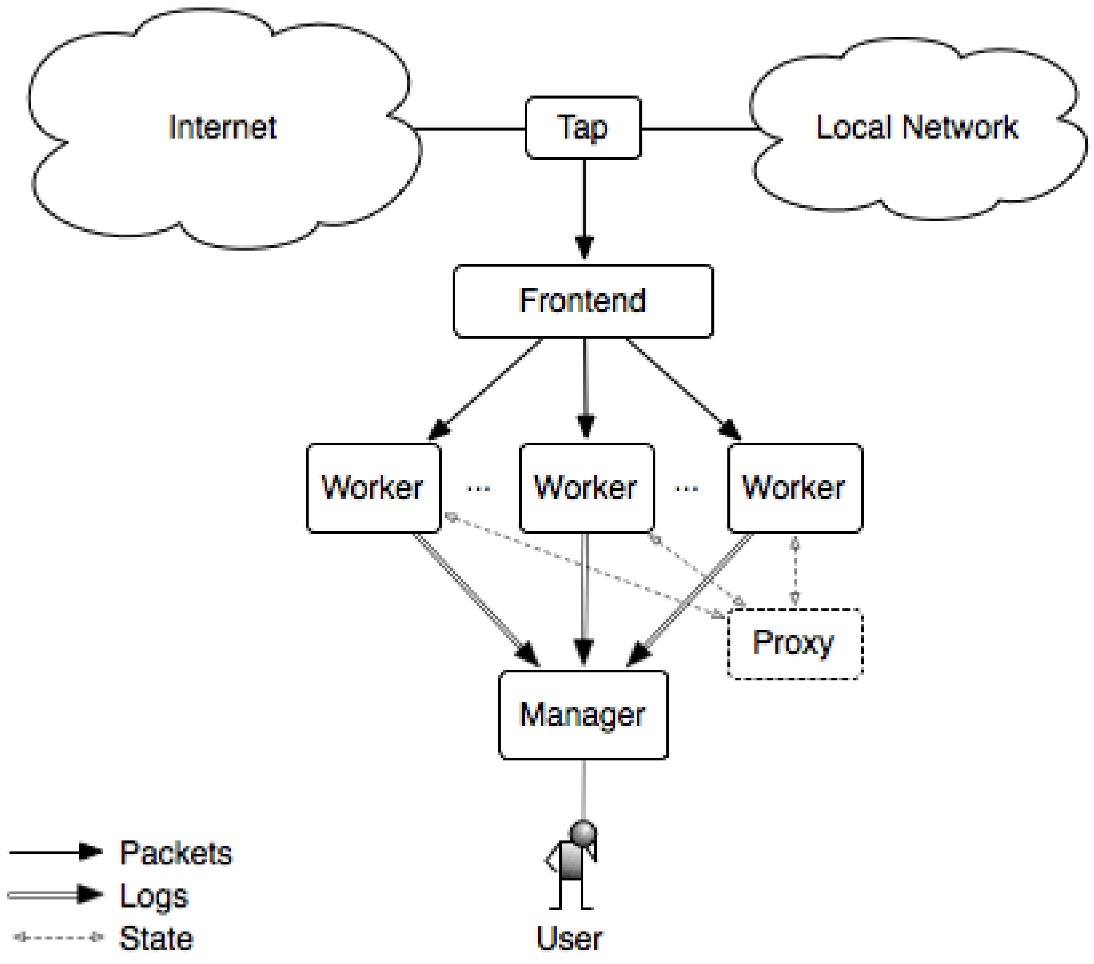
Such semantics are instead derived by Bro's second main component, the *script interpreter*, which executes a set of *event handlers* written in Bro's custom scripting language. These scripts can express a site's security policy, i.e., what actions to take when the monitor detects different types of activity. More generally they can derive any desired properties and statistics from the input traffic. Bro's language comes with extensive domain-specific types and support functionality; and, crucially, allows scripts to maintain state over time, enabling them to track and correlate the evolution of what they observe across connection and host boundaries. Bro scripts can generate real-time alerts and also execute arbitrary external programs on demand, e.g., to trigger an active response to an attack.

## 1.2 Bro Cluster Architecture

Bro is not multithreaded, so once the limitations of a single processor core are reached the only option currently is to spread the workload across many cores, or even many physical computers. The cluster deployment scenario for Bro is the current solution to build these larger systems. The tools and scripts that accompany Bro provide the structure to easily manage many Bro processes examining packets and doing correlation activities but acting as a singular, cohesive entity. This document describes the Bro cluster architecture. For information on how to configure a Bro cluster, see the documentation for *BroControl*.

### 1.2.1 Architecture

The figure below illustrates the main components of a Bro cluster.



### Tap

The tap is a mechanism that splits the packet stream in order to make a copy available for inspection. Examples include the monitoring port on a switch and an optical splitter on fiber networks.

### Frontend

The frontend is a discrete hardware device or on-host technique that splits traffic into many streams or flows. The Bro binary does not do this job. There are numerous ways to accomplish this task, some of which are described below in *Frontend Options*.

### Manager

The manager is a Bro process that has two primary jobs. It receives log messages and notices from the rest of the nodes in the cluster using the Bro communications protocol (note that if you are using a logger, then the logger receives all logs instead of the manager). The result is a single log instead of many discrete logs that you have to combine in some manner with post-processing. The manager also takes the opportunity to de-duplicate notices, and it has the ability to do so since it's acting as the choke point for notices and how notices might be processed into actions (e.g., emailing, paging, or blocking).

The manager process is started first by BroControl and it only opens its designated port and waits for connections, it doesn't initiate any connections to the rest of the cluster. Once the workers are started and connect to the manager, logs and notices will start arriving to the manager process from the workers.

### Logger

The logger is an optional Bro process that receives log messages from the rest of the nodes in the cluster using the Bro communications protocol. The purpose of having a logger receive logs instead of the manager is to reduce the load on the manager. If no logger is needed, then the manager will receive logs instead.

The logger process is started first by BroControl and it only opens its designated port and waits for connections, it doesn't initiate any connections to the rest of the cluster. Once the rest of the cluster is started and connect to the logger, logs will start arriving to the logger process.

### Proxy

The proxy is a Bro process that manages synchronized state. Variables can be synchronized across connected Bro processes automatically. Proxies help the workers by alleviating the need for all of the workers to connect directly to each other.

Examples of synchronized state from the scripts that ship with Bro include the full list of "known" hosts and services (which are hosts or services identified as performing full TCP handshakes) or an analyzed protocol has been found on the connection. If worker A detects host 1.2.3.4 as an active host, it would be beneficial for worker B to know that as well. So worker A shares that information as an insertion to a set which travels to the cluster's proxy and the proxy sends that same set insertion to worker B. The result is that worker A and worker B have shared knowledge about host and services that are active on the network being monitored.

The proxy model extends to having multiple proxies when necessary for performance reasons. It only adds one additional step for the Bro processes. Each proxy connects to another proxy in a ring and the workers are shared between them as evenly as possible. When a proxy receives some new bit of state it will share that with its proxy, which is then shared around the ring of proxies, and down to all of the workers. From a practical standpoint, there are no rules of thumb established for the number of proxies necessary for the number of workers they are serving. It is best to start with a single proxy and add more if communication performance problems are found.

Bro processes acting as proxies don't tend to be extremely hard on CPU or memory and users frequently run proxy processes on the same physical host as the manager.

### Worker

The worker is the Bro process that sniffs network traffic and does protocol analysis on the reassembled traffic streams. Most of the work of an active cluster takes place on the workers and as such, the workers typically represent the bulk of the Bro processes that are running in a cluster. The fastest memory and CPU core speed you can afford is recommended since all of the protocol parsing and most analysis will take place here. There are no particular requirements for the disks in workers since almost all logging is done remotely to the manager, and normally very little is written to disk.

The rule of thumb we have followed recently is to allocate approximately 1 core for every 250Mbps of traffic that is being analyzed. However, this estimate could be extremely traffic mix-specific. It has generally worked for mixed traffic with many users and servers. For example, if your traffic peaks around 2Gbps (combined) and you want to handle traffic at peak load, you may want to have 8 cores available (2048 / 250 == 8.2). If the 250Mbps estimate works for your traffic, this could be handled by 2 physical hosts dedicated to being workers with each one containing a quad-core processor.

Once a flow-based load balancer is put into place this model is extremely easy to scale. It is recommended that you estimate the amount of hardware you will need to fully analyze your traffic. If more is needed it's relatively easy to increase the size of the cluster in most cases.

### 1.2.2 Frontend Options

There are many options for setting up a frontend flow distributor. In many cases it is beneficial to do multiple stages of flow distribution on the network and on the host.

#### Discrete hardware flow balancers

##### cPacket

If you are monitoring one or more 10G physical interfaces, the recommended solution is to use either a cFlow or cVu device from cPacket because they are used successfully at a number of sites. These devices will perform layer-2 load balancing by rewriting the destination Ethernet MAC address to cause each packet associated with a particular flow to have the same destination MAC. The packets can then be passed directly to a monitoring host where each worker has a BPF filter to limit its visibility to only that stream of flows, or onward to a commodity switch to split the traffic out to multiple 1G interfaces for the workers. This greatly reduces costs since workers can use relatively inexpensive 1G interfaces.

##### OpenFlow Switches

We are currently exploring the use of OpenFlow based switches to do flow-based load balancing directly on the switch, which greatly reduces frontend costs for many users. This document will be updated when we have more information.

#### On host flow balancing

##### PF_RING

The PF_RING software for Linux has a “clustering” feature which will do flow-based load balancing across a number of processes that are sniffing the same interface. This allows you to easily take advantage of multiple cores in a single physical host because Bro’s main event loop is single threaded and can’t natively utilize all of the cores. If you want to use PF_RING, see the documentation on how to configure Bro with PF_RING.

##### Netmap

FreeBSD has an in-progress project named Netmap which will enable flow-based load balancing as well. When it becomes viable for real world use, this document will be updated.

##### Click! Software Router

Click! can be used for flow based load balancing with a simple configuration. This solution is not recommended on Linux due to Bro’s PF_RING support and only as a last resort on other operating systems since it causes a lot of overhead due to context switching back and forth between kernel and userland several times per packet.

## 1.3 Installation

### 1.3.1 Installing Bro

**Contents**

### Prerequisites

Before installing Bro, you'll need to ensure that some dependencies are in place.

### Required Dependencies

Bro requires the following libraries and tools to be installed before you begin:

- Libpcap (http://www.tcpdump.org)
- OpenSSL libraries (http://www.openssl.org)
- BIND8 library
- Libz
- Bash (for BroControl)
- Python (for BroControl)

To build Bro from source, the following additional dependencies are required:

- CMake 2.8 or greater (http://www.cmake.org)
- Make
- C/C++ compiler with C++11 support (GCC 4.8+ or Clang 3.3+)
- SWIG (http://www.swig.org)
- Bison (GNU Parser Generator)
- Flex (Fast Lexical Analyzer)
- Libpcap headers (http://www.tcpdump.org)
- OpenSSL headers (http://www.openssl.org)
- zlib headers
- Python

To install the required dependencies, you can use:

- RPM/RedHat-based Linux:
- DEB/Debian-based Linux:

- FreeBSD:

  Most required dependencies should come with a minimal FreeBSD install except for the following.

  Note that in older versions of FreeBSD, you might have to use the "pkg_add -r" command instead of "pkg install".

  For older versions of FreeBSD (especially FreeBSD 9.x), the system compiler is not new enough to compile Bro. For these systems, you will have to install a newer compiler using pkg; the `clang34` package should work.

  You will also have to define several environment variables on these older systems to use the new compiler and headers similar to this before calling configure:

- Mac OS X:

  Compiling source code on Macs requires first installing Xcode (in older versions of Xcode, you would then need to go through its "Preferences..." -> "Downloads" menus to install the "Command Line Tools" component).

  OS X comes with all required dependencies except for CMake, SWIG, and OpenSSL. (OpenSSL used to be part of OS X versions 10.10 and older, for which it does not need to be installed manually. It was removed in OS X 10.11). Distributions of these dependencies can likely be obtained from your preferred Mac OS X package management system (e.g. Homebrew, MacPorts, or Fink). Specifically for Homebrew, the `cmake`, `swig`, and `openssl` packages provide the required dependencies.

## Optional Dependencies

Bro can make use of some optional libraries and tools if they are found at build time:

- C++ Actor Framework (CAF) version 0.14 (http://actor-framework.org)
- LibGeoIP (for geolocating IP addresses)
- sendmail (enables Bro and BroControl to send mail)
- curl (used by a Bro script that implements active HTTP)
- gperftools (tcmalloc is used to improve memory and CPU usage)
- jemalloc (http://www.canonware.com/jemalloc/)
- PF_RING (Linux only, see *Cluster Configuration*)
- ipsumdump (for trace-summary; http://www.cs.ucla.edu/~kohler/ipsumdump)

LibGeoIP is probably the most interesting and can be installed on most platforms by following the instructions for *installing libGeoIP and the GeoIP database*.

## Installing Bro

Bro can be downloaded in either pre-built binary package or source code forms.

## Using Pre-Built Binary Release Packages

See the bro downloads page for currently supported/targeted platforms for binary releases and for installation instructions.

- Linux Packages

  Linux based binary installations are usually performed by adding information about the Bro packages to the respective system packaging tool. Then the usual system utilities such as `apt`, `dnf`, `yum`, or `zypper` are used to perform the installation.

The primary install prefix for binary packages is `/opt/bro`.

### Installing from Source

Bro releases are bundled into source packages for convenience and are available on the bro downloads page.

Alternatively, the latest Bro development version can be obtained through git repositories hosted at `git.bro.org`. See our git development documentation for comprehensive information on Bro's use of git revision control, but the short story for downloading the full source code experience for Bro via git is:

**Note:** If you choose to clone the `bro` repository non-recursively for a "minimal Bro experience", be aware that compiling it depends on several of the other submodules as well.

The typical way to build and install from source is (for more options, run `./configure --help`):

If the `configure` script fails, then it is most likely because it either couldn't find a required dependency or it couldn't find a sufficiently new version of a dependency. Assuming that you already installed all required dependencies, then you may need to use one of the `--with-*` options that can be given to the `configure` script to help it locate a dependency.

The default installation path is `/usr/local/bro`, which would typically require root privileges when doing the `make install`. A different installation path can be chosen by specifying the `configure` script `--prefix` option. Note that `/usr` and `/opt/bro` are the standard prefixes for binary Bro packages to be installed, so those are typically not good choices unless you are creating such a package.

OpenBSD users, please see our FAQ if you are having problems installing Bro.

Depending on the Bro package you downloaded, there may be auxiliary tools and libraries available in the `aux/` directory. Some of them will be automatically built and installed along with Bro. There are `--disable-*` options that can be given to the configure script to turn off unwanted auxiliary projects that would otherwise be installed automatically. Finally, use `make install-aux` to install some of the other programs that are in the `aux/bro-aux` directory.

Finally, if you want to build the Bro documentation (not required, because all of the documentation for the latest Bro release is available on the Bro web site), there are instructions in `doc/README` in the source distribution.

### Configure the Run-Time Environment

You may want to adjust your `PATH` environment variable according to the platform/shell/package you're using. For example:

Bourne-Shell Syntax:

C-Shell Syntax:

Or substitute `/opt/bro/bin` instead if you installed from a binary package.

### 1.3.2 Upgrading Bro

#### How to Upgrade

If you're doing an upgrade install (rather than a fresh install), there's two suggested approaches: either install Bro using the same installation prefix directory as before, or pick a new prefix and copy local customizations over.

Regardless of which approach you choose, if you are using BroControl, then before doing the upgrade you should stop all running Bro processes with the "broctl stop" command. After the upgrade is complete then you will need to run "broctl deploy".

In the following we summarize general guidelines for upgrading, see the *Release Notes* for version-specific information.

#### Reusing Previous Install Prefix

If you choose to configure and install Bro with the same prefix directory as before, local customization and configuration to files in `$prefix/share/bro/site` and `$prefix/etc` won't be overwritten (`$prefix` indicating the root of where Bro was installed). Also, logs generated at run-time won't be touched by the upgrade. Backing up local changes before upgrading is still recommended.

After upgrading, remember to check `$prefix/share/bro/site` and `$prefix/etc` for `.example` files, which indicate that the distribution's version of the file differs from the local one, and therefore, may include local changes. Review the differences and make adjustments as necessary. Use the new version for differences that aren't a result of a local change.

#### Using a New Install Prefix

To install the newer version in a different prefix directory than before, copy local customization and configuration files from `$prefix/share/bro/site` and `$prefix/etc` to the new location (`$prefix` indicating the root of where Bro was originally installed). Review the files for differences before copying and make adjustments as necessary (use the new version for differences that aren't a result of a local change). Of particular note, the copied version of `$prefix/etc/broctl.cfg` is likely to need changes to any settings that specify a pathname.

#### Release Notes

**Contents**

This document summarizes the most important changes in the current Bro release. For an exhaustive list of changes, see the `CHANGES` file (note that submodules, such as BroControl and Broccoli, come with their own `CHANGES`.)

## Bro 2.5

### New Dependencies

- Bro now requires a compiler with C++11 support for building the source code.
- Bro now requires Python instead of Perl to compile the source code.
- When enabling Broker (which is disabled by default), Bro now requires version 0.14 of the C++ Actor Framework.

### New Functionality

- SMB analyzer. This is the rewrite that has been in development for several years. The scripts are currently not loaded by default and must be loaded manually by loading policy/protocols/smb. The next release will load the smb scripts by default.

  - Implements SMB1+2.
  - Fully integrated with the file analysis framework so that files transferred over SMB can be analyzed.
  - Includes GSSAPI and NTLM analyzer and reimplements the DCE-RPC analyzer.
  - New logs: smb_files.log, smb_mapping.log, ntlm.log, and dce_rpc.log
  - Not every possible SMB command or functionality is implemented, but generally, file handling should work whenever files are transferred. Please speak up on the mailing list if there is an obvious oversight.
- Bro now includes the NetControl framework. The framework allows for easy interaction of Bro with hard- and software switches, firewalls, etc.
- Bro's Intelligence Framework was refactored and new functionality has been added:
  - The framework now supports the new indicator type Intel::SUBNET. As subnets are matched against seen addresses, the field 'matched' was introduced to indicate which indicator type(s) caused the hit.
  - The new function remove() allows to delete intelligence items.
  - The intel framework now supports expiration of intelligence items. Expiration can be configured by using Intel::item_expiration and can be handled by using the item_expired() hook. The new script do_expire.bro removes expired items.
  - The new hook extend_match() allows extending the framework. The new policy script whitelist.bro uses the hook to implement whitelisting.
  - Intel notices are now suppressible and mails for intel notices now list the identified services as well as the intel source.
- There is a new file entropy analyzer for files.
- Bro now supports the remote framebuffer protocol (RFB) that is used by VNC servers for remote graphical displays.
- Bro now supports the Radiotap header for 802.11 frames.
- Bro now has rudimentary IMAP and XMPP analyzers examinig the initial phases of the protocol. Right now these analyzer only identify STARTTLS sessions, handing them over to TLS analysis. The analyzer does not yet analyze any further IMAP/XMPP content.
- The new event ssl_extension_signature_algorithm allows access to the TLS signature_algorithms extension that lists client supported signature and hash algorithm pairs.
- Bro now tracks VLAN IDs. To record them inside the connection log, load protocols/conn/vlan-logging.bro.
- The new misc/stats.bro records Bro executions statistics in a standard Bro log file.
- A new dns_CAA_reply event gives access to DNS Certification Authority Authorization replies.
- A new per-packet event raw_packet() provides access to layer 2 information. Use with care, generating events per packet is expensive.
- A new built-in function, decode_base64_conn() for Base64 decoding. It works like decode_base64() but receives an additional connection argument that will be used for decoding errors into weird.log (instead of reporter.log).
- A new get_current_packet_header bif returns the headers of the current packet.
- Two new built-in functions for handling set[subnet] and table[subnet]:
  - check_subnet(subnet, table) checks if a specific subnet is a member of a set/table. This is different from the "in" operator, which always performs a longest prefix match.
  - matching_subnets(subnet, table) returns all subnets of the set or table that contain the given subnet.

  - filter_subnet_table(subnet, table) works like check_subnet, but returns a table containing all matching entries.
- Several built-in functions for handling IP addresses and subnets were added:
  - is_v4_subnet(subnet) checks whether a subnet specification is IPv4.
  - is_v6_subnet(subnet) checks whether a subnet specification is IPv6.
  - addr_to_subnet(addr) converts an IP address to a /32 subnet.
  - subnet_to_addr(subnet) returns the IP address part of a subnet.
  - subnet_width(subnet) returns the width of a subnet.
- The IRC analyzer now recognizes StartTLS sessions and enable the SSL analyzer for them.
- A set of new built-in function for gathering execution statistics:

  get_net_stats(), get_conn_stats(), get_proc_stats(), get_event_stats(), get_reassembler_stats(), get_dns_stats(), get_timer_stats(), get_file_analysis_stats(), get_thread_stats(), get_gap_stats(), get_matcher_stats(),
- Two new functions haversine_distance() and haversine_distance_ip() for calculating geographic distances. They requires that Bro be built with libgeoip.
- Table expiration timeout expressions are evaluated dynamically as timestmaps are updated.
- The pcap buffer size can be set through the new option Pcap::bufsize.
- Input framework readers Table and Event can now define a custom event to receive logging messages.
- The logging framework now supports user-defined record separators, renaming of column names, as well as extension data columns that can be added to specific or all logfiles (e.g., to add noew names).
- New BroControl functionality in aux/broctl:
  - There is a new node type “logger” that can be specified in node.cfg (that file has a commented-out example). The purpose of this new node type is to receive logs from all nodes in a cluster in order to reduce the load on the manager node. However, if there is no “logger” node, then the manager node will handle logging as usual.
  - The post-terminate script will send email if it fails to archive any log files. These mails can be turned off by changing the value of the new BroControl option MailArchiveLogFail.
  - Added the ability for “broctl deploy” to reload the BroControl configuration (both broctl.cfg and node.cfg). This happens automatically if broctl detects any changes to those config files since the last time the config was loaded. Note that this feature is relevant only when using the BroControl shell interactively.
  - The BroControl plugin API has a new function “broctl_config”. This gives plugin authors the ability to add their own script code to the autogenerated broctl-config.bro script.
  - There is a new BroControl plugin for custom load balancing. This plugin can be used by setting “lb_method=custom” for your worker nodes in node.cfg. To support packet source plugins, it allows configuration of a prefix and suffix for the interface name.
- New Bro plugins in aux/plugins:
  - af_packet: Native AF_PACKET support.
  - kafka : Log writer interfacing to Kafka.
  - myricom: Native Myricom SNF v3 support.
  - pf_ring: Native PF_RING support.
  - postgresql: A PostgreSQL reader/writer.

- redis: An experimental log writer for Redis.
- tcprs: An TCP-level analyzer detecting retransmissions, reordering, and more.

## Changed Functionality

- Log changes:
  - Connections * The 'history' field gains two new flags: '^' indicates that

    Bro heuristically flipped to direction of the connection. 't/T' indicates the first TCP payload retransmission from originator or responder, respectively.
  - DNS * New 'rtt' field to indicate the round trip time between when a

    request was sent and when a reply started.
  - SMTP * New 'cc' field which includes the 'Cc' header from MIME

    messages sent over SMTP.

    * Changes in 'mailfrom' and 'rcptto' fields to remove some non-address cruft that will tend to be found. The main example is the change from "<user@domain>" to "user@domain.com".
  - HTTP * Removed 'filename' field.
    * New 'orig_filenames' and 'resp_filenames' fields which each contain a vector of filenames seen in entities transferred.
- The BrokerComm and BrokerStore namespaces were renamed to Broker. The Broker "print" function was renamed to Broker::send_print, and "event" to "Broker::send_event".
- `SSH::skip_processing_after_detection` was removed. The functionality was replaced by `SSH::disable_analyzer_after_detection`.
- `net_stats()` and `resource_usage()` have been superseded by the new execution statistics functions (see above).
- Some script-level identifier have changed their names:

  snaplen -> Pcap::snaplen precompile_pcap_filter() -> Pcap::precompile_pcap_filter() install_pcap_filter() -> Pcap::install_pcap_filter() pcap_error() -> Pcap::pcap_error()
- In http.log, the "filename" field (which it turns out was never filled out in the first place) has been split into to "orig_filenames" and "resp_filenames".
- TCP analysis was changed to process connections without the initial SYN packet. In the past, connections without a full handshake were treated as partial, meaning that most application-layer analyzers would refuse to inspect the payload. Now, Bro will consider these connections as complete and all analyzers will process them notmally.
- Changed BroControl functionality in aux/broctl:
  - The networks.cfg file now contains private IP space 172.16.0.0/12 by default.
  - Upon startup, if broctl can't get IP addresses from the "ifconfig" command for any reason, then broctl will now also try to use the "ip" command.
  - BroControl will now automatically search the Bro plugin directory for BroControl plugins (in addition to all the other places where BroControl searches). This enables automatic loading of BroControl plugins that are provided by a Bro plugin.
  - Changed the default value of the StatusCmdShowAll option so that the "broctl status" command runs faster. This also means that there is no longer a "Peers" column in the status output by default.

  - Users can now specify a more granular log expiration interval. The BroControl option LogExpireInterval can be set to an arbitrary time interval instead of just an integer number of days. The time interval is specified as an integer followed by a time unit: “day”, “hr”, or “min”. For backward compatibility, an integer value without a time unit is still interpreted as a number of days.
  - Changed the text of crash report emails. Now crash reports tell the user to forward the mail to the Bro team only when a backtrace is included in the crash report. If there is no backtrace, then the crash report includes instructions on how to get backtraces included in future crash reports.

### Removed Functionality

- The app-stats scripts have been removed because they weren’t being maintained and they were becoming inaccurate. They were also prone to needing more regular updates as the internet changed and will likely be more relevant if maintained externally.
- The event ack_above_hole() has been removed, as it was a subset of content_gap() and led to plenty noise.
- The command line options –set-seed and –md5-hashkey have been removed.
- The packaging scripts pkg/make-*-packages are gone. They aren’t used anymore for the binary Bro packages that the projects distributes; haven’t been supported in a while; and have problems.

### Deprecated Functionality

- The built-in functions decode_base64_custom() and encode_base64_custom() are no longer needed and will be removed in the future. Their functionality is now provided directly by decode_base64() and encode_base64(), which take an optional parameter to change the Base64 alphabet.
- The ElasticSearch log writer hasn’t been maintained for a while and is now deprecated. It will be removed with the next release.

## Bro 2.4

### New Functionality

- Bro now has support for external plugins that can extend its core functionality, like protocol/file analysis, via shared libraries. Plugins can be developed and distributed externally, and will be pulled in dynamically at startup (the environment variables BRO_PLUGIN_PATH and BRO_PLUGIN_ACTIVATE can be used to specify the locations and names of plugins to activate). Currently, a plugin can provide custom protocol analyzers, file analyzers, log writers, input readers, packet sources and dumpers, and new built-in functions. A plugin can furthermore hook into Bro’s processing at a number of places to add custom logic.

  See https://www.bro.org/sphinx-git/devel/plugins.html for more information on writing plugins.
- Bro now has support for the MySQL wire protocol. Activity gets logged into mysql.log.
- Bro now parses DTLS traffic. Activity gets logged into ssl.log.
- Bro now has support for the Kerberos KRB5 protocol over TCP and UDP. Activity gets logged into kerberos.log.
- Bro now has an RDP analyzer. Activity gets logged into rdp.log.
- Bro now has a file analyzer for Portable Executables. Activity gets logged into pe.log.
- Bro now has support for the SIP protocol over UDP. Activity gets logged into sip.log.

- Bro now features a completely rewritten, enhanced SSH analyzer. The new analyzer is able to determine if logins failed or succeeded in most circumstances, logs a lot more more information about SSH sessions, supports v1, and introduces the intelligence type `Intel::PUBKEY_HASH` and location `SSH::IN_SERVER_HOST_KEY`. The analayzer also generates a set of additional events (`ssh_auth_successful`, `ssh_auth_failed`, `ssh_capabilities`, `ssh2_server_host_key`, `ssh1_server_host_key`, `ssh_encrypted_packet`, `ssh2_dh_server_params`, `ssh2_gss_error`, `ssh2_ecc_key`). See next section for incompatible SSH changes.
- Bro's file analysis now supports reassembly of files that are not transferred/seen sequentially. The default file reassembly buffer size is set with the `Files::reassembly_buffer_size` variable.
- Bro's file type identification has been greatly improved (new file types, bug fixes, and performance improvements).
- Bro's scripting language now has a `while` statement:

```
while ( i < 5 )
    print ++i;
```

  `next` and `break` can be used inside the loop's body just like with `for` loops.
- Bro now integrates Broker, a new communication library. See aux/broker/README for more information on Broker, and doc/frameworks/broker.rst for the corresponding Bro script API.

  With Broker, Bro has the similar capabilities of exchanging events and logs with remote peers (either another Bro process or some other application that uses Broker). It also includes a key-value store API that can be used to share state between peers and optionally allow data to persist on disk for longer-term storage.

  Broker support is by default off for now; it can be enabled at configure time with –enable-broker. It requires CAF version 0.13+ (https://github.com/actor-framework/actor-framework) as well as a C++11 compiler (e.g. GCC 4.8+ or Clang 3.3+).

  Broker will become a mandatory dependency in future Bro versions and replace the current communication and serialization system.
- Add –enable-c++11 configure flag to compile Bro's source code in C++11 mode with a corresponding compiler. Note that 2.4 will be the last version of Bro that compiles without C++11 support.
- The SSL analysis now alerts when encountering SSL connections with old protocol versions or unsafe cipher suites. It also gained extended reporting of weak keys, caching of already validated certificates, and full support for TLS record defragmentation. SSL generally became much more robust and added several fields to ssl.log (while removing some others).
- A new icmp_sent_payload event provides access to ICMP payload.
- The input framework's raw reader now supports seeking by adding an option "offset" to the config map. Positive offsets are interpreted to be from the beginning of the file, negative from the end of the file (-1 is end of file).
- One can now raise events when a connection crosses a given size threshold in terms of packets or bytes. The primary API for that functionality is in base/protocols/conn/thresholds.bro.
- There is a new command-line option -Q/–time that prints Bro's execution time and memory usage to stderr.
- BroControl now has a new command "deploy" which is equivalent to running the "check", "install", "stop", and "start" commands (in that order).
- BroControl now has a new option "StatusCmdShowAll" that controls whether or not the broctl "status" command gathers all of the status information. This option can be used to make the "status" command run significantly faster (in this case, the "Peers" column will not be shown in the output).

- BroControl now has a new option "StatsLogEnable" that controls whether or not broctl will record information to the "stats.log" file. This option can be used to make the "broctl cron" command run slightly faster (in this case, "broctl cron" will also no longer send email about not seeing any packets on the monitoring interfaces).
- BroControl now has a new option "MailHostUpDown" which controls whether or not the "broctl cron" command will send email when it notices that a host in the cluster is up or down.
- BroControl now has a new option "CommandTimeout" which specifies the number of seconds to wait for a command that broctl ran to return results.

### Changed Functionality

- bro-cut has been rewritten in C, and is hence much faster.
- File analysis
  - Removed `fa_file` record's `mime_type` and `mime_types` fields. The event `file_sniff` has been added which provides the same information. The `mime_type` field of `Files::Info` also still has this info.
  - The earliest point that new mime type information is available is in the `file_sniff` event which comes after the `file_new` and `file_over_new_connection` events. Scripts which inspected mime type info within those events will need to be adapted. (Note: for users that worked w/ versions of Bro from git, for a while there was also an event called `file_mime_type` which is now replaced with the `file_sniff` event).
  - Removed `Files::add_analyzers_for_mime_type` function.
  - Removed `offset` parameter of the `file_extraction_limit` event. Since file extraction now internally depends on file reassembly for non-sequential files, "offset" can be obtained with other information already available – adding together `seen_bytes` and `missed_bytes` fields of the `fa_file` record gives how many bytes have been written so far (i.e. the "offset").
- The SSH changes come with a few incompatibilities. The following events have been renamed:
  - `SSH::heuristic_failed_login` to `ssh_auth_failed`
  - `SSH::heuristic_successful_login` to `ssh_auth_successful`

  The `SSH::Info` status field has been removed and replaced with the `auth_success` field. This field has been changed from a string that was previously `success`, `failure` or `undetermined` to a boolean. a boolean that is `T`, `F`, or unset.
- The has_valid_octets function now uses a string_vec parameter instead of string_array.
- conn.log gained a new field local_resp that works like local_orig, just for the responder address of the connection.
- GRE tunnels are now identified as `Tunnel::GRE` instead of `Tunnel::IP`.
- The default name for extracted files changed from extract-protocol-id to extract-timestamp-protocol-id.
- The weird named "unmatched_HTTP_reply" has been removed since it can be detected at the script-layer and is handled correctly by the default HTTP scripts.
- When adding a logging filter to a stream, the filter can now inherit a default `path` field from the associated `Log::Stream` record.
- When adding a logging filter to a stream, the `Log::default_path_func` is now only automatically added to the filter if it has neither a `path` nor a `path_func` already explicitly set. Before, the default path function would always be set for all filters which didn't specify their own `path_func`.

- BroControl now establishes only one ssh connection from the manager to each remote host in a cluster configuration (previously, there would be one ssh connection per remote Bro process).
- BroControl now uses SQLite to record state information instead of a plain text file (the file "spool/broctl.dat" is no longer used). On FreeBSD, this means that there is a new dependency on the package "py27-sqlite3".
- BroControl now records the expected running state of each Bro node right before each start or stop. The "broctl cron" command uses this info to either start or stop Bro nodes as needed so that the actual state matches the expected state (previously, "broctl cron" could only start nodes in the "crashed" state, and could never stop a node).
- BroControl now sends all normal command output (i.e., not error messages) to stdout. Error messages are still sent to stderr, however.
- The capability of processing NetFlow input has been removed for the time being. Therefore, the -y/–flowfile and -Y/–netflow command-line options have been removed, and the netflow_v5_header and netflow_v5_record events have been removed.
- The -D/–dfa-size command-line option has been removed.
- The -L/–rule-benchmark command-line option has been removed.
- The -O/–optimize command-line option has been removed.
- The deprecated fields "hot" and "addl" have been removed from the connection record. Likewise, the functions append_addl() and append_addl_marker() have been removed.
- Log files now escape non-printable characters consistently as "xXX'. Furthermore, backslashes are escaped as "\", making the representation fully reversible.

### Deprecated Functionality

- The split* family of functions are to be replaced with alternate versions that return a vector of strings rather than a table of strings. This also allows deprecation for some related string concatenation/extraction functions. Note that the new functions use 0-based indexing, rather than 1-based.

  The full list of now deprecated functions is:

  - split: use split_string instead.
  - split1: use split_string1 instead.
  - split_all: use split_string_all instead.
  - split_n: use split_string_n instead.
  - cat_string_array: see join_string_vec instead.
  - cat_string_array_n: see join_string_vec instead.
  - join_string_array: see join_string_vec instead.
  - sort_string_array: use sort instead.
  - find_ip_addresses: use extract_ip_addresses instead.

## Bro 2.3

### Dependencies

- Libmagic is no longer a dependency.

### New Functionality

- Support for GRE tunnel decapsulation, including enhanced GRE headers. GRE tunnels are treated just like IP-in-IP tunnels by parsing past the GRE header in between the delivery and payload IP packets.
- The DNS analyzer now actually generates the dns_SRV_reply() event. It had been documented before, yet was never raised.
- Bro now uses "file magic signatures" to identify file types. These are defined via two new constructs in the signature rule parsing grammar: "file-magic" gives a regular expression to match against, and "file-mime" gives the MIME type string of content that matches the magic and an optional strength value for the match. (See also "Changed Functionality" below for changes due to switching from using libmagic to such signatures.)
- A new built-in function, "file_magic", can be used to get all file magic matches and their corresponding strength against a given chunk of data.
- The SSL analyzer now supports heartbeats as well as a few extensions, including server_name, alpn, and ec-curves.
- The SSL analyzer comes with Heartbleed detector script in protocols/ssl/heartbleed.bro. Note that loading this script changes the default value of "SSL::disable_analyzer_after_detection" from true to false to prevent encrypted heartbeats from being ignored.
- StartTLS is now supported for SMTP and POP3.
- The X509 analyzer can now perform OSCP validation.
- Bro now has analyzers for SNMP and Radius, which produce corresponding snmp.log and radius.log output (as well as various events of course).
- BroControl has a new option "BroPort" which allows a user to specify the starting port number for Bro.
- BroControl has a new option "StatsLogExpireInterval" which allows a user to specify when entries in the stats.log file expire.
- BroControl has a new option "PFRINGClusterType" which allows a user to specify a PF_RING cluster type.
- BroControl now supports PF_RING+DNA. There is also a new option "PFRINGFirstAppInstance" that allows a user to specify the starting application instance number for processes running on a DNA cluster. See the BroControl documentation for more details.
- BroControl now warns a user to run "broctl install" if Bro has been upgraded or if the broctl or node configuration has changed since the most recent install.

### Changed Functionality

- string slices now exclude the end index (e.g., "123"[1:2] returns "2"). Generally, Bro's string slices now behave similar to Python.
- ssl_client_hello() now receives a vector of ciphers, instead of a set, to preserve their order.
- Notice::end_suppression() has been removed.
- Bro now parses X.509 extensions headers and, as a result, the corresponding event got a new signature:

  event x509_extension(c: connection, is_orig: bool, cert: X509, ext: X509_extension_info);
- In addition, there are several new, more specialized events for a number of x509 extensions.
- Generally, all x509 events and handling functions have changed their signatures.
- X509 certificate verification now returns the complete certificate chain that was used for verification.

- Bro no longer special-cases SYN/FIN/RST-filtered traces by not reporting missing data. Instead, if Bro never sees any data segments for analyzed TCP connections, the new base/misc/find-filtered-trace.bro script will log a warning in reporter.log and to stderr. The old behavior can be reverted by redef'ing "detect_filtered_trace".
- We have removed the packet sorter component.
- Bro no longer uses libmagic to identify file types but instead now comes with its own signature library (which initially is still derived from libmagic's database). This leads to a number of further changes with regards to MIME types:
  - The second parameter of the "identify_data" built-in function can no longer be used to get verbose file type descriptions, though it can still be used to get the strongest matching file magic signature.
  - The "file_transferred" event's "descr" parameter no longer contains verbose file type descriptions.
  - The BROMAGIC environment variable no longer changes any behavior in Bro as magic databases are no longer used/installed.
  - Removed "binary" and "octet-stream" mime type detections. They don't provide any more information than an uninitialized mime_type field.
  - The "fa_file" record now contains a "mime_types" field that contains all magic signatures that matched the file content (where the "mime_type" field is just a shortcut for the strongest match).
- dns_TXT_reply() now supports more than one string entry by receiving a vector of strings.
- BroControl now runs the "exec" and "df" broctl commands only once per host, instead of once per Bro node. The output of these commands has been changed slightly to include both the host and node names.
- Several performance improvements were made. Particular emphasis was put on the File Analysis system, which generally will now emit far fewer file handle request events due to protocol analyzers now caching that information internally.

## Bro 2.2

### New Functionality

- A completely overhauled intelligence framework for consuming external intelligence data. It provides an abstracted mechanism for feeding data into the framework to be matched against the data available. It also provides a function named `Intel::match` which makes any hits on intelligence data available to the scripting language.

  Using input framework, the intel framework can load data from text files. It can also update and add data if changes are made to the file being monitored. Files to monitor for intelligence can be provided by redef-ing the `Intel::read_files` variable.

  The intel framework is cluster-ready. On a cluster, the manager is the only node that needs to load in data from disk, the cluster support will distribute the data across a cluster automatically.

  Scripts are provided at `policy/frameworks/intel/seen` that provide a broad set of sources of data to feed into the intel framwork to be matched.
- A new file analysis framework moves most of the processing of file content from script-land into the core, where it belongs. See `doc/file-analysis.rst`, or the online documentation, for more information.

  Much of this is an internal change, but the framework also comes with the following user-visible functionality (some of that was already available before but is done differently, and more efficiently, now):
  - HTTP:
    - Identify MIME type of messages.

    * Extract messages to disk.
    * Compute MD5 for messages.
  - SMTP:
    * Identify MIME type of messages.
    * Extract messages to disk.
    * Compute MD5 for messages.
    * Provide access to start of entity data.
  - FTP data transfers:
    * Identify MIME types of data.
    * Record to disk.
  - IRC DCC transfers: Record to disk.
  - Support for analyzing data transferred via HTTP range requests.
  - A binary input reader interfaces the input framework with the file analysis, allowing to inject files on disk into Bro's content processing.

- A new framework for computing a wide array of summary statistics, such as counters and thresholds checks, standard deviation and mean, set cardinality, top K, and more. The framework operates in real-time, independent of the underlying data, and can aggregate information from many independent monitoring points (including clusters). It provides a transparent, easy-to-use user interface, and can optionally deploy a set of probabilistic data structures for memory-efficient operation. The framework is located in `scripts/base/frameworks/sumstats`.

  A number of new applications now ship with Bro that are built on top of the summary statistics framework:

  - Scan detection: Detectors for port and address scans. See `policy/misc/scan.bro` (these scan detectors used to exist in Bro versions <2.0; it's now back, but quite different).
  - Tracerouter detector: `policy/misc/detect-traceroute.bro`
  - Web application detection/measurement: `policy/misc/app-stats/*`
  - FTP and SSH brute-forcing detector: `policy/protocols/ftp/detect-bruteforcing.bro`, `policy/protocols/ssh/detect-bruteforcing.bro`
  - HTTP-based SQL injection detector: `policy/protocols/http/detect-sqli.bro` (existed before, but now ported to the new framework)

- GridFTP support. This is an extension to the standard FTP analyzer and includes:
  - An analyzer for the GSI mechanism of GSSAPI FTP AUTH method. GSI authentication involves an encoded TLS/SSL handshake over the FTP control session. For FTP sessions that attempt GSI authentication, the `service` field of the connection log will include `gridftp` (as well as also `ftp` and `ssl`).
  - An example of a GridFTP data channel detection script. It relies on the heuristics of GridFTP data channels commonly default to SSL mutual authentication with a NULL bulk cipher and that they usually transfer large datasets (default threshold of script is 1 GB). For identified GridFTP data channels, the `services` fields of the connection log will include `gridftp-data`.

- Modbus and DNP3 support. Script-level support is only basic at this point but see `src/analyzer/protocol/{modbus,dnp3}/events.bif`, or the online documentation, for the events Bro generates. For Modbus, there are also some example policies in `policy/protocols/modbus/*`.

- The documentation now includes a new introduction to writing Bro scripts. See `doc/scripting/index.rst` or, much better, the online version. There's also the beginning of a chapter on "Using Bro" in `doc/using/index.rst`.
- GPRS Tunnelling Protocol (GTPv1) decapsulation.
- The scripting language now provide "hooks", a new flavor of functions that share characteristics of both standard functions and events. They are like events in that multiple bodies can be defined for the same hook identifier. They are more like functions in the way they are invoked/called, because, unlike events, their execution is immediate and they do not get scheduled through an event queue. Also, a unique feature of a hook is that a given hook handler body can short-circuit the execution of remaining hook handlers simply by exiting from the body as a result of a `break` statement (as opposed to a `return` or just reaching the end of the body). See `doc/scripts/builtins.rst`, or the online documentation, for more informatin.
- Bro's language now has a working `switch` statement that generally behaves like C-style switches (except that case labels can be comprised of multiple literal constants delimited by commas). Only atomic types are allowed for now. Case label bodies that don't execute a `return` or `break` statement will fall through to subsequent cases. A `default` case label is supported.
- Bro's language now has a new set of types `opaque of X`. Opaque values can be passed around like other values but they can only be manipulated with BiF functions, not with other operators. Currently, the following opaque types are supported:

  ```
  opaque of md5
  opaque of sha1
  opaque of sha256
  opaque of cardinality
  opaque of topk
  opaque of bloomfilter
  ```

  These go along with the corrsponding BiF functions `md5_*`, `sha1_*`, `sha256_*`, `entropy_*`, etc. . Note that where these functions existed before, they have changed their signatures to work with opaques types rather than global state.
- The scripting language now supports constructing sets, tables, vectors, and records by name:

  ```
  type MyRecordType: record {
      c: count;
      s: string &optional;
  };

  global r: MyRecordType = record($c = 7);

  type MySet: set[MyRec];
  global s = MySet([$c=1], [$c=2]);
  ```

- Strings now support the subscript operator to extract individual characters and substrings (e.g., `s[4]`, `s[1:5]`). The index expression can take up to two indices for the start and end index of the substring to return (e.g. `mystring[1:3]`).
- Functions now support default parameters, e.g.:

  ```
  global foo: function(s: string, t: string &default="abc", u: count &default=0);
  ```

- Scripts can now use two new "magic constants" `@DIR` and `@FILENAME` that expand to the directory path of the current script and just the script file name without path, respectively.
- `ssl.log` now also records the subject client and issuer certificates.
- The ASCII writer can now output CSV files on a per filter basis.

- New SQLite reader and writer plugins for the logging framework allow to read/write persistent data from on disk SQLite databases.
- A new packet filter framework supports BPF-based load-balancing, shunting, and sampling; plus plugin support to customize filters dynamically.
- Bro now provides Bloom filters of two kinds: basic Bloom filters supporting membership tests, and counting Bloom filters that track the frequency of elements. The corresponding functions are:

```
bloomfilter_basic_init(fp: double, capacity: count, name: string &default=""):␣
↪opaque of bloomfilter
bloomfilter_basic_init2(k: count, cells: count, name: string &default=""): opaque␣
↪of bloomfilter
bloomfilter_counting_init(k: count, cells: count, max: count, name: string &
↪default=""): opaque of bloomfilter
bloomfilter_add(bf: opaque of bloomfilter, x: any)
bloomfilter_lookup(bf: opaque of bloomfilter, x: any): count
bloomfilter_merge(bf1: opaque of bloomfilter, bf2: opaque of bloomfilter): opaque␣
↪of bloomfilter
bloomfilter_clear(bf: opaque of bloomfilter)
```

  See `src/probabilistic/bloom-filter.bif`, or the online documentation, for full documentation.

- Bro now provides a probabilistic data structure for computing "top k" elements. The corresponding functions are:

```
topk_init(size: count): opaque of topk
topk_add(handle: opaque of topk, value: any)
topk_get_top(handle: opaque of topk, k: count)
topk_count(handle: opaque of topk, value: any): count
topk_epsilon(handle: opaque of topk, value: any): count
topk_size(handle: opaque of topk): count
topk_sum(handle: opaque of topk): count
topk_merge(handle1: opaque of topk, handle2: opaque of topk)
topk_merge_prune(handle1: opaque of topk, handle2: opaque of topk)
```

  See `src/probabilistic/top-k.bif`, or the online documentation, for full documentation.

- Bro now provides a probabilistic data structure for computing set cardinality, using the HyperLogLog algorithm. The corresponding functions are:

```
hll_cardinality_init(err: double, confidence: double): opaque of cardinality
hll_cardinality_add(handle: opaque of cardinality, elem: any): bool
hll_cardinality_merge_into(handle1: opaque of cardinality, handle2: opaque of␣
↪cardinality): bool
hll_cardinality_estimate(handle: opaque of cardinality): double
hll_cardinality_copy(handle: opaque of cardinality): opaque of cardinality
```

  See `src/probabilistic/cardinality-counter.bif`, or the online documentation, for full documentation.

- `base/utils/exec.bro` provides a module to start external processes asynchronously and retrieve their output on termination. `base/utils/dir.bro` uses it to monitor a directory for changes, and `base/utils/active-http.bro` for providing an interface for querying remote web servers.
- BroControl can now pin Bro processes to CPUs on supported platforms: To use CPU pinning, a new per-node option `pin_cpus` can be specified in node.cfg if the OS is either Linux or FreeBSD.
- BroControl now returns useful exit codes. Most BroControl commands return 0 if everything was OK, and 1 otherwise. However, there are a few exceptions. The "status" and "top" commands return 0 if all Bro nodes are

running, and 1 if not all nodes are running. The "cron" command always returns 0 (but it still sends email if there were any problems). Any command provided by a plugin always returns 0.

- BroControl now has an option "env_vars" to set Bro environment variables. The value of this option is a comma-separated list of environment variable assignments (e.g., "VAR1=value, VAR2=another"). The "env_vars" option can apply to all Bro nodes (by setting it in broctl.cfg), or can be node-specific (by setting it in node.cfg). Environment variables in node.cfg have priority over any specified in broctl.cfg.
- BroControl now supports load balancing with PF_RING while sniffing multiple interfaces. Rather than assigning the same PF_RING cluster ID to all workers on a host, cluster ID assignment is now based on which interface a worker is sniffing (i.e., all workers on a host that sniff the same interface will share a cluster ID). This is handled by BroControl automatically.
- BroControl has several new options: MailConnectionSummary (for disabling the sending of connection summary report emails), MailAlarmsInterval (for specifying a different interval to send alarm summary emails), CompressCmd (if archived log files will be compressed, this specifies the command that will be used to compress them), CompressExtension (if archived log files will be compressed, this specifies the file extension to use).
- BroControl comes with its own test-suite now. `make test` in `aux/broctl` will run it.

In addition to these, Bro 2.2 comes with a large set of smaller extensions, tweaks, and fixes across the whole code base, including most submodules.

## Changed Functionality

- Previous versions of `$prefix/share/bro/site/local.bro` (where "$prefix" indicates the installation prefix of Bro), aren't compatible with Bro 2.2. This file won't be overwritten when installing over a previous Bro installation to prevent clobbering users' modifications, but an example of the new version is located in `$prefix/share/bro/site/local.bro.example`. So if no modification has been done to the previous local.bro, just copy the new example version over it, else merge in the differences. For reference, a common error message when attempting to use an outdated local.bro looks like:

```
fatal error in /usr/local/bro/share/bro/policy/frameworks/software/vulnerable.
↪bro, line 41: BroType::AsRecordType (table/record) (set[record { min:record {␣
↪major:count; minor:count; minor2:count; minor3:count; addl:string; }; max:
↪record { major:count; minor:count; minor2:count; minor3:count; addl:string; }; }
↪])
```

- The type of `Software::vulnerable_versions` changed to allow more flexibility and range specifications. An example usage:
- The interface to extracting content from application-layer protocols (including HTTP, SMTP, FTP) has changed significantly due to the introduction of the new file analysis framework (see above).
- Removed the following, already deprecated, functionality:
  - **Scripting language:**
    - `&disable_print_hook attribute.`
  - **BiF functions:**
    - `parse_dotted_addr()`, `dump_config()`, `make_connection_persistent()`, `generate_idmef()`, `split_complete()`
    - `md5_*`, `sha1_*`, `sha256_*`, and `entropy_*` have all changed their signatures to work with opaque types (see above).
- Removed a now unused argument from `do_split` helper function.

- `this` is no longer a reserved keyword.
- The Input Framework's `update_finished` event has been renamed to `end_of_data`. It will now not only fire after table-reads have been completed, but also after the last event of a whole-file-read (or whole-db-read, etc.).
- Renamed the option defining the frequency of alarm summary mails to `Logging::default_alarm_mail_interval`. When using BroControl, the value can now be set with the new broctl.cfg option `MailAlarmsInterval`.
- We have completely rewritten the `notice_policy` mechanism. It now no longer uses a record of policy items but a `hook`, a new language element that's roughly equivalent to a function with multiple bodies (see above). For existing code, the two main changes are:
  - What used to be a `redef` of `Notice::policy` now becomes a hook implementation. Example:

    Old:

    ```
    redef Notice::policy += {
        [$pred(n: Notice::Info) = {
            return n$note == SSH::Login && n$id$resp_h == 10.0.0.1;
            },
        $action = Notice::ACTION_EMAIL]
        };
    ```

    New:

    ```
    hook Notice::policy(n: Notice::Info)
        {
        if ( n$note == SSH::Login && n$id$resp_h == 10.0.0.1 )
            add n$actions[Notice::ACTION_EMAIL];
        }
    ```

  - notice() is now likewise a hook, no longer an event. If you have handlers for that event, you'll likely just need to change the type accordingly. Example:

    Old:

    ```
    event notice(n: Notice::Info) { ... }
    ```

    New:

    ```
    hook notice(n: Notice::Info) { ... }
    ```

- The `notice_policy.log` is gone. That's a result of the new notice policy setup.
- Removed the `byte_len()` and `length()` bif functions. Use the `|...|` operator instead.
- The `SSH::Login` notice has been superseded by an corresponding intelligence framework observation (`SSH::SUCCESSFUL_LOGIN`).
- `PacketFilter::all_packets` has been replaced with `PacketFilter::enable_auto_protocol_capture_fil`
- We removed the BitTorrent DPD signatures pending further updates to that analyzer.
- In previous versions of BroControl, running "broctl cron" would create a file `$prefix/logs/stats/www` (where "$prefix" indicates the installation prefix of Bro). Now, it is created as a directory. Therefore, if you perform an upgrade install and you're using BroControl, then you may see an email (generated by "broctl cron") containing an error message: "error running update-stats". To fix this problem, either remove that file (it is not needed) or rename it.

- Due to lack of maintenance the Ruby bindings for Broccoli are now deprecated, and the build process no longer includes them by default. For the time being, they can still be enabled by configuring with `--enable-ruby`, however we plan to remove Broccoli's Ruby support with the next Bro release.

## Bro 2.1

### New Functionality

- Bro now comes with extensive IPv6 support. Past versions offered only basic IPv6 functionality that was rarely used in practice as it had to be enabled explicitly. IPv6 support is now fully integrated into all parts of Bro including protocol analysis and the scripting language. It's on by default and no longer requires any special configuration.

  Some of the most significant enhancements include support for IPv6 fragment reassembly, support for following IPv6 extension header chains, and support for tunnel decapsulation (6to4 and Teredo). The DNS analyzer now handles AAAA records properly, and DNS lookups that Bro itself performs now include AAAA queries, so that, for example, the result returned by script-level lookups is a set that can contain both IPv4 and IPv6 addresses. Support for the most common ICMPv6 message types has been added. Also, the FTP EPSV and EPRT commands are now handled properly. Internally, the way IP addresses are stored has been improved, so Bro can handle both IPv4 and IPv6 by default without any special configuration.

  In addition to Bro itself, the other Bro components have also been made IPv6-aware by default. In particular, significant changes were made to trace-summary, PySubnetTree, and Broccoli to support IPv6.

- Bro now decapsulates tunnels via its new tunnel framework located in scripts/base/frameworks/tunnels. It currently supports Teredo, AYIYA, IP-in-IP (both IPv4 and IPv6), and SOCKS. For all these, it logs the outer tunnel connections in both conn.log and tunnel.log, and then proceeds to analyze the inner payload as if it were not tunneled, including also logging that session in conn.log. For SOCKS, it generates a new socks.log in addition with more information.

- Bro now features a flexible input framework that allows users to integrate external information in real-time into Bro while it's processing network traffic. The most direct use-case at the moment is reading data from ASCII files into Bro tables, with updates picked up automatically when the file changes during runtime. See doc/input.rst for more information.

  Internally, the input framework is structured around the notion of "reader plugins" that make it easy to interface to different data sources. We will add more in the future.

- BroControl now has built-in support for host-based load-balancing when using either PF_RING, Myricom cards, or individual interfaces. Instead of adding a separate worker entry in node.cfg for each Bro worker process on each worker host, it is now possible to just specify the number of worker processes on each host and BroControl configures everything correctly (including any neccessary enviroment variables for the balancers).

  This change adds three new keywords to the node.cfg file (to be used with worker entries): lb_procs (specifies number of workers on a host), lb_method (specifies what type of load balancing to use: pf_ring, myricom, or interfaces), and lb_interfaces (used only with "lb_method=interfaces" to specify which interfaces to load-balance on).

- Bro's default ASCII log format is not exactly the most efficient way for storing and searching large volumes of data. An alternatives, Bro now comes with experimental support for two alternative output formats:
  - DataSeries: an efficient binary format for recording structured bulk data. DataSeries is developed and maintained at HP Labs. See doc/logging-dataseries for more information.
  - ElasticSearch: a distributed RESTful, storage engine and search engine built on top of Apache Lucene. It scales very well, both for distributed indexing and distributed searching. See doc/logging-elasticsearch.rst for more information.

Note that at this point, we consider Bro's support for these two formats as prototypes for collecting experience with alternative outputs. We do not yet recommend them for production (but welcome feedback!)

### Changed Functionality

The following summarizes the most important differences in existing functionality. Note that this list is not complete, see CHANGES for the full set.

- Changes in dependencies:
  - Bro now requires CMake >= 2.6.3.
  - On Linux, Bro now links in tcmalloc (part of Google perftools) if found at configure time. Doing so can significantly improve memory and CPU use.

    On the other platforms, the new configure option –enable-perftools can be used to enable linking to tcmalloc. (Note that perftools's support for non-Linux platforms may be less reliable).
- The configure switch –enable-brov6 is gone.
- DNS name lookups performed by Bro now also query AAAA records. The results of the A and AAAA queries for a given hostname are combined such that at the scripting layer, the name resolution can yield a set with both IPv4 and IPv6 addresses.
- The connection compressor was already deprecated in 2.0 and has now been removed from the code base.
- We removed the "match" statement, which was no longer used by any of the default scripts, nor was it likely to be used by anybody anytime soon. With that, "match" and "using" are no longer reserved keywords.
- The syntax for IPv6 literals changed from "2607:f8b0:4009:802::1012" to "[2607:f8b0:4009:802::1012]". When an IP address variable or IP address literal is enclosed in pipes (for example, `|[fe80::db15]|`) the result is now the size of the address in bits (32 for IPv4 and 128 for IPv6).
- Bro now spawns threads for doing its logging. From a user's perspective not much should change, except that the OS may now show a bunch of Bro threads.
- We renamed the configure option –enable-perftools to –enable-perftools-debug to indicate that the switch is only relevant for debugging the heap.
- Bro's ICMP analyzer now handles both IPv4 and IPv6 messages with a joint set of events. The *icmp_conn* record got a new boolean field 'v6' that indicates whether the ICMP message is v4 or v6.
- Log postprocessor scripts get an additional argument indicating the type of the log writer in use (e.g., "ascii").
- BroControl's make-archive-name script also receives the writer type, but as its 2nd(!) argument. If you're using a custom version of that script, you need to adapt it. See the shipped version for details.
- Signature files can now be loaded via the new "@load-sigs" directive. In contrast to the existing (and still supported) signature_files constant, this can be used to load signatures relative to the current script (e.g., "@load-sigs ./foo.sig").
- The options "tunnel_port" and "parse_udp_tunnels" have been removed. Bro now supports decapsulating tunnels directly for protocols it understands.
- ASCII logs now record the time when they were opened/closed at the beginning and end of the file, respectively (wall clock). The options LogAscii::header_prefix and LogAscii::include_header have been renamed to LogAscii::meta_prefix and LogAscii::include_meta, respectively.
- The ASCII writers "header_*" options have been renamed to "meta_*" (because there's now also a footer).

- Some built-in functions have been removed: "addr_to_count" (use "addr_to_counts" instead), "bro_has_ipv6" (this is no longer relevant because Bro now always supports IPv6), "active_connection" (use "connection_exists" instead), and "connection_record" (use "lookup_connection" instead).
- The "NFS3::mode2string" built-in function has been renamed to "file_mode".
- Some built-in functions have been changed: "exit" (now takes the exit code as a parameter), "to_port" (now takes a string as parameter instead of a count and transport protocol, but "count_to_port" is still available), "connect" (now takes an additional string parameter specifying the zone of a non-global IPv6 address), and "listen" (now takes three additional parameters to enable listening on IPv6 addresses).
- Some Bro script variables have been renamed: "LogAscii::header_prefix" has been renamed to "LogAscii::meta_prefix", "LogAscii::include_header" has been renamed to "LogAscii::include_meta".
- Some Bro script variables have been removed: "tunnel_port", "parse_udp_tunnels", "use_connection_compressor", "cc_handle_resets", "cc_handle_only_syns", and "cc_instantiate_on_data".
- A couple events have changed: the "icmp_redirect" event now includes the target and destination addresses and any Neighbor Discovery options in the message, and the last parameter of the "dns_AAAA_reply" event has been removed because it was unused.
- The format of the ASCII log files has changed very slightly. Two new lines are automatically added, one to record the time when the log was opened, and the other to record the time when the log was closed.
- In BroControl, the option (in broctl.cfg) "CFlowAddr" was renamed to "CFlowAddress".

### Bro 2.0

As the version number jump from 1.5 suggests, Bro 2.0 is a major upgrade and lots of things have changed. Most importantly, we have rewritten almost all of Bro's default scripts from scratch, using quite different structure now and focusing more on operational deployment. The result is a system that works much better "out of the box", even without much initial site-specific configuration. The down-side is that 1.x configurations will need to be adapted to work with the new version. The two rules of thumb are:

1. If you have written your own Bro scripts that do not depend on any of the standard scripts formerly found in `policy/`, they will most likely just keep working (although you might want to adapt them to use some of the new features, like the new logging framework; see below).
2. If you have custom code that depends on specifics of 1.x default scripts (including most configuration tuning), that is unlikely to work with 2.x. We recommend to start by using just the new scripts first, and then port over any customizations incrementally as necessary (they may be much easier to do now, or even unnecessary). Send mail to the Bro user mailing list if you need help.

Below we summarize changes from 1.x to 2.x in more detail. This list isn't complete, see the `CHANGES` file in the distribution. for the full story.

### Script Organization

In versions before 2.0, Bro scripts were all maintained in a flat directory called `policy/` in the source tree. This directory is now renamed to `scripts/` and contains major subdirectories `base/`, `policy/`, and `site/`, each of which may also be subdivided further.

The contents of the new `scripts/` directory, like the old/flat `policy/` still gets installed under the `share/bro` subdirectory of the installation prefix path just like previous versions. For example, if Bro was compiled like `./configure --prefix=/usr/local/bro && make && make install`, then the script hierarchy can be found in `/usr/local/bro/share/bro`.

The main subdirectories of that hierarchy are as follows:

- `base/` contains all scripts that are loaded by Bro by default (unless the `-b` command line option is used to run Bro in a minimal configuration). Note that is a major conceptual change: rather than not loading anything by default, Bro now uses an extensive set of default scripts out of the box.

  The scripts under this directory generally either accumulate/log useful state/protocol information for monitored traffic, configure a default/recommended mode of operation, or provide extra Bro scripting-layer functionality that has no significant performance cost.
- `policy/` contains all scripts that a user will need to explicitly tell Bro to load. These are scripts that implement functionality/analysis that not all users may want to use and may have more significant performance costs. For a new installation, you should go through these and see what appears useful to load.
- `site/` remains a directory that can be used to store locally developed scripts. It now comes with some preinstalled example scripts that contain recommended default configurations going beyond the `base/` setup. E.g. `local.bro` loads extra scripts from `policy/` and does extra tuning. These files can be customized in place without being overwritten by upgrades/reinstalls, unlike scripts in other directories.

With version 2.0, the default `BROPATH` is set to automatically search for scripts in `policy/`, `site/` and their parent directory, but **not** `base/`. Generally, everything under `base/` is loaded automatically, but for users of the `-b` option, it's important to know that loading a script in that directory requires the extra `base/` path qualification. For example, the following two scripts:

- `$PREFIX/share/bro/base/protocols/ssl/main.bro`
- `$PREFIX/share/bro/policy/protocols/ssl/validate-certs.bro`

are referenced from another Bro script like:

Notice how `policy/` can be omitted as a convenience in the second case. `@load` can now also use relative path, e.g., `@load ../main`.

### Logging Framework

- The logs generated by scripts that ship with Bro are entirely redone to use a standardized, machine parsable format via the new logging framework. Generally, the log content has been restructured towards making it more directly useful to operations. Also, several analyzers have been significantly extended and thus now log more information. Take a look at `ssl.log`.
  - A particular format change that may be useful to note is that the `conn.log service` field is derived from DPD instead of well-known ports (while that was already possible in 1.5, it was not the default).
  - Also, `conn.log` now reports raw number of packets/bytes per endpoint.
- The new logging framework makes it possible to extend, customize, and filter logs very easily.
- A common pattern found in the new scripts is to store logging stream records for protocols inside the `connection` records so that state can be collected until enough is seen to log a coherent unit of information regarding the activity of that connection. This state is now frequently seen/accessible in event handlers, for example, like `c$<protocol>` where `<protocol>` is replaced by the name of the protocol. This field is added to the `connection` record by `redef`'ing it in a `base/protocols/<protocol>/main.bro` script.
- The logging code has been rewritten internally, with script-level interface and output backend now clearly separated. While ASCII logging is still the default, we will add further output types in the future (binary format, direct database logging).

### Notice Framework

The way users interact with "notices" has changed significantly in order to make it easier to define a site policy and more extensible for adding customized actions.

### New Default Settings

- Dynamic Protocol Detection (DPD) is now enabled/loaded by default.
- The default packet filter now examines all packets instead of dynamically building a filter based on which protocol analysis scripts are loaded. See `PacketFilter::all_packets` for how to revert to old behavior.

### API Changes

- The `@prefixes` directive works differently now. Any added prefixes are now searched for and loaded *after* all input files have been parsed. After all input files are parsed, Bro searches `BROPATH` for prefixed, flattened versions of all of the parsed input files. For example, if `lcl` is in `@prefixes`, and `site.bro` is loaded, then a file named `lcl.site.bro` that's in `BROPATH` would end up being automatically loaded as well. Packages work similarly, e.g. loading `protocols/http` means a file named `lcl.protocols.http.bro` in `BROPATH` gets loaded automatically.
- The `make_addr` BIF now returns a `subnet` versus an `addr`

### Variable Naming

- `Module` is more widely used for namespacing. E.g. the new `site.bro` exports the `local_nets` identifier (among other things) into the `Site` module.
- Identifiers may have been renamed to conform to new scripting conventions

### Removed Functionality

We have remove a bunch of functionality that was rarely used and/or had not been maintained for a while already:

- The `net` script data type.
- The `alarm` statement; use the notice framework instead.
- Trace rewriting.
- DFA state expiration in regexp engine.
- Active mapping.
- Native DAG support (may come back eventually)
- ClamAV support.
- The connection compressor is now disabled by default, and will be removed in the future.

### BroControl Changes

BroControl looks pretty much similar to the version coming with Bro 1.x, but has been cleaned up and streamlined significantly internally.

BroControl has a new `process` command to process a trace on disk offline using a similar configuration to what BroControl installs for live analysis.

BroControl now has an extensive plugin interface for adding new commands and options. Note that this is still considered experimental.

We have removed the `analysis` command, and BroControl currently does not send daily alarm summaries anymore (this may be restored later).

### Development Infrastructure

Bro development has moved from using SVN to Git for revision control. Users that want to use the latest Bro development snapshot by checking it out from the source repositories should see the development process. Note that all the various sub-components now reside in their own repositories. However, the top-level Bro repository includes them as git submodules so it's easy to check them all out simultaneously.

Bro now uses CMake for its build system so that is a new required dependency when building from source.

Bro now comes with a growing suite of regression tests in `testing/`.

### Detailed Version History

**Contents**

### Bro

```
2.4-921 | 2016-08-10 20:29:48 -0700

  * Add logging framework ext-data mechanism. It is now possible to
    extend logs by adding new data columns by them - either to specific
    ones, or globally to all logs. This can, e.g., be used to add node
    names to all logs. (Seth Hall)

  * Add unrolling separator & field name map to logging framework.
    One can now use logging separators other than ".", as well as
    change specific column names in logs. (Seth Hall)

  * Fix memory leak in EnumType. (Johanna Amann)

  * Fix configure warning when compiling with --enable-broker. (Johanna Amann)

  * Add netcontrol-connectors to aux directory. (Johanna Amann)

  * Update Mozilla CA list. (Johanna Amann)

  * update scripts loaded by default in local.bro. Traceroute is now disabled
    by default, stats and capture-loss enabled by default. (Johanna Amann)

2.4-907 | 2016-08-09 15:42:17 -0400

  * Updating NEWS.

2.4-905 | 2016-08-09 08:19:37 -0700

  * GSSAPI analyzer now forwards authentication blobs more correctly.
    (Seth Hall)

  * The KRB analyzer now includes support for the PA_ENCTYPE_INFO2
    pre-auth data type. (Seth Hall)

  * Add an argument to "disable_analyzer" function to not do a
    reporter message by default. (Seth Hall)

2.4-902 | 2016-08-08 16:50:35 -0400

  * Adding SMB analyzer. (Seth Hall, Vlad Grigorescu and many others)

  * NetControl: allow reasons in remove_rule calls. Addresses BIT-1655
    (Johanna Amann)

2.4-893 | 2016-08-05 15:43:04 -0700

  * Remove -z/--analysis option. (Johanna Amann)

  * Remove already defunct code for XML serialization. (Johanna Amann)

2.4-885 | 2016-08-05 15:03:59 -0700

  * Reverting SMB analyzer merge. (Robin Sommer)

2.4-883 | 2016-08-05 12:57:26 -0400

  * Add a new node type for logging with the cluster framework scripts by
```

```
    adding a new Bro node type for doing logging (this is intended to
    reduce the load on the manager). If a user chooses not to specify a
    logger node in the cluster configuration, then the manager will
    write logs locally as usual. (Daniel Thayer)

2.4-874 | 2016-08-05 12:43:06 -0400

  * SMB analyzer (Seth Hall, Vlad Grigorescu and many others)

2.4-759 | 2016-08-05 09:32:42 -0400

  * Intel framework improvements (Jan Grashoefer)
    * Added expiration for intelligence items.
    * Improved intel notices.
    * Added hook to allow extending the intel log.
    * Added support for subnets to intel-framework.

2.4-742 | 2016-08-02 15:28:31 -0700

  * Fix duplicate SSH authentication failure events. Addresses BIT-1641.
    (Robin Sommer)

  * Remove OpenSSL dependency for plugins. (Robin Sommer)

2.4-737 | 2016-08-02 11:38:07 -0700

  * Fix some Coverity warnings. (Robin Sommer)

2.4-735 | 2016-08-02 11:05:36 -0700

  * Added string slicing examples to documentation. (Moshe Kaplan)

2.4-733 | 2016-08-01 09:09:29 -0700

  * Fixing a CMake dependency issue for the pcap bifs. (Robin Sommer)

2.4-732 | 2016-08-01 08:33:00 -0700

  * Removing pkg/make-*-packages scripts. BIT-1509 #closed (Robin
    Sommer)

2.4-731 | 2016-08-01 08:14:06 -0700

  * Correct endianness of IP addresses in SNMP. Addresses BIT-1644.
    (Anony Mous)

2.4-729 | 2016-08-01 08:00:54 -0700

  * Fix behavior of connection_pending event. It is now really only
    raised when Bro is terminating. Also adds a test-case that raises
    the event. (Johanna Amann)

  * Retired remove -J/-K options (set md5/hash key) from the manpage.
    They had already been removed from the code. (Johanna Amann)

  * NetControl: Add catch-and-release event when IPs are forgotten.
    This adds an event catch_release_forgotten() that is raised once
    Catch & Release ceases block management for an IP address because
    the IP has not been seen in traffic during the watch interval.
    (Johanna Amann)

2.4-723 | 2016-07-26 15:04:26 -0700

  * Add error events to input framework. (Johanna Amann)

    This change introduces error events for Table and Event readers.
    Users can now specify an event that is called when an info,
    warning, or error is emitted by their input reader. This can,
    e.g., be used to raise notices in case errors occur when reading
    an important input stream.

    Example:

        event error_event(desc: Input::TableDescription, msg: string, level: Reporter:
↪:Level)
            {
            ...
            }

        event bro_init()
            {
            Input::add_table([$source="a", $error_ev=error_event, ...]);
            }

     Addresses BIT-1181.

  * Calling Error() in an input reader now automatically will disable
    the reader and return a failure in the Update/Heartbeat calls.
    (Johanna Amann)

  * Convert all errors in the ASCII formatter into warnings (to show
    that they are non-fatal. (Johanna Amann)

  * Enable SQLite shared cache mode. This allows all threads accessing
    the same database to share sqlite objects. See
    https://www.sqlite.org/sharedcache.html. Addresses BIT-1325.
    (Johanna Amann)

  * NetControl: Adjust default priority of ACTION_DROP hook to standad
    level. (Johanna Amann)

  * Fix types when constructing SYN_packet record. Fixes BIT-1650.
    (Grant Moyer).

2.4-715 | 2016-07-23 07:27:05 -0700

  * SQLite writer: Remove unused string formatting function. (Johanna Amann)

  * Deprecated the ElasticSearch log writer. (Johanna Amann)

2.4-709 | 2016-07-15 09:05:20 -0700

  * Change Bro's hashing for short inputs and Bloomfilters from H3 to
    Siphash, which produces much better results for HLL in particular.
    (Johanna Amann)
  * Fix a long-standing bug which truncated hash values to 32-bit on
    most machines. (Johanna Amann)

  * Fixes to HLL. Addresses BIT-1612. (Johanna Amann)

  * Add test checking the quality of HLL. (Johanna Amann)

  * Remove the -K/-J options for setting keys. (Johanna Amann)

  * SSL: Fix memory management problem. (Johanna Amann)

2.4-693 | 2016-07-12 11:29:17 -0700

  * Change TCP analysis to process connections without the initial SYN as
    non-partial connections. Addresses BIT-1492. (Robin Sommer).

2.4-691 | 2016-07-12 09:58:38 -0700

  * SSL: add support for signature_algorithms extension. (Johanna
    Amann)

2.4-688 | 2016-07-11 11:10:33 -0700

  * Disable broker by default. To enable it, use --enable-broker.
    Addresses BIT-1645. (Daniel Thayer)

2.4-686 | 2016-07-08 19:14:43 -0700

  * Added flagging of retransmission to the connection history.
    Addresses BIT-977. (Robin Sommer)

2.4-683 | 2016-07-08 14:55:04 -0700

  * Extendign connection history field to flag with '^' when Bro flips
    a connection's endpoints. Addresses BIT-1629. (Robin Sommer)

2.4-680 | 2016-07-06 09:18:21 -0700

  * Remove ack_above_hole() event, which was a subset of content_gap
    and led to plenty noise. Addresses BIT-688. (Robin Sommer)

2.4-679 | 2016-07-05 16:35:53 -0700

  * Fix segfault when an existing enum identifier is added again with
    a different value. Addresses BIT-931. (Robin Sommer)

  * Escape the empty indicator in logs if it occurs literally as a
    field's actual content. Addresses BIT-931. (Robin Sommer)

2.4-676 | 2016-06-30 17:27:54 -0700

  * A larger series of NetControl updates. (Johanna Amann)

        * Add NetControl framework documentation to the Bro manual.

        * Use NetControl for ACTION_DROP of notice framework. So far,
          this action did nothing by default.
        * Rewrite of catch-and-release.

        * Fix several small logging issues.

        * find_rules_subnet() now works in cluster mode. This
          introduces two new events, NetControl::rule_new and
          NetControl::rule_destroyed, which are raised when rules are
          first added and then deleted from the internal state
          tracking.

        * Fix acld whitelist command.

        * Add rule existance as a state besides added and failure.

        * Suppress duplicate "plugin activated" messages.

        * Make new Broker plugin options accessible.

        * Add predicates to Broker plugin.

  * Tweak SMTP scripts to not to pull in the notice framework.

2.4-658 | 2016-06-30 16:55:32 -0700

  * Fix a number of documentation building errors. (Johanna Amann)

  * Input/Logging: Make bool conversion operator explicit. (Johanna Amann)

  * Add new TLS ciphers from RFC 7905. (Johanna Amann)

2.4-648 | 2016-06-21 18:33:22 -0700

  * Fix memory leaks. Reported by Dk Jack. (Johanna Amann)

2.4-644 | 2016-06-21 13:59:05 -0400

  * Fix an off-by-one error when grabbing x-originating-ip header in
    email. (Seth Hall, Aashish Sharma)

2.4-642 | 2016-06-18 13:18:23 -0700

  * Fix potential mismatches when ignoring duplicate weirds. (Johanna Amann)

  * Weird: Rewrite internals of weird logging. (Johanna Amann)

      - "flow weirds" now actually log information about the flow
      that they occur in.

      - weirds can now be generated by calling Weird::weird() with
      the info record directly, allowing more fine-granular passing
      of information. This is e.g. used for DNS weirds.

    Addresses BIT-1578 (Johanna Amann)

  * Exec: fix reader cleanup when using read_files, preventing file
    descriptors from leaking every time it was used. (Johanna Amann)

  * Raw Writer: Make code more c++11-y, remove raw pointers. (Johanna
    Amann)

  * Add separate section with logging changes to NEWS. (Seth Hall)

2.4-635 | 2016-06-18 01:40:17 -0400

  * Add some documentation for modbus data types. Addresses
    BIT-1216. (Seth Hall)

  * Removed app-stats scripts. Addresses BIT-1171. (Seth Hall)

2.4-631 | 2016-06-16 16:45:10 -0400

  * Fixed matching mail address intel and added test (Jan Grashoefer)

  * A new utilities script named email.bro with some utilities
    for parsing out email addresses from strings. (Seth Hall)

  * SMTP "rcptto" and "mailfrom" fields now do some minimal
    parsing to clean up email addresses. (Seth Hall)

  * Added "cc" to the SMTP log and feed it into the Intel framework
    with the policy/frameworks/intel/seen/smtp.bro script. (Seth Hall)

2.4-623 | 2016-06-15 17:31:12 -0700

  * &default values are no longer overwritten with uninitialized
          by the input framework. (Jan Grashoefer)

2.4-621 | 2016-06-15 09:18:02 -0700

  * Fixing memory leak in changed table expiration code. (Robin
    Sommer)

  * Fixing test portability. (Robin Sommer)

  * Move the HTTP "filename" field (which was never filled out
    anyways) to "orig_filenames" and "resp_filenames".  (Seth Hall)

  * Add a round trip time (rtt) field to dns.log. (Seth Hall)

  * Add ACE archive files to the identified file types. Addresses
    BIT-1609. (Stephen Hosom)

2.4-613 | 2016-06-14 18:10:37 -0700

  * Preventing the event processing from looping endlessly when an
    event reraised itself during execution of its handlers. (Robin
    Sommer)

2.4-612 | 2016-06-14 17:42:52 -0700

  * Improved handling of 802.11 headers. (Jan Grashoefer)

2.4-609 | 2016-06-14 17:15:28 -0700

  * Fixed table expiration evaluation. The expiration attribute
    expression is now evaluated for every use. Thus later adjustments
    of the value (e.g. by redefining a const) will now take effect.
    Values less than 0 will disable expiration. (Jan Grashoefer)

2.4-606 | 2016-06-14 16:11:07 -0700

  * Fix parsing precedence of "hook" expression. Addresses BIT-1619
    (Johanna Amann)

  * Update the "configure" usage message for --with-caf (Daniel
    Thayer)

2.4-602 | 2016-06-13 08:16:34 -0700

  * Fixing Covertity warning (CID 1356391). (Robin Sommer)

  * Guarding against reading beyond packet data when accessing L2
    address in Radiotap header. (Robin Sommer)

2.4-600 | 2016-06-07 15:53:19 -0700

  * Fixing typo in BIF macros. Reported by Jeff Barber. (Robin Sommer)

2.4-599 | 2016-06-07 12:37:32 -0700

  * Add new functions haversine_distance() and haversine_distance_ip()
    for calculating geographic distances. They requires that Bro be
    built with libgeoip. (Aashish Sharma/Daniel Thayer).

2.4-597 | 2016-06-07 11:46:45 -0700

  * Fixing memory leak triggered by new MAC address logging. (Robin
    Sommer)

2.4-596 | 2016-06-07 11:07:29 -0700

  * Don't create debug.log immediately upon startup (BIT-1616).
    (Daniel Thayer)

2.4-594 | 2016-06-06 18:11:16 -0700

  * ASCII Input: Accept DOS/Windows newlines. Addresses BIT-1198
    (Johanna Amann)

  * Fix BinPAC exception in RFB analyzer. (Martin van Hensbergen)

  * Add URL decoding for the unofficial %u00AE style of encoding. (Seth Hall)

  * Remove the unescaped_special_char HTTP weird. (Seth Hall)

2.4-588 | 2016-06-06 17:59:34 -0700

  * Moved link-layer addresses into endpoints. The link-layer
    addresses are now part of the connection endpoints following the
    originator/responder pattern. (Jan Grashoefer)

  * Link-layer addresses are extracted for 802.11 plus RadioTap. (Jan
    Grashoefer)
  * Fix coverity error (uninitialized variable) (Johanna Amann)

  * Use ether_ntoa instead of ether_ntoa_r

    The latter is thread-safe, but a GNU addition which does not exist on
    OS-X. Since the function only is called in the main thread, it should
    not matter if it is or is not threadsafe. (Johanna Amann)

  * Fix FreeBSD/OSX compile problem due to headers (Johanna Amann)

2.4-581 | 2016-05-30 10:58:19 -0700

  * Adding missing new script file mac-logging.bro. (Robin Sommer)

2.4-580 | 2016-05-29 13:41:10 -0700

  * Add Ethernet MAC addresses to connection record. c$eth_src and
    c$eth_dst now contain the Ethernet address if available. A new
    script protocols/conn/mac-logging.bro adds these to conn.log when
    loaded. (Robin Sommer)

2.4-579 | 2016-05-29 08:54:57 -0700

  * Fixing Coverity warning. Addresses CID 1356116. (Robin Sommer)

  * Fixing FTP cwd getting overlue long. (Robin Sommer)

  * Clarifying notice documentation. Addresses BIT-1405. (Robin
    Sommer)

  * Changing protocol_{confirmation,violation} events to queue like
    any other event. Addresses BIT-1530. (Robin Sommer)

  * Normalizing test baseline. (Robin Sommer)

  * Do not use scientific notations when printing doubles in logs.
    Addresses BIT-1558. (Robin Sommer)

2.4-573 | 2016-05-23 13:21:03 -0700

  * Ignoring packets with negative timestamps. Addresses BIT-1562 and
    BIT-1443. (Robin Sommer)

2.4-572 | 2016-05-23 12:45:23 -0700

  * Fix for a table refering to a expire function that's not defined.
    Addresses BIT-1597. (Robin Sommer)

2.4-571 | 2016-05-23 08:26:43 -0700

  * Fixing a few Coverity warnings. (Robin Sommer)

2.4-569 | 2016-05-18 07:39:35 -0700

  * DTLS: Use magix constant from RFC 5389 for STUN detection.
    (Johanna Amann)

  * DTLS: Fix binpac bug with DTLSv1.2 client hellos. (Johanna Amann)
  * DTLS: Fix interaction with STUN. Now the DTLS analyzer cleanly
    skips all STUN messages. (Johanna Amann)

  * Fix the way that child analyzers are added. (Johanna Amann)

2.4-563 | 2016-05-17 16:25:21 -0700

  * Fix duplication of new_connection_contents event. Addresses
    BIT-1602 (Johanna Amann)

  * SMTP: Support SSL upgrade via X-ANONYMOUSTLS This seems to be a
    non-standardized microsoft extension that, besides having a
    different name, works pretty much the same as StartTLS. We just
    treat it as such. (Johanna Amann)

  * Fixing control framework's net_stats and peer_status commands. For
    the latter, this removes most of the values returned, as we don't
    have access to them anymore. (Robin Sommer)

2.4-555 | 2016-05-16 20:10:15 -0700

  * Fix failing plugin tests on OS X 10.11. (Daniel Thayer)

  * Fix failing test on Debian/FreeBSD. (Johanna Amann)

2.4-552 | 2016-05-12 08:04:33 -0700

  * Fix a bug in receiving remote logs via broker. (Daniel Thayer)

  * Fix Bro and unit tests when broker is not enabled. (Daniel Thayer)

  * Added interpreter error for local event variables. (Jan Grashoefer)

2.4-544 | 2016-05-07 12:19:07 -0700

  * Switching all use of gmtime and localtime to use reentrant
    variants. (Seth Hall)

2.4-541 | 2016-05-06 17:58:45 -0700

  * A set of new built-in function for gathering execution statistics:

      get_net_stats(), get_conn_stats(), get_proc_stats(),
      get_event_stats(), get_reassembler_stats(), get_dns_stats(),
      get_timer_stats(), get_file_analysis_stats(), get_thread_stats(),
      get_gap_stats(), get_matcher_stats().

    net_stats() resource_usage() have been superseded by these. (Seth
    Hall)

  * New policy script misc/stats.bro that records Bro execution
    statistics in a standard Bro log file. (Seth Hall)

  * A series of documentation improvements. (Daniel Thayer)

  * Rudimentary XMPP StartTLS analyzer. It parses certificates out of
    XMPP connections using StartTLS. It aborts processing if StartTLS
    is not found. (Johanna Amann)

2.4-507 | 2016-05-03 11:18:16 -0700

  * Fix incorrect type tags in Bro broker source code. These are just
    used for error reporting. (Daniel Thayer)

  * Update docs and tests of the fmt() function. (Daniel Thayer)

2.4-500 | 2016-05-03 11:16:50 -0700

  * Updating submodule(s).

2.4-498 | 2016-04-28 11:34:52 -0700

  * Rename Broker::print to Broker::send_print and Broker::event to
    Broker::send_event to avoid using reserved keywords as function
    names. (Daniel Thayer)

  * Add script wrapper functions for Broker BIFs. This faciliates
    documenting them through Broxygen. (Daniel Thayer)

  * Extend, update, and clean up Broker tests. (Daniel Thayer)

  * Intel: Allow to provide uid/fuid instead of conn/file. (Johanna
    Amann)

  * Provide file IDs for hostname matches in certificates. (Johanna
    Amann)

  * Rudimentary IMAP StartTLS analyzer. It parses certificates out of
    IMAP connections using StartTLS. It aborts processing if StartTLS
    is not found. (Johanna Amann)

2.4-478 | 2016-04-28 09:56:24

  * Fix parsing of x509 pre-y2k dates. (Johanna Amann)

  * Fix small error in bif documentation. (Johanna Amann)

  * Fix unknown data link type error message. (Vitaly Repin)

  * Correcting spelling errors. (Jeannette Dopheide)

  * Minor cleanup in ARP analyzer. (Johanna Amann)

  * Fix parsing of pre-y2k dates in X509 certificates. (Johanna Amann)

  * Fix small error in get_current_packet documentation. (Johanna Amann)

2.4-471 | 2016-04-25 15:37:15 -0700

  * Add DNS tests for huge TLLs and CAA. (Johanna Amann)

  * Add DNS "CAA" RR type and event. (Mark Taylor)

  * Fix DNS response parsing: TTLs are unsigned. (Mark Taylor)
```

```
2.4-466 | 2016-04-22 16:25:33 -0700

  * Rename BrokerStore and BrokerComm to Broker. Also split broker main.bro
    into two scripts. (Daniel Thayer)

  * Add get_current_packet_header bif. (Jan Grashoefer)

2.4-457 | 2016-04-22 08:36:27 -0700

  * Fix Intel framework not checking the CERT_HASH indicator type. (Johanna Amann)

2.4-454 | 2016-04-14 10:06:58 -0400

  * Additional mime types for file identification and a few fixes. (Seth Hall)

    New file mime types:
     - .ini files
     - MS Registry policy files
     - MS Registry files
     - MS Registry format files (e.g. DESKTOP.DAT)
     - MS Outlook PST files
     - Apple AFPInfo files

    Mime type fixes:
     - MP3 files with ID3 tags.
     - JSON and XML matchers were extended

  * Avoid a macro name conflict on FreeBSD. (Seth Hall, Daniel Thayer)

2.4-452 | 2016-04-13 01:15:20 -0400

  * Add a simple file entropy analyzer. (Seth Hall)

  * Analyzer and bro script for RFB/VNC protocol (Martin van Hensbergen)

    This analyzer parses the Remote Frame Buffer
    protocol, usually referred to as the 'VNC protocol'.

    It supports several dialects (3.3, 3.7, 3.8) and
    also handles the Apple Remote Desktop variant.

    It will log such facts as client/server versions,
    authentication method used, authentication result,
    height, width and name of the shared screen.


2.4-430 | 2016-04-07 13:36:36 -0700

  * Fix regex literal in scripting documentation. (William Tom)

2.4-428 | 2016-04-07 13:33:08 -0700

  * Confirm protocol in SNMP/SIP only if we saw a response SNMP/SIP
    packet. (Vlad Grigorescu)

2.4-424 | 2016-03-24 13:38:47 -0700

  * Only load openflow/netcontrol if compiled with broker. (Johanna Amann)
  * Adding canonifier to test. (Robin Sommer)

2.4-422 | 2016-03-21 19:48:30 -0700

  * Adapt to recent change in CAF CMake script. (Matthias Vallentin)

  * Deprecate --with-libcaf in favor of --with-caf, as already done in
    Broker. (Matthias Vallentin)

2.4-418 | 2016-03-21 12:22:15 -0700

  * Add protocol confirmation to MySQL analyzer. (Vlad Grigorescu)

  * Check that there is only one of &read_expire, &write_expire,
    &create_expire. (Johanna Amann)

  * Fixed &read_expire for subnet-indexed tables, plus test case. (Jan
    Grashoefer)

  * Add filter_subnet_table() that works similar to matching_subnet()
    but returns a filtered view of the original set/table only
    containing the changed subnets. (Jan Grashoefer)

  * Fix bug in tablue values' tracking read operations. (Johanna
    Amann)

  * Update TLS constants and extensions from IANA. (Johanna Amann)

2.4-406 | 2016-03-11 14:27:47 -0800

  * Add NetControl and OpenFlow frameworks.  (Johanna Amann)

2.4-313 | 2016-03-08 07:47:57 -0800

  * Remove old string functions in C++ code. This removes the
    functions: strcasecmp_n, strchr_n, and strrchr_n. (Johanna Amann)

2.4-307 | 2016-03-07 13:33:45 -0800

  * Add "disable_analyzer_after_detection" and remove
    "skip_processing_after_detection". Addresses BIT-1545.
    (Aaron Eppert & Johanna Amann)

  * Add bad_HTTP_request_with_version weird (William Glodek)

2.4-299 | 2016-03-04 12:51:55 -0800

  * More detailed installation instructions for FreeBSD 9.X. (Johanna Amann)

  * Update CMake OpenSSL checks. (Johanna Amann)

  * "SUBSCRIBE" is a valid SIP. message per RFC 3265. Addresses
     BIT-1529. (Johanna Amann)

  * Update documentation for connection log's RSTR. Addresses BIT-1535
    (Johanna Amann)
```

```
2.4-284 | 2016-02-17 14:12:15 -0800

  * Fix sometimes failing dump-events test. (Johanna Amann)

2.4-282 | 2016-02-13 10:48:21 -0800

  * Add missing break in in StartTLS case of IRC analyzer. Found by
    Aaron Eppert. (Johanna Amann)

2.4-280 | 2016-02-13 10:40:16 -0800

  * Fix memory leaks in stats.cc and smb.cc. (Johanna Amann)

2.4-278 | 2016-02-12 18:53:35 -0800

  * Better multi-space separator handline. (Mark Taylor & Johanna Amann)

2.4-276 | 2016-02-10 21:29:33 -0800

  * Allow IRC commands to not have parameters. (Mark Taylor)

2.4-272 | 2016-02-08 14:27:58 -0800

  * fix memory leaks in find_all() and IRC analyzer. (Dirk Leinenbach)

2.4-270 | 2016-02-08 13:00:57 -0800

  * Removed duplicate parameter for IRC "QUIT" event handler. (Mark Taylor)

2.4-267 | 2016-02-01 12:38:32 -0800

  * Add testcase for CVE-2015-3194. (Johanna Amann)

  * Fix portability issue with use of mktemp. (Daniel Thayer)

2.4-260 | 2016-01-28 08:05:27 -0800

  * Correct irc_privmsg_message event handling bug. (Mark Taylor)

  * Update copyright year for Sphinx. (Johanna Amann)

2.4-253 | 2016-01-20 17:41:20 -0800

  * Support of RadioTap encapsulation for 802.11 (Seth Hall)

    Radiotap support should be fully functional with Radiotap
    packets that include IPv4 and IPv6. Other radiotap packets are
    silently ignored.

2.4-247 | 2016-01-19 10:19:48 -0800

  * Fixing C++11 compiler warnings. (Seth Hall)

  * Updating plugin documentation building. (Johanna Amann)

2.4-238 | 2016-01-15 12:56:33 -0800

  * Add HTTP version information to HTTP log file. (Aaron Eppert)
  * Add NOTIFY as a valid SIP message, per RFC 3265. (Aaron Eppert)

  * Improve HTTP parser's handling of requests that don't have a URI.
    (William Glodek/Robin Sommer)

  * Fix crash when deleting non existing record member. Addresses
    BIT-1519. (Johanna Amann)

2.4-228 | 2015-12-19 13:40:09 -0800

  * Updating BroControl submodule.

2.4-227 | 2015-12-18 17:47:24 -0800

  * Update host name in windows-version-detection.bro. (Aaron Eppert)

  * Update installation instructions to mention OpenSSL dependency for
    newer OS X version. (Johanna Amann)

  * Change a stale bro-ids.org to bro.org. (Johanna Amann)

  * StartTLS support for IRC. (Johanna Amann)

  * Adding usage guard to canonifier script. (Robin Sommer)

2.4-217 | 2015-12-04 16:50:46 -0800

  * SIP scripts code cleanup. (Seth Hall)

     - Daniel Guerra pointed out a type issue for SIP request and
       response code length fields which is now corrected.

     - Some redundant code was removed.

     - if/else tree modified to use switch instead.

2.4-214 | 2015-12-04 16:40:15 -0800

  * Delaying BinPAC initializaton until afte plugins have been
    activated. (Robin Sommer)

2.4-213 | 2015-12-04 15:25:48 -0800

  * Use better data structure for storing BPF filters. (Robin Sommer)

2.4-211 | 2015-11-17 13:28:29 -0800

  * Making cluster reconnect timeout configurable. (Robin Sommer)

  * Bugfix for child process' communication loop. (Robin Sommer)

2.4-209 | 2015-11-16 07:31:22 -0800

  * Updating submodule(s).

2.4-207 | 2015-11-10 13:34:42 -0800
```

```
  * Fix to compile with OpenSSL that has SSLv3 disalbed. (Christoph
    Pietsch)

  * Fix potential race condition when logging VLAN info to conn.log.
    (Daniel Thayer)

2.4-201 | 2015-10-27 16:11:15 -0700

  * Updating NEWS. (Robin Sommer)

2.4-200 | 2015-10-26 16:57:39 -0700

  * Adding missing file. (Robin Sommer)

2.4-199 | 2015-10-26 16:51:47 -0700

  * Fix problem with the JSON Serialization code. (Aaron Eppert)

2.4-188 | 2015-10-26 14:11:21 -0700

  * Extending rexmit_inconsistency() event to receive an additional
    parameter with the packet's TCP flags, if available. (Robin
    Sommer)

2.4-187 | 2015-10-26 13:43:32 -0700

  * Updating NEWS for new plugins. (Robin Sommer)

2.4-186 | 2015-10-23 15:07:06 -0700

  * Removing pcap options for AF_PACKET support. Addresses BIT-1363.
    (Robin Sommer)

  * Correct a typo in controller.bro documentation. (Daniel Thayer)

  * Extend SSL DPD signature to allow alert before server_hello.
    (Johanna Amann)

  * Make join_string_vec work with vectors containing empty elements.
    (Johanna Amann)

  * Fix support for HTTP CONNECT when server adds headers to response.
    (Eric Karasuda).

  * Load static CA list for validation tests too. (Johanna Amann)

  * Remove cluster certificate validation script. (Johanna Amann)

  * Fix a bug in diff-remove-x509-names canonifier. (Daniel Thayer)

  * Fix test canonifiers in scripts/policy/protocols/ssl. (Daniel
    Thayer)

2.4-169 | 2015-10-01 17:21:21 -0700

  * Fixed parsing of V_ASN1_GENERALIZEDTIME timestamps in x509
    certificates. (Yun Zheng Hu)
  * Improve X509 end-of-string-check code. (Johanna Amann)

  * Refactor X509 generalizedtime support and test. (Johanna Amann)

  * Fix case of offset=-1 (EOF) for RAW reader. Addresses BIT-1479.
    (Johanna Amann)

  * Improve a number of test canonifiers. (Daniel Thayer)

  * Remove unnecessary use of TEST_DIFF_CANONIFIER. (Daniel Thayer)

  * Fixed some test canonifiers to read only from stdin

  * Remove unused test canonifier scripts. (Daniel Thayer)

  * A potpourri of updates and improvements across the documentation.
    (Daniel Thayer)

  * Add configure option to disable Broker Python bindings. Also
    improve the configure summary output to more clearly show whether
    or not Broker Python bindings will be built. (Daniel Thayer)

2.4-131 | 2015-09-11 12:16:39 -0700

  * Add README.rst symlink. Addresses BIT-1413 (Vlad Grigorescu)

2.4-129 | 2015-09-11 11:56:04 -0700

  * hash-all-files.bro depends on base/files/hash (Richard van den Berg)

  * Make dns_max_queries redef-able, and bump default to 25. Addresses
    BIT-1460 (Vlad Grigorescu)

2.4-125 | 2015-09-03 20:10:36 -0700

  * Move SIP analyzer to flowunit instead of datagram Addresses
    BIT-1458 (Vlad Grigorescu)

2.4-122 | 2015-08-31 14:39:41 -0700

  * Add a number of out-of-bound checks to layer 2 code. Addresses
    BIT-1463 (Johanna Amann)

  * Fix error in 2.4 release notes regarding SSH events. (Robin
    Sommer)

2.4-118 | 2015-08-31 10:55:29 -0700

  * Fix FreeBSD build errors (Johanna Amann)

2.4-117 | 2015-08-30 22:16:24 -0700

  * Fix initialization of a pointer in RDP analyzer. (Daniel
    Thayer/Robin Sommer)

2.4-115 | 2015-08-30 21:57:35 -0700

  * Enable Bro to leverage packet fanout mode on Linux. (Kris
    Nielander).

        ## Toggle whether to do packet fanout (Linux-only).
        const Pcap::packet_fanout_enable = F &redef;

        ## If packet fanout is enabled, the id to sue for it. This should be shared␣
↪amongst
        ## worker processes processing the same socket.
        const Pcap::packet_fanout_id = 0 &redef;

        ## If packet fanout is enabled, whether packets are to be defragmented before
        ## fanout is applied.
        const Pcap::packet_fanout_defrag = T &redef;

  * Allow libpcap buffer size to be set via configuration. (Kris Nielander)

        ## Number of Mbytes to provide as buffer space when capturing from live
        ## interfaces.
        const Pcap::bufsize = 128 &redef;

  * Move the pcap-related script-level identifiers into the new Pcap
    namespace. (Robin Sommer)

        snaplen                  -> Pcap::snaplen
        precompile_pcap_filter() -> Pcap::precompile_pcap_filter()
        install_pcap_filter()    -> Pcap::install_pcap_filter()
        pcap_error()             -> Pcap::pcap_error()


2.4-108 | 2015-08-30 20:14:31 -0700

   * Update Base64 decoding.  (Jan Grashoefer)

        - A new built-in function, decode_base64_conn() for Base64
          decoding. It works like decode_base64() but receives an
          additional connection argument that will be used for
          reporting decoding errors into weird.log (instead of
          reporter.log).

        - FTP, POP3, and HTTP analyzers now likewise log Base64
          decoding errors to weird.log.

        - The built-in functions decode_base64_custom() and
          encode_base64_custom() are now deprecated. Their
          functionality is provided directly by decode_base64() and
          encode_base64(), which take an optional parameter to change
          the Base64 alphabet.

  * Fix potential crash if TCP header was captured incompletely.
     (Robin Sommer)

2.4-103 | 2015-08-29 10:51:55 -0700

  * Make ASN.1 date/time parsing more robust. (Johanna Amann)

  * Be more permissive on what characters we accept as an unquoted
    multipart boundary. Addresses BIT-1459. (Johanna Amann)
```

```
2.4-99 | 2015-08-25 07:56:57 -0700

  * Add ``Q`` and update ``I`` documentation for connection history
    field. Addresses BIT-1466. (Vlad Grigorescu)

2.4-96 | 2015-08-21 17:37:56 -0700

  * Update SIP analyzer. (balintm)

        - Allows space on both sides of ':'.
        - Require CR/LF after request/reply line.

2.4-94 | 2015-08-21 17:31:32 -0700

  * Add file type detection support for video/MP2T. (Mike Freemon)

2.4-93 | 2015-08-21 17:23:39 -0700

  * Make plugin install honor DESTDIR= convention. (Jeff Barber)

2.4-89 | 2015-08-18 07:53:36 -0700

  * Fix diff-canonifier-external to use basename of input file.
  (Daniel Thayer)

2.4-87 | 2015-08-14 08:34:41 -0700

  * Removing the yielding_teredo_decapsulation option. (Robin Sommer)

2.4-86 | 2015-08-12 17:02:24 -0700

  * Make Teredo DPD signature more precise. (Martina Balint)

2.4-84 | 2015-08-10 14:44:39 -0700

  * Add hook 'HookSetupAnalyzerTree' to allow plugins access to a
    connection's initial analyzer tree for customization. (James
    Swaro)

  * Plugins now look for a file "__preload__.bro" in the top-level
    script directory. If found, they load it first, before any scripts
    defining BiF elements. This can be used to define types that the
    BiFs already depend on (like a custom type for an event argument).
    (Robin Sommer)

2.4-81 | 2015-08-08 07:38:42 -0700

  * Fix a test that is failing very frequently. (Daniel Thayer)

2.4-78 | 2015-08-06 22:25:19 -0400

  * Remove build dependency on Perl (now requiring Python instad).
    (Daniel Thayer)

  * CID 1314754: Fixing unreachable code in RSH analyzer. (Robin
    Sommer)

  * CID 1312752: Add comment to mark 'case' fallthrough as ok. (Robin
    Sommer)

  * CID 1312751: Removing redundant assignment. (Robin Sommer)

2.4-73 | 2015-07-31 08:53:49 -0700

  * BIT-1429: SMTP logs now include CC: addresses. (Albert Zaharovits)

2.4-70 | 2015-07-30 07:23:44 -0700

  * Updated detection of Flash and AdobeAIR. (Jan Grashoefer)

  * Adding tests for Flash version parsing and browser plugin
    detection. (Robin Sommer)

2.4-63 | 2015-07-28 12:26:37 -0700

  * Updating submodule(s).

2.4-61 | 2015-07-28 12:13:39 -0700

  * Renaming config.h to bro-config.h. (Robin Sommer)

2.4-58 | 2015-07-24 15:06:07 -0700

  * Add script protocols/conn/vlan-logging.bro to record VLAN data in
    conn.log. (Aaron Brown)

  * Add field "vlan" and "inner_vlan" to connection record. (Aaron
    Brown)

  * Save the inner vlan in the Packet object for Q-in-Q setups. (Aaron
    Brown)

  * Increasing plugin API version for recent packet source changes.
    (Robin Sommer)

  * Slightly earlier protocol confirmation for POP3. (Johanna Amann)

2.4-46 | 2015-07-22 10:56:40 -0500

  * Fix broker python bindings install location to track --prefix.
    (Jon Siwek)

2.4-45 | 2015-07-21 15:19:43 -0700

  * Enabling Broker by default. This means CAF is now a required
    dependency, altjough for now at least, there's still a switch
    --disable-broker to turn it off.

  * Requiring a C++11 compiler, and turning on C++11 support. (Robin
    Sommer)

  * Tweaking the listing of hooks in "bro -NN" for consistency. (Robin
    Sommer)

2.4-41 | 2015-07-21 08:35:17 -0700
```

```
  * Fixing compiler warning. (Robin Sommer)

  * Updates to IANA TLS registry. (Johanna Amann)

2.4-38 | 2015-07-20 15:30:35 -0700

  * Refactor code to use a common Packet type throught. (Jeff
    Barber/Robin Sommer)

  * Extend parsing layer 2 and keeping track of layer 3 protoco. (Jeff Barber)

  * Add a raw_packet() event that generated for all packets and
    include layer 2 information. (Jeff Barber)

2.4-27 | 2015-07-15 13:31:49 -0700

  * Fix race condition in intel test. (Johanna Amann)

2.4-24 | 2015-07-14 08:04:11 -0700

  * Correct Perl package name on FreeBSD in documentation.(Justin Azoff)

  * Adding an environment variable to BTest configuration for external
    scripts. (Robin Sommer)

2.4-20 | 2015-07-03 10:40:21 -0700

  * Adding a weird for when truncated packets lead TCP reassembly to
    ignore content. (Robin Sommer)

2.4-19 | 2015-07-03 09:04:54 -0700

  * A set of tests exercising IP defragmentation and TCP reassembly.
    (Robin Sommer)

2.4-17 | 2015-06-28 13:02:41 -0700

  * BIT-1314: Add detection for Quantum Insert attacks. The TCP
    reassembler can now keep a history of old TCP segments using the
    tcp_max_old_segments option. An overlapping segment with different
    data will then generate an rexmit_inconsistency event. The default
    for tcp_max_old_segments is zero, which disabled any additional
    buffering. (Yun Zheng Hu/Robin Sommer)

2.4-14 | 2015-06-28 12:30:12 -0700

  * BIT-1400: Allow '<' and '>' in MIME multipart boundaries. The spec
    doesn't actually seem to permit these, but they seem to occur in
    the wild. (Jon Siwek)

2.4-12 | 2015-06-28 12:21:11 -0700

  * BIT-1399: Trying to decompress deflated HTTP content even when
    zlib headers are missing. (Seth Hall)

2.4-10 | 2015-06-25 07:11:17 -0700

  * Correct a name used in a header identifier (Justin Azoff)
```

```
2.4-8 | 2015-06-24 07:50:50 -0700

  * Restore the --load-seeds cmd-line option and enable the short
    options -G/-H for --load-seeds/--save-seeds. (Daniel Thayer)

2.4-6 | 2015-06-19 16:26:40 -0700

  * Generate protocol confirmations for Modbus, making it appear as a
    confirmed service in conn.log. (Seth Hall)

  * Put command line options in alphabetical order. (Daniel Thayer)

  * Removing dead code for no longer supported -G switch. (Robin
    Sommer) (Robin Sommer)

2.4 | 2015-06-09 07:30:53 -0700

  * Release 2.4.

  * Fixing tiny thing in NEWS. (Robin Sommer)

2.4-beta-42 | 2015-06-08 09:41:39 -0700

  * Fix reporter errors with GridFTP traffic. (Robin Sommer)

2.4-beta-40 | 2015-06-06 08:20:52 -0700

  * PE Analyzer: Change how we calculate the rva_table size. (Vlad Grigorescu)

2.4-beta-39 | 2015-06-05 09:09:44 -0500

  * Fix a unit test to check for Broker requirement. (Jon Siwek)

2.4-beta-38 | 2015-06-04 14:48:37 -0700

  * Test for Broker termination. (Robin Sommer)

2.4-beta-37 | 2015-06-04 07:53:52 -0700

  * BIT-1408: Improve I/O loop and Broker IOSource. (Jon Siwek)

2.4-beta-34 | 2015-06-02 10:37:22 -0700

  * Add signature support for F4M files. (Seth Hall)

2.4-beta-32 | 2015-06-02 09:43:31 -0700

  * A larger set of documentation updates, fixes, and extentions.
    (Daniel Thayer)

2.4-beta-14 | 2015-06-02 09:16:44 -0700

  * Add memleak btest for attachments over SMTP. (Vlad Grigorescu)

  * BIT-1410: Fix flipped tx_hosts and rx_hosts in files.log. Reported
    by Ali Hadi. (Vlad Grigorescu)
  * Updating the Mozilla root certs. (Seth Hall)

  * Updates for the urls.bro script. Fixes BIT-1404. (Seth Hall)

2.4-beta-6 | 2015-05-28 13:20:44 -0700

  * Updating submodule(s).

2.4-beta-2 | 2015-05-26 08:58:37 -0700

  * Fix segfault when DNS is not available. Addresses BIT-1387. (Frank
    Meier and Robin Sommer)

2.4-beta | 2015-05-07 21:55:31 -0700

  * Release 2.4-beta.

  * Update local-compat.test (Johanna Amann)

2.3-913 | 2015-05-06 09:58:00 -0700

  * Add /sbin to PATH in btest.cfg and remove duplicate default_path.
    (Daniel Thayer)

2.3-911 | 2015-05-04 09:58:09 -0700

  * Update usage output and list of command line options. (Daniel
    Thayer)

  * Fix to ssh/geo-data.bro for unset directions. (Vlad Grigorescu)

  * Improve SIP logging and remove reporter messages. (Seth Hall)

2.3-905 | 2015-04-29 17:01:30 -0700

  * Improve SIP logging and remove reporter messages. (Seth Hall)

2.3-903 | 2015-04-27 17:27:59 -0700

  * BIT-1350: Improve record coercion type checking. (Jon Siwek)

2.3-901 | 2015-04-27 17:25:27 -0700

  * BIT-1384: Remove -O (optimize scripts) command-line option, which
    hadn't been working for a while already. (Jon Siwek)

2.3-899 | 2015-04-27 17:22:42 -0700

  * Fix the -J/--set-seed cmd-line option. (Daniel Thayer)

  * Remove unused -l, -L, and -Z cmd-line options. (Daniel Thayer)

2.3-892 | 2015-04-27 08:22:22 -0700

  * Fix typos in the Broker BIF documentation. (Daniel Thayer)

  * Update installation instructions and remove outdated references.
    (Johanna Amann)
  * Easier support for systems with tcmalloc_minimal installed. (Seth
    Hall)

2.3-884 | 2015-04-23 12:30:15 -0500

  * Fix some outdated documentation unit tests. (Jon Siwek)

2.3-883 | 2015-04-23 07:10:36 -0700

  * Fix -N option to work with builtin plugins as well. (Robin Sommer)

2.3-882 | 2015-04-23 06:59:40 -0700

  * Add missing .pac dependencies for some binpac analyzer targets.
    (Jon Siwek)

2.3-879 | 2015-04-22 10:38:07 -0500

  * Fix compile errors. (Jon Siwek)

2.3-878 | 2015-04-22 08:21:23 -0700

  * Fix another compiler warning in DTLS. (Johanna Amann)

2.3-877 | 2015-04-21 20:14:16 -0700

  * Adding missing include. (Robin Sommer)

2.3-876 | 2015-04-21 16:40:10 -0700

  * Attempt at fixing a potential std::length_error exception in RDP
    analyzer. Addresses BIT-1337. (Robin Sommer)

  * Fixing compile problem caused by overeager factorization. (Robin
    Sommer)

2.3-874 | 2015-04-21 16:09:20 -0700

  * Change details of escaping when logging/printing. (Seth Hall/Robin
    Sommer)

        - Log files now escape non-printable characters consistently
          as "\xXX'. Furthermore, backslashes are escaped as "\\",
          making the representation fully reversible.

        - When escaping via script-level functions (escape_string,
          clean), we likewise now escape consistently with "\xXX" and
          "\\".

        - There's no "alternative" output style anymore, i.e., fmt()
          '%A' qualifier is gone.

    Addresses BIT-1333.

  * Remove several BroString escaping methods that are no longer
    useful. (Seth Hall)
```

```
2.3-864 | 2015-04-21 15:24:02 -0700

  * A SIP protocol analyzer. (Vlad Grigorescu)

        Activity gets logged into sip.log. It generates the following
    events:

            event sip_request(c: connection, method: string, original_URI: string,␣
↪version: string);
        event sip_reply(c: connection, version: string, code: count, reason: string);
        event sip_header(c: connection, is_orig: bool, name: string, value: string);
        event sip_all_headers(c: connection, is_orig: bool, hlist: mime_header_list);
        event sip_begin_entity(c: connection, is_orig: bool);
        event sip_end_entity(c: connection, is_orig: bool);

    The analyzer support SIP over UDP currently.

  * BIT-1343: Factor common ASN.1 code from RDP, SNMP, and Kerberos
    analyzers. (Jon Siwek/Robin Sommer)

2.3-838 | 2015-04-21 13:40:12 -0700

  * BIT-1373: Fix vector index assignment reference count bug. (Jon Siwek)

2.3-836 | 2015-04-21 13:37:31 -0700

  * Fix SSH direction field being unset. Addresses BIT-1365. (Vlad
    Grigorescu)

2.3-835 | 2015-04-21 16:36:00 -0500

  * Clarify Broker examples. (Jon Siwek)

2.3-833 | 2015-04-21 12:38:32 -0700

  * A Kerberos protocol analyzer. (Vlad Grigorescu)

        Activity gets logged into kerberos.log. It generates the following
    events:

        event krb_as_request(c: connection, msg: KRB::KDC_Request);
        event krb_as_response(c: connection, msg: KRB::KDC_Response);
        event krb_tgs_request(c: connection, msg: KRB::KDC_Request);
        event krb_tgs_response(c: connection, msg: KRB::KDC_Response);
        event krb_ap_request(c: connection, ticket: KRB::Ticket, opts: KRB::AP_
↪Options);
        event krb_priv(c: connection, is_orig: bool);
        event krb_safe(c: connection, is_orig: bool, msg: KRB::SAFE_Msg);
        event krb_cred(c: connection, is_orig: bool, tickets: KRB::Ticket_Vector);
        event krb_error(c: connection, msg: KRB::Error_Msg);

2.3-793 | 2015-04-20 20:51:00 -0700

  * Add decoding of PROXY-AUTHORIZATION header to HTTP analyze,
    treating it the same as AUTHORIZATION. (Josh Liburdi)

  * Remove deprecated fields "hot" and "addl" from the connection
    record. Remove the functions append_addl() and
    append_addl_marker(). (Robin Sommer)

  * Removing the NetFlow analyzer, which hasn't been used anymore
    since then corresponding command-line option went away. (Robin
    Sommer)

2.3-787 | 2015-04-20 19:15:23 -0700

  * A file analyzer for Portable Executables. (Vlad Grigorescu/Seth
    Hall).

        Activity gets logged into pe.log. It generates the following
        events:

        event pe_dos_header(f: fa_file, h: PE::DOSHeader);
        event pe_dos_code(f: fa_file, code: string);
        event pe_file_header(f: fa_file, h: PE::FileHeader);
        event pe_optional_header(f: fa_file, h: PE::OptionalHeader);
        event pe_section_header(f: fa_file, h: PE::SectionHeader);

2.3-741 | 2015-04-20 13:12:39 -0700

  * API changes to file analysis mime type detection. Removed
    "file_mime_type" and "file_mime_types" event, replacing them with
    a new event called "file_metadata_inferred". Addresses BIT-1368.
    (Jon Siwek)

  * A large series of improvements for file type identification. This
    inludes a many signature updates (new types, cleanup, performance
    improvments) and splitting out signatures into subfiles. (Seth
    Hall)

  * Fix an issue with files having gaps before the bof_buffer is
    filled, which could lead to file type identification not working
    correctly. (Seth Hall)

  * Fix an issue with packet loss in HTTP file reporting for file type
    identification wasn't working correctly zero-length bodies. (Seth
    Hall)

  * X.509 certificates are now populating files.log with the mime type
    application/pkix-cert. (Seth Hall)

  * Normalized some FILE_ANALYSIS debug messages. (Seth Hall)

2.3-725 | 2015-04-20 12:54:54 -0700

  * Updating submodule(s).

2.3-724 | 2015-04-20 14:11:02 -0500

  * Fix uninitialized field in raw input reader. (Jon Siwek)

2.3-722 | 2015-04-20 12:59:03 -0500

  * Remove unneeded documentation cross-referencing. (Jon Siwek)

2.3-721 | 2015-04-20 12:47:05 -0500
```

```
  * BIT-1380: Improve Broxygen output of &default expressions.
    (Jon Siwek)

2.3-720 | 2015-04-17 14:18:26 -0700

  * Updating NEWS.

2.3-716 | 2015-04-17 13:06:37 -0700

  * Add seeking functionality to raw reader. One can now add an option
    "offset" to the config map. Positive offsets are interpreted to be
    from the beginning of the file, negative from the end of the file
    (-1 is end of file). Only works for raw reader in streaming or
    manual mode. Does not work with executables. Addresses BIT-985.
    (Johanna Amann)

  * Allow setting packet and byte thresholds for connections. (Johanna Amann)

    This extends the ConnSize analyzer to be able to raise events when
    each direction of a connection crosses a certain amount of bytes
    or packets.

    Thresholds are set using:
        - set_conn_bytes_threshold(c$id, [num-bytes], [direction]);
        - set_conn_packets_threshold(c$id, [num-packets], [direction]);

    They raise the events, respectively:
        - event conn_bytes_threshold_crossed(c: connection, threshold: count, is_orig:
↪ bool)
        - event conn_packets_threshold_crossed(c: connection, threshold: count, is_
↪orig: bool)

    Current thresholds can be examined using get_conn_bytes_threshold()
    and get_conn_packets_threshold().

    Only one threshold can be set per connection.

  * Add high-level API for packet/bytes thresholding in
    base/protocols/conn/thresholds.bro that holds lists of thresholds
    and raises an event for each threshold exactly once. (Johanna
    Amann)

  * Fix a bug where child packet analyzers of the TCP analyzer
    where not found using FindChild.

  * Update GridFTP analyzer to use connection thresholding instead of
    polling. (Johanna Amann)

2.3-709 | 2015-04-17 12:37:32 -0700

  * Fix addressing the dreaded "internal error: unknown msg type 115
  in Poll()". (Jon Siwek)

    This patch removes the error handling code for overload conditions
    in the main process that could cause trouble down the road. The
    "chunked_io_buffer_soft_cap" script variable can now tune when the
    client process begins shutting down peer connections, and the
    default setting is now double what it used to be. Addresses
    BIT-1376.

2.3-707 | 2015-04-17 10:57:59 -0500

  * Add more info about Broker to NEWS. (Jon Siwek)

2.3-705 | 2015-04-16 08:16:45 -0700

  * Update Mozilla CA list. (Johanna Amann)

  * Update tests to have them keep using older certificates where
    appropiate. (Johanna Amann)

2.3-699 | 2015-04-16 09:51:58 -0500

  * Fix the to_count function to use strtoull versus strtoll.
    (Jon Siwek)

2.3-697 | 2015-04-15 09:51:15 -0700

  * Removing error check verifying that an ASCII writer has been
    properly finished. Instead of aborting, we now just clean up in
    that case and proceed. Addresses BIT-1331. (Robin Sommer)

2.3-696 | 2015-04-14 15:56:36 -0700

  * Update sqlite to 3.8.9

2.3-695 | 2015-04-13 10:34:42 -0500

  * Fix iterator invalidation in broker::Manager dtor. (Jon Siwek)

  * Add paragraph to plugin documentation. (Robin Sommer)

2.3-693 | 2015-04-11 10:56:31 -0700

  * BIT-1367: improve coercion of anonymous records in set constructor.
    (Jon Siwek)

  * Allow to specify ports for sftp log rotator. (Johanna Amann)

2.3-690 | 2015-04-10 21:51:10 -0700

  * Make sure to always delete the remote serializer. Addresses
    BIT-1306 and probably also BIT-1356. (Robin Sommer)

  * Cleaning up --help. -D and -Y/y were still listed, even though
    they had no effect anymore. Removing some dead code along with -D.
    Addresses BIT-1372. (Robin Sommer)

2.3-688 | 2015-04-10 08:10:44 -0700

  * Update SQLite to 3.8.8.3.

2.3-687 | 2015-04-10 07:32:52 -0700

  * Remove stale signature benchmarking code (-L command-line option).
```

```
    (Jon Siwek)

  * BIT-844: fix UDP payload signatures to match packet-wise. (Jon
    Siwek)

2.3-682 | 2015-04-09 12:07:00 -0700

  * Fixing input readers' component type. (Robin Sommer)

  * Tiny spelling correction. (Seth Hall)

2.3-680 | 2015-04-06 16:02:43 -0500

  * BIT-1371: remove CMake version check from binary package scripts.
    (Jon Siwek)

2.3-679 | 2015-04-06 10:16:36 -0500

  * Increase some unit test timeouts. (Jon Siwek)

  * Fix Coverity warning in RDP analyzer. (Jon Siwek)

2.3-676 | 2015-04-02 10:10:39 -0500

  * BIT-1366: improve checksum offloading warning.
    (Frank Meier, Jon Siwek)

2.3-675 | 2015-03-30 17:05:05 -0500

  * Add an RDP analyzer. (Josh Liburdi, Seth Hall, Johanna Amann)

2.3-640 | 2015-03-30 13:51:51 -0500

  * BIT-1359: Limit maximum number of DTLS fragments to 30. (Johanna Amann)

2.3-637 | 2015-03-30 12:02:07 -0500

  * Increase timeout duration in some broker tests. (Jon Siwek)

2.3-636 | 2015-03-30 11:26:32 -0500

  * Updates related to SSH analysis. (Jon Siwek)

    - Some scripts used wrong SSH module/namespace scoping on events.
    - Fix outdated notice documentation related to SSH password guessing.
    - Add a unit test for SSH pasword guessing notice.

2.3-635 | 2015-03-30 11:02:45 -0500

  * Fix outdated documentation unit tests. (Jon Siwek)

2.3-634 | 2015-03-30 10:22:45 -0500

  * Add a canonifier to a unit test's output. (Jon Siwek)

2.3-633 | 2015-03-25 18:32:59 -0700

  * Log::write in signature framework was missing timestamp.
```

```
    (Andrew Benson/Michel Laterman)

2.3-631 | 2015-03-25 11:03:12 -0700

  * New SSH analyzer. (Vlad Grigorescu)

2.3-600 | 2015-03-25 10:23:46 -0700

  * Add defensive checks in code to calculate log rotation intervals.
    (Pete Nelson).

2.3-597 | 2015-03-23 12:50:04 -0700

  * DTLS analyzer. (Johanna Amann)

  * Implement correct parsing of TLS record fragmentation. (Johanna
    Amann)

2.3-582 | 2015-03-23 11:34:25 -0700

  * BIT-1313: In debug builds, "bro -B <x>" now supports "all" and
    "help" for "<x>". "all" enables all debug streams. "help" prints a
    list of available debug streams. (John Donnelly/Robin Sommer).

  * BIT-1324: Allow logging filters to inherit default path from
    stream. This allows the path for the default filter to be
    specified explicitly through $path="..." when creating a stream.
    Adapted the existing Log::create_stream calls to explicitly
    specify a path value. (Jon Siwek)

  * BIT-1199: Change the way the input framework deals with values it
    cannot convert into BroVals, raising error messages instead of
    aborting execution. (Johanna Amann)

  * BIT-788: Use DNS QR field to better identify flow direction. (Jon
    Siwek)

2.3-572 | 2015-03-23 13:04:53 -0500

  * BIT-1226: Fix an example in quickstart docs. (Jon siwek)

2.3-570 | 2015-03-23 09:51:20 -0500

  * Correct a spelling error (Daniel Thayer)

  * Improvement to SSL analyzer failure mode. (Johanna Amann)

2.3-565 | 2015-03-20 16:27:41 -0500

  * BIT-978: Improve documentation of 'for' loop iterator invalidation.
    (Jon Siwek)

2.3-564 | 2015-03-20 11:12:02 -0500

  * BIT-725: Remove "unmatched_HTTP_reply" weird. (Jon Siwek)

2.3-562 | 2015-03-20 10:31:02 -0500
```

```
  * BIT-1207: Add unit test to catch breaking changes to local.bro
    (Jon Siwek)

  * Fix failing sqlite leak test (Johanna Amann)

2.3-560 | 2015-03-19 13:17:39 -0500

  * BIT-1255: Increase default values of
    "tcp_max_above_hole_without_any_acks" and "tcp_max_initial_window"
    from 4096 to 16384 bytes. (Jon Siwek)

2.3-559 | 2015-03-19 12:14:33 -0500

  * BIT-849: turn SMTP reporter warnings into weirds,
    "smtp_nested_mail_transaction" and "smtp_unmatched_end_of_data".
    (Jon Siwek)

2.3-558 | 2015-03-18 22:50:55 -0400

  * DNS: Log the type number for the DNS_RR_unknown_type weird. (Vlad Grigorescu)

2.3-555 | 2015-03-17 15:57:13 -0700

  * Splitting test-all Makefile target into Bro tests and test-aux.
    (Robin Sommer)

2.3-554 | 2015-03-17 15:40:39 -0700

  * Deprecate &rotate_interval, &rotate_size, &encrypt. Addresses
    BIT-1305. (Jon Siwek)

2.3-549 | 2015-03-17 09:12:18 -0700

  * BIT-1077: Fix HTTP::log_server_header_names. Before, it just
    re-logged fields from the client side. (Jon Siwek)

2.3-547 | 2015-03-17 09:07:51 -0700

  * Update certificate validation script to cache valid intermediate
    chains that it encounters on the wire and use those to try to
    validate chains that might be missing intermediate certificates.
    (Johanna Amann)

2.3-541 | 2015-03-13 15:44:08 -0500

  * Make INSTALL a symlink to doc/install/install.rst (Jon siwek)

  * Fix Broxygen coverage. (Jon Siwek)

2.3-539 | 2015-03-13 14:19:27 -0500

  * BIT-1335: Include timestamp in default extracted file names.
    And add a policy script to extract all files. (Jon Siwek)

  * BIT-1311: Identify GRE tunnels as Tunnel::GRE, not Tunnel::IP.
    (Jon Siwek)

  * BIT-1309: Add Connection class getter methods for flow labels.
```

```
    (Jon Siwek)

2.3-536 | 2015-03-12 16:16:24 -0500

  * Fix Broker leak tests. (Jon Siwek)

2.3-534 | 2015-03-12 10:59:49 -0500

  * Update NEWS file. (Jon Siwek)

2.3-533 | 2015-03-12 10:18:53 -0500

  * Give broker python bindings default install path within --prefix.
    (Jon Siwek)

2.3-530 | 2015-03-10 13:22:39 -0500

  * Fix broker data stores in absence of --enable-debug. (Jon Siwek)

2.3-529 | 2015-03-09 13:14:27 -0500

  * Fix format specifier in SSL protocol violation. (Jon Siwek)

2.3-526 | 2015-03-06 12:48:49 -0600

  * Fix build warnings, clarify broker requirements, update submodule.
    (Jon Siwek)

  * Rename comm/ directories to broker/ (Jon Siwek)

  * Rename broker-related namespaces. (Jon Siwek)

  * Improve remote logging via broker by only sending fields w/ &log.
    (Jon Siwek)

  * Disable a stream's remote logging via broker if it fails. (Jon Siwek)

  * Improve some broker communication unit tests. (Jon Siwek)

2.3-518 | 2015-03-04 13:13:50 -0800

  * Add bytes_recvd to stats.log recording the number of bytes
    received, according to packet headers. (Mike Smiley)

2.3-516 | 2015-03-04 12:30:06 -0800

  * Extract most specific Common Name from SSL certificates (Johanna
    Amann)

  * Send CN and SAN fields of SSL certificates to the Intel framework.
    (Johanna Amann)

2.3-511 | 2015-03-02 18:07:17 -0800

  * Changes to plugin meta hooks for function calls. (Gilbert Clark)

        - Add frame argument.
        - Change return value to tuple unambigiously whether hook
          returned a result.

2.3-493 | 2015-03-02 17:17:32 -0800

  * Extend the SSL weak-keys policy file to also alert when
    encountering SSL connections with old versions as well as unsafe
    cipher suites. (Johanna Amann)

  * Make the notice suppression handling of other SSL policy files a
    tad more robust. (Johanna Amann)

2.3-491 | 2015-03-02 17:12:56 -0800

  * Updating docs for recent addition of local_resp. (Robin Sommer)

2.3-489 | 2015-03-02 15:29:30 -0800

  * Integrate Broker, Bro's new communication library. (Jon Siwek)

    See aux/broker/README for more information on Broker, and
    doc/frameworks/comm.rst for the corresponding Bro script API.

    Broker support is by default off for now; it can be enabled at
    configure time with --enable-broker. It requires CAF
    (https://github.com/actor-framework/actor-framework); for now iot
    needs CAF's "develop" branch. Broker also requires a C++11
    compiler.

    Broker will become a mandatory dependency in future Bro versions.

  * Add --enable-c++11 configure flag to compile Bro's source code in
    C++11 mode with a corresponding compiler. (Jon Siwek)

2.3-451 | 2015-02-24 16:37:08 -0800

  * Updating submodule(s).

2.3-448 | 2015-02-23 16:58:10 -0800

  * Updating NEWS. (Robin Sommer)

2.3-447 | 2015-02-23 16:28:30 -0800

  * Fix potential crash in logging framework when deserializing
    WriterInfo from remote. where config is present. Testcase crashes
    on unpatched versions of Bro. (Aaron Eppert)

  * Fix wrong value test in WriterBackend. (Aaron Eppert)

2.3-442 | 2015-02-23 13:29:30 -0800

  * Add a "local_resp" field to conn.log, along the lines of the
    existing "local_orig". (Mike Smiley)

2.3-440 | 2015-02-23 11:39:17 -0600

  * Updating plugin docs to recent changes. (Robin Sommer)
  * Updating plugin tests to recent changes. (Robin Sommer)

  * Making plugin names case-insensitive for some internal comparisions.
    Makes plugin system more tolerant against spelling inconsistencies
    are hard to catch otherwise. (Robin Sommer)

  * Explicitly removing some old scripts on install that have moved
    into plugins to prevent them causing confusion. (Robin Sommer)

  * BIT-1312: Removing setting installation plugin path from
    bro-path-dev.sh.  Also, adding to existing BRO_PLUGIN_PATH rather
    than replacing. (Robin Sommer)

  * Creating the installation directory for plugins at install time.
    (Robin Sommer)

2.3-427 | 2015-02-20 13:49:33 -0800

  * Removing dependency on PCAP_NETMASK_UNKNOWN to compile with
    libpcap < 1.1.1. (Robin Sommer)

2.3-426 | 2015-02-20 12:45:51 -0800

  * Add 'while' statement to Bro language. Really. (Jon Siwek)

2.3-424 | 2015-02-20 12:39:10 -0800

  * Add the ability to remove surrounding braces from the JSON
    formatter. (Seth Hall)

2.3-419 | 2015-02-13 09:10:44 -0600

  * BIT-1011: Update the SOCKS analyzer to support user/pass login.
    (Nicolas Retrain, Seth Hall, Jon Siwek)

    - Add a new field to socks.log: "password".
    - Two new events: "socks_login_userpass_request" and
      "socks_login_userpass_reply".
    - Two new weirds for unsupported SOCKS authentication method or
      version.
    - A new test for authenticated socks traffic.

2.3-416 | 2015-02-12 12:18:42 -0600

  * Submodule update - newest sqlite version (Johanna Amann)

  * Fix use of deprecated gperftools headers. (Jon Siwek)

2.3-413 | 2015-02-08 18:23:05 -0800

  * Fixing analyzer tag types for some Files::* functions. (Robin Sommer)

  * Changing load order for plugin scripts. (Robin Sommer)

2.3-411 | 2015-02-05 10:05:48 -0600

  * Fix file analysis of files with total size below the bof_buffer size
```

```
    never delivering content to stream analyzers. (Seth Hall)

  * Add/fix log fields in x509 diff canonifier. (Jon Siwek)

  * "id" not defined for debug code when using -DPROFILE_BRO_FUNCTIONS
    (Mike Smiley)

2.3-406 | 2015-02-03 17:02:45 -0600

  * Add x509 canonifier to a unit test. (Jon Siwek)

2.3-405 | 2015-02-02 11:14:24 -0600

  * Fix memory leak in new split_string* functions. (Jon Siwek)

2.3-404 | 2015-01-30 14:23:27 -0800

  * Update documentation (broken links, outdated tests). (Jon Siwek)

  * Deprecate split* family of BIFs. (Jon Siwek)

    These functions are now deprecated in favor of alternative versions that
    return a vector of strings rather than a table of strings.

    Deprecated functions:

    - split: use split_string instead.
    - split1: use split_string1 instead.
    - split_all: use split_string_all instead.
    - split_n: use split_string_n instead.
    - cat_string_array: see join_string_vec instead.
    - cat_string_array_n: see join_string_vec instead.
    - join_string_array: see join_string_vec instead.
    - sort_string_array: use sort instead instead.
    - find_ip_addresses: use extract_ip_addresses instead.

    Changed functions:

    - has_valid_octets: uses a string_vec parameter instead of string_array.

    Addresses BIT-924.

  * Add a new attribute: &deprecated. While scripts are parsed, a
    warning is raised for each usage of an identifier marked as
    &deprecated.  This also works for BIFs. Addresses BIT-924,
    BIT-757. (Jon Siwek)

2.3-397 | 2015-01-27 10:13:10 -0600

  * Handle guess_lexer exceptions in pygments reST directive (Jon Siwek)

2.3-396 | 2015-01-23 10:49:15 -0600

  * DNP3: fix reachable assertion and buffer over-read/overflow.
    CVE number pending. (Travis Emmert, Jon Siwek)

  * Update binpac: Fix potential out-of-bounds memory reads in generated
    code. CVE-2014-9586. (John Villamil and Chris Rohlf - Yahoo
    Paranoids, Jon Siwek)

  * Fixing (harmless) Coverity warning. (Robin Sommer)

2.3-392 | 2015-01-15 09:44:15 -0800

  * Small changes to EC curve names in a newer draft. (Johanna Amann)

2.3-390 | 2015-01-14 13:27:34 -0800

  * Updating MySQL analyses. (Vlad Grigorescu)
     - Use a boolean success instead of a result string.
     - Change the affected_rows response detail string to a "rows" count.
     - Fix the state tracking to log incomplete command.

  * Extend DNP3 to support communication over UDP. (Hui Lin)

  * Fix a bug in DNP3 determining the length of an object in some
    cases. (Hui Lin)

2.3-376 | 2015-01-12 09:38:10 -0600

  * Improve documentation for connection_established event. (Jon Siwek)

2.3-375 | 2015-01-08 13:10:09 -0600

  * Increase minimum required CMake version to 2.8. (Jon Siwek)

2.3-374 | 2015-01-07 10:03:17 -0600

  * Improve documentation of the Intelligence Framework. (Daniel Thayer)

2.3-371 | 2015-01-06 09:58:09 -0600

  * Update/improve file mime type identification. (Seth Hall)

     - Change to the default BOF buffer size to 3000 (was 1024).

     - Reorganized MS signatures into a separate file.

     - Remove all of the x-c detections.  Nearly all false positives.

     - Improve TAR detections, removing old, back up TAR detections.

     - Remove one of the x-elc detections that was too loose
       and caused many false positives.

     - Improved lots of the signatures and added new ones. (Seth Hall)

  * Add support for file reassembly in the file analysis framework
    (Seth Hall, Jon Siwek).

     - The reassembly behavior can be modified per-file by enabling or
       disabling the reassembler and/or modifying the size of the
       reassembly buffer.

     - Changed the file extraction analyzer to use stream-wise input to
       avoid issues with the chunk-wise approach not immediately
```

```
      triggering the file_new event due to mime-type detection delay.
      Before, early chunks frequently ended up lost.  Extraction also
      will now explicitly NUL-fill gaps in the file instead of
      implicitly relying on pwrite to do it.

2.3-349 | 2015-01-05 15:21:13 -0600

  * Fix race condition in unified2 file analyzer startup. (Jon siwek)

2.3-348 | 2014-12-31 09:19:34 -0800

  * Changing Makefile's test-all to run test-all for broctl, which now
    executes trace-summary tests as well. (Robin Sommer)

2.3-345 | 2014-12-31 09:06:15 -0800

  * Correct a typo in the Notice framework doc. (Daniel Thayer)

2.3-343 | 2014-12-12 12:43:46 -0800

  * Fix PIA packet replay to deliver copy of IP header. This prevented
    one from writing a packet-wise analyzer that needs access to IP
    headers and can be attached to a connection via signature match.
    Addresses BIT-1298 (Jon Siwek)

2.3-338 | 2014-12-08 13:56:19 -0800

  * Add man page for Bro. (Raúl Benencia)

  * Updating doc baselines. (Robin Sommer)

2.3-334 | 2014-12-03 14:22:07 -0800

  * Fix compound assignment to require proper L-value. Addresses
    BIT-1295. (Jon Siwek)

2.3-332 | 2014-12-03 14:14:11 -0800

  * Make using local IDs in @if directives an error. Addresses
    BIT-1296. (Jon Siwek)

2.3-330 | 2014-12-03 14:10:39 -0800

  * Fix some "make doc" warnings and update some doc tests. (Daniel
    Thayer)

2.3-328 | 2014-12-02 08:13:10 -0500

  * Update windows-version-detection.bro to add support for
    Windows 10. (Michal Purzynski)

2.3-326 | 2014-12-01 12:10:27 -0600

  * BIFScanner: fix invalid characters in generated preprocessor macros.
    (Hilko Bengen)

  * BIT-1294: fix exec.bro from mutating Input::end_of_data event
    parameters. (Johanna Amann)
  * Add/invoke "distclean" for testing directories. (Raúl Benencia)

  * Delete prebuilt python bytecode files from git. (Jon Siwek)

  * Add Windows detection based on CryptoAPI HTTP traffic as a software
    framework policy script. (Vlad Grigorescu)

2.3-316 | 2014-11-25 17:35:06 -0800

  * Make the SSL analyzer skip further processing once encountering
    situations which are very probably non-recoverable. (Johanna
    Amann)

2.3-313 | 2014-11-25 14:27:07 -0800

  * Make SSL v2 protocol tests more strict. In its former state they
    triggered on http traffic over port 443 sometimes. Found by Michał
    Purzyński. (Johanna Amann)

  * Fix X509 analyzer to correctly return ECDSA as the key_type for
    ECDSA certs. Bug found by Michał Purzyński. (Johanna Amann)

2.3-310 | 2014-11-19 10:56:59 -0600

  * Disable verbose bison output. (Jon Siwek)

2.3-309 | 2014-11-18 12:17:53 -0800

  * New decompose_uri() function in base/utils/urls that splits a URI
    into its pieces. (Anthony Kasza).

2.3-305 | 2014-11-18 11:09:04 -0800

  * Improve coercion of &default expressions. Addresses BIT-1288. (Jon
    Siwek)

2.3-303 | 2014-11-18 10:53:04 -0800

  * For DH key exchanges, use p as the parameter for weak key
    exchanges. (Johanna Amann)

2.3-301 | 2014-11-11 13:47:27 -0800

  * Add builtin function enum_to_int() that converts an enum into a
    integer. (Christian Struck)

2.3-297 | 2014-11-11 11:50:47 -0800

  * Removing method from SSL analyzer that's no longer used. (Robin
    Sommer)

2.3-296 | 2014-11-11 11:42:38 -0800

  * A new analyzer parsing the MySQL wire protocol. Activity gets
    logged into mysql.log. Supports protocol versions 9 and 10. (Vlad
    Grigorescu)
```

```
2.3-280 | 2014-11-05 09:46:33 -0500

  * Add Windows detection based on CryptoAPI HTTP traffic as a
    software framework policy script. (Vlad Grigorescu)

2.3-278 | 2014-11-03 18:55:18 -0800

  * Add new curves from draft-ietf-tls-negotiated-ff-dhe to SSL
    analysis. (Johanna Amann)

2.3-274 | 2014-10-31 17:45:25 -0700

  * Adding call to new binpac::init() function. (Robin Sommer)

2.3-272 | 2014-10-31 16:29:42 -0700

  * Fix segfault if when statement's RHS is unitialized. Addresses
    BIT-1176. (Jon Siwek)

  * Fix checking vector indices via "in". Addresses BIT-1280.  (Jon
    Siwek)

2.3-268 | 2014-10-31 12:12:22 -0500

  * BIT-1283: Fix crash when using &encrypt. (Jon Siwek)

2.3-267 | 2014-10-31 10:35:02 -0500

  * BIT-1284: Allow arbitrary when statement timeout expressions
    (Jon Siwek)

2.3-266 | 2014-10-31 09:21:28 -0500

  * BIT-1166: Add configure options to fine tune local state dirs used
    by BroControl. (Jon Siwek)

2.3-264 | 2014-10-30 13:25:57 -0500

  * Fix some minor Coverity Scan complaints. (Jon Siwek)

2.3-263 | 2014-10-28 15:09:10 -0500

  * Fix checking of fwrite return values (Johanna Amann)

2.3-260 | 2014-10-27 12:54:17 -0500

  * Fix errors/warnings when compiling with -std=c++11 (Jon Siwek)

2.3-259 | 2014-10-27 10:04:04 -0500

  * Documentation fixes. (Vicente Jimenez Aguilar and Stefano Azzalini)

2.3-256 | 2014-10-24 15:33:45 -0700

  * Adding missing test baseline. (Robin Sommer)

2.3-255 | 2014-10-24 13:39:44 -0700
```

```
  * Fixing unstable active-http test. (Robin Sommer)

2.3-254 | 2014-10-24 11:40:51 -0700

  * Fix active-http.bro to deal reliably with empty server responses,
    which will now be passed back as empty files. (Christian Struck)

2.3-248 | 2014-10-23 14:20:59 -0700

  * Change order in which a plugin's scripts are loaded at startup.
    (Robin Sommer)

2.3-247 | 2014-10-21 13:42:38 -0700

  * Updates to the SSL analyzer. (Johanna Amann)

        * Mark everything below 2048 bit as a weak key.

        * Fix notice suppression.

        * Add information about server-chosen protocol to ssl.log, if
          provided by application_layer_next_protocol.

        * Add boolean flag to ssl.log signaling if a session was
          resumed. Remove the (usually not really that useful) session
          ID that the client sent.

2.3-240 | 2014-10-21 13:36:33 -0700

  * Fix Coverity-reported issues in DNP3 analyzer. (Seth Hall)

2.3-238 | 2014-10-16 06:51:49 -0700

  * Fix multipart HTTP/MIME entity file analysis so that (1) singular
    CR or LF characters in multipart body content are no longer
    converted to a full CRLF (thus corrupting the file) and (2) it
    also no longer considers the CRLF before the multipart boundary as
    part of the content. Addresses BIT-1235. (Jon Siwek)

2.3-235 | 2014-10-15 10:20:47 -0500

  * BIT-1273: Add error message for bad enum declaration syntax.
    (Jon Siwek)

2.3-234 | 2014-10-14 14:42:09 -0500

  * Documentation fixes. (Steve Smoot)

2.3-233 | 2014-10-09 16:00:27 -0500

  * Change find-bro-logs unit test to follow symlinks. (Jon Siwek)

  * Add error checks and messages to a test script (Daniel Thayer)

2.3-230 | 2014-10-08 08:15:17 -0700

  * Further baseline normalization for plugin test portability. (Robin
    Sommer)
```

```
2.3-229 | 2014-10-07 20:18:11 -0700

  * Fix for test portability. (Robin Sommer)

2.3-228 | 2014-10-07 15:32:37 -0700

  * Include plugin unit tests into the top-level btest configuration. (Robin Sommer)

  * Switching the prefix separator for packet source/dumper plugins
    once more, now to "::". Addresses BIT-1267. (Robin Sommer)

  * Fix for allowing a packet source/dumper plugin to support multiple
    prefixes with a colon. (Robin Sommer)

2.3-225 | 2014-10-07 15:13:35 -0700

  * Updating plugin documentation. (Robin Sommer)

2.3-224 | 2014-10-07 14:32:17 -0700

  * Improved the log file reference documentation. (Jeannette Dopheide
    and Daniel Thayer)

  * Improves shockwave flash file signatures. (Seth Hall)

     - This moves the signatures out of the libmagic imported signatures
       and into our own general.sig.

     - Expand the detection to LZMA compressed flash files.

  * Add new script language reference documentation on operators,
    statements, and directives.  Also improved the documentation on
    types and attributes by splitting them into two docs, and
    providing more examples and adding a chart on the top of each page
    with links to each type and attribute for easier access to the
    information. (Daniel Thayer)

  * Split the types and attributes reference doc into two docs.
    (Daniel Thayer)

2.3-208 | 2014-10-03 09:38:52 -0500

  * BIT-1268: Fix uninitialized router_list argument in
    dhcp_offer/dhcp_ack. (Jon Siwek)

2.3-207 | 2014-10-02 16:39:17 -0700

  * Updating plugin docs. (Robin Sommer)

  * Fix packet sources being treated as idle when a packet is
    available. Addresses BIT-1266. (Jon Siwek)

  * Fix regression causing the main loop to spin more frequently.
    Addresses BIT-1266. (Jon Siwek)

2.3-203 | 2014-09-29 20:06:54 -0700
```

```
  * Fix to use length parameter in DNP3 time conversion correctly now.
    (Robin Sommer)

2.3-202 | 2014-09-29 17:05:18 -0700

  * New SSL extension type from IANA and a few other SSL const
    changes. (Johanna Amann)

  * Make unexpected pipe errors fatal as precaution. Addresses
    BIT-1260. (Jon Siwek)

  * Adding a function for DNP3 to translate the timestamp format. (Hui
    Lin)

2.3-197 | 2014-09-29 10:42:01 -0500

  * Fix possible seg fault in TCP reassembler. (Jon Siwek)

2.3-196 | 2014-09-25 17:53:27 -0700

  * Changing prefix for packet sources/dumper from ':' to '%'.
    Addresses BIT-1249. (Robin Sommer)

  * Remove timeouts from remote communication loop. The select() now
    blocks until there's work to do instead of relying on a small
    timeout value which can cause unproductive use of cpu cycles. (Jon
    Siwek)

  * Improve error message when failing to activate a plugin. Also fix
    a unit test helper script that checks plugin availability. (Jon
    Siwek)

2.3-183 | 2014-09-24 10:08:04 -0500

  * Add a "node" field to Intel::Seen struture and intel.log to
    indicate which node discovered a hit on an intel item. (Seth Hall)

  * BIT-1261: Fixes to plugin quick start doc. (Jon Siwek)

2.3-180 | 2014-09-22 12:52:41 -0500

  * BIT-1259: Fix issue w/ duplicate TCP reassembly deliveries.
    (Jon Siwek)

2.3-178 | 2014-09-18 14:29:46 -0500

  * BIT-1256: Fix file analysis events from coming after bro_done().
    (Jon Siwek)

2.3-177 | 2014-09-17 09:41:27 -0500

  * Documentation fixes. (Chris Mavrakis)

2.3-174 | 2014-09-17 09:37:09 -0500

  * Fixed some "make doc" warnings caused by reST formatting
    (Daniel Thayer).
```

```
2.3-172 | 2014-09-15 13:38:52 -0500

  * Remove unneeded allocations for HTTP messages. (Jon Siwek)

2.3-171 | 2014-09-15 11:14:57 -0500

  * Fix a compile error on systems without pcap-int.h. (Jon Siwek)

2.3-170 | 2014-09-12 19:28:01 -0700

  * Fix incorrect data delivery skips after gap in HTTP Content-Range.
    Addresses BIT-1247. (Jon Siwek)

  * Fix file analysis placement of data after gap in HTTP
    Content-Range. Addresses BIT-1248. (Jon Siwek)

  * Fix issue w/ TCP reassembler not delivering some segments.
    Addresses BIT-1246. (Jon Siwek)

  * Fix MIME entity file data/gap ordering and raise http_entity_data
    in line with data arrival. Addresses BIT-1240. (Jon Siwek)

  * Implement file ID caching for MIME_Mail. (Jon Siwek)

  * Fix a compile error. (Jon Siwek)

2.3-161 | 2014-09-09 12:35:38 -0500

  * Bugfixes and test updates/additions. (Robin Sommer)

  * Interface tweaks and docs for PktSrc/PktDumper. (Robin Sommer)

  * Moving PCAP-related bifs to iosource/pcap.bif. (Robin Sommer)

  * Moving some of the BPF filtering code into base class.
    This will allow packet sources that don't support BPF natively to
    emulate the filtering via libpcap. (Robin Sommer)

  * Removing FlowSrc. (Robin Sommer)

  * Removing remaining pieces of the 2ndary path, and left-over
    files of packet sorter. (Robin Sommer)

  * A bunch of infrastructure work to move IOSource, IOSourceRegistry
    (now iosource::Manager) and PktSrc/PktDumper code into iosource/,
    and over to a plugin structure. (Robin Sommer)

2.3-137 | 2014-09-08 19:01:13 -0500

  * Fix Broxygen's rendering of opaque types. (Jon Siwek)

2.3-136 | 2014-09-07 20:50:46 -0700

  * Change more http links to https. (Johanna Amann)

2.3-134 | 2014-09-04 16:16:36 -0700

  * Fixed a number of issues with OCSP reply validation. Addresses
```

```
    BIT-1212. (Johanna Amann)

  * Fix null pointer dereference in OCSP verification code in case no
    certificate is sent as part as the ocsp reply. Addresses BIT-1212.
    (Johanna Amann)

2.3-131 | 2014-09-04 16:10:32 -0700

  * Make links in documentation templates protocol relative. (Johanna
    Amann)

2.3-129 | 2014-09-02 17:21:21 -0700

  * Simplify a conditional with equivalent branches. (Jon Siwek)

  * Change EDNS parsing code to use rdlength more cautiously. (Jon
    Siwek)

  * Fix a memory leak when bind() fails due to EADDRINUSE. (Jon Siwek)

  * Fix possible buffer over-read in DNS TSIG parsing. (Jon Siwek)

2.3-124 | 2014-08-26 09:24:19 -0500

  * Better documentation for sub_bytes (Jimmy Jones)

  * BIT-1234: Fix build on systems that already have ntohll/htonll
    (Jon Siwek)

2.3-121 | 2014-08-22 15:22:15 -0700

  * Detect functions that try to bind variables from an outer scope
    and raise an error saying that's not supported. Addresses
    BIT-1233. (Jon Siwek)

2.3-116 | 2014-08-21 16:04:13 -0500

  * Adding plugin testing to Makefile's test-all. (Robin Sommer)

  * Converting log writers and input readers to plugins.
    DataSeries and ElasticSearch plugins have moved to the new
    bro-plugins repository, which is now a git submodule in the
    aux/plugins directory. (Robin Sommer)

2.3-98 | 2014-08-19 11:03:46 -0500

  * Silence some doc-related warnings when using `bro -e`.
    Closes BIT-1232. (Jon Siwek)

  * Fix possible null ptr derefs reported by Coverity. (Jon Siwek)

2.3-96 | 2014-08-01 14:35:01 -0700

  * Small change to DHCP documentation. In server->client messages the
    host name may differ from the one requested by the client.
    (Johanna Amann)

  * Split DHCP log writing from record creation. This allows users to
```

```
    customize dhcp.log by changing the record in their own dhcp_ack
    event. (Johanna Amann)

  * Update PATH so that documentation btests can find bro-cut. (Daniel
    Thayer)

  * Remove gawk from list of optional packages in documentation.
    (Daniel Thayer)

  * Fix for redefining built-in constants. (Robin Sommer)

2.3-86 | 2014-07-31 14:19:58 -0700

  * Fix for redefining built-in constants. (Robin Sommer)

  * Adding missing check that a plugin's API version matches what Bro
    defines. (Robin Sommer)

  * Adding NEWS entry for plugins. (Robin Sommer)

2.3-83 | 2014-07-30 16:26:11 -0500

  * Minor adjustments to plugin code/docs. (Jon Siwek)

  * Dynamic plugin support. (Rpbin Sommer)

    Bro now supports extending core functionality, like protocol and
    file analysis, dynamically with external plugins in the form of
    shared libraries. See doc/devel/plugins.rst for an overview of the
    main functionality. Changes coming with this:

        - Replacing the old Plugin macro magic with a new API.

        - The plugin API changed to generally use std::strings instead
          of const char*.

        - There are a number of invocations of PLUGIN_HOOK_
          {VOID,WITH_RESULT} across the code base, which allow plugins
          to hook into the processing at those locations.

        - A few new accessor methods to various classes to allow
          plugins to get to that information.

        - network_time cannot be just assigned to anymore, there's now
          function net_update_time() for that.

        - Redoing how builtin variables are initialized, so that it
          works for plugins as well. No more init_net_var(), but
          instead bifcl-generated code that registers them.

        - Various changes for adjusting to the now dynamic generation
          of analyzer instances.

    - same_type() gets an optional extra argument allowing record type
      comparision to ignore if field names don't match. (Robin Sommer)

    - Further unify file analysis API with the protocol analyzer API
      (assigning IDs to analyzers; adding Init()/Done() methods;
      adding subtypes). (Robin Sommer)

    - A new command line option -Q that prints some basic execution
      time stats. (Robin Sommer)

    - Add support to the file analysis for activating analyzers by
      MIME type. (Robin Sommer)

          - File::register_for_mime_type(tag: Analyzer::Tag, mt:
            string): Associates a file analyzer with a MIME type.

          - File::add_analyzers_for_mime_type(f: fa_file, mtype:
            string): Activates all analyzers registered for a MIME
            type for the file.

          - The default file_new() handler calls
            File::add_analyzers_for_mime_type() with the file's MIME
            type.

2.3-20 | 2014-07-22 17:41:02 -0700

  * Updating submodule(s).

2.3-19 | 2014-07-22 17:29:19 -0700

  * Implement bytestring_to_coils() in Modbus analyzer so that coils
    gets passed to the corresponding events. (Hui Lin)

  * Add length field to ModbusHeaders. (Hui Lin)

2.3-12 | 2014-07-10 19:17:37 -0500

  * Include yield of vectors in Broxygen's type descriptions.
    Addresses BIT-1217. (Jon Siwek)

2.3-11 | 2014-07-10 14:49:27 -0700

  * Fixing DataSeries output. It was using a now illegal value as its
    default compression level. (Robin Sommer)

2.3-7 | 2014-06-26 17:35:18 -0700

  * Extending "make test-all" to include aux/bro-aux. (Robin Sommer)

2.3-6 | 2014-06-26 17:24:10 -0700

  * DataSeries compilation issue fixed. (mlaterman)

  * Fix a reference counting bug in ListVal ctor. (Jon Siwek)

2.3-3 | 2014-06-26 15:41:04 -0500

  * Support tilde expansion when Bro tries to find its own path. (Jon
    Siwek)

2.3-2 | 2014-06-23 16:54:15 -0500

  * Remove references to line numbers in tutorial text. (Daniel Thayer)
```

```
2.3 | 2014-06-16 09:48:25 -0500

  * Release 2.3.

2.3-beta-33 | 2014-06-12 11:59:28 -0500

  * Documentation improvements/fixes. (Daniel Thayer)

2.3-beta-24 | 2014-06-11 15:35:31 -0500

  * Fix SMTP state tracking when server response is missing.
    (Robin Sommer)

2.3-beta-22 | 2014-06-11 12:31:38 -0500

  * Fix doc/test that broke due to a Bro script change. (Jon Siwek)

  * Remove unused --with-libmagic configure option. (Jon Siwek)

2.3-beta-20 | 2014-06-10 18:16:51 -0700

  * Fix use-after-free in some cases of reassigning a table index.
    Addresses BIT-1202. (Jon Siwek)

2.3-beta-18 | 2014-06-06 13:11:50 -0700

  * Add two more SSL events, one triggered for each handshake message
    and one triggered for the tls change cipherspec message. (Johanna
    Amann)

  * Small SSL bug fix. In case SSL::disable_analyzer_after_detection
    was set to false, the ssl_established event would fire after each
    data packet once the session is established. (Johanna Amann)

2.3-beta-16 | 2014-06-06 13:05:44 -0700

  * Re-activate notice suppression for expiring certificates.
    (Johanna Amann)

2.3-beta-14 | 2014-06-05 14:43:33 -0700

  * Add new TLS extension type numbers from IANA (Johanna Amann)

  * Switch to double hashing for Bloomfilters for better performance.
    (Matthias Vallentin)

  * Bugfix to use full digest length instead of just one byte for
    Bloomfilter's universal hash function. Addresses BIT-1140.
    (Matthias Vallentin)

  * Make buffer for X509 certificate subjects larger. Addresses
    BIT-1195 (Johanna Amann)

2.3-beta-5 | 2014-05-29 15:34:42 -0500

  * Fix misc/load-balancing.bro's reference to
    PacketFilter::sampling_filter (Jon Siwek)
```

```
2.3-beta-4 | 2014-05-28 14:55:24 -0500

  * Fix potential mem leak in remote function/event unserialization.
    (Jon Siwek)

  * Fix reference counting bug in table coercion expressions (Jon Siwek)

  * Fix an "unused value" warning. (Jon Siwek)

  * Remove a duplicate unit test baseline dir. (Jon Siwek)

2.3-beta | 2014-05-19 16:36:50 -0500

  * Release 2.3-beta

  * Clean up OpenSSL data structures on exit. (Johanna Amann)

  * Fixes for OCSP & x509 analysis memory leak issues. (Johanna Amann)

  * Remove remaining references to BROMAGIC (Daniel Thayer)

  * Fix typos and formatting in event and BiF documentation (Daniel Thayer)

  * Update intel framework plugin for ssl server_name extension API
    changes. (Johanna Amann, Justin Azoff)

  * Fix expression errors in SSL/x509 scripts when unparseable data
    is in certificate chain. (Johanna Amann)

2.2-478 | 2014-05-19 15:31:33 -0500

  * Change record ctors to only allow record-field-assignment
    expressions. (Jon Siwek)

2.2-477 | 2014-05-19 14:13:00 -0500

  * Fix X509::Result record's "result" field to be set internally as type int instead␣
↪of type count. (Johanna Amann)

  * Fix a couple of doc build warnings (Daniel Thayer)

2.2-470 | 2014-05-16 15:16:32 -0700

  * Add a new section "Cluster Configuration" to the docs that is
    intended as a how-to for configuring a Bro cluster.  Most of this
    content was moved here from the BroControl doc (which is now
    intended as more of a reference guide for more experienced users)
    and the load balancing FAQ on the website. (Daniel Thayer)

  * Update some doc tests and line numbers (Daniel Thayer)

2.2-457 | 2014-05-16 14:38:31 -0700

  * New script policy/protocols/ssl/validate-ocsp.bro that adds OSCP
    validation to ssl.log. The work is done by a new bif
    x509_ocsp_verify(). (Johanna Amann)
  * STARTTLS support for POP3 and SMTP. The SSL analyzer takes over
    when seen. smtp.log now logs when a connection switches to SSL.
    (Johanna Amann)

  * Replace errors when parsing x509 certs with weirds. (Johanna
    Amann)

  * Improved Heartbleed attack/scan detection. (Johanna Amann)

  * Let TLS analyzer fail better when no longer in sync with the data
    stream. (Johanna Amann)

2.2-444 | 2014-05-16 14:10:32 -0500

  * Disable all default AppStat plugins except facebook. (Jon Siwek)

  * Update for the active http test to force it to use ipv4. (Seth Hall)

2.2-441 | 2014-05-15 11:29:56 -0700

  * A new RADIUS analyzer. (Vlad Grigorescu)

    It produces a radius.log and generates two events:

        event radius_message(c: connection, result: RADIUS::Message);
        event radius_attribute(c: connection, attr_type: count, value: string);

2.2-427 | 2014-05-15 13:37:23 -0400

  * Fix dynamic SumStats update on clusters (Johanna Amann)

2.2-425 | 2014-05-08 16:34:44 -0700

  * Fix reassembly of data w/ sizes beyond 32-bit capacities. (Jon Siwek)

    Reassembly code (e.g. for TCP) now uses int64/uint64 (signedness
    is situational) data types in place of int types in order to
    support delivering data to analyzers that pass 2GB thresholds.
    There's also changes in logic that accompany the change in data
    types, e.g. to fix TCP sequence space arithmetic inconsistencies.

    Another significant change is in the Analyzer API: the *Packet and
    *Undelivered methods now use a uint64 in place of an int for the
    relative sequence space offset parameter.

    Addresses BIT-348.

  * Fixing compiler warnings. (Robin Sommer)

  * Update SNMP analyzer's DeliverPacket method signature. (Jon Siwek)

2.2-417 | 2014-05-07 10:59:22 -0500

  * Change handling of atypical OpenSSL error case in x509 verification. (Jon Siwek)

  * Fix memory leaks in X509 certificate parsing/verification. (Jon Siwek)

  * Fix new []/delete mismatch in input::reader::Raw::DoClose(). (Jon Siwek)
  * Fix buffer over-reads in file_analysis::Manager::Terminate() (Jon Siwek)

  * Fix buffer overlows in IP address masking logic. (Jon Siwek)

    That could occur either in taking a zero-length mask on an IPv6 address
    (e.g. [fe80::]/0) or a reverse mask of length 128 on any address (e.g.
    via the remask_addr BuiltIn Function).

  * Fix new []/delete mismatch in ~Base64Converter. (Jon Siwek)

2.2-410 | 2014-05-02 12:49:53 -0500

  * Replace an unneeded OPENSSL_malloc call. (Jon Siwek)

2.2-409 | 2014-05-02 12:09:06 -0500

  * Clean up and documentation for base SNMP script. (Jon Siwek)

  * Update base SNMP script to now produce a snmp.log. (Seth Hall)

  * Add DH support to SSL analyzer.  When using DHE or DH-Anon, sever
    key parameters are now available in scriptland.  Also add script to
    alert on weak certificate keys or weak dh-params. (Johanna Amann)

  * Add a few more ciphers Bro did not know at all so far. (Johanna Amann)

  * Log chosen curve when using ec cipher suite in TLS. (Johanna Amann)

2.2-397 | 2014-05-01 20:29:20 -0700

  * Fix reference counting for lookup_ID() usages. (Jon Siwek)

2.2-395 | 2014-05-01 20:25:48 -0700

  * Fix missing "irc-dcc-data" service field from IRC DCC connections.
    (Jon Siwek)

  * Correct a notice for heartbleed. The notice is thrown correctly,
    just the message conteined wrong values. (Johanna Amann)

  * Improve/standardize some malloc/realloc return value checks. (Jon
    Siwek)

  * Improve file analysis manager shutdown/cleanup. (Jon Siwek)

2.2-388 | 2014-04-24 18:38:07 -0700

  * Fix decoding of MIME quoted-printable. (Mareq)

2.2-386 | 2014-04-24 18:22:29 -0700

  * Do a Intel::ADDR lookup for host field if we find an IP address
    there. (jshlbrd)

2.2-381 | 2014-04-24 17:08:45 -0700

  * Add Java version to software framework. (Brian Little)
```

```
2.2-379 | 2014-04-24 17:06:21 -0700

  * Remove unused Val::attribs member. (Jon Siwek)

2.2-377 | 2014-04-24 16:57:54 -0700

  * A larger set of SSL improvements and extensions. Addresses
    BIT-1178. (Johanna Amann)

        - Fixes TLS protocol version detection. It also should
          bail-out correctly on non-tls-connections now

        - Adds support for a few TLS extensions, including
          server_name, alpn, and ec-curves.

        - Adds support for the heartbeat events.

        - Add Heartbleed detector script.

        - Adds basic support for OCSP stapling.

  * Fix parsing of DNS TXT RRs w/ multiple character-strings.
    Addresses BIT-1156. (Jon Siwek)

2.2-353 | 2014-04-24 16:12:30 -0700

  * Adapt HTTP partial content to cache file analysis IDs. (Jon Siwek)

  * Adapt SSL analyzer to generate file analysis handles itself. (Jon
    Siwek)

  * Adapt more of HTTP analyzer to use cached file analysis IDs. (Jon
    Siwek)

  * Adapt IRC/FTP analyzers to cache file analysis IDs. (Jon Siwek)

  * Refactor regex/signature AcceptingSet data structure and usages.
    (Jon Siwek)

  * Enforce data size limit when checking files for MIME matches. (Jon
    Siwek)

  * Refactor file analysis file ID lookup. (Jon Siwek)

2.2-344 | 2014-04-22 20:13:30 -0700

  * Refactor various hex escaping code. (Jon Siwek)

2.2-341 | 2014-04-17 18:01:41 -0500

  * Fix duplicate DNS log entries. (Robin Sommer)

2.2-341 | 2014-04-17 18:01:01 -0500

  * Refactor initialization of ASCII log writer options. (Jon Siwek)

  * Fix a memory leak in ASCII log writer. (Jon Siwek)
```

```
2.2-338 | 2014-04-17 17:48:17 -0500

  * Disable input/logging threads setting their names on every
    heartbeat. (Jon Siwek)

  * Fix bug when clearing Bloom filter contents. Reported by
    @colonelxc. (Matthias Vallentin)

2.2-335 | 2014-04-10 15:04:57 -0700

  * Small logic fix for main SSL script. (Johanna Amann)

  * Update DPD signatures for detecting TLS 1.2. (Johanna Amann)

  * Remove unused data member of SMTP_Analyzer to silence a Coverity
    warning. (Jon Siwek)

  * Fix missing @load dependencies in some scripts. Also update the
    unit test which is supposed to catch such errors. (Jon Siwek)

2.2-326 | 2014-04-08 15:21:51 -0700

  * Add SNMP datagram parsing support.This supports parsing of SNMPv1
    (RFC 1157), SNMPv2 (RFC 1901/3416), and SNMPv2 (RFC 3412).  An
    event is raised for each SNMP PDU type, though there's not
    currently any event handlers for them and not a default snmp.log
    either.  However, simple presence of SNMP is currently visible now
    in conn.log service field and known_services.log. (Jon Siwek)

2.2-319 | 2014-04-03 15:53:25 -0700

  * Improve __load__.bro creation for .bif.bro stubs. (Jon Siwek)

2.2-317 | 2014-04-03 10:51:31 -0400

  * Add a uid field to the signatures.log.  Addresses BIT-1171
    (Anthony Verez)

2.2-315 | 2014-04-01 16:50:01 -0700

  * Change logging's "#types" description of sets to "set". Addresses
    BIT-1163 (Johanna Amann)

2.2-313 | 2014-04-01 16:40:19 -0700

  * Fix a couple nits reported by Coverity.(Jon Siwek)

  * Fix potential memory leak in IP frag reassembly reported by
    Coverity. (Jon Siwek)

2.2-310 | 2014-03-31 18:52:22 -0700

  * Fix memory leak and unchecked dynamic cast reported by Coverity.
    (Jon Siwek)

  * Fix potential memory leak in x509 parser reported by Coverity.
    (Johanna Amann)
```

```
2.2-304 | 2014-03-30 23:05:54 +0200

  * Replace libmagic w/ Bro signatures for file MIME type
    identification. Addresses BIT-1143. (Jon Siwek)

    Includes:

    - libmagic is no longer used at all.  All MIME type detection is
      done through new Bro signatures, and there's no longer a means
      to get verbose file type descriptions. The majority of the
      default file magic signatures are derived from the default magic
      database of libmagic ~5.17.

    - File magic signatures consist of two new constructs in the
      signature rule parsing grammar: "file-magic" gives a regular
      expression to match against, and "file-mime" gives the MIME type
      string of content that matches the magic and an optional strength
      value for the match.

    - Modified signature/rule syntax for identifiers: they can no
      longer start with a '-', which made for ambiguous syntax when
      doing negative strength values in "file-mime".  Also brought
      syntax for Bro script identifiers in line with reality (they
      can't start with numbers or include '-' at all).

    - A new built-in function, "file_magic", can be used to get all
      file magic matches and their corresponding strength against a
      given chunk of data.

    - The second parameter of the "identify_data" built-in function
      can no longer be used to get verbose file type descriptions,
      though it can still be used to get the strongest matching file
      magic signature.

    - The "file_transferred" event's "descr" parameter no longer
      contains verbose file type descriptions.

    - The BROMAGIC environment variable no longer changes any behavior
      in Bro as magic databases are no longer used/installed.

    - Removed "binary" and "octet-stream" mime type detections. They
      don' provide any more information than an uninitialized
      mime_type field which implicitly means no magic signature
      matches and so the media type is unknown to Bro.

    - The "fa_file" record now contains a "mime_types" field that
      contains all magic signatures that matched the file content
      (where the "mime_type" field is just a shortcut for the
      strongest match).

    - Reverted back to minimum requirement of CMake 2.6.3 from 2.8.0.

  * The logic for adding file ids to {orig,resp}_fuids fields of the
    http.log incorrectly depended on the state of
    {orig,resp}_mime_types fields, so sometimes not all file ids
    associated w/ the session were logged. (Jon Siwek)
  * Fix MHR script's use of fa_file$mime_type before checking if it's
    initialized. (Jon Siwek)

2.2-294 | 2014-03-30 22:08:25 +0200

  * Rework and move X509 certificate processing from the SSL protocol
    analyzer to a dedicated file analyzer. This will allow us to
    examine X509 certificates from sources other than SSL in the
    future. Furthermore, Bro now parses more fields and extensions
    from the certificates (e.g. elliptic curve information, subject
    alternative names, basic constraints). Certificate validation also
    was improved, should be easier to use and exposes information like
    the full verified certificate chain. (Johanna Amann)

    This update changes the format of ssl.log and adds a new x509.log
    with certificate information. Furthermore all x509 events and
    handling functions have changed.

2.2-271 | 2014-03-30 20:25:17 +0200

  * Add unit tests covering vector/set/table ctors/inits. (Jon Siwek)

  * Fix parsing of "local" named table constructors. (Jon Siwek)

  * Improve type checking of records. Addresses BIT-1159. (Jon Siwek)

2.2-267 | 2014-03-30 20:21:43 +0200

  * Improve documentation of Bro clusters. Addresses BIT-1160.
    (Daniel Thayer)

2.2-263 | 2014-03-30 20:19:05 +0200

  * Don't include locations into serialization when cloning values.
    (Robin Sommer)

2.2-262 | 2014-03-30 20:12:47 +0200

  * Refactor SerializationFormat::EndWrite and ChunkedIO::Chunk memory
    management. (Jon Siwek)

  * Improve SerializationFormat's write buffer growth strategy. (Jon
    Siwek)

  * Add --parse-only option to exit after parsing scripts. May be
    useful for syntax-checking tools. (Jon Siwek)

2.2-256 | 2014-03-30 19:57:28 +0200

  * For the summary statistics framewirk, change all &create_expire
    attributes to &read_expire in the cluster part. (Johanna Amann)

2.2-254 | 2014-03-30 19:55:22 +0200

  * Update instructions on how to build Bro docs. (Daniel Thayer)

2.2-251 | 2014-03-28 08:37:37 -0400
```

```
  * Quick fix to the ElasticSearch writer. (Seth Hall)

2.2-250 | 2014-03-19 17:20:55 -0400

  * Improve performance of MHR script by reducing cloned Vals in
    a "when" scope. (Jon Siwek)

2.2-248 | 2014-03-19 14:47:40 -0400

  * Make SumStats work incrementally and non-blocking in non-cluster
    mode, but force it to operate by blocking if Bro is shutting
    down. (Seth Hall)

2.2-244 | 2014-03-17 08:24:17 -0700

  * Fix compile errror on FreeBSD caused by wrong include file order.
    (Johanna Amann)

2.2-240 | 2014-03-14 10:23:54 -0700

  * Derive results of DNS lookups from from input when in BRO_DNS_FAKE
    mode. Addresses BIT-1134. (Jon Siwek)

  * Fixing a few cases of undefined behaviour introduced by recent
    formatter work.

  * Fixing compiler error. (Robin Sommer)

  * Fixing (very unlikely) double delete in HTTP analyzer when
    decapsulating CONNECTs. (Robin Sommer)

2.2-235 | 2014-03-13 16:21:19 -0700

  * The Ascii writer has a new option LogAscii::use_json for writing
        out logs as JSON. (Seth Hall)

  * Ascii input reader now supports all config options as per-input
    stream "config" values. (Seth Hall)

  * Refactored formatters and updated the the writers a bit. (Seth
        Hall)

2.2-229 | 2014-03-13 14:58:30 -0700

  * Refactoring analyzer manager code to reuse
    ApplyScheduledAnalyzers(). (Robin Sommer)

2.2-228 | 2014-03-13 14:25:53 -0700

  * Teach async DNS lookup builtin-functions about BRO_DNS_FAKE.
    Addresses BIT-1134. (Jon Siwek)

  * Enable fake DNS mode for test suites.

  * Improve analysis of TCP SYN/SYN-ACK reversal situations. (Jon
    Siwek)

    - Since it's just the handshake packets out of order, they're no
```

```
      longer treated as partial connections, which some protocol analyzers
      immediately refuse to look at.

    - The TCP_Reassembler "is_orig" state failed to change, which led to
      protocol analyzers sometimes using the wrong value for that.

    - Add a unit test which exercises the Connection::FlipRoles() code
      path (i.e. the SYN/SYN-ACK reversal situation).

    Addresses BIT-1148.

  * Fix bug in Connection::FlipRoles. It didn't swap address values
    right and also didn't consider that analyzers might be scheduled
    for the new connection tuple. Reported by Kevin McMahon. Addresses
    BIT-1148. (Jon Siwek)

2.2-221 | 2014-03-12 17:23:18 -0700

  * Teach configure script --enable-jemalloc, --with-jemalloc.
    Addresses BIT-1128. (Jon Siwek)

2.2-218 | 2014-03-12 17:19:45 -0700

  * Improve DBG_LOG macro (perf. improvement for --enable-debug mode).
    (Jon Siwek)

  * Silences some documentation warnings from Sphinx. (Jon Siwek)

2.2-215 | 2014-03-10 11:10:15 -0700

  * Fix non-deterministic logging of unmatched DNS msgs. Addresses
    BIT-1153 (Jon Siwek)

2.2-213 | 2014-03-09 08:57:37 -0700

  * No longer accidentally attempting to parse NBSTAT RRs as SRV RRs
    in DNS analyzer. (Seth Hall)

  * Fix DNS SRV responses and a small issue with NBNS queries and
    label length. (Seth Hall)

     - DNS SRV responses never had the code written to actually
       generate the dns_SRV_reply event.  Adding this required
       extending the event a bit to add extra information.  SRV responses
       now appear in the dns.log file correctly.

     - Fixed an issue where some Microsoft NetBIOS Name Service lookups
       would exceed the max label length for DNS and cause an incorrect
       "DNS_label_too_long" weird.

2.2-210 | 2014-03-06 22:52:36 -0500

  * Improve SSL logging so that connections are logged even when the
    ssl_established event is not generated as well as other small SSL
    fixes. (Johanna Amann)

2.2-206 | 2014-03-03 16:52:28 -0800
```

```
  * HTTP CONNECT proxy support. The HTTP analyzer now supports
    handling HTTP CONNECT proxies. (Seth Hall)

  * Expanding the HTTP methods used in the DPD signature to detect
    HTTP traffic. (Seth Hall)

  * Fixing removal of support analyzers. (Robin Sommer)

2.2-199 | 2014-03-03 16:34:20 -0800

  * Allow iterating over bif functions with result type vector of any.
    This changes the internal type that is used to signal that a
    vector is unspecified from any to void. Addresses BIT-1144
    (Johanna Amann)

2.2-197 | 2014-02-28 15:36:58 -0800

  * Remove test code. (Robin Sommer)

2.2-194 | 2014-02-28 14:50:53 -0800

  * Remove packet sorter. Addresses BIT-700. (Johanna Amann)

2.2-192 | 2014-02-28 09:46:43 -0800

  * Update Mozilla root bundle. (Johanna Amann)

2.2-190 | 2014-02-27 07:34:44 -0800

  * Adjust timings of a few leak tests. (Johanna Amann)

2.2-187 | 2014-02-25 07:24:42 -0800

  * More Google TLS extensions that are being actively used. Johanna(
    Amann)

  * Remove unused, and potentially unsafe, function
    ListVal::IncludedInString. (Johanna Amann)

2.2-184 | 2014-02-24 07:28:18 -0800

  * New TLS constants from
    https://tools.ietf.org/html/draft-bmoeller-tls-downgrade-scsv-01.
    (Johanna Amann)

2.2-180 | 2014-02-20 17:29:14 -0800

  * New SSL alert descriptions from
    https://tools.ietf.org/html/draft-ietf-tls-applayerprotoneg-04.
    (Johanna Amann)

  * Update SQLite. (Johanna Amann)

2.2-177 | 2014-02-20 17:27:46 -0800

  * Update to libmagic version 5.17. Addresses BIT-1136. (Jon Siwek)

2.2-174 | 2014-02-14 12:07:04 -0800
```

```
  * Support for MPLS over VLAN. (Chris Kanich)

2.2-173 | 2014-02-14 10:50:15 -0800

  * Fix misidentification of SOCKS traffic that in particiular seemed
    to happen a lot with DCE/RPC traffic. (Vlad Grigorescu)

2.2-170 | 2014-02-13 16:42:07 -0800

  * Refactor DNS script's state management to improve performance.
    (Jon Siwek)

  * Revert "Expanding the HTTP methods used in the signature to detect
    HTTP traffic." (Robin Sommer)

2.2-167 | 2014-02-12 20:17:39 -0800

  * Increase timeouts of some unit tests. (Jon Siwek)

  * Fix memory leak in modbus analyzer. Would happen if there's a
    'modbus_read_fifo_queue_response' event handler. (Jon Siwek)

  * Add channel_id TLS extension number. This number is not IANA
    defined, but we see it being actively used. (Johanna Amann)

  * Test baseline updates for DNS change. (Robin Sommer)

2.2-158 | 2014-02-09 23:45:39 -0500

  * Change dns.log to include only standard DNS queries. (Jon Siwek)

  * Improve DNS analysis. (Jon Siwek)

    - Fix parsing of empty question sections (when QDCOUNT == 0).  In this
      case, the DNS parser would extract two 2-byte fields for use in either
      "dns_query_reply" or "dns_rejected" events (dependent on value of
      RCODE) as qclass and qtype parameters.  This is not correct, because
      such fields don't actually exist in the DNS message format when
      QDCOUNT is 0.  As a result, these events are no longer raised when
      there's an empty question section.  Scripts that depends on checking
      for an empty question section can do that in the "dns_message" event.

    - Add a new "dns_unknown_reply" event, for when Bro does not know how
      to fully parse a particular resource record type.  This helps fix a
      problem in the default DNS scripts where the logic to complete
      request-reply pair matching doesn't work because it's waiting on more
      RR events to complete the reply.  i.e. it expects ANCOUNT number of
      dns_*_reply events and will wait until it gets that many before
      completing a request-reply pair and logging it to dns.log.  This could
      cause bogus replies to match a previous request if they happen to
      share a DNS transaction ID. (Jon Siwek)

    - The previous method of matching queries with replies was still
      unreliable in cases where the reply contains no answers.  The new code
      also takes extra measures to avoid pending state growing too large in
      cases where the condition to match a query with a corresponding reply is
      never met, but yet DNS messages continue to be exchanged over the same
      connection 5-tuple (preventing cleanup of the pending state). (Jon Siwek)

  * Updates to httpmonitor and mimestats documentation. (Jeannette Dopheide)

  * Updates to Logs and Cluster documentation (Jeannette Dopheide)

2.2-147 | 2014-02-07 08:06:53 -0800

  * Fix x509-extension test sometimes failing. (Johanna Amann)

2.2-144 | 2014-02-06 20:31:18 -0800

  * Fixing bug in POP3 analyzer. With certain input the analyzer could
    end up trying to write to non-writable memory. (Robin Sommer)

2.2-140 | 2014-02-06 17:58:04 -0800

  * Fixing memory leaks in input framework. (Robin Sommer)

  * Add script to detect filtered TCP traces. Addresses BIT-1119. (Jon
    Siwek)

2.2-137 | 2014-02-04 09:09:55 -0800

  * Minor unified2 script documentation fix. (Jon Siwek)

2.2-135 | 2014-01-31 11:09:36 -0800

  * Added some grammar and spelling corrections to Installation and
    Quick Start Guide. (Jeannette Dopheide)

2.2-131 | 2014-01-30 16:11:11 -0800

  * Extend file analysis API to allow file ID caching. This allows an
    analyzer to either provide file IDs associated with some file
    content or to cache a file ID that was already determined by
    script-layer logic so that subsequent calls to the file analysis
    interface can bypass costly detours through script-layer.  This
    can yield a decent performance improvement for analyzers that are
    able to take advantage of it and deal with streaming content (like
    HTTP, which has been adapted accordingly). (Jon Siwek)

2.2-128 | 2014-01-30 15:58:47 -0800

  * Add leak test for Exec module. (Johanna Amann)

  * Fix file_over_new_connection event to trigger when entire file is
    missed. (Jon Siwek)

  * Improve TCP connection size reporting for half-open connections.
    (Jon Siwek)

  * Improve gap reporting in TCP connections that never see data. We
    no longer accomodate SYN/FIN/RST-filtered traces by not reporting
    missing data. The behavior can be reverted by redef'ing
    "detect_filtered_trace". (Jon Siwek)

  * Improve TCP FIN retransmission handling. (Jon Siwek)
```

```
2.2-120 | 2014-01-28 10:25:23 -0800

  * Fix and extend x509_extension() event, which now actually returns
    the extension. (Johanna Amann)

    New event signauture:

        event x509_extension(c: connection, is_orig: bool, cert:X509, extension: X509_
↪extension_info)

2.2-117 | 2014-01-23 14:18:19 -0800

  * Fixing initialization context in anonymous functions. (Robin
    Sommer)

2.2-115 | 2014-01-22 12:11:18 -0800

  * Add unit tests for new Bro Manual docs. (Jon Siwek)

  * New content for the "Using Bro" section of the manual. (Rafael
    Bonilla/Jon Siwek)

2.2-105 | 2014-01-20 12:16:48 -0800

  * Support GRE tunnel decapsulation, including enhanced GRE headers.
    GRE tunnels are treated just like IP-in-IP tunnels by parsing past
    the GRE header in between the delivery and payload IP packets.
    Addresses BIT-867. (Jon Siwek)

  * Simplify FragReassembler memory management. (Jon Siwek)

2.2-102 | 2014-01-20 12:00:29 -0800

  * Include file information (MIME type and description) into notice
    emails if available. (Justin Azoff)

2.2-100 | 2014-01-20 11:54:58 -0800

  * Fix caching of recently validated SSL certifcates. (Justin Azoff)

2.2-98 | 2014-01-20 11:50:32 -0800

  * For notice suppresion, instead of storing the entire notice in
    Notice::suppressing, just store the time the notice should be
    suppressed until. This saves significant memory but can no longer
    raise end_suppression, which has been removed. (Justin Azoff)

2.2-96 | 2014-01-20 11:41:07 -0800

  * Integrate libmagic 5.16. Bro now now always relies on
    builtin/shipped magic library/database. (Jon Siwek)

  * Bro now requires a CMake 2.8.x, but no longer a pre-installed
    libmagic. (Jon Siwek)

2.2-93 | 2014-01-13 09:16:51 -0800
```

```
  * Fixing compile problems with some versions of libc++. Reported by
    Craig Leres. (Robin Sommer)

2.2-91 | 2014-01-13 01:33:28 -0800

  * Improve GeoIP City database support. When trying to open a city
    database, it now considers both the "REV0" and "REV1" versions of
    the city database instead of just the former. (Jon Siwek)

  * Broxygen init fixes. Addresses BIT-1110. (Jon Siwek)

    - Don't check mtime of bro binary if BRO_DISABLE_BROXYGEN env var set.

    - Fix failure to locate bro binary if invoking from a relative
      path and '.' isn't in PATH.

  * Fix for packet writing to make it use the global snap length.
    (Seth Hall)

  * Fix for traffic with TCP segmentation offloading with IP header
    len field being set to zero. (Seth Hall)

  * Canonify output of a unit test. (Jon Siwek)

  * A set of documentation updates. (Daniel Thayer)

      - Fix typo in Bro 2.2 NEWS on string indexing.
      - Fix typo in the Quick Start Guide, and clarified the
        instructions about modifying crontab.
      - Add/fix documentation for missing/misnamed event parameters.
      - Fix typos in BIF documentation of hexstr_to_bytestring.
      - Update the documentation of types and attributes.
      - Documented the new substring extraction functionality.
      - Clarified the description of "&priority" and "void".

2.2-75 | 2013-12-18 08:36:50 -0800

  * Fixing segfault with mismatching set &default in record fields.
    (Robin Sommer)

2.2-74 | 2013-12-16 08:49:55 -0800

  * Improve warnings emitted from raw/execute input reader. (Jon
    Siwek)

  * Further improve core.when-interpreter-exceptions unit test. (Jon
    Siwek)

2.2-72 | 2013-12-12 07:12:47 -0800

  * Improve the core.when-interpreter-exceptions unit test to prevent
    it from occasionally timing out. (Jon Siwek)

2.2-70 | 2013-12-10 15:02:50 -0800

  * Fix (harmless) uninitialized field in basename/dirname util
    wrapper. (Jon Siwek)
```

```
2.2-68 | 2013-12-09 15:19:37 -0800

  * Several improvements to input framework error handling for more
    robustness and more helpful error messages. Includes tests for
    many cases. (Johanna Amann)

2.2-66 | 2013-12-09 13:54:16 -0800

  * Fix table &default reference counting for record ctor expressions.
    (Jon Siwek)

  * Close signature files after done parsing. (Jon Siwek)

  * Fix unlikely null ptr deref in broxygen::Manager. (Jon Siwek)

  * FreeBSD build fix addendum: unintended variable shadowing. (Jon
    Siwek)

  * Fix build on FreeBSD. basename(3)/dirname(3) const-ness may vary
    w/ platform. (Jon Siwek)

  * Updated software framework to support parsing IE11 user-agent
    strings. (Seth Hall)

  * Fix the irc_reply event for several server message types. (Seth
    Hall)

  * Fix memory leak in input framework. If the input framework was
    used to read event streams and those streams contained records
    with more than one field, not all elements of the threading Values
    were cleaned up. Addresses BIT-1103. (Johanna Amann)

  * Minor Broxygen improvements. Addresses BIT-1098. (Jon Siwek)

2.2-51 | 2013-12-05 07:53:37 -0800

  * Improve a unit test involving 'when' conditionals. (Jon Siwek)

2.2-48 | 2013-12-04 13:45:47 -0800

  * Support omission of string slice low/high indices, BIT-1097.

    Omission of the low index defaults to 0:

        s = "12345"; s[:3] == "123"

    Omission of the high index defaults to length of the string:

        s = "12345"; s[3:] == "45" (Jon Siwek)

  * Tweak to SMTP script to adjust for new string slicing behaviour.
    (Robin Sommer)

  * Test updates. (Robin Sommer)

2.2-44 | 2013-12-04 12:41:51 -0800

  * Fix string slice notation. Addresses BIT-1097. (Jon Siwek)
```

```
    Slice ranges were not correctly determined for negative indices
    and also off by one in general (included one more element at the
    end of the substring than what actually matched the index range).
    It's now equivalent to Python slice notation.  Accessing a string
    at a single index is also the same as Python except that an
    out-of-range index returns an empty string instead of throwing an
    expection.

2.2-41 | 2013-12-04 12:40:51 -0800

  * Updating tests. (Robin Sommer)

2.2-40 | 2013-12-04 12:16:38 -0800

  * ssl_client_hello() now receives a vector of ciphers, instead of a
    set, to preserve their order. (Johanna Amann)

2.2-38 | 2013-12-04 12:10:54 -0800

  * New script misc/dump-events.bro, along with core support, that
    dumps events Bro is raising in an easily readable form for
    debugging. (Robin Sommer)

  * Prettyfing Describe() for record types. If a record type has a
    name and ODesc is set to short, we now print the name instead of
    the full field list. (Robin Sommer)

2.2-35 | 2013-12-04 10:10:32 -0800

  * Rework the automated script-reference documentation generation
    process, broxygen. Addresses BIT-701 and BIT-751. (Jon Siwek)

    Highlights:

        - Remove --doc-scripts and -Z options to toggle documentation
          mode. The parser is now always instrumented to gather
          documentation from comments of the form "##", "##!", or
          "##<".

        - Raw comments are available at runtime through several BIF
          functions: get_*_comments;

        - Add --broxygen and -X options to toggle generating
          reST-format documentation output, driven by a config file
          argument.

        - Add a "broxygen" Sphinx extension domain, allowing certain
          pieces of documentation to be generated on-the-fly via
          invoking a Bro process. Re-organized/cleaned up the Sphinx
          source tree in doc/ to use this in some places.

2.2-11 | 2013-12-03 10:56:28 -0800

  * Unit test for broccoli vector support. (Jon Siwek)

  * Changed ordering of Bro type tag enum, which was out of sync. (Jon
    Siwek)
```

```
2.2-9 | 2013-11-18 14:03:21 -0800

  * Update local.bro for Bro >= 2.2. The commented out Notice::policy
    example didn't work anymore. (Daniel Thayer)

2.2-6 | 2013-11-15 07:05:15 -0800

  * Make "install-example-configs" target use DESTDIR. (Jon Siwek)

2.2-5 | 2013-11-11 13:47:54 -0800

  * Fix the irc_reply event for certain server message types. (Seth
    Hall)

  * Fixed Segmentation fault in SQLite Writer. (Jon Crussell)

2.2 | 2013-11-07 10:25:50 -0800

  * Release 2.2.

  * Removing location information from ssh.log in external tests.
    (Robin Sommer)

2.2-beta-199 | 2013-11-07 00:36:46 -0800

  * Fixing warnings during doc build. (Robin Sommer)

2.2-beta-198 | 2013-11-06 22:54:30 -0800

  * Update docs and tests for a recent change to detect-MHR.bro
    (Daniel Thayer)

  * Update tests and baselines for sumstats docs. (Daniel Thayer)

2.2-beta-194 | 2013-11-06 14:39:50 -0500

  * Remove resp_size from the ssh log. Refactor when we write out to
    the log a bit. Geodata now works reliably. (Vlad Grigorescu)

  * Update VirusTotal URL to work with changes to their website and
    changed it to a redef. (Vlad Grigorescu)

  * Added a document for the SumStats framework. (Seth Hall)

2.2-beta-184 | 2013-11-03 22:53:42 -0800

  * Remove swig-ruby from required packages section of install doc.
    (Daniel Thayer)

2.2-beta-182 | 2013-11-01 05:26:05 -0700

  * Adding source and original copyright statement to Mozilla cert
    list. (Robin Sommer)

  * Canonfying an intel test to not depend on output order. (Robin
    Sommer)
```

```
2.2-beta-177 | 2013-10-30 04:54:54 -0700

  * Fix thread processing/termination conditions. (Jon Siwek)

2.2-beta-175 | 2013-10-29 09:30:09 -0700

  * Return the Dir module to file name tracking instead of inode
    tracking to avoid missing files that reuse a formerly seen inode.
    (Seth Hall)

  * Deprecate Broccoli Ruby bindings and no longer build them by
    default; use --enable-ruby to do so. (Jon Siwek)

2.2-beta-167 | 2013-10-29 06:02:38 -0700

  * Change percent_lost in capture-loss from a string to a double.
    (Vlad Grigorescu)

  * New version of the threading queue deadlock fix. (Robin Sommer)

  * Updating README with download/git information. (Robin Sommer)

2.2-beta-161 | 2013-10-25 15:48:15 -0700

  * Add curl to list of optional dependencies. It's used by the
    active-http.bro script. (Daniel Thayer)

  * Update test and baseline for a recent doc test fix. (Daniel
    Thayer)

2.2-beta-158 | 2013-10-25 15:05:08 -0700

  * Updating README with download/git information. (Robin Sommer)

2.2-beta-157 | 2013-10-25 11:11:17 -0700

  * Extend the documentation of the SQLite reader/writer framework.
    (Johanna Amann)

  * Fix inclusion of wrong example file in scripting tutorial.
    Reported by Michael Auger @LM4K. (Johanna Amann)

  * Alternative fix for the thrading deadlock issue to avoid potential
    performance impact. (Johanna Amann)

2.2-beta-152 | 2013-10-24 18:16:49 -0700

  * Fix for input readers occasionally dead-locking. (Robin Sommer)

2.2-beta-151 | 2013-10-24 16:52:26 -0700

  * Updating submodule(s).

2.2-beta-150 | 2013-10-24 16:32:14 -0700

  * Change temporary ASCII reader workaround for getline() on
    Mavericks to permanent fix. (Johanna Amann)
```

```
2.2-beta-148 | 2013-10-24 14:34:35 -0700

  * Add gawk to list of optional packages. (Daniel Thayer)

  * Add more script package README files. (Daniel Thayer)

  * Add NEWS about new features of BroControl and upgrade info.
    (Daniel Thayer)

  * Intel framework notes added to NEWS. (Seth Hall)

  * Temporary OSX Mavericks libc++ issue workaround for getline()
    problem in ASCII reader. (Johanna Amann)

  * Change test of identify_data BIF to ignore charset as it may vary
    with libmagic version. (Jon Siwek)

  * Ensure that the starting BPF filter is logged on clusters. (Seth
    Hall)

  * Add UDP support to the checksum offload detection script. (Seth
    Hall)

2.2-beta-133 | 2013-10-23 09:50:16 -0700

  * Fix record coercion tolerance of optional fields. (Jon Siwek)

  * Add NEWS about incompatible local.bro changes, addresses BIT-1047.
    (Jon Siwek)

  * Fix minor formatting problem in NEWS. (Jon Siwek)

2.2-beta-129 | 2013-10-23 09:47:29 -0700

  * Another batch of documentation fixes and updates. (Daniel Thayer)

2.2-beta-114 | 2013-10-18 14:17:57 -0700

  * Moving the SQLite examples into separate Bro files to turn them
    into sphinx-btest tests. (Robin Sommer)

2.2-beta-112 | 2013-10-18 13:47:13 -0700

  * A larger chunk of documentation fixes and cleanup. (Daniel Thayer)

    Apart from many smaller improves this includes in particular:

        * Add README files for most Bro frameworks and base/protocols.
        * Add README files for base/protocols.
        * Update installation instructions.
        * Improvements to file analysis docs and conversion to using
          btest sphinx.

2.2-beta-80 | 2013-10-18 13:18:05 -0700

  * SQLite reader/writer documentation. (Johanna Amann)

  * Check that the SQLite reader is only used in MANUAL reading mode.
```

```
    (Johanna Amann)

  * Rename the SQLite writer "dbname" configuration option to
    "tablename". (Johanna Amann)

  * Remove the "dbname" configuration option from the SQLite reader as
    it wasn't used there. (Johanna Amann)

2.2-beta-73 | 2013-10-14 14:28:25 -0700

  * Fix misc. Coverity-reported issues (leaks, potential null pointer
    deref, dead code, uninitialized values,
    time-of-check-time-of-use). (Jon Siwek)

  * Add check for sqlite3 command to tests that require it. (Daniel
    Thayer)

2.2-beta-68 | 2013-10-14 09:26:09 -0700

  * Add check for curl command to active-http.test. (Daniel Thayer)

2.2-beta-64 | 2013-10-14 09:20:04 -0700

  * Review usage of Reporter::InternalError, addresses BIT-1045.

    Replaced some with InternalWarning or AnalyzerError, the later
    being a new method which signals the analyzer to not process
    further input. (Jon Siwek)

  * Add new event for TCP content file write failures:
    "contents_file_write_failure". (Jon Siwek)

2.2-beta-57 | 2013-10-11 17:23:25 -0700

  * Improve Broxygen end-of-sentence detection. (Jon Siwek)

2.2-beta-55 | 2013-10-10 13:36:38 -0700

  * A couple of new TLS extension numbers. (Johanna Amann)

  * Suport for three more new TLS ciphers. (Johanna Amann)

  * Removing ICSI notary from default site config. (Robin Sommer)

2.2-beta-51 | 2013-10-07 17:33:56 -0700

  * Polishing the reference and scripting sections of the manual.
    (Robin Sommer)

  * Fixing the historical CHANGES record. (Robin Sommer)

  * Updating copyright notice. (Robin Sommer)

2.2-beta-38 | 2013-10-02 11:03:29 -0700

  * Fix uninitialized (or unused) fields. (Jon Siwek)

  * Remove logically dead code. (Jon Siwek)
  * Remove dead/unfinished code in unary not expression.  (Jon Siwek)

  * Fix logic for failed DNS TXT lookups. (Jon Siwek)

  * A couple null ptr checks. (Jon Siwek)

  * Improve return value checking and error handling. (Jon Siwek)

  * Remove unused variable assignments. (Jon Siwek)

  * Prevent division/modulo by zero in scripts. (Jon Siwek)

  * Fix unintentional always-false condition. (Jon Siwek)

  * Fix invalidated iterator usage. (Jon Siwek)

  * Fix DNS_Mgr iterator mismatch. (Jon Siwek)

  * Set safe umask when creating script profiler tmp files. (Jon Siwek)

  * Fix nesting/indent level whitespace mismatch. (Jon Siwek)

  * Add checks to avoid improper negative values use. (Jon Siwek)

2.2-beta-18 | 2013-10-02 10:28:17 -0700

  * Add support for further TLS cipher suites. (Johanna Amann)

2.2-beta-13 | 2013-10-01 11:31:55 -0700

  * Updating bifcl usage message. (Robin Sommer)

  * Fix bifcl getopt() usage. (Jon Siwek)

2.2-beta-8 | 2013-09-28 11:16:29 -0700

  * Fix a "make doc" warning. (Daniel Thayer)

2.2-beta-4 | 2013-09-24 13:23:30 -0700

  * Fix for setting REPO in Makefile. (Robin Sommer)

  * Whitespace fix. (Robin Sommer)

  * Removing :doc: roles so that we can render this with docutils
    directly. (Robin Sommer)

2.2-beta | 2013-09-23 20:57:48 -0700

  * Update 'make dist' target. (Jon Siwek)

2.1-1387 | 2013-09-23 11:54:48 -0700

  * Change submodules to fixed URL. (Jon Siwek)

  * Updating NEWS. (Robin Sommer)
  * Fixing an always false condition. (Robin Sommer)

  * Fix required for compiling with clang 3.3. (Robin Sommer)

2.1-1377 | 2013-09-20 14:38:15 -0700

  * Updates to the scripting introduction. (Scott Runnels)

  * Kill raw input reader's child by process group to reliably clean
    it up. (Jon Siwek)

2.1-1368 | 2013-09-19 20:07:57 -0700

  * Add more links in the GeoLocation document (Daniel Thayer)

2.1-1364 | 2013-09-19 15:12:08 -0700

  * Add links to Intelligence Framework documentation. (Daniel Thayer)

  * Update Mozilla root CA list. (Johanna Amann, Jon Siwek)

  * Update documentation of required packages. (Daniel Thayer)

2.1-1359 | 2013-09-18 15:01:50 -0700

  * Make client and server random available on script-level. Addresses
    BIT-950. (Eric Wustrow)

2.1-1357 | 2013-09-18 14:58:52 -0700

  * Update HLL API and its documentation. (Johanna Amann)

  * Fix case in HLL where hll_error_margin could be undefined.
    (Johanna Amann)

2.1-1352 | 2013-09-18 14:42:28 -0700

  * Fix a number of compiler warnings. (Daniel Thayer)

  * Fix cmake warning about ENABLE_PERFTOOLS not being used. (Daniel
    Thayer)

2.1-1344 | 2013-09-16 16:20:55 -0500

  * Refactor Analyzer::AddChildAnalyzer and usages. (Jon Siwek)

  * Minor refactor to SSL BinPAC grammer. (Jon Siwek)

  * Minor refactor to Broxygen enum comments. (Jon Siwek)

  * Fix possible (unlikely) use of uninitialized value. (Jon Siwek)

  * Fix/improve dereference-before-null-checks. (Jon Siwek)

  * Fix out-of-bounds memory accesses, and remove a
    variable-length-array usage. (Jon Siwek)

  * Fix potential mem leak. (Jon Siwek)
  * Fix double-free and deallocator mismatch. (Jon Siwek)

  * Fix another function val reference counting bug. (Jon Siwek)

2.1-1335 | 2013-09-12 16:13:53 -0500

  * Documentation fixes (Daniel Thayer, Jon Siwek)

  * Fix various potential memory leaks. (Jon Siwek)

  * Fix significant memory leak in function unserialization. (Jon Siwek)

  * Fix use-after-free and invalid/mismatch deallocator bugs. (Jon Siwek)

  * Fixed an issue with the HLL_UNIQUE SumStats plugin that caused a reporter error.␣
↪(Seth Hall)

  * Make the notice $actions field have a default empty set to avoid having to check␣
↪for it's presence. (Seth Hall)

  * Fix signatures that use identifiers of type table. (Jon Siwek)

  * Fix memory leak if a DNS request fails to be made. (Jon Siwek)

  * Fix memory leak in DNS TXT lookups. (Jon Siwek)

  * Fix raw execution input reader's signal blocking which resulted in lingering␣
↪processes. (Jon Siwek)

2.1-1306 | 2013-08-31 16:06:05 -0700

  * Reorganized and signifcantly extended documentation. This includes
    two new chapters contributed by Scott Runnels.

2.1-1216 | 2013-08-31 10:39:40 -0700


  * Support for probabilistic set cardinality, using the HyperLogLog
    algorithm. (Johanna Amann, Soumya Basu)

    Bro now provides the following BiFs:

        hll_cardinality_init(err: double, confidence: double): opaque of cardinality
        hll_cardinality_add(handle: opaque of cardinality, elem: any): bool
        hll_cardinality_merge_into(handle1: opaque of cardinality, handle2: opaque of␣
↪cardinality): bool
        hll_cardinality_estimate(handle: opaque of cardinality): double
        hll_cardinality_copy(handle: opaque of cardinality): opaque of cardinality

2.1-1154 | 2013-08-30 08:27:45 -0700

  * Fix global opaque val segfault. Addresses BIT-1071. (Jon Siwek)

  * Fix malloc/delete mismatch. (Jon Siwek)

  * Fix invalid pointer dereference in AsciiFormatter. (Jon Siwek)
```

```
2.1-1150 | 2013-08-29 13:43:01 -0700

  * Fix input framework memory leaks. (Jon Siwek)
```

## BroControl

```
  * Fix memory leak in SOCKS analyzer for bad addr types. (Jon Siwek)
```

```
1.4-150 | 2016-08-09 13:38:17 -0400

  * Show python stack trace if unexpected exception is raised.
    (Daniel Thayer)

  * Improve broctl error messages and error handling across the board.
    (Daniel Thayer)

  * Add a new optional node type "logger" that will handle logging
    instead of the manager. (Daniel Thayer)

1.4-132 | 2016-07-14 18:23:27 -0400

  * Don't run capstats on interfaces with packet source prefix. (Daniel Thayer)

1.4-130 | 2016-07-13 14:36:34 -0400

  * Improve the text of crash reports with instructions on how to
    get a backtrace, which should reduce the amount of useless crash
    reports mailed to the Bro team. (Daniel Thayer)

1.4-127 | 2016-07-06 08:58:18 -0500

  * Ignore packet source prefix of interface name when using capstats. (Jan␣
↪Grashoefer)

1.4-125 | 2016-07-02 17:53:42 -0500

  * New plugin function "broctl_config" so plugin authors can add their own
    script code to the autogenerated broctl-config.bro script. (Seth Hall)

1.4-122 | 2016-07-02 12:05:23 -0500

  * Follow symlinks to directories when searching for plugins. (Jon Siwek)

1.4-119 | 2016-06-28 11:11:19 -0400

  * Fix race condition in reading/writing broctl-config.sh (Daniel Thayer)

1.4-117 | 2016-06-22 12:14:37 -0400

  * Improve broctl behavior when unable to stop a node. (Daniel Thayer)

1.4-112 | 2016-06-14 16:14:52 -0700

  * Fix a failing test on some platforms and improve its error
     message. (Daniel Thayer)

  * Add Bro plugin directory to broctl plugin search path. (Daniel Thayer)

  * Update test baselines. (Daniel Thayer)
```

```
  * Use macros to create file analyzer plugin classes. (Jon Siwek)

  * Add options to limit extracted file sizes w/ 100MB default. (Jon
    Siwek)

2.1-1117 | 2013-08-22 08:44:12 -0700
```

```
  * Changed the default value of the StatusCmdShowAll option so that
    the broctl status command runs faster. (Daniel Thayer)

  * Changed the status-timefmt test so that it can be run in parallel
    with the other tests. (Daniel Thayer)

  * Remove dead code and update docs. (Daniel Thayer)

  * Rename serialization set for cluster tests. (Daniel Thayer)

  * Change node hostname resolution to be more consistent. (Daniel Thayer)

  * Add another test for broctl start command. (Daniel Thayer)

  * Prevent start helper from getting in infinite loop. (Daniel Thayer)

1.4-100 | 2016-05-17 16:22:25 -0700

  * Updating baseline for Bro control framework change. (Robin Sommer)

  * Fix for running broctl tests on OS X 10.11 (Daniel Thayer)

1.4-96 | 2016-04-28 13:43:22 -0400

  * Fix inconsistent return value data type for some commands, so that
    they always return a CmdResult. (Daniel Thayer)

1.4-94 | 2016-04-28 13:29:34 -0400

  * Fix the top command on OS X 10.10 or newer. (Daniel Thayer)

  * Fix build-bro script for running broctl tests on FreeBSD. (Daniel Thayer)

1.4-91 | 2016-03-31 15:08:24 -0500

  * Explicitly close the Broccoli connection to avoid resource leak. (Aaron Eppert)

1.4-89 | 2016-03-31 12:02:19 -0500

  * Prevent ssh login banners from appearing in broctl output. (Jon Schipp)

1.4-87 | 2016-03-31 10:35:47 -0400

  * Eliminate unnecessary writes to the state db. (Daniel Thayer)

1.4-84 | 2016-03-11 16:32:46 -0600

  * Support ip command for getting local IP addrs. (Jon Schipp)

1.4-77 | 2016-01-20 14:44:36 -0500

  * Changed LogExpireInterval to allow users to specify a more
    granular log expire interval, which is a number followed by
    a unit: "day", "hr", or "min".  An integer value with no unit
    is still allowed and interpreted the same as before. (Daniel Thayer)

  * More verbose error message for logexpireinterval value. (Daniel Thayer)
  * Prevent log expire interval from being less than rotation interval. (Daniel␣
↪Thayer)

  * Improve the ps test diff canonifier. (Daniel Thayer)

  * Improve the cron-expire test script. (Daniel Thayer)


1.4-70 | 2016-01-19 22:42:10 -0600

  * Fix custom plugin commands to behave more like built-in commands. (Aaron Eppert/
↪Daniel Thayer)

  * Add README.rst -> doc/broctl.rst symlink. Addresses BIT-1413 (Johanna Amann)

1.4-61 | 2015-12-19 13:39:47 -0800

   * Add broctl.cfg options PcapSnaplen and PcapBuflen to set pcap's
     packet snap length and buffer size, respectively. (Jan Grashoefer)

1.4-57 | 2015-12-11 12:00:07 -0500

  * Simplify some code and fix a test that can fail on OS X. (Daniel Thayer)

  * Improvements to broctl documentation. (Daniel Thayer)

  * Improve diagnostic and error messages. (Daniel Thayer)

  * Add more private IP space to etc/networks.cfg (Daniel Thayer)

  * Add a new broctl option, MailArchiveLogFail, to control sending
    log archive mail. (Daniel Thayer)

  * Check for invalid option names and values more carefully. (Daniel Thayer)

  * Fix use of ssh to always use IP address to avoid host key verification
    failures, and use BatchMode consistently to avoid a misleading
    error message when rsync fails. (Daniel Thayer)

  * Changed post-terminate to attempt to archive logs that have already
    been rotated.  Also changed crash-diag output file extension to no
    longer use ".log" in order to avoid post-terminate trying to
    archive it. (Daniel Thayer)

  * Send email if post-terminate fails to archive logs, and changed
    the post-terminate script to run archive-log serially instead
    of multiple instances simultaneously in the background.
    (Daniel Thayer)

  * Rename logs in the spool/tmp/post-terminate directory to indicate
    they were successfully archived when archive-log is run with the "-c"
    option.  (Daniel Thayer)

  * Capture output of background post-terminate script to file
    "post-terminate.out" which might be helpful for debugging
    problems with log archival. (Daniel Thayer)
  * Add bro node type to post-terminate dir name (Daniel Thayer)

1.4-36 | 2015-12-08 13:21:05 -0500

  * Fix problem of unexpected ifconfig output with some locales (Daniel Thayer)

1.4-34 | 2015-10-27 21:13:15 -0500

  * Added plugin for custom load balancing (Jan Grashoefer)

1.4-30 | 2015-08-21 17:23:39 -0700

  * Updating submodule(s).

1.4-28 | 2015-07-29 15:33:37 -0500

  * Handle a missing broctl-config.sh symlink (Justin Azoff)

1.4-26 | 2015-07-27 14:13:43 -0400

  * Create broctl-config.sh automatically (Daniel Thayer)

  * Undo a previous change for lb_procs error checking (Daniel Thayer)

  * Update broctl.rst by running "make doc" (Daniel Thayer)

  * Convert boolean config values to python bool type (Daniel Thayer)

1.4-20 | 2015-07-27 09:12:44 -0400

  * Merge remote-tracking branch 'origin/topic/dnthayer/ticket1434' (Justin Azoff)

  * Improve the broctl top helper script for FreeBSD (Daniel Thayer)

1.4-18 | 2015-07-27 09:03:22 -0400

  * Improve error message for invalid broctl plugin config values (Daniel Thayer)

  * Improve error message for invalid broctl config values (Daniel Thayer)

  * Improve error checking for local IP addresses (Daniel Thayer)

  * Cleanup some error msgs and source code comments (Daniel Thayer)

  * Close ssh connections upon config reload (Daniel Thayer)

  * Check for dangling Bro nodes every time node.cfg is loaded (Daniel Thayer)

  * Improve check for dangling Bro nodes (Daniel Thayer)

  * Remove unnecessary state variable type conversions (Daniel Thayer)

  * Convert config option values to correct data type (Daniel Thayer)

  * Check config file contents rather than timestamp (Daniel Thayer)

  * Add ability for broctl to reload its configuration, which the
    deploy command will do if a config file change is detected. (Daniel Thayer)
  * Avoid caching config values because config might change (Daniel Thayer)

  * Update a broctl test file (Daniel Thayer)

  * Keep track of both loaded plugins and active plugins (Daniel Thayer)

  * Reorganize some code (no changes in functionality) (Daniel Thayer)

  * Remove some config options and add a new one (Daniel Thayer)


1.4-1 | 2015-07-22 13:20:49 -0500

  * Fix test setup script to not overwrite LD_LIBRARY_PATH (Jon Siwek)

1.4 | 2015-06-09 09:19:56 -0500

  * Release 1.4.

1.4-beta-22 | 2015-06-02 10:34:44 -0500

  * Update broctl man page for deploy command (Daniel Thayer)

  * Updating baselines. (Robin Sommer)


1.4-beta-20 | 2015-05-28 12:15:28 -0700

  * Slight output tweaks. (Robin Sommer)

1.4-beta-19 | 2015-05-28 11:59:39 -0700

  * Improve documentation on site-specific customization. (Daniel
    Thayer)

  * Don't use daemon threads in ssh_runner. (Daniel Thayer)

  * Improve broctl documentation. (Daniel Thayer)

  * Fix minor error with restart clean. (Daniel Thayer)

  * Improve and extend tests. (Daniel Thayer)

  * Improve error messages related to the env_vars option. (Daniel Thayer)

  * Remove code that was automatically removing quoted values of the
    env_vars option. (Daniel Thayer)

  * Show help when user runs broctl with unknown command. (Daniel
    Thayer)

  * Improve visibility of archive-log error messages. (Daniel Thayer)

  * Add sanity checks on broctl options. (Daniel Thayer)

  * Improve error messages involving the state database file.
    Addresses BIT-1397 (Daniel Thayer)
  * Fixed error when a broctl command outputs binary data. (Daniel
    Thayer)

  * Fix the config change warnings on Python 3. (Daniel Thayer)

  * Fix an issue with the ps plugin where the "run-bro" script would
    appear in the output on some systems. (Daniel Thayer)

  * Inform user to run broctl deploy to get started. (Daniel Thayer)

  * Fix communication with muxer for newer Python versions. (Daniel
    Thayer)

  * Set correct Python path in Python scripts. (Daniel Thayer)

1.4-beta | 2015-05-07 20:26:22 -0700

  * Release 1.4-beta.

1.3-221 | 2015-04-22 15:20:20 -0500

  * Improve the test build script to show build error output. (Daniel Thayer)

1.3-220 | 2015-04-21 14:54:49 -0400

  * Fix problem where use of broargs causes error message (Daniel Thayer)

  * Avoid unnecessary string building in logging functions (Daniel Thayer)

  * Handle broctl output messages more consistently (Daniel Thayer)

  * Don't show certain warnings when they're not useful (Daniel Thayer)

  * Fix the interactive command tab completion feature (Daniel Thayer)

  * Simplify some SQL and remove unused code in the state database (Daniel Thayer)

1.3-212 | 2015-04-17 15:27:14 -0500

  * Fix the use of the "first-line" helper script (Daniel Thayer)

  * Added a new broctl option "CommandTimeout" that specifies the number
    of seconds to wait for a command to return results.  This value is
    passed to ssh_runner. (Daniel Thayer)

  * Improve error reporting for ssh_runner (Daniel Thayer)

  * Changed the status command to run only one helper script so that the
    status command takes half as long to run in the worst-case scenario.
    This involved replacing the "cat-file" helper with a new one that
    can handle multiple files, and only outputs the first line of each file.
    (Daniel Thayer)

  * Remove unused default timeout values in ssh_runner.  Also changed the
    ping timeout and changed the code to actually use it. (Daniel Thayer)

  * Fix response handling (Justin Azoff)
  * Enable json serialization of CmdResult objects (Justin Azoff)

  * Enable BatchMode for ssh

    From the ssh manual:

        If set to ``yes'', passphrase/password querying will be disabled.
        This option is useful in scripts and other batch jobs where no user
        is present to supply the password. (Justin Azoff)

  * Improve some error messages (Daniel Thayer)

  * Fix to prevent broctl from hanging when an exception occurs.
    Make sure that the finish method is called (to signal that we're done
    to the ssh_runner worker threads). (Daniel Thayer)


1.3-197 | 2015-04-16 16:15:25 -0500

  * Use daemon threads only for remote hosts (Daniel Thayer)

  * Fix to prevent the broctl stop command from hanging (Daniel Thayer)

  * Remove the run-cmd helper script (Daniel Thayer)

1.3-185 | 2015-04-03 14:54:06 -0400

  * Update test baselines. (Daniel Thayer)

  * Improved error reporting in several cases. (Daniel Thayer)

  * Added checks if there are any nodes to start or stop to avoid
    executing code unnecessarily. (Daniel Thayer)

  * Preserve order of hosts in command lists to be executed. (Daniel
    Thayer)

  * Catch the KeyboardInterrupt exception. (Daniel Thayer)

  * Reorganize code for the df command. (Daniel Thayer)

  * Python 3 compatibility fixes. (Daniel Thayer)

  * Make sure "broctl deploy" error messages are visible. (Daniel Thayer)

  * Speedup the deploy command by checking only one node of each node
    type. (Daniel Thayer)

  * Fix a race condition that results in data loss on the SSH control
    channels. (Daniel Thayer)

  * While waiting for lock, show owning PID of lock. (Daniel Thayer)

  * Make sure broctl always closes any file that it opens. (Daniel Thayer)

  * Update broctl install requirements list. (Daniel Thayer)
  * Don't show log header lines in "broctl scripts" output. (Daniel
    Thayer)

  * Added functions to cleanup before broctl terminates (Daniel
    Thayer)

1.3-165 | 2015-03-30 13:46:23 -0500

  * BIT-1326: Add configure-time check for required sqlite3 python
    module. (Jon Siwek)

1.3-162 | 2015-03-17 09:36:26 -0700

  * Update the documentation. (Daniel Thayer)

  * Add a new command "deploy" which does a "check", "install", and
    "restart".  The intention of this command is to reduce the chance
    that users will forget to install after modifying their
    configuration. (Daniel Thayer)

  * Sort broctl command output for easy readability.

  * Remove duplicate nodes from input so that broctl can't run a
    command twice for the same Bro node. (Daniel Thayer)

  * Improve error output. (Daniel Thayer)

  * Allow specifying alternate Bro script directory via "--scriptdir"
    option of the configure script when building Bro. (Daniel Thayer)

  * Allow specifying alternate location for etc/ directory via the
    "--conf-files-dir" option of the configure script when building
    Bro. (Daniel Thayer)

  * Simplify internals of the main broctl script. (Daniel Thayer)

  * Removed the use of BROCTL_INSTALL_PREFIX for modifying the install
    prefix at run-time.  This was only intended for use by the test
    scripts. Now the test setup scripts just modify all the files
    where the install prefix is hard-coded. (Daniel Thayer)

1.3-150 | 2015-03-04 12:17:42 -0800

  * Significant improvements (mostly internal), reorganization, and
    cleanup across the whole code base. (Justin Azoff and Daniel
    Thayer)

    This includes:

      - Refactor broctl to make it usable as a library (reduce global
        state, module-level setup code, and functions return results
        instead of printing).

      - Integrate ssh_runner code into broctl to fix current problems
        (use only one connection per host instead of one per Bro node;
        broctl shouldn't hang when a host goes down or if we forgot to
        run "broctl install"),
      - Write state info using SQLite state storage instead of writing
        to a plain text file (broctl.dat).

      - When the node config changes, we now do additional checks if
        there are any Bro nodes running that are no longer in our node
        config and warn user if any are detected.

      - Keep track of the expected state (running or stopped) of each
        Bro node, and have broctl cron start or stop nodes as needed.

      - Improved broctl cron by adding two new options (MailHostUpDown
        and StatsLogEnable) to enable users the option to turn off
        unwanted functionality to speed up broctl cron and reduce the
        chance of errors.

      - When broctl cron tries to send email but fails, now it will
        output a message that includes the text it was trying to mail.

      - Silence warning messages that are intended for interactive use
        of broctl when broctl cron runs to reduce unwanted emails from
        cron.

      - Added new broctl option StatusCmdShowAll to enable users to
        speed up "broctl status" significantly.

      - Fixed the stats-to-csv script to not create files that can
        never include any data.

      - Fixed archive-log script to detect exit status of gzip or cp
        command, so that we don't delete log file when the archival
        fails.

      - Improved post-terminate script to process log files more
        consistently.

      - Made all broctl command output go to stdout (previously, some
        output would go to stderr, which made grepping or redirecting
        the output more difficult),

      - Improved the default broctl.cfg file to show more of the
        useful options.

      - Added more error checks to help catch errors earlier.

      - Some error message output is more specific and helpful now.

1.3-12 | 2014-12-08 13:53:23 -0800

  * Add man page for broctl. (Raúl Benencia)

1.3-9 | 2014-12-01 12:03:53 -0600

  * Remove execute permission on scripts not needing it. (Raúl Benencia)

1.3-8 | 2014-10-31 09:17:27 -0500

  * BIT-1166: Add configure options to fine tune local state dirs.
    (Jon Siwek)
```

```
1.3 | 2014-06-02 08:59:01 -0700

  * Fix for capstats to display correct interface name when using
    PF_RING+DNA with pfdnacluster_master. (Daniel Thayer)

  * Fix for capstats with PF_RING+DNA pfdnacluster_master.
    (Daniel Thayer)

1.3-beta | 2014-05-19 16:29:36 -0500

  * Improve documentation of PFRINGFirstAppInstance option (Daniel Thayer)

  * Update broctl.rst with "make doc" (no other changes) (Daniel Thayer)

  * Move some content into the main Bro docs in a new section "Cluster
    Configuration". (Daniel Thayer)

  * Rename the broctl option pfringdnafirstappinstance to
    pfringfirstappinstance. (Daniel Thayer)

  * Remove references to the now unused BROMAGIC (Daniel Thayer)

1.2-129 | 2014-05-01 20:58:28 -0700

  * A bug fix and feature add for PF_Ring support. (Seth Hall)

     - Reset the app_instance for the case where there
       are multiple dnaclusters on a single host.

     - Add naming support for zerocopy (zc) clusters.

  * Use a hash to determine if a config change occurred. (Daniel Thayer)

  * Change hosts() function in the plugin API to return a list of
    nodes instead of just hostnames. (Daniel Thayer)

  * Add warnings when node config or broctl.cfg has changed. (Daniel Thayer)

  * Code simplification, remove the unused broctl "home" option, and
    improved a couple warning messages. (Daniel Thayer)

  * Fixed a bug where broctl cron could email about the "$total"
    pseudo-node not receiving any packets. (Daniel Thayer)

  * Code reorganization for the getDf function to avoid direct output
    and thereby reporting the same error message multiple times for
    the same host. (Daniel Thayer)

  * Cleanup some code for style consistency, reformat some comments to
    fit on an 80-column display, and remove some dead code. (Daniel
    Thayer)

  * Replace the update-stats script with Python code. (Daniel Thayer)

  * Gather disk usage by host rather than by node. The output now also
    shows both node and host names and is now sorted by node type.
  * Adjust column widths for top, netstats, peerstatus commands.
    (Daniel Thayer)

  * Change the broctl exec command to run only once per host. (Daniel
    Thayer)

  * Changed the hosts() function so that it preserves the order of the
    returned node list as it was sorted by the nodes() function.
    (Daniel Thayer)

1.2-106 | 2014-04-10 08:32:18 -0700

  * Update test baselines, and minor code cleanup. (Daniel Thayer)

1.2-104 | 2014-04-05 01:01:29 -0400

  * Updated PF_Ring plugin now supports PF_Ring+DNA. (Seth Hall)

1.2-99 | 2014-03-30 22:21:20 +0200

  * Update documentation with better install/setup instructions.
    Addresses BIT-1160 (Daniel Thayer)

1.2-97 | 2014-03-16 07:40:31 -0700

  * Minor doc update for a broctl option. (Daniel Thayer)

  * Adjust broctl status output to avoid bad column alignment. (Daniel
    Thayer)

  * Do not ping when checking if a host is alive. Removed the ping
    from the host alive check because the ping might be blocked by a
    firewall, and neither Bro nor broctl needs the ability to ping
    hosts. (Daniel Thayer)

  * If the current version of Bro doesn't match the version when
    broctl install was previously run, then a warning message (to run
    broctl install) is displayed when broctl starts. Addresses
    BIT-1152. (Daniel Thayer)

  * Reduce the risk of losing track of state info. Changed the way
    broctl updates PIDs and crash flags by writing the new values to
    disk immediately, one at a time, as soon as each new value is
    available. Also changed the way that the state file is updated
    when each command finishes by doing the update as an atomic
    operation. (Daniel Thayer)

  * Better error handling for a number of broctl commands. (Daniel Thayer)

  * Improve error output when broctl install has not been run yet.
    (Daniel Thayer)

  * Fix a failing test on FreeBSD 10. (Daniel Thayer)

  * Changed the output of the check command to be more specific about
    what it is actually checking. (Daniel Thayer)

  * Improve handling of dead hosts and closed/hanging connections.
```

```
    (Daniel Thayer)

  * Fixed a typo in the run-bro script that was causing the memlimit
    option to be ignored. Added added a test to verify that memlimit
    is used. (Daniel Thayer)

  * Simplify code that execs commands locally. (Daniel Thayer)

  * Prevent infinite loop in start helper script if it cannot execute
    the run-bro script. (Daniel Thayer)

  * pf_ring plugin: Show error if lb_procs is needed but not given,
    and disable plugin if not used. (Daniel Thayer)

  * Catch an exception that is raised when loading a plugin that does
    not override all required methods, and output an error message.
    (Daniel Thayer)

  * Fix start helper script to return nonzero on error. (Daniel
    Thayer)

  * Improve start/stop command output for crashed nodes.

  * Added a test for stopping a node that crashes during shutdown.
    (Daniel Thayer)


1.2-73 | 2014-02-28 14:44:51 -0800

  * Added ability of broctl cron to expire entries in stats.log that
    are older than the number of days specified in the new broctl
    option StatsLogExpireInterval. Addresses BIT-123. (Daniel Thayer)

  * Add broctl option BroPort to change the starting Bro port.
    Addresses BIT-1117. (Daniel Thayer)

1.2-66 | 2014-02-06 20:29:20 -0800

  * Make sure logs are archived after broctl kills Bro. Addresses
    BIT-1126. (Daniel Thayer)

1.2-63 | 2014-02-04 09:10:39 -0800

  * Fix a few sporadic test failures. (Daniel Thayer)

1.2-61 | 2014-01-31 11:11:39 -0800

  * Fix error handling for process command. (Daniel Thayer)

  * Update and improve the tests of broctl process. (Daniel Thayer)

  * Improve broctl help message for the process command. (Daniel
    Thayer)

  * Reorder the broctl process command Bro arguments. Addresses
    BIT-1124. (Daniel Thayer)

1.2-56 | 2014-01-28 15:54:14 -0800
```

```
  * A large set of improvements to the test build scripts to address
    error scenarios, fix failures to report problems, and provide
    convenience features.  (Daniel Thayer)

    Includes:

        - New Makefile target "rerun" to more easily re-run failed
          tests.

        - Two new environment variables recognized by test scripts:

            * If Bro fails to build, you can define an environment
              variable BROCTL_TEST_BUILDARGS which specifies
              additional options that will be passed to Bro's
              "configure" script.

            * Defining BROCTL_TEST_USEBUILD will use the Bro default
              build directory (instead of a custom build directory for
              the broctl tests).

  * Add lots of new tests. (Daniel Thayer)

1.2-28 | 2014-01-22 10:47:49 -0800

  * Fix bug with timemachineport broctl option. (Daniel Thayer)

  * Improved formatting of cluster-layout.bro for readability. (Daniel
    Thayer)

1.2-26 | 2014-01-21 07:12:38 -0800

  * Update the docs. (Daniel Thayer)

1.2-23 | 2014-01-20 12:22:42 -0800

  * Move some output about slow nodes to debug.log. (Daniel Thayer)

  * Improve broctl output formatting. (Daniel Thayer)

  * Fix redundant emails from broctl cron when dead host found.
    (Daniel Thayer)

  * Fix broctl top on OS X Mavericks. (Daniel Thayer)

  * Fix plugin init return values. This also fixes the myricom plugin,
    which wasn't explicitly returning a value from its init method and
    therefore was being disabled as a result. (Daniel Thayer)

  * Enable dead hosts caching while in cron mode. (Justin Azoff)

  * Use getattr for looking up plugin methods for simplifying the
    plugin code. (Justin Azoff)

  * Remove redundant plugin initialization. (Justin Azoff)

1.2-12 | 2014-01-20 11:23:23 -0800
```

```
  * Fix bug with IPv6Comm broctl option, which had no effect. (Daniel Thayer)

1.2-10 | 2014-01-13 01:57:53 -0800

  * Add a new option "PFRINGClusterType" that allows a user to specify
    a PF_RING cluster type; it defaults to 4-tuple (which is different
    from the 6-tuple that previous versions used). The PF_RING plugin
    uses this information to set the corrresponding environment
    variable for a PF_RING-aware libpcap. Addresses BIT-1108. (Daniel
    Thayer)

  * Minor reorganization of the README to avoid redundancy. (Daniel
    Thayer)

1.2-3 | 2013-12-09 13:24:28 -0800

  * Remove unused Broxygen-style script comments. (Jon Siwek)

1.2 | 2013-11-07 07:04:54 -0800

  * Release 1.2.

1.2-beta-28 | 2013-11-06 00:22:24 -0800

  * Improve check-pid helper script. (Daniel Thayer)

1.2-beta-26 | 2013-11-01 04:51:57 -0700

  * Add another warning message when a host is not alive. (Daniel
    Thayer)

1.2-beta-24 | 2013-10-31 00:19:41 -0700

  * Do not check if the local host is "alive". (Daniel Thayer)

1.2-beta-22 | 2013-10-26 19:19:31 -0700

  * Document which broctl options override Bro script variables.
    (Daniel Thayer)

  * Updates and clarifications to docs. (Daniel Thayer)

1.2-beta-17 | 2013-10-18 13:22:16 -0700

  * Fix internal lookup of nodes, which would fail to return the right
    items in some cases when node naming didn't match standard
    terminology. Addresses BIT-1091. (Daniel Thayer)

1.2-beta-13 | 2013-10-10 13:38:58 -0700

  * Updating copyright notice. (Robin Sommer)

  * Fix the broctl "top" command output on Linux. (Daniel Thayer)

  * Fix a race condition when sendmail option is empty string. (Daniel
    Thayer)

  * Fix a deadlock when capturing output from local command. (Daniel
    Thayer)

  * Improve portability of shell scripts used by broctl. (Daniel
    Thayer)

  * Fix for setting REPO in Makefile. (Robin Sommer)

1.2-beta | 2013-09-23 20:30:31 -0700

  * Update 'make dist' target. (Jon Siwek)

  * Fix problem with the "broargs" options that would occur when a
    command-line argument in broargs contained a space. (Daniel
    Thayer)

  * Change submodules to fixed URL. (Jon Siwek)

1.1-190 | 2013-09-20 14:26:41 -0700

  * Add more links in BroControl documentation. (Daniel Thayer)

1.1-188 | 2013-09-18 14:46:10 -0700

  * Add tests for new BroControl features (CPU pinning, PF_RING
    multiple cluster IDs, "env_vars") (Daniel Thayer)

  * Fix link to git repo to be consistent with other links. (Daniel
    Thayer)

  * Fix broken doc links. (Jon Siwek)

1.1-182 | 2013-08-27 13:32:35 -0700

  * Improve CPU pinning documentation and error message. Addresses
    BIT-1068 (Daniel Thayer)

  * Switching to relative submodule paths. (Robin Sommer)

  * Documentation fixes. (Daniel Thayer)

  * Minor fixes for broctl tests. (Daniel Thayer)

  * Fix bug with usage of cmd_restart_pre method. (Daniel Thayer)

  * Remove unused subdirectory "spool/scripts". (Daniel Thayer)

  * Remove unused imports, variables, and semicolons. (Daniel Thayer)

1.1-171 | 2013-08-16 15:36:14 -0700

  * Changed and document the behavior of the SitePolicyPath broctl
    option to not clobber existing files/directories when copying, in
    order to match the expected behavior (directories earlier in the
    list take precedence over directories later in the list when
    duplicate filenames are encountered). Addresses BIT-714. (Daniel
    Thayer)

  * A series of changes to make broctl return useful exit codes. (Vlad
    Grigorescu, Daniel Thayer).

    Generally, broctl now returns 0 if everything went ok with regards
    to what the documentation says should have happened, and 1
    otherwise. We keep the following exceptions for now though:

        - "cron" always returns 0.
        - "status" and "top" return 0 if all bro nodes are
          running, and returns 1 otherwise.
        - commands provides by plugins always return 0.

1.1-158 | 2013-08-02 17:06:57 -0700

  * Add ability to set environment variables in node.cfg and
    broctl.cfg via new "env_vars" options taking a comma-separated
    list (e.g., "env_vars=VAR1=1,VAR2=2"). Variables in node.cfg take
    prioroty over broctl.cfg. Addresses BIT-1010. (Daniel Thayer)

1.1-150 | 2013-07-14 08:00:44 -0700

  * Fix broken link in README. (Bernhard Amann, thanks kraigu)

1.1-148 | 2013-07-03 17:06:44 -0700

  * Updates to test infrastructure. (Daniel Thayer)

    - Fix canonifier script for handling missing gdb.
    - Update baselines for recent changes to crash-diag.
    - Remove "make quick" from the README.
    - Minor cleanup of the build script.
    - Remove unused Makefile variable.
    - Remove the "-j" option to make as it can cause lock-ups on
      some machines.
    - Replace realpath command with more portable Python equivalent.

1.1-140 | 2013-06-07 16:35:08 -0700

  * Adding OS to crash output. (Robin Sommer)

  * Giving the broctl test suite its own build directory. (Robin Sommer)

1.1-137 | 2013-05-31 17:16:14 -0700

  * New regression test suite for BroControl. "make test" runs it. See
    testing/README for more information. (Daniel Thayer)

1.1-101 | 2013-05-24 17:55:41 -0700

  * Add support for CPU pinning. To use CPU pinning, a new per-node
    option "pin_cpus" can be specified in node.cfg, and the OS must be
    either Linux or FreeBSD (if such a node.cfg is used on another OS,
    then the "pin_cpus" option is ignored). Addresses #996. (Daniel
    Thayer)

1.1-99 | 2013-05-24 17:34:44 -0700

  * Allow multiple conn-summary.log files to be processed to avoid
    conflicts when stopping Bro shortly after a log rotation. (Daniel
    Thayer)

  * Prevent deletion of unarchived logs during "broctl stop" when
    archiving takes a while. (Daniel Thayer)

1.1-94 | 2013-05-17 13:29:04 -0700

  * Don't import readline, it's loaded implicitly already. (Daniel
    Thayer)

1.1-92 | 2013-05-17 07:37:13 -0700

  * Removing uncessary directory check. (Robin Sommer)

1.1-91 | 2013-05-16 20:25:00 -0700

  * Stop trying to create the stats/www directory if it already
    exists. Addresses #1007. (Seth Hall)

  * Another batch of fixes. (Daniel Thayer)

    This includes:

    - Fix usage of PF_RING interface containing semicolons.
    - Fix broctl exec command to check for errors.
    - Fix a race condition during broctl start.
    - Remove some dead code.
    - Fix exit status output in debug log.

  * Add support for the "--scriptdir" configure option. Adresses
    #993. (Daniel Thayer)

1.1-79 | 2013-05-10 19:39:55 -0700

  * A set of bug fixes and robustness improvements. (Daniel Thayer)

    This includes:

    - Add more error checking and reporting to cron command.
    - Improve error checking of top helper output.
    - Improve error checking of capstats output.
    - Fix a bug when the time command is not found.
    - Fix the broctl top and cron commands on OS X.
    - Fix a couple of bugs in the broctl ps plugin.
    - Remove unused broctl scripts.
    - Improve the check-pid helper script.

1.1-63 | 2013-04-25 16:14:51 -0400

  * Add support for multiple PF_RING cluster IDs

    Instead of assigning the same PF_RING cluster ID to every worker
    in a Bro cluster, the pf_ring broctl plugin has been modified to
    automatically assign a different PF_RING cluster ID for each se
    of workers on a host that all sniff the same interface.  The firs
    such set of workers on a host are assigned the globally-configured
    PF_RING cluster ID (this is the "pfringclusterid" broctl option in
    broctl.cfg).  Each subsequent set of workers on a host that sniff
    another interface are assigned a different value (incremented by
    one from previous value). Addresses #943. (Daniel Thayer)

1.1-61 | 2013-03-22 12:25:22 -0700

  * Fix problem with the cron command hanging sometimes. Addresses
    #591. (Seth Hall)

1.1-59 | 2013-03-17 13:36:04 -0700

  * Lots of small fixes, cleanup, and documentation improvemets (in
    particular, but not only, to the plugin API). (Daniel Thayer).

    This includes:

        - Check for plugins with same prefix
        - Prevent capstats from being run with invalid args
        - Fix plugin inconsistency for certain broctl commands
        - Document the broctl user option KeepLogs?
        - Add a note in documentation about editing crontab
        - Fix broctl plugin option names to be case-insensitive
        - Remove reserved word "cluster" from node args
        - Fix documentation of broctl commands
        - Add calls to plugin cmd_restart_pre/post methods
        - Fix instructions for adding plugin directories
        - Fix the broctl check command to report results
        - Fix handling of cmd_diag_pre for diag command
        - Changed return value of plugin API "execute" method
        - Add return value to some cmd_<cmd>_pre methods
        - Add a check for state variables in broctl.cfg
        - Changed "hosts" method to return list of hosts
        - Call "done" method from plugin API
        - Call hostStatusChanged with correct arg type
        - Fix the parseNodes method in plugin API
        - Fix the "error" method in broctl plugin API
        - Fixed tab-completion of commands with node args
        - Fix broctl plugin API documentation errors
        - Fix typos in TestPlugin? output messages
        - Add cron "--no-watch" option to broctl "help" output
        - Fix the "execute" method of the Plugin class
        - Fix various bugs and remove some unused code

1.1-26 | 2012-12-20 17:53:52 -0800

  * Add Bro version to crash reports. (Robin Sommer)

  * Add a new broctl option "MailConnectionSummary" that specifies
    whether or not to mail the connection summary reports.  (Daniel
    Thayer)

1.1-23 | 2012-12-06 15:52:20 -0800

  * Update documentation for recent MailFrom change. (Daniel Thayer)

1.1-21 | 2012-12-06 08:34:14 -0800

  * MailFrom broctl.cfg option now adds a redef for Notice::mail_from.
    (Jon Siwek)
  * Bump CPack RPM package requirement to python >= 2.6.0. (Jon Siwek)

1.1-18 | 2012-10-31 14:24:27 -0700

  * Add new broctl.cfg option "MailAlarmsInterval" to allow user to
    specify alarm mail interval. Default is once per day. (Daniel
    Thayer)

1.1-12 | 2012-10-24 15:53:48 -0700

    * Add a message at the top of broctl-generated crash report emails
      that explains how to submit the crash report to a mailing list
      address. Addresses #876. (Daniel Thayer)

1.1-10 | 2012-10-19 15:10:20 -0700

  * Fix `broctl install` to now also copy subdirs in SitePolicyPath.
    Addresses #902. (Jon Siwek)

1.1-8 | 2012-10-19 14:52:23 -0700

  * Add options CompressCmd and CompressExtension to customize log
    compressions scheme. (Justin Azoff)

1.1-3 | 2012-09-25 06:23:34 -0700

  * Updates to documentation. (Daniel Thayer)

1.1 | 2012-08-24 15:09:04 -0700

  * Fix MailAlarmsTo broctl config option. Addresses #814. (Daniel
    Thayer)

  * Fix configure script to exit with non-zero status on error. (Jon
    Siwek)

1.1-beta-2 | 2012-08-10 12:29:56 -0700

  * Updates to disable STDERR printing from the reporter framework.
    (Seth Hall)

1.1-beta | 2012-07-20 07:03:21 -0700

  * Fix broctl startup when using custom config file dirs. (Jon Siwek)

  * Change crash report info to include stack traces from all threads.
    (Jon Siwek)

  * Changed the invocation of gdb that produces the crash report. (Jon
    Siwek)

1.0-64 | 2012-07-10 16:07:50 -0700

  * Remove automatic override of config file directory with /usr prefix.

  * Small updates to BroControl docs. (Daniel Thayer)
```

```
1.0-58 | 2012-07-02 15:55:06 -0700

  * Improvements to built-in load-balancing support. Instead of adding
    a separate worker entry in node.cfg for each Bro worker process on
    each worker host, it is now possible to just specify the number of
    worker processes on each host. (Daniel Thayer)

    This change adds three new keywords to the node.cfg file (to be
    used with worker entries): lb_procs (specifies number of workers
    on a host), lb_method (specifies what type of load balancing to
    use: pf_ring, myricom, or interfaces), and lb_interfaces (used
    only with "lb_method=interfaces" to specify which interfaces to
    load-balance on).

    Two new broctl plugins (which operate automatically and the user
    doesn't need to be aware of them) are added to set the appropriate
    environment variables when either PF_RING or myricom
    load-balancing is being used.

1.0-43 | 2012-07-02 15:40:01 -0700

  * Improve README. Rewrote the section on site-specific customization
    so that it is more clear about the load order of scripts relevant
    to site-specific customization.  Removed the description of
    several features that don't seem to work: "worker-1.local.bro" is
    not automatically loaded, there is no example policy in
    local-manager.bro, local-manager.bro and local-worker.bro do not
    automatically load local.bro, and proxies do not automatically
    load local-worker.bro. (Daniel Thayer)

1.0-40 | 2012-06-06 11:52:06 -0700

    * Fix the "cron disable" command, which didn't work. This also
      removes the config option CronEnabled. The command is now the
      only way to turn off cron operation. (Daniel Thayer)

1.0-38 | 2012-05-24 17:42:37 -0700

  * Improvements to IPv6 support. (Jon Siwek)

    - Add ability to manage a cluster over non-global IPv6 scope (e.g.
      link-local), by specifying "zone_id" keys per node in node.cfg
      and "ZoneID" option in broctl.cfg.

    - Replace socket.gethostbyname lookups with socket.getaddrinfo to
      support IPv6.

    - ::1 is now recognized as the IPv6 loopback and a "local" address
      where before 127.0.0.1 was expected.

    - Update usages of ping, ssh, rsync, and ifconfig to work with IPv6
      addresses.

    - New "IPv6Comm" option in broctl.cfg can be set to 0 to turn off
      IPv6-based communication capabilities (on by default).

1.0-35 | 2012-05-17 11:57:30 -0700
```

```
  * BroControl tweaks to support non-ASCII logs. (Robin)

        - The main change is that we give another argument to
          post-processors that indicates the writer type that produced
          the log. That comes with an incompatible part: the
          make-archive-name script now receives the writer as its
          2nd(!) argument. Customized versions need be adapted.

        - The standard postprocessors now check whether they are
          processing something else than ASCII logs and adapt their
          behaviour accordingly (e.g., by not compressing, and or not
          running trace-summary).

1.0-32 | 2012-05-14 17:20:17 -0700

  * Fix typos in broctl docs. (Daniel Thayer)

1.0-29 | 2012-05-03 11:34:29 -0700

  * Added an option to specify 'etc' directory. Addresses #801.
    (Daniel Thayer)

  * Fix typos. (Daniel Thayer)

1.0-24 | 2012-04-24 14:37:49 -0700

  * Update some broctl option descriptions. (Daniel Thayer)

1.0-22 | 2012-04-19 09:52:44 -0700

  * Options SitePolicyStandalone, SitePolicyManager, and
    SitePolicyWorker were unused. Now they are, and they replace the
    hard-coded defaults if defined. Addresses #797. (Daniel Thayer)

1.0-20 | 2012-04-19 09:08:32 -0700

  * Remove unused broctl options and fixed a couple of typos in the
    option names. (Daniel Thayer)

1.0-17 | 2012-04-16 18:06:28 -0700

  * Fixed lots of documentation typos and broken links. (Daniel
    Thayer)

  * Update broctl help information. (Daniel Thayer)


1.0-13 | 2012-04-09 15:59:17 -0700

  * Remove "-p" option from broctl "scripts" command help. (Daniel
    Thayer)

  * Updating helper script to work with conn.log in Bro 2.0. (Daniel
    Thayer)


1.0-9 | 2012-03-28 15:46:02 -0700
```

```
  * Improve error message when failing to update broctl-config.sh
    symlink (Jon Siwek)

  * Raise minimum required CMake version to 2.6.3. (Jon Siwek)

  * Remove the unused "PolicyDirBroCtl" option. (Daniel Thayer)

  * Rename the spool/policy directory so it is less visible. Addresses
    #767. (Daniel Thayer)

1.0 | 2012-01-10 18:57:50 -0800

  * Tweaks for OpenBSD support. (Jon Siwek)

0.5-beta-43 | 2012-01-03 14:45:40 -0800

  * broctl now creates spool directories it finds missing. Addresses
    #716. (Edward Groenendaal)

0.5-beta-39 | 2011-12-16 02:49:28 -0800

  * Add StopTimeout option to broctl.cfg that sets the number of
    seconds to wait after issuing the 'stop' command before sending a
    SIGKILL to Bro instances. Adresses #608. (Jon Siwek)

  * Add CommTimeout option to broctl.cfg that sets the number of
    seconds to timeout Broccoli connnections. Addresses #608. (Jon
    Siwek)

  * Re-order the way local.bro and local-<node>.bro scripts are
    loaded. Node-specific local scripts now load after local.bro so
    tha identifiers defined by the loading of local.bro can be used in
    them. Addresses #663 (Jon Siwek)

0.5-beta-34 | 2011-12-02 17:17:14 -0800

  * Make BroControl more robust when a node dies. (Robin Sommer)

  * Disable collecting of prof.logs. The logs can get huge, which lets
    cron take a while. (Robin Sommer)

  * Fix standalone->cluster upgrade failing to update logs/current
    symlink. Fixes #676. (Jon Siwek)

  * Fix broctl 'scripts' command in cluster mode. Fixes #655. (Jon
    Siwek)

  * Teach 'check' command to generate temporary versions of autogen.
    files. Addresses #658. (Jon Siwek)

  * Submodule README conformity changes. (Jon Siwek)

0.5-beta-20 | 2011-11-14 20:04:21 -0800

  * Fixing some platforms behaving poorly during configure-time checks
    when a superproject's languages didn't encompass a subproject's.
    (Jon Siwek)
  * Configure sendmail option in options.py instead of broctl.cfg.
    Fixed #645. (Jon Siwek)

  * Fix extraneous installation of BroControl plugins. (Jon Siwek)

  * Apply patch for BroControl Python 2.3/2.4 compatibility. Closes
    #662. (William Jones)

  * Avoid rerunning the previous command when hitting just enter in
    broctl. (Justin Azoff)

0.5-beta-12 | 2011-11-06 19:23:43 -0800

  * broctl.cfg now determines sendmail location at configure-time.
    Addreses #645 (Jon Siwek)

  * Disable log expiration by default. Addresses #613. (Jon Siwek)

  * Make symlink to broctl-config.sh update with `broctl install`.
    Addresses #648 (Jon Siwek)

  * Fixed a problem when host= in standalone is not 127.0.0.1 or
    localhost. (Seth Hall)

0.5-beta | 2011-10-27 17:45:15 -0700

  * Updating submodule(s). (Robin Sommer)

0.41-143 | 2011-10-26 10:15:16 -0500

  * Update submodules. (Jon Siwek)

0.41-142 | 2011-10-25 20:17:25 -0700

  * Updating submodule(s). (Robin Sommer)

0.41-137 | 2011-10-25 15:44:18 -0700

  * Updating CHANGES and VERSION. (Robin Sommer)

  * Make dist now cleans the copied source. (Jon Siwek)

0.41-130 | 2011-10-18 08:03:35 -0700

  * Distribution cleanup and some README fixes. (Robin Sommer)

  * Fixed a bug caused by communication framework API update. Reported
    by Daniel. (Seth Hall)

0.41-128 | 2011-10-06 17:23:03 -0700

  * Change broctl.cfg LogRotationInterval to be specificed in seconds. (Jon Siwek)

  * Force broctl 'process' command to enable local logging. Addresses
    #632 (Jon Siwek)

0.41-124 | 2011-10-05 16:58:10 -0700
```

```
   * New broctl.cfg option for log rotation interval. Addresses #630.
     (Jon Siwek)

   * Removed some of the broct/nodes/* scripts and instead
     consolidated their functionality into the node-specific scripts
     that come with Bro's cluster framework. (Jon Siwek)

   * Within the cluster framework, local-<node>.bro scripts should now
     be loaded after the distributions <node>.bro script so things can
     be overrided. (Jon Siwek)

   * Auto-generated broctl scripts are loaded after all node-specific
     scripts and can override their options. (Jon Siwek)

  * Move configuration of PFRINGClusterID from broctl.cfg.in to
    options.py. Addresses #621. (Jon Siwek)

  * Add configure-time check for libpcap PF_RING support. Addresses
    #621 (Jon Siwek)

  * Fixing typo with process command. (Robin Sommer)

  * Script cleanup.  (Seth Hall)

    - Reshuffling "check" functionality into check.bro.

    - Removing some code to deal with the non-existent react framework.

  * Give check command its own script for tuning options. Addresses
    #618). (Jon Siwek)

  * Stop and restart command now stop worker nodes first. Addresses
    #596. (Jon Siwek)

  * broctl check no longer rotates logs. Addresses #618. (Jon Siwek)

0.41-101 | 2011-09-08 02:20:28 -0400

  * Implementing PF_RING environment variables. (Seth Hall)

0.41-99 | 2011-09-04 09:08:59 -0700

  * Added --with-pcap configure option. (Jon Siwek)

  * Various smaller tweaks to CMake setup. (Jon Siwek)

  * Removed alarm log mailing postprocessing script from BroControl.
    (Jon Siwek)

  * Log rotation is disabled when using the 'process' command to
    analyze trace files. (Jon Siwek)

  * Fixed 'scripts' command. (Jon Siwek)

  * Fixed inconsistent rotated-log naming. (Jon Siwek)

  * Changed the 'mail-log' postprocessor to mail alarm.log's. (Jon
    Siwek)
  * Fix Config.state key capitalization inconsistencies. (Jon Siwek)

  * Fixes for broctl 'check' command. Addresses #548. (Seth Hall and
    Jon Siwek)

  * Updated README. (Jon Siwek)

  * Copy bro binary only in NFS mode (fixes #361). (Jon Siwek)

  * Fix install command failing because of missing parent dirs. (Jon Siwek)

  * Removing the analysis.dat file since it's not used anymore. (Seth Hall)

  * Better informational output if attempt to remove old scripts
    before installing new ones failes. Addresses #470. (Craig Leres)

  * Updating log rotation support for the new logging rotation code.
    (Seth Hall)

  * Updates for cleanup and meshing with Bro reorg. (Seth Hall)

0.41-73 | 2011-08-13 12:14:28 -0700

  * Moving README*. into subdir doc. The top-level README is now
    auto-generated. (Robin Sommer)

0.41-68 | 2011-08-05 12:49:30 -0700

  * Install example config files dynamically when the distribution
    version differs from existing version on disk. (Jon Siwek)

0.41-63 | 2011-08-03 22:10:40 -0700

  * Revamped how the work is split between Bro and BroControl. Much of
    functionality previously found in BroControl policy scripts has
    moved over to Bro. (Seth Hall)

  * Adapted BroControl to Bro 2.0 policy scripts.

  * A new plugin interface allows external Python code to hook into
    BroControl processing. See README for more information. (Robin
    Sommer)

    Two example plugins are shipped: (1) "ps.bro" shows all Bro
    processes currently running on any cluster node, even if not
    managed by BroControl; (2) "TestPlugin" is a demo plugin
    demonstrating all the functionality a plugin can use (but doesn't
    do anything sensible with it).

  * A new offline mode for processing a trace. The new command
    "process <trace>" runs Bro offline on the given trace, using the
    current BroControl configuration. One can optionally give give
    further Bro command line options and scripts. In cluster mode the
    the Bro process loads both manager and worker configurations
    simultaniously.

    Addresses #273. (Robin Sommer)
  * Removed the "analysis" command. (Seth Hall)

  * Installation does no longer differentiate between standalone and
    cluster mode. node.cfg now fully controls this. (Seth Hall)

  * Tons of little fixes, improvements, and polishing (Seth Hall, Jon
    Siwek, and Robin Sommer)

0.41-9 | 2011-06-01 11:35:36 -0700

  * Standardize shell script hashbang on install. (Jon Siwek)

  * Fix binary package broctl-config.sh symlink installation
    regression. (Jon Siwek)

  * Changes to allow DEB packaging via CPack, addresses #458. (Jon Siwek)

  * Fixed a problem with the "update" command, which could delete data
    from many global state tables unintentionally. (Seth Hall)

0.41-2 | 2011-05-02 11:29:07 -0700

  * Symlink install scripted at install time for CMake 2.6
    compatibility. (Jon Siwek)

0.41 | 2011-04-07 21:14:53 -0700

  * Tweaks to the documentation generation. (Robin Sommer)

  * CMake tweaks. (Jon Siwek)

  * Bugfix: trace-summary sampled in standalone mode rather than cluster
    mode. (Robin Sommer)

  * Bugfix: Creating links from the log directory to the current log files
    didn't work in standalone mode. (Robin Sommer)

0.4-19 | 2011-01-31 15:26:48 -0800

  * A new option CompressLogs (default on), indicating whether
    archived logs are to be gzipped. (Robin Sommer)

  * A lot of configure/cmake/install/package tuning. (Jon Siwek)

  * Adding /sbin and /usr/sbin to path local-interfaces script
    searches for ifconfig. Closes #293. (Robin Sommer)

  * Fixing uncaught exception in lock file handling. (Seth Hall).

  * Making cluster event specifications redefinable. (Seth Hall).

  * Fixing for pretty printing numerical values. (Seth Hall).

  * Fixing "netstats" command distinction between cluster and
    standalone mode. (Justin Azoff)

0.4-10 | 2011-01-15 14:14:05 -0800
```

```
  * Changes for CPack binary packaging (Jon Siwek)

  * Fix package configuration macro returning from sub-project too early (Jon Siwek)

  * Add warning when building and installing are done by different users (Jon Siwek)

  * Changes to broctl's "make install" process (Jon Siwek)

    - Simplify install by not compiling python code.
    - The broctl-config.sh symlink needs to be made at configure time
      and install()'ed in order for CPack packaging to correctly bundle it
    - Reverted a change in (90ddc4d) to that caused spool/ and logs/
      directories to not be installed in the case that they existed at
      configure time.

  * Fix for PackageMaker not accepting non-numeric versions (Jon Siwek)

0.4-9 | 2011-01-12 08:51:11 -0800

  * Making df portably deal with long lines in the OS's df output.
    (Robin Sommer)

0.4-8 | 2011-01-04 20:30:41 -0800

  * Changing some installation paths. "broctl install" copied a
    number of files to share/bro/*, which violates the common
    assumption that things there are static. It can also create
    permission problems if the user running "broctl install" is not
    the one installing Bro. So now the pieces copied/generated by
    "broctl install" are moved to spool/*. (Robin Sommer)

  * The CMake install does no longer recreate some of the top-level
    directories when they already exist. That makes it possible to
    now symlink them somewhere else after the first install. (Robin
    Sommer)

  * When broctl doesn't find spool/broctl.dat it no longer aborts
    but just warns. That allows CMake to skip installing an empty
    one. (Robin Sommer)

  * Deleting an unused policy file. (Robin Sommer)

  * Updating update-changes script. (Robin Sommer)

0.4-5 | 2010-12-20 14:10:25 -0800 | 768a9e550c3554de2e0bf9e3af2ae99400203046

  * New helper script for maintaing CHANGES file. (Robin Sommer)

0.4-1 | 2010-12-20 12:03:34 -0800 | a05be1242b4e06dca1bb1a38ed871e7e2d78181b

  * Fix for dealing with large vsize values reported by "top" (Craig
    Leres)

  * Fixed the top helper script to assign the command variable
    appropriately. (Seth Hall)

  * Escape commands given to CMake's execute_process (Jon Siwek)
```

```
0.4 | Fri Dec 10 01:35:36 2010 -0800 | df922e8a64a631aadb485b5044fe9ae1046d47ca

- Moving BroControl to its own git repository.

- Converting README to reST format.

- Renamed "Capstats" config option to "CapstatsPath".

- Merge with Subversion repository as of r7098. Incorporated changes:

  o Increasing default timeouts for scan detector significantly.

  o Increasing the manager's max_remote_events_processed to
    something large, as it would slow down the process too much
    otherwise and there's no other work to be interleaved with it
    anyway.

  o Adding debug output to cluster's part of catch-and-release
    (extends the debugging already present in policy/debug.bro)

  o Fixing typo in util.py. Closes #223.

  o Added note to README pointing to HTML version.

  o Disabling print_hook for proxies' remote.log.

  o broctl's capstats now reports a total as well, and stats.log
    tracks these totals. Closes #160.

  o Avoiding spurious "waiting for lock" messages in cron mode.
    Closes #206.

  o Bug fixes for installation on NFS.

  o Bug fix for top command on FreeBSD 8.

  o crash-diag now checks whether gdb is available.

  o trace-summary reports the sample factor in use in its output,
    and now also applies it to the top-local-networks output (not
    doing the latter was a bug).

  o Removed the default twice-a-day rotation for conn.log. The
    default rotation for conn.log now is now once every 24h, just
    like for all other logs with the exception of mail.log (which is
    still rotated twice a day, and thus the alarms are still mailed
    out twice a day).

  o Fixed the problem of logs sometimes being filed into the wrong
    directory (see the (now gone) FAQ entry in the README).

  o One can now customize the archive naming scheme. See the
    corresponding FAQ entry in the README.

  o Cleaned up, and extended, collection of cluster statistics.

    ${logdir}/stats now looks like this:
      drwxr-xr-x   4 bro  wheel      59392 Apr  5 17:55 .
      drwxr-xr-x  96 bro  wheel       2560 Apr  6 12:00 ..
      -rw-r--r--   1 bro  wheel        576 Apr  6 16:40 meta.dat
      drwxr-xr-x   2 bro  wheel       2048 Apr  6 16:40 profiling
      -rw-r--r--   1 bro  wheel  771834825 Apr  6 16:40 stats.log
      drwxr-xr-x   2 bro  wheel       2048 Apr  6 16:25 www

    stats.log accumulates cluster statistics collected every time
    "cron" is called.

    - profiling/ keeps the nodes' prof.logs.

    - www/ keeps a subset of stats.log in CSV format for easy plotting.

    - meta.dat contains meta information about the current cluster
    state (in particular which nodes we have, and when the last
    stats update was done).

    Note that there is not Web setup yet to actually plot the data
    in www/.

  o BroControl now automatically maintains links inside today's log
    archive directory pointing to the current live version of the
    corresponding log file (if Bro is running). For example:

    smtp.log.11:52:18-current -> /usr/local/cluster/spool/manager/smtp.log

  o Alarms mailed out by BroControl now (1) have the notice msg in the
    subject; and (2) come with the full mail.log entry in the body.
```

### Broccoli

```
1.97-14 | 2016-04-07 13:31:52 -0700

  * Fix some typos in the Broccoli user manual. (Daniel Thayer)

1.97-9 | 2016-03-04 12:37:43 -0800

  * Update for new CMake OpenSSL script (Johanna Amann)

  * Add README.rst -> README symlink. Addresses BIT-1413 (Johanna
    Amann)

1.97-3 | 2015-11-10 13:31:13 -0800

  * Fix to compile with OpenSSL that has SSLv3 disalbed. (Christoph
    Pietsch)

1.97 | 2015-05-07 11:48:17 -0700

  * Use @rpath in broccoli.dylib's install_name on OS X.

  * Fix a memory leak: table attributes weren't freed. (Jon Siwek)
```

```
1.96 | 2014-05-19 16:17:14 -0500

  * Remove code corresponding w/ Bro's unused Val::attribs. (Jon Siwek)

1.95-13 | 2013-12-09 13:23:48 -0800

  * Remove unused code in bro_vector_set_nth_val(). (Jon Siwek)

  * Fix memory leaks in relation to freeing BroVectors. (Jon Siwek)

1.95-10 | 2013-12-04 09:34:59 -0800

  * Update type serialization format/process to align with Bro's
    changes to preserve type name info and remove old compatibility
    stuff. (Jon Siwek)

1.95-3 | 2013-12-03 10:53:34 -0800

  * Add support for consuming events w/ vector args. (Jon Siwek)

    Producing events w/ vector args is still unsupported, and bindings
    are still missing support as well.

1.95 | 2013-11-06 00:23:50 -0800

  * Don't build ruby bindings by default, use --enable-ruby to do so.
    (Jon Siwek)

1.94 | 2013-10-24 16:49:30 -0700

  * Release.

1.93-17 | 2013-10-15 11:19:19 -0700

  * Fix a minor memory leak recently introduced. (Jon Siwek)

1.93-15 | 2013-10-14 14:20:20 -0700

  * Fix misc. issues reported by Coverity (Return value checks,
    time-of-check-time-of-use, null ptr checking, and a
    use-after-free). (Jon Siwek)

  * Fixed __bro_list_val_pop_front() to not erase the entire list but
    remove only the first element. (Jon Siwek)

  * Updating copyright notice. (Robin Sommer)

1.93-10 | 2013-10-02 10:38:27 -0700

  * Remove dead code. (Jon Siwek)

  * Fix mem leaks. (Jon Siwek)

  * Updated specfile and configure script for libdir.  (Derek Ditch)

    Package maintainers and those that would otherwise compile from
    source were unable to specify the installation directory of
    architecture dependent libraries. Namely, many distributions use
    lib64/ versus lib/ for the installation of architecture dependent
    library archives.

    * Add new 'configure' option, --libdir
    * Defaults to old behavior of "$prefix/lib"
    * Follows Kitware example for ProjectConfig.cmake on wiki

    See https://github.com/bro/broccoli/pull/1

  * Added back config-file bits to CMakeLists.txt (Derek Ditch)

  * Fix for setting REPO in Makefile. (Robin Sommer)

1.93 | 2013-09-23 20:21:20 -0700

  * Update 'make dist' target. (Jon Siwek)

  * Change submodules to fixed URL. (Jon Siwek)

  * Fix a compiler warning. (Daniel Thayer)

  * Fix a broken link in documentation. (Daniel Thayer)

  * Switching to relative submodule paths. (Robin Sommer)

  * s/bro-ids.org/bro.org/g. (Robin Sommer)

1.92-9 | 2013-01-31 12:17:39 -0800

  * A test program for sending packets through Broccoli, moved over
    from the Time Machine repository. (Seth Hall)

1.92-7 | 2012-12-20 12:13:39 -0800

  * Sync up with attribute definitions in Bro. (Daniel Thayer)

  * Rebuild only necessary files for new prefix. (Daniel Thayer)

1.92-4 | 2012-12-05 15:37:54 -0800

  * Improved error checking/reporting in case of out of memory
    situations. (Bill Parker)

1.92-3 | 2012-11-23 19:51:14 -0800

  * Bump data serialization format version for Bro's new "hook"
    function. (Jon Siwek)

1.92 | 2012-08-22 16:15:18 -0700

  * Fix configure script to exit with non-zero status on error (Jon Siwek)

1.91 | 2012-07-10 16:08:50 -0700

  * Add --conf-files-dir option to configure wrapper script. (Jon Siwek)

1.9 | 2012-07-05 12:59:54 -0700
```

```
  * Fix a warning, and fix other typos in documentation. (Daniel Thayer)

1.8-28 | 2012-05-24 17:37:35 -0700

  * Tweak a test script to register events with both IPv4 & IPv6
    loopback. (Jon Siwek)

  * BROCCOLI_CONFIG_FILE env. variable can now specify config file
    path. (Jon Siwek)

  * Add ability to connect to Bro peers over IPv6. (Jon Siwek)

1.8-23 | 2012-05-03 11:33:03 -0700

  * Fix typos and a few reST formatting problems. (Daniel Thayer)

1.8-21 | 2012-04-24 14:48:44 -0700

  * Add option to set 'etc' directory. Addresses #801. (Daniel Thayer)

  * Change BroAddr to use standard IPv4 in IPv6 mapping. (Jon Siwek)

    The size field is now removed and the bro_util_is_v4_addr()
    function can be used instead to check whether the BroAddr is IPv4
    or not. Addresses #800.

  * Add timeout to broccoli-v6addrs.c test. Addresses #793. (Jon
    Siwek)

  * Update IPv6 literal syntax in test scripts. (Jon Siwek)


1.8-8 | 2012-03-09 15:13:14 -0800

  * Bump data format version corresponding to Bro's removal of match
    expression. (Jon Siwek)

  * Adding missing include needed on FreeBSD. (Robin Sommer)

  * Update Broccoli library to handle IPv6 addrs/subnets. Addresses
    #448. Addresses now use a new BroAddr struct to hold the address
    data and BroSubnet changed to use a BroAddr member instead of a
    single uint32 to represent the address. (Jon Siwek)

  * Raise minimum required CMake version to 2.6.3. (Jon Siwek)

1.8 | 2012-01-10 19:33:08 -0800

  * Tweaks for OpenBSD support. (Jon Siwek)

1.71-26 | 2012-01-03 15:41:37 -0800

  * Remove record base type list since it's been removed from Bro.
    (Jon Siwek)

1.71-22 | 2011-12-03 15:58:34 -0800

  * Support for more types (not exposed at the API-level yet) to allow
```

```
    exchanging more complex record types. Adresses #606. (Christian
    Kreibich)

  * Broccoli now identifies itself as such when connecting to a peer.
    This allows Bro to adapt its serialization format based on what's
    supported by Broccoli. Adresses #606. (Christian Kreibich)

1.71-11 | 2011-11-07 05:44:16 -0800

  * Fixing compiler warnings. Addresses #388. (Jon Siwek)

  * Update broccoli-ruby submodule. (Jon Siwek)

  * Fix CMake warning when python bindings are disabled. Fixes #605.
    (Jon Siwek)

1.71 | 2011-10-27 17:42:45 -0700

  * Update submodules. (Jon Siwek)

1.7 | 2011-10-25 20:18:58 -0700

  * Make dist now cleans the copied source. (Jon Siwek)

  * Distribution cleanup. (John Siwek and Robin Sommer)

  * Changed communications protocol option to listen_ssl from
    listen_encrypted. (Seth Hall)

  * Bug fix for a Bro test. (Seth Hall)

  * Updates to make broccoli work with communication API updates.

1.6-35 | 2011-09-15 16:48:01 -0700

  * Adding Ruby bindings for Broccoli. (Seth Hall)

  * Broccoli API docs are now generated via Doxygen. Addresses #563.
    (Jon Siwek)

  * Converting manual to reST-format. (Don Appleman and Jon Siwek)

1.6-26 | 2011-09-04 09:26:47 -0700

  * FindPCAP now links against thread library when necessary (e.g.
    PF_RING's libpcap). (Jon Siwek)

  * Install binaries with an RPATH. (Jon Siwek)

  * Remove the 'net' type from Broccoli. Addresses #535. (Jon Siwek)

  * Workaround for FreeBSD CMake port missing debug flags. (Jon Siwek)

1.6-13 | 2011-08-08 16:18:24 -0700

  * Update broping.c test to use 64-bit int width for Bro counts (Jon Siwek)

  * Install example config files dynamically when the distribution
```

```
    version differs from existing version on disk. (Jon Siwek)

1.6-6 | 2011-07-19 17:54:57 -0700

  * Update broccoli tests scripts to use new Bro policy organization
    (Jon Siwek and Robin Sommer)

1.6 | 2011-05-05 20:32:42 -0700

  * Moving ChangeLog to CHANGES for consistency. (Robin Sommer)

  * Fixing write/read functionality for Bro's values that are now
    64-bit. (Jon Siwek)

  * Converting build process to CMake (Jon Siwek).

  * Import of Bro's aux/broccoli subdir from SVN r7107 (Jon Siwek)

===== Old Subversion ChangeLog starts here.

Wed Mar  2 15:38:02 PST 2011             Christian <christian@whoop.org>

- Accept empty strings ("") as values in the configuration file
  (Craig Leres).
- Support for specifying a separate host key for SSL-enabled operation,
  with documentation update (Craig Leres).
- Version bump to 1.5.3.

------------------------------------------------------------------------

Fri Oct  9 18:42:05 PDT 2009             Christian <christian@whoop.org>

- Version bump to 1.5.

------------------------------------------------------------------------

Fri Sep 25 10:09:03 PDT 2009             Christian <christian@whoop.org>

- Bropipe fixes: set a connection class for robustness reasons;
  removes some C/C++ confusion (Seth Hall).

------------------------------------------------------------------------

Mon Jun 29 17:56:00 PDT 2009             Christian <christian@whoop.org>

- SWIG bindings update.

------------------------------------------------------------------------

Mon Jun 29 15:29:35 PDT 2009             Christian <christian@whoop.org>

- Support for sending raw serialized events via the new API function
  bro_event_send_raw(), with much help from Matthias Vallentin.

------------------------------------------------------------------------

Mon Jun 29 15:20:58 PDT 2009             Christian <christian@whoop.org>
```

```
- Fix for buffered data remaining in transmit buffer when calling
  for_event_queue_flush().

- Added bro_conn_get_connstats() which reports statistical information
  about a connection in a new dedicated structure BroConnStats. For now
  this is only the amount of data buffered in the rx/tx buffers.

------------------------------------------------------------------------

Mon Jun 29 15:18:10 PDT 2009            Christian <christian@whoop.org>

- All multiprocess/-threading synchronization code has been removed.

------------------------------------------------------------------------

Mon Jun 29 15:10:59 PDT 2009            Christian <christian@whoop.org>

- Broccoli now requires initialization before any connections may be
  created. The reason is twofold: (i) it provides a clean method for
  initializing relevant parts of Broccoli in multithreaded environments,
  and (ii) it allows configuration of parts of Broccoli where the
  normal approach via configuration files is insufficient.

  For details on the initialization process, refer to the manual, but
  generally speaking, a call to

    bro_init(NULL);

  at the beginning of your application is all that is required. For the
  time being, a number of high-level API calls double-check whether you
  have called bro_init() previously.

- Broccoli now supports the callback functions OpenSSL requires for
  thread-safe operation.  Implement those callbacks as required by your
  threading library, hook them into a BroCtx structure previously
  initialized using bro_ctx_init(), and pass the structure to
  bro_init().  This will hook the callbacks into OpenSSL for you.

  O'Reilly's book "Network Security with OpenSSL" provides an example
  of how to implement the callbacks.

------------------------------------------------------------------------

Thu Jun 25 16:46:37 PDT 2009            Christian <christian@whoop.org>

- Fix to Python bindings: added required bro_init() call (Matthias
  Vallentin).

------------------------------------------------------------------------

Thu May 28 10:27:30 PDT 2009            Christian <christian@whoop.org>

- The BroEvMeta structure used in compact event callbacks now allows
  access to the timestamp of event creation.

------------------------------------------------------------------------

Fri Mar 27 23:39:10 CET 2009            Christian <christian@whoop.org>
```

```
- Fixed a memory leak triggered by bro_event_send() but actually caused
  by lack of cleanup after an underlying string duplication. Thanks to
  Steve Chan and Matthias Vallentin for helpful feedback.

------------------------------------------------------------------------

Wed Mar 25 11:26:16 CET 2009            Christian <christian@whoop.org>

Formatting robustness fixes to bropipe (Steve Chan).

------------------------------------------------------------------------

Thu Feb 12 19:28:24 PST 2009            Christian <christian@whoop.org>

- Updates to contributed bropipe command (Steve Chan):
  - Proper parsing of default host/port.
  - Support for "urlstring" type, which urlencodes spaces in strings
    and other special characters.

------------------------------------------------------------------------

Thu Dec 11 09:37:12 PST 2008            Christian <christian@whoop.org>

- Optimization: the internal slots vector of hashtables is now lazily
  allocated when the first actual insertion happens. Since hashtables
  are used in various places in the BroVal structures but frequently
  remain empty, the savings are substantial. Thanks to Matthias
  Vallentin for pointing this out.

------------------------------------------------------------------------

Mon Nov  3 11:07:49 PST 2008            Christian <christian@whoop.org>

- Fixes for I/O deadlocking problems:

  - A bug in the implementation of BRO_CFLAG_YIELD has been
    fixed. Input processing now only yields after the
    handshake is complete on *both* endpoints.

  - When events arrive during bro_conn_connect(), it could happen
    that deadlock ensues if no additional data are sent and
    __bro_io_process_input() can not read new input data. It no
    longer returns immediately in that case, and instead attempts
    to process any available input data.

------------------------------------------------------------------------

Sat Oct  4 15:05:07 CEST 2008           Christian <christian@whoop.org>

- Added bro_record_get_nth_name() to the API (Seth Hall).
- make install no longer worked for documentation, apparently as part
  of Bro's make install cleanup. This isn't quite right since gtk-doc
  documentation is normally installed in a well-known place and
  Broccoli will normally need to be installed via "make install", but
  for now I'm leaving it uninstalled and instead provide a specific
  "install-docs" target for people who want documentation installed.
- Documentation updated where missing, and rebuilt.
- Copyright years updated.

------------------------------------------------------------------------

Mon Sep 22 21:34:13 CEST 2008           Christian <christian@whoop.org>

- Updated broping.bro (and broping-record.bro, slightly) to explicitly
  declare the used event types ahead of their use.

------------------------------------------------------------------------

Mon Sep  8 11:30:35 CEST 2008           Christian <christian@whoop.org>

- Use of caching on received objects is now disabled by default, but can
  be enabled using the new connection flag BRO_CFLAG_CACHE.  The old
  BRO_CFLAG_DONTCACHE is kept for backward compatibility but no longer
  does anything. Keeping the caches between Bro instances and Broccoli
  synchronized still needs to be implemented completely, and in the
  meantime no caching is the safer default.  Thanks to Stephen Chan for
  helping track this down.

------------------------------------------------------------------------

Wed Jul 16 01:47:16 PDT 2008            Christian <christian@whoop.org>

- Python bindings for Broccoli are now provided in the bindings/python
  subdirectory (Robin Sommer).  They are not built automatically. See
  the instructions in bindings/python/README for details.
- Minor documentation setup tweaks.

------------------------------------------------------------------------

Thu May 15 14:05:10 PDT 2008            Christian <christian@whoop.org>

Event callbacks of the "compact" type are now able to obtain start- and
end pointers of the currently processed event in serialized form, from
the receive buffer stored with the connection handle.

------------------------------------------------------------------------

Wed Feb 20 13:53:51 PST 2008            Christian <christian@whoop.org>

- Fix to __bro_openssl_read(), which handled some error cases
  reported by BIO_read() incorrectly. (Robin Sommer)
- Clarifications to documentation of bro_conn_active() and
  bro_conn_process_input().
- Version bump to 1.4.0.

------------------------------------------------------------------------

Thu Sep 13 13:56:58 PDT 2007            Christian <christian@whoop.org>

- autogen.sh now uses --force when running libtoolize, which at least
  in some setups seems to be necessary to avoid bizarre build issues.
  (In the particular case encountered, these looked like run-together
  ar and runlib invocations). Thanks to Po-Ching Lin for helping nail
  this down.
```

```
-------------------------------------------------------------------------

Mon Sep 10 18:17:29 PDT 2007            Christian <christian@whoop.org>

- Broccoli now supports table and set container types. Have a look at
  the bro_table_...() and bro_set_...() families of functions in
  broccoli.h, the updated manual, and the updated broconn and brotable
  examples in the test/ directory.

-------------------------------------------------------------------------

Tue Sep  4 15:53:27 PDT 2007            Christian <christian@whoop.org>

- Major bugfix for capabilities exchange during handshake: Broccoli did
  not convert into NBO, causing successful event exchange to fail. :(
  Amazingly, this means disabling cache usage per Broccoli's request
  never worked...

-------------------------------------------------------------------------

Tue Sep  4 12:36:53 PDT 2007            Christian <christian@whoop.org>

- Changed the way compact argument passing to event callbacks works.
  All event metadata is now represented by a single argument, a pointer
  to a BroEvMeta structure. It contains the name of the event, the
  number of arguments, and the arguments along with their types.

  Updated documentation and broping demo accordingly.

  NOTE: This introduces an API incompatibility. If you were previously
        using the compact callback API, you will need to update your
        code! I bumped up the library version info to 2:0:0 to signal
        this.

- Fixed a bug in the implementation of BRO_CFLAG_YIELD and some SGML-
  violating documentation of same.

-------------------------------------------------------------------------

Thu Aug 16 15:24:51 CEST 2007           Christian <christian@whoop.org>

- Include autogen.sh in the distribution.

-------------------------------------------------------------------------

Sat Aug 11 04:59:35 PDT 2007                     Robin <robin@icir.org>

- New flag for Broccoli's connections: with BRO_CFLAG_YIELD,
  bro_conn_process_input() processes at most one event at a time and then
  returns (Robin Sommer).

- The new Broccoli function bro_conn_new_socket() creates a connection
  from an existing socket, which can then be used with listen()/accept()
  to have Broccoli listen for incoming connections (Robin Sommer).

-------------------------------------------------------------------------

Fri Jul  6 18:18:05 PDT 2007            Christian <christian@whoop.org>
```

```
- Bumped up the version number to 1.3. Now that Broccoli is bundled
  with Bro, it makes sense to keep Broccoli's release version number
  in synch with Bro's.
- Added the automake-provided ylwrap wrapper script to the distribution.
  This is for compatibility reasons: some automakes believe that
  Broccoli requires ylwrap, others don't. The distcheck target however
  needs ylwrap when it *is* required, so it's easiest to just provide
  one. It can always be overwritten locally, should the need arise.

------------------------------------------------------------------------

Wed Mar  7 10:49:25 PST 2007             Christian <christian@whoop.org>

- Data format version number bumped up, in sync with Bro again.

------------------------------------------------------------------------

Mon Dec  4 12:07:12 PST 2006             Christian <christian@whoop.org>

- Updated broconn.c to new bro_record_get_named_val().

------------------------------------------------------------------------

Tue Nov 28 11:16:04 PST 2006             Christian <christian@whoop.org>

- Run-time type information is now also available for the values stored
  in records (previously there was only type-checking, but no way to
  obtain the type of the vals). See the manual and API documentation of
  the functions bro_record_get_nth_val() and bro_record_get_named_val()
  for details.

------------------------------------------------------------------------

Mon Nov 27 18:38:06 PST 2006             Christian <christian@whoop.org>

- Compact argument passing for event callbacks: as an alternative to the
  argument passing style used so far for event callbacks (dubbed "expan-
  ded"), one can now request "compressed" passing by using the
  bro_event_registry_add_compact() variant. Instead of passing every
  event argument as a separate pointer, compact passing provides only
  the number of arguments, and a pointer to an array of BroEvArgs.
  The elements of this array then provide pointers to the actual argu-
  ments as well as pointers to the new BroValMeta metadata structure,
  which currently contains type information about the argument.

  This style is better suited for applications that don't know the type
  of events they will have to handle at compile time, for example when
  writing language bindings.

  broping.c features example code, also see the manual for detailed
  explanation.

------------------------------------------------------------------------

Mon Nov 27 16:32:52 PST 2006             Christian <christian@whoop.org>

- Bumped up version to 0.9
- I'm starting to use shared library version numbers to indicate API
  changes. Their correspondence to the release version number will be
  listed in VERSION.
- Fixed a warning in bro_packet.c

------------------------------------------------------------------------

Mon Nov 27 16:23:46 PST 2006            Christian <christian@whoop.org>

- Renamed cvs.pl to svn.pl
- Bumped up BRO_DATA_FORMAT_VERSION to 13, to match that of Bro trunk.

------------------------------------------------------------------------

Mon Nov 27 16:21:28 PST 2006            Christian <christian@whoop.org>

- Updating my commit script to SVN -- let's see if this works...

------------------------------------------------------------------------

Mon May 15 19:21:30 BST 2006            Christian <christian@whoop.org>

- Correction to the explanation of bro_event_registry_add(), pointed
  out by Robin Sommer.

------------------------------------------------------------------------

Mon May  8 08:14:31 PDT 2006            Christian <christian@whoop.org>

- Added config.sub and config.guess versions that seem to work well with
  MacOS X to the tree, to remove the dependency on the libtool/automake
  versions installed on the machine where tarballs are built.

- Removed -f from libtoolize invocation in autogen.sh, so we don't
  overwrite the above.

- Fixed COPYING, which wasn't actually referring to Broccoli. :)

------------------------------------------------------------------------

Sat May  6 20:17:32 BST 2006            Christian <christian@whoop.org>

- Last-minute tweaks bring last-minute brokenness, especially when
  configuring without --enable-debug... :(

------------------------------------------------------------------------

Tue May  2 13:25:31 BST 2006            Christian <christian@whoop.org>

- Added generated HTML documentation to CVS, so it is guaranteed to be
  included in tarballs generated via dist/distcheck, regardless of
  whether GtkDoc support exists on the build system or not.

------------------------------------------------------------------------

Tue May  2 02:31:39 BST 2006            Christian <christian@whoop.org>

- Changed connection setup debugging output to state more clearly
```

```
  whether an SSL or cleartext connection is attempted, as suggested
  by Brian Tierney.
- New configuration item /broccoli/use_ssl to enable/disable SSL
  connections, as suggested by Jason Lee. Documentation and sample
  configuration in broccoli.conf updated accordingly, look at the latter
  for a quick explanation.
- A bunch of small tweaks to get distcheck to work properly when invoked
  from the Bro tree.
- Other doc/Makefile.am cleanups.

------------------------------------------------------------------------

Sat Apr 29 19:12:07 PDT 2006             Christian <christian@whoop.org>

- Fixed bogusness in docs/Makefile.am's dist-hook target. Should now
  work much better in general, and in particular not bomb out with
  non-GNU make.

------------------------------------------------------------------------

Fri Apr  7 23:52:20 BST 2006             Christian <christian@whoop.org>

- Bumped up BRO_DATA_FORMAT_VERSION to 12, to match the one in Bro's
  CVS HEAD again.

------------------------------------------------------------------------

Mon Mar 27 22:59:04 BST 2006             Christian <christian@whoop.org>

- This should fix a memleak detected by Jim Mellander and reported with
  a test case by Mark Dedlow.

------------------------------------------------------------------------

Fri Mar  3 16:40:56 GMT 2006             Christian <christian@whoop.org>

- Warning for invalid permissions on ~/.broccoli.conf has been upgraded
  from debugging output to stderr, per request from Mark Dedlow.
- Only check validity of config file name assembled via getenv("HOME")
  if it yields a filename different from the one assembled via the
  passwd entry.

------------------------------------------------------------------------

Thu Mar  2 17:57:49 GMT 2006             Christian <christian@whoop.org>

- Reintroducing file needed for distcheck.

------------------------------------------------------------------------

Thu Mar  2 16:27:55 GMT 2006             Christian <christian@whoop.org>

- Debugging fixlet.

------------------------------------------------------------------------

Fri Feb  3 20:31:08 GMT 2006             Christian <christian@whoop.org>
```

```
- Embarrassing debugging output fixes.

------------------------------------------------------------------------

Fri Jan 27 23:40:23 GMT 2006            Christian <christian@whoop.org>

- Only do lock operations when there's any need for them.

------------------------------------------------------------------------

Fri Jan 27 18:30:06 GMT 2006            Christian <christian@whoop.org>

I am *so* fired. Overlooked a very clear warning that bro_io.c:lock()
wasn't returning a value.

------------------------------------------------------------------------

Wed Jan 18 10:45:33 GMT 2006            Christian <christian@whoop.org>

- Fixed call trace debugging inconsistencies, this will hopefully fix a
  case of runaway call trace indentation depth that Robin + Stefan have
  bumped into.

------------------------------------------------------------------------

Wed Jan  4 16:21:07 GMT 2006            Christian <christian@whoop.org>

- Documentation fixlet, pointed out by Stefan Kornexl.

------------------------------------------------------------------------

Thu Dec 22 00:48:20 GMT 2005            Christian <christian@whoop.org>

- Attempt at a more portable detecting of [g]libtoolize. Let me know if
  this works any better.

------------------------------------------------------------------------

Mon Dec 19 17:48:19 PST 2005            Christian <christian@whoop.org>

- Moved brosendpkts.c and rcvpackets.bro from test/ to contrib/, i.e.,
  out of the default build process. brosendpkts.c defines variables in
  the middle of main(), which some compilers tolerate while others
  don't. This should fix build issues reported by Brian Tierney.

------------------------------------------------------------------------

Thu Dec 15 18:38:18 GMT 2005            Christian <christian@whoop.org>

Configuration tweaks to run smoothly when invoked from a Bro build.

- Added AC_CONFIG_AUX_DIR(.) to make sure things are exclusively run
  out of our tree.
- Added flags to autogen.sh and configure.in to indicate that we're
  part of a Bro build.

------------------------------------------------------------------------
```

```
Fri Dec  2 14:04:05 GMT 2005           Christian <christian@whoop.org>

- Removed EXTRA_DIST for the test app policies, since they are included
  in the tarball and installed anyway via pkgdata_DATA.

------------------------------------------------------------------------

Fri Dec  2 13:59:27 GMT 2005           Christian <christian@whoop.org>

- Added "brosendpkts", a test program for sending pcap packets to a Bro,
  plus the accompanying Bro policy. Contributed by Stefan Kornexl and
  Robin Sommer, with a tiny tweak to build only when pcap support is
  available.

------------------------------------------------------------------------

Wed Nov 23 11:59:03 PST 2005           Christian <christian@whoop.org>

- Avoided the keyword "class" to prevent problems with using broccoli.h
  in a C++ context. Pointed out by Stefan Kornexl.

------------------------------------------------------------------------

Tue Nov  8 14:10:23 PST 2005           Christian <christian@whoop.org>

- Added support for connection classes, updated documentation.

------------------------------------------------------------------------

Mon Oct 31 19:37:55 PST 2005           Christian <christian@whoop.org>

- Support for specifying type names along with values. This is done
through a new and optional argument to bro_event_add_val(), bro_
record_add_val(), and friends. See manual for details.

- Added a test program "broenum" for demonstrating this. When running
Bro with the provided broenum.bro policy, it sends a single event with
an enum val to the remote Bro, which will print both numerical and
string representations of the value. For example, broenum.bro defines
an enum type

  type enumtype: enum { ENUM1, ENUM2, ENUM3, ENUM4 };

Given this,

  $ broenum -n 0              yields        Received enum val 0/ENUM1
  $ broenum -n 1              yields        Received enum val 1/ENUM2
  $ broenum -n 4              yields        Received enum val 4/<undefined>

You can also test predefined enums:

  $ broenum -t transport_proto -n 1

yields

  Received enum val 1/tcp

------------------------------------------------------------------------
```

```
Mon Oct 31 17:07:15 PST 2005             Christian <christian@whoop.org>

Changed commit script to pass the commit message through the generated
file via -F, instead of via -m and the command line. D'oh.

------------------------------------------------------------------------

Mon Oct 31 17:03:47 PST 2005             Christian <christian@whoop.org>

- Support for the new abbreviated serialization format for types. Need
to come up with a decent API for actually using this feature now.

------------------------------------------------------------------------

Mon Oct 31 11:25:22 PST 2005             Christian <christian@whoop.org>

Several changes to handshake implementation and API(!).

- Refactored the handshake code to make the multiple phases of the
connection's initialization phase more explicit. Our own and the peer's
handshake state are now tracked separately. conn_init_configure() takes
care of our state machine with a separate function per phase, and
__bro_io_process_input() handles the peer's state.

- Added support for capabilities. The only capability Broccoli currently
supports is a non-capability: it can ask the remote Bro not to use the
serialization cache. In order to do so, pass BRO_CONN_DONTCACHE as
a connection flag when obtaining the connection handle. Needs more
testing.

- Several API changes. Given the more complex handshake procedure that
is in place now, the old approach of only completing the handshake half-
way in bro_connect() so the user can requests before calling
bro_conn_await_handshake() (or alternatively, passing
BRO_CONN_COMPLETE_HANDSHAKE as a connection flag) is just too messy now.
The two steps of obtaining a connection handle and establishing a
connection have been split into separate functions, so the user can
register event handlers in between.

What was

 BroConn *bc = bro_connect(..., BRO_CFLAGS_NONE);

 bro_event_registry_add(bc,...);
 bro_event_registry_add(bc,...);
 bro_event_registry_request(bc);

 bro_conn_await_handshake(bc);
 /* ... */
 bro_disconnect(bc);

is now

 BroConn *bc = bro_conn_new(..., BRO_CFLAGS_NONE);

 bro_event_registry_add(bc,...);
 bro_event_registry_add(bc,...);
 bro_conn_connect(bc);
 /* ... */
 bro_conn_delete(bc);

Note that the explicit call to bro_event_registry_request() is gone as
bro_conn_connect() will automatically request event types for which
handlers have been installed via bro_event_registry_add(). What was

 BroConn *bc = bro_connect(..., BRO_CFLAGS_COMPLETE_HANDSHAKE);
 bro_disconnect(bc);

is now

 BroConn *bc = bro_conn_new(..., BRO_CFLAGS_NONE);
 bro_conn_connect(bc);
 /* ... */
 bro_conn_delete(bc);

I might add bro_conn_disconnect() in the near future. It'd allow us
to keep a completely configured connection handle around and use it
repeatedly for establishing connections.

Sorry for the inconvenience but I really think this is a lot nicer than
the old API. The examples and documentation have been updated accor-
dingly.

------------------------------------------------------------------------

Sat Oct 29 15:43:18 PDT 2005            Christian <christian@whoop.org>

Added an optional age list to the hash table implementation. We'll
need this to duplicate Bro's object serialization caching strategy.

------------------------------------------------------------------------

Fri Oct 28 15:26:55 PDT 2005            Christian <christian@whoop.org>

Brothers and sisters, hallelujah! On the 27th day Christian looked at
record vals in the Broccoli, and he saw that it was leaking like a
sieve. So Christian ran the valgrind. On the 28th day Christian still
looked at Broccoli, with tired eyes, ground the vals[1] a bit more,
and he saw that it was plugged[2].

Amen. :)

[1] Really really bad pun. Sorry.
[2] I get zero memleaks on broping -r -c 100 now. :)

------------------------------------------------------------------------

Thu Oct 27 20:02:39 PDT 2005            Christian <christian@whoop.org>

First crack at reference-counted sobjects. I need reference counting
in order to get rid of objects in the serialization cache (since they
can contain nested objects etc -- it's nasty), which I had ignored so
far. There are still leaks in the event transmission code, dammit. :(
------------------------------------------------------------------------

Thu Oct 27 15:06:10 PDT 2005             Christian <christian@whoop.org>

Added my own list implementation due to suckiness of the TAILQ_xxx
macro stuff which I never liked anyway. The problem is that elements
of lists built using these macros can only have each member exactly
once as the prev/next pointers are part of the structs.

A few uses of TAILQ_xxx remain, these will go in the near future.

------------------------------------------------------------------------

Tue Oct 25 19:57:42 PDT 2005             Christian <christian@whoop.org>

Partial support for enum vals, per request from Weidong. Sending enum
vals should work, though the underlying enum types aren't fully handled
yet.

------------------------------------------------------------------------

Mon Oct 24 16:31:56 PDT 2005             Christian <christian@whoop.org>

TODO item: clean up generated parser/lexer files when we know we can
regenerate them. make clean currently does not erase them, which caused
Weidong some trouble.

------------------------------------------------------------------------

Fri Oct 21 17:48:51 PDT 2005             Christian <christian@whoop.org>

Clarification to the manual, after a question from Weidong.

------------------------------------------------------------------------

Fri Oct 14 18:05:39 PDT 2005             Christian <christian@whoop.org>

Transparent reconnects should work again (took all *day*, argh -- I
totally broke it with the connection sharing stuff). Try broping while
occasionally stopping and restarting the Bro side.

Fixed a number of memleaks -- broping is now leak-free according to
valgrind.

Clarifications in the debugging output.

------------------------------------------------------------------------

Fri Oct 14 12:07:10 PDT 2005             Christian <christian@whoop.org>

Added documentation for the new user data argument to
bro_event_registry_add().

------------------------------------------------------------------------

Fri Oct 14 11:48:00 PDT 2005             Christian <christian@whoop.org>

Added user data to event handler callbacks. This is necessary for
```

```
example when using class members in C++ as callbacks since the object
needs to be provided at the time of dereferencing. It's also easier to
use than the existing bro_conn_{set,get}_data() mechanism.

Updated documentation with more details on the broccoli-config script.

------------------------------------------------------------------------

Thu Oct 13 15:08:56 PDT 2005            Christian <christian@whoop.org>

When supporting packets (the default), check whether pcap.h actually
exists. This has thus far just been assumed. We don't actually use
the library, so there's no need to test for it.

------------------------------------------------------------------------

Mon Oct 10 20:37:15 PDT 2005            Christian <christian@whoop.org>

Changed bro_record_get_named_val() and bro_record_get_nth_val() to
return a pointer to the queried value directly, instead of through
a pointer argument. These arguments' type used to be void* though it
should really be void**, but switching to void** causes lots of warnings
with current GCCs ('dereferencing type-punned pointer will break
strict-aliasing rules'). NULL is perfectly usable as an error indicator
here, and thus used from now on. Updated manual, broping, and broconn
accordingly.

------------------------------------------------------------------------

Tue Sep 20 17:19:58 PDT 2005            Christian <christian@whoop.org>

Fixed a varargs buglet that is tolerated on Linux but not BSD. Pointed
out by Scott Campbell.

------------------------------------------------------------------------

Fri Sep  9 18:48:54 PDT 2005            Christian <christian@whoop.org>

Support for textual tags on packets, also an upgrade to more complex
handshake procedure that allows for synchronization of state (Robin
Sommer).

Note: as of this change, at least Bro 1.0a2 is required.

------------------------------------------------------------------------

Wed Aug 10 01:36:47 BST 2005            Christian <christian@whoop.org>

Fixed my insecure usage of snprintf.

------------------------------------------------------------------------

Tue Jul 19 10:11:49 PDT 2005            Christian <christian@whoop.org>

Forgot to include broconn's policy file in the distribution.

------------------------------------------------------------------------
```

```
Mon Jul 18 16:34:22 PDT 2005            Christian <christian@whoop.org>

Fixed a bug that caused the lookup of record fields by name to fail.

------------------------------------------------------------------------

Fri Jul  1 00:44:49 BST 2005            Christian <christian@whoop.org>

The sequence of tests determining which config file to read from
failed to fall back properly to the global config file in case of
incorrect user permissions. Fixed.

------------------------------------------------------------------------

Mon Jun 27 19:34:56 PDT 2005            Christian <christian@whoop.org>

Added bro_buf_reset() to the user-visible API.

------------------------------------------------------------------------

Mon Jun 27 17:58:53 PDT 2005            Christian <christian@whoop.org>

When a configuration item cannot be found in the current config file
section, a lookup is also attempted in the default section (the one
at the top of the file, before any sections are defined). This allows
the sections to override the default section, which is what one would
expect.

------------------------------------------------------------------------

Mon Jun 27 14:43:56 PDT 2005            Christian <christian@whoop.org>

Debugging output tweak. When providing the SSL cert passphrase via
the config file, do no longer report it in the debugging output.

------------------------------------------------------------------------

Mon Jun 27 12:33:52 PDT 2005            Christian <christian@whoop.org>

Cosmetics in the debugging output of __bro_openssl_write().

------------------------------------------------------------------------

Fri Jun 24 18:13:49 PDT 2005            Christian <christian@whoop.org>

Added --build flag to broccoli-config. It reports various details
about the build, for example whether debugging support was compiled in.

------------------------------------------------------------------------

Fri Jun 24 10:37:23 PDT 2005            Christian <christian@whoop.org>

I'm adding a little test app that subscribes to a few connection
events and prints out the fields of the received connection records,
both for testing and code demonstration purposes. So far it has
highlighted a bug in Bro that occurs when a remote app is a pure
requester of events and not sending anything. Fix pending.
```

```
------------------------------------------------------------------------

Mon Jun 20 18:21:24 PDT 2005            Christian <christian@whoop.org>

Show the names of requested events in the debugging output -- it
had to be deciphered from the hex string which isn't that much fun.

------------------------------------------------------------------------

Thu Jun 16 14:02:59 PDT 2005            Christian <christian@whoop.org>

Better documentation of how to extract record fields.

------------------------------------------------------------------------

Thu Jun 16 11:51:02 PDT 2005            Christian <christian@whoop.org>

- Added bro_string_get_data() and bro_string_get_length() to avoid
making people access BroString's internal fields directly.

- Moved BroString's internal storage format to uchar*.

------------------------------------------------------------------------

Sun Jun 12 19:17:31 PDT 2005            Christian <christian@whoop.org>

Debugging output now shows the correct function and line numbers again.
I had accidentially moved __FUNCTION__ and __LINE__ into bro_debug.c :(

------------------------------------------------------------------------

Fri Jun  3 15:00:48 PDT 2005            Christian <christian@whoop.org>

I broke the sanity checks for semaphore initialization when I moved
the semaphore structures to shared memory. Fixed.

------------------------------------------------------------------------

Mon May 16 22:25:41 PDT 2005            Christian <christian@whoop.org>

- Debugging output now goes to stderr instead of stdout. That keeps it
out of the way if an instrumented app dups() stdout to another file
descriptor.
- Debugging output is now disabled by default (even when compiled in),
so it needs to be enabled explicitly in the code or in the config file.

------------------------------------------------------------------------

Fri May 13 18:24:23 PDT 2005            Christian <christian@whoop.org>

Synchronization fixes and minor cleanups.

- Unsuccessful connection attempts to remote Bros in combination with
connection sharing caused the caller to hang indefinitely. This should
now be fixed, but required some fairly intricate tweaks to the locking
constructs. Still needs more testing.

- Bumped version to 0.8.
```

```
-------------------------------------------------------------------------

Fri May  6 23:09:29 BST 2005            Christian <christian@whoop.org>

This is the 0.7.1 release.

-------------------------------------------------------------------------

Fri May  6 14:44:53 PDT 2005            Christian <christian@whoop.org>

Documentation for shareable connection handles.

-------------------------------------------------------------------------

Fri May  6 12:11:17 PDT 2005            Christian <christian@whoop.org>

Build fixlets.

- Don't only test for the first of the documentation extraction tools,
but also for those used later on.

- Few more signedness warnings fixed.

-------------------------------------------------------------------------

Wed May  4 18:33:40 PDT 2005            Christian <christian@whoop.org>

Fixed a whole bunch of signedness warnings reported by gcc 4 on MacOS
10.4. Thanks to Roger for the quick reply.

-------------------------------------------------------------------------

Wed May  4 17:41:40 PDT 2005            Christian <christian@whoop.org>

Fix for a little-endian bug that I managed to introduce when testing on
Solaris ... *sigh* :(

-------------------------------------------------------------------------

Wed May  4 17:30:07 PDT 2005            Christian <christian@whoop.org>

A number of portability fixes after testing the build on Linux, FreeBSD
and Solaris.

-------------------------------------------------------------------------

Mon May  2 20:17:04 PDT 2005            Christian <christian@whoop.org>

Fixed an obvious bug the config file parser. I'm baffled as to how it
could go unnoticed for so long.

-------------------------------------------------------------------------

Mon May  2 20:11:25 PDT 2005            Christian <christian@whoop.org>

Portability fixes.
- Use -pthread (not -lpthread) in both the --cflags and --libs options
to broccoli-config, if required. -lpthread does not work on BSDs, where
-pthread has different effects on the linker.

- s/System V/SYSV/ in configure script output for consistency.

- Bumped version to 0.7.1.

It should build correctly on BSDs and Linux now. Still need to check
whether synchronization actually works on the BSDs.

------------------------------------------------------------------------

Fri Apr 29 23:12:01 BST 2005            Christian <christian@whoop.org>

If the configure script determines we need -lpthread, it's a good idea
to actually reflect that in broccoli-config.

------------------------------------------------------------------------

Fri Apr 29 22:36:26 BST 2005            Christian <christian@whoop.org>

Fix for SYSV semaphores pointed out by Craig Leres -- I completely
forgot to test the SYSV stuff before the release. *sigh*.

------------------------------------------------------------------------

Thu Apr 28 13:46:57 BST 2005            Christian <christian@whoop.org>

- This is the 0.7 release.

------------------------------------------------------------------------

Thu Apr 28 13:43:44 BST 2005            Christian <christian@whoop.org>

RPM spec file fixlet.

------------------------------------------------------------------------

Wed Apr 27 18:04:57 BST 2005            Christian <christian@whoop.org>

Preparations for the 0.7 release.

------------------------------------------------------------------------

Wed Mar 16 18:34:27 GMT 2005            Christian <christian@whoop.org>

I think shared connections w/ SSL work. :) They key aspects are

- We want to be able to use a single connection handle in arbitrary
process/thread scenarios: in sshd, a single handle created in the
listening process should work in all forked children (right now I'm
created separate ones in each child, yuck), in Apache it should work
in all servicing threads (creating a separate connection in each
servicing thread would be far too costly), etc.

- However, all SSL I/O on a single BIO must happen in the same *thread*
according to openssl-users -- same process seems intuitive because of
```

```
cipher streams etc; why it's per thread I don't know.

The approach is now as follows: when a connection handle is marked as
shareable, an I/O handler process is forked off during handle setup
that processes all I/O for a single connection handle exclusively.
Data are processed through separate tx/rx buffers that live in shared
memory and are protected by semaphores. Additionally, a number of
fields in the connection handle also live in shared memory so can be
used to send back and forth messages etc. By using global semaphores as
condition variables, rx/tx requests are dispatched to the I/O handler
process. Therefore this should work for all multi-process/thread
scenarios in which processes/threads are created after the connection
handle is set up.

This all is transparent when a connection is not marked shareable. The
main optimization left to do now is to make the locking more fine-
grained -- a throughput comparison is going to be interesting...

I haven't tried transparent reconnects again; I'd presume I managed
to break them in the process.

------------------------------------------------------------------------

Mon Mar 14 17:31:17 GMT 2005            Christian <christian@whoop.org>

- Lots of work on shared connection handles. This is going to take a
while to work robustly. For now steer clear of BRO_CFLAG_SHAREABLE.

- Fixed wrong ordering of semaphore locks in __bro_io_msg_queue_flush().

- The connection hack to work around OpenSSL's 'temporary unavailable'
beliefs is now only used when the problem occurs, namely during
reconnects.

- Fixed a bug in the Posix version of __bro_sem_new() that prevented
processes from creating more than one different semaphores. Doh.

- Bumped BRO_DATA_FORMAT_VERSION to 9, to sync up with Bro tree.

- Added __bro_sem_get(), returning the current value of a sempahore,
with implementations for Posix + SYSV.

- Lots of calltracing added.

------------------------------------------------------------------------

Mon Mar 14 10:24:54 GMT 2005            Christian <christian@whoop.org>

Code for shared connection handles with SSL enabled. Pretty much done,
but needs a lot of testing now.

------------------------------------------------------------------------

Sat Mar 12 18:13:58 GMT 2005            Christian <christian@whoop.org>

Beginning of support for sharing connection handles for SSL-enabled
connections. Since supporting this is complex, it will be optional,
and enabled by using the new BRO_CFLAG_SHAREABLE connection flag.
```

```
------------------------------------------------------------------------

Fri Mar 11 14:50:23 GMT 2005            Christian <christian@whoop.org>

Move to AC_PROG_LIBTOOL.

------------------------------------------------------------------------

Fri Mar 11 14:33:57 GMT 2005            Christian <christian@whoop.org>

Portability and robustness fixes.

- auto* calls in autgen.sh are now checked for success and cause the
script to abort on error.
- Instead of trying to figure out what libraries the various OSs need
in order to be able to use Posix semaphors, I'm now attempting to use
the -pthread flag directly. If that fails, we just fall back to SYSV
semaphores.
- All semaphore + shmem implementations are now included in the tarball,
the point is to include them selectively in the *build*.
- Stevens' ifdef magic for union semun doesn't work on at least OpenBSD
so I'm using the BSD_HOST macro from config.h now.
- Apparently AM_PROG_LIBTOOL causes some people trouble so we need to
check how to get that working realiably :(

------------------------------------------------------------------------

Mon Feb 21 14:45:51 GMT 2005            Christian <christian@whoop.org>

- Partial-write bugfix. When we succeed only partially in writing out
a message, report success, not failure. Failure is handled by queuing
the message for later transmission, but we have already sent it
partially and the rest is still stuck in the output buffer, so if we
queue it again, it'll get sent at least twice.

I had noticed that out of 100000 events sent by 100 processes in
parallel, typically around 100020 arrived :)

------------------------------------------------------------------------

Sat Feb 19 21:04:46 GMT 2005            Christian <christian@whoop.org>

- Lots of synchronization work. This generally seems to work now! :) It
required one major addition: support for shared memory. The problem is
that if multiple threads/processes attempt to write at the same time
and one write succeeds only partially, then *whichever* thread/process
gets to write next needs to write out the rest before writing any new
messages. The only alternative is to have write operations block until
entire messages are sent, which seems dangerous from an instrumentation
point of view. To share the remaining message data, shared memory is
required: both the tx and rx buffers now operate in shared memory and
are protected by semaphores. The current implementation uses SYSV shared
memory.

I think shared memory is a good idea in general; for example it could be
used during instrumentation to get information from one corner of an app
to another without changing the application's structure. I don't think
```

```
we'll need  this right away, but it's nice to have a possible technique
for it.

- bro_disconnect() is now more tricky to use than before: if you use
it in a parallel setting, you *must* call it from the same process that
called bro_connect() and you must do so *after* all the other processes
have finished using the connection (typically this is not hard to do, so
I think we can live with that).

The reason is that semaphores + shared memory need to be uninstalled
specifically and I haven't yet figured out a way to automate reference
counting so that the last thread/process using a connection could do
this automatically. It would be very cool if the functions that are
used for deinstallation could be asked to fail while the IPC objects are
still in use, but that's not the case.

- You can still build the whole thing without semaphores or shared mem
and it'll work for single-threaded apps. The configure script now issues
a warning if not all tools required for stable parallel operation can be
found.

- Added bro_event_queue_length_max() to allow applications to find out
the maximum queue length before messages will get dropped. brohose uses
this to wait until the queue gets half full before insisting on a flush.

------------------------------------------------------------------------

Fri Feb 18 17:14:40 GMT 2005            Christian <christian@whoop.org>

- SYSV semaphore implementation. Configure checks are included
and work as follows: if both Posix + SYSV semaphores are found,
Posix are preferred, however the user can override this by passing
--disable-posix-semaphores. Semaphores are still not actually used.


------------------------------------------------------------------------

Thu Feb 17 22:24:12 GMT 2005            Christian <christian@whoop.org>

- First shot at semaphore support. Checking for Posix named semaphores
and making sure they actually work at configure time was the hardest
part; actual semaphore code untested and still unused. No ifdefs
anywhere :)

------------------------------------------------------------------------

Thu Feb 17 20:06:00 GMT 2005            Christian <christian@whoop.org>

- Incompletely sent chunks are now recognized and remaining parts are
shipped as soon as possible: repeated brohose -n 1 -e 1000 runs do not
take out Bro any more. :)

------------------------------------------------------------------------

Thu Feb 17 19:21:15 GMT 2005            Christian <christian@whoop.org>

- Added "brohose", which lets you hose a Bro with events by forking a
configurable number of processes, and having each process pump out an
```

```
event a configurable number of times as fast as possible. This is meant
as both a stress-testing tool for the protocol as well as obviously for
the synchronization stuff that'll go into Broccoli soon.

------------------------------------------------------------------------

Wed Feb 16 17:40:47 GMT 2005             Christian <christian@whoop.org>

- Documentation for the configuration options for debugging output.

------------------------------------------------------------------------

Thu Feb 10 11:39:57 GMT 2005             Christian <christian@whoop.org>

- Changed bro_event_queue_empty() to bro_event_queue_length(),
which is more useful in general and can be used to find out
whether the queue is empty, too.

------------------------------------------------------------------------

Tue Feb  8 14:45:58 GMT 2005             Christian <christian@whoop.org>

- This is release 0.6.

------------------------------------------------------------------------

Mon Feb  7 14:54:15 GMT 2005             Christian <christian@whoop.org>

- Additional byte swaps for IP addresses + subnets for compatibility
with Bro.

------------------------------------------------------------------------

Sun Feb  6 23:55:07 GMT 2005             Christian <christian@whoop.org>

- Debugging output can now be configured from the config file,
using the /broccoli/debug_messages and /broccoli/debug_calltrace
config items.

------------------------------------------------------------------------

Tue Feb  1 21:34:17 GMT 2005             Christian <christian@whoop.org>

- During handshake, data format compatibility is now confirmed as well
as matching protocol version.

------------------------------------------------------------------------

Tue Feb  1 21:04:43 GMT 2005             Christian <christian@whoop.org>

- Initial commit of support for sending/receiving libpcap packets.
Totally untested, and not documented yet. More on this once support
for packets is committed into the Bro tree.

------------------------------------------------------------------------

Tue Feb  1 18:39:02 GMT 2005             Christian <christian@whoop.org>
```

```
- Transparent reconnects now also work for non-SSL connections. I was
just lucky that the SSL handshake prevented the same problem from
occurring in the SSL-enabled case. Two fixes were necessary:

 1) a separate attempt to connect to the peer that I have full control
    over, and
 2) a fixlet in queue management that caused the event that
    triggers the reconnect to be sent before any handshake information
    for the new connection, thus causing a connection teardown by the
    Bro end because the version number was not seen at the right time.


------------------------------------------------------------------------

Mon Jan 31 19:38:36 GMT 2005             Christian <christian@whoop.org>

- Fixed a few spots where D_ENTER was not balanced with D_RETURN
- Added an int-to-string table for message types, for debugging
- Added a flag to the connection structure that prevents reconnect
attempts while one is already in progress
- Made io_msg_queue() private to bro_io.c because it was only called
from there.

------------------------------------------------------------------------

Fri Jan 28 12:35:03 GMT 2005             Christian <christian@whoop.org>

- Changed the error semantics of in __bro_io_msg_queue() so that queuing
a message after failure to send is not a failure. This fixes an issue
with handshake completion that I have observed with broping across
different machines, where events could still get lost despite explicit
request to complete the handshake.

------------------------------------------------------------------------

Sun Jan 16 20:45:42 GMT 2005             Christian <christian@whoop.org>

- Serialization/Unserialization for ports fixed, support for ICMP ports.

------------------------------------------------------------------------

Sat Jan 15 13:58:16 GMT 2005             Christian <christian@whoop.org>

- Sending and receiving IP addresses and subnets was broken, fixed now.
- Fixed a small memleak when first-time connection setup fails.

------------------------------------------------------------------------

Thu Jan 13 21:03:45 GMT 2005             Christian <christian@whoop.org>

- When using reconnects, Broccoli will now not attempt to reconnect
more than once every 5s.

------------------------------------------------------------------------

Thu Jan 13 20:43:13 GMT 2005             Christian <christian@whoop.org>

- Added connection flag BRO_CFLAG_ALWAYS_QUEUE that causes events
```

```
always to be queued in the connection's event queue regardless of
whether the peer is currently dead or not.

- Moved the test of whether the peer requested an event that is
about to be sent or not to the point where the event actually is
about to be sent, from the point where it is requested to be sent.
The difference is that now an event will get silently dropped on
the floor if after a connection outage and a reconnect, a change
in the events requested from the peer will prevent the old queued
events to be sent anyway, even if they are no longer requested.

------------------------------------------------------------------------

Wed Jan 12 20:46:10 GMT 2005              Christian <christian@whoop.org>

- Added support for transparent reconnects for broken connections.
When using BRO_CFLAG_RECONNECT, Broccoli now attempts to reconnect
whenever a peer died and the user tries to read from or write to
the peer. This can aways be triggered manually using
bro_reconnect().

- Added bro_conn_alive() to determine if a connection is currently
alive or not.

------------------------------------------------------------------------

Tue Jan 11 17:33:51 GMT 2005              Christian <christian@whoop.org>

- Added connection flags parameter to bro_connect() and
bro_connect_str(): BRO_CFLAG_COMPLETE_HANDSHAKE completes
the handshake right away before returning from bro_connect()/
bro_connect_str(), and BRO_CFLAG_RECONNECT still needs to be
implemented. Documentation updated accordingly.

------------------------------------------------------------------------

Sat Jan  8 21:07:30 CET 2005              Christian <christian@whoop.org>

- Allow empty (or comments-only) configuration files.

------------------------------------------------------------------------

Sat Jan  8 20:52:56 CET 2005              Christian <christian@whoop.org>

- Fixed the home directory lookup via getpwent() -- now correctly looks
up the entry of the current effective user. Doh.

- Beginning of code for connection flags to use when creating a
connection, for example for handshake behaviour, automatic reconnection
attempts, etc.

------------------------------------------------------------------------

Tue Jan  4 23:28:59 CET 2005              Christian <christian@whoop.org>

- constness fixes for functions that accept values for events and
record fields.
```

```
------------------------------------------------------------------------

Tue Jan  4 22:07:35 CET 2005            Christian <christian@whoop.org>

- Encrpyted connections now extract as much data as possible from
the underlying buffer by calling BIO_read() optimistically.

- For encrypted connections, the passphrase for the certificate's
private key can now be specified in the configuration file using key
"/broccoli/host_pass".

- Added support for the handshake message in the Bro protocol.

- If the ca_cert or host_cert keys are found in the config file, but
there is a problem loading the crypto files, don't attempt to connect.

- Completed documentation on encrypted communication, explaining the
use of ca-create and ca-issue.

- Fixed several bugs in the handling of sections in config files.
Matching of domain names is now case-insensitive.

- The ~/.broccoli.conf file is now only used when it is readable only
by the user owning it.

- More robustness for corner cases of buffer sizes.

- Fixed a bug in sending messages that consist of only a single chunk
(like the handshake message).

- The library now attempts to initialize the random number generator
in OpenSSL from /dev/random if possible.

------------------------------------------------------------------------

Fri Dec 24 11:58:08 CET 2004            Christian <christian@whoop.org>

- If the ca_cert or host_cert keys are found in the config file, but
there is a problem loading the crypto files, don't attempt to connect.

- Completed documentation on encrypted communication, explaining the
use of ca-create and ca-issue.

- Fixed several bugs in the handling of sections in config files.

------------------------------------------------------------------------

Thu Dec 23 14:33:56 GMT 2004            Christian <christian@whoop.org>

- Added sections support for configuration files. Sections can be
declared at arbitrary points in the config file, using the same syntax
as in OpenSSL config files. There can be a global section at the
beginning of the file, before the first declared sections. Sections are
selected using bro_conf_set_domain().

- Support for a per-user config file in ~/.broccoli.conf. This does
not override settings in the global config file but completely replaces
it, i.e., when the user-specific file is found, the global one is
```

```
ignored.

- Added bro_conn_await_handshake() that blocks for limitable amount of
time, waiting for the handshake of a new Bro connection to complete.
This still needs some fixing, but is definitely necessary to prevent
weird races from occurring when a client tries to use a new connection
that has not yet been established completely.

- Test applications are now linked to static libraries. This will
hopefully keep the build more portable.

- Use of LFLAGS and YFLAGS moved to AM_LFLAGS and AM_YFLAGS, given the
warnings issued when using automake 1.9.

- First shot at fixing the buffer flushing issues I see when using
encrypted connections.

------------------------------------------------------------------------

Fri Dec 10 16:31:26 GMT 2004            Christian <christian@whoop.org>

- Added + fixed OpenSSL code to support encrypted communication.
- Added OpenSSL as requirement to spec file.
- Changed broping policies to always use the same port
- Updated broccoli.conf: added keys for the CA's and the host's cert.

------------------------------------------------------------------------

Thu Dec  9 14:59:24 GMT 2004            Christian <christian@whoop.org>

- Build fixes in case documentation tools are not found
- Documentation polishing -- only SSL setup section todo still.

------------------------------------------------------------------------

Thu Dec  9 00:48:05 GMT 2004            Christian <christian@whoop.org>

- Final documentation passes for the 0.6 release.

------------------------------------------------------------------------

Mon Dec  6 17:18:55 GMT 2004            Christian <christian@whoop.org>

- More documentation, explaining the data types, records, Bro policy
configuration, started section on SSL setup (copied from Robin right
now), and minor fixes.

------------------------------------------------------------------------

Mon Dec  6 15:17:05 GMT 2004            Christian <christian@whoop.org>

- Added spec file for building RPMs -- seems to work
- Aest policies are now installed in $prefix/share/broccoli

------------------------------------------------------------------------

Mon Dec  6 00:22:02 GMT 2004            Christian <christian@whoop.org>
```

```
- Dropped the ..._raw() functions for records. These won't be used
internally ever. Their implementation moved to bro.c, and only the high-
level code remained in bro_record.c.

- Added bro_event_set_val() to replace a val in an existing event.
There's not much use in resending an existing event unless it is
identical, which is not that useful. High-level code is in
__bro_event_set_val().

- Made it more clear in the comments explaining the
bro_record_get_..._val() functions that the "result" argument must
actually be the address of a pointer. (void * as argument type means
that the compiler does not issue a warning when passing in, say, a
double * -- but it would do so if we would use void **.)

------------------------------------------------------------------------

Sun Dec  5 22:05:53 GMT 2004            Christian <christian@whoop.org>

- Updates to the cvs wrapper script: surround with date and name
only in the ChangeLog, not in the commit message itself.

------------------------------------------------------------------------

Sun Dec  5 02:15:29 GMT 2004            Christian <christian@whoop.org>

- Fixed a bug in __bro_val_clone(): forgot to handle BRO_INTTYPE_OTHER.

- Changed --enable-debugging flag to --enable-debug, for consistency
with the Bro tree.

- Fixed bugs in several cloning implementations that didn't call the
parent's implementation.

------------------------------------------------------------------------

Sun Dec  5 01:40:52 GMT 2004            Christian <christian@whoop.org>

- Added __bro_event_copy() to clone events internally.

- Events are now duplicated in __bro_io_event_queue() before they're
sent so the user's event remains unaffected (and thus could be sent
repeatedly etc).

- Extensive pass over the documentation; still a good deal to do.

------------------------------------------------------------------------

Sat Dec  4 03:09:05 GMT 2004            Christian <christian@whoop.org>

More work on documentation, much is outdated now.

------------------------------------------------------------------------

Sat Dec  4 02:05:30 GMT 2004            Christian <christian@whoop.org>

- Started a ChangeLog. No detailed ChangeLog information was kept
previous to this commit.
```

```
-------------------------------------------------------------------------
```

### Broccoli Python

```
0.59 | 2015-04-27 08:25:18 -0700

  * Release 0.59

0.58-9 | 2015-02-13 18:01:05 -0600

  * Install broccoli_intern.py (Jon Siwek)

0.58-8 | 2015-02-13 16:02:49 -0600

  * In broccoli.py, import from broccoli_intern not _broccoli_intern.
    (Jon Siwek)

0.58-1 | 2014-07-08 09:56:42 -0700

  * Fix setup.py to work with path changes. Addresses BIT-1213.
    (Nicholas Weaver)

0.58 | 2014-04-03 15:53:50 -0700

  * Release 0.58

0.57-3 | 2014-01-23 16:59:19 -0800

  * Supply connAlive() and connDelete() methods. (jpohlmann)

0.57 | 2013-11-06 00:21:37 -0800

  * Installation section was missing necessary steps. (Bernhard Amann)

0.56 | 2013-10-14 09:24:55 -0700

  * Updating copyright notice. (Robin Sommer)

0.55-2 | 2013-10-02 10:34:29 -0700

  * Fix mem leaks. (Jon Siwek)

0.55 | 2013-09-23 13:14:46 -0500

  * Change submodules to fixed URL. (Jon Siwek)

  * Switching to relative submodule paths. (Robin Sommer)

  * s/bro-ids.org/bro.org/g. (Robin Sommer)

0.54 | 2012-08-01 13:56:08 -0500

  * Fix configure script to exit with non-zero status on error (Jon Siwek)
```

```
0.53 | 2012-06-11 17:25:05 -0700

  * Fix overflow problems in converting Python IP addresses to
    Broccoli. (Jon Siwek)

  * Update bindings to work with Broccoli's IPv4-mapped BroAddrs. (Jon
    Siwek)

  * Update IPv6 literal syntax in test scripts. (Jon Siwek)

  * Update broccoli-python for IPv6 addr/subnet support. Addresses
    #448. (Jon Siwek)

  * Raise minimum required CMake version to 2.6.3. (Jon Siwek)


0.52 | 2012-01-10 16:56:13 -0800

  * Submodule README conformity changes (Jon Siwek)

  * Simplify finding of Python headers/libraries. Addresses #666 (Jon
    Siwek)

0.51-3 | 2011-11-03 15:17:19 -0700

  * Fixing compiler warnings. Addresses #388. (Jon Siwek)

0.51 | 2011-10-27 17:41:32 -0700

  * Compile SWIG bindings with no-strict-aliasing. Addresses #644.
    (Jon Siwek)

0.5 | 2011-10-25 20:18:20 -0700

  * Make dist now cleans the copied source. (Jon Siwek)

  * Add configure-time check that swig can generate python wrappers.
    Addresses #642. (Jon Siwek)

  * Updates for changes to communication API. (Seth Hall)

  * Distribution cleanup. (Jon Siwek and Robin Sommer)

  * Install binaries with an RPATH (Jon Siwek)

  * Remove the 'net' type from Broccoli python bindings. Aaddresses
    #535).

  * Workaround for FreeBSD CMake port missing debug flags. (Jon Siwek)

  * Adjust how python-broccoli test script prints floats.

  * Allow record instances that don't initialize all fields.  (Jon
    Siwek)

  * Update tests w/ example of sending a partial records. (Jon Siwek)

  * Fix pybroccoli record instantiation. Declaring more than one
```

```
    record_type could cause it to break because only the last-declared
    record_type was used in the instantiation.  (Jon Siwek)

  * Change to fill in record field names. Before, it was sending
    records with hardcoded "<unknown>" field names. (Jon Siwek)

  * Adating CMake's include path based on output of python-config.
    (Robin Sommer)

  * Teaching CMake to use python-config for finding libraries. (Robin
    Sommer)

  * Making python-broccoli work with 64-bit integers. (Robin Sommer)

0.4 | 2011-07-19 17:54:35 -0700

  * Improvements and fixes to record implementation. (Jon Siwek)

      * Allow record instances that don't initialize all fields.
      * Update tests w/ example of sending a partial records.
      * Fix pybroccoli record instantiation.
      * Change pybroccoli to fill in record field names.

  * Update test bro script for new Bro policy script organization. (Jon Siwek)

  * Cleanup (Seth Hall, Robin Sommer)

0.3 | 2011-05-05 20:42:47 -0700

  * CMake build system. (Jon Siwek).

  * Adapting to work with 64-bit integers, which Bro and Broccoli are
    now using. (Robin Sommer)

0.2
    - Repository switched to git, and README converted to reSt.
    - License changed to BSD-style.
```

### Broccoli Ruby

```

1.58 | 2015-04-27 08:25:18 -0700

  * Release 1.58

1.57 | 2014-04-03 15:53:50 -0700

  * Release 1.57

1.56 | 2013-10-14 09:24:55 -0700

  * Updating copyright notice. (Robin Sommer)

  * Fix for setting REPO in Makefile. (Robin Sommer)

1.55 | 2013-09-23 14:42:03 -0500
```

```
  * Update 'make dist' target. (Jon Siwek)

  * Change submodules to fixed URL. (Jon Siwek)

  * Switching to relative submodule paths. (Robin Sommer)

  * s/bro-ids.org/bro.org/g. (Robin Sommer)

1.54 | 2012-08-01 13:56:22 -0500

  * Fix configure script to exit with non-zero status on error (Jon Siwek)

1.53 | 2012-06-11 17:25:05 -0700

  * Update bindings to work with Broccoli's IPv4-mapped BroAddrs. (Jon Siwek)

  * Fix count/enum being treated same as addr. (Jon Siwek)

  * Update broccoli-ruby for IPv6 addr/subnet support. Addresses #448.
    (Jon Siwek)

  * Raise minimum required CMake version to 2.6.3 (Jon Siwek)

1.52 | 2012-01-09 16:11:01 -0800

  * Submodule README conformity changes (Jon Siwek)

1.51-10 | 2011-11-07 05:44:17 -0800

  * Ignoring some SWIG warnings. Addresses #388. (Jon Siwek)

  * Changes to broccoli-ruby installation scheme. Fixes #652.

    - `--home` and `--prefix` configure options are now respected when
      installing as the main CMake project.  If not given, the Ruby

    - When being installed as a CMake sub-project, then the
      "home"-style installation is performed. (Jon Siwek)

1.51 | 2011-10-26 13:51:22 -0700

  * Compile SWIG bindings with no-strict-aliasing (addresses #644).
    (Jon Siwek)

1.5 | 2011-10-25 17:41:31 -0700

  * Make dist now cleans the copied source. (Jon Siwek)

  * Add configure-time check that swig can generate Ruby wrappers.
    Addresses #642. (Jon Siwek)

  * Distribution cleanup. (Robin Sommer)

  * Updates to work with communication API changes. (Seth Hall)

  * Reorganized the module names.  From ruby, a user now loads the
    "broccoli" module.  This automatically pulls in the swig wrapper
    named "broccoli_ext".  (Seth Hall)

  * Building with cmake completely works now. (Seth Hall)

  * Updates for the change to 64-bit ints. (Seth Hall)

  * Fixes for the example script. (Seth Hall)

  * New example script that points out a bug in broccoli. (Seth Hall)

  * Remove the 'net' type from Broccoli ruby bindings. Addresses #535.
    (Jon Siwek)

  * Install binaries with an RPATH (Jon Siwek)

1.4 | 2011-02-25 21:26:49 -0500

  * Cleaning up and adding a configure script. (Seth Hall)

  * Ruby 1.8 is now required. (Seth Hall)

  * CMake fixes. (Seth Hall and Jon Siwek)

  * Initial import. (Seth Hall)
```

### Capstats

```

0.22 | 2015-04-27 08:25:19 -0700

  * Release 0.22

0.21 | 2014-04-03 15:53:51 -0700

  * Release 0.21

0.20 | 2013-10-14 09:24:55 -0700

  * Release.

0.19-4 | 2013-10-07 17:07:40 -0700

  * Fix getopt_long() usage. (Daniel Thayer)

  * Updating copyright notice. (Robin Sommer)

  * Fix for setting REPO in Makefile. (Robin Sommer)

0.19 | 2013-09-23 20:22:43 -0700

  * Update 'make dist' target. (Jon Siwek)

  * Correct a few errors in the README. (Daniel Thayer)
  * s/bro-ids.org/bro.org/g. (Robin Sommer)

0.18 | 2012-08-01 13:57:19 -0500

  * Fix configure script to exit with non-zero status on error (Jon Siwek)

0.17 | 2012-07-05 12:53:52 -0700

  * Raise minimum required CMake version to 2.6.3 (Jon Siwek)

0.16 | 2012-01-09 16:11:02 -0800

  * Submodule README conformity changes. (Jon Siwek)

  * Fix parallel make portability. (Jon Siwek)

0.15 | 2011-10-25 17:41:31 -0700

  * Make dist now cleans the copied source (Jon Siwek)

0.14-26 | 2011-10-14 15:09:34 -0700

  * Distribution cleanup. (Robin Sommer)

0.14-25 | 2011-10-14 15:06:20 -0700

  * Distribution cleanup. (Jon Siwek and Robin Sommer)

  * config.h wasn't being configured by CMake correctly (Jon Siwek)

  * Adding 'C' language to CMake project as some configure checks depend on it. (Jon␣
↪Siwek)

0.14-14 | 2011-09-04 08:55:48 -0700

  * FindPCAP now links against thread library when necessary (e.g.
    PF_RING's libpcap). (Jon Siwek)

  * Install binaries with an RPATH. (Jon Siwek)

  * Workaround for FreeBSD CMake port missing debug flags. (Jon Siwek)

0.14-5 | 2011-07-24 08:28:06 -0700

  * Fixing memory initialization error. (David Binderman)

0.14-3 | 2011-03-14 17:41:03 -0700

  * CMake tuning. (Jon Siwek)

0.14-2 | 2011-02-01 12:38:34 -0800

  * Linking directly to the found pcap library. (Jon Siwek)

0.14 | 2011-01-28 11:09:04 -0800

  * Fix for compiling on OpenBSD and NetBSD. (Kevin Lo)
  * Converting README from AsciiDoc to REST. (Robin Sommer)

  * Ported to CMake. (Jon Siwek)

  * New CHANGES format.

0.13
  * Do not output anything to syslog if not explicitly enabled.

  * New option -q <n> suppresses all normal output but exits with
    status code 0 if at least n packets have been received, and 1
    otherwise.

0.12

  * New option --select/-N which for live pcap input uses select() to
    check for new packets. This is primarily for testing purposes and
    shouldn't produce any different results. The code mimics pretty
    closely how the Bro NIDS uses select() on pcap file handles.

  * Fix disabling option -d when compiled without DAG support. (Justin
    Azoff)

  * autotools compatibility fixes.

  * Man page contributed by Justin Azoff.

0.11

  * Fixed potential segfault.

0.1

  * Initial release
```

### Trace-Summary

```

0.84-16 | 2016-05-17 16:21:13 -0700

  * Adjust IP address column widths as needed for IPv6 addrs to
    improve readability of the output. Addresses BIT-1571. (Daniel
    Thayer)

  * Add README.rst -> README symlink. Addresses BIT-1413 (Johanna
    Amann)

0.84-2 | 2015-08-18 07:54:36 -0700

  * Fix typo in a TEST_DIFF_CANONIFIER script name. (Daniel Thayer)

0.84 | 2015-04-27 08:25:19 -0700

  * Release 0.84

0.83-19 | 2015-03-06 14:52:27 -0800
```

```
  * Update code to work with Python 3. Bump minimum required Python
    version to 2.6. (Daniel Thayer)

  * Fix timestamps to not loose precision unnecessarily. (Daniel
    Thayer)

  * Add more error checks so errors are reported more clearly. (Daniel
    Thayer)

  * Add regression tests. (Daniel Thayer)

0.83-9 | 2014-12-08 13:54:39 -0800

  * Add man page for trace-summary. (Raúl Benencia)

0.83 | 2014-04-03 15:53:51 -0700

  * Release 0.83

0.82 | 2013-10-14 09:24:55 -0700

  * Updating copyright notice. (Robin Sommer)

0.81 | 2013-09-23 20:24:46 -0700

  * Fixing sampling in pcap mode. (Robin Sommer)

  * s/bro-ids.org/bro.org/g (Robin Sommer)

0.8 | 2012-07-05 12:54:50 -0700

  * Fix typos. (Daniel Thayer)

  * trace-summary now works with IPv6 traffic. It needs a current
    pysubnettree for that. (Daniel Thayer)

  * Raise minimum required CMake version to 2.6.3. (Jon Siwek)

0.73 | 2012-01-09 16:11:02 -0800

  * Submodule README conformity changes. (Jon Siwek)

0.72 | 2011-10-25 17:57:00 -0700

  * New make dist/distclean targets. (Jon Siwek)

  * Adding executable permission back to script. (Robin Sommer)

  * Cleaning up the distribution. (Robin Sommer)

  * Updating README (Jon Siwek)

0.71-19 | 2011-09-08 12:52:20 -0700

  * Now ignoring all lines starting with a pound Closes #602. (Robin
    Sommer)
  * Install binaries with an RPATH (Jon Siwek)

0.71-16 | 2011-08-03 16:18:15 -0700

  * Switching to new update-changes script. (Robin Sommer)

0.71-15 | 2011-08-03 16:02:14 -0700

  * trace-summary now parses both Bro 1.x and 2.x conn.log formats.
    The default setting is to make an educated guess at the format,
    but can be explicitly set via the new --conn-version switch. (Jon
    Siwek)

0.71-6 | 2011-03-14 17:41:05 -0700

  * CMake tweaks. (Jon Siwek)

  * Prettyfing the message about sampling being in effect.

0.71-3 | 2011-01-15 14:14:07 -0800

  * Updating update-changes. (Robin Sommer)

  * Let CMake infer install prefix (Jon Siwek)

  * Add warning when building and installing are done by different users (Jon Siwek)

0.71-1 | 2011-01-04 19:02:06 -0800

  * Tweaking update-changes. (Robin Sommer)

0.71 | 2011-01-04 18:36:36 -0800

  * Better error message when missing Python package. (Jon Siwek)

  * Better error message if ipsumdump not installed. (Jon Siwek)

  * Migrated from os.popen (deprecated since Python 2.6) to
    subprocess.Popen (available since Python 2.4). (Jon Siwek)

  * Switch to CMake-based installation (Jon Siwek)

0.7
    - Repository switched to git, and README converted to reSt.

    - Sample factor now included in output.

    - Bugfix: Sample factor was not applied to local subnets
      break-down.

0.6
    License changed to BSD-style.

0.5
    First release.
```

## BinPAC

```
0.44-24 | 2016-08-02 11:08:13 -0700

  * Fix memory leak in pac_parse.yy. (Bryon Gloden)

0.44-21 | 2016-06-14 17:41:28 -0700

  * Bug fix for pac_swap function with int32 type of argument.
    (Bartolo Otrit)

0.44-18 | 2016-05-23 08:25:49 -0700

  * Fixing Coverity warning. (Robin Sommer)

0.44-17 | 2016-05-06 16:52:37 -0700

  * Add a comment in the generated C++ code for fall through in
    switch. Coverity raised an error about this. (Vlad Grigorescu)

0.44-11 | 2016-03-04 12:36:57 -0800

  * Update for new CMake OpenSSL script. (Johanna Amann)

0.44-7 | 2016-01-19 10:05:37 -0800

  * Fixed compiler complaining about recursive function. (Seth Hall)

0.44-3 | 2015-09-11 12:24:21 -0700

  * Add README.rst symlink. Addresses BIT-1413 (Vlad Grigorescu)

0.44 | 2015-04-27 08:25:17 -0700

        * Release 0.44.

0.43-8 | 2015-04-21 20:11:06 -0700

  * Adding missing include. (Robin Sommer)

0.43-7 | 2015-04-21 13:45:20 -0700

  * BIT-1343: Extend %include to work with relative paths. (Jon Siwek)

0.43-5 | 2015-04-09 12:09:04 -0700

  * BIT-1361: Improve boundary checks of records that use &length.
    (Jon Siwek)

0.43 | 2015-01-23 09:56:59 -0600

  * Fix potential out-of-bounds memory reads in generated code.
    CVE-2014-9586.  (John Villamil and Chris Rohlf - Yahoo Paranoids,
    Jon Siwek)

0.42-9 | 2014-11-03 10:05:17 -0600
```

```
  * Separate declaration of binpac::init from definition. (Jon Siwek)

0.42-6 | 2014-10-31 17:42:21 -0700

  * Adding a new binpac::init() function that must be called by the
    host before anything else. Internally, this function compiles all
    regular expressions, avoiding to do that inside the regexp
    constructor. (Robin Sommer)

0.42 | 2014-04-08 15:24:11 -0700

  * Release 0.42.

0.41-5 | 2014-04-08 15:23:48 -0700

  * Request format macros from inttypes.h explicitly. This helps
    ensure the availability of PRI* macros from .pac files, which
    cannot create this definition themselves since the inclusion of
    binpac.h is hardcoded to be placed very early in the generated
    code and already includes inttypes.h itself. (Jon Siwek)

0.41 | 2013-10-14 09:24:54 -0700

  * Updating copyright notice. (Robin Sommer)

0.4-5 | 2013-10-02 10:33:05 -0700

  * Fix uninitialized (or unused) fields. (Jon Siwek)

  * Generate initialization code for external types. Numeric/pointer
    types can be initialized to 0. (Jon Siwek)

  * Optimize negative string length check. (Jon Siwek)

  * Fix for setting REPO in Makefile. (Robin Sommer)

0.4 | 2013-09-23 20:56:19 -0700

  * Update 'make dist' target. (Jon Siwek)

  * Change submodules to fixed URL. (Jon Siwek)

  * Add virtual dtor to RefCount base class. (Jon Siwek)

0.34-24 | 2013-09-12 15:49:51 -0500

  * Add missing break to switch statement case. (Jon Siwek)

  * Remove unreachable code. (Jon Siwek)

  * Add missing va_end()'s to match va_start()'s. (Jon Siwek)

  * Fix two use-after-free bugs. (Jon Siwek)

  * Fix double-free. (Jon Siwek)

  * Remove some answers from the Q&A section of README (Daniel Thayer)
  * Add BinPAC documentation from the old Bro wiki (Daniel Thayer)

0.34-11 | 2013-07-24 18:35:28 -0700

  * Adding an interface to manually control the buffering for
    generated parsers. (Robin Sommer)

    This consists of two parts:

        1. The generated Flow classes expose their flow buffers via a new
           method flow_buffer().

        2. Flow buffers get two new methods:

            // Interface for delayed parsing. Sometimes BinPAC doesn't get the
            // buffering right and then one can use these to feed parts
            // individually and assemble them internally. After calling
            // FinishBuffer(), one can send the uppper-layer flow an FlowEOF()
            // to trigger parsing.
            void BufferData(const_byteptr data, const_byteptr end);
            void FinishBuffer(); (Robin Sommer)

0.34-8 | 2013-04-27 15:04:23 -0700

  * Fix an exception slicing issue in binpac generated cleanup code.
    (Jon Siwek)

  * s/bro-ids.org/bro.org/g (Robin Sommer)

0.34-3 | 2012-11-13 17:24:24 -0800

  * Add scoping to usages of binpac::Exception classes in generated
    code. This allows analyzers to define their own types of the same
    name without mistakingly overshadowing the usages of
    binpac::Exception and its derived types in the generated parser
    code. (Jon Siwek)

0.34 | 2012-08-01 13:54:39 -0500

  * Fix configure script to exit with non-zero status on error (Jon
    Siwek)

0.33 | 2012-07-24 09:05:37 -0700

  * Silence warning for generated code when compiling with clang.
    (Robin Sommer)

0.32 | 2012-06-11 17:25:04 -0700

  * Change binpac.h integral typedefs and reimplement 64-bit
    pac_swap(). Addresses #761. (Jon Siwek)

  * Adding int64 and uint64 types to binpac. (Seth Hall)

  * Raise minimum required CMake version to 2.6.3 (Jon Siwek)

0.31 | 2012-01-09 16:11:01 -0800
```

```
  * Submodule README conformity changes. (Jon Siwek)

  * Fix parallel make portability. (Jon Siwek)

0.3 | 2011-10-25 17:41:31 -0700

  * Change distclean to only remove build dir. (Jon Siwek)

  * Make dist now cleans the copied source. (Jon Siwek)

  * Distribution cleanup. (Jon Siwek and Robin Sommer)

  * Arrays now suport the &transient attribute.

    If set, parsed elements won't actually be added to the array, and
    read access to the array aren't permitted. This is helpful to save
    memory in the case of large arrays for which elements don't need
    (or can't) be buffered. (Robin Sommer)

  * Install binaries with an RPATH. (Jon Siwek)

  * Workaround for FreeBSD CMake port missing debug flags. (Jon Siwek)


0.2 | 2011-04-18 12:50:21 -0700

  * Converting build process to CMake (Jon Siwek).

  * Fixing crash with undefined case expressions. (Robin Sommer)

    Found by Emmanuele Zambon.

  * A command line -q flag to quiet the output, plus a fix for a small
    compiler warning. (Seth Hall)

  * Initial import of Bro's binpac subdirectory from SVN r7088. (Jon Siwek)
```

### Bro-Aux

```

0.35-27 | 2016-06-21 18:31:33 -0700

  * Fix bro-cut to allow unset or zero time values. (Daniel Thayer)

  * Fix failure to build plugins on OS X 10.11 with init-plugin.
    (Daniel Thayer)

0.35-18 | 2016-03-04 12:38:16 -0800

  * Update for new CMake OpenSSL script. (Johanna Amann)

0.35-15 | 2016-02-01 12:37:46 -0800

  * Fix the init-plugin script to be more portable. (Daniel Thayer)

0.35-8 | 2015-08-10 14:56:24 -0700
```

```
  * Plugin skeletons now include a __preload__.bro that pulls in
    types.bro for defining types. (Robin Sommer)

0.35-7 | 2015-08-10 12:58:35 -0700

  * Fix a test for large time values that fails on some systems.
    (Daniel Thayer)

0.35-6 | 2015-08-06 22:29:36 -0400

  * Improved handling of malformed input, avoiding crashes. (Justin
    Azoff and Daniel Thayer)

  * Remove unused code and fix initialization of long_opts. (Daniel
    Thayer)

0.35-4 | 2015-07-21 09:38:58 -0700

  * Bringing back the ``--help`` option for bro-cut. (Justin Azoff)

0.35-2 | 2015-07-10 07:14:52 -0700

  * Add more documentation for bro-cut. (Daniel Thayer)

0.35 | 2015-06-03 09:02:49 -0700

  * Release 0.35.

0.34-5 | 2015-06-03 09:02:10 -0700

  * Fix replace_version_in_rst function in update-changes script to
    cope with "beta" in version string. (Daniel Thayer)

  * Portability fix for plugin configure script. (Daniel Thayer)

  * Fix minor typo in init-plugin error message. (Daniel Thayer)

0.34 | 2015-05-07 20:30:43 -0700

  * Release 0.34.

  * Change make-release to assume sign-file is in path (Johanna Amann)

0.33-76 | 2015-04-27 08:23:18 -0700

  * Fix sed regex for replacing version in header file. (Jon Siwek)

0.33-74 | 2015-04-23 06:58:37 -0700

  * Correct a few typos in update-changes script. (Daniel Thayer)

  * Adding function to update-changes that updates version in a C
    header file. (Robin Sommer)

  * Fix plugin configure skeletons to work on more shells. (Jon Siwek)

0.33-68 | 2015-02-23 11:26:14 -0600
```

```
  * Plugin skeleton updates. (Robin Sommer)

    - Updating plugin skeleton license.

    - Removing the plugin MAINTAINER skeleton file.

    - Adding hooks to configure script so that plugins can add options
      without modifying the scripts itself.

    - BIT-1302: Extending plugin skeleton Makefile to reload cached
      CMake variables when Bro has been reconfigured. (Robin Sommer)

    - Removing bdist and sdist make targets. The former is superseded by
      the new build process which always creates a binary distribution
      tarball. The latter is easy enough to do manually now that all
      dynamic stuff is in build/

    - Added a VERSION file; content goes into name of the binary tarball

    - Move README.edit-me to README.

    - Allowing relative paths for --bro-dist

  * Changing init-plugin to take an additional parameter specifying the
    directory where to create the plugin skeleton. (Robin Sommer)

0.33-58 | 2015-02-12 12:15:39 -0600

  * Fix bro-cut compile warning on FreeBSD (Johanna Amann)

0.33-56 | 2015-01-08 13:06:36 -0600

  * Increase minimum required CMake version to 2.8. (Jon Siwek)

0.33-55 | 2014-12-08 13:49:37 -0800

  * Add man page for bro-cut. (Raúl Benencia)

  * Add --install-root to plugin skeleton's configure. (Robin
    Sommer)

  * Fix get-bro-env script to use sh equality operator. (Jon Siwek)

  * Add an option to update-changes that prevents it from adding
    author names to entries. (Robin Sommer)

0.33-45 | 2014-08-21 15:47:29 -0500

  * Various tweaks to the plugin skeleton. (Robin Sommer)

0.33-38 | 2014-08-01 14:03:49 -0700

  * bro-cut has been rewritten in C, and is hence much faster. (Daniel
    Thayer, based on an initial version by Justin Azoff).

0.33-26 | 2014-07-30 15:51:42 -0500
```

```
  * Remove a superfluous file from plugin skeleton. (Jon Siwek)

  * init-plugin now creates a Plugin.h as well. (Robin Sommer)

  * Adding a basic btest setup to the plugin skeleton. (Robin Sommer)

  * Updating plugin skeleton to new API. (Robin Sommer)

  * Updates to the init-plugin script/skeleton. (Robin Sommer)

  * A script to setup a skeleton for a new dynamic plugin. (Robin Sommer)

0.33-11 | 2014-07-08 20:42:32 -0700

  * Add more tests of bro-cut. (Daniel Thayer)

  * Fix bug in bro-cut when duplicate fields are specified. (Daniel Thayer)

  * Fix bug in bro-cut when log file has missing field. (Daniel Thayer)

  * Fix bug in bro-cut output of "#types" header line. (Daniel Thayer)

  * Fix bug in bro-cut when separator is not hexadecimal. (Daniel Thayer)

  * Adding test target to top-level Makefile. (Robin Sommer)

0.33-4 | 2014-06-26 17:31:25 -0700

  * Test-suite for bro-cut. (Daniel Thayer)

0.33-2 | 2014-06-26 17:27:09 -0700

  * Change bro-cut UTC options to not always override local time.
    (Daniel Thayer).

  * Updated the bro-cut usage message to make it more clear that the
    BRO_CUT_TIMEFMT environment variable affects only the -u and -d
    options. (Daniel Thayer).

0.33 | 2014-05-08 16:27:10 -0700

  * Release 0.33.

0.32-5 | 2014-05-08 16:25:55 -0700

  * Adding git-move-submodules scriptm, which moves all submodules to
    the head of a given branch and adapts parent modules
    correspondingly. (Robin Sommer)

0.32-4 | 2014-04-22 21:34:23 -0700

  * A git hook script to prevent pushs when the external test suite has
    new commits pending. (Robin Sommer)

0.32 | 2013-11-01 05:24:56 -0700

  * Extending Mozialla cert script to include source URL and copyright
    in output. (Robin Sommer)
```

```
0.31 | 2013-10-14 09:24:54 -0700

  * Release.

0.3-5 | 2013-10-07 17:19:14 -0700

  * Fix for release script. (Robin Sommer)

  * Updating copyright notice. (Robin Sommer)

0.3-3 | 2013-09-28 11:17:42 -0700

  * Don't show error message in bro-cut when gawk not found, which
    could appear on some systems. (Daniel Thayer)

0.3-1 | 2013-09-24 13:41:02 -0700

  * Fix for setting REPO in Makefile, and some tweaks to release
    scripts. (Robin Sommer)

0.3 | 2013-09-23 14:42:56 -0500

  * Update 'make dist' target. (Jon Siwek)

  * Change submodules to fixed URL. (Jon Siwek)

  * make-release nows ignores modules that aren't tagged for release
    or beta. (Robin Sommer)

  * Prettyfing check-release output. (Robin Sommer)

  * Update gen-mozilla-ca-list.rb to retrieve the Mozilla
    root CA list from a current url. (Bernhard Amann)

0.26-25 | 2013-09-18 14:44:35 -0700

  * A set of README updates, including installation instructions and
    description of bro-cut. (Daniel Thayer)

  * Switching to relative submodule paths. (Robin Sommer)

0.26-21 | 2013-08-19 11:21:11 -0700

  * Fixing git-show-fastpath handling of non-existing fastpath
    branches. (Robin Sommer)

0.26-19 | 2013-07-31 20:09:52 -0700

  * Making git-show-fastpath save against repositories that don't have
    a fastpath. (Robin Sommer)

0.26-16 | 2013-05-17 07:45:24 -0700

  * A negate option -n for bro-cut printing all fields *except* those
    listed on the command-line. (Robin Sommer)

0.26-14 | 2013-03-22 12:17:54 -0700
```

```
  * Fixing bro-cut to work with older gawk versions. (Chris Kanich)

  * s/bro-ids.org/bro.org/g (Robin Sommer)

0.26-5 | 2012-11-01 14:24:25 -0700

  * Portability fix: removing interface option on non-Linux. (Robin Sommer)

0.26-4 | 2012-10-31 14:39:03 -0700

  * rst learns a new option "-i <if>" to set the interface to use.
    (Vlad Grigorescu).

0.26 | 2012-08-24 15:10:04 -0700

  * Fixing update-changes, which could pick the wrong control file. (Robin Sommer)

  * Fixing GPG signing script. (Robin Sommer)

0.25 | 2012-08-01 13:55:46 -0500

  * Fix configure script to exit with non-zero status on error (Jon Siwek)

0.24 | 2012-07-05 12:50:43 -0700

  * Raise minimum required CMake version to 2.6.3 (Jon Siwek)

  * Adding script to delete old fully-merged branches. (Robin Sommer)

0.23-2 | 2012-01-25 13:24:01 -0800

  * Fix a bro-cut error message. (Daniel Thayer)

0.23 | 2012-01-11 12:16:11 -0800

  * Tweaks to release scripts, plus a new one for signing files.
    (Robin Sommer)

0.22 | 2012-01-10 16:45:19 -0800

  * Tweaks for OpenBSD support. (Jon Siwek)

  * bro-cut extensions and fixes.  (Robin Sommer)

    - If no field names are given on the command line, we now pass through
      all fields. Adresses #657.

    - Removing some GNUism from awk script. Addresses #653.

    - Added option for time output in UTC. Addresses #668.

    - Added output field separator option -F. Addresses #649.

    - Fixing option -c: only some header lines were passed through
      rather than all. (Robin Sommer)

  * Fix parallel make portability. (Jon Siwek)
```

```
0.21-9 | 2011-11-07 05:44:14 -0800

  * Fixing compiler warnings. Addresses #388. (Jon Siwek)

0.21-2 | 2011-11-02 18:12:13 -0700

  * Fix for misnaming temp file in update-changes script. (Robin Sommer)

0.21-1 | 2011-11-02 18:10:39 -0700

  * Little fix for make-release script, which could pick out the wrong
    tag. (Robin Sommer)

0.21 | 2011-10-27 17:40:45 -0700

  * Fixing bro-cut's usage message and argument error handling. (Robin Sommer)

  * Bugfix in update-changes script. (Robin Sommer)

  * update-changes now ignores commits it did itself. (Robin Sommer)

  * Fix a bug in the update-changes script. (Robin Sommer)

  * bro-cut now always installs to $prefix/bin by `make install`. (Jon Siwek)

  * Options to adjust time format for bro-cut. (Robin Sommer)

    The default with -d is now ISO format. The new option "-D <fmt>"
    specifies a custom strftime()-style format string. Alternatively,
    the environment variable BRO_CUT_TIMEFMT can set the format as
    well.

  * bro-cut now understands the field separator header. (Robin Sommer)

  * Renaming options -h/-H -> -c/-C, and doing some general cleanup.

0.2 | 2011-10-25 19:53:57 -0700

  * Adding support for replacing version string in a setup.py. (Robin
    Sommer)

  * Change generated root cert DN indices format for RFC2253
    compliance. (Jon Siwek)

  * New tool devel-tools/check-release to run before making releases.
    (Robin Sommer)

  * devel-tools/update-changes gets a new option -a to amend to
    previous commit if possible. Default is now not to (used to be the
    opposite). (Robin Sommer)

  * Change Mozilla trust root generation to index certs by subject DN. (Jon Siwek)

  * Change distclean to only remove build dir. (Jon Siwek)

  * Make dist now cleans the copied source (Jon Siwek)
  * Small tweak to make-release for forced git-clean. (Jon Siwek)

  * Fix to not let updates scripts loose their executable permissions.
    (Robin Sommer)

  * devel-tools/update-changes now looks for a 'release' tag to
    idenfify the stable version, and 'beta' for the beta versions.
    (Robin Sommer).

  * Distribution cleanup. (Robin Sommer)

  * New script devel-tools/make-release to create source tar balls.
    (Robin Sommer)

  * Removing bdcat. With the new log format, this isn't very useful
    anymore. (Robin Sommer)

  * Adding script that shows all pending git fastpath commits. (Robin
    Sommer)

  * Script to measure CPU time by loading an increasing set of
    scripts. (Robin Sommer)

  * extract-conn script now deals wit *.gz files. (Robin Sommer)

  * Tiny update to output a valid CA list file for SSL cert
    validation. (Seth Hall)

  * Adding "install-aux" target. Addresses #622. (Jon Siwek)

  * Distribution cleanup. (Jon Siwek and Robin Sommer)

  * FindPCAP now links against thread library when necessary (e.g.
    PF_RING's libpcap) (Jon Siwek)

  * Install binaries with an RPATH (Jon Siwek)

  * Workaround for FreeBSD CMake port missing debug flags (Jon Siwek)

  * Rewrite of the update-changes script. (Robin Sommer)

0.1-1 | 2011-06-14 21:12:41 -0700

  * Add a script for generating Mozilla's CA list for the SSL analyzer.
    (Seth Hall)

0.1 | 2011-04-01 16:28:22 -0700

  * Converting build process to CMake. (Jon Siwek)

  * Removing cf/hf/ca-* from distribution. The README has a note where
    to find them now. (Robin Sommer)

  * General cleanup. (Robin Sommer)

  * Initial import of bro/aux from SVN r7088. (Jon Siwek)
```

### BTest

```
0.54-65 | 2016-02-23 14:00:10 -0800

  * Fine-tuning diagnostic output. It needlessly stripped leading
    whitespace. (Robin Sommer)

0.54-63 | 2016-02-07 19:39:54 -0800

  * Extending --groups to allow running everything *except* a set of
    groups. (Robin Sommer)

  * Fix portability issue with use of mktemp. (Daniel Thayer)

0.54-60 | 2015-11-16 07:30:38 -0800

  * Updates for Python 3. (Fabian Affolter)

0.54-58 | 2015-10-01 16:04:51 -0700

  * Improved test of TEST_DIFF_FILE_MAX_LINES. (Daniel Thayer)

  * Added ability for a user to override the default number of lines
    to show for diffs by setting the environment variable
    TEST_DIFF_FILE_MAX_LINES. Reduced the default to 100. (Daniel
    Thayer)

  * When no baseline exists, changed btest-diff to always just show
    the entire file. (Daniel Thayer)

0.54-55 | 2015-08-25 07:47:22 -0700

  * Port to Python 3. (Daniel Thayer)

  * Various cleanup, bug fix, simplifications, and smaller
    improvements. (Daniel Thayer)

  * Improve and extend test suite substantially. (Daniel Thayer)

0.54-9 | 2015-07-03 18:21:52 -0700

  * Make sure IgnoreDirs works with toplevel globbing. (Robin Sommer)

0.54-8 | 2015-07-03 16:31:24 -0700

  * Expanding globs in TestDirs, relative to TestBase. (Robin Sommer)

0.54-7 | 2015-06-22 13:07:42 -0700

  * Allow BTEST_TEST_BASE overriding in alternative configuration.
    (Vlad Grigorescu)

  * Create README symlink for GitHub rendering. (Vlad Grigorescu)

0.54-1 | 2015-06-18 09:08:34 -0700

  * Add support for BTEST_TEST_BASE environment variable for
```

```
    overriding the test base directory. (Robin Sommer)

0.54 | 2015-03-02 17:22:22 -0800

  * Release 0.54.

0.53-6 | 2015-03-02 17:21:26 -0800

  * Improve documentation of timing functionality. (Daniel Thayer)

  * Add a new section to documentation that lists the BTest
    prerequisites. (Daniel Thayer)

  * Add warning when btest cannot create timing baseline. (Daniel
    Thayer)

0.53-3 | 2015-01-22 07:25:01 -0800

  * Fix some typos in the README. (Daniel Thayer)

0.53-1 | 2014-11-11 13:21:10 -0800

  * In diagnostics, do not show verbose output for tests known to
    fail. (Robin Sommer)

0.53 | 2014-07-22 17:36:24 -0700

  * Release 0.53.

0.52-2 | 2014-07-22 17:36:15 -0700

  * Update MANIFEST.in and setup.py to fix packaging. (Jon Siwek)

0.52 | 2014-03-13 14:05:44 -0700

  * Release 0.52.

0.51-14 | 2014-03-13 14:05:36 -0700

  * Fix a link in the README. (Jon Siwek)

0.51-12 | 2014-02-11 16:12:44 -0800

  * Work-around for systems reporting that a socket path is too long.
    Addresses BIT-862. (Robin Sommer)

0.51-11 | 2014-02-11 15:37:40 -0800

  * Fix for Linux systems that have the perf tool but don't support
    measuring instructions. (Robin Sommer)

  * No longer tracking tests that are expected to fail in state file.
    (Robin Sommer)

  * Refactoring the timing code to no longer execute at all when not
    needed.(Robin Sommer)

0.51-7 | 2014-02-06 21:06:40 -0800
```

```
  * Fix for platforms that don't support timing measurements yet.
    (Robin Sommer)

0.51-6 | 2014-02-06 18:19:08 -0800

  * Adding a timing mode that records test execution times per host.
    This is for catching regressions (or improvements :) that lets
    execution times divert significantly. Linux only for now. See the
    README for more information. (Robin Sommer)

  * Adding color to test status when writing to console. (Robin Sommer)

  * A bit of refactoring to define the status messages ("ok", "failed")
    only at a single location.

    Also added a note when a test declared as expecting failure in fact
    succeeds. (Robin Sommer)


0.51-2 | 2013-11-17 20:21:08 -0800

  * New keyword ``TEST-KNOWN-FAILURE`` to mark tests that are
    currently known to fail. (Robin Sommer)

0.51-1 | 2013-11-11 13:36:36 -0800

  * Fixing bug with tests potentially being ignored when using
    alternatives. (Robin Sommer)

0.51 | 2013-10-07 17:29:50 -0700

  * Updating copyright notice. (Robin Sommer)

0.5-1 | 2013-10-07 17:26:30 -0700

  * Polishing how included commands and files are shown. (Robin Sommer)

        - Enabling CSS styling to command lines and shown file names
          via the new "btest-include" and "btest-cmd" classes.

        - Fix to enable showing line numbers in btest-sphinx generated
          output.

        - Fix to enable Pygments coloring in output.

0.5 | 2013-09-20 14:48:01 -0700

  * Fix the btest-rst-pipe script. (Daniel Thayer)

  * A set of of documentation fixes, clarifications, and extensions.
    (Daniel Thayer)

  * A set of changes to Sphinx commands and directives. (Robin Sommer)

    btest-rst-*:
        - Always show line numbers.
        - Highlight the command executed.

        - rst-cmd-include gets an option -n <i> to include only upto i lines.

        - rst-cmd-include prefixes output with "<file>" to show what we're
          including.

    btest-include:
        - Set Pygments language automatically if we show a file with an
          extension we know (in particular ".bro").

        - Prefix output with "<file>" to show what we're including.

0.4-63 | 2013-08-28 21:10:39 -0700

  * btest-sphinx now provides a new directive btest-include. This
    works like literalinclude (with all its options) but it also saves
    a version of the included text as a test to detect changes. (Robin
    Sommer)

0.4-60 | 2013-08-28 18:54:51 -0700

  * Fix typos and reST formatting in README (Daniel Thayer)

  * Fix a couple of error messages. (Daniel Thayer)

  * Fixed a reference to a non-existent variable which was causing the
    "-w" option to have no effect. (Daniel Thayer)

  * Test portability fix.  (Robin Sommer)

0.4-55 | 2013-08-22 16:09:21 -0700

  * New "Sphinx-mode" for BTest, activated with -S. This allows to
    capture a test's diagnostic output when running from inside
    Sphinx; the output will now be inserted into the generated
    document. (Robin Sommer)

  * Adding an option -n to btest-rst-cmd that truncates output longer
    than N lines. (Robin Sommer)

  * Adding a PartFinalizer that runs a commmand at the completion of
    each test part. (Robin Sommer)

0.4-51 | 2013-08-22 10:36:34 -0700

  * Improve cleanup of processes that don't terminate with
    btest-bg-wait. (Jon Siwek)

0.4-49 | 2013-08-13 18:43:03 -0700

  * Fixing test portability problems. (Daniel Thayer)

  * Adding TEST_BASE environment variable. The existing TESTBASE isn't
    always behaving as expected and wasn't documented to begin with.
    (Robin Sommer)

0.4-43 | 2013-08-12 16:04:53 -0700
```

```
  * Bugfix for ignored tests. (Robin Sommer)

0.4-42 | 2013-07-31 20:46:30 -0700

  * Adding support for "parts": One can split a single test across
    multiple files by adding a numerical ``#<n>`` postfix to their
    names, where each ``<n>`` represents a separate part of the test.
    ``btests`` will combine all of a test's parts in numerical order
    and execute them subsequently within the same sandbox. Example in
    the README. (Robin Sommer)

  * When running a command, TEST_PART contains the current part
    number. (Robin Sommer)

  * Extending Sphinx support. (Robin Sommer)

        * Adding tests for Sphinx functionality.

        * Support for parts in Sphinx directives. If multiple btest
          directives reference the same test name, each will turn into
          a part of a single test.

        * Internal change restructuring the btest Sphinx directive. We
          now process it in two passes: one to save the test at parse
          time, and one later to execute once everything has been
          parsed.

        * Adding Sphinx sandbox for testing.

  * Fix for tests returning no output to render at all. (Robin Sommer)

0.4-28 | 2013-07-17 21:56:18 -0700

  * btest-diff now passes the name of the file under consideration on to
    canonifiers. (Robin Sommer)

0.4-27 | 2013-07-14 21:19:59 -0700

  * When searching for tests, BTest now ignores a directories if it finds
    a file ".btest-ignore" in there. (Robin Sommer)

0.4-26 | 2013-07-08 20:46:22 -0700

  * Fixing bug with @TEST-START-NEXT naming. (Robin Sommer)

0.4-25 | 2013-07-08 13:25:50 -0700

  * A test-suite for btest. Using, of course, btest. "make test" will
    test most of btest's features. The main missing piece is testing
    the Sphinx support, we will add that next. (Robin Sommer)

  * When creating directories, we know also create intermediaries.
    That in particular means that "@TEST-START-FILE a/b/c" now creates
    a directory "a/b" automatically and puts the file in there. (Robin
    Sommer)

  * IgnoreDirs now also works for sub directories. (Robin Sommer)
  * Documentation updates. (Robin Sommer)

  * Adding "Initializer" option, which runs a command before each
    test. (Robin Sommer)

  * Adding "CommandPrefix" option that changes the naming of all btest
    commands by replacing the "@TEST-" prefix with a custom string.
    (Robin Sommer)

  * Default configuration file can be overriden via BTEST_CFG
    environment variable. (Robin Sommer)

  * s/bro-ids.org/bro.org/g (Robin Sommer)

  * Bugfix for -j without number. (Robin Sommer)

  * New @TEST-ALTERNATIVE that activates tests only for the given
    alternative. Renamed @TEST-NO-ALTERNATIVE to
    @TEST-NOT-ALTERNATIVE, and allowing "default" for both
    @TEST-ALTERNATIVE and @TEST-NOTALTERNATIVE to specify the case
    that BTest runs without any alternative given. (Robin Sommer)

  * Fix for alternative names containing white spaces. (Robin Sommer)

0.4-14 | 2013-01-23 18:11:22 -0800

  * Fixing links in README and removing TODOs. (Robin Sommer)

0.4-13 | 2013-01-23 14:33:23 -0800

  * Allowing use of -j without a value. BTest then uses the number of
    CPU cores as reported by the OS. (Robin Sommer)

0.4-11 | 2013-01-21 17:50:40 -0800

  * Adding a new "alternative" concept that combines filters and
    substitutions, and adds per-alternative environment variables.
    (Robin Sommer)

    Instead of defining filters and substitutions separately, one now
    specifies an alternative configuration to run with "-A <name>" and
    that then checks for both "[substitutions-<name>]" and
    "[filter-<name>]" section. In addition, "[environment-<name>]"
    allows to define alternative-specific environment variables.

    The old filter/substitutions options -F and -s are gone. The
    sections for substitutions are renamed to "[substitutions-<name>]"
    from "[subst-<name>]".

0.4-10 | 2013-01-07 09:45:35 -0800

  * btest now sets a new environment variable TEST_VERBOSE, giving the
    path of a file where a test can record further information about
    its execution that will be included with btest's ``--verbose``
    output. (Robin Sommer)

0.4-9 | 2012-12-20 12:20:44 -0800
```

```
  * Documentation fixes/clarifications. (Daniel Thayer)

  * Fix the btest "-c" option, which didn't work when the specified
    config file was not in the current working directory. (Daniel
    Thayer)

0.4-6 | 2012-11-08 16:33:51 -0800

  * Putting a limit on how many input line btest-diff shows. (Robin
    Sommer)

0.4-5 | 2012-11-01 16:14:29 -0700

  * Making Sphinx module tolerant against docutils version change.
    (Robin Sommer)

0.4-4 | 2012-09-25 06:24:59 -0700

  * Fix a couple of reST formatting problems. (Daniel Thayer)

0.4-2 | 2012-09-24 11:41:06 -0700

  * Add option -x to output test results in an XML (JUnit-like)
    format. (Jon Siwek)

0.4 | 2012-06-15 15:15:13 -0700

  * Remove code to expand environment variables on command line. (Not
    needed because the command line is just passed to the shell.)
    (Daniel Thayer)

  * Clarify explanation about expansion of environment variables.
    (Daniel Thayer)

  * Fix errors in README and btest help output; added documentation
    for the -q option. (Daniel Thayer)

  * Fixed a bug in btest where it was looking for "filters-" (instead
    of "filter-") in the btest config file. (Daniel Thayer)

0.31-45 | 2012-05-24 16:43:14 -0700

  * Correct typos in documentation. (Daniel Thayer)

  * Failed tests are now only recorded into the state file when we're
    not updating. That allows to run "btest -r" repeatedly while
    updating baselines in between. (Robin Sommer)

  * Experimentation Sphinx directive to write a btest with a Sphinx
    document. See README for more information.

  * Fixing typos, plus an console output tweak. (Robin Sommer)

  * Option -q now implies -b as well. (Robin Sommer)

0.31-33 | 2012-05-13 17:08:15 -0700
```

```
  * New command to copy a file into a test's directory.

    ``@TEST-COPY-FILE: <file>``
        Copy the given file into the test's directory before the test is
        run. If ``<file>`` is a relative path, it's interpreted relative
        to the BTest's base directory. Environment variables in ``<file>``
        will be replaced if enclosed in ``${..}``. This command can be
        given multiple times. (Robin Sommer)

  * Suppressing error messages when btest-diff can't remove diag file.
    (Robin Sommer)

  * Adding option -q/--quiet to suppress informational non-error
    output. (Robin Sommer)

  * Option -F also takes a comma-separated list to specify multiple
    filters , rather than having to give -F multiple times. (Robin
    Sommer)

0.31-28 | 2012-05-06 21:27:15 -0700

  * Separating semantics of groups and thread serialization into
    separate options. -g still specifices @TEST-GROUPs that are to be
    executed, but these groups don't any longer control which tests
    get serialized in a parallel execution. For that, there's a new
    "@TEST-SERIALIZE: <tag>" command that takes a tag and then makes
    sure all tests with the same tag are run within the same thread.
    (Robin Sommer)

  * TEST-GROUPS can now be given multiple times now to assign a test
    to a set of groups. (Robin Sommer)

  * Extended -g to accept a comma-separated list of groups names to
    run more than one test group. (Robin Sommer)

  * New output handler for console output. This output is now the
    default when stdout is a terminal. It prints out a compressed
    output that updates as btest goes through; it also indicates the
    progress so far. If btest's output is redirected to a
    non-terminal, is switches back to the old style. (Robin Sommer)

  * New test command @TEST-NO-FILTER: <filter>

    This allows to ignore a test when running a specific filter. (Robin Sommer)

  * Changing the way filters are activated.

    -F <filter> now activates only the given filter, but doesn't run
    the standard tests in addition. But one can now give -F a
    command-separated list of filters to activate them all, and refer
    to the standard tests without filter as ``-``. (Robin Sommer)

  * Fix to allow numbered test to be given individually on the command
    line. (E.g., integer.geq-3 for a file that contains three tests).
    (Robin Sommer)


0.31-23 | 2012-04-16 18:10:02 -0700
```

```
  * A number of smaller fixes for bugs, plus polishing, caused by the
    recent restructuring. (Robin Sommer)

  * Removing the error given when using -r with tests on the command
    line. It's unnessary and confusing compared to when listing tests
    in btest.cfg. (Robin Sommer)

  * Adding a new "finalizer" option.

    ``Finalizer``
        An executable that will be executed each time any test has
        succesfully run. It runs in the same directory as the test itself
        and receives the name of the test as its parameter. The return
        value indicates whether the test should indeed be considered
        succeeded. By default, there's no finalizer set. (Robin Sommer)

  * btest is now again overwriting old diag files instead of appending
    (i.e., back to as it used to be). (Robin Sommer)

  * Diag output is now line-buffered. (Daniel Thayer)


0.31-13 | 2012-03-13 15:59:51 -0700

  * Adding new option -r that reruns all tests that failed last time.
    btest now always records all failed tests in a file called. (Robin
    Sommer)

  * Internal restructuring to factor output out into sublcasses.
    (Robin Sommer)

  * Adding parallel test execution to btest. (Robin Sommer)

        - A new option "-j <n>" allows to run up to <n> tests in
          parallel.

        - A new @TEST-GROUP directive allows to group tests that can't
          be parallelized. All tests of the same group will be
          executed sequentially.

        - A new option "-g <group>" allows to run only tests of a
          certain group, or with "-g -" all tests that don't have a
          group.


0.31-2 | 2012-01-25 16:58:29 -0800

  * Don't add btest's path to PATH anymore. (Jon Siwek)

0.31 | 2011-11-29 12:11:49 -0600

  * Submodule README conformity changes. (Jon Siwek)

0.3 | 2011-10-25 19:58:26 -0700

  * More graceful error handling at startup if btest.cfg not found.
    (Robin Sommer)
  * Python 2.4 compat changes. (Jon Siwek)

  * When in brief mode, btest-diff now shows full output if we don't
    have a baseline yet. (Robin Sommer)

  * Adding executable permission back to script. (Robin Sommer)

  * Cleaning up distribution. (Robin Sommer)

0.22-28 | 2011-09-15 15:18:11 -0700

  * New environment variable TEST_DIFF_BRIEF. If set btest-diff no
    longer includes a mismatching file's full content it the
    diagnostic output. This can be useful if the file being compared
    is very large. (Robin Sommer)

0.22-27 | 2011-08-12 22:56:12 -0700

  * Fix btest-bg-wait's kill trap and -k option. (Jon Siwek)

0.22-18 | 2011-07-23 11:54:07 -0700

  * A new option -u for interactively updating baselines.

  * Teach btest's TEST-START-FILE to make subdirectories (Jon Siwek)

  * Output polishing. (Robin Sommer)

  * Have distutils install 'btest-setsid' script. (Jon Siwek)

  * A portable setsid. (Robin Sommer)

  * Fixes for background execution of processes.

  * Fixing exit codes. (Robin Sommer)

0.22-6 | 2011-07-19 17:38:03 -0700

  * Teach btest's TEST-START-FILE to make subdirectories (Jon Siwek)

0.22-5 | 2011-05-02 08:41:34 -0700

  * A number of bug fixes, and output polishing. (Robin Sommer)

  * More robust background execution by btest-bg-*. (Robin Sommer)

0.22-4 | 2011-03-29 21:38:13 -0700

  * A test command can now signal to btest that even if it fails
    subsequent test commands should still run by returning exit code 100.
    btest-diff uses this to continue in the case that no baseline has
    yet been established.

  * New test option @TEST-REQUIRES for running a test conditionally.
    See the README for more information.

0.22-2 | 2011-03-03 21:44:18 -0800
```

```
  * Two new helper scripts for spawning processes in the background.
    See README for more information.

  * btest-diff can now deal with files specificied with paths.

0.22 | 2011-02-08 14:06:13 -0800

  * BTest is now hosted along with the other Bro repositories on
    git.bro-ids.org.

0.21 | 2011-01-09 21:29:18 -0800

  * In btest.cfg, option values can now include commands to execute in
    backticks.

    Example:

        [environment]
        CC=clang -emit-llvm -g `hilti-config --cflags`

  * Limiting substitutions to replacing whole words.

  * Adding "substitutions". Substitutions are similar to filters, yet
    they do not adapt the input but the command line being exectued.
    See README for more information.

  * Instead of giving a test's file name on the command line, one can
    now also use its "dotted" name as it's printed out when btest is
    running (e.g., "foo.bar"). That allows for easier copy/paste.

  * Starting CHANGES.
```

### PySubnetTree

```
0.24-7 | 2016-01-25 14:22:14 -0800

  * Added prefixes() method to return all prefixes in the tree as a
    set of strings, with or without length.  Also supports returning
    IPv4 addresses in their "native" format. (James Royalty)

0.24 | 2015-05-07 20:24:57 -0700

  * Release 0.24.

  * Update dist Makefile target (Johanna Amann)

0.23-23 | 2015-03-23 10:37:20 -0500

  * Update try..except syntax in one example in the README.
    (Daniel Thayer)

  * BIT-1303: Reorganize tests to use btest and add more test cases.
    (Daniel Thayer)
```

```
0.23-19 | 2015-03-23 09:36:00 -0500

  * Add IPv6 example in the docs (David Salisbury)

0.23-18 | 2015-03-17 09:27:14 -0700

  * Python 3 compatibility fixes. (Jon Siwek)

0.23-12 | 2014-12-12 10:44:38 -0800

  * Use IPv6 as canonical storage format (IPv4 -> IPv4-mapped IPv6).
    Addresses GitHub issues #4 and maybe #5. (Jon Siwek)

  * Update MANIFEST.in. Include SubnetTree.i in the distribution so
    the swig file (SubnetTree.cc) can be regenerated with the tarball
    if needed. . (Scott Kitterman)

0.23 | 2014-04-03 15:53:51 -0700

  * Release 0.23.

0.23 | 2014-03-31 18:05:31 -0700

  * Updated setup.py to work with setuptools. Uploaded to PyPI. (Henry
    Stern/Robin Sommer)

0.22 | 2013-10-14 09:47:15 -0700

  * Release.

0.21 | 2013-10-14 09:46:34 -0700

  * Fixing version number in setup.cfg. Addresses BIT-1088. (Robin
    Sommer)

  * Remove dead code. (Jon Siwek)

  * Fix allocator/deallocator mismatch. (Jon Siwek)

0.20 | 2013-09-23 11:53:17 -0700

  * Fix an error in README and improve the examples (Daniel Thayer)

  * Fix a broken link in the documentation. (Daniel Thayer)

0.19-9 | 2013-03-08 09:19:54 -0800

  * Fix a compiler warning. (John Siwek)

  * s/bro-ids.org/bro.org/g. (Robin Sommer)

0.19-3 | 2012-09-29 14:10:39 -0700

  * Fix compile error with Python C API. Addresses #887. (Matthias
    Vallentin)

0.19-1 | 2012-09-24 16:11:13 -0700
```

```
  * Fixing memory leak. When deleting a PySubnetTree, the values
    weren't unref'ed. (Simon Arlott)

0.19 | 2012-08-01 13:57:31 -0500

  * Fix configure script to exit with non-zero status on error (Jon Siwek)

0.18 | 2012-07-05 12:33:40 -0700

  * Improve check for SWIG/Python version incompatibility. Addresses
    #843. (Jon Siwek)

0.17-16 | 2012-07-02 14:53:05 -0700

  * Cleanup and update of SubnetTree_wrap.cc file. (Daniel Thayer)

  * Fix compile warnings and dependencies of swig-generated files. (Jon Siwek)

  * Fix typos. (Daniel Thayer)

0.17-8 | 2012-04-09 15:36:47 -0700

  * pysubnettree now supports IPv6 addresses and prefixes. (Henry
    Stern, updated by Daniel Thayer).

  * SubnetTree now have a binary mode as well in in which single
    addresses are passed in the form of packed binary strings as,
    e.g., returned by `socket.inet_aton. (Henry Stern, updated by
    Daniel Thayer).

  * Raise minimum required CMake version to 2.6.3 (Jon Siwek)


0.17 | 2012-01-09 16:11:02 -0800

  * Submodule README conformity changes. (Jon Siwek)

  * Simplify finding of Python headers/libraries. Addresses #666. (Jon
    Siwek).

0.16-3 | 2011-11-03 15:17:21 -0700

  * Fixing compiler warnings. Addresses #388. (Jon Siwek)

0.16 | 2011-10-26 13:50:28 -0700

  * Compile SWIG bindings with no-strict-aliasing (closes #644) (Jon Siwek)

0.15 | 2011-10-25 20:15:08 -0700

  * New make dist/distclean targets. (Jon Siwek)

  * Cleaning up the distribution. (Jon Siwek and Robin Sommer)

0.14-8 | 2011-09-04 09:19:08 -0700

  * Install binaries with an RPATH. (Jon Siwek)
  * Add check for incompatible swig+python versions. Addresses #562.
    (Jon Siwek)

  * Workaround for FreeBSD CMake port missing debug flags. (Jon Siwek)

0.14 | 2011-05-05 20:59:58 -0700

  * CMake build setup. (Jon Siwek)

  * Cleanup, and converting README to REST. (Robin Sommer)

  * Initial import to git. (Robin Sommer)
```

## 1.4 Quick Start Guide

**Contents**

Bro works on most modern, Unix-based systems and requires no custom hardware. It can be downloaded in either pre-built binary package or source code forms. See *Installing Bro* for instructions on how to install Bro.

In the examples below, `$PREFIX` is used to reference the Bro installation root directory, which by default is `/usr/local/bro` if you install from source.

### 1.4.1 Managing Bro with BroControl

BroControl is an interactive shell for easily operating/managing Bro installations on a single system or even across multiple systems in a traffic-monitoring cluster. This section explains how to use BroControl to manage a stand-alone Bro installation. For a complete reference on BroControl, see the *BroControl* documentation. For instructions on how to configure a Bro cluster, see the *Cluster Configuration* documentation.

### A Minimal Starting Configuration

These are the basic configuration changes to make for a minimal BroControl installation that will manage a single Bro instance on the `localhost`:

1. In `$PREFIX/etc/node.cfg`, set the right interface to monitor.
2. In `$PREFIX/etc/networks.cfg`, comment out the default settings and add the networks that Bro will consider local to the monitored environment.
3. In `$PREFIX/etc/broctl.cfg`, change the `MailTo` email address to a desired recipient and the `LogRotationInterval` to a desired log archival frequency.

Now start the BroControl shell like:

Since this is the first-time use of the shell, perform an initial installation of the BroControl configuration:

Then start up a Bro instance:

If there are errors while trying to start the Bro instance, you can can view the details with the `diag` command. If started successfully, the Bro instance will begin analyzing traffic according to a default policy and output the results in `$PREFIX/logs`.

**Note:** The user starting BroControl needs permission to capture network traffic. If you are not root, you may need to grant further privileges to the account you're using; see the FAQ. Also, if it looks like Bro is not seeing any traffic, check out the FAQ entry on checksum offloading.

You can leave it running for now, but to stop this Bro instance you would do:

We also recommend to insert the following entry into the crontab of the user running BroControl:

```
0-59/5 * * * * $PREFIX/bin/broctl cron
```

This will perform a number of regular housekeeping tasks, including verifying that the process is still running (and restarting if not in case of any abnormal termination).

### Browsing Log Files

By default, logs are written out in human-readable (ASCII) format and data is organized into columns (tab-delimited). Logs that are part of the current rotation interval are accumulated in `$PREFIX/logs/current/` (if Bro is not running, the directory will be empty). For example, the `http.log` contains the results of Bro HTTP protocol analysis. Here are the first few columns of `http.log`:

```
# ts            uid           orig_h         orig_p  resp_h          resp_p
1311627961.8  HSH4uV8KVJg  192.168.1.100  52303   192.150.187.43  80
```

Logs that deal with analysis of a network protocol will often start like this: a timestamp, a unique connection identifier (UID), and a connection 4-tuple (originator host/port and responder host/port). The UID can be used to identify all logged activity (possibly across multiple log files) associated with a given connection 4-tuple over its lifetime.

The remaining columns of protocol-specific logs then detail the protocol-dependent activity that's occurring. E.g. `http.log`'s next few columns (shortened for brevity) show a request to the root of Bro website:

```
# method   host          uri   referrer   user_agent
GET        bro.org   /     -          <...>Chrome/12.0.742.122<...>
```

Some logs are worth explicit mention:

**`conn.log`** Contains an entry for every connection seen on the wire, with basic properties such as time and duration, originator and responder IP addresses, services and ports, payload size, and much more. This log provides a comprehensive record of the network's activity.

**`notice.log`** Identifies specific activity that Bro recognizes as potentially interesting, odd, or bad. In Bro-speak, such activity is called a "notice".

By default, `BroControl` regularly takes all the logs from `$PREFIX/logs/current` and archives/compresses them to a directory named by date, e.g. `$PREFIX/logs/2011-10-06`. The frequency at which this is done can be configured via the `LogRotationInterval` option in `$PREFIX/etc/broctl.cfg`.

### Deployment Customization

The goal of most Bro *deployments* may be to send email alarms when a network event requires human intervention/investigation, but sometimes that conflicts with Bro's goal as a *distribution* to remain policy and site neutral – the events on one network may be less noteworthy than the same events on another. As a result, deploying Bro can be an iterative process of updating its policy to take different actions for events that are noticed, and using its scripting language to programmatically extend traffic analysis in a precise way.

One of the first steps to take in customizing Bro might be to get familiar with the notices it can generate by default and either tone down or escalate the action that's taken when specific ones occur.

Let's say that we've been looking at the `notice.log` for a bit and see two changes we want to make:

1. `SSL::Invalid_Server_Cert` (found in the `note` column) is one type of notice that means an SSL connection was established and the server's certificate couldn't be validated using Bro's default trust roots, but we want to ignore it.
2. `SSL::Certificate_Expired` is a notice type that is triggered when an SSL connection was established using an expired certificate. We want email when that happens, but only for certain servers on the local network (Bro can also proactively monitor for certs that will soon expire, but this is just for demonstration purposes).

We've defined *what* we want to do, but need to know *where* to do it. The answer is to use a script written in the Bro programming language, so let's do a quick intro to Bro scripting.

### Bro Scripts

Bro ships with many pre-written scripts that are highly customizable to support traffic analysis for your specific environment. By default, these will be installed into `$PREFIX/share/bro` and can be identified by the use of a `.bro` file name extension. These files should **never** be edited directly as changes will be lost when upgrading to newer versions of Bro. The exception to this rule is the directory `$PREFIX/share/bro/site` where local site-specific files can be put without fear of being clobbered later. The other main script directories under `$PREFIX/share/bro` are `base` and `policy`. By default, Bro automatically loads all scripts under `base` (unless the `-b` command line option is supplied), which deal either with collecting basic/useful state about network activities or providing frameworks/utilities that extend Bro's functionality without any performance cost. Scripts under the `policy` directory may be more situational or costly, and so users must explicitly choose if they want to load them.

The main entry point for the default analysis configuration of a standalone Bro instance managed by BroControl is the `$PREFIX/share/bro/site/local.bro` script. We'll be adding to that in the following sections, but first we have to figure out what to add.

### Redefining Script Option Variables

Many simple customizations just require you to redefine a variable from a standard Bro script with your own value, using Bro's `redef` operator.

The typical way a standard Bro script advertises tweak-able options to users is by defining variables with the `&redef` attribute and `const` qualifier. A redefineable constant might seem strange, but what that really means is that the variable's value may not change at run-time, but whose initial value can be modified via the `redef` operator at parse-time.

Let's continue on our path to modify the behavior for the two SSL notices. Looking at /scripts/base/frameworks/notice/main.bro, we see that it advertises:

That's exactly what we want to do for the first notice. Add to `local.bro`:

**Note:** The `Notice` namespace scoping is necessary here because the variable was declared and exported inside the `Notice` module, but is being referenced from outside of it. Variables declared and exported inside a module do not have to be scoped if referring to them while still inside the module.

Then go into the BroControl shell to check whether the configuration change is valid before installing it and then restarting the Bro instance:

Now that the SSL notice is ignored, let's look at how to send an email on the other notice. The notice framework has a similar option called `emailed_types`, but using that would generate email for all SSL servers with expired certificates and we only want email for connections to certain ones. There is a `policy` hook that is actually what is used to implement the simple functionality of `ignored_types` and `emailed_types`, but it's extensible such that the condition and action taken on notices can be user-defined.

In `local.bro`, let's define a new `policy` hook handler body:

```
conditional-notice.bro

@load protocols/ssl/expiring-certs

const watched_servers: set[addr] = {
        87.98.220.10,
} &redef;

# Site::local_nets usually isn't something you need to modify if
# BroControl automatically sets it up from networks.cfg.  It's
# shown here for completeness.
redef Site::local_nets += {
        87.98.0.0/16,
};

hook Notice::policy(n: Notice::Info)
        {
        if ( n$note != SSL::Certificate_Expired )
                return;

        if ( n$id$resp_h !in watched_servers )
                return;

        add n$actions[Notice::ACTION_EMAIL];
        }
```

```
# bro -r tls/tls-expired-cert.trace conditional-notice.bro
```

```
# cat notice.log
#separator \x09
#set_separator    ,
#empty_field      (empty)
```

```
#unset_field      -
#path     notice
#open     2016-10-01-03-42-13
#fields   ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p  ␣
↪     fuid     file_mime_type  file_desc        proto   note     msg      sub      src    ␣
↪dst      p        n        peer_descr      actions suppress_for    dropped remote_
↪location.country_code    remote_location.region  remote_location.city    remote_
↪location.latitude        remote_location.longitude
#types    time    string  addr    port    addr    port    string  string  string ␣
↪enum     enum    string  string  addr    addr    port    count   string  set[enum]  ␣
↪     interval        bool    string  string  string  double  double
1394745603.293028 CXWv6p3arKYeMETxOg      192.168.4.149   60539   87.98.220.10    443␣
↪     F1fX1R2cDOzbvg17ye      -       -       tcp     SSL::Certificate_Expired       ␣
↪Certificate CN=www.spidh.org,OU=COMODO SSL,OU=Domain Control Validated expired at␣
↪2014-03-04-23:59:59.000000000 -       192.168.4.149   87.98.220.10    443     -      ␣
↪  bro     Notice::ACTION_EMAIL,Notice::ACTION_LOG 86400.000000    F       -       -␣
↪       -       -       -
#close    2016-10-01-03-42-14
```

You'll just have to trust the syntax for now, but what we've done is first declare our own variable to hold a set of watched addresses, `watched_servers`; then added a hook handler body to the policy that will generate an email whenever the notice type is an SSL expired certificate and the responding host stored inside the `Info` record's connection field is in the set of watched servers.

**Note:** Record field member access is done with the '$' character instead of a '.' as might be expected from other languages, in order to avoid ambiguity with the built-in address type's use of '.' in IPv4 dotted decimal representations.

Remember, to finalize that configuration change perform the `check`, `install`, `restart` commands in that order inside the BroControl shell.

### Next Steps

By this point, we've learned how to set up the most basic Bro instance and tweak the most basic options. Here's some suggestions on what to explore next:

- We only looked at how to change options declared in the notice framework, there's many more options to look at in other script packages.
- Continue reading with *Using Bro* chapter which goes into more depth on working with Bro; then look at *Writing Bro Scripts* for learning how to start writing your own scripts.
- Look at the scripts in `$PREFIX/share/bro/policy` for further ones you may want to load; you can browse their documentation at the *overview of script packages*.
- Reading the code of scripts that ship with Bro is also a great way to gain further understanding of the language and how scripts tend to be structured.
- Review the FAQ.
- Continue reading below for another mini-tutorial on using Bro as a standalone command-line utility.

## 1.4.2 Bro as a Command-Line Utility

If you prefer not to use BroControl (e.g. don't need its automation and management features), here's how to directly control Bro for your analysis activities from the command line for both live traffic and offline working from traces.

### Monitoring Live Traffic

Analyzing live traffic from an interface is simple:

`en0` can be replaced by the interface of your choice and for the list of scripts, you can just use "all" for now to perform all the default analysis that's available.

Bro will output log files into the working directory.

---

**Note:** The FAQ entries about capturing as an unprivileged user and checksum offloading are particularly relevant at this point.

---

To use the site-specific `local.bro` script, just add it to the command-line:

This will cause Bro to print a warning about lacking the `Site::local_nets` variable being configured. You can supply this information at the command line like this (supply your "local" subnets in place of the example subnets):

### Reading Packet Capture (pcap) Files

Capturing packets from an interface and writing them to a file can be done like this:

Where `en0` can be replaced by the correct interface for your system as shown by e.g. `ifconfig`. (The `-s 0` argument tells it to capture whole packets; in cases where it's not supported use `-s 65535` instead).

After a while of capturing traffic, kill the `tcpdump` (with ctrl-c), and tell Bro to perform all the default analysis on the capture which primarily includes :

Bro will output log files into the working directory.

If you are interested in more detection, you can again load the `local` script that we include as a suggested configuration:

### Telling Bro Which Scripts to Load

A command-line invocation of Bro typically looks like:

Where the last arguments are the specific policy scripts that this Bro instance will load. These arguments don't have to include the `.bro` file extension, and if the corresponding script resides under the default installation path, `$PREFIX/share/bro`, then it requires no path qualification. Further, a directory of scripts can be specified as an argument to be loaded as a "package" if it contains a `__load__.bro` script that defines the scripts that are part of the package.

This example does all of the base analysis (primarily protocol logging) and adds SSL certificate validation.

You might notice that a script you load from the command line uses the `@load` directive in the Bro language to declare dependence on other scripts. This directive is similar to the `#include` of C/C++, except the semantics are, "load this script if it hasn't already been loaded."

---

**Note:** If one wants Bro to be able to load scripts that live outside the default directories in Bro's installation root, the `BROPATH` environment variable will need to be extended to include all the directories that need to be searched for scripts. See the default search path by doing `bro --help`.

---

### Running Bro Without Installing

For developers that wish to run Bro directly from the `build/` directory (i.e., without performing `make install`), they will have to first adjust `BROPATH` to look for scripts and additional files inside the build directory. Sourcing either `build/bro-path-dev.sh` or `build/bro-path-dev.csh` as appropriate for the current shell accomplishes this and also augments your `PATH` so you can use the Bro binary directly:

```
./configure
make
source build/bro-path-dev.sh
bro <options>
```

## 1.5 Cluster Configuration

**Contents**

A *Bro Cluster* is a set of systems jointly analyzing the traffic of a network link in a coordinated fashion. You can operate such a setup from a central manager system easily using BroControl because BroControl hides much of the complexity of the multi-machine installation.

This section gives examples of how to setup common cluster configurations using BroControl. For a full reference on BroControl, see the *BroControl* documentation.

### 1.5.1 Preparing to Setup a Cluster

In this document we refer to the user account used to set up the cluster as the "Bro user". When setting up a cluster the Bro user must be set up on all hosts, and this user must have ssh access from the manager to all machines in the cluster, and it must work without being prompted for a password/passphrase (for example, using ssh public key authentication). Also, on the worker nodes this user must have access to the target network interface in promiscuous mode.

Additional storage must be available on all hosts under the same path, which we will call the cluster's prefix path. We refer to this directory as `<prefix>`. If you build Bro from source, then `<prefix>` is the directory specified with the `--prefix` configure option, or `/usr/local/bro` by default. The Bro user must be able to either create this directory or, where it already exists, must have write permission inside this directory on all hosts.

When trying to decide how to configure the Bro nodes, keep in mind that there can be multiple Bro instances running on the same host. For example, it's possible to run a proxy and the manager on the same host. However, it is recommended to run workers on a different machine than the manager because workers can consume a lot of CPU resources. The maximum recommended number of workers to run on a machine should be one or two less than the number of CPU cores available on that machine. Using a load-balancing method (such as PF_RING) along with CPU pinning can decrease the load on the worker machines. Also, in order to reduce the load on the manager process, it is recommended to have a logger in your configuration. If a logger is defined in your cluster configuration, then it will receive logs instead of the manager process.

### 1.5.2 Basic Cluster Configuration

With all prerequisites in place, perform the following steps to setup a Bro cluster (do this as the Bro user on the manager host only):

- Edit the BroControl configuration file, `<prefix>/etc/broctl.cfg`, and change the value of any BroControl options to be more suitable for your environment. You will most likely want to change the value of the `MailTo` and `LogRotationInterval` options. A complete reference of all BroControl options can be found in the *BroControl* documentation.
- Edit the BroControl node configuration file, `<prefix>/etc/node.cfg` to define where logger, manager, proxies, and workers are to run. For a cluster configuration, you must comment-out (or remove) the standalone node in that file, and either uncomment or add node entries for each node in your cluster (logger, manager, proxy, and workers). For example, if you wanted to run five Bro nodes (two workers, one proxy, a logger, and a manager) on a cluster consisting of three machines, your cluster configuration would look like this:

  ```
  [logger]
  type=logger
  host=10.0.0.10

  [manager]
  type=manager
  host=10.0.0.10

  [proxy-1]
  type=proxy
  host=10.0.0.10

  [worker-1]
  type=worker
  host=10.0.0.11
  interface=eth0

  [worker-2]
  type=worker
  host=10.0.0.12
  interface=eth0
  ```

  For a complete reference of all options that are allowed in the `node.cfg` file, see the *BroControl* documentation.
- Edit the network configuration file `<prefix>/etc/networks.cfg`. This file lists all of the networks which the cluster should consider as local to the monitored environment.
- Install Bro on all machines in the cluster using BroControl:

  ```
  > broctl install
  ```
- Some tasks need to be run on a regular basis. On the manager node, insert a line like this into the crontab of the user running the cluster:

```
0-59/5 * * * * <prefix>/bin/broctl cron
```

(Note: if you are editing the system crontab instead of a user's own crontab, then you need to also specify the user which the command will be run as. The username must be placed after the time fields and before the broctl command.)

Note that on some systems (FreeBSD in particular), the default PATH for cron jobs does not include the directories where bash and python are installed (the symptoms of this problem would be that "broctl cron" works when run directly by the user, but does not work from a cron job). To solve this problem, you would either need to create symlinks to bash and python in a directory that is in the default PATH for cron jobs, or specify a new PATH in the crontab.

### 1.5.3 PF_RING Cluster Configuration

PF_RING allows speeding up the packet capture process by installing a new type of socket in Linux systems. It supports 10Gbit hardware packet filtering using standard network adapters, and user-space DNA (Direct NIC Access) for fast packet capture/transmission.

#### Installing PF_RING

1. Download and install PF_RING for your system following the instructions here. The following commands will install the PF_RING libraries and kernel module (replace the version number 5.6.2 in this example with the version that you downloaded):

```
cd /usr/src
tar xvzf PF_RING-5.6.2.tar.gz
cd PF_RING-5.6.2/userland/lib
./configure --prefix=/opt/pfring
make install

cd ../libpcap
./configure --prefix=/opt/pfring
make install

cd ../tcpdump-4.1.1
./configure --prefix=/opt/pfring
make install

cd ../../kernel
make install

modprobe pf_ring enable_tx_capture=0 min_num_slots=32768
```

Refer to the documentation for your Linux distribution on how to load the pf_ring module at boot time. You will need to install the PF_RING library files and kernel module on all of the workers in your cluster.

2. Download the Bro source code.
3. Configure and install Bro using the following commands:

```
./configure --with-pcap=/opt/pfring
make
make install
```

4. Make sure Bro is correctly linked to the PF_RING libpcap libraries:

```
ldd /usr/local/bro/bin/bro | grep pcap
      libpcap.so.1 => /opt/pfring/lib/libpcap.so.1 (0x00007fa6d7d24000)
```

5. Configure BroControl to use PF_RING (explained below).
6. Run “broctl install” on the manager. This command will install Bro and required scripts to all machines in your cluster.

### Using PF_RING

In order to use PF_RING, you need to specify the correct configuration options for your worker nodes in BroControl’s node configuration file. Edit the `node.cfg` file and specify `lb_method=pf_ring` for each of your worker nodes. Next, use the `lb_procs` node option to specify how many Bro processes you’d like that worker node to run, and optionally pin those processes to certain CPU cores with the `pin_cpus` option (CPU numbering starts at zero). The correct `pin_cpus` setting to use is dependent on your CPU architecture (Intel and AMD systems enumerate processors in different ways). Using the wrong `pin_cpus` setting can cause poor performance. Here is what a worker node entry should look like when using PF_RING and CPU pinning:

```
[worker-1]
type=worker
host=10.0.0.50
interface=eth0
lb_method=pf_ring
lb_procs=10
pin_cpus=2,3,4,5,6,7,8,9,10,11
```

### Using PF_RING+DNA with symmetric RSS

You must have a PF_RING+DNA license in order to do this. You can sniff each packet only once.

1. Load the DNA NIC driver (i.e. ixgbe) on each worker host.
2. Run “ethtool -L dna0 combined 10” (this will establish 10 RSS queues on your NIC) on each worker host. You must make sure that you set the number of RSS queues to the same as the number you specify for the lb_procs option in the node.cfg file.
3. On the manager, configure your worker(s) in node.cfg:

```
[worker-1]
type=worker
host=10.0.0.50
interface=dna0
lb_method=pf_ring
lb_procs=10
```

### Using PF_RING+DNA with pfdnacluster_master

You must have a PF_RING+DNA license and a libzero license in order to do this. You can load balance between multiple applications and sniff the same packets multiple times with different tools.

1. Load the DNA NIC driver (i.e. ixgbe) on each worker host.
2. Run “ethtool -L dna0 1” (this will establish 1 RSS queues on your NIC) on each worker host.
3. Run the pfdnacluster_master command on each worker host. For example:

```
pfdnacluster_master -c 21 -i dna0 -n 10
```

Make sure that your cluster ID (21 in this example) matches the interface name you specify in the node.cfg file. Also make sure that the number of processes you're balancing across (10 in this example) matches the lb_procs option in the node.cfg file.

4. If you are load balancing to other processes, you can use the pfringfirstappinstance variable in broctl.cfg to set the first application instance that Bro should use. For example, if you are running pfdnacluster_master with "-n 10,4" you would set pfringfirstappinstance=4. Unfortunately that's still a global setting in broctl.cfg at the moment but we may change that to something you can set in node.cfg eventually.

5. On the manager, configure your worker(s) in node.cfg:

```
[worker-1]
type=worker
host=10.0.0.50
interface=dnacluster:21
lb_method=pf_ring
lb_procs=10
```

CHAPTER

TWO

# USING BRO SECTION

## 2.1 Bro Logging

**Contents**

Once Bro has been deployed in an environment and monitoring live traffic, it will, in its default configuration, begin to produce human-readable ASCII logs. Each log file, produced by Bro's *Logging Framework*, is populated with organized, mostly connection-oriented data. As the standard log files are simple ASCII data, working with the data contained in them can be done from a command line terminal once you have been familiarized with the types of data that can be found in each file. In the following, we work through the logs general structure and then examine some standard ways of working with them.

### 2.1.1 Working with Log Files

Generally, all of Bro's log files are produced by a corresponding script that defines their individual structure. However, as each log file flows through the Logging Framework, they share a set of structural similarities. Without breaking into the scripting aspect of Bro here, a bird's eye view of how the log files are produced progresses as follows. The script's author defines the kinds of data, such as the originating IP address or the duration of a connection, which will make up the fields (i.e., columns) of the log file. The author then decides what network activity should generate a single log file entry (i.e., one line). For example, this could be a connection having been completed or an HTTP `GET` request being issued by an originator. When these behaviors are observed during operation, the data is passed to the Logging Framework which adds the entry to the appropriate log file.

As the fields of the log entries can be further customized by the user, the Logging Framework makes use of a header block to ensure that it remains self-describing. This header entry can be see by running the Unix utility `head` and outputting the first lines of the file:

```
# bro -r wikipedia.trace
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     conn
#open     2016-10-01-03-42-11
#fields   ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p
↪    proto   service duration        orig_bytes      resp_bytes      conn_state
↪local_orig      local_resp      missed_bytes    history orig_pkts       orig_ip_
↪bytes   resp_pkts       resp_ip_bytes   tunnel_parents
#types    time    string  addr    port    addr    port    enum    string  interval
↪    count   count   string  bool    bool    count   string  count   count   count
↪count   set[string]
1300475167.096535 CXWv6p3arKYeMETxOg      141.142.220.202 5353    224.0.0.251
↪5353    udp     dns     -       -       -       S0      -       -       0       D
↪    1       73      0       0       (empty)
1300475167.097012 CjhGID4nQcgTWjvg4c      fe80::217:f2ff:fed7:cf65        5353
↪ff02::fb        5353    udp     dns     -       -       -       S0      -       -
↪    0       D       1       199     0       0       (empty)
1300475167.099816 CCvvfg3TEfuqmmG4bh      141.142.220.50  5353    224.0.0.251
↪5353    udp     dns     -       -       -       S0      -       -       0       D
↪    1       179     0       0       (empty)
1300475168.853899 CPbrpk1qSsw6ESzHV4      141.142.220.118 43927   141.142.2.2     53
↪    udp     dns     0.000435        38      89      SF      -       -       0
↪Dd      1       66      1       117     (empty)
1300475168.854378 C6pKV8GSxOnSLghOa       141.142.220.118 37676   141.142.2.2     53
↪    udp     dns     0.000420        52      99      SF      -       -       0
↪Dd      1       80      1       127     (empty)
1300475168.854837 CIPOse170MGiRM1Qf4      141.142.220.118 40526   141.142.2.2     53
↪    udp     dns     0.000392        38      183     SF      -       -       0
↪Dd      1       66      1       211     (empty)
1300475168.857956 CMXxB5GvmoxJFXdTa       141.142.220.118 32902   141.142.2.2     53
↪    udp     dns     0.000317        38      89      SF      -       -       0
↪Dd      1       66      1       117     (empty)
[...]
```

As you can see, the header consists of lines prefixed by # and includes information such as what separators are being used for various types of data, what an empty field looks like and what an unset field looks like. In this example, the default TAB separator is being used as the delimiter between fields (`\x09` is the tab character in hex). It also lists the comma as the separator for set data, the string `(empty)` as the indicator for an empty field and the `-` character as the indicator for a field that hasn't been set. The timestamp for when the file was created is included under `#open`. The header then goes on to detail the fields being listed in the file and the data types of those fields, in `#fields` and `#types`, respectively. These two entries are often the two most significant points of interest as they detail not only the field names but the data types used. When navigating through the different log files with tools like `sed`, `awk`, or `grep`, having the field definitions readily available saves the user some mental leg work. The field names are also a key resource for using the *bro-cut* utility included with Bro, see below.

Next to the header follows the main content. In this example we see 7 connections with their key properties, such as originator and responder IP addresses (note how Bro transparently handles both IPv4 and IPv6), transport-layer ports, application-layer services ( - the `service` field is filled in as Bro determines a specific protocol to be in use, independent of the connection's ports), payload size, and more. See `Conn::Info` for a description of all fields.

In addition to `conn.log`, Bro generates many further logs by default, including:

**`dpd.log`** A summary of protocols encountered on non-standard ports.

**`dns.log`** All DNS activity.

**`ftp.log`** A log of FTP session-level activity.

**`files.log`** Summaries of files transferred over the network. This information is aggregated from different protocols, including HTTP, FTP, and SMTP.

**http.log** A summary of all HTTP requests with their replies.

**known_certs.log** SSL certificates seen in use.

**smtp.log** A summary of SMTP activity.

**ssl.log** A record of SSL sessions, including certificates being used.

**weird.log** A log of unexpected protocol-level activity. Whenever Bro's protocol analysis encounters a situation it would not expect (e.g., an RFC violation) it logs it in this file. Note that in practice, real-world networks tend to exhibit a large number of such "crud" that is usually not worth following up on.

As you can see, some log files are specific to a particular protocol, while others aggregate information across different types of activity. For a complete list of log files and a description of its purpose, see *Log Files*.

### Using `bro-cut`

The `bro-cut` utility can be used in place of other tools to build terminal commands that remain flexible and accurate independent of possible changes to the log file itself. It accomplishes this by parsing the header in each file and allowing the user to refer to the specific columnar data available (in contrast to tools like `awk` that require the user to refer to fields referenced by their position). For example, the following command extracts just the given columns from a `conn.log`:

```
# cat conn.log | bro-cut id.orig_h id.orig_p id.resp_h duration
141.142.220.202   5353    224.0.0.251     -
fe80::217:f2ff:fed7:cf65  5353    ff02::fb        -
141.142.220.50    5353    224.0.0.251     -
141.142.220.118   43927   141.142.2.2     0.000435
141.142.220.118   37676   141.142.2.2     0.000420
141.142.220.118   40526   141.142.2.2     0.000392
141.142.220.118   32902   141.142.2.2     0.000317
141.142.220.118   59816   141.142.2.2     0.000343
141.142.220.118   59714   141.142.2.2     0.000375
141.142.220.118   58206   141.142.2.2     0.000339
[...]
```

The corresponding `awk` command will look like this:

```
# awk '/^[^#]/ {print $3, $4, $5, $6, $9}' conn.log
141.142.220.202 5353 224.0.0.251 5353 -
fe80::217:f2ff:fed7:cf65 5353 ff02::fb 5353 -
141.142.220.50 5353 224.0.0.251 5353 -
141.142.220.118 43927 141.142.2.2 53 0.000435
141.142.220.118 37676 141.142.2.2 53 0.000420
141.142.220.118 40526 141.142.2.2 53 0.000392
141.142.220.118 32902 141.142.2.2 53 0.000317
141.142.220.118 59816 141.142.2.2 53 0.000343
141.142.220.118 59714 141.142.2.2 53 0.000375
141.142.220.118 58206 141.142.2.2 53 0.000339
[...]
```

While the output is similar, the advantages to using bro-cut over `awk` lay in that, while `awk` is flexible and powerful, `bro-cut` was specifically designed to work with Bro's log files. Firstly, the `bro-cut` output includes only the log file entries, while the `awk` solution needs to skip the header manually. Secondly, since `bro-cut` uses the field descriptors to identify and extract data, it allows for flexibility independent of the format and contents of the log file. It's not uncommon for a Bro configuration to add extra fields to various log files as required by the environment. In this case, the fields in the `awk` command would have to be altered to compensate for the new position whereas the `bro-cut` output would not change.

**Note:** The sequence of field names given to `bro-cut` determines the output order, which means you can also use `bro-cut` to reorder fields. That can be helpful when piping into, e.g., `sort`.

As you may have noticed, the command for `bro-cut` uses the output redirection through the `cat` command and `|` operator. Whereas tools like `awk` allow you to indicate the log file as a command line option, bro-cut only takes input through redirection such as `|` and `<`. There are a couple of ways to direct log file data into `bro-cut`, each dependent upon the type of log file you're processing. A caveat of its use, however, is that all of the header lines must be present.

**Note:** `bro-cut` provides an option `-c` to include a corresponding format header into the output, which allows to chain multiple `bro-cut` instances or perform further post-processing that evaluates the header information.

In its default setup, Bro will rotate log files on an hourly basis, moving the current log file into a directory with format `YYYY-MM-DD` and gzip compressing the file with a file format that includes the log file type and time range of the file. In the case of processing a compressed log file you simply adjust your command line tools to use the complementary `z*` versions of commands such as `cat` (`zcat`) or `grep` (`zgrep`).

### Working with Timestamps

`bro-cut` accepts the flag `-d` to convert the epoch time values in the log files to human-readable format. The following command includes the human readable time stamp, the unique identifier, the HTTP `Host`, and HTTP `URI` as extracted from the `http.log` file:

```
# bro-cut -d ts uid host uri < http.log
2011-03-18T19:06:08+0000  CRJuHdVW0XPVINV8a       bits.wikimedia.org      /skins-1.5/
↪monobook/main.css
2011-03-18T19:06:08+0000  CJ3xTn1c4Zw9TmAE05      upload.wikimedia.org    /wikipedia/
↪commons/6/63/Wikipedia-logo.png
2011-03-18T19:06:08+0000  C7XEbhP654jzLoe3a       upload.wikimedia.org    /wikipedia/
↪commons/thumb/b/bb/Wikipedia_wordmark.svg/174px-Wikipedia_wordmark.svg.png
2011-03-18T19:06:08+0000  C3SfNE4BWaU4aSuwkc      upload.wikimedia.org    /wikipedia/
↪commons/b/bd/Bookshelf-40x201_6.png
2011-03-18T19:06:08+0000  CyAhVIzHqb7t7kv28       upload.wikimedia.org    /wikipedia/
↪commons/thumb/8/8a/Wikinews-logo.png/35px-Wikinews-logo.png
[...]
```

Often times log files from multiple sources are stored in UTC time to allow easy correlation. Converting the timestamp from a log file to UTC can be accomplished with the `-u` option:

```
# bro-cut -u ts uid host uri < http.log
2011-03-18T19:06:08+0000  CRJuHdVW0XPVINV8a       bits.wikimedia.org      /skins-1.5/
↪monobook/main.css
2011-03-18T19:06:08+0000  CJ3xTn1c4Zw9TmAE05      upload.wikimedia.org    /wikipedia/
↪commons/6/63/Wikipedia-logo.png
2011-03-18T19:06:08+0000  C7XEbhP654jzLoe3a       upload.wikimedia.org    /wikipedia/
↪commons/thumb/b/bb/Wikipedia_wordmark.svg/174px-Wikipedia_wordmark.svg.png
2011-03-18T19:06:08+0000  C3SfNE4BWaU4aSuwkc      upload.wikimedia.org    /wikipedia/
↪commons/b/bd/Bookshelf-40x201_6.png
2011-03-18T19:06:08+0000  CyAhVIzHqb7t7kv28       upload.wikimedia.org    /wikipedia/
↪commons/thumb/8/8a/Wikinews-logo.png/35px-Wikinews-logo.png
[...]
```

The default time format when using the `-d` or `-u` is the `strftime` format string `%Y-%m-%dT%H:%M:%S%z` which results in a string with year, month, day of month, followed by hour, minutes, seconds and the timezone offset. The default format can be altered by using the `-D` and `-U` flags, using the standard `strftime` syntax. For example, to format the timestamp in the US-typical "Middle Endian" you could use a format string of: `%d-%m-%YT%H:%M:%S%z`

```
# bro-cut -D %d-%m-%YT%H:%M:%S%z ts uid host uri < http.log
18-03-2011T19:06:08+0000  CRJuHdVW0XPVINV8a       bits.wikimedia.org      /skins-1.5/
↪monobook/main.css
18-03-2011T19:06:08+0000  CJ3xTn1c4Zw9TmAE05      upload.wikimedia.org    /wikipedia/
↪commons/6/63/Wikipedia-logo.png
18-03-2011T19:06:08+0000  C7XEbhP654jzLoe3a       upload.wikimedia.org    /wikipedia/
↪commons/thumb/b/bb/Wikipedia_wordmark.svg/174px-Wikipedia_wordmark.svg.png
18-03-2011T19:06:08+0000  C3SfNE4BWaU4aSuwkc      upload.wikimedia.org    /wikipedia/
↪commons/b/bd/Bookshelf-40x201_6.png
18-03-2011T19:06:08+0000  CyAhVIzHqb7t7kv28       upload.wikimedia.org    /wikipedia/
↪commons/thumb/8/8a/Wikinews-logo.png/35px-Wikinews-logo.png
[...]
```

See `man strfime` for more options for the format string.

### Using UIDs

While Bro can do signature-based analysis, its primary focus is on behavioral detection which alters the practice of log review from "reactionary review" to a process a little more akin to a hunting trip. A common progression of review includes correlating a session across multiple log files. As a connection is processed by Bro, a unique identifier is assigned to each session. This unique identifier is generally included in any log file entry associated with that connection and can be used to cross-reference different log files.

A simple example would be to cross-reference a UID seen in a `conn.log` file. Here, we're looking for the connection with the largest number of bytes from the responder by redirecting the output for `cat conn.log` into bro-cut to extract the UID and the resp_bytes, then sorting that output by the resp_bytes field.

```
# cat conn.log | bro-cut uid resp_bytes | sort -nrk2 | head -5
CyAhVIzHqb7t7kv28 734
CkDsfG2YIeWJmXWNWj        734
CJ3xTn1c4Zw9TmAE05        734
C3SfNE4BWaU4aSuwkc        734
CzA03V1VcgagLjnO92        733
```

Taking the UID of the first of the top responses, we can now crossreference that with the UIDs in the `http.log` file.

```
ERROR executing test 'doc.sphinx.using_bro' (part 8)

% 'btest-rst-cmd "cat http.log | bro-cut uid id.resp_h method status_code host uri |␣
↪grep UM0KZ3MLUfNB0cl11"' failed unexpectedly (exit code 1)
% cat .stderr
```

As you can see there are two HTTP `GET` requests within the session that Bro identified and logged. Given that HTTP is a stream protocol, it can have multiple `GET`/`POST`/etc requests in a stream and Bro is able to extract and track that information for you, giving you an in-depth and structured view into HTTP traffic on your network.

## 2.2 Monitoring HTTP Traffic with Bro

Bro can be used to log the entire HTTP traffic from your network to the http.log file. This file can then be used for analysis and auditing purposes.

In the sections below we briefly explain the structure of the http.log file, then we show you how to perform basic HTTP traffic monitoring and analysis tasks with Bro. Some of these ideas and techniques can later be applied to monitor different protocols in a similar way.

## 2.2.1 Introduction to the HTTP log

The http.log file contains a summary of all HTTP requests and responses sent over a Bro-monitored network. Here are the first few columns of `http.log`:

```
# ts           uid           orig_h         orig_p  resp_h          resp_p
1311627961.8  HSH4uV8KVJg  192.168.1.100 52303   192.150.187.43 80
```

Every single line in this log starts with a timestamp, a unique connection identifier (UID), and a connection 4-tuple (originator host/port and responder host/port). The UID can be used to identify all logged activity (possibly across multiple log files) associated with a given connection 4-tuple over its lifetime.

The remaining columns detail the activity that's occurring. For example, the columns on the line below (shortened for brevity) show a request to the root of Bro website:

```
# method   host          uri  referrer  user_agent
GET        bro.org  /    -          <...>Chrome/12.0.742.122<...>
```

Network administrators and security engineers, for instance, can use the information in this log to understand the HTTP activity on the network and troubleshoot network problems or search for anomalous activities. We must stress that there is no single right way to perform an analysis. It will depend on the expertise of the person performing the analysis and the specific details of the task.

For more information about how to handle the HTTP protocol in Bro, including a complete list of the fields available in http.log, go to Bro's HTTP script reference.

## 2.2.2 Detecting a Proxy Server

A proxy server is a device on your network configured to request a service on behalf of a third system; one of the most common examples is a Web proxy server. A client without Internet access connects to the proxy and requests a web page, the proxy sends the request to the web server, which receives the response, and passes it to the original client.

Proxies were conceived to help manage a network and provide better encapsulation. Proxies by themselves are not a security threat, but a misconfigured or unauthorized proxy can allow others, either inside or outside the network, to access any web site and even conduct malicious activities anonymously using the network's resources.

### What Proxy Server traffic looks like

In general, when a client starts talking with a proxy server, the traffic consists of two parts: (i) a GET request, and (ii) an HTTP/ reply:

```
Request: GET http://www.bro.org/ HTTP/1.1
Reply:   HTTP/1.0 200 OK
```

This will differ from traffic between a client and a normal Web server because GET requests should not include "http" on the string. So we can use this to identify a proxy server.

We can write a basic script in Bro to handle the http_reply event and detect a reply for a `GET http://` request.

```
http_proxy_01.bro

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( /^[hH][tT][tT][pP]:/ in c$http$uri && c$http$status_code == 200 )
                print fmt("A local server is acting as an open proxy: %s", c$id$resp_
↪h);
        }
```

```
# bro -r http/proxy.pcap http_proxy_01.bro
A local server is acting as an open proxy: 192.168.56.101
```

Basically, the script is checking for a "200 OK" status code on a reply for a request that includes "http:" (case insensitive). In reality, the HTTP protocol defines several success status codes other than 200, so we will extend our basic script to also consider the additional codes.

```
http_proxy_02.bro


module HTTP;

export {

        global success_status_codes: set[count] = {
                200,
                201,
                202,
                203,
                204,
                205,
                206,
                207,
                208,
                226,
                304
        };
}

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( /^[hH][tT][tT][pP]:/ in c$http$uri &&
             c$http$status_code in HTTP::success_status_codes )
                print fmt("A local server is acting as an open proxy: %s", c$id$resp_
↪h);
        }
```

```
# bro -r http/proxy.pcap http_proxy_02.bro
A local server is acting as an open proxy: 192.168.56.101
```

Next, we will make sure that the responding proxy is part of our local network.

```
http_proxy_03.bro


@load base/utils/site

redef Site::local_nets += { 192.168.0.0/16 };
```

```

module HTTP;

export {

        global success_status_codes: set[count] = {
                200,
                201,
                202,
                203,
                204,
                205,
                206,
                207,
                208,
                226,
                304
        };
}

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( Site::is_local_addr(c$id$resp_h) &&
             /^[hH][tT][tT][pP]:/ in c$http$uri &&
             c$http$status_code in HTTP::success_status_codes )
                print fmt("A local server is acting as an open proxy: %s", c$id$resp_
↪h);
        }
```

```
# bro -r http/proxy.pcap http_proxy_03.bro
A local server is acting as an open proxy: 192.168.56.101
```

**Note:** The redefinition of `Site::local_nets` is only done inside this script to make it a self-contained example. It's typically redefined somewhere else.

Finally, our goal should be to generate an alert when a proxy has been detected instead of printing a message on the console output. For that, we will tag the traffic accordingly and define a new `Open_Proxy Notice` type to alert of all tagged communications. Once a notification has been fired, we will further suppress it for one day. Below is the complete script.

```
http_proxy_04.bro

@load base/utils/site
@load base/frameworks/notice

redef Site::local_nets += { 192.168.0.0/16 };

module HTTP;

export {

        redef enum Notice::Type += {
                Open_Proxy
        };

        global success_status_codes: set[count] = {
                200,
                201,
                202,
                203,
                204,
                205,
                206,
                207,
                208,
                226,
                304
        };
}

event http_reply(c: connection, version: string, code: count, reason: string)
        {
        if ( Site::is_local_addr(c$id$resp_h) &&
             /^[hH][tT][tT][pP]:/ in c$http$uri &&
             c$http$status_code in HTTP::success_status_codes )
                NOTICE([$note=HTTP::Open_Proxy,
                        $msg=fmt("A local server is acting as an open proxy: %s",
                                 c$id$resp_h),
                        $conn=c,
                        $identifier=cat(c$id$resp_h),
                        $suppress_for=1day]);
        }
```

```
# bro -r http/proxy.pcap http_proxy_04.bro
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     notice
#open     2016-10-01-03-42-09
#fields   ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p  ␣
↪    fuid    file_mime_type  file_desc       proto   note    msg     sub     src    ␣
↪dst     p       n       peer_descr      actions suppress_for    dropped remote_
↪location.country_code    remote_location.region  remote_location.city    remote_
↪location.latitude        remote_location.longitude
#types    time    string  addr    port    addr    port    string  string  string ␣
↪enum    enum    string  string  addr    addr    port    count   string  set[enum]  ␣
↪    interval        bool    string  string  string  double  double
1389654450.449603 CXWv6p3arKYeMETxOg      192.168.56.1    52679   192.168.56.101  80 ␣
↪    -       -       -       tcp     HTTP::Open_Proxy        A local server is␣
↪acting as an open proxy: 192.168.56.101       -       192.168.56.1    192.168.56.
↪101  80      -       bro     Notice::ACTION_LOG      86400.000000    F       -      ␣
↪ -       -       -       -
#close    2016-10-01-03-42-09
```

Note that this script only logs the presence of the proxy to `notice.log`, but if an additional email is desired (and email functionality is enabled), then that's done simply by redefining `Notice::emailed_types` to add the `Open_proxy` notice type to it.

### 2.2.3 Inspecting Files

Files are often transmitted on regular HTTP conversations between a client and a server. Most of the time these files are harmless, just images and some other multimedia content, but there are also types of files, specially executable files, that can damage your system. We can instruct Bro to create a copy of all files of certain types that it sees using the *File Analysis Framework* (introduced with Bro 2.2):

```
file_extraction.bro


global mime_to_ext: table[string] of string = {
        ["application/x-dosexec"] = "exe",
        ["text/plain"] = "txt",
        ["image/jpeg"] = "jpg",
        ["image/png"] = "png",
        ["text/html"] = "html",
};

event file_sniff(f: fa_file, meta: fa_metadata)
        {
        if ( f$source != "HTTP" )
                return;

        if ( ! meta?$mime_type )
                return;

        if ( meta$mime_type !in mime_to_ext )
                return;

        local fname = fmt("%s-%s.%s", f$source, f$id, mime_to_ext[meta$mime_type]);
        print fmt("Extracting file %s", fname);
        Files::add_analyzer(f, Files::ANALYZER_EXTRACT, [$extract_filename=fname]);
        }
```

```
# bro -r http/bro.org.pcap file_extraction.bro
Extracting file HTTP-FiIpIB2hRQSDBOSJRg.html
Extracting file HTTP-FMG4bMmVV64eOsCb.txt
Extracting file HTTP-FnaT2a3UDd093opCB9.txt
Extracting file HTTP-FfQGqj4Fhh3pH7nVQj.txt
Extracting file HTTP-FsvATF146kf1Emc21j.txt
[...]
```

Here, the `mime_to_ext` table serves two purposes. It defines which mime types to extract and also the file suffix of the extracted files. Extracted files are written to a new `extract_files` subdirectory. Also note that the first conditional in the `file_new` event handler can be removed to make this behavior generic to other protocols besides HTTP.

## 2.3 Bro IDS

An Intrusion Detection System (IDS) allows you to detect suspicious activities happening on your network as a result of a past or active attack. Because of its programming capabilities, Bro can easily be configured to behave like traditional IDSs and detect common attacks with well known patterns, or you can create your own scripts to detect conditions specific to your particular case.

In the following sections, we present a few examples of common uses of Bro as an IDS.

## 2.3.1 Detecting an FTP Brute-force Attack and Notifying

For the purpose of this exercise, we define FTP brute-forcing as too many rejected usernames and passwords occurring from a single address. We start by defining a threshold for the number of attempts, a monitoring interval (in minutes), and a new notice type.

```
detect-bruteforcing.bro

module FTP;

export {
        redef enum Notice::Type += {
                ## Indicates a host bruteforcing FTP logins by watching for too
                ## many rejected usernames or failed passwords.
                Bruteforcing
        };

        ## How many rejected usernames or passwords are required before being
        ## considered to be bruteforcing.
        const bruteforce_threshold: double = 20 &redef;

        ## The time period in which the threshold needs to be crossed before
        ## being reset.
        const bruteforce_measurement_interval = 15mins &redef;
}
```

Using the ftp_reply event, we check for error codes from the 500 series for the "USER" and "PASS" commands, representing rejected usernames or passwords. For this, we can use the `FTP::parse_ftp_reply_code` function to break down the reply code and check if the first digit is a "5" or not. If true, we then use the *Summary Statistics Framework* to keep track of the number of failed attempts.

```
detect-bruteforcing.bro

event ftp_reply(c: connection, code: count, msg: string, cont_resp: bool)
        {
        local cmd = c$ftp$cmdarg$cmd;
        if ( cmd == "USER" || cmd == "PASS" )
                {
                if ( FTP::parse_ftp_reply_code(code)$x == 5 )
                        SumStats::observe("ftp.failed_auth", [$host=c$id$orig_h], [
↪$str=cat(c$id$resp_h)]);
                }
        }
```

Next, we use the SumStats framework to raise a notice of the attack when the number of failed attempts exceeds the specified threshold during the measuring interval.

```
detect-bruteforcing.bro

event bro_init()
        {
        local r1: SumStats::Reducer = [$stream="ftp.failed_auth", $apply=set(SumStats:
↪:UNIQUE), $unique_max=double_to_count(bruteforce_threshold+2)];
        SumStats::create([$name="ftp-detect-bruteforcing",
                          $epoch=bruteforce_measurement_interval,
                          $reducers=set(r1),
                          $threshold_val(key: SumStats::Key, result: SumStats::
↪Result) =
                                        {
                                        return result["ftp.failed_auth"]$num+0.0;
                                        },
                                $threshold=bruteforce_threshold,
                                $threshold_crossed(key: SumStats::Key, result: SumStats::
↪Result) =
                                        {
                                        local r = result["ftp.failed_auth"];
                                        local dur = duration_to_mins_secs(r$end-r$begin);
                                        local plural = r$unique>1 ? "s" : "";
                                        local message = fmt("%s had %d failed logins on %d␣
↪FTP server%s in %s", key$host, r$num, r$unique, plural, dur);
                                        NOTICE([$note=FTP::Bruteforcing,
                                                $src=key$host,
                                                $msg=message,
                                                $identifier=cat(key$host)]);
                                        }]);
        }
```

Below is the final code for our script.

```
detect-bruteforcing.bro

##! FTP brute-forcing detector, triggering when too many rejected usernames or
##! failed passwords have occurred from a single address.

@load base/protocols/ftp
@load base/frameworks/sumstats

@load base/utils/time

module FTP;

export {
        redef enum Notice::Type += {
                ## Indicates a host bruteforcing FTP logins by watching for too
                ## many rejected usernames or failed passwords.
                Bruteforcing
        };

        ## How many rejected usernames or passwords are required before being
        ## considered to be bruteforcing.
        const bruteforce_threshold: double = 20 &redef;

        ## The time period in which the threshold needs to be crossed before
        ## being reset.
        const bruteforce_measurement_interval = 15mins &redef;
}


event bro_init()
        {
        local r1: SumStats::Reducer = [$stream="ftp.failed_auth", $apply=set(SumStats:
↪:UNIQUE), $unique_max=double_to_count(bruteforce_threshold+2)];
        SumStats::create([$name="ftp-detect-bruteforcing",
                                $epoch=bruteforce_measurement_interval,
                                $reducers=set(r1),
                                $threshold_val(key: SumStats::Key, result: SumStats::
↪Result) =
                                  {
                                  return result["ftp.failed_auth"]$num+0.0;
                                  },
                          $threshold=bruteforce_threshold,
                          $threshold_crossed(key: SumStats::Key, result: SumStats::
↪Result) =
                                  {
                                  local r = result["ftp.failed_auth"];
                                  local dur = duration_to_mins_secs(r$end-r$begin);
                                  local plural = r$unique>1 ? "s" : "";
                                  local message = fmt("%s had %d failed logins on %d␣
↪FTP server%s in %s", key$host, r$num, r$unique, plural, dur);
                                  NOTICE([$note=FTP::Bruteforcing,
                                          $src=key$host,
                                          $msg=message,
                                          $identifier=cat(key$host)]);
                                  }]);
        }

event ftp_reply(c: connection, code: count, msg: string, cont_resp: bool)
        {
        local cmd = c$ftp$cmdarg$cmd;
        if ( cmd == "USER" || cmd == "PASS" )
                {
                if ( FTP::parse_ftp_reply_code(code)$x == 5 )
                        SumStats::observe("ftp.failed_auth", [$host=c$id$orig_h], [
↪$str=cat(c$id$resp_h)]);
                }
        }
```

```
# bro -r ftp/bruteforce.pcap protocols/ftp/detect-bruteforcing.bro
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     notice
#open     2016-10-01-03-41-52
#fields   ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p   ␣
↪     fuid     file_mime_type  file_desc       proto   note     msg      sub      src     ␣
↪dst      p        n        peer_descr      actions suppress_for    dropped remote_
↪location.country_code    remote_location.region  remote_location.city    remote_
↪location.latitude        remote_location.longitude
#types    time    string  addr    port    addr    port    string  string  string ␣
↪enum     enum    string  string  addr    addr    port    count   string  set[enum]   ␣
↪     interval        bool    string  string  string  double  double
1389721084.522861 -       -       -       -       -       -       -       -       -    ␣
↪     FTP::Bruteforcing       192.168.56.1 had 20 failed logins on 1 FTP server in␣
↪0m37s    -       192.168.56.1    -       -       -       bro     Notice::ACTION_
↪LOG      3600.000000     F       -       -       -       -       -
#close    2016-10-01-03-41-52
```

As a final note, the detect-bruteforcing.bro script above is included with Bro out of the box. Use this feature by loading this script during startup.

### 2.3.2 Other Attacks

#### Detecting SQL Injection Attacks

#### Checking files against known malware hashes

Files transmitted on your network could either be completely harmless or contain viruses and other threats. One possible action against this threat is to compute the hashes of the files and compare them against a list of known malware hashes. Bro simplifies this task by offering a detect-MHR.bro script that creates and compares hashes against the Malware Hash Registry maintained by Team Cymru. Use this feature by loading this script during startup.

## 2.4 MIME Type Statistics

Files are constantly transmitted over HTTP on regular networks. These files belong to a specific category (e.g., executable, text, image) identified by a Multipurpose Internet Mail Extension (MIME). Although MIME was originally developed to identify the type of non-text attachments on email, it is also used by a web browser to identify the type of files transmitted and present them accordingly.

In this tutorial, we will demonstrate how to use the Sumstats Framework to collect statistical information based on MIME types; specifically, the total number of occurrences, size in bytes, and number of unique hosts transmitting files over HTTP per each type. For instructions on extracting and creating a local copy of these files, visit *this tutorial*.

### 2.4.1 MIME Statistics with Sumstats

When working with the *Summary Statistics Framework*, you need to define three different pieces: (i) Observations, where the event is observed and fed into the framework. (ii) Reducers, where observations are collected and measured. (iii) Sumstats, where the main functionality is implemented.

We start by defining our observation along with a record to store all statistical values and an observation interval. We are conducting our observation on the `HTTP::log_http` event and are interested in the MIME type, size of the file ("response_body_len"), and the originator host ("orig_h"). We use the MIME type as our key and create observers for the other two values.

```
mimestats.bro

module MimeMetrics;

export {

        redef enum Log::ID += { LOG };

        type Info: record {
                ## Timestamp when the log line was finished and written.
                ts:         time   &log;
                ## Time interval that the log line covers.
                ts_delta:   interval &log;
                ## The mime type
                mtype:        string &log;
                ## The number of unique local hosts that fetched this mime type
                uniq_hosts: count  &log;
                ## The number of hits to the mime type
                hits:       count  &log;
                ## The total number of bytes received by this mime type
                bytes:      count  &log;
        };

        ## The frequency of logging the stats collected by this script.
        const break_interval = 5mins &redef;
}
event HTTP::log_http(rec: HTTP::Info)
        {
        if ( Site::is_local_addr(rec$id$orig_h) && rec?$resp_mime_types )
                {
                local mime_type = rec$resp_mime_types[0];
                SumStats::observe("mime.bytes", [$str=mime_type],
                                  [$num=rec$response_body_len]);
                SumStats::observe("mime.hits",  [$str=mime_type],
                                  [$str=cat(rec$id$orig_h)]);
                }
        }
```

Next, we create the reducers. The first will accumulate file sizes and the second will make sure we only store a host ID once. Below is the partial code from a `bro_init` handler.

```
mimestats.bro

        local r1: SumStats::Reducer = [$stream="mime.bytes",
                                       $apply=set(SumStats::SUM)];
        local r2: SumStats::Reducer = [$stream="mime.hits",
                                       $apply=set(SumStats::UNIQUE)];
```

In our final step, we create the SumStats where we check for the observation interval. Once it expires, we populate the record (defined above) with all the relevant data and write it to a log.

```
mimestats.bro

        SumStats::create([$name="mime-metrics",
                          $epoch=break_interval,
                          $reducers=set(r1, r2),
                          $epoch_result(ts: time, key: SumStats::Key, result:␣
↪SumStats::Result) =
                                {
                                local l: Info;
                                l$ts         = network_time();
                                l$ts_delta   = break_interval;
                                l$mtype      = key$str;
                                l$bytes      = double_to_count(floor(result["mime.
↪bytes"]$sum));
                                l$hits       = result["mime.hits"]$num;
                                l$uniq_hosts = result["mime.hits"]$unique;
                                Log::write(MimeMetrics::LOG, l);
                                }]);
```

After putting the three pieces together we end up with the following final code for our script.

```
mimestats.bro

@load base/utils/site
@load base/frameworks/sumstats

redef Site::local_nets += { 10.0.0.0/8 };

```

```
module MimeMetrics;

export {

        redef enum Log::ID += { LOG };

        type Info: record {
                ## Timestamp when the log line was finished and written.
                ts:         time   &log;
                ## Time interval that the log line covers.
                ts_delta:   interval &log;
                ## The mime type
                mtype:        string &log;
                ## The number of unique local hosts that fetched this mime type
                uniq_hosts: count  &log;
                ## The number of hits to the mime type
                hits:       count  &log;
                ## The total number of bytes received by this mime type
                bytes:      count  &log;
        };

        ## The frequency of logging the stats collected by this script.
        const break_interval = 5mins &redef;
}

event bro_init() &priority=3
        {
        Log::create_stream(MimeMetrics::LOG, [$columns=Info, $path="mime_metrics"]);
        local r1: SumStats::Reducer = [$stream="mime.bytes",
                                       $apply=set(SumStats::SUM)];
        local r2: SumStats::Reducer = [$stream="mime.hits",
                                       $apply=set(SumStats::UNIQUE)];
        SumStats::create([$name="mime-metrics",
                          $epoch=break_interval,
                          $reducers=set(r1, r2),
                          $epoch_result(ts: time, key: SumStats::Key, result:␣
 ↪SumStats::Result) =
                                {
                                local l: Info;
                                l$ts         = network_time();
                                l$ts_delta   = break_interval;
                                l$mtype      = key$str;
                                l$bytes      = double_to_count(floor(result["mime.
 ↪bytes"]$sum));
                                l$hits       = result["mime.hits"]$num;
                                l$uniq_hosts = result["mime.hits"]$unique;
                                Log::write(MimeMetrics::LOG, l);
                                }]);
        }

event HTTP::log_http(rec: HTTP::Info)
        {
        if ( Site::is_local_addr(rec$id$orig_h) && rec?$resp_mime_types )
                {
                local mime_type = rec$resp_mime_types[0];
                SumStats::observe("mime.bytes", [$str=mime_type],
                                  [$num=rec$response_body_len]);
                SumStats::observe("mime.hits",  [$str=mime_type],
                                  [$str=cat(rec$id$orig_h)]);
                }
        }
```

```
# bro -r http/bro.org.pcap mimestats.bro
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     mime_metrics
#open     2016-10-01-03-42-12
#fields   ts       ts_delta         mtype    uniq_hosts      hits     bytes
#types    time     interval         string   count    count    count
1389719059.311698 300.000000       text/html        1        2        42231
1389719059.311698 300.000000       image/jpeg       1        1        186859
1389719059.311698 300.000000       application/pgp-signature        1        1        836
1389719059.311698 300.000000       text/plain       1        15       128001
1389719059.311698 300.000000       image/gif        1        1        172
1389719059.311698 300.000000       image/png        1        9        82176
1389719059.311698 300.000000       image/x-icon     1        2        2300
#close    2016-10-01-03-42-12
```

**Note:** The redefinition of `Site::local_nets` is only done inside this script to make it a self-contained example. It's typically redefined somewhere else.

## 2.5 Writing Bro Scripts

**Contents**

### 2.5.1 Understanding Bro Scripts

Bro includes an event-driven scripting language that provides the primary means for an organization to extend and customize Bro's functionality. Virtually all of the output generated by Bro is, in fact, generated by Bro scripts. It's almost easier to consider Bro to be an entity behind-the-scenes processing connections and generating events while Bro's scripting language is the medium through which we mere mortals can achieve communication. Bro scripts effectively notify Bro that should there be an event of a type we define, then let us have the information about the connection so we can perform some function on it. For example, the `ssl.log` file is generated by a Bro script that walks the entire certificate chain and issues notifications if any of the steps along the certificate chain are invalid. This entire process is setup by telling Bro that should it see a server or client issue an SSL `HELLO` message, we want to know about the information about that connection.

It's often easiest to understand Bro's scripting language by looking at a complete script and breaking it down into its identifiable components. In this example, we'll take a look at how Bro checks the SHA1 hash of various files extracted from network traffic against the Team Cymru Malware hash registry. Part of the Team Cymru Malware Hash registry includes the ability to do a host lookup on a domain with the format `<MALWARE_HASH>.malware.hash.cymru.com` where `<MALWARE_HASH>` is the SHA1 hash of a file. Team Cymru also populates the TXT record of their DNS responses with both a "first seen" timestamp and a numerical "detection rate". The important aspect to understand is Bro already generating hashes for files via the Files framework, but it is the script `detect-MHR.bro` that is responsible for generating the appropriate DNS lookup, parsing the response, and generating a notice if appropriate.

```
detect-MHR.bro

##! Detect file downloads that have hash values matching files in Team
##! Cymru's Malware Hash Registry (http://www.team-cymru.org/Services/MHR/).

@load base/frameworks/files
@load base/frameworks/notice
@load frameworks/files/hash-all-files

module TeamCymruMalwareHashRegistry;

export {
	redef enum Notice::Type += {
		## The hash value of a file transferred over HTTP matched in the
		## malware hash registry.
		Match
```

```
        };

        ## File types to attempt matching against the Malware Hash Registry.
        const match_file_types = /application\/x-dosexec/ |
                                 /application\/vnd.ms-cab-compressed/ |
                                 /application\/pdf/ |
                                 /application\/x-shockwave-flash/ |
                                 /application\/x-java-applet/ |
                                 /application\/jar/ |
                                 /video\/mp4/ &redef;

        ## The Match notice has a sub message with a URL where you can get more
        ## information about the file. The %s will be replaced with the SHA-1
        ## hash of the file.
        const match_sub_url = "https://www.virustotal.com/en/search/?query=%s" &redef;

        ## The malware hash registry runs each malware sample through several
        ## A/V engines.  Team Cymru returns a percentage to indicate how
        ## many A/V engines flagged the sample as malicious. This threshold
        ## allows you to require a minimum detection rate.
        const notice_threshold = 10 &redef;
}

function do_mhr_lookup(hash: string, fi: Notice::FileInfo)
        {
        local hash_domain = fmt("%s.malware.hash.cymru.com", hash);

        when ( local MHR_result = lookup_hostname_txt(hash_domain) )
                {
                # Data is returned as "<dateFirstDetected> <detectionRate>"
                local MHR_answer = split_string1(MHR_result, / /);

                if ( |MHR_answer| == 2 )
                        {
                        local mhr_detect_rate = to_count(MHR_answer[1]);

                        if ( mhr_detect_rate >= notice_threshold )
                                {
                                local mhr_first_detected = double_to_time(to_
↪double(MHR_answer[0]));
                                local readable_first_detected = strftime("%Y-%m-%d %H:
↪%M:%S", mhr_first_detected);
                                local message = fmt("Malware Hash Registry Detection␣
↪rate: %d%%  Last seen: %s", mhr_detect_rate, readable_first_detected);
                                local virustotal_url = fmt(match_sub_url, hash);
                                # We don't have the full fa_file record here in order␣
↪to
                                # avoid the "when" statement cloning it (expensive!).
                                local n: Notice::Info = Notice::Info($note=Match,
↪$msg=message, $sub=virustotal_url);
                                Notice::populate_file_info2(fi, n);
                                NOTICE(n);
                                }
                        }
                }
        }

event file_hash(f: fa_file, kind: string, hash: string)
```

```
        {
        if ( kind == "sha1" && f?$info && f$info?$mime_type &&
             match_file_types in f$info$mime_type )
                do_mhr_lookup(hash, Notice::create_file_info(f));
        }
```

Visually, there are three distinct sections of the script. First, there is a base level with no indentation where libraries are included in the script through `@load` and a namespace is defined with `module`. This is followed by an indented and formatted section explaining the custom variables being provided (`export`) as part of the script's namespace. Finally there is a second indented and formatted section describing the instructions to take for a specific event (`event file_hash`). Don't get discouraged if you don't understand every section of the script; we'll cover the basics of the script and much more in following sections.

```
detect-MHR.bro

@load base/frameworks/files
@load base/frameworks/notice
@load frameworks/files/hash-all-files
```

The first part of the script consists of `@load` directives which process the `__load__.bro` script in the respective directories being loaded. The `@load` directives are often considered good practice or even just good manners when writing Bro scripts to make sure they can be used on their own. While it's unlikely that in a full production deployment of Bro these additional resources wouldn't already be loaded, it's not a bad habit to try to get into as you get more experienced with Bro scripting. If you're just starting out, this level of granularity might not be entirely necessary. The `@load` directives are ensuring the Files framework, the Notice framework and the script to hash all files has been loaded by Bro.

```
detect-MHR.bro

export {
        redef enum Notice::Type += {
                ## The hash value of a file transferred over HTTP matched in the
                ## malware hash registry.
                Match
        };

        ## File types to attempt matching against the Malware Hash Registry.
        const match_file_types = /application\/x-dosexec/ |
                                 /application\/vnd.ms-cab-compressed/ |
                                 /application\/pdf/ |
                                 /application\/x-shockwave-flash/ |
                                 /application\/x-java-applet/ |
                                 /application\/jar/ |
                                 /video\/mp4/ &redef;

        ## The Match notice has a sub message with a URL where you can get more
        ## information about the file. The %s will be replaced with the SHA-1
        ## hash of the file.
        const match_sub_url = "https://www.virustotal.com/en/search/?query=%s" &redef;

        ## The malware hash registry runs each malware sample through several
        ## A/V engines.  Team Cymru returns a percentage to indicate how
        ## many A/V engines flagged the sample as malicious. This threshold
        ## allows you to require a minimum detection rate.
        const notice_threshold = 10 &redef;
}
```

The export section redefines an enumerable constant that describes the type of notice we will generate with the Notice framework. Bro allows for re-definable constants, which at first, might seem counter-intuitive. We'll get more in-depth with constants in a later chapter, for now, think of them as variables that can only be altered before Bro starts running. The notice type listed allows for the use of the `NOTICE` function to generate notices of type `TeamCymruMalwareHashRegistry::Match` as done in the next section. Notices allow Bro to generate some kind of extra notification beyond its default log types. Often times, this extra notification comes in the form of an email generated and sent to a preconfigured address, but can be altered depending on the needs of the deployment. The export section is finished off with the definition of a few constants that list the kind of files we want to match against and the minimum percentage of detection threshold in which we are interested.

Up until this point, the script has merely done some basic setup. With the next section, the script starts to define instructions to take in a given event.

```
detect-MHR.bro

function do_mhr_lookup(hash: string, fi: Notice::FileInfo)
        {
        local hash_domain = fmt("%s.malware.hash.cymru.com", hash);

        when ( local MHR_result = lookup_hostname_txt(hash_domain) )
                {
                # Data is returned as "<dateFirstDetected> <detectionRate>"
                local MHR_answer = split_string1(MHR_result, / /);

                if ( |MHR_answer| == 2 )
                        {
                        local mhr_detect_rate = to_count(MHR_answer[1]);

                        if ( mhr_detect_rate >= notice_threshold )
                                {
                                local mhr_first_detected = double_to_time(to_
 ↪double(MHR_answer[0]));
                                local readable_first_detected = strftime("%Y-%m-%d %H:
 ↪%M:%S", mhr_first_detected);
                                local message = fmt("Malware Hash Registry Detection␣
 ↪rate: %d%%  Last seen: %s", mhr_detect_rate, readable_first_detected);
                                local virustotal_url = fmt(match_sub_url, hash);
                                # We don't have the full fa_file record here in order␣
 ↪to
                                # avoid the "when" statement cloning it (expensive!).
                                local n: Notice::Info = Notice::Info($note=Match,
 ↪$msg=message, $sub=virustotal_url);
                                Notice::populate_file_info2(fi, n);
                                NOTICE(n);
                                }
                        }
                }
        }

event file_hash(f: fa_file, kind: string, hash: string)
        {
        if ( kind == "sha1" && f?$info && f$info?$mime_type &&
             match_file_types in f$info$mime_type )
                do_mhr_lookup(hash, Notice::create_file_info(f));
```

The workhorse of the script is contained in the event handler for `file_hash`. The *`file_hash`* event allows scripts to access the information associated with a file for which Bro's file analysis framework has generated a hash. The event handler is passed the file itself as `f`, the type of digest algorithm used as `kind` and the hash generated as `hash`.

In the `file_hash` event handler, there is an `if` statement that is used to check for the correct type of hash, in this case a SHA1 hash. It also checks for a mime type we've defined as being of interest as defined in the constant `match_file_types`. The comparison is made against the expression `f$info$mime_type`, which uses the `$` dereference operator to check the value `mime_type` inside the variable `f$info`. If the entire expression evaluates to true, then a helper function is called to do the rest of the work. In that function, a local variable is defined to hold a string comprised of the SHA1 hash concatenated with `.malware.hash.cymru.com`; this value will be the domain queried in the malware hash registry.

The rest of the script is contained within a `when` block. In short, a `when` block is used when Bro needs to perform asynchronous actions, such as a DNS lookup, to ensure that performance isn't effected. The `when` block performs a DNS TXT lookup and stores the result in the local variable `MHR_result`. Effectively, processing for this event continues and upon receipt of the values returned by `lookup_hostname_txt`, the `when` block is executed. The `when` block splits the string returned into a portion for the date on which the malware was first detected and the detection rate by splitting on an text space and storing the values returned in a local table variable. In the `do_mhr_lookup` function, if the table returned by `split1` has two entries, indicating a successful split, we store the detection date in `mhr_first_detected` and the rate in `mhr_detect_rate` using the appropriate conversion functions. From this point on, Bro knows it has seen a file transmitted which has a hash that has been seen by the Team Cymru Malware Hash Registry, the rest of the script is dedicated to producing a notice.

The detection time is processed into a string representation and stored in `readable_first_detected`. The script then compares the detection rate against the `notice_threshold` that was defined earlier. If the detection rate is high enough, the script creates a concise description of the notice and stores it in the `message` variable. It also creates a possible URL to check the sample against `virustotal.com`'s database, and makes the call to `NOTICE` to hand the relevant information off to the Notice framework.

In approximately a few dozen lines of code, Bro provides an amazing utility that would be incredibly difficult to implement and deploy with other products. In truth, claiming that Bro does this in such a small number of lines is a misdirection; there is a truly massive number of things going on behind-the-scenes in Bro, but it is the inclusion of the scripting language that gives analysts access to those underlying layers in a succinct and well defined manner.

### 2.5.2 The Event Queue and Event Handlers

Bro's scripting language is event driven which is a gear change from the majority of scripting languages with which most users will have previous experience. Scripting in Bro depends on handling the events generated by Bro as it processes network traffic, altering the state of data structures through those events, and making decisions on the information provided. This approach to scripting can often cause confusion to users who come to Bro from a procedural or functional language, but once the initial shock wears off it becomes more clear with each exposure.

Bro's core acts to place events into an ordered "event queue", allowing event handlers to process them on a first-come-first-serve basis. In effect, this is Bro's core functionality as without the scripts written to perform discrete actions on events, there would be little to no usable output. As such, a basic understanding of the event queue, the events being generated, and the way in which event handlers process those events is a basis for not only learning to write scripts for Bro but for understanding Bro itself.

Gaining familiarity with the specific events generated by Bro is a big step towards building a mind set for working with Bro scripts. The majority of events generated by Bro are defined in the built-in-function files or `.bif` files which also act as the basis for online event documentation. These in-line comments are compiled into an online documentation system using Broxygen. Whether starting a script from scratch or reading and maintaining someone else's script, having the built-in event definitions available is an excellent resource to have on hand. For the 2.0 release the Bro developers put significant effort into organization and documentation of every event. This effort resulted in built-in-function files organized such that each entry contains a descriptive event name, the arguments passed to the event, and a concise explanation of the functions use.

```
Bro_DNS.events.bif.bro

## Generated for DNS requests. For requests with multiple queries, this event
## is raised once for each.
##
## See `Wikipedia <http://en.wikipedia.org/wiki/Domain_Name_System>`__ for more
## information about the DNS protocol. Bro analyzes both UDP and TCP DNS
## sessions.
##
## c: The connection, which may be UDP or TCP depending on the type of the
##    transport-layer session being analyzed.
##
## msg: The parsed DNS message header.
##
## query: The queried name.
##
## qtype: The queried resource record type.
##
## qclass: The queried resource record class.
##
## .. bro:see:: dns_AAAA_reply dns_A_reply dns_CNAME_reply dns_EDNS_addl
##    dns_HINFO_reply dns_MX_reply dns_NS_reply dns_PTR_reply dns_SOA_reply
##    dns_SRV_reply dns_TSIG_addl dns_TXT_reply dns_WKS_reply dns_end
##    dns_full_request dns_mapping_altered dns_mapping_lost_name dns_mapping_new_name
##    dns_mapping_unverified dns_mapping_valid dns_message dns_query_reply
##    dns_rejected non_dns_request dns_max_queries dns_session_timeout dns_skip_addl
##    dns_skip_all_addl dns_skip_all_auth dns_skip_auth
global dns_request: event(c: connection , msg: dns_msg , query: string , qtype: count␣
↪, qclass: count );
```

Above is a segment of the documentation for the event *dns_request* (and the preceding link points to the documentation generated out of that). It's organized such that the documentation, commentary, and list of arguments precede the actual event definition used by Bro. As Bro detects DNS requests being issued by an originator, it issues this event and any number of scripts then have access to the data Bro passes along with the event. In this example, Bro passes not only the message, the query, query type and query class for the DNS request, but also a record used for the connection itself.

### 2.5.3 The Connection Record Data Type

Of all the events defined by Bro, an overwhelmingly large number of them are passed the `connection` record data type, in effect, making it the backbone of many scripting solutions. The connection record itself, as we will see in a moment, is a mass of nested data types used to track state on a connection through its lifetime. Let's walk through the process of selecting an appropriate event, generating some output to standard out and dissecting the connection record so as to get an overview of it. We will cover data types in more detail later.

While Bro is capable of packet level processing, its strengths lay in the context of a connection between an originator and a responder. As such, there are events defined for the primary parts of the connection life-cycle as you'll see from the small selection of connection-related events below.

```
event.bif.bro

## Generated for every new connection. This event is raised with the first
## packet of a previously unknown connection. Bro uses a flow-based definition
## of "connection" here that includes not only TCP sessions but also UDP and
## ICMP flows.
global new_connection: event(c: connection );
## Generated when a TCP connection timed out. This event is raised when
## no activity was seen for an interval of at least
## :bro:id:`tcp_connection_linger`, and either one endpoint has already
## closed the connection or one side never became active.
global connection_timeout: event(c: connection );
## Generated when a connection's internal state is about to be removed from
## memory. Bro generates this event reliably once for every connection when it
## is about to delete the internal state. As such, the event is well-suited for
## script-level cleanup that needs to be performed for every connection.  This
## event is generated not only for TCP sessions but also for UDP and ICMP
## flows.
global connection_state_remove: event(c: connection );
```

Of the events listed, the event that will give us the best insight into the connection record data type will be `connection_state_remove` . As detailed in the in-line documentation, Bro generates this event just before it decides to remove this event from memory, effectively forgetting about it. Let's take a look at a simple example script, that will output the connection record for a single connection.

```
connection_record_01.bro

@load base/protocols/conn

event connection_state_remove(c: connection)
    {
    print c;
    }
```

Again, we start with `@load`, this time importing the /scripts/base/protocols/conn/index scripts which supply the tracking and logging of general information and state of connections. We handle the `connection_state_remove` event and simply print the contents of the argument passed to it. For this example we're going to run Bro in "bare mode" which loads only the minimum number of scripts to retain operability and leaves the burden of loading required scripts to the script being run. While bare mode is a low level functionality incorporated into Bro, in this case, we're going to use it to demonstrate how different features of Bro add more and more layers of information about a connection. This will give us a chance to see the contents of the connection record without it being overly populated.

```
# bro -b -r http/get.trace connection_record_01.bro
[id=[orig_h=141.142.228.5, orig_p=59856/tcp, resp_h=192.150.187.43, resp_p=80/tcp],␣
↪orig=[size=136, state=5, num_pkts=7, num_bytes_ip=512, flow_label=0],␣
↪resp=[size=5007, state=5, num_pkts=7, num_bytes_ip=5379, flow_label=0], start_
↪time=1362692526.869344, duration=0.211484, service={

}, history=ShADadFf, uid=CXWv6p3arKYeMETxOg, tunnel=<uninitialized>,␣
↪conn=[ts=1362692526.869344, uid=CXWv6p3arKYeMETxOg, id=[orig_h=141.142.228.5, orig_
↪p=59856/tcp, resp_h=192.150.187.43, resp_p=80/tcp], proto=tcp, service=
↪<uninitialized>, duration=0.211484, orig_bytes=136, resp_bytes=5007, conn_state=SF,␣
↪local_orig=<uninitialized>, local_resp=<uninitialized>, missed_bytes=0,␣
↪history=ShADadFf, orig_pkts=7, orig_ip_bytes=512, resp_pkts=7, resp_ip_bytes=5379,␣
↪tunnel_parents={

}], extract_orig=F, extract_resp=F, thresholds=<uninitialized>]
```

As you can see from the output, the connection record is something of a jumble when printed on its own. Regularly taking a peek at a populated connection record helps to understand the relationship between its fields as well as allowing an opportunity to build a frame of reference for accessing data in a script.

Bro makes extensive use of nested data structures to store state and information gleaned from the analysis of a connection as a complete unit. To break down this collection of information, you will have to make use of Bro's field delimiter `$`. For example, the originating host is referenced by `c$id$orig_h` which if given a narrative relates to `orig_h` which is a member of `id` which is a member of the data structure referred to as `c` that was passed into the event handler. Given that the responder port `c$id$resp_p` is `80/tcp`, it's likely that Bro's base HTTP scripts can further populate the connection record. Let's load the `base/protocols/http` scripts and check the output of our script.

Bro uses the dollar sign as its field delimiter and a direct correlation exists between the output of the connection record and the proper format of a dereferenced variable in scripts. In the output of the script above, groups of information are collected between brackets, which would correspond to the `$`-delimiter in a Bro script.

```
connection_record_02.bro

@load base/protocols/conn
@load base/protocols/http

event connection_state_remove(c: connection)
    {
    print c;
    }
```

```
# bro -b -r http/get.trace connection_record_02.bro
[id=[orig_h=141.142.228.5, orig_p=59856/tcp, resp_h=192.150.187.43, resp_p=80/tcp],␣
↪orig=[size=136, state=5, num_pkts=7, num_bytes_ip=512, flow_label=0],␣
↪resp=[size=5007, state=5, num_pkts=7, num_bytes_ip=5379, flow_label=0], start_
↪time=1362692526.869344, duration=0.211484, service={

}, history=ShADadFf, uid=CXWv6p3arKYeMETxOg, tunnel=<uninitialized>,␣
↪conn=[ts=1362692526.869344, uid=CXWv6p3arKYeMETxOg, id=[orig_h=141.142.228.5, orig_
↪p=59856/tcp, resp_h=192.150.187.43, resp_p=80/tcp], proto=tcp, service=
↪<uninitialized>, duration=0.211484, orig_bytes=136, resp_bytes=5007, conn_state=SF,␣
↪local_orig=<uninitialized>, local_resp=<uninitialized>, missed_bytes=0,␣
↪history=ShADadFf, orig_pkts=7, orig_ip_bytes=512, resp_pkts=7, resp_ip_bytes=5379,␣
↪tunnel_parents={

}], extract_orig=F, extract_resp=F, thresholds=<uninitialized>, http=[ts=1362692526.
↪939527, uid=CXWv6p3arKYeMETxOg, id=[orig_h=141.142.228.5, orig_p=59856/tcp, resp_
↪h=192.150.187.43, resp_p=80/tcp], trans_depth=1, method=GET, host=bro.org, uri=/
↪download/CHANGES.bro-aux.txt, referrer=<uninitialized>, user_agent=Wget/1.14␣
↪(darwin12.2.0), request_body_len=0, response_body_len=4705, status_code=200, status_
↪msg=OK, info_code=<uninitialized>, info_msg=<uninitialized>, filename=
↪<uninitialized>, tags={

}, username=<uninitialized>, password=<uninitialized>, capture_password=F, proxied=
↪<uninitialized>, range_request=F, orig_fuids=<uninitialized>, orig_mime_types=
↪<uninitialized>, resp_fuids=[FakNcS1Jfe01uljb3], resp_mime_types=[text/plain],␣
↪current_entity=<uninitialized>, orig_mime_depth=1, resp_mime_depth=1], http_
↪state=[pending={

}, current_request=1, current_response=1, trans_depth=1]]
```

The addition of the `base/protocols/http` scripts populates the `http=[]` member of the connection record. While Bro is doing a massive amount of work in the background, it is in what is commonly called "script-land" that details are being refined and decisions being made. Were we to continue running in "bare mode" we could slowly keep adding infrastructure through `@load` statements. For example, were we to `@load base/frameworks/logging`, Bro would generate a `conn.log` and `http.log` for us in the current working directory. As mentioned above, including the appropriate `@load` statements is not only good practice, but can also help to indicate which functionalities are being used in a script. Take a second to run the script without the `-b` flag and check the output when all of Bro's functionality is applied to the trace file.

## 2.5.4 Data Types and Data Structures

### Scope

Before embarking on a exploration of Bro's native data types and data structures, it's important to have a good grasp of the different levels of scope available in Bro and the appropriate times to use them within a script. The declarations of variables in Bro come in two forms. Variables can be declared with or without a definition in the form `SCOPE name: TYPE` or `SCOPE name = EXPRESSION` respectively; each of which produce the same result if `EXPRESSION` evaluates to the same type as `TYPE`. The decision as to which type of declaration to use is likely to be dictated by personal preference and readability.

```
data_type_declaration.bro

event bro_init()
    {
    local a: int;
    a = 10;
    local b = 10;

    if ( a == b )
        print fmt("A: %d, B: %d", a, b);
    }
```

### Global Variables

A global variable is used when the state of variable needs to be tracked, not surprisingly, globally. While there are some caveats, when a script declares a variable using the global scope, that script is granting access to that variable from other scripts. However, when a script uses the `module` keyword to give the script a namespace, more care must be given to the declaration of globals to ensure the intended result. When a global is declared in a script with a namespace there are two possible outcomes. First, the variable is available only within the context of the namespace. In this scenario, other scripts within the same namespace will have access to the variable declared while scripts using a different namespace or no namespace altogether will not have access to the variable. Alternatively, if a global variable is declared within an `export { ... }` block that variable is available to any other script through the naming convention of `MODULE::variable_name`.

The declaration below is taken from the /scripts/policy/protocols/conn/known-hosts.bro script and declares a variable called `known_hosts` as a global set of unique IP addresses within the `Known` namespace and exports it for use outside of the `Known` namespace. Were we to want to use the `known_hosts` variable we'd be able to access it through `Known::known_hosts`.

```
known-hosts.bro

module Known;

export {
        global known_hosts: set[addr] &create_expire=1day &synchronized &redef;
}
```

The sample above also makes use of an `export { ... }` block. When the module keyword is used in a script, the variables declared are said to be in that module's "namespace". Where as a global variable can be accessed by its name alone when it is not declared within a module, a global variable declared within a module must be exported and then accessed via `MODULE_NAME::VARIABLE_NAME`. As in the example above, we would be able to access the `known_hosts` in a separate script variable via `Known::known_hosts` due to the fact that `known_hosts` was declared as a global variable within an export block under the `Known` namespace.

### Constants

Bro also makes use of constants, which are denoted by the `const` keyword. Unlike globals, constants can only be set or altered at parse time if the `&redef` attribute has been used. Afterwards (in run-time) the constants are unalterable. In most cases, re-definable constants are used in Bro scripts as containers for configuration options. For example, the configuration option to log passwords decrypted from HTTP streams is stored in `HTTP::default_capture_password` as shown in the stripped down excerpt from /scripts/base/protocols/http/main.bro below.

Because the constant was declared with the `&redef` attribute, if we needed to turn this option on globally, we could do so by adding the following line to our `site/local.bro` file before firing up Bro.

```
data_type_const_simple.bro

@load base/protocols/http

redef HTTP::default_capture_password = T;
```

While the idea of a re-definable constant might be odd, the constraint that constants can only be altered at parse-time remains even with the `&redef` attribute. In the code snippet below, a table of strings indexed by ports is declared as a constant before two values are added to the table through `redef` statements. The table is then printed in a `bro_init` event. Were we to try to alter the table in an event handler, Bro would notify the user of an error and the script would fail.

```
data_type_const.bro

const port_list: table[port] of string &redef;

redef port_list += { [6666/tcp] = "IRC"};
redef port_list += { [80/tcp] = "WWW" };

event bro_init()
    {
    print port_list;
    }
```

```
# bro -b data_type_const.bro
{
[6666/tcp] = IRC,
[80/tcp] = WWW
}
```

### Local Variables

Whereas globals and constants are widely available in scriptland through various means, when a variable is defined with a local scope, its availability is restricted to the body of the event or function in which it was declared. Local variables tend to be used for values that are only needed within a specific scope and once the processing of a script passes beyond that scope and no longer used, the variable is deleted. Bro maintains names of locals separately from globally visible ones, an example of which is illustrated below.

```
data_type_local.bro

function add_two(i: count): count
    {
    local added_two = i+2;
    print fmt("i + 2 = %d", added_two);
    return added_two;
    }

event bro_init()
    {
    local test = add_two(10);
    }
```

The script executes the event handler `bro_init` which in turn calls the function `add_two(i: count)` with an argument of `10`. Once Bro enters the `add_two` function, it provisions a locally scoped variable called `added_two` to hold the value of `i+2`, in this case, `12`. The `add_two` function then prints the value of the `added_two` variable and returns its value to the `bro_init` event handler. At this point, the variable `added_two` has fallen out of scope and no longer exists while the value `12` still in use and stored in the locally scoped variable `test`. When Bro finishes processing the `bro_init` function, the variable called `test` is no longer in scope and, since there exist no other references to the value `12`, the value is also deleted.

## Data Structures

It's difficult to talk about Bro's data types in a practical manner without first covering the data structures available in Bro. Some of the more interesting characteristics of data types are revealed when used inside of a data structure, but given that data structures are made up of data types, it devolves rather quickly into a "chicken-and-egg" problem. As such, we'll introduce data types from a bird's eye view before diving into data structures and from there a more complete exploration of data types.

The table below shows the atomic types used in Bro, of which the first four should seem familiar if you have some scripting experience, while the remaining six are less common in other languages. It should come as no surprise that a scripting language for a Network Security Monitoring platform has a fairly robust set of network-centric data types and taking note of them here may well save you a late night of reinventing the wheel.

| Data Type | Description |
|---|---|
| int | 64 bit signed integer |
| count | 64 bit unsigned integer |
| double | double precision floating precision |
| bool | boolean (T/F) |
| addr | IP address, IPv4 and IPv6 |
| port | transport layer port |
| subnet | CIDR subnet mask |
| time | absolute epoch time |
| interval | a time interval |
| pattern | regular expression |

## Sets

Sets in Bro are used to store unique elements of the same data type. In essence, you can think of them as "a unique set of integers" or "a unique set of IP addresses". While the declaration of a set may differ based on the data type being collected, the set will always contain unique elements and the elements in the set will always be of the same data type. Such requirements make the set data type perfect for information that is already naturally unique such as ports or IP addresses. The code snippet below shows both an explicit and implicit declaration of a locally scoped set.

```
data_struct_set_declaration.bro

event bro_init()
```

```
    {
    local ssl_ports: set[port];
    local non_ssl_ports = set( 23/tcp, 80/tcp, 143/tcp, 25/tcp );
    }
```

As you can see, sets are declared using the format `SCOPE var_name: set[TYPE]`. Adding and removing elements in a set is achieved using the `add` and `delete` statements. Once you have elements inserted into the set, it's likely that you'll need to either iterate over that set or test for membership within the set, both of which are covered by the `in` operator. In the case of iterating over a set, combining the `for` statement and the `in` operator will allow you to sequentially process each element of the set as seen below.

```
data_struct_set_declaration.bro

    for ( i in ssl_ports )
        print fmt("SSL Port: %s", i);

    for ( i in non_ssl_ports )
        print fmt("Non-SSL Port: %s", i);
```

Here, the `for` statement loops over the contents of the set storing each element in the temporary variable `i`. With each iteration of the `for` loop, the next element is chosen. Since sets are not an ordered data type, you cannot guarantee the order of the elements as the `for` loop processes.

To test for membership in a set the `in` statement can be combined with an `if` statement to return a true or false value. If the exact element in the condition is already in the set, the condition returns true and the body executes. The `in` statement can also be negated by the `!` operator to create the inverse of the condition. While we could rewrite the corresponding line below as `if ( !( 587/tcp in ssl_ports ))` try to avoid using this construct; instead, negate the in operator itself. While the functionality is the same, using the `!in` is more efficient as well as a more natural construct which will aid in the readability of your script.

```
data_struct_set_declaration.bro

    # Check for SMTPS
    if ( 587/tcp !in ssl_ports )
        add ssl_ports[587/tcp];
```

You can see the full script and its output below.

```
data_struct_set_declaration.bro

event bro_init()
    {
    local ssl_ports: set[port];
    local non_ssl_ports = set( 23/tcp, 80/tcp, 143/tcp, 25/tcp );

    # SSH
    add ssl_ports[22/tcp];
    # HTTPS
    add ssl_ports[443/tcp];
    # IMAPS
    add ssl_ports[993/tcp];

    # Check for SMTPS
    if ( 587/tcp !in ssl_ports )
        add ssl_ports[587/tcp];

    for ( i in ssl_ports )
```

```
        print fmt("SSL Port: %s", i);

    for ( i in non_ssl_ports )
        print fmt("Non-SSL Port: %s", i);
    }
```

```
# bro data_struct_set_declaration.bro
SSL Port: 993/tcp
SSL Port: 22/tcp
SSL Port: 587/tcp
SSL Port: 443/tcp
Non-SSL Port: 143/tcp
Non-SSL Port: 25/tcp
Non-SSL Port: 80/tcp
Non-SSL Port: 23/tcp
```

### Tables

A table in Bro is a mapping of a key to a value or yield. While the values don't have to be unique, each key in the table must be unique to preserve a one-to-one mapping of keys to values.

```
data_struct_table_declaration.bro

event bro_init()
    {
    # Declaration of the table.
    local ssl_services: table[string] of port;

    # Initialize the table.
    ssl_services = table(["SSH"] = 22/tcp, ["HTTPS"] = 443/tcp);

    # Insert one key-yield pair into the table.
    ssl_services["IMAPS"] = 993/tcp;

    # Check if the key "SMTPS" is not in the table.
    if ( "SMTPS" !in ssl_services )
        ssl_services["SMTPS"] = 587/tcp;

    # Iterate over each key in the table.
    for ( k in ssl_services )
        print fmt("Service Name:  %s - Common Port: %s", k, ssl_services[k]);
    }
```

```
# bro data_struct_table_declaration.bro
Service Name:  IMAPS - Common Port: 993/tcp
Service Name:  HTTPS - Common Port: 443/tcp
Service Name:  SSH - Common Port: 22/tcp
Service Name:  SMTPS - Common Port: 587/tcp
```

In this example, we've compiled a table of SSL-enabled services and their common ports. The explicit declaration and constructor for the table are on two different lines and lay out the data types of the keys (strings) and the data types of the yields (ports) and then fill in some sample key and yield pairs. You can also use a table accessor to insert one key-yield pair into the table. When using the `in` operator on a table, you are effectively working with the keys of the table. In the case of an `if` statement, the `in` operator will check for membership among the set of keys and return a true or false value. The example shows how to check if `SMTPS` is not in the set of keys for the `ssl_services` table and if the condition holds true, we add the key-yield pair to the table. Finally, the example shows how to use a `for` statement to iterate over each key currently in the table.

Simple examples aside, tables can become extremely complex as the keys and values for the table become more intricate. Tables can have keys comprised of multiple data types and even a series of elements called a "tuple". The flexibility gained with the use of complex tables in Bro implies a cost in complexity for the person writing the scripts but pays off in effectiveness given the power of Bro as a network security platform.

```
data_struct_table_complex.bro

event bro_init()
    {
    local samurai_flicks: table[string, string, count, string] of string;

    samurai_flicks["Kihachi Okamoto", "Toho", 1968, "Tatsuya Nakadai"] = "Kiru";
    samurai_flicks["Hideo Gosha", "Fuji", 1969, "Tatsuya Nakadai"] = "Goyokin";
    samurai_flicks["Masaki Kobayashi", "Shochiku Eiga", 1962, "Tatsuya Nakadai" ] =
↪"Harakiri";
    samurai_flicks["Yoji Yamada", "Eisei Gekijo", 2002, "Hiroyuki Sanada" ] =
↪"Tasogare Seibei";

    for ( [d, s, y, a] in samurai_flicks )
        print fmt("%s was released in %d by %s studios, directed by %s and starring %s
↪", samurai_flicks[d, s, y, a], y, s, d, a);
    }
```

```
# bro -b data_struct_table_complex.bro
Kiru was released in 1968 by Toho studios, directed by Kihachi Okamoto and starring␣
↪Tatsuya Nakadai
Goyokin was released in 1969 by Fuji studios, directed by Hideo Gosha and starring␣
↪Tatsuya Nakadai
Harakiri was released in 1962 by Shochiku Eiga studios, directed by Masaki Kobayashi␣
↪and starring Tatsuya Nakadai
Tasogare Seibei was released in 2002 by Eisei Gekijo studios, directed by Yoji Yamada␣
↪and starring Hiroyuki Sanada
```

This script shows a sample table of strings indexed by two strings, a count, and a final string. With a tuple acting as an aggregate key, the order is important as a change in order would result in a new key. Here, we're using the table to track the director, studio, year or release, and lead actor in a series of samurai flicks. It's important to note that in the case of the `for` statement, it's an all or nothing kind of iteration. We cannot iterate over, say, the directors; we have to iterate with the exact format as the keys themselves. In this case, we need squared brackets surrounding four temporary variables to act as a collection for our iteration. While this is a contrived example, we could easily have had keys containing IP addresses (`addr`), ports (`port`) and even a `string` calculated as the result of a reverse hostname lookup.

## Vectors

If you're coming to Bro with a programming background, you may or may not be familiar with a vector data type depending on your language of choice. On the surface, vectors perform much of the same functionality as associative arrays with unsigned integers as their indices. They are however more efficient than that and they allow for ordered access. As such any time you need to sequentially store data of the same type, in Bro you should reach for a vector. Vectors are a collection of objects, all of which are of the same data type, to which elements can be dynamically added or removed. Since Vectors use contiguous storage for their elements, the contents of a vector can be accessed through a zero-indexed numerical offset.

The format for the declaration of a Vector follows the pattern of other declarations, namely, `SCOPE v: vector` of T where v is the name of your vector, and T is the data type of its members. For example, the following snippet shows an explicit and implicit declaration of two locally scoped vectors. The script populates the first vector by inserting values at the end; it does that by placing the vector name between two vertical pipes to get the vector's current length before printing the contents of both Vectors and their current lengths.

```
data_struct_vector_declaration.bro

event bro_init()
    {
    local v1: vector of count;
    local v2 = vector(1, 2, 3, 4);

    v1[|v1|] = 1;
    v1[|v1|] = 2;
    v1[|v1|] = 3;
    v1[|v1|] = 4;

    print fmt("contents of v1: %s", v1);
    print fmt("length of v1: %d", |v1|);
    print fmt("contents of v2: %s", v2);
    print fmt("length of v2: %d", |v2|);
    }
```

```
# bro data_struct_vector_declaration.bro
contents of v1: [1, 2, 3, 4]
length of v1: 4
contents of v2: [1, 2, 3, 4]
length of v2: 4
```

In a lot of cases, storing elements in a vector is simply a precursor to then iterating over them. Iterating over a vector is easy with the for keyword. The sample below iterates over a vector of IP addresses and for each IP address, masks that address with 18 bits. The for keyword is used to generate a locally scoped variable called i which will hold the index of the current element in the vector. Using i as an index to addr_vector we can access the current item in the vector with addr_vector[i].

```
data_struct_vector_iter.bro

event bro_init()
    {
    local addr_vector: vector of addr = vector(1.2.3.4, 2.3.4.5, 3.4.5.6);

    for (i in addr_vector)
        print mask_addr(addr_vector[i], 18);
    }
```

```
# bro -b data_struct_vector_iter.bro
1.2.0.0/18
2.3.0.0/18
3.4.0.0/18
```

## Data Types Revisited

### addr

The `addr`, or address, data type manages to cover a surprisingly large amount of ground while remaining succinct. IPv4, IPv6 and even hostname constants are included in the `addr` data type. While IPv4 addresses use the default dotted quad formatting, IPv6 addresses use the RFC 2373 defined notation with the addition of squared brackets wrapping the entire address. When you venture into hostname constants, Bro performs a little slight of hand for the benefit of the user; a hostname constant is, in fact, a set of addresses. Bro will issue a DNS request when it sees a hostname constant in use and return a set whose elements are the answers to the DNS request. For example, if you were to use `local google = www.google.com;` you would end up with a locally scoped `set[addr]` with elements that represent the current set of round robin DNS entries for google. At first blush, this seems trivial, but it is yet another example of Bro making the life of the common Bro scripter a little easier through abstraction applied in a practical manner. (Note however that these IP addresses will never get updated during Bro's processing, so often this mechanism most useful for addresses that are expected to remain static.).

### port

Transport layer port numbers in Bro are represented in the format of `<unsigned integer>/<protocol name>`, e.g., `22/tcp` or `53/udp`. Bro supports TCP(`/tcp`), UDP(`/udp`), ICMP(`/icmp`) and UNKNOWN(`/unknown`) as protocol designations. While ICMP doesn't have an actual port, Bro supports the concept of ICMP "ports" by using the ICMP message type and ICMP message code as the source and destination port respectively. Ports can be compared for equality using the == or != operators and can even be compared for ordering. Bro gives the protocol designations the following "order": `unknown` < `tcp` < `udp` < `icmp`. For example `65535/tcp` is smaller than `0/udp`.

### subnet

Bro has full support for CIDR notation subnets as a base data type. There is no need to manage the IP and the subnet mask as two separate entities when you can provide the same information in CIDR notation in your scripts. The following example below uses a Bro script to determine if a series of IP addresses are within a set of subnets using a 20 bit subnet mask.

```
data_type_subnets.bro

event bro_init()
    {
    local subnets = vector(172.16.0.0/20, 172.16.16.0/20, 172.16.32.0/20, 172.16.48.0/
↪20);
    local addresses = vector(172.16.4.56, 172.16.47.254, 172.16.22.45, 172.16.1.1);

    for ( a in addresses )
        {
        for ( s in subnets )
            {
            if ( addresses[a] in subnets[s] )
                print fmt("%s belongs to subnet %s", addresses[a], subnets[s]);
            }
        }

    }
```

Because this is a script that doesn't use any kind of network analysis, we can handle the event `bro_init` which is always generated by Bro's core upon startup. In the example script, two locally scoped vectors are created to hold our lists of subnets and IP addresses respectively. Then, using a set of nested `for` loops, we iterate over every subnet and every IP address and use an `if` statement to compare an IP address against a subnet using the `in` operator. The `in` operator returns true if the IP address falls within a given subnet based on the longest prefix match calculation. For example, `10.0.0.1 in 10.0.0.0/8` would return true while `192.168.2.1 in 192.168.1.0/24` would return false. When we run the script, we get the output listing the IP address and the subnet in which it belongs.

```
# bro data_type_subnets.bro
172.16.4.56 belongs to subnet 172.16.0.0/20
172.16.47.254 belongs to subnet 172.16.32.0/20
172.16.22.45 belongs to subnet 172.16.16.0/20
172.16.1.1 belongs to subnet 172.16.0.0/20
```

### time

While there is currently no supported way to add a time constant in Bro, two built-in functions exist to make use of the `time` data type. Both `network_time` and `current_time` return a `time` data type but they each return a time based on different criteria. The `current_time` function returns what is called the wall-clock time as defined by the operating system. However, `network_time` returns the timestamp of the last packet processed be it from a live data stream or saved packet capture. Both functions return the time in epoch seconds, meaning `strftime` must be used to turn the output into human readable output. The script below makes use of the *`connection_established`* event handler to generate text every time a SYN/ACK packet is seen responding to a SYN packet as part of a TCP handshake. The text generated, is in the format of a timestamp and an indication of who the originator and responder were. We use the `strftime` format string of `%Y%M%d %H:%m:%S` to produce a common date time formatted time stamp.

```
data_type_time.bro

event connection_established(c: connection)
    {
    print fmt("%s:  New connection established from %s to %s\n", strftime("%Y/%M/%d
↪%H:%m:%S", network_time()), c$id$orig_h, c$id$resp_h);
    }
```

When the script is executed we get an output showing the details of established connections.

```
# bro -r wikipedia.trace data_type_time.bro
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪118\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪3\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪3\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪3\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪3\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪3\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪3\x0a
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪2\x0a
2011/06/18 19:03:09:  New connection established from 141.142.220.235 to 173.192.163.
↪128\x0a
```

### interval

The interval data type is another area in Bro where rational application of abstraction makes perfect sense. As a data type, the interval represents a relative time as denoted by a numeric constant followed by a unit of time. For example, 2.2 seconds would be `2.2sec` and thirty-one days would be represented by `31days`. Bro supports `usec`, `msec`, `sec`, `min`, `hr`, or `day` which represent microseconds, milliseconds, seconds, minutes, hours, and days respectively. In fact, the interval data type allows for a surprising amount of variation in its definitions. There can be a space between the numeric constant or they can be crammed together like a temporal portmanteau. The time unit can be either singular or plural. All of this adds up to to the fact that both `42hrs` and `42 hr` are perfectly valid and logically equivalent in Bro. The point, however, is to increase the readability and thus maintainability of a script. Intervals can even be negated, allowing for `-10mins` to represent "ten minutes ago".

Intervals in Bro can have mathematical operations performed against them allowing the user to perform addition, subtraction, multiplication, division, and comparison operations. As well, Bro returns an interval when comparing two `time` values using the `-` operator. The script below amends the script started in the section above to include a time delta value printed along with the connection establishment report.

```
data_type_interval.bro

# Store the time the previous connection was established.
global last_connection_time: time;

# boolean value to indicate whether we have seen a previous connection.
global connection_seen: bool = F;

event connection_established(c: connection)
    {
    local net_time: time  = network_time();

    print fmt("%s:  New connection established from %s to %s", strftime("%Y/%M/%d %H:
↪%m:%S", net_time), c$id$orig_h, c$id$resp_h);

    if ( connection_seen )
        print fmt("     Time since last connection: %s", net_time - last_connection_
↪time);

    last_connection_time = net_time;
    connection_seen = T;
    }
```

This time, when we execute the script we see an additional line in the output to display the time delta since the last fully established connection.

```
# bro -r wikipedia.trace data_type_interval.bro
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.
↪118
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.3
     Time since last connection: 132.0 msecs 97.0 usecs
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.3
     Time since last connection: 177.0 usecs
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.3
     Time since last connection: 2.0 msecs 177.0 usecs
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.3
     Time since last connection: 33.0 msecs 898.0 usecs
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.3
     Time since last connection: 35.0 usecs
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.3
```

```
     Time since last connection: 2.0 msecs 532.0 usecs
2011/06/18 19:03:08:  New connection established from 141.142.220.118 to 208.80.152.2
     Time since last connection: 7.0 msecs 866.0 usecs
2011/06/18 19:03:09:  New connection established from 141.142.220.235 to 173.192.163.
↪128
     Time since last connection: 817.0 msecs 703.0 usecs
```

### Pattern

Bro has support for fast text searching operations using regular expressions and even goes so far as to declare a native data type for the patterns used in regular expressions. A pattern constant is created by enclosing text within the forward slash characters. Bro supports syntax very similar to the Flex lexical analyzer syntax. The most common use of patterns in Bro you are likely to come across is embedded matching using the `in` operator. Embedded matching adheres to a strict format, requiring the regular expression or pattern constant to be on the left side of the `in` operator and the string against which it will be tested to be on the right.

```
data_type_pattern_01.bro

event bro_init()
    {
    local test_string = "The quick brown fox jumps over the lazy dog.";
    local test_pattern = /quick|lazy/;

    if ( test_pattern in test_string )
        {
        local results = split(test_string, test_pattern);
        print results[1];
        print results[2];
        print results[3];
        }
    }
```

In the sample above, two local variables are declared to hold our sample sentence and regular expression. Our regular expression in this case will return true if the string contains either the word `quick` or the word `lazy`. The `if` statement in the script uses embedded matching and the `in` operator to check for the existence of the pattern within the string. If the statement resolves to true, `split` is called to break the string into separate pieces. `Split` takes a string and a pattern as its arguments and returns a table of strings indexed by a count. Each element of the table will be the segments before and after any matches against the pattern but excluding the actual matches. In this case, our pattern matches twice, and results in a table with three entries. The `print` statements in the script will print the contents of the table in order.

```
# bro data_type_pattern_01.bro
The
 brown fox jumps over the
 dog.
```

Patterns can also be used to compare strings using equality and inequality operators through the == and != operators respectively. When used in this manner however, the string must match entirely to resolve to true. For example, the script below uses two ternary conditional statements to illustrate the use of the == operator with patterns. The output is altered based on the result of the comparison between the pattern and the string.

```
data_type_pattern_02.bro

event bro_init()
    {
```

```
    local test_string = "equality";

    local test_pattern = /equal/;
    print fmt("%s and %s %s equal", test_string, test_pattern, test_pattern == test_
↪string ? "are" : "are not");

    test_pattern = /equality/;
    print fmt("%s and %s %s equal", test_string, test_pattern, test_pattern == test_
↪string ? "are" : "are not");
    }
```

```
# bro data_type_pattern_02.bro
equality and /^?(equal)$?/ are not equal
equality and /^?(equality)$?/ are equal
```

### Record Data Type

With Bro's support for a wide array of data types and data structures, an obvious extension is to include the ability to create custom data types composed of atomic types and further data structures. To accomplish this, Bro introduces the `record` type and the `type` keyword. Similar to how you would define a new data structure in C with the `typedef` and `struct` keywords, Bro allows you to cobble together new data types to suit the needs of your situation.

When combined with the `type` keyword, `record` can generate a composite type. We have, in fact, already encountered a complex example of the `record` data type in the earlier sections, the `connection` record passed to many events. Another one, `Conn::Info`, which corresponds to the fields logged into `conn.log`, is shown by the excerpt below.

Looking at the structure of the definition, a new collection of data types is being defined as a type called `Info`. Since this type definition is within the confines of an export block, what is defined is, in fact, `Conn::Info`.

The formatting for a declaration of a record type in Bro includes the descriptive name of the type being defined and the separate fields that make up the record. The individual fields that make up the new record are not limited in type or number as long as the name for each field is unique.

```
data_struct_record_01.bro

type Service: record {
    name: string;
    ports: set[port];
    rfc: count;
};

function print_service(serv: Service): string
    {
    print fmt("Service: %s(RFC%d)",serv$name, serv$rfc);

    for ( p in serv$ports )
        print fmt("  port: %s", p);
    }

event bro_init()
    {
    local dns: Service = [$name="dns", $ports=set(53/udp, 53/tcp), $rfc=1035];
    local http: Service = [$name="http", $ports=set(80/tcp, 8080/tcp), $rfc=2616];

    print_service(dns);
    print_service(http);
    }
```

```
# bro data_struct_record_01.bro
Service: dns(RFC1035)
  port: 53/tcp
  port: 53/udp
Service: http(RFC2616)
  port: 80/tcp
  port: 8080/tcp
```

The sample above shows a simple type definition that includes a string, a set of ports, and a count to define a service type. Also included is a function to print each field of a record in a formatted fashion and a `bro_init` event handler to show some functionality of working with records. The definitions of the DNS and HTTP services are both done in-line using squared brackets before being passed to the `print_service` function. The `print_service` function makes use of the `$` dereference operator to access the fields within the newly defined Service record type.

As you saw in the definition for the `Conn::Info` record, other records are even valid as fields within another record. We can extend the example above to include another record that contains a Service record.

```
data_struct_record_02.bro

type Service: record {
    name: string;
    ports: set[port];
    rfc: count;
    };

type System: record {
    name: string;
    services: set[Service];
    };

function print_service(serv: Service): string
    {
    print fmt("  Service: %s(RFC%d)",serv$name, serv$rfc);

    for ( p in serv$ports )
        print fmt("    port: %s", p);
    }

function print_system(sys: System): string
    {
    print fmt("System: %s", sys$name);

    for ( s in sys$services )
        print_service(s);
    }

event bro_init()
    {
    local server01: System;
    server01$name = "morlock";
    add server01$services[[ $name="dns", $ports=set(53/udp, 53/tcp), $rfc=1035]];
    add server01$services[[ $name="http", $ports=set(80/tcp, 8080/tcp), $rfc=2616]];
    print_system(server01);


    # local dns: Service = [ $name="dns", $ports=set(53/udp, 53/tcp), $rfc=1035];
    # local http: Service = [ $name="http", $ports=set(80/tcp, 8080/tcp), $rfc=2616];
    # print_service(dns);
    # print_service(http);
    }
```

```
# bro data_struct_record_02.bro
System: morlock
  Service: dns(RFC1035)
    port: 53/tcp
    port: 53/udp
  Service: http(RFC2616)
    port: 80/tcp
    port: 8080/tcp
```

The example above includes a second record type in which a field is used as the data type for a set. Records can be repeatedly nested within other records, their fields reachable through repeated chains of the $ dereference operator.

It's also common to see a `type` used to simply alias a data structure to a more descriptive name. The example below shows an example of this from Bro's own type definitions file.

```
init-bare.bro

type string_array: table[count] of string;
type string_set: set[string];
type addr_set: set[addr];
```

The three lines above alias a type of data structure to a descriptive name. Functionally, the operations are the same, however, each of the types above are named such that their function is instantly identifiable. This is another place in Bro scripting where consideration can lead to better readability of your code and thus easier maintainability in the future.

### 2.5.5 Custom Logging

Armed with a decent understanding of the data types and data structures in Bro, exploring the various frameworks available is a much more rewarding effort. The framework with which most users are likely to have the most interaction is the Logging Framework. Designed in such a way to so as to abstract much of the process of creating a file and appending ordered and organized data into it, the Logging Framework makes use of some potentially unfamiliar nomenclature. Specifically, Log Streams, Filters and Writers are simply abstractions of the processes required to manage a high rate of incoming logs while maintaining full operability. If you've seen Bro employed in an environment with a large number of connections, you know that logs are produced incredibly quickly; the ability to process a large set of data and write it to disk is due to the design of the Logging Framework.

Data is written to a Log Stream based on decision making processes in Bro's scriptland. Log Streams correspond to a single log as defined by the set of name/value pairs that make up its fields. That data can then be filtered, modified, or redirected with Logging Filters which, by default, are set to log everything. Filters can be used to break log files into subsets or duplicate that information to another output. The final output of the data is defined by the writer. Bro's default writer is simple tab separated ASCII files but Bro also includes support for DataSeries and Elasticsearch outputs as well as additional writers currently in development. While these new terms and ideas may give the impression that the Logging Framework is difficult to work with, the actual learning curve is, in actuality, not very steep at all. The abstraction built into the Logging Framework makes it such that a vast majority of scripts needs not go past the basics. In effect, writing to a log file is as simple as defining the format of your data, letting Bro know that you wish to create a new log, and then calling the `Log::write` method to output log records.

The Logging Framework is an area in Bro where, the more you see it used and the more you use it yourself, the more second nature the boilerplate parts of the code will become. As such, let's work through a contrived example of simply logging the digits 1 through 10 and their corresponding factorial to the default ASCII log writer. It's always best to work through the problem once, simulating the desired output with `print` and `fmt` before attempting to dive into the Logging Framework.

```
framework_logging_factorial_01.bro

module Factor;

function factorial(n: count): count
    {
    if ( n == 0 )
        return 1;
    else
        return ( n * factorial(n - 1) );
    }

event bro_init()
    {
    local numbers: vector of count = vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    for ( n in numbers )
        print fmt("%d", factorial(numbers[n]));
    }
```

```
# bro framework_logging_factorial_01.bro
1
2
6
24
120
720
5040
40320
362880
3628800
```

This script defines a factorial function to recursively calculate the factorial of a unsigned integer passed as an argument to the function. Using `print` and `fmt` we can ensure that Bro can perform these calculations correctly as well get an idea of the answers ourselves.

The output of the script aligns with what we expect so now it's time to integrate the Logging Framework.

```
framework_logging_factorial_02.bro

module Factor;

export {
    # Append the value LOG to the Log::ID enumerable.
    redef enum Log::ID += { LOG };

    # Define a new type called Factor::Info.
    type Info: record {
        num:           count &log;
        factorial_num: count &log;
        };
    }
```

```

function factorial(n: count): count
    {
    if ( n == 0 )
        return 1;

    else
        return ( n * factorial(n - 1) );
    }

event bro_init()
    {
    # Create the logging stream.
    Log::create_stream(LOG, [$columns=Info, $path="factor"]);
    }

event bro_done()
    {
    local numbers: vector of count = vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    for ( n in numbers )
        Log::write( Factor::LOG, [$num=numbers[n],
                                   $factorial_num=factorial(numbers[n])]);
    }
```

As mentioned above we have to perform a few steps before we can issue the `Log::write` method and produce a logfile. As we are working within a namespace and informing an outside entity of workings and data internal to the namespace, we use an `export` block. First we need to inform Bro that we are going to be adding another Log Stream by adding a value to the `Log::ID` enumerable. In this script, we append the value `LOG` to the `Log::ID` enumerable, however due to this being in an export block the value appended to `Log::ID` is actually `Factor::Log`. Next, we need to define the name and value pairs that make up the data of our logs and dictate its format. This script defines a new record datatype called `Info` (actually, `Factor::Info`) with two fields, both unsigned integers. Each of the fields in the `Factor::Log` record type include the `&log` attribute, indicating that these fields should be passed to the Logging Framework when `Log::write` is called. Were there to be any name value pairs without the `&log` attribute, those fields would simply be ignored during logging but remain available for the lifespan of the variable. The next step is to create the logging stream with `Log::create_stream` which takes a `Log::ID` and a record as its arguments. In this example, we call the `Log::create_stream` method and pass `Factor::LOG` and the `Factor::Info` record as arguments. From here on out, if we issue the `Log::write` command with the correct `Log::ID` and a properly formatted `Factor::Info` record, a log entry will be generated.

Now, if we run this script, instead of generating logging information to stdout, no output is created. Instead the output is all in `factor.log`, properly formatted and organized.

```
# bro framework_logging_factorial_02.bro
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     factor
#open     2016-10-01-03-42-31
#fields   num     factorial_num
#types    count   count
1  1
2  2
3  6
4  24
5  120
```

```
6 720
7 5040
8 40320
9 362880
10          3628800
#close      2016-10-01-03-42-31
```

While the previous example is a simplistic one, it serves to demonstrate the small pieces of script code hat need to be in place in order to generate logs. For example, it's common to call `Log::create_stream` in `bro_init` and while in a live example, determining when to call `Log::write` would likely be done in an event handler, in this case we use `bro_done` .

If you've already spent time with a deployment of Bro, you've likely had the opportunity to view, search through, or manipulate the logs produced by the Logging Framework. The log output from a default installation of Bro is substantial to say the least, however, there are times in which the way the Logging Framework by default isn't ideal for the situation. This can range from needing to log more or less data with each call to `Log::write` or even the need to split log files based on arbitrary logic. In the later case, Filters come into play along with the Logging Framework. Filters grant a level of customization to Bro's scriptland, allowing the script writer to include or exclude fields in the log and even make alterations to the path of the file in which the logs are being placed. Each stream, when created, is given a default filter called, not surprisingly, `default`. When using the `default` filter, every key value pair with the `&log` attribute is written to a single file. For the example we've been using, let's extend it so as to write any factorial which is a factor of 5 to an alternate file, while writing the remaining logs to factor.log.

```
framework_logging_factorial_03.bro

event bro_init()
    {
    Log::create_stream(LOG, [$columns=Info, $path="factor"]);

    local filter: Log::Filter = [$name="split-mod5s", $path_func=mod5];
    Log::add_filter(Factor::LOG, filter);
    Log::remove_filter(Factor::LOG, "default");
    }
```

To dynamically alter the file in which a stream writes its logs, a filter can specify a function that returns a string to be used as the filename for the current call to `Log::write`. The definition for this function has to take as its parameters a `Log::ID` called id, a string called `path` and the appropriate record type for the logs called `rec`. You can see the definition of `mod5` used in this example conforms to that requirement. The function simply returns `factor-mod5` if the factorial is divisible evenly by 5, otherwise, it returns `factor-non5`. In the additional `bro_init` event handler, we define a locally scoped `Log::Filter` and assign it a record that defines the `name` and `path_func` fields. We then call `Log::add_filter` to add the filter to the `Factor::LOG` `Log::ID` and call `Log::remove_filter` to remove the `default` filter for `Factor::LOG`. Had we not removed the `default` filter, we'd have ended up with three log files: `factor-mod5.log` with all the factorials that are a factors of 5, `factor-non5.log` with the factorials that are not factors of 5, and `factor.log` which would have included all factorials.

```
# bro framework_logging_factorial_03.bro
```

```
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     factor-mod5
#open     2016-10-01-03-42-32
#fields   num     factorial_num
#types    count   count
5 120
```

```
6 720
7 5040
8 40320
9 362880
10	3628800
#close	2016-10-01-03-42-32
```

The ability of Bro to generate easily customizable and extensible logs which remain easily parsable is a big part of the reason Bro has gained a large measure of respect. In fact, it's difficult at times to think of something that Bro doesn't log and as such, it is often advantageous for analysts and systems architects to instead hook into the logging framework to be able to perform custom actions based upon the data being sent to the Logging Frame. To that end, every default log stream in Bro generates a custom event that can be handled by anyone wishing to act upon the data being sent to the stream. By convention these events are usually in the format `log_x` where x is the name of the logging stream; as such the event raised for every log sent to the Logging Framework by the HTTP parser would be `log_http`. In fact, we've already seen a script handle the `log_http` event when we broke down how the `detect-MHR.bro` script worked. In that example, as each log entry was sent to the logging framework, post-processing was taking place in the `log_http` event. Instead of using an external script to parse the `http.log` file and do post-processing for the entry, post-processing can be done in real time in Bro.

Telling Bro to raise an event in your own Logging stream is as simple as exporting that event name and then adding that event in the call to `Log::create_stream`. Going back to our simple example of logging the factorial of an integer, we add `log_factor` to the `export` block and define the value to be passed to it, in this case the `Factor::Info` record. We then list the `log_factor` function as the `$ev` field in the call to `Log::create_stream`

```
framework_logging_factorial_04.bro

module Factor;

export {
    redef enum Log::ID += { LOG };

    type Info: record {
        num:           count &log;
        factorial_num: count &log;
        };

    global log_factor: event(rec: Info);
    }

function factorial(n: count): count
    {
    if ( n == 0 )
        return 1;

    else
        return (n * factorial(n - 1));
    }

event bro_init()
    {
    Log::create_stream(LOG, [$columns=Info, $ev=log_factor, $path="factor"]);
    }

event bro_done()
    {
    local numbers: vector of count = vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    for ( n in numbers )
```

```
        Log::write( Factor::LOG, [$num=numbers[n],
                                  $factorial_num=factorial(numbers[n])]);
    }

function mod5(id: Log::ID, path: string, rec: Factor::Info) : string
    {
    if ( rec$factorial_num % 5 == 0 )
        return "factor-mod5";

    else
        return "factor-non5";
    }

event bro_init()
    {
    local filter: Log::Filter = [$name="split-mod5s", $path_func=mod5];
    Log::add_filter(Factor::LOG, filter);
    Log::remove_filter(Factor::LOG, "default");
    }
```

### 2.5.6 Raising Notices

While Bro's Logging Framework provides an easy and systematic way to generate logs, there still exists a need to indicate when a specific behavior has been detected and a method to allow that detection to come to someone's attention. To that end, the Notice Framework is in place to allow script writers a codified means through which they can raise a notice, as well as a system through which an operator can opt-in to receive the notice. Bro holds to the philosophy that it is up to the individual operator to indicate the behaviors in which they are interested and as such Bro ships with a large number of policy scripts which detect behavior that may be of interest but it does not presume to guess as to which behaviors are "action-able". In effect, Bro works to separate the act of detection and the responsibility of reporting. With the Notice Framework it's simple to raise a notice for any behavior that is detected.

To raise a notice in Bro, you only need to indicate to Bro that you are provide a specific `Notice::Type` by exporting it and then make a call to `NOTICE` supplying it with an appropriate `Notice::Info` record. Often times the call to `NOTICE` includes just the `Notice::Type`, and a concise message. There are however, significantly more options available when raising notices as seen in the definition of `Notice::Info`. The only field in `Notice::Info` whose attributes make it a required field is the `note` field. Still, good manners are always important and including a concise message in `$msg` and, where necessary, the contents of the connection record in `$conn` along with the `Notice::Type` tend to comprise the minimum of information required for an notice to be considered useful. If the `$conn` variable is supplied the Notice Framework will auto-populate the `$id` and `$src` fields as well. Other fields that are commonly included, `$identifier` and `$suppress_for` are built around the automated suppression feature of the Notice Framework which we will cover shortly.

One of the default policy scripts raises a notice when an SSH login has been heuristically detected and the originating hostname is one that would raise suspicion. Effectively, the script attempts to define a list of hosts from which you would never want to see SSH traffic originating, like DNS servers, mail servers, etc. To accomplish this, the script adheres to the separation of detection and reporting by detecting a behavior and raising a notice. Whether or not that notice is acted upon is decided by the local Notice Policy, but the script attempts to supply as much information as possible while staying concise.

```
interesting-hostnames.bro

##! This script will generate a notice if an apparent SSH login originates
##! or heads to a host with a reverse hostname that looks suspicious.  By
##! default, the regular expression to match "interesting" hostnames includes
##! names that are typically used for infrastructure hosts like nameservers,
##! mail servers, web servers and ftp servers.

@load base/frameworks/notice

module SSH;

export {
        redef enum Notice::Type += {
                ## Generated if a login originates or responds with a host where
                ## the reverse hostname lookup resolves to a name matched by the
                ## :bro:id:`SSH::interesting_hostnames` regular expression.
                Interesting_Hostname_Login,
        };

        ## Strange/bad host names to see successful SSH logins from or to.
        const interesting_hostnames =
                        /^d?ns[0-9]*\./ |
                        /^smtp[0-9]*\./ |
                        /^mail[0-9]*\./ |
                        /^pop[0-9]*\./  |
                        /^imap[0-9]*\./ |
                        /^www[0-9]*\./  |
                        /^ftp[0-9]*\./  &redef;
}

event ssh_auth_successful(c: connection, auth_method_none: bool)
        {
        for ( host in set(c$id$orig_h, c$id$resp_h) )
                {
                when ( local hostname = lookup_addr(host) )
                        {
                        if ( interesting_hostnames in hostname )
                                {
                                NOTICE([$note=Interesting_Hostname_Login,
                                        $msg=fmt("Possible SSH login involving a %s
↪%s with an interesting hostname.",
                                                 Site::is_local_addr(host) ? "local" :
↪ "remote",
                                                 host == c$id$orig_h ? "client" :
↪"server"),
                                        $sub=hostname, $conn=c]);
                                }
                        }
                }
        }
```

While much of the script relates to the actual detection, the parts specific to the Notice Framework are actually quite interesting in themselves. The script's `export` block adds the value `SSH::Interesting_Hostname_Login` to the enumerable constant `Notice::Type` to indicate to the Bro core that a new type of notice is being defined. The script then calls `NOTICE` and defines the `$note`, `$msg`, `$sub` and `$conn` fields of the `Notice::Info` record. There are two ternary if statements that modify the `$msg` text depending on whether the host is a local address and whether it is the client or the server. This use of `fmt` and ternary operators is a concise way to lend readability to the notices that are generated without the need for branching `if` statements that each raise a specific notice.

The opt-in system for notices is managed through writing `Notice::policy` hooks. A `Notice::policy` hook takes as its argument a `Notice::Info` record which will hold the same information your script provided in its call to `NOTICE`. With access to the `Notice::Info` record for a specific notice you can include logic such as in statements in the body of your hook to alter the policy for handling notices on your system. In Bro, hooks are akin to a mix of functions and event handlers: like functions, calls to them are synchronous (i.e., run to completion and return); but like events, they can have multiple bodies which will all execute. For defining a notice policy, you define a hook and Bro will take care of passing in the `Notice::Info` record. The simplest kind of `Notice::policy` hooks simply check the value of `$note` in the `Notice::Info` record being passed into the hook and performing an action based on the answer. The hook below adds the `Notice::ACTION_EMAIL` action for the `SSH::Interesting_Hostname_Login` notice raised in the /scripts/policy/protocols/ssh/interesting-hostnames.bro script.

```
framework_notice_hook_01.bro

@load policy/protocols/ssh/interesting-hostnames.bro

hook Notice::policy(n: Notice::Info)
  {
  if ( n$note == SSH::Interesting_Hostname_Login )
      add n$actions[Notice::ACTION_EMAIL];
  }
```

In the example above we've added `Notice::ACTION_EMAIL` to the `n$actions` set. This set, defined in the Notice Framework scripts, can only have entries from the `Notice::Action` type, which is itself an enumerable that defines the values shown in the table below along with their corresponding meanings. The `Notice::ACTION_LOG` action writes the notice to the `Notice::LOG` logging stream which, in the default configuration, will write each notice to the `notice.log` file and take no further action. The `Notice::ACTION_EMAIL` action will send an email to the address or addresses defined in the `Notice::mail_dest` variable with the particulars of the notice as the body of the email. The last action, `Notice::ACTION_ALARM` sends the notice to the `Notice::ALARM_LOG` logging stream which is then rotated hourly and its contents emailed in readable ASCII to the addresses in `Notice::mail_dest`.

| | |
|---|---|
| ACTION_NONE | Take no action |
| ACTION_LOG | Send the notice to the Notice::LOG logging stream. |
| ACTION_EMAIL | Send an email with the notice in the body. |
| ACTION_ALARM | Send the notice to the Notice::Alarm_LOG stream. |

While actions like the `Notice::ACTION_EMAIL` action have appeal for quick alerts and response, a caveat of its use is to make sure the notices configured with this action also have a suppression. A suppression is a means through which notices can be ignored after they are initially raised if the author of the script has set an identifier. An identifier is a unique string of information collected from the connection relative to the behavior that has been observed by Bro.

```
expiring-certs.bro

                NOTICE([$note=Certificate_Expires_Soon,
                        $msg=fmt("Certificate %s is going to expire at %T", cert
↪$subject, cert$not_valid_after),
                        $conn=c, $suppress_for=1day,
                        $identifier=cat(c$id$resp_h, c$id$resp_p, hash),
                        $fuid=fuid]);
```

In the /scripts/policy/protocols/ssl/expiring-certs.bro script which identifies when SSL certificates are set to expire and raises notices when it crosses a predefined threshold, the call to `NOTICE` above also sets the `$identifier` entry by concatenating the responder IP, port, and the hash of the certificate. The selection of responder IP, port and certificate hash fits perfectly into an appropriate identifier as it creates a unique identifier with which the suppression can be matched. Were we to take out any of the entities used for the identifier, for example the certificate hash, we could be setting our suppression too broadly, causing an analyst to miss a notice that should have been raised. Depending on the available data for the identifier, it can be useful to set the `$suppress_for` variable as well. The `expiring-certs.bro` script sets `$suppress_for` to `1day`, telling the Notice Framework to suppress the notice for 24 hours after the first notice is raised. Once that time limit has passed, another notice can be raised which will again set the `1day` suppression time. Suppressing for a specific amount of time has benefits beyond simply not filling up an analyst's email inbox; keeping the notice alerts timely and succinct helps avoid a case where an analyst might see the notice and, due to over exposure, ignore it.

The `$suppress_for` variable can also be altered in a `Notice::policy` hook, allowing a deployment to better suit the environment in which it is be run. Using the example of `expiring-certs.bro`, we can write a `Notice::policy` hook for `SSL::Certificate_Expires_Soon` to configure the `$suppress_for` variable to a shorter time.

```
framework_notice_hook_suppression_01.bro

@load policy/protocols/ssl/expiring-certs.bro

hook Notice::policy(n: Notice::Info)
   {
   if ( n$note == SSL::Certificate_Expires_Soon )
       n$suppress_for = 12hrs;
   }
```

While `Notice::policy` hooks allow you to build custom predicate-based policies for a deployment, there are bound to be times where you don't require the full expressiveness that a hook allows. In short, there will be notice policy considerations where a broad decision can be made based on the `Notice::Type` alone. To facilitate these types of decisions, the Notice Framework supports Notice Policy shortcuts. These shortcuts are implemented through the means of a group of data structures that map specific, predefined details and actions to the effective name of a notice. Primarily implemented as a set or table of enumerables of `Notice::Type`, Notice Policy shortcuts can be placed as a single directive in your `local.bro` file as a concise readable configuration. As these variables are all constants, it bears mentioning that these variables are all set at parse-time before Bro is fully up and running and not set dynamically.

| Name | Description | Data Type |
|---|---|---|
| Notice::ignored_types | Ignore the Notice::Type entirely | set[Notice::Type] |
| Notice::emailed_types | Set Notice::ACTION_EMAIL to this Notice::Type | set[Notice::Type] |
| Notice::alarmed_types | Set Notice::ACTION_ALARM to this Notice::Type | set[Notice::Type] |
| Notice::not_suppressed_types | Remove suppression from this Notice::Type | set[Notice::Type] |
| No-tice::type_suppression_intervals | Alter the $suppress_for value for this Notice::Type | table[Notice::Type] of interval |

The table above details the five Notice Policy shortcuts, their meaning and the data type used to implement them. With the exception of `Notice::type_suppression_intervals` a `set` data type is employed to hold the `Notice::Type` of the notice upon which a shortcut should applied. The first three shortcuts are fairly self explanatory, applying an action to the `Notice::Type` elements in the set, while the latter two shortcuts alter details of the suppression being applied to the Notice. The shortcut `Notice::not_suppressed_types` can be used to remove the configured suppression from a notice while `Notice::type_suppression_intervals` can be used to alter the suppression interval defined by $suppress_for in the call to `NOTICE`.

```
framework_notice_shortcuts_01.bro

@load policy/protocols/ssh/interesting-hostnames.bro
@load base/protocols/ssh/

redef Notice::emailed_types += {
    SSH::Interesting_Hostname_Login
};
```

The Notice Policy shortcut above adds the `Notice::Type` of `SSH::Interesting_Hostname_Login` to the `Notice::emailed_types` set while the shortcut below alters the length of time for which those notices will be suppressed.

```
framework_notice_shortcuts_02.bro

@load policy/protocols/ssh/interesting-hostnames.bro
@load base/protocols/ssh/

redef Notice::type_suppression_intervals += {
    [SSH::Interesting_Hostname_Login] = 1day,
};
```

CHAPTER

THREE

# REFERENCE SECTION

## 3.1 Frameworks

### 3.1.1 File Analysis

In the past, writing Bro scripts with the intent of analyzing file content could be cumbersome because of the fact that the content would be presented in different ways, via events, at the script-layer depending on which network protocol was involved in the file transfer. Scripts written to analyze files over one protocol would have to be copied and modified to fit other protocols. The file analysis framework (FAF) instead provides a generalized presentation of file-related information. The information regarding the protocol involved in transporting a file over the network is still available, but it no longer has to dictate how one organizes their scripting logic to handle it. A goal of the FAF is to provide analysis specifically for files that is analogous to the analysis Bro provides for network connections.

**Contents**

#### File Lifecycle Events

The key events that may occur during the lifetime of a file are: `file_new`, `file_over_new_connection`, `file_timeout`, `file_gap`, and `file_state_remove`. Handling any of these events provides some information about the file such as which network `connection` and protocol are transporting the file, how many bytes have been transferred so far, and its MIME type.

Here's a simple example:

```
file_analysis_01.bro

event connection_state_remove(c: connection)
    {
    print "connection_state_remove";
    print c$uid;
    print c$id;
    for ( s in c$service )
        print s;
    }

event file_state_remove(f: fa_file)
    {
    print "file_state_remove";
    print f$id;
    for ( cid in f$conns )
        {
        print f$conns[cid]$uid;
        print cid;
        }
    print f$source;
    }
```

```
# bro -r http/get.trace file_analysis_01.bro
file_state_remove
FakNcS1Jfe01uljb3
CXWv6p3arKYeMETxOg
[orig_h=141.142.228.5, orig_p=59856/tcp, resp_h=192.150.187.43, resp_p=80/tcp]
HTTP
connection_state_remove
CXWv6p3arKYeMETxOg
[orig_h=141.142.228.5, orig_p=59856/tcp, resp_h=192.150.187.43, resp_p=80/tcp]
HTTP
```

This doesn't perform any interesting analysis yet, but does highlight the similarity between analysis of connections and files. Connections are identified by the usual 5-tuple or a convenient UID string while files are identified just by a string of the same format as the connection UID. So there's unique ways to identify both files and connections and files hold references to a connection (or connections) that transported it.

### Adding Analysis

There are builtin file analyzers which can be attached to files. Once attached, they start receiving the contents of the file as Bro extracts it from an ongoing network connection. What they do with the file contents is up to the particular file analyzer implementation, but they'll typically either report further information about the file via events (e.g. *`Files::ANALYZER_MD5`* will report the file's MD5 checksum via *`file_hash`* once calculated) or they'll have some side effect (e.g. *`Files::ANALYZER_EXTRACT`* will write the contents of the file out to the local file system).

In the future there may be file analyzers that automatically attach to files based on heuristics, similar to the Dynamic Protocol Detection (DPD) framework for connections, but many will always require an explicit attachment decision.

Here's a simple example of how to use the MD5 file analyzer to calculate the MD5 of plain text files:

```
file_analysis_02.bro

event file_sniff(f: fa_file, meta: fa_metadata)
    {
        if ( ! meta?$mime_type ) return;
    print "new file", f$id;
    if ( meta$mime_type == "text/plain" )
        Files::add_analyzer(f, Files::ANALYZER_MD5);
    }

event file_hash(f: fa_file, kind: string, hash: string)
    {
```

```
    print "file_hash", f$id, kind, hash;
    }
```

```
# bro -r http/get.trace file_analysis_02.bro
new file, FakNcS1Jfe01uljb3
file_hash, FakNcS1Jfe01uljb3, md5, 397168fd09991a0e712254df7bc639ac
```

Some file analyzers might have tunable parameters that need to be specified in the call to `Files::add_analyzer`:

In this case, the file extraction analyzer doesn't generate any further events, but does have the effect of writing out the file contents to the local file system at the location resulting from the concatenation of the path specified by `FileExtract::prefix` and the string, `myfile`. Of course, for a network with more than a single file being transferred, it's probably preferable to specify a different extraction path for each file, unlike this example.

Regardless of which file analyzers end up acting on a file, general information about the file (e.g. size, time of last data transferred, MIME type, etc.) are logged in `files.log`.

### Input Framework Integration

The FAF comes with a simple way to integrate with the *Input Framework*, so that Bro can analyze files from external sources in the same way it analyzes files that it sees coming over traffic from a network interface it's monitoring. It only requires a call to `Input::add_analysis`:

```
file_analysis_03.bro

redef exit_only_after_terminate = T;

event file_new(f: fa_file)
    {
    print "new file", f$id;
    Files::add_analyzer(f, Files::ANALYZER_MD5);
    }

event file_state_remove(f: fa_file)
    {
    print "file_state_remove";
    Input::remove(f$source);
    terminate();
    }

event file_hash(f: fa_file, kind: string, hash: string)
    {
    print "file_hash", f$id, kind, hash;
    }

event bro_init()
    {
    local source: string = "./myfile";
    Input::add_analysis([$source=source, $name=source]);
    }
```

Note that the "source" field of `fa_file` corresponds to the "name" field of `Input::AnalysisDescription` since that is what the input framework uses to uniquely identify an input stream.

The output of the above script may be (assuming a file called "myfile" exists):

```
# bro file_analysis_03.bro
new file, FZedLu4Ajcvge02jA8
file_hash, FZedLu4Ajcvge02jA8, md5, f0ef7081e1539ac00ef5b761b4fb01b3
file_state_remove
```

Nothing that special, but it at least verifies the MD5 file analyzer saw all the bytes of the input file and calculated the checksum correctly!

## 3.1.2 GeoLocation

During the process of creating policy scripts the need may arise to find the geographic location for an IP address. Bro has support for the GeoIP library at the policy script level beginning with release 1.3 to account for this need. To use this functionality, you need to first install the libGeoIP software, and then install the GeoLite city database before building Bro.

**Contents**

### Install libGeoIP

Before building Bro, you need to install libGeoIP.

- FreeBSD:
- RPM/RedHat-based Linux:
- DEB/Debian-based Linux:
- Mac OS X:

  You need to install from your preferred package management system (e.g. MacPorts, Fink, or Homebrew). The name of the package that you need may be libgeoip, geoip, or geoip-dev, depending on which package management system you are using.

### GeoIPLite Database Installation

A country database for GeoIPLite is included when you do the C API install, but for Bro, we are using the city database which includes cities and regions in addition to countries.

Download the GeoLite city binary database:

Next, the file needs to be renamed and put in the GeoIP database directory. This directory should already exist and will vary depending on which platform and package you are using. For FreeBSD, use `/usr/local/share/GeoIP`. For Linux, use `/usr/share/GeoIP` or `/var/lib/GeoIP` (choose whichever one already exists).

Note that there is a separate database for IPv6 addresses, which can also be installed if you want GeoIP functionality for IPv6.

### Testing

Before using the GeoIP functionality, it is a good idea to verify that everything is setup correctly. After installing libGeoIP and the GeoIP city database, and building Bro, you can quickly check if the GeoIP functionality works by running a command like this:

If you see an error message similar to "Failed to open GeoIP City database", then you may need to either rename or move your GeoIP city database file (the error message should give you the full pathname of the database file that Bro is looking for).

If you see an error message similar to "Bro was not configured for GeoIP support", then you need to rebuild Bro and make sure it is linked against libGeoIP. Normally, if libGeoIP is installed correctly then it should automatically be found when building Bro. If this doesn't happen, then you may need to specify the path to the libGeoIP installation (e.g. `./configure --with-geoip=<path>`).

### Usage

There is a built-in function that provides the GeoIP functionality:

The return value of the `lookup_location` function is a record type called `geo_location`, and it consists of several fields containing the country, region, city, latitude, and longitude of the specified IP address. Since one or more fields in this record will be uninitialized for some IP addresses (for example, the country and region of an IP address might be known, but the city could be unknown), a field should be checked if it has a value before trying to access the value.

### Example

To show every ftp connection from hosts in Ohio, this is now very easy:

## 3.1.3 Input Framework

Bro features a flexible input framework that allows users to import data into Bro. Data is either read into Bro tables or converted to events which can then be handled by scripts. This document gives an overview of how to use the input framework with some examples. For more complex scenarios it is worthwhile to take a look at the unit tests in `testing/btest/scripts/base/frameworks/input/`.

**Contents**

### Reading Data into Tables

Probably the most interesting use-case of the input framework is to read data into a Bro table.

By default, the input framework reads the data in the same format as it is written by the logging framework in Bro - a tab-separated ASCII file.

We will show the ways to read files into Bro with a simple example. For this example we assume that we want to import data from a blacklist that contains server IP addresses as well as the timestamp and the reason for the block.

An example input file could look like this (note that all fields must be tab-separated):

```
#fields ip timestamp reason
192.168.17.1 1333252748 Malware host
192.168.27.2 1330235733 Botnet server
192.168.250.3 1333145108 Virus detected
```

To read a file into a Bro table, two record types have to be defined. One contains the types and names of the columns that should constitute the table keys and the second contains the types and names of the columns that should constitute the table values.

In our case, we want to be able to lookup IPs. Hence, our key record only contains the server IP. All other elements should be stored as the table content.

The two records are defined as:

Note that the names of the fields in the record definitions must correspond to the column names listed in the '#fields' line of the log file, in this case 'ip', 'timestamp', and 'reason'. Also note that the ordering of the columns does not matter, because each column is identified by name.

The log file is read into the table with a simple call of the `Input::add_table` function:

With these three lines we first create an empty table that should contain the blacklist data and then instruct the input framework to open an input stream named `blacklist` to read the data into the table. The third line removes the input stream again, because we do not need it any more after the data has been read.

Because some data files can - potentially - be rather big, the input framework works asynchronously. A new thread is created for each new input stream. This thread opens the input data file, converts the data into a Bro format and sends it back to the main Bro thread.

Because of this, the data is not immediately accessible. Depending on the size of the data source it might take from a few milliseconds up to a few seconds until all data is present in the table. Please note that this means that when Bro is running without an input source or on very short captured files, it might terminate before the data is present in the table (because Bro already handled all packets before the import thread finished).

Subsequent calls to an input source are queued until the previous action has been completed. Because of this, it is, for example, possible to call `add_table` and `remove` in two subsequent lines: the `remove` action will remain queued until the first read has been completed.

Once the input framework finishes reading from a data source, it fires the `Input::end_of_data` event. Once this event has been received all data from the input file is available in the table.

The table can be used while the data is still being read - it just might not contain all lines from the input file before the event has fired. After the table has been populated it can be used like any other Bro table and blacklist entries can easily be tested:

### Re-reading and streaming data

For many data sources, like for many blacklists, the source data is continually changing. For these cases, the Bro input framework supports several ways to deal with changing data files.

The first, very basic method is an explicit refresh of an input stream. When an input stream is open (this means it has not yet been removed by a call to `Input::remove`), the function `Input::force_update` can be called. This will trigger a complete refresh of the table; any changed elements from the file will be updated. After the update is finished the `Input::end_of_data` event will be raised.

In our example the call would look like:

Alternatively, the input framework can automatically refresh the table contents when it detects a change to the input file. To use this feature, you need to specify a non-default read mode by setting the `mode` option of the `Input::add_table` call. Valid values are `Input::MANUAL` (the default), `Input::REREAD` and `Input::STREAM`. For example, setting the value of the `mode` option in the previous example would look like this:

When using the reread mode (i.e., `$mode=Input::REREAD`), Bro continually checks if the input file has been changed. If the file has been changed, it is re-read and the data in the Bro table is updated to reflect the current state. Each time a change has been detected and all the new data has been read into the table, the `end_of_data` event is raised.

When using the streaming mode (i.e., `$mode=Input::STREAM`), Bro assumes that the source data file is an append-only file to which new data is continually appended. Bro continually checks for new data at the end of the file and will add the new data to the table. If newer lines in the file have the same index as previous lines, they will overwrite the values in the output table. Because of the nature of streaming reads (data is continually added to the table), the `end_of_data` event is never raised when using streaming reads.

### Receiving change events

When re-reading files, it might be interesting to know exactly which lines in the source files have changed.

For this reason, the input framework can raise an event each time when a data item is added to, removed from, or changed in a table.

The event definition looks like this (note that you can change the name of this event in your own Bro script):

The event must be specified in `$ev` in the `add_table` call:

The `description` argument of the event contains the arguments that were originally supplied to the add_table call. Hence, the name of the stream can, for example, be accessed with `description$name`. The `tpe` argument of the event is an enum containing the type of the change that occurred.

If a line that was not previously present in the table has been added, then the value of `tpe` will be `Input::EVENT_NEW`. In this case `left` contains the index of the added table entry and `right` contains the values of the added entry.

If a table entry that already was present is altered during the re-reading or streaming read of a file, then the value of `tpe` will be `Input::EVENT_CHANGED`. In this case `left` contains the index of the changed table entry and `right` contains the values of the entry before the change. The reason for this is that the table already has been updated when the event is raised. The current value in the table can be ascertained by looking up the current table value. Hence it is possible to compare the new and the old values of the table.

If a table element is removed because it was no longer present during a re-read, then the value of `tpe` will be `Input::EVENT_REMOVED`. In this case `left` contains the index and `right` the values of the removed element.

### Filtering data during import

The input framework also allows a user to filter the data during the import. To this end, predicate functions are used. A predicate function is called before a new element is added/changed/removed from a table. The predicate can either accept or veto the change by returning true for an accepted change and false for a rejected change. Furthermore, it can alter the data before it is written to the table.

The following example filter will reject adding entries to the table when they were generated over a month ago. It will accept all changes and all removals of values that are already present in the table.

To change elements while they are being imported, the predicate function can manipulate `left` and `right`. Note that predicate functions are called before the change is committed to the table. Hence, when a table element is changed (`typ` is `Input::EVENT_CHANGED`), `left` and `right` contain the new values, but the destination (`blacklist` in our example) still contains the old values. This allows predicate functions to examine the changes between the old and the new version before deciding if they should be allowed.

### Different readers

The input framework supports different kinds of readers for different kinds of source data files. At the moment, the default reader reads ASCII files formatted in the Bro log file format (tab-separated values with a "#fields" header line). Several other readers are included in Bro.

The raw reader reads a file that is split by a specified record separator (newline by default). The contents are returned line-by-line as strings; it can, for example, be used to read configuration files and the like and is probably only useful in the event mode and not for reading data to tables.

The binary reader is intended to be used with file analysis input streams (and is the default type of reader for those streams).

The benchmark reader is being used to optimize the speed of the input framework. It can generate arbitrary amounts of semi-random data in all Bro data types supported by the input framework.

Currently, Bro supports the following readers in addition to the aforementioned ones:

### Logging To and Reading From SQLite Databases

Starting with version 2.2, Bro features a SQLite logging writer as well as a SQLite input reader. SQLite is a simple, file-based, widely used SQL database system. Using SQLite allows Bro to write and access data in a format that is easy to use in interchange with other applications. Due to the transactional nature of SQLite, databases can be used by several applications simultaneously. Hence, they can, for example, be used to make data that changes regularly available to Bro on a continuing basis.

**Contents**

### Warning

In contrast to the ASCII reader and writer, the SQLite plugins have not yet seen extensive use in production environments. While we are not aware of any issues with them, we urge to caution when using them in production environments. There could be lingering issues which only occur when the plugins are used with high amounts of data or in high-load environments.

### Logging Data into SQLite Databases

Logging support for SQLite is available in all Bro installations starting with version 2.2. There is no need to load any additional scripts or for any compile-time configurations.

Sending data from existing logging streams to SQLite is rather straightforward. You have to define a filter which specifies SQLite as the writer.

The following example code adds SQLite as a filter for the connection log:

```
sqlite-conn-filter.bro

event bro_init()
    {
    local filter: Log::Filter =
        [
        $name="sqlite",
        $path="/var/db/conn",
        $config=table(["tablename"] = "conn"),
        $writer=Log::WRITER_SQLITE
        ];

     Log::add_filter(Conn::LOG, filter);
    }
```

Bro will create the database file `/var/db/conn.sqlite`, if it does not already exist. It will also create a table with the name `conn` (if it does not exist) and start appending connection information to the table.

At the moment, SQLite databases are not rotated the same way ASCII log-files are. You have to take care to create them in an adequate location.

If you examine the resulting SQLite database, the schema will contain the same fields that are present in the ASCII log files:

```
# sqlite3 /var/db/conn.sqlite

SQLite version 3.8.0.2 2013-09-03 17:11:13
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .schema
CREATE TABLE conn (
'ts' double precision,
'uid' text,
'id.orig_h' text,
'id.orig_p' integer,
...
```

Note that the ASCII `conn.log` will still be created. To prevent this file from being created, you can remove the default filter:

To create a custom SQLite log file, you have to create a new log stream that contains just the information you want to commit to the database. Please refer to the *Logging Framework* documentation on how to create custom log streams.

### Reading Data from SQLite Databases

Like logging support, support for reading data from SQLite databases is built into Bro starting with version 2.2.

Just as with the text-based input readers (please refer to the *Input Framework* documentation for them and for basic information on how to use the input framework), the SQLite reader can be used to read data - in this case the result of SQL queries - into tables or into events.

### Reading Data into Tables

To read data from a SQLite database, we first have to provide Bro with the information, how the resulting data will be structured. For this example, we expect that we have a SQLite database, which contains host IP addresses and the user accounts that are allowed to log into a specific machine.

The SQLite commands to create the schema are as follows:

```
create table machines_to_users (
host text unique not null,
users text not null);

insert into machines_to_users values ('192.168.17.1', 'bernhard,matthias,seth');
insert into machines_to_users values ('192.168.17.2', 'bernhard');
insert into machines_to_users values ('192.168.17.3', 'seth,matthias');
```

After creating a file called `hosts.sqlite` with this content, we can read the resulting table into Bro:

```
sqlite-read-table.bro

type Idx: record {
    host: addr;
};

type Val: record {
    users: set[string];
};

global hostslist: table[addr] of Val = table();

event bro_init()
    {
    Input::add_table([$source="/var/db/hosts",
        $name="hosts",
        $idx=Idx,
        $val=Val,
        $destination=hostslist,
        $reader=Input::READER_SQLITE,
        $config=table(["query"] = "select * from machines_to_users;")
        ]);

    Input::remove("hosts");
    }

event Input::end_of_data(name: string, source: string)
```

```
    {
    if ( name != "hosts" )
        return;

    # now all data is in the table
    print "Hosts list has been successfully imported";

    # List the users of one host.
    print hostslist[192.168.17.1]$users;
    }
```

Afterwards, that table can be used to check logins into hosts against the available userlist.

### Turning Data into Events

The second mode is to use the SQLite reader to output the input data as events. Typically there are two reasons to do this. First, when the structure of the input data is too complicated for a direct table import. In this case, the data can be read into an event which can then create the necessary data structures in Bro in scriptland.

The second reason is, that the dataset is too big to hold it in memory. In this case, the checks can be performed on-demand, when Bro encounters a situation where it needs additional information.

An example for this would be an internal huge database with malware hashes. Live database queries could be used to check the sporadically happening downloads against the database.

The SQLite commands to create the schema are as follows:

```
create table malware_hashes (
hash text unique not null,
description text not null);

insert into malware_hashes values ('86f7e437faa5a7fce15d1ddcb9eaeaea377667b8',
↪'malware a');
insert into malware_hashes values ('e9d71f5ee7c92d6dc9e92ffdad17b8bd49418f98',
↪'malware b');
insert into malware_hashes values ('84a516841ba77a5b4648de2cd0dfcb30ea46dbb4',
↪'malware c');
insert into malware_hashes values ('3c363836cf4e16666669a25da280a1865c2d2874',
↪'malware d');
insert into malware_hashes values ('58e6b3a414a1e090dfc6029add0f3555ccba127f',
↪'malware e');
insert into malware_hashes values ('4a0a19218e082a343a1b17e5333409af9d98f0f5',
↪'malware f');
insert into malware_hashes values ('54fd1711209fb1c0781092374132c66e79e2241b',
↪'malware g');
insert into malware_hashes values ('27d5482eebd075de44389774fce28c69f45c8a75',
↪'malware h');
insert into malware_hashes values ('73f45106968ff8dc51fba105fa91306af1ff6666', 'ftp-
↪trace');
```

The following code uses the file-analysis framework to get the sha1 hashes of files that are transmitted over the network. For each hash, a SQL-query is run against SQLite. If the query returns with a result, we had a hit against our malware-database and output the matching hash.

```
sqlite-read-events.bro

@load frameworks/files/hash-all-files
```

```

type Val: record {
    hash: string;
    description: string;
};

event line(description: Input::EventDescription, tpe: Input::Event, r: Val)
    {
    print fmt("malware-hit with hash %s, description %s", r$hash, r$description);
    }

global malware_source = "/var/db/malware";

event file_hash(f: fa_file, kind: string, hash: string)
    {

    # check all sha1 hashes
    if ( kind=="sha1" )
        {
        Input::add_event(
            [
            $source=malware_source,
            $name=hash,
            $fields=Val,
            $ev=line,
            $want_record=T,
            $config=table(
                ["query"] = fmt("select * from malware_hashes where hash='%s';", hash)
                ),
            $reader=Input::READER_SQLITE
            ]);
        }
    }

event Input::end_of_data(name: string, source:string)
    {
    if ( source == malware_source )
        Input::remove(name);
    }
```

If you run this script against the trace in `testing/btest/Traces/ftp/ipv4.trace`, you will get one hit.

### Reading Data to Events

The second supported mode of the input framework is reading data to Bro events instead of reading them to a table.

Event streams work very similarly to table streams that were already discussed in much detail. To read the blacklist of the previous example into an event stream, the `Input::add_event` function is used. For example:

The main difference in the declaration of the event stream is, that an event stream needs no separate index and value declarations – instead, all source data types are provided in a single record definition.

Apart from this, event streams work exactly the same as table streams and support most of the options that are also supported for table streams.

## 3.1.4 Intelligence Framework

### Intro

Intelligence data is critical to the process of monitoring for security purposes. There is always data which will be discovered through the incident response process and data which is shared through private communities. The goals of Bro's Intelligence Framework are to consume that data, make it available for matching, and provide infrastructure around improving performance, memory utilization, and generally making all of this easier.

Data in the Intelligence Framework is an atomic piece of intelligence such as an IP address or an e-mail address along with a suite of metadata about it such as a freeform source field, a freeform descriptive field and a URL which might lead to more information about the specific item. The metadata in the default scripts has been deliberately kept minimal so that the community can find the appropriate fields that need to be added by writing scripts which extend the base record using the normal record extension mechanism.

### Quick Start

Refer to the "Loading Intelligence" section below to see the format for Intelligence Framework text files, then load those text files with this line in local.bro:

```
redef Intel::read_files += { "/somewhere/yourdata.txt" };
```

The text files need to reside only on the manager if running in a cluster.

Add the following line to local.bro in order to load the scripts that send "seen" data into the Intelligence Framework to be checked against the loaded intelligence data:

```
@load policy/frameworks/intel/seen
```

Intelligence data matches will be logged to the intel.log file.

### Architecture

The Intelligence Framework can be thought of as containing three separate portions. The first part is how intelligence is loaded, followed by the mechanism for indicating to the intelligence framework that a piece of data which needs to be checked has been seen, and thirdly the part where a positive match has been discovered.

### Loading Intelligence

Intelligence data can only be loaded through plain text files using the Input Framework conventions. Additionally, on clusters the manager is the only node that needs the intelligence data. The intelligence framework has distribution mechanisms which will push data out to all of the nodes that need it.

Here is an example of the intelligence data format (note that there will be additional fields if you are using CIF intelligence data or if you are using the policy/frameworks/intel/do_notice script). Note that all fields must be separated by a single tab character and fields containing only a hyphen are considered to be null values.

```
#fields indicator       indicator_type  meta.source     meta.desc       meta.url
1.2.3.4 Intel::ADDR     source1 Sending phishing email  http://source1.com/badhosts/1.
↪2.3.4
a.b.com Intel::DOMAIN   source2 Name used for data exfiltration -
```

For a list of all built-in *indicator_type* values, please refer to the documentation of `Intel::Type`.

Note that if you are using data from the Collective Intelligence Framework, then you will need to add the following line to your local.bro in order to support additional metadata fields used by CIF:

```
@load policy/integration/collective-intel
```

There is a simple mechanism to raise a Bro notice (of type Intel::Notice) for user-specified intelligence matches. To use this feature, add the following line to local.bro in order to support additional metadata fields (documented in the `Intel::MetaData` record):

```
@load policy/frameworks/intel/do_notice
```

To load the data once the files are created, use the following example to specify which files to load (with your own file names of course):

```
redef Intel::read_files += {
        "/somewhere/feed1.txt",
        "/somewhere/feed2.txt",
};
```

Remember, the files only need to be present on the file system of the manager node on cluster deployments.

#### Seen Data

When some bit of data is extracted (such as an email address in the "From" header in a message over SMTP), the Intelligence Framework needs to be informed that this data was discovered so that its presence will be checked within the loaded intelligence data. This is accomplished through the `Intel::seen` function, however typically users won't need to work with this function due to the scripts included with Bro that will call this function.

To load all of the scripts included with Bro for sending "seen" data to the intelligence framework, just add this line to local.bro:

```
@load policy/frameworks/intel/seen
```

Alternatively, specific scripts in that directory can be loaded. Keep in mind that as more data is sent into the intelligence framework, the CPU load consumed by Bro will increase depending on how many times the `Intel::seen` function is being called which is heavily traffic dependent.

#### Intelligence Matches

Against all hopes, most networks will eventually have a hit on intelligence data which could indicate a possible compromise or other unwanted activity. The Intelligence Framework provides an event that is generated whenever a match is discovered named `Intel::match`.

Due to design restrictions placed upon the intelligence framework, there is no assurance as to where this event will be generated. It could be generated on the worker where the data was seen or on the manager. When the `Intel::match` event is handled, only the data given as event arguments to the event can be assured since the host where the data was seen may not be where `Intel::match` is handled.

Intelligence matches are logged to the intel.log file. For a description of each field in that file, see the documentation for the `Intel::Info` record.

### 3.1.5 Logging Framework

Bro comes with a flexible key-value based logging interface that allows fine-grained control of what gets logged and how it is logged. This document describes how logging can be customized and extended.

**Contents**

### Terminology

Bro's logging interface is built around three main abstractions:

**Streams** A log stream corresponds to a single log. It defines the set of fields that a log consists of with their names and types. Examples are the `conn` stream for recording connection summaries, and the `http` stream for recording HTTP activity.

**Filters** Each stream has a set of filters attached to it that determine what information gets written out. By default, each stream has one default filter that just logs everything directly to disk. However, additional filters can be added to record only a subset of the log records, write to different outputs, or set a custom rotation interval. If all filters are removed from a stream, then output is disabled for that stream.

**Writers** Each filter has a writer. A writer defines the actual output format for the information being logged. The default writer is the ASCII writer, which produces tab-separated ASCII files. Other writers are available, like for binary output or direct logging into a database.

There are several different ways to customize Bro's logging: you can create a new log stream, you can extend an existing log with new fields, you can apply filters to an existing log stream, or you can customize the output format by setting log writer options. All of these approaches are described in this document.

### Streams

In order to log data to a new log stream, all of the following needs to be done:

- A `record` type must be defined which consists of all the fields that will be logged (by convention, the name of this record type is usually "Info").

- A log stream ID (an *enum* with type name "Log::ID") must be defined that uniquely identifies the new log stream.
- A log stream must be created using the `Log::create_stream` function.
- When the data to be logged becomes available, the `Log::write` function must be called.

In the following example, we create a new module "Foo" which creates a new log stream.

In the definition of the "Info" record above, notice that each field has the *&log* attribute. Without this attribute, a field will not appear in the log output. Also notice one field has the *&optional* attribute. This indicates that the field might not be assigned any value before the log record is written. Finally, a field with the *&default* attribute has a default value assigned to it automatically.

At this point, the only thing missing is a call to the `Log::write` function to send data to the logging framework. The actual event handler where this should take place will depend on where your data becomes available. In this example, the *connection_established* event provides our data, and we also store a copy of the data being logged into the `connection` record:

If you run Bro with this script, a new log file `foo.log` will be created. Although we only specified four fields in the "Info" record above, the log output will actually contain seven fields because one of the fields (the one named "id") is itself a record type. Since a `conn_id` record has four fields, then each of these fields is a separate column in the log output. Note that the way that such fields are named in the log output differs slightly from the way we would refer to the same field in a Bro script (each dollar sign is replaced with a period). For example, to access the first field of a `conn_id` in a Bro script we would use the notation `id$orig_h`, but that field is named `id.orig_h` in the log output.

When you are developing scripts that add data to the `connection` record, care must be given to when and how long data is stored. Normally data saved to the connection record will remain there for the duration of the connection and from a practical perspective it's not uncommon to need to delete that data before the end of the connection.

### Add Fields to a Log

You can add additional fields to a log by extending the record type that defines its content, and setting a value for the new fields before each log record is written.

Let's say we want to add a boolean field `is_private` to `Conn::Info` that indicates whether the originator IP address is part of the **RFC 1918** space:

As this example shows, when extending a log stream's "Info" record, each new field must always be declared either with a `&default` value or as `&optional`. Furthermore, you need to add the `&log` attribute or otherwise the field won't appear in the log file.

Now we need to set the field. Although the details vary depending on which log is being extended, in general it is important to choose a suitable event in which to set the additional fields because we need to make sure that the fields are set before the log record is written. Sometimes the right choice is the same event which writes the log record, but at a higher priority (in order to ensure that the event handler that sets the additional fields is executed before the event handler that writes the log record).

In this example, since a connection's summary is generated at the time its state is removed from memory, we can add another handler at that time that sets our field correctly:

Now `conn.log` will show a new field `is_private` of type `bool`. If you look at the Bro script which defines the connection log stream /scripts/base/protocols/conn/main.bro, you will see that `Log::write` gets called in an event handler for the same event as used in this example to set the additional fields, but at a lower priority than the one used in this example (i.e., the log record gets written after we assign the `is_private` field).

For extending logs this way, one needs a bit of knowledge about how the script that creates the log stream is organizing its state keeping. Most of the standard Bro scripts attach their log state to the `connection` record where it can then be accessed, just like `c$conn` above. For example, the HTTP analysis adds a field `http` of type `HTTP::Info` to the `connection` record.

### Define a Logging Event

Sometimes it is helpful to do additional analysis of the information being logged. For these cases, a stream can specify an event that will be generated every time a log record is written to it. To do this, we need to modify the example module shown above to look something like this:

All of Bro's default log streams define such an event. For example, the connection log stream raises the event `Conn::log_conn`. You could use that for example for flagging when a connection to a specific destination exceeds a certain duration:

Often, these events can be an alternative to post-processing Bro logs externally with Perl scripts. Much of what such an external script would do later offline, one may instead do directly inside of Bro in real-time.

### Disable a Stream

One way to "turn off" a log is to completely disable the stream. For example, the following example will prevent the conn.log from being written:

Note that this must run after the stream is created, so the priority of this event handler must be lower than the priority of the event handler where the stream was created.

### Filters

A stream has one or more filters attached to it (a stream without any filters will not produce any log output). When a stream is created, it automatically gets a default filter attached to it. This default filter can be removed or replaced, or other filters can be added to the stream. This is accomplished by using either the `Log::add_filter` or `Log::remove_filter` function. This section shows how to use filters to do such tasks as rename a log file, split the output into multiple files, control which records are written, and set a custom rotation interval.

### Rename Log File

Normally, the log filename for a given log stream is determined when the stream is created, unless you explicitly specify a different one by adding a filter.

The easiest way to change a log filename is to simply replace the default log filter with a new filter that specifies a value for the "path" field. In this example, "conn.log" will be changed to "myconn.log":

Keep in mind that the "path" field of a log filter never contains the filename extension. The extension will be determined later by the log writer.

### Add a New Log File

Normally, a log stream writes to only one log file. However, you can add filters so that the stream writes to multiple files. This is useful if you want to restrict the set of fields being logged to the new file.

In this example, a new filter is added to the Conn::LOG stream that writes two fields to a new log file:

Notice how the "include" filter attribute specifies a set that limits the fields to the ones given. The names correspond to those in the `Conn::Info` record (however, because the "id" field is itself a record, we can specify an individual field of "id" by the dot notation shown in the example).

Using the code above, in addition to the regular `conn.log`, you will now also get a new log file `origs.log` that looks like the regular `conn.log`, but will have only the fields specified in the "include" filter attribute.

If you want to skip only some fields but keep the rest, there is a corresponding `exclude` filter attribute that you can use instead of `include` to list only the ones you are not interested in.

If you want to make this the only log file for the stream, you can remove the default filter:

### Determine Log Path Dynamically

Instead of using the "path" filter attribute, a filter can determine output paths *dynamically* based on the record being logged. That allows, e.g., to record local and remote connections into separate files. To do this, you define a function that returns the desired path, and use the "path_func" filter attribute:

Running this will now produce two new files, `conn-local.log` and `conn-remote.log`, with the corresponding entries (for this example to work, the `Site::local_nets` must specify your local network). One could extend this further for example to log information by subnets or even by IP address. Be careful, however, as it is easy to create many files very quickly.

The `myfunc` function has one drawback: it can be used only with the `Conn::LOG` stream as the record type is hardcoded into its argument list. However, Bro allows to do a more generic variant:

This function can be used with all log streams that have records containing an `id: conn_id` field.

### Filter Log Records

We have seen how to customize the columns being logged, but you can also control which records are written out by providing a predicate that will be called for each log record:

This will result in a new log file `conn-http.log` that contains only the log records from `conn.log` that are analyzed as HTTP traffic.

### Rotation

The log rotation interval is globally controllable for all filters by redefining the `Log::default_rotation_interval` option (note that when using BroControl, this option is set automatically via the BroControl configuration).

Or specifically for certain `Log::Filter` instances by setting their `interv` field. Here's an example of changing just the `Conn::LOG` stream's default filter rotation.

### Writers

Each filter has a writer. If you do not specify a writer when adding a filter to a stream, then the ASCII writer is the default.

There are two ways to specify a non-default writer. To change the default writer for all log filters, just redefine the `Log::default_writer` option. Alternatively, you can specify the writer to use on a per-filter basis by setting a value for the filter's "writer" field. Consult the documentation of the writer to use to see if there are other options that are needed.

### ASCII Writer

By default, the ASCII writer outputs log files that begin with several lines of metadata, followed by the actual log output. The metadata describes the format of the log file, the "path" of the log (i.e., the log filename without file extension), and also specifies the time that the log was created and the time when Bro finished writing to it. The ASCII writer has a number of options for customizing the format of its output, see /scripts/base/frameworks/logging/writers/ascii.bro. If you change the output format options, then be careful to check whether your postprocessing scripts can still recognize your log files.

Some writer options are global (i.e., they affect all log filters using that log writer). For example, to change the output format of all ASCII logs to JSON format:

Some writer options are filter-specific (i.e., they affect only the filters that explicitly specify the option). For example, to change the output format of the `conn.log` only:

### Other Writers

Bro supports the following additional built-in output formats:

Additional writers are available as external plugins:

### A Collection of Plugins for Bro

This repository contains a set of Bro plugins provided by the Bro Project, as well as plugins contributed by the Bro community.

These plugins are not part of a standard Bro installation, but can be built and installed separately (and individually). Please note that some might be experimental, and they generally also see less testing due to often depending on non-standard dependencies. Use at your own risk.

### Bro::AF_Packet

This plugin provides native AF_Packet <http://man7.org/linux/man-pages/man7/packet.7.html> support for Bro.

### Installation

Make sure the kernel headers are installed and your kernel supports PACKET_FANOUT and TPACKET_V3. The following will then compile and install the af_packet plugin alongside Bro, assuming it can find the kernel headers in a standard location:

```
# ./configure && make && make install
```

If the headers are installed somewhere non-standard, add `--with-kernel=<kernel-header-directory>` to the `configure` command. If everything built and installed correctly, you should see this:

```
# bro -N Bro::AF_Packet
Bro::AF_Packet - Packet acquisition via AF_Packet (dynamic, version 1.0)
[Packet Source] AF_PacketReader (interface prefix "af_packet"; supports live input)
[Constant] AF_Packet::buffer_size
[Constant] AF_Packet::enable_hw_timestamping
[Constant] AF_Packet::enable_fanout
[Constant] AF_Packet::fanout_id
```

#### Usage

Once installed, you can use AF_Packet interfaces/ports by prefixing them with `af_packet::` on the command line. For example, to use AF_Packet to monitor interface `eth0`:

```
# bro -i af_packet::eth0
```

To use AF_Packet, running Bro without root privileges, the Bro processes needs the CAP_NET_RAW capability. You can set it with the following command (on each sensor, after broctl install):

```
# setcap cap_net_raw+eip <path_to_bro>/bin/bro
```

The AF_Packet plugin automatically enables promiscuous mode on the interfaces. As the plugin is using PACKET_ADD_MEMBERSHIP to enter the promiscuous mode without interfering others, the PROMISC flag is not touched. To verify that the interface entered promiscuous mode you can use `dmesg`.

To adapt the plugin to your needs, you can set a couple of parameters like buffer size. See scripts/init.bro for the default values.

### Indexed Logging Output with ElasticSearch

#### Intro

Bro's default ASCII log format is not exactly the most efficient way for searching large volumes of data. ElasticSearch is a new data storage technology for dealing with tons of data. It's also a search engine built on top of Apache's Lucene project. It scales very well, both for distributed indexing and distributed searching.

**Contents**

#### Warning

This writer plugin only supports ElasticSearch 1; it will not work with ElasticSearch version 2 and above. This writer plugin is deprecated and will be removed from the Bro distribution in the future. This plugin is experimental and not recommended for production use; it is for example missing error handling and may loose messages.

#### Installing ElasticSearch

Download the latest version from: http://www.elasticsearch.org/download/. Once extracted, start ElasticSearch with:

```
# ./bin/elasticsearch
```

For more detailed information, refer to the ElasticSearch installation documentation: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/setup.html

### Installing the ElasticSearch Plugin

First, ensure that you have libcurl (headers and library) installed. Then the following will compile and install the plugin alongside Bro:

```
# ./configure && make && make install
```

See the output of `./configure --help` for additional options if it can't find any of the prerequisites.

If everything built and installed correctly, you should see this:

```
# bro -N Bro::ElasticSearch
Bro::ElasticSearch - ElasticSearch log writer (dynamic, version 1.0)
```

### Activating ElasticSearch

The easiest way to enable ElasticSearch output is to load the plugin's `logs-to-elasticsearch.bro` script. If you are using BroControl, the following line in local.bro will enable it:

With that, Bro will now write most of its logs into ElasticSearch in addition to maintaining the Ascii logs like it would do by default. That script has some tunable options for choosing which logs to send to ElasticSearch, refer to the autogenerated script documentation for those options.

There is an interface being written specifically to integrate with the data that Bro outputs into ElasticSearch named Brownian. It can be found here:

```
https://github.com/grigorescu/Brownian
```

### Tuning

A common problem encountered with ElasticSearch is too many files being held open. The ElasticSearch documentation has some suggestions on this and other issues.

- http://www.elastic.co/guide/en/elasticsearch/reference/1.3/setup-configuration.html

### TODO

Lots.

- Perform multicast discovery for server.
- Better error detection.
- Better defaults (don't index loaded-plugins, for instance).

### Writing Logging Output to Kafka

A log writer that sends logging output to Kafka. This provides a convenient means for tools in the Hadoop ecosystem, such as Storm, Spark, and others, to process the data generated by Bro.

**Contents**

#### Installation

Install librdkafka (https://github.com/edenhill/librdkafka), a native client library for Kafka. This plugin has been tested against the latest release of librdkafka, which at the time of this writing is v0.8.6:

```
# curl -L https://github.com/edenhill/librdkafka/archive/0.8.6.tar.gz | tar xvz
# cd librdkafka-0.8.6/
# ./configure
# make
# sudo make install
```

Then compile this Bro plugin using the following commands:

```
# ./configure --bro-dist=$BRO_SRC
# make
# sudo make install
```

Run the following command to ensure that the plugin was installed successfully:

```
# bro -N Bro::Kafka
Bro::Kafka - Writes logs to Kafka (dynamic, version 0.1)
```

#### Activation

The easiest way to enable Kafka output is to load the plugin's `logs-to-kafka.bro` script. If you are using BroControl, any of the following examples added to local.bro will activate it.

In this example, all HTTP, DNS, and Conn logs will be sent to a Kafka Broker running on the localhost. By default, the log stream's path will define the topic name. The `Conn::LOG` will be sent to the topic `conn` and the `HTTP::LOG` will be sent to the topic named `http`.

If all log streams need to be sent to the same topic, define the name of the topic in a variable called `topic_name`. In this example, both `Conn::LOG` and `HTTP::LOG` will be sent to the topic named `bro`.

It is also possible to send each log stream to a unique topic and also customize those topic names. This can be done through the same mechanism in which the name of a log file for a stream is customized. Here is an old example (look for the $path_func field) http://blog.bro.org/2012/02/filtering-logs-with-bro.html.

#### Settings

`kafka_conf`

The global configuration settings for Kafka. These values are passed through directly to librdkafka. Any valid librdkafka settings can be defined in this table.

`topic_name`

The name of the topic in Kafka that *all* Bro log streams will be sent to. If each log stream needs to be sent to a unique topic, this value should be left undefined.

`max_wait_on_shutdown`

The maximum number of milliseconds that the plugin will wait for any backlog of queued messages to be sent to Kafka before forced shutdown.

`tag_json`

If true, a log stream identifier is appended to each JSON-formatted message. For example, a Conn::LOG message will look like `{ 'conn' : { ... }}`.

### Bro::Myricom

This plugin provides native *Myricom SNF v3+v4* support for Bro.

#### Installation

Follow Myricom's instructions to get its kernel module and userspace libraries installed, then use the following commands to configure and build the plugin. In most cases, if you are building this plugin from the Bro source tree, you won't need any configure arguments:

```
./configure --with-myricom=<path to sniffer sources> --bro-dist=<path to bro sources>
make && sudo make install
```

If everything built and installed correctly, you should see this:

```
# bro -N Bro::Myricom
Bro::Myricom - Packet acquisition via Myricom SNF v3 (dynamic, version 1.0)
```

You may run Bro as unprivileged user.

#### Usage

Once installed, you can use Myricom interfaces/ports by prefixing them with `myricom::` on the command line. For example, to use Myricom SNF to monitor interface `p2p1`:

```
bro -i myricom::p2p1
```

To use it in production with multiple Bro processes, use a configuration similar to this in node.cfg:

```
[worker-1]
type=worker
host=localhost
lb_method=custom
```

```
lb_procs=<number of processes, like 16>
interface=myricom::<interface name, like p2p1>
```

If you would like to sniff all Myricom interfaces on a system and merge them together, there is a special interface name available of `myricom::*`. It uses a special feature of the Myricom SNF library for port aggregation.

To run a cluster sniffing all Myricom interfaces on a system, you can use the same configuration as above, but with the special interface name that aggregates all Myricom ports like this:

```
[worker-1]
type=worker
host=localhost
lb_method=custom
lb_procs=<number of processes, like 16>
interface=myricom::*
```

### Tuning

You may wish to tune the amount of memory used for the global packet buffer. This setting is available in the Bro script interface to the plugin, but it's also available as a global option in `broctl.cfg` or as a per-node option in `node.cfg`. The following like in either config file will set the SNF packet buffer ring size to 16GB:

```
myricom.snf_ring_size=16384
```

Enjoy!

## Bro::Netmap

This plugin provides native netmap support for Bro.

### Installation

Follow netmap's instructions to get its kernel module and, potentially, custom drivers installed. The following will then compile and install the netmap plugin alongside Bro, assuming it can find the netmap headers in a standard location:

```
# ./configure && make && make install
```

If the headers are installed somewhere non-standard, add `--with-netmap=<netmap-base-directory>` to the `configure` command. If everything built and installed correctly, you should see this:

```
# bro -N Bro::Netmap
Bro::Netmap - Packet acquisition via netmap (dynamic, version 1.0)
```

To use netmap, Bro needs read and write access to `/dev/netmap`. If you give that permission to a user, you can run Bro as non-root.

### Usage

Once installed, you can use netmap interfaces/ports by prefixing them with either `netmap::` or `vale::` on the command line. For example, to use netmap to monitor interface `eth0`:

```
bro -i netmap::eth0
```

Netmap does not enable promiscuous mode on interfaces, you'll have to do that yourself. For example, on Linux:

```
ifconfig eth0 promisc
```

### Bro::PF_RING

This plugin provides native PF_RING support for Bro.

#### Installation

Follow PF_RING's instructions to get its kernel module and, potentially, custom drivers installed. The following will then compile and install the PF_RING plugin alongside Bro, assuming it can find the PF_RING headers in a standard location:

```
./configure && make && make install
```

If the headers are installed somewhere non-standard, add `--with-pfring=<PF_RING-base-directory>` to the `configure` command. If everything built and installed correctly, you should see this:

```
# bro -N Bro::PF_RING
Bro::PF_RING - Packet acquisition via PF_RING (dynamic, version 1.0)
```

To use PF_RING, you should run Bro as root.

#### Usage

Once installed, you can use PF_RING interfaces/ports by prefixing them with `pf_ring::` on the command line. For example, to use PF_RING to monitor interface `eth0`:

```
bro -i pf_ring::eth0
```

### Bro Redis Logging

Log filter for the Redis key/value DB; see http://redis.io.

This is not a very common logging format for Bro, as you lose query ability on all log fields except the key field. One can find it useful to store (temporary) metadata about specific network events.

All log fields get formatted as a JSON string and saved as a Redis value. The keys for these values are another log field, selectable for each log filter.

Be aware that duplicate key values will overwrite the corresponding associated values.

#### Warning

This plugin, almost certainly, will be useful mainly for custom log streams (see https://www.bro.org/sphinx/frameworks/logging.html). This code is not production ready! If there are problems with the DB connection, buffered data will be lost!

### Installing

First, install the libhiredis library and headers. On Ubuntu do:

```
sudo apt-get install libhiredis-dev
```

To install this plugin run:

```
./configure --bro-dist=<path_to_bro_build> && make && sudo make install
```

To check if everything installed succesfully run:

```
# bro -N Bro::Redis
Bro::Redis - Redis log writer (dynamic, version 1.0)
```

There are also a set of tests that can be run:

```
make test
```

### Howto

Global default configs for all Redis filters:

```
<bro_install>/lib/bro/plugins/Bro_Redis/scripts/init.bro
```

Defaults can be changed with redef statements, or by setting a $config table for each filter.

You can select the log field that will become the key, either with ‘key_index’ or ‘key’. With ‘key’ log field names will be checked and the first field that matches will become the key for this log stream (it will overwrite ‘key_index’). The key can also be prepended (namespaced) by setting ‘key_prefix’, or you can change the database by setting ‘db’. Setting ‘unix_path’ overwrites ‘server_host’.

### Example

Filter for a custom log stream that outputs extracted file’s metadata. The ‘dump_file’ log field is the Redis key.

### Extended TCP Analysis

TCPRS is a TCP traffic analyzer that specializes in the detection and classification of retransmission and network reordering events.

The following forms of events are available in the TCPRS analyzer:

- Dead connection detection
- TCP option detection
- Retransmission detection and classification
- Limited Transmit and Fast Recovery detection
- Network reordering detection and classification
- RTT and initial RTO measurements

To activate all of the new functionality, load `Bro/TCPRS`. To use the analyzer without the use of any of the provided scripts, you can enable it inside a `bro_init` handler:

```
event bro_init()
        {
    TCPRS::EnableTCPRSAnalyzer();
    }
```

Included with the analyzer is a collection of 103 test cases that are used for iterative design and refinement of the analyzer. Each test case is used to verify a specific function of the analyzer or general classification of events.

Please refer to the documentation for each plugin for information on installation and usage; refer to their COPYING files for licensing information. For more information and feedback, please see the MAINTAINER file for contact information.

The Bro plugins git repository is located at git://git.bro.org/bro-plugins.git. You can browse the repository here. If you would like to see your plugin included in the repository, please write to info@bro.org.

### 3.1.6 NetControl Framework

Bro can connect with network devices like, for example, switches or soft- and hardware firewalls using the NetControl framework. The NetControl framework provides a flexible, unified interface for active response and hides the complexity of heterogeneous network equipment behind a simple task-oriented API, which is easily usable via Bro scripts. This document gives an overview of how to use the NetControl framework in different scenarios; to get a better understanding of how it can be used in practice, it might be worthwhile to take a look at the unit tests.

**Contents**

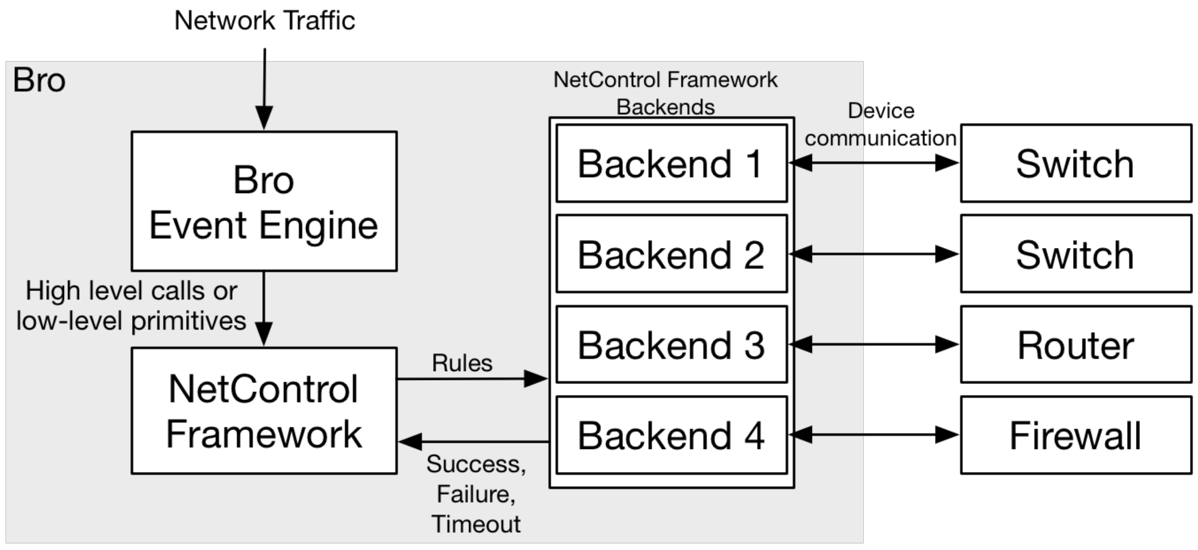


Fig. 3.1: NetControl architecture (click to enlarge).

### NetControl Architecture

The basic architecture of the NetControl framework is shown in the figure above. Conceptually, the NetControl framework sits inbetween the user provided scripts (which use the Bro event engine) and the network device (which can either be a hardware or software device), that is used to implement the commands.

The NetControl framework supports a number of high-level calls, like the `NetControl::drop_address` function, or lower a lower level rule syntax. After a rule has been added to the NetControl framework, NetControl sends the rule to one or several of its *backends*. Each backend is responsible to communicate with a single hard- or software device. The NetControl framework tracks rules throughout their entire lifecycle and reports the status (like success, failure and timeouts) back to the user scripts.

The backends are implemented as Bro scripts using a plugin based API; an example for this is /scripts/base/frameworks/netcontrol/plugins/broker.bro. This document will show how to write plugins in *NetControl Plugins*.

### NetControl API

#### High-level NetControl API

In this section, we will introduce the high level NetControl API. As mentioned above, NetControl uses *backends* to communicate with the external devices that will implement the rules. You will need at least one active backend before you can use NetControl. For our examples, we will just use the debug plugin to create a backend. This plugin outputs all actions that are taken to the standard output.

Backends should be initialized in the `NetControl::init` event, calling the `NetControl::activate` function after the plugin instance has been initialized. The debug plugin can be initialized as follows:

After at least one backend has been added to the NetControl framework, the framework can be used and will send added rules to the added backend.

The NetControl framework contains several high level functions that allow users to drop connections of certain addresses and networks, shunt network traffic, etc. The following table shows and describes all of the currently available high-level functions.

| Function | Description |
|---|---|
| `NetControl::drop_address` | Calling this function causes NetControl to block all packets involving an IP address from being forwarded |
| `NetControl::drop_connection` | Calling this function stops all packets of a specific connection (identified by its 5-tuple) from being forwarded. |
| `NetControl::drop_address` | Calling this function causes NetControl to block all packets involving an IP address from being forwarded |
| `NetControl::drop_address_catch_release` | Calling this function causes all packets of a specific source IP to be blocked. This function uses catch-and-release functionality and the IP address is only dropped for a short amount of time to conserve rule space in the network hardware. It is immediately re-dropped when it is seen again in traffic. See *Catch and Release* for more information. |
| `NetControl::shunt_flow` | Calling this function causes NetControl to stop forwarding a uni-directional flow of packets to Bro. This allows Bro to conserve resources by shunting flows that have been identified as being benign. |
| `NetControl::redirect_flow` | Calling this function causes NetControl to redirect an uni-directional flow to another port of the networking hardware. |
| `NetControl::quarantine_host` | Calling this function allows Bro to quarantine a host by sending DNS traffic to a host with a special DNS server, which resolves all queries as pointing to itself. The quarantined host is only allowed between the special server, which will serve a warning message detailing the next steps for the user |
| `NetControl::whitelist_address` | Calling this function causes NetControl to push a whitelist entry for an IP address to the networking hardware. |
| `NetControl::whitelist_subnet` | Calling this function causes NetControl to push a whitelist entry for a subnet to the networking hardware. |

After adding a backend, all of these functions can immediately be used and will start sending the rules to the added backend(s). To give a very simple example, the following script will simply block the traffic of all connections that it sees being established:

Running this script on a file containing one connection will cause the debug plugin to print one line to the standard output, which contains information about the rule that was added. It will also cause creation of *netcontrol.log*, which contains information about all actions that are taken by NetControl:

```
ERROR executing test 'doc.sphinx.netcontrol-1-drop-with-debug.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/tls/ecdhe.pcap ${DOC_ROOT}/frameworks/netcontrol-
↪1-drop-with-debug.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-1-drop-with-debug.bro
```

In our case, *netcontrol.log* contains several `NetControl::MESSAGE` entries, which show that the debug plugin has been initialized and added. Afterwards, there are two `NetControl::RULE` entries; the first shows that the addition of a rule has been requested (state is `NetControl::REQUESTED`). The following line shows that the rule was successfully added (the state is `NetControl::SUCCEEDED`). The remainder of the log line gives more information about the added rule, which in our case applies to a specific 5-tuple.

In addition to the netcontrol.log, the drop commands also create a second, additional log called *netcontrol_drop.log*. This log file is much more succinct and only contains information that is specific to drops that are enacted by NetControl:

```
ERROR executing test 'doc.sphinx.netcontrol-1-drop-with-debug.bro' (part 2)

% 'btest-rst-cmd cat netcontrol_drop.log' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
 ↪netcontrol-1-drop-with-debug.bro
 cat: netcontrol_drop.log: No such file or directory
```

While this example of blocking all connections is usually not very useful, the high-level API gives an easy way to take action, for example when a host is identified doing some harmful activity. To give a more realistic example, the following code automatically blocks a recognized SSH guesser:

```
ERROR executing test 'doc.sphinx.netcontrol-2-ssh-guesser.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/ssh/sshguess.pcap ${DOC_ROOT}/frameworks/
 ↪netcontrol-2-ssh-guesser.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
 ↪netcontrol-2-ssh-guesser.bro
```

Note that in this case, instead of calling NetControl directly, we also can use the `Notice::ACTION_DROP` action of the notice framework:

```
ERROR executing test 'doc.sphinx.netcontrol-3-ssh-guesser.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/ssh/sshguess.pcap ${DOC_ROOT}/frameworks/
 ↪netcontrol-3-ssh-guesser.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
 ↪netcontrol-3-ssh-guesser.bro
```

Using the `Notice::ACTION_DROP` action of the notice framework also will cause the *dropped* column in *notice.log* to be set to true each time that the NetControl framework enacts a block:

```
ERROR executing test 'doc.sphinx.netcontrol-3-ssh-guesser.bro' (part 2)

% 'btest-rst-cmd cat notice.log' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
 ↪netcontrol-3-ssh-guesser.bro
 cat: notice.log: No such file or directory
```

## Rule API

As already mentioned in the last section, in addition to the high-level API, the NetControl framework also supports a Rule based API which allows greater flexibility while adding rules. Actually, all the high-level functions are implemented using this lower-level rule API; the high-level functions simply convert their arguments into the lower-level rules and then add the rules directly to the NetControl framework (by calling `NetControl::add_rule`).

The following figure shows the main components of NetControl rules:

The types that are used to make up a rule are defined in /scripts/base/frameworks/netcontrol/types.bro.

Rules are defined as a `NetControl::Rule` record. Rules have a *type*, which specifies what kind of action is taken. The possible actions are to **drop** packets, to **modify** them, to **redirect** or to **whitelist** them. The *target* of a rule specifies if the rule is applied in the *forward path*, and affects packets as they are forwarded through the network, or if it affects the *monitor path* and only affects the packets that are sent to Bro, but not the packets that traverse the network. The *entity* specifies the address, connection, etc. that the rule applies to. In addition, each notice has a *timeout* (which

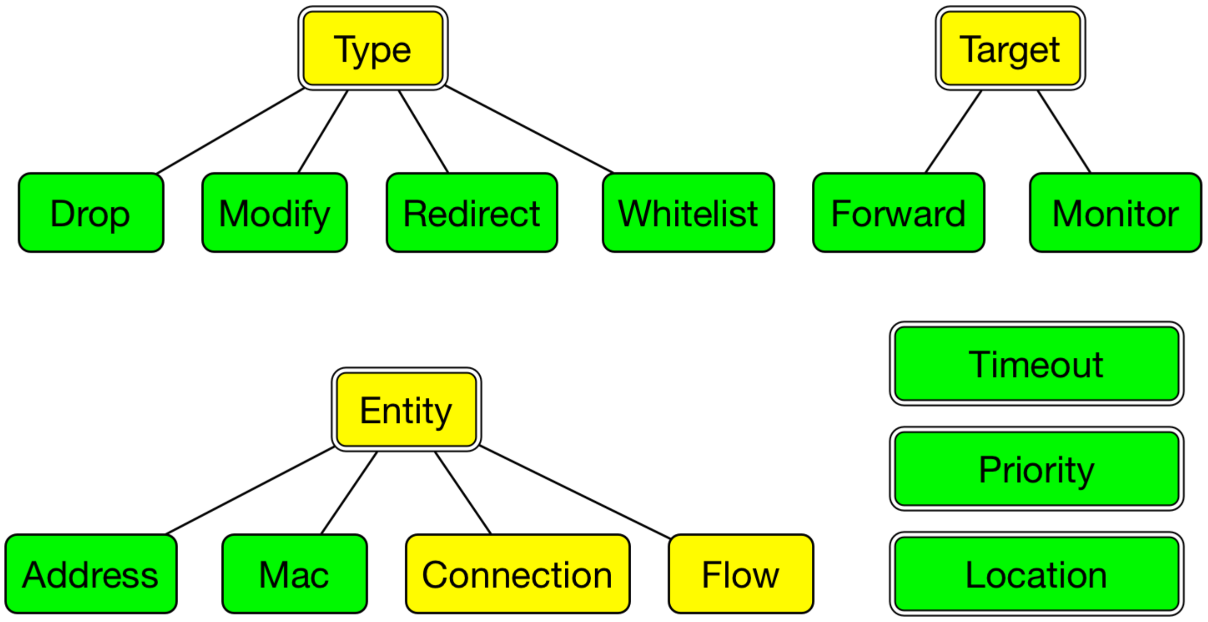


Fig. 3.2: NetControl Rule overview (click to enlarge).

can be left empty), a *priority* (with higher priority rules overriding lower priority rules). Furthermore, a *location* string with more text information about each rule can be provided.

There are a couple more fields that only needed for some rule types. For example, when you insert a redirect rule, you have to specify the port that packets should be redirected too. All these fields are shown in the `NetControl::Rule` documentation.

To give an example on how to construct your own rule, we are going to write our own version of the `NetControl::drop_connection` function. The only difference between our function and the one provided by NetControl is the fact that the NetControl function has additional functionality, e.g. for logging.

Once again, we are going to test our function with a simple example that simply drops all connections on the Network:

```
ERROR executing test 'doc.sphinx.netcontrol-4-drop.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/tls/ecdhe.pcap ${DOC_ROOT}/frameworks/netcontrol-
↪4-drop.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-4-drop.bro
```

The last example shows that `NetControl::add_rule` returns a string identifier that is unique for each rule (uniqueness is not preserved across restarts or Bro). This rule id can be used to later remove rules manually using `NetControl::remove_rule`.

Similar to `NetControl::add_rule`, all the high-level functions also return their rule IDs, which can be removed in the same way.

### Interacting with Rules

The NetControl framework offers a number of different ways to interact with Rules. Before a rule is applied by the framework, a number of different hooks allow you to either modify or discard rules before they are added. Furthermore, a number of events can be used to track the lifecycle of a rule while it is being managed by the NetControl framework. It is also possible to query and access the current set of active rules.

### Rule Policy

The hook `NetControl::rule_policy` provides the mechanism for modifying or discarding a rule before it is sent onwards to the backends. Hooks can be thought of as multi-bodied functions and using them looks very similar to handling events. In difference to events, they are processed immediately. Like events, hooks can have priorities to sort the order in which they are applied. Hooks can use the `break` keyword to show that processing should be aborted; if any `NetControl::rule_policy` hook uses `break`, the rule will be discarded before further processing.

Here is a simple example which tells Bro to discard all rules for connections originating from the 192.168.* network:

```
ERROR executing test 'doc.sphinx.netcontrol-5-hook.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/tls/ecdhe.pcap ${DOC_ROOT}/frameworks/netcontrol-
↪5-hook.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-5-hook.bro
```

### NetControl Events

In addition to the hooks, the NetControl framework offers a variety of events that are raised by the framework to allow users to track rules, as well as the state of the framework.

We already encountered and used one event of the NetControl framework, `NetControl::init`, which is used to initialize the framework. After the framework has finished initialization and will start accepting rules, the `NetControl::init_done` event will be raised.

When rules are added to the framework, the following events will be called in this order:

| Event | Description |
|---|---|
| `NetControl::rule_new` | Signals that a new rule is created by the NetControl framework due to `NetControl::add_rule`. At this point of time, the rule has not yet been added to any backend. |
| `NetControl::rule_added` | Signals that a new rule has successfully been added by a backend. |
| `NetControl::rule_exists` | This event is raised instead of `NetControl::rule_added` when a backend reports that a rule was already existing. |
| `NetControl::rule_timeout` | Signals that a rule timeout was reached. If the hardware does not support automatic timeouts, the NetControl framework will automatically call bro:see:*NetControl::remove_rule*. |
| `NetControl::rule_removed` | Signals that a new rule has successfully been removed a backend. |
| `NetControl::rule_destroyed` | This event is the pendant to `NetControl::rule_added`, and reports that a rule is no longer be tracked by the NetControl framework. This happens, for example, when a rule was removed from all backend. |
| `NetControl::rule_error` | This event is raised whenever an error occurs during any rule operation. |

### Finding active rules

The NetControl framework provides two functions for finding currently active rules: `NetControl::find_rules_addr` finds all rules that affect a certain IP address and `NetControl::find_rules_subnet` finds all rules that affect a specified subnet.

Consider, for example, the case where a Bro instance monitors the traffic at the border, before any firewall or switch rules were applied. In this case, Bro will still be able to see connection attempts of already blocked IP addresses. In this case, `NetControl::find_rules_addr` could be used to check if an address already was blocked in the past.

Here is a simple example, which uses a trace that contains two connections from the same IP address. After the first connection, the script recognizes that the address is already blocked in the second connection.

```
ERROR executing test 'doc.sphinx.netcontrol-6-find.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/tls/google-duplicate.trace ${DOC_ROOT}/
↪frameworks/netcontrol-6-find.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-6-find.bro
```

Notice that the functions return vectors because it is possible that several rules exist simultaneously that affect one IP; either there could be rules with different priorities, or rules for the subnet that an IP address is part of.

### Catch and Release

We already mentioned earlier that in addition to the `NetControl::drop_connection` and `NetControl::drop_address` functions, which drop a connection or address for a specified amount of time, NetControl also comes with a blocking function that uses an approach called *catch and release*.

Catch and release is a blocking scheme that conserves valuable rule space in your hardware. Instead of using long-lasting blocks, catch and release first only installs blocks for short amount of times (typically a few minutes). After these minutes pass, the block is lifted, but the IP address is added to a watchlist and the IP address will immediately be re-blocked again (for a longer amount of time), if it is seen reappearing in any traffic, no matter if the new traffic triggers any alert or not.

This makes catch and release blocks similar to normal, longer duration blocks, while only requiring a small amount of space for the currently active rules. IP addresses that only are seen once for a short time are only blocked for a few minutes, monitored for a while and then forgotten. IP addresses that keep appearing will get re-blocked for longer amounts of time.

In difference to the other high-level functions that we documented so far, the catch and release functionality is much more complex and adds a number of different specialized functions to NetControl. The documentation for catch and release is contained in the file /scripts/base/frameworks/netcontrol/catch-and-release.bro.

Using catch and release in your scripts is easy; just use `NetControl::drop_address_catch_release` like in this example:

```
ERROR executing test 'doc.sphinx.netcontrol-7-catch-release.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/tls/ecdhe.pcap ${DOC_ROOT}/frameworks/netcontrol-
↪7-catch-release.bro' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-7-catch-release.bro
```

Note that you do not have to provide the block time for catch and release; instead, catch and release uses the time intervals specified in `NetControl::catch_release_intervals` (by default 10 minutes, 1 hour, 24 hours, 7 days). That means when an address is first blocked, it is blocked for 10 minutes and monitored for 1 hour. If the address reappears after the first 10 minutes, it is blocked for 1 hour and then monitored for 24 hours, etc.

Catch and release adds its own new logfile in addition to the already existing ones (netcontrol_catch_release.log):

```
ERROR executing test 'doc.sphinx.netcontrol-7-catch-release.bro' (part 2)

% 'btest-rst-cmd cat netcontrol_catch_release.log' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
 ↪netcontrol-7-catch-release.bro
 cat: netcontrol_catch_release.log: No such file or directory
```

In addition to the blocking function, catch and release comes with the `NetControl::get_catch_release_info` function to check if an address is already blocked by catch and release (and get information about the block). The `NetControl::unblock_address_catch_release` function can be used to unblock addresses from catch and release.

**Note:** Since catch and release does its own connection tracking in addition to the tracking used by the NetControl framework, it is not sufficient to remove rules that were added by catch and release using `NetControl::remove_rule`. You have to use `NetControl::unblock_address_catch_release` in this case.

### NetControl Plugins

#### Using the existing plugins

In the API part of the documentation, we exclusively used the debug plugin, which simply outputs its actions to the screen. In addition to this debugging plugin, Bro ships with a small number of plugins that can be used to interface the NetControl framework with your networking hard- and software.

The plugins that currently ship with NetControl are:

| Plugin name | Description |
|---|---|
| OpenFlow plugin | This is the most fully featured plugin which allows the NetControl framework to be interfaced with OpenFlow switches. The source of this plugin is contained in /scripts/base/frameworks/netcontrol/plugins/openflow.bro. |
| Broker plugin | This plugin provides a generic way to send NetControl commands using the new Bro communication library (Broker). External programs can receive the rules and take action; we provide an example script that calls command-line programs triggered by NetControl. The source of this plugin is contained in /scripts/base/frameworks/netcontrol/plugins/broker.bro. |
| acld plugin | This plugin adds support for the acld daemon, which can interface with several switches and routers. The current version of acld is available from the LBL ftp server. The source of this plugin is contained in /scripts/base/frameworks/netcontrol/plugins/acld.bro. |
| PacketFilter plugin | This plugin adds uses the Bro process-level packet filter (see `install_src_net_filter` and `install_dst_net_filter`). Since the functionality of the PacketFilter is limited, this plugin is mostly for demonstration purposes. The source of this plugin is contained in /scripts/base/frameworks/netcontrol/plugins/packetfilter.bro. |
| Debug plugin | The debug plugin simply outputs its action to the standard output. The source of this plugin is contained in /scripts/base/frameworks/netcontrol/plugins/debug.bro. |

### Activating plugins

In the API reference part of this document, we already used the debug plugin. To use the plugin, we first had to instantiate it by calling `NetControl::NetControl::create_debug` and then add it to NetControl by calling `NetControl::activate`.

As we already hinted before, NetControl supports having several plugins that are active at the same time. The second argument to the *NetControl::activate* function is the priority of the backend that was just added. Each rule is sent to all plugins in order, from highest priority to lowest priority. The backend can then choose if it accepts the rule and pushes it out to the hardware that it manages. Or, it can opt to reject the rule. In this case, the NetControl framework will try to apply the rule to the backend with the next lower priority. If no backend accepts a rule, the rule insertion is marked as failed.

The choice if a rule is accepted or rejected stays completely with each plugin. The debug plugin we used so far just accepts all rules. However, for other plugins you can specify what rules they will accept. Consider, for example, a network with two OpenFlow switches. The first switch forwards packets from the network to the external world, the second switch sits in front of your Bro cluster to provide packet shunting. In this case, you can add two OpenFlow backends to NetControl. When you create the instances using `NetControl::create_openflow`, you set the *monitor* and *forward* attributes of the configuration in `NetControl::OfConfig` appropriately. Afterwards, one of the backends will only accept rules for the monitor path; the other backend will only accept rules for the forward path.

Commonly, plugins also support predicate functions, that allow the user to specify restrictions on the rules that they will accept. This can for example be used if you have a network where certain switches are responsible for specified subnets. The predicate can examine the subnet of the rule and only accept the rule if the rule matches the subnet that the specific switch is responsible for.

To give an example, the following script adds two backends to NetControl. One backend is the NetControl debug backend, which just outputs the rules to the console. The second backend is an OpenFlow backend, which uses the OpenFlow debug mode that outputs the openflow rules to openflow.log. The OpenFlow backend uses a predicate function to only accept rules with a source address in the 192.168.17.0/24 network; all other rules will be passed on to the debug plugin. We manually block a few addresses in the `NetControl::init_done` event to verify the correct functionality.

```
ERROR executing test 'doc.sphinx.netcontrol-8-multiple.bro' (part 1)

% 'btest-rst-cmd bro ${DOC_ROOT}/frameworks/netcontrol-8-multiple.bro' failed␣
↪unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-8-multiple.bro
```

As you can see, only the single block affecting the 192.168.17.0/24 network is output to the command line. The other two lines are handled by the OpenFlow plugin. We can verify this by looking at netcontrol.log. The plugin column shows which plugin handled a rule and reveals that two rules were handled by OpenFlow:

```
ERROR executing test 'doc.sphinx.netcontrol-8-multiple.bro' (part 2)

% 'btest-rst-cmd cat netcontrol.log' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-8-multiple.bro
 cat: netcontrol.log: No such file or directory
```

Furthermore, openflow.log also shows the two added rules, converted to OpenFlow flow mods:

```
ERROR executing test 'doc.sphinx.netcontrol-8-multiple.bro' (part 3)

% 'btest-rst-cmd cat openflow.log' failed unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-8-multiple.bro
 cat: netcontrol.log: No such file or directory
 cat: openflow.log: No such file or directory
```

**Note:** You might have asked yourself what happens when you add two or more with the same priority. In this case, the rule is sent to all the backends simultaneously. This can be useful, for example when you have redundant switches that should keep the same rule state.

### Interfacing with external hardware

Now that we know which plugins exist, and how they can be added to NetControl, it is time to discuss how we can interface Bro with actual hardware. The typical way to accomplish this is to use the Bro communication library (Broker), which can be used to exchange Bro events with external programs and scripts. The NetControl plugins can use Broker to send events to external programs, which can then take action depending on these events.

The following figure shows this architecture with the example of the OpenFlow plugin. The OpenFlow plugin uses Broker to send events to an external Python script, which uses the Ryu SDN controller to communicate with the Switch.

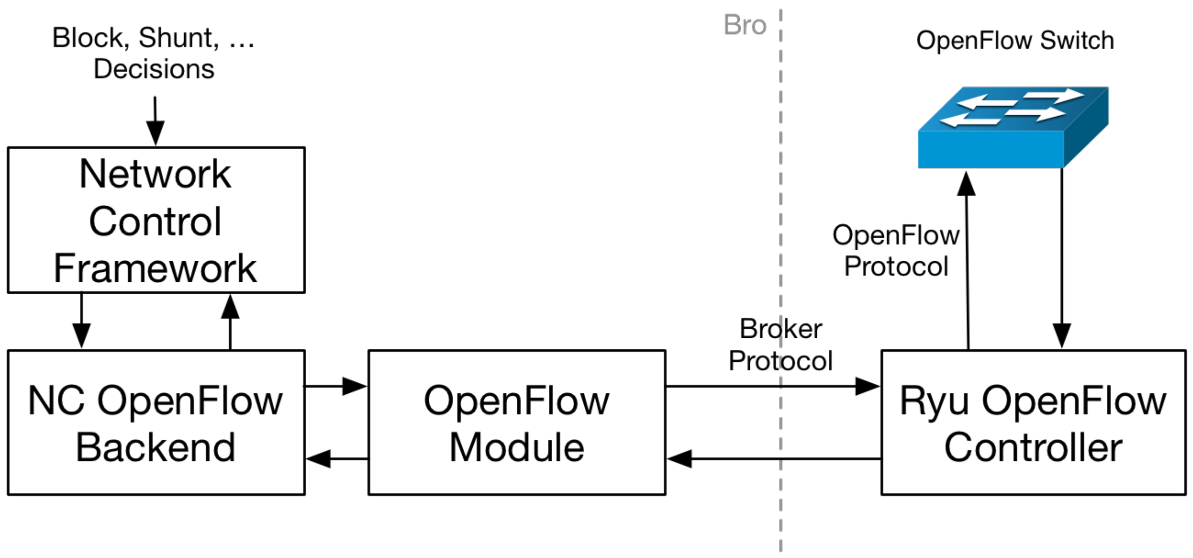


Fig. 3.3: NetControl and OpenFlow architecture (click to enlarge).

The Python scripts that are used to interface with the available NetControl plugins are contained in the *bro-netcontrol* repository (github link). The repository contains scripts for the OpenFlow as well as the acld plugin. Furthermore, it contains a script for the broker plugin which can be used to call configureable command-line programs when used with the broker plugin.

The repository also contains documentation on how to install these connectors. The *netcontrol* directory contains an API that allows you to write your own connectors to the broker plugin.

**Note:** Note that the API of the Broker communication library is not finalized yet. You might have to rewrite any scripts for use in future Bro versions.

#### Writing plugins

In addition to using the plugins that are part of NetControl, you can write your own plugins to interface with hard- or software that we currently do not support out of the Box.

Creating your own plugin is easy; besides a bit of boilerplate, you only need to create two functions: one that is called when a rule is added, and one that is called when a rule is removed. The following script creates a minimal plugin that just outputs a rule when it is added or removed. Note that you have to raise the `NetControl::rule_added` and `NetControl::rule_removed` events in your plugin to let NetControl know when a rule was added and removed successfully.

This example is already fully functional and we can use it with a script similar to our very first example:

```
ERROR executing test 'doc.sphinx.netcontrol-9-skeleton.bro' (part 1)

% 'btest-rst-cmd bro -C -r ${TRACES}/tls/ecdhe.pcap ${DOC_ROOT}/frameworks/netcontrol-
↪9-skeleton.bro ${DOC_ROOT}/frameworks/netcontrol-10-use-skeleton.bro' failed␣
↪unexpectedly (exit code 1)
% cat .stderr
 fatal error: can't find /root/src/bro-2.4.1/testing/btest/../../doc/frameworks/
↪netcontrol-9-skeleton.bro
```

If you want to write your own plugins, it will be worthwhile to look at the plugins that ship with the NetControl framework to see how they define the predicates and interact with Broker.

### 3.1.7 Notice Framework

One of the easiest ways to customize Bro is writing a local notice policy. Bro can detect a large number of potentially interesting situations, and the notice policy hook which of them the user wants to be acted upon in some manner. In particular, the notice policy can specify actions to be taken, such as sending an email or compiling regular alarm emails. This page gives an introduction into writing such a notice policy.

**Contents**

### Overview

Let's start with a little bit of background on Bro's philosophy on reporting things. Bro ships with a large number of policy scripts which perform a wide variety of analyses. Most of these scripts monitor for activity which might be of interest for the user. However, none of these scripts determines the importance of what it finds itself. Instead, the scripts only flag situations as *potentially* interesting, leaving it to the local configuration to define which of them are in fact actionable. This decoupling of detection and reporting allows Bro to address the different needs that sites have. Definitions of what constitutes an attack or even a compromise differ quite a bit between environments, and activity deemed malicious at one site might be fully acceptable at another.

Whenever one of Bro's analysis scripts sees something potentially interesting it flags the situation by calling the `NOTICE` function and giving it a single `Notice::Info` record. A Notice has a `Notice::Type`, which reflects the kind of activity that has been seen, and it is usually also augmented with further context about the situation.

More information about raising notices can be found in the *Raising Notices* section.

Once a notice is raised, it can have any number of actions applied to it by writing `Notice::policy` hooks which is described in the *Notice Policy* section below. Such actions can be to send a mail to the configured address(es) or to simply ignore the notice. Currently, the following actions are defined:

| Action | Description |
|---|---|
| Notice::ACTION_LOG | Write the notice to the `Notice::LOG` logging stream. |
| Notice::ACTION_ALARM | Log into the `Notice::ALARM_LOG` stream which will rotate hourly and email the contents to the email address or addresses defined in the `Notice::mail_dest` variable. |
| Notice::ACTION_EMAIL | Send the notice in an email to the email address or addresses given in the `Notice::mail_dest` variable. |
| Notice::ACTION_PAGE | Send an email to the email address or addresses given in the `Notice::mail_page_dest` variable. |

How these notice actions are applied to notices is discussed in the *Notice Policy* and *Notice Policy Shortcuts* sections.

### Processing Notices

### Notice Policy

The hook `Notice::policy` provides the mechanism for applying actions and generally modifying the notice before it's sent onward to the action plugins. Hooks can be thought of as multi-bodied functions and using them looks very similar to handling events. The difference is that they don't go through the event queue like events. Users can alter notice processing by directly modifying fields in the `Notice::Info` record given as the argument to the hook.

Here's a simple example which tells Bro to send an email for all notices of type `SSH::Password_Guessing` if the guesser attempted to log in to the server at 192.168.56.103:

```
notice_ssh_guesser.bro


@load protocols/ssh/detect-bruteforcing

redef SSH::password_guesses_limit=10;

```

```
hook Notice::policy(n: Notice::Info)
        {
        if ( n$note == SSH::Password_Guessing && /192\.168\.56\.103/ in n$sub )
                add n$actions[Notice::ACTION_EMAIL];
        }
```

```
# bro -C -r ssh/sshguess.pcap notice_ssh_guesser.bro
```

```
# cat notice.log
#separator \x09
#set_separator    ,
#empty_field      (empty)
#unset_field      -
#path     notice
#open     2016-10-01-03-42-05
#fields   ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p  ␣
↪     fuid    file_mime_type  file_desc       proto   note    msg     sub     src    ␣
↪dst      p       n       peer_descr      actions suppress_for    dropped remote_
↪location.country_code    remote_location.region  remote_location.city    remote_
↪location.latitude        remote_location.longitude
#types    time    string  addr    port    addr    port    string  string  string ␣
↪enum     enum    string  string  addr    addr    port    count   string  set[enum]  ␣
↪     interval        bool    string  string  string  double  double
1427726711.398575 -       -       -       -       -       -       -       -       -  ␣
↪     SSH::Password_Guessing  192.168.56.1 appears to be guessing SSH passwords (seen␣
↪in 10 connections).      Sampled servers:  192.168.56.103, 192.168.56.103, 192.168.
↪56.103, 192.168.56.103, 192.168.56.103        192.168.56.1    -       -       -  ␣
↪  bro     Notice::ACTION_EMAIL,Notice::ACTION_LOG 3600.000000     F       -       -␣
↪      -       -       -
#close    2016-10-01-03-42-05
```

**Note:** Keep in mind that the semantics of the `SSH::Password_Guessing` notice are such that it is only raised when Bro heuristically detects a failed login.

Hooks can also have priorities applied to order their execution like events with a default priority of 0. Greater values are executed first. Setting a hook body to run before default hook bodies might look like this:

Hooks can also abort later hook bodies with the `break` keyword. This is primarily useful if one wants to completely preempt processing by lower priority `Notice::policy` hooks.

### Notice Policy Shortcuts

Although the notice framework provides a great deal of flexibility and configurability there are many times that the full expressiveness isn't needed and actually becomes a hindrance to achieving results. The framework provides a default `Notice::policy` hook body as a way of giving users the shortcuts to easily apply many common actions to notices.

These are implemented as sets and tables indexed with a `Notice::Type` enum value. The following table shows and describes all of the variables available for shortcut configuration of the notice framework.

| Variable name | Description |
|---|---|
| `Notice::ignored_types` | Adding a `Notice::Type` to this set results in the notice being ignored. It won't have any other action applied to it, not even `Notice::ACTION_LOG`. |
| `Notice::emailed_types` | Adding a `Notice::Type` to this set results in `Notice::ACTION_EMAIL` being applied to the notices of that type. |
| `Notice::alarmed_types` | Adding a `Notice::Type` to this set results in `Notice::ACTION_ALARM` being applied to the notices of that type. |
| `Notice::not_suppressed_types` | Adding a `Notice::Type` to this set results in that notice no longer undergoing the normal notice suppression that would take place. Be careful when using this in production it could result in a dramatic increase in the number of notices being processed. |
| `Notice::type_suppression_intervals` | This is a table indexed on `Notice::Type` and yielding an interval. It can be used as an easy way to extend the default suppression interval for an entire `Notice::Type` without having to create a whole `Notice::policy` entry and setting the `$suppress_for` field. |

## Raising Notices

A script should raise a notice for any occurrence that a user may want to be notified about or take action on. For example, whenever the base SSH analysis scripts sees enough failed logins to a given host, it raises a notice of the type `SSH::Password_Guessing`. The code in the base SSH analysis script which raises the notice looks like this:

`NOTICE` is a normal function in the global namespace which wraps a function within the `Notice` namespace. It takes a single argument of the `Notice::Info` record type. The most common fields used when raising notices are described in the following table:

| Field name | Description |
|---|---|
| `$note` | This field is required and is an enum value which represents the notice type. |
| `$msg` | This is a human readable message which is meant to provide more information about this particular instance of the notice type. |
| `$sub` | This is a sub-message meant for human readability but will frequently also be used to contain data meant to be matched with the `Notice::policy`. |
| `$conn` | If a connection record is available when the notice is being raised and the notice represents some attribute of the connection, then the connection record can be given here. Other fields such as `$id` and `$src` will automatically be populated from this value. |
| `$id` | If a conn_id record is available when the notice is being raised and the notice represents some attribute of the connection, then the connection can be given here. Other fields such as `$src` will automatically be populated from this value. |
| `$src` | If the notice represents an attribute of a single host then it's possible that only this field should be filled out to represent the host that is being "noticed". |
| `$n` | This normally represents a number if the notice has to do with some number. It's most frequently used for numeric tests in the `Notice::policy` for making policy decisions. |
| `$identifier` | This represents a unique identifier for this notice. This field is described in more detail in the *Automated Suppression* section. |
| `$suppress_for` | This field can be set if there is a natural suppression interval for the notice that may be different than the default value. The value set to this field can also be modified by a user's `Notice::policy` so the value is not set permanently and unchangeably. |

When writing Bro scripts which raise notices, some thought should be given to what the notice represents and what data should be provided to give a consumer of the notice the best information about the notice. If the notice is representative of many connections and is an attribute of a host (e.g. a scanning host) it probably makes most sense to fill out the `$src` field and not give a connection or conn_id. If a notice is representative of a connection attribute (e.g. an apparent SSH login) then it makes sense to fill out either `$conn` or `$id` based on the data that is available when the notice is raised. Using care when inserting data into a notice will make later analysis easier when only the data to fully represent the occurrence that raised the notice is available. If complete connection information is available when an SSL server certificate is expiring, the logs will be very confusing because the connection that the certificate was detected on is a side topic to the fact that an expired certificate was detected. It's possible in many cases that two or more separate notices may need to be generated. As an example, one could be for the detection of the expired SSL certificate and another could be for if the client decided to go ahead with the connection neglecting the expired certificate.

### Automated Suppression

The notice framework supports suppression for notices if the author of the script that is generating the notice has indicated to the notice framework how to identify notices that are intrinsically the same. Identification of these "intrinsically duplicate" notices is implemented with an optional field in `Notice::Info` records named `$identifier` which is a simple string. If the `$identifier` and `$note` fields are the same for two notices, the notice framework actually considers them to be the same thing and can use that information to suppress duplicates for a configurable period of time.

**Note:** If the `$identifier` is left out of a notice, no notice suppression takes place due to the framework's inability to identify duplicates. This could be completely legitimate usage if no notices could ever be considered to be duplicates.

The `$identifier` field is typically comprised of several pieces of data related to the notice that when combined represent a unique instance of that notice. Here is an example of the script /scripts/policy/protocols/ssl/validate-certs.bro raising a notice for session negotiations where the certificate or certificate chain did not validate successfully against the available certificate authority certificates.

In the above example you can see that the `$identifier` field contains a string that is built from the responder IP address and port, the validation status message, and the MD5 sum of the server certificate. Those fields in particular are chosen because different SSL certificates could be seen on any port of a host, certificates could fail validation for different reasons, and multiple server certificates could be used on that combination of IP address and port with the `server_name` SSL extension (explaining the addition of the MD5 sum of the certificate). The result is that if a certificate fails validation and all four pieces of data match (IP address, port, validation status, and certificate hash) that particular notice won't be raised again for the default suppression period.

Setting the `$identifier` field is left to those raising notices because it's assumed that the script author who is raising the notice understands the full problem set and edge cases of the notice which may not be readily apparent to users. If users don't want the suppression to take place or simply want a different interval, they can set a notice's suppression interval to `0secs` or delete the value from the `$identifier` field in a `Notice::policy` hook.

### Extending Notice Framework

There are a couple of mechanism currently for extending the notice framework and adding new capability.

### Extending Notice Emails

If there is extra information that you would like to add to emails, that is possible to add by writing `Notice::policy` hooks.

There is a field in the `Notice::Info` record named `$email_body_sections` which will be included verbatim when email is being sent. An example of including some information from an HTTP request is included below.

### Cluster Considerations

As a user/developer of Bro, the main cluster concern with the notice framework is understanding what runs where. When a notice is generated on a worker, the worker checks to see if the notice should be suppressed based on information locally maintained in the worker process. If it's not being suppressed, the worker forwards the notice directly to the manager and does no more local processing. The manager then runs the `Notice::policy` hook and executes all of the actions determined to be run.

## 3.1.8 Signature Framework

Bro relies primarily on its extensive scripting language for defining and analyzing detection policies. In addition, however, Bro also provides an independent *signature language* for doing low-level, Snort-style pattern matching. While signatures are *not* Bro's preferred detection tool, they sometimes come in handy and are closer to what many people are familiar with from using other NIDS. This page gives a brief overview on Bro's signatures and covers some of their technical subtleties.

**Contents**

### Basics

Let's look at an example signature first:

This signature asks Bro to match the regular expression `.*root` on all TCP connections going to port 80. When the signature triggers, Bro will raise an event `signature_match` of the form:

Here, `state` contains more information on the connection that triggered the match, `msg` is the string specified by the signature's event statement (`Found root!`), and data is the last piece of payload which triggered the pattern match.

To turn such `signature_match` events into actual alarms, you can load Bro's /scripts/base/frameworks/signatures/main.bro script. This script contains a default event handler that raises `Signatures::Sensitive_Signature` *Notices* (as well as others; see the beginning of the script).

As signatures are independent of Bro's policy scripts, they are put into their own file(s). There are three ways to specify which files contain signatures: By using the `-s` flag when you invoke Bro, or by extending the Bro variable `signature_files` using the += operator, or by using the `@load-sigs` directive inside a Bro script. If a signature file is given without a full path, it is searched for along the normal `BROPATH`. Additionally, the `@load-sigs` directive can be used to load signature files in a path relative to the Bro script in which it's placed, e.g. `@load-sigs ./mysigs.sig` will expect that signature file in the same directory as the Bro script. The default extension of the file name is `.sig`, and Bro appends that automatically when necessary.

### Signature Language for Network Traffic

Let's look at the format of a signature more closely. Each individual signature has the format `signature <id> { <attributes> }`. `<id>` is a 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 discuss these all in more detail in the following.

### Conditions

### Header Conditions

Header conditions limit the applicability of the signature to a subset of traffic that contains matching packet headers. This type of matching is performed only for the first packet of a connection.

There are pre-defined header conditions for some of the most used header fields. All of them generally have the format `<keyword> <cmp> <value-list>`, where `<keyword>` names the header field; `cmp` is one of `==`, `!=`, `<`, `<=`, `>`, `>=`; and `<value-list>` is a list of comma-separated values to compare against. The following keywords are defined:

**`src-ip/dst-ip <cmp> <address-list>`** Source and destination address, respectively. Addresses can be given as IPv4 or IPv6 addresses or CIDR masks. For IPv6 addresses/masks the colon-hexadecimal representation of the address must be enclosed in square brackets (e.g. `[fe80::1]` or `[fe80::0]/16`).

**`src-port/dst-port <cmp> <int-list>`** Source and destination port, respectively.

**`ip-proto <cmp> tcp|udp|icmp|icmp6|ip|ip6`** IPv4 header's Protocol field or the Next Header field of the final IPv6 header (i.e. either Next Header field in the fixed IPv6 header if no extension headers are present or that field from the last extension header in the chain). Note that the IP-in-IP forms of tunneling are automatically decapsulated by default and signatures apply to only the inner-most packet, so specifying `ip` or `ip6` is a no-op.

For lists of multiple values, they are sequentially compared against the corresponding header field. If at least one of the comparisons evaluates to true, the whole header condition matches (exception: with `!=`, the header condition only matches if all values differ).

In addition to these pre-defined header keywords, a general header condition can be defined either as

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 be `ip`, `ip6`, `tcp`, `udp`, `icmp` or `icmp6`). `size` is either 1, 2, or 4 and specifies the value to have a size of this many bytes. If the optional `& <integer>` is given, the packet's value is first masked with the integer before it is compared to the value-list. `cmp` 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 corresponds to the CIDR notation for netmasks and is translated into a corresponding bitmask applied to the packet's value prior to the comparison (similar to the optional `& integer`). IPv6 address values are not allowed in the value-list, though you can still inspect any 1, 2, or 4 byte section of an IPv6 header using this keyword.

Putting it all together, this is an example condition that is equivalent to `dst-ip == 1.2.3.4/16,5.6.7.8/24`:

Note that the analogous example for IPv6 isn't currently possible since 4 bytes is the max width of a value that can be compared.

### 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 each packet (for ICMP, UDP, and non-reassembled TCP). Second, 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:

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 HTTP requests. Obsolete alias: `http`.

**`http-request-header /<regular expression>/`** The regular expression is matched against client-side HTTP headers.

**`http-request-body /<regular expression>/`** The regular expression is matched against client-side bodys of HTTP requests.

**`http-reply-header /<regular expression>/`** The regular expression is matched against server-side HTTP headers.

**`http-reply-body /<regular expression>/`** The regular expression is matched against server-side bodys of HTTP replys.

**`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 finger requests.

For example, `http-request /.*(etc/(passwd|shadow)/` matches any URI containing either `etc/passwd` or `etc/shadow`. To filter on request types, e.g. `GET`, use `payload /GET /`.

Note that HTTP pipelining (that is, multiple HTTP transactions in a single TCP connection) has some side effects on signature matches. If multiple conditions are specified within a single signature, this signature matches if all conditions are met by any HTTP transaction (not necessarily always the same!) in a pipelined connection.

### Dependency Conditions

To define dependencies between 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 (using this defers the match decision until the connection terminates).

**`requires-reverse-signature [!] <id>`** Similar to `requires-signature`, but `id` has to match for the opposite direction of the same connection, compared to the current signature. This allows to model the notion of requests and replies.

### Context Conditions

Context conditions pass the match decision on to other components of Bro. They are only evaluated if all other conditions have already matched. The following context conditions are defined:

**`eval <policy-function>`** The given policy function is called and has to return a boolean confirming the match. If false is returned, no signature match is going to be triggered. The function has to be of type `function cond(state: signature_state, data: string): bool`. Here, `data` may contain the most recent content chunk available at the time the signature was matched. If no such chunk is available, `data` will be the empty string. See `signature_state` for its definition.

**payload-size <cmp> <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 very well defined.

**same-ip** Evaluates to true if the source address of the IP packets equals its destination address.

**tcp-state <state-list>** Imposes restrictions on the current TCP state of the connection. `state-list` is a comma-separated list of the keywords `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).

### Actions

Actions define what to do if a signature matches. Currently, there are two actions defined:

**event <string>** Raises a `signature_match` event. The event handler has the following type:

The given string is passed in 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).

**enable <string>** Enables the protocol analyzer `<string>` for the matching connection (`"http"`, `"ftp"`, etc.). This is used by Bro's dynamic protocol detection to activate analyzers on the fly.

## Signature Language for File Content

The signature framework can also be used to identify MIME types of files irrespective of the network protocol/connection over which the file is transferred. A special type of signature can be written for this purpose and will be used automatically by the *Files Framework* or by Bro scripts that use the `file_magic` built-in function.

### Conditions

File signatures use a single type of content condition in the form of a regular expression:

`file-magic /<regular expression>/`

This is analogous to the `payload` content condition for the network traffic signature language described above. The difference is that `payload` signatures are applied to payloads of network connections, but `file-magic` can be applied to any arbitrary data, it does not have to be tied to a network protocol/connection.

### Actions

Upon matching a chunk of data, file signatures use the following action to get information about that data's MIME type:

`file-mime <string> [,<integer>]`

The arguments include the MIME type string associated with the file magic regular expression and an optional "strength" as a signed integer. Since multiple file magic signatures may match against a given chunk of data, the strength value may be used to help choose a "winner". Higher values are considered stronger.

## Things to keep in mind when writing signatures

- Each signature is reported at most once for every connection, further matches of the same signature are ignored.

- The content conditions perform pattern matching on elements extracted from an application protocol dialogue. For example, `http /.*passwd/` scans URLs requested within HTTP sessions. The thing to keep in mind here is that these conditions only perform any matching when the corresponding application analyzer is actually *active* for a connection. Note that by default, analyzers are not enabled if the corresponding Bro script has not been loaded. A good way to double-check whether an analyzer "sees" a connection is checking its log file for corresponding entries. If you cannot find the connection in the analyzer's log, very likely the signature engine has also not seen any application data.
- As the name indicates, the `payload` keyword matches on packet *payload* only. You cannot use it to match on packet headers; use the header conditions for that.
- For TCP connections, header conditions are only evaluated for the *first packet from each endpoint*. If a header condition does not match the initial packets, the signature will not trigger. Bro optimizes for the most common application here, which is header conditions selecting the connections to be examined more closely with payload statements.
- For UDP and ICMP flows, the payload matching is done on a per-packet basis; i.e., any content crossing packet boundaries will not be found. For TCP connections, the matching semantics depend on whether Bro is *reassembling* the connection (i.e., putting all of a connection's packets in sequence). By default, Bro is reassembling the first 1K of every TCP connection, which means that within this window, matches will be found without regards to packet order or boundaries (i.e., *stream-wise matching*).
- For performance reasons, by default Bro *stops matching* on a connection after seeing 1K of payload; see the section on options below for how to change this behaviour. The default was chosen with Bro's main user of signatures in mind: dynamic protocol detection works well even when examining just connection heads.
- Regular expressions are implicitly anchored, i.e., they work as if prefixed with the ^ operator. For reassembled TCP connections, they are anchored at the first byte of the payload *stream*. For all other connections, they are anchored at the first payload byte of each packet. To match at arbitrary positions, you can prefix the regular expression with `.*`, as done in the examples above.
- To match on non-ASCII characters, Bro's regular expressions support the `\x<hex>` operator. CRs/LFs are not treated specially by the signature engine and can be matched with `\r` and `\n`, respectively. Generally, Bro follows flex's regular expression syntax. See the DPD signatures in `base/frameworks/dpd/dpd.sig` for some examples of fairly complex payload patterns.
- The data argument of the `signature_match` handler might not carry the full text matched by the regular expression. Bro performs the matching incrementally as packets come in; when the signature eventually fires, it can only pass on the most recent chunk of data.

## Options

The following options control details of Bro's matching process:

**`dpd_reassemble_first_packets`: `bool` (default: `T`)** If true, Bro reassembles the beginning of every TCP connection (of up to `dpd_buffer_size` bytes, see below), to facilitate reliable matching across packet boundaries. If false, only connections are reassembled for which an application-layer analyzer gets activated (e.g., by Bro's dynamic protocol detection).

**`dpd_match_only_beginning` : `bool` (default: `T`)** If true, Bro performs packet matching only within the initial payload window of `dpd_buffer_size`. If false, it keeps matching on subsequent payload as well.

**`dpd_buffer_size`: `count` (default: `1024`)** Defines the buffer size for the two preceding options. In addition, this value determines the amount of bytes Bro buffers for each connection in order to activate application analyzers even after parts of the payload have already passed through. This is needed by the dynamic protocol detection capability to defer the decision which analyzers to use.

#### So, how about using Snort signatures with Bro?

There was once a script, `snort2bro`, that converted Snort signatures automatically into Bro's signature syntax. However, in our experience this didn't turn out to be a very useful thing to do because by simply using Snort signatures, one can't benefit from the additional capabilities that Bro provides; the approaches of the two systems are just too different. We therefore stopped maintaining the `snort2bro` script, and there are now many newer Snort options which it doesn't support. The script is now no longer part of the Bro distribution.

## 3.1.9 Summary Statistics

Measuring aspects of network traffic is an extremely common task in Bro. Bro provides data structures which make this very easy as well in simplistic cases such as size limited trace file processing. In real-world deployments though, there are difficulties that arise from clusterization (many processes sniffing traffic) and unbounded data sets (traffic never stops). The Summary Statistics (otherwise referred to as SumStats) framework aims to define a mechanism for consuming unbounded data sets and making them measurable in practice on large clustered and non-clustered Bro deployments.

**Contents**

### Overview

The Sumstat processing flow is broken into three pieces. Observations, where some aspect of an event is observed and fed into the Sumstats framework. Reducers, where observations are collected and measured, typically by taking some sort of summary statistic measurement like average or variance (among others). Sumstats, where reducers have an epoch (time interval) that their measurements are performed over along with callbacks for monitoring thresholds or viewing the collected and measured data.

### Terminology

Observation

A single point of data. Observations have a few components of their own. They are part of an arbitrarily named observation stream, they have a key that is something the observation is about, and the actual observation itself.

Reducer

Calculations are applied to an observation stream here to reduce the full unbounded set of observations down to a smaller representation. Results are collected within each reducer per-key so care must be taken to keep the total number of keys tracked down to a reasonable level.

Sumstat

The final definition of a Sumstat where one or more reducers is collected over an interval, also known as an epoch. Thresholding can be applied here along with a callback in the event that a threshold is crossed. Additionally, a callback can be provided to access each result (per-key) at the end of each epoch.

### Examples

These examples may seem very simple to an experienced Bro script developer and they're intended to look that way. Keep in mind that these scripts will work on small single process Bro instances as well as large many-worker clusters. The complications from dealing with flow based load balancing can be ignored by developers writing scripts that use Sumstats due to its built-in cluster transparency.

#### Printing the number of connections

Sumstats provides a simple way of approaching the problem of trying to count the number of connections over a given time interval. Here is a script with inline documentation that does this with the Sumstats framework:

```
sumstats-countconns.bro

@load base/frameworks/sumstats

event connection_established(c: connection)
        {
        # Make an observation!
        # This observation is global so the key is empty.
        # Each established connection counts as one so the observation is always 1.
        SumStats::observe("conn established",
                          SumStats::Key(),
                          SumStats::Observation($num=1));
        }

event bro_init()
        {
        # Create the reducer.
        # The reducer attaches to the "conn established" observation stream
        # and uses the summing calculation on the observations.
        local r1 = SumStats::Reducer($stream="conn established",
                                     $apply=set(SumStats::SUM));

        # Create the final sumstat.
        # We give it an arbitrary name and make it collect data every minute.
        # The reducer is then attached and a $epoch_result callback is given
        # to finally do something with the data collected.
        SumStats::create([$name = "counting connections",
                          $epoch = 1min,
                          $reducers = set(r1),
                          $epoch_result(ts: time, key: SumStats::Key, result:␣
↪SumStats::Result) =
                                {
                                # This is the body of the callback that is called␣
↪when a single
                                # result has been collected.  We are just printing␣
↪the total number
                                # of connections that were seen.  The $sum field is␣
↪provided as a
                                # double type value so we need to use %f as the␣
↪format specifier.
```

```
                                    print fmt("Number of connections established: %.0f
↪", result["conn established"]$sum);
                                    }]);
        }
```

When run on a sample PCAP file from the Bro test suite, the following output is created:

```
# bro -r workshop_2011_browse.trace sumstats-countconns.bro
Number of connections established: 6
```

### Toy scan detection

Taking the previous example even further, we can implement a simple detection to demonstrate the thresholding functionality. This example is a toy to demonstrate how thresholding works in Sumstats and is not meant to be a real-world functional example, that is left to the /scripts/policy/misc/scan.bro script that is included with Bro.

```
sumstats-toy-scan.bro

@load base/frameworks/sumstats

# We use the connection_attempt event to limit our observations to those
# which were attempted and not successful.
event connection_attempt(c: connection)
        {
        # Make an observation!
        # This observation is about the host attempting the connection.
        # Each established connection counts as one so the observation is always 1.
        SumStats::observe("conn attempted",
                          SumStats::Key($host=c$id$orig_h),
                          SumStats::Observation($num=1));
        }

event bro_init()
        {
        # Create the reducer.
        # The reducer attaches to the "conn attempted" observation stream
        # and uses the summing calculation on the observations. Keep
        # in mind that there will be one result per key (connection originator).
        local r1 = SumStats::Reducer($stream="conn attempted",
                                     $apply=set(SumStats::SUM));

        # Create the final sumstat.
        # This is slightly different from the last example since we're providing
        # a callback to calculate a value to check against the threshold with
        # $threshold_val.  The actual threshold itself is provided with $threshold.
        # Another callback is provided for when a key crosses the threshold.
        SumStats::create([$name = "finding scanners",
                          $epoch = 5min,
                          $reducers = set(r1),
                          # Provide a threshold.
                          $threshold = 5.0,
                          # Provide a callback to calculate a value from the result
                          # to check against the threshold field.
                          $threshold_val(key: SumStats::Key, result: SumStats::
↪Result) =
                                    {
                                        return result["conn attempted"]$sum;
                                        },
                                # Provide a callback for when a key crosses the threshold.
                                $threshold_crossed(key: SumStats::Key, result: SumStats::
↪Result) =
                                        {
                                        print fmt("%s attempted %.0f or more connections",␣
↪key$host, result["conn attempted"]$sum);
                                        }]);
        }
```

Let's see if there are any hosts that crossed the threshold in a PCAP file containing a host running nmap:

```
# bro -r nmap-vsn.trace sumstats-toy-scan.bro
192.168.1.71 attempted 5 or more connections
```

It seems the host running nmap was detected!

## 3.1.10 Broker-Enabled Communication Framework

Bro can now use the Broker Library to exchange information with other Bro processes.

**Contents**

### Connecting to Peers

Communication via Broker must first be turned on via `Broker::enable`.

Bro can accept incoming connections by calling `Broker::listen` and then monitor connection status updates via the `Broker::incoming_connection_established` and `Broker::incoming_connection_broken` events.

```
connecting-listener.bro


const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "listener";

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::listen(broker_port, "127.0.0.1");
        }

event BrokerComm::incoming_connection_established(peer_name: string)
        {
        print "BrokerComm::incoming_connection_established", peer_name;
        }

event BrokerComm::incoming_connection_broken(peer_name: string)
        {
        print "BrokerComm::incoming_connection_broken", peer_name;
        terminate();
        }
```

Bro can initiate outgoing connections by calling `Broker::connect` and then monitor connection status updates via the `Broker::outgoing_connection_established`, `Broker::outgoing_connection_broken`, and `Broker::outgoing_connection_incompatible` events.

```
connecting-connector.bro


const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "connector";

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::connect("127.0.0.1", broker_port, 1sec);
        }

event BrokerComm::outgoing_connection_established(peer_address: string,
                                                   peer_port: port,
                                                   peer_name: string)
        {
        print "BrokerComm::outgoing_connection_established",
                  peer_address, peer_port, peer_name;
        terminate();
        }
```

## Remote Printing

To receive remote print messages, first use the `Broker::subscribe_to_prints` function to advertise to peers a topic prefix of interest and then create an event handler for `Broker::print_handler` to handle any print messages that are received.

```
printing-listener.bro


const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "listener";
global msg_count = 0;

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::subscribe_to_prints("bro/print/");
        BrokerComm::listen(broker_port, "127.0.0.1");
        }

event BrokerComm::incoming_connection_established(peer_name: string)
        {
        print "BrokerComm::incoming_connection_established", peer_name;
        }

event BrokerComm::print_handler(msg: string)
        {
        ++msg_count;
        print "got print message", msg;

        if ( msg_count == 3 )
                terminate();
        }
```

To send remote print messages, just call `Broker::send_print`.

```
printing-connector.bro

const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "connector";

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::connect("127.0.0.1", broker_port, 1sec);
        }

event BrokerComm::outgoing_connection_established(peer_address: string,
                                                  peer_port: port,
                                                  peer_name: string)
        {
        print "BrokerComm::outgoing_connection_established",
              peer_address, peer_port, peer_name;
        BrokerComm::print("bro/print/hi", "hello");
        BrokerComm::print("bro/print/stuff", "...");
        BrokerComm::print("bro/print/bye", "goodbye");
        }

event BrokerComm::outgoing_connection_broken(peer_address: string,
                                             peer_port: port)
        {
        terminate();
        }
```

Notice that the subscriber only used the prefix “bro/print/”, but is able to receive messages with full topics of “bro/print/hi”, “bro/print/stuff”, and “bro/print/bye”. The model here is that the publisher of a message checks for all subscribers who advertised interest in a prefix of that message’s topic and sends it to them.

### Message Format

For other applications that want to exchange print messages with Bro, the Broker message format is simply:

### Remote Events

Receiving remote events is similar to remote prints. Just use the `Broker::subscribe_to_events` function and possibly define any new events along with handlers that peers may want to send.

```
events-listener.bro


const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "listener";
global msg_count = 0;
global my_event: event(msg: string, c: count);
global my_auto_event: event(msg: string, c: count);

event bro_init()
	{
	BrokerComm::enable();
	BrokerComm::subscribe_to_events("bro/event/");
	BrokerComm::listen(broker_port, "127.0.0.1");
	}

event BrokerComm::incoming_connection_established(peer_name: string)
	{
	print "BrokerComm::incoming_connection_established", peer_name;
	}

event my_event(msg: string, c: count)
	{
	++msg_count;
	print "got my_event", msg, c;

	if ( msg_count == 5 )
		terminate();
	}

event my_auto_event(msg: string, c: count)
	{
	++msg_count;
	print "got my_auto_event", msg, c;

	if ( msg_count == 5 )
		terminate();
	}
```

There are two different ways to send events. The first is to call the `Broker::send_event` function directly. The second option is to call the `Broker::auto_event` function where you specify a particular event that will be automatically sent to peers whenever the event is called locally via the normal event invocation syntax.

```
events-connector.bro

const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "connector";
global my_event: event(msg: string, c: count);
global my_auto_event: event(msg: string, c: count);

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::connect("127.0.0.1", broker_port, 1sec);
        BrokerComm::auto_event("bro/event/my_auto_event", my_auto_event);
        }

event BrokerComm::outgoing_connection_established(peer_address: string,
                                                  peer_port: port,
                                                  peer_name: string)
        {
        print "BrokerComm::outgoing_connection_established",
              peer_address, peer_port, peer_name;
        BrokerComm::event("bro/event/my_event", BrokerComm::event_args(my_event, "hi
↪", 0));
        event my_auto_event("stuff", 88);
        BrokerComm::event("bro/event/my_event", BrokerComm::event_args(my_event, "...
↪", 1));
        event my_auto_event("more stuff", 51);
        BrokerComm::event("bro/event/my_event", BrokerComm::event_args(my_event, "bye
↪", 2));
        }

event BrokerComm::outgoing_connection_broken(peer_address: string,
                                             peer_port: port)
        {
        terminate();
        }
```

Again, the subscription model is prefix-based.

### Message Format

For other applications that want to exchange event messages with Bro, the Broker message format is:

The first parameter is the name of the event and the remaining `...` are its arguments, which are any of the supported Broker data types as they correspond to the Bro types for the event named in the first parameter of the message.

### Remote Logging

```
testlog.bro


module Test;

export {
        redef enum Log::ID += { LOG };

        type Info: record {
                msg: string &log;
                num: count &log;
        };

        global log_test: event(rec: Test::Info);
}

event bro_init() &priority=5
        {
        BrokerComm::enable();
        Log::create_stream(Test::LOG, [$columns=Test::Info, $ev=log_test, $path="test
↪"]);
        }
```

Use the `Broker::subscribe_to_logs` function to advertise interest in logs written by peers. The topic names that Bro uses are implicitly of the form "bro/log/<stream-name>".

```
logs-listener.bro

@load ./testlog

const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "listener";

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::subscribe_to_logs("bro/log/Test::LOG");
        BrokerComm::listen(broker_port, "127.0.0.1");
        }

event BrokerComm::incoming_connection_established(peer_name: string)
        {
        print "BrokerComm::incoming_connection_established", peer_name;
        }

event Test::log_test(rec: Test::Info)
        {
        print "wrote log", rec;

        if ( rec$num == 5 )
                terminate();
        }
```

To send remote logs either redef `Log::enable_remote_logging` or use the `Broker::enable_remote_logs` function. The former allows any log stream to be sent to peers while the latter enables remote logging for particular streams.

```
logs-connector.bro

@load ./testlog

const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;
redef BrokerComm::endpoint_name = "connector";
redef Log::enable_local_logging = F;
redef Log::enable_remote_logging = F;
global n = 0;

```

```
event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::enable_remote_logs(Test::LOG);
        BrokerComm::connect("127.0.0.1", broker_port, 1sec);
        }

event do_write()
        {
        if ( n == 6 )
                return;

        Log::write(Test::LOG, [$msg = "ping", $num = n]);
        ++n;
        event do_write();
        }

event BrokerComm::outgoing_connection_established(peer_address: string,
                                                  peer_port: port,
                                                  peer_name: string)
        {
        print "BrokerComm::outgoing_connection_established",
              peer_address, peer_port, peer_name;
        event do_write();
        }

event BrokerComm::outgoing_connection_broken(peer_address: string,
                                             peer_port: port)
        {
        terminate();
        }
```

### Message Format

For other applications that want to exchange log messages with Bro, the Broker message format is:

The enum value corresponds to the stream's `Log::ID` value, and the record corresponds to a single entry of that log's columns record, in this case a `Test::Info` value.

### Tuning Access Control

By default, endpoints do not restrict the message topics that it sends to peers and do not restrict what message topics and data store identifiers get advertised to peers. These are the default `Broker::EndpointFlags` supplied to `Broker::enable`.

If not using the `auto_publish` flag, one can use the `Broker::publish_topic` and `Broker::unpublish_topic` functions to manipulate the set of message topics (must match exactly) that are allowed to be sent to peer endpoints. These settings take precedence over the per-message `peers` flag supplied to functions that take a `Broker::SendFlags` such as `Broker::send_print`, `Broker::send_event`, `Broker::auto_event` or `Broker::enable_remote_logs`.

If not using the `auto_advertise` flag, one can use the `Broker::advertise_topic` and `Broker::unadvertise_topic` functions to manipulate the set of topic prefixes that are allowed to be advertised to peers. If an endpoint does not advertise a topic prefix, then the only way peers can send messages to it is via the `unsolicited` flag of `Broker::SendFlags` and choosing a topic with a matching prefix (i.e. full topic may be longer than receivers prefix, just the prefix needs to match).

## Distributed Data Stores

There are three flavors of key-value data store interfaces: master, clone, and frontend.

A frontend is the common interface to query and modify data stores. That is, a clone is a specific type of frontend and a master is also a specific type of frontend, but a standalone frontend can also exist to e.g. query and modify the contents of a remote master store without actually "owning" any of the contents itself.

A master data store can be cloned from remote peers which may then perform lightweight, local queries against the clone, which automatically stays synchronized with the master store. Clones cannot modify their content directly, instead they send modifications to the centralized master store which applies them and then broadcasts them to all clones.

Master and clone stores get to choose what type of storage backend to use. E.g. In-memory versus SQLite for persistence. Note that if clones are used, then data store sizes must be able to fit within memory regardless of the storage backend as a single snapshot of the master store is sent in a single chunk to initialize the clone.

Data stores also support expiration on a per-key basis either using an absolute point in time or a relative amount of time since the entry's last modification time.

```
stores-listener.bro

const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;

global h: opaque of BrokerStore::Handle;
global expected_key_count = 4;
global key_count = 0;

function do_lookup(key: string)
	{
	when ( local res = BrokerStore::lookup(h, BrokerComm::data(key)) )
		{
		++key_count;
		print "lookup", key, res;

		if ( key_count == expected_key_count )
			terminate();
		}
	timeout 10sec
		{ print "timeout", key; }
	}

event ready()
	{
	h = BrokerStore::create_clone("mystore");

	when ( local res = BrokerStore::keys(h) )
		{
		print "clone keys", res;
		do_lookup(BrokerComm::refine_to_string(BrokerComm::vector_lookup(res
↪$result, 0)));
		do_lookup(BrokerComm::refine_to_string(BrokerComm::vector_lookup(res
↪$result, 1)));
		do_lookup(BrokerComm::refine_to_string(BrokerComm::vector_lookup(res
↪$result, 2)));
```

```
                do_lookup(BrokerComm::refine_to_string(BrokerComm::vector_lookup(res
↪$result, 3)));
                }
        timeout 10sec
                { print "timeout"; }
        }

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::subscribe_to_events("bro/event/ready");
        BrokerComm::listen(broker_port, "127.0.0.1");
        }
```

```
stores-connector.bro

const broker_port: port = 9999/tcp &redef;
redef exit_only_after_terminate = T;

global h: opaque of BrokerStore::Handle;

function dv(d: BrokerComm::Data): BrokerComm::DataVector
        {
        local rval: BrokerComm::DataVector;
        rval[0] = d;
        return rval;
        }

global ready: event();

event BrokerComm::outgoing_connection_broken(peer_address: string,
                                              peer_port: port)
        {
        terminate();
        }

event BrokerComm::outgoing_connection_established(peer_address: string,
                                                   peer_port: port,
                                                   peer_name: string)
        {
        local myset: set[string] = {"a", "b", "c"};
        local myvec: vector of string = {"alpha", "beta", "gamma"};
        h = BrokerStore::create_master("mystore");
        BrokerStore::insert(h, BrokerComm::data("one"), BrokerComm::data(110));
        BrokerStore::insert(h, BrokerComm::data("two"), BrokerComm::data(223));
        BrokerStore::insert(h, BrokerComm::data("myset"), BrokerComm::data(myset));
        BrokerStore::insert(h, BrokerComm::data("myvec"), BrokerComm::data(myvec));
        BrokerStore::increment(h, BrokerComm::data("one"));
        BrokerStore::decrement(h, BrokerComm::data("two"));
        BrokerStore::add_to_set(h, BrokerComm::data("myset"), BrokerComm::data("d"));
        BrokerStore::remove_from_set(h, BrokerComm::data("myset"), BrokerComm::data("b
↪"));
        BrokerStore::push_left(h, BrokerComm::data("myvec"), dv(BrokerComm::data(
↪"delta")));
        BrokerStore::push_right(h, BrokerComm::data("myvec"), dv(BrokerComm::data(
↪"omega")));

        when ( local res = BrokerStore::size(h) )
```

```
                {
                print "master size", res;
                event ready();
                }
        timeout 10sec
                { print "timeout"; }
        }

event bro_init()
        {
        BrokerComm::enable();
        BrokerComm::connect("127.0.0.1", broker_port, 1secs);
        BrokerComm::auto_event("bro/event/ready", ready);
        }
```

In the above example, if a local copy of the store contents isn't needed, just replace the `Broker::create_clone` call with `Broker::create_frontend`. Queries will then be made against the remote master store instead of the local clone.

Note that all data store queries must be made within Bro's asynchronous `when` statements and must specify a timeout block.

## 3.2 Script Reference

### 3.2.1 Operators

The Bro scripting language supports the following operators. Note that each data type only supports a subset of these operators. For more details, see the documentation about the data types.

#### Relational operators

The relational operators evaluate to type *bool*.

| Name | Syntax |
|---|---|
| Equality | *a* == *b* |
| Inequality | *a* != *b* |
| Less than | *a* < *b* |
| Less than or equal | *a* <= *b* |
| Greater than | *a* > *b* |
| Greater than or equal | *a* >= *b* |

#### Logical operators

The logical operators require operands of type *bool*, and evaluate to type *bool*.

| Name | Syntax |
|---|---|
| Logical AND | *a* && *b* |
| Logical OR | *a* \|\| *b* |
| Logical NOT | ! *a* |

### Arithmetic operators

| Name | Syntax | Notes |
|---|---|---|
| Addition | *a* + *b* | For `string` operands, this performs string concatenation. |
| Subtraction | *a* - *b* | |
| Multiplication | *a* * *b* | |
| Division | *a* / *b* | For `int` or `count` operands, the fractional part of the result is dropped. |
| Modulo | *a* % *b* | Operand types cannot be "double". |
| Unary plus | + *a* | |
| Unary minus | - *a* | |
| Pre-increment | ++ *a* | Operand type cannot be "double". |
| Pre-decrement | −− *a* | Operand type cannot be "double". |
| Absolute value | \| *a* \| | If operand is `string`, `set`, `table`, or `vector`, this evaluates to number of elements. |

### Assignment operators

The assignment operators evaluate to the result of the assignment.

| Name | Syntax |
|---|---|
| Assignment | *a* = *b* |
| Addition assignment | *a* += *b* |
| Subtraction assignment | *a* -= *b* |

### Record field operators

The record field operators take a `record` as the first operand, and a field name as the second operand. For both operators, the specified field name must be in the declaration of the record type.

| Name | Syntax | Notes |
|---|---|---|
| Field access | *a* $ *b* | |
| Field value existence test | *a* ?$ *b* | Evaluates to type `bool`. True if the specified field has been assigned a value, or false if not. |

### Other operators

| Name | Syntax | Notes |
|---|---|---|
| Membership test | *a* in *b* | Evaluates to type `bool`. Do not confuse this use of "in" with that used in a `for` statement. |
| Non-membership test | *a* !in *b* | This is the logical NOT of the "in" operator. For example: "a !in b" is equivalent to "!(a in b)". |
| Table or vector element access | *a* [ *b* ] | This operator can also be used with a `set`, but only with the `add` or `delete` statement. |
| Substring extraction | *a* [ *b* : *c* ] | See the `string` type for more details. |
| Create a deep copy | copy ( *a* ) | This is relevant only for data types that are assigned by reference, such as `vector`, `set`, `table`, and `record`. |
| Module namespace access | *a* :: *b* | The first operand is the module name, and the second operand is an identifier that refers to a global variable, enumeration constant, or user-defined type that was exported from the module. |
| Conditional | *a* ? *b* : *c* | The first operand must evaluate to type `bool`. If true, then the second expression is evaluated and is the result of the entire expression. Otherwise, the third expression is evaluated and is the result of the entire expression. The types of the second and third operands must be compatible. |

## 3.2.2 Types

The Bro scripting language supports the following built-in types:

| Name | Description |
|---|---|
| `bool` | Boolean |
| `count`, `int`, `double` | Numeric types |
| `time`, `interval` | Time types |
| `string` | String |
| `pattern` | Regular expression |
| `port`, `addr`, `subnet` | Network types |
| `enum` | Enumeration (user-defined type) |
| `table`, `set`, `vector`, `record` | Container types |
| `function`, `event`, `hook` | Executable types |
| `file` | File type (only for writing) |
| `opaque` | Opaque type (for some built-in functions) |
| `any` | Any type (for functions or containers) |

Here is a more detailed description of each type:

**`bool`**

Reflects a value with one of two meanings: true or false. The two "bool" constants are `T` and `F`.

The "bool" type supports the following operators: equality/inequality (`==`, `!=`), logical and/or (`&&`, `||`), logical negation (`!`), and absolute value (where `|T|` is 1, and `|F|` is 0, and in both cases the result type is `count`).

**`int`**

A numeric type representing a 64-bit signed integer. An "int" constant is a string of digits preceded by a "+" or "-" sign, e.g. `-42` or `+5` (the "+" sign is optional but see note about type inferencing below). An "int" constant can also be written in hexadecimal notation (in which case "0x" must be between the sign and the hex digits), e.g. `-0xFF` or `+0xabc123`.

The "int" type supports the following operators: arithmetic operators (`+`, `-`, `*`, `/`, `%`), comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`), assignment operators (`=`, `+=`, `-=`), pre-increment (`++`), pre-decrement (`--`), unary plus and minus (`+`, `-`), and absolute value (e.g., `|-3|` is 3, but the result type is *`count`*).

When using type inferencing use care so that the intended type is inferred, e.g. "local size_difference = 0" will infer "*`count`*", while "local size_difference = +0" will infer "int".

**`count`**

A numeric type representing a 64-bit unsigned integer. A "count" constant is a string of digits, e.g. `1234` or `0`. A "count" can also be written in hexadecimal notation (in which case "0x" must precede the hex digits), e.g. `0xff` or `0xABC123`.

The "count" type supports the same operators as the "*`int`*" type, but a unary plus or minus applied to a "count" results in an "int".

**`double`**

A numeric type representing a double-precision floating-point number. Floating-point constants are written as a string of digits with an optional decimal point, optional scale-factor in scientific notation, and optional "+" or "-" sign. Examples are `-1234`, `-1234e0`, `3.14159`, and `.003E-23`.

The "double" type supports the following operators: arithmetic operators (`+`, `-`, `*`, `/`), comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`), assignment operators (`=`, `+=`, `-=`), unary plus and minus (`+`, `-`), and absolute value (e.g., `|-3.14|` is 3.14).

When using type inferencing use care so that the intended type is inferred, e.g. "local size_difference = 5" will infer "*`count`*", while "local size_difference = 5.0" will infer "double".

**`time`**

A temporal type representing an absolute time. There is currently no way to specify a `time` constant, but one can use the `double_to_time`, `current_time`, or `network_time` built-in functions to assign a value to a `time`-typed variable.

Time values support the comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`). A `time` value can be subtracted from another `time` value to produce an *`interval`* value. An `interval` value can be added to, or subtracted from, a `time` value to produce a `time` value. The absolute value of a `time` value is a *`double`* with the same numeric value.

**`interval`**

A temporal type representing a relative time. An `interval` constant can be written as a numeric constant followed by a time unit where the time unit is one of `usec`, `msec`, `sec`, `min`, `hr`, or `day` which respectively represent microseconds, milliseconds, seconds, minutes, hours, and days. Whitespace between the numeric constant and time unit is optional. Appending the letter "s" to the time unit in order to pluralize it is also optional (to no semantic effect). Examples of `interval` constants are `3.5 min` and `3.5mins`. An `interval` can also be negated, for example `-12 hr` represents "twelve hours in the past".

Intervals support addition and subtraction, the comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`), the assignment operators (`=`, `+=`, `-=`), and unary plus and minus (`+`, `-`).

Intervals also support division (in which case the result is a *`double`* value). An `interval` can be multiplied or divided by an arithmetic type (`count`, `int`, or `double`) to produce an `interval` value. The absolute value of an `interval` is a `double` value equal to the number of seconds in the `interval` (e.g., `|-1 min|` is 60.0).

**`string`**

A type used to hold character-string values which represent text, although strings in a Bro script can actually contain any arbitrary binary data.

String constants are created by enclosing text within a pair of double quotes ("). A string constant cannot span multiple lines in a Bro script. The backslash character (\) introduces escape sequences. The following escape sequences are recognized: `\n`, `\t`, `\v`, `\b`, `\r`, `\f`, `\a`, `\ooo` (where each 'o' is an octal digit), `\xhh` (where each 'h' is a hexadecimal digit). For escape sequences that don't match any of these, Bro will just remove the backslash (so to represent a literal backslash in a string constant, you just use two consecutive backslashes).

Strings support concatenation (+), and assignment (=, +=). Strings also support the comparison operators (==, `!=`, <, <=, >, >=). The number of characters in a string can be found by enclosing the string within pipe characters (e.g., `|"abc"|` is 3). Substring searching can be performed using the "in" or "!in" operators (e.g., "bar" in "foobar" yields true).

The subscript operator can extract a substring of a string. To do this, specify the starting index to extract (if the starting index is omitted, then zero is assumed), followed by a colon and index one past the last character to extract (if the last index is omitted, then the extracted substring will go to the end of the original string). However, if both the colon and last index are omitted, then a string of length one is extracted. String indexing is zero-based, but an index of -1 refers to the last character in the string, and -2 refers to the second-to-last character, etc. Here are a few examples:

```
local orig = "0123456789";
local second_char = orig[1];         # "1"
local last_char = orig[-1];          # "9"
local first_two_chars = orig[:2];    # "01"
local last_two_chars = orig[8:];     # "89"
local no_first_and_last = orig[1:9]; # "12345678"
local no_first = orig[1:];           # "123456789"
local no_last = orig[:-1];           # "012345678"
local copy_orig = orig[:];           # "0123456789"
```

Note that the subscript operator cannot be used to modify a string (i.e., it cannot be on the left side of an assignment operator).

**pattern**

A type representing regular-expression patterns which can be used for fast text-searching operations. Pattern constants are created by enclosing text within forward slashes (/) and is the same syntax as the patterns supported by the flex lexical analyzer. The speed of regular expression matching does not depend on the complexity or size of the patterns. Patterns support two types of matching, exact and embedded.

In exact matching the == equality relational operator is used with one "pattern" operand and one "*string*" operand (order of operands does not matter) to check whether the full string exactly matches the pattern. In exact matching, the ^ beginning-of-line and $ end-of-line anchors are redundant since the pattern is implicitly anchored to the beginning and end of the line to facilitate an exact match. For example:

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

yields true, while:

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

yields false. The `!=` operator would yield the negation of ==.

In embedded matching the `in` operator is used with one "pattern" operand (which must be on the left-hand side) and one "*string*" operand, but tests whether the pattern appears anywhere within the given string. For example:

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

yields true, while:

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

is false since "oob" does not appear at the start of "foobar". The `!in` operator would yield the negation of `in`.

**port**

A type representing transport-level port numbers (besides TCP and UDP ports, there is a concept of an ICMP "port" where the source port is the ICMP message type and the destination port the ICMP message code). A `port` constant is written as an unsigned integer followed by one of `/tcp`, `/udp`, `/icmp`, or `/unknown`.

Ports support the comparison operators (==, `!=`, <, <=, >, >=). When comparing order across transport-level protocols, `unknown` < `tcp` < `udp` < `icmp`, for example `65535/tcp` is smaller than `0/udp`.

Note that you can obtain the transport-level protocol type of a `port` with the `get_port_transport_proto` built-in function, and the numeric value of a `port` with the `port_to_count` built-in function.

**addr**

A type representing an IP address.

IPv4 address constants are written in "dotted quad" format, `A1.A2.A3.A4`, where Ai all lie between 0 and 255.

IPv6 address constants are written as colon-separated hexadecimal form as described by **RFC 2373** (including the mixed notation with embedded IPv4 addresses as dotted-quads in the lower 32 bits), but additionally encased in square brackets. Some examples: `[2001:db8::1]`, `[::ffff:192.168.1.100]`, or `[aaaa:bbbb:cccc:dddd:eeee:ffff:1111:2222]`.

Note that IPv4-mapped IPv6 addresses (i.e., addresses with the first 80 bits zero, the next 16 bits one, and the remaining 32 bits are the IPv4 address) are treated internally as IPv4 addresses (for example, `[::ffff:192.168.1.100]` is equal to `192.168.1.100`).

Addresses can be compared for equality (==, `!=`), and also for ordering (<, <=, >, >=). The absolute value of an address gives the size in bits (32 for IPv4, and 128 for IPv6). Addresses can also be masked with `/` to produce a *`subnet`*:

And checked for inclusion within a *`subnet`* using `in` or `!in`:

You can check if a given `addr` is IPv4 or IPv6 using the `is_v4_addr` and `is_v6_addr` built-in functions.

Note that hostname constants can also be used, but since a hostname can correspond to multiple IP addresses, the type of such a variable is "set[addr]". For example:

**subnet**

A type representing a block of IP addresses in CIDR notation. A `subnet` constant is written as an *`addr`* followed by a slash (/) and then the network prefix size specified as a decimal number. For example, `192.168.0.0/16` or `[fe80::]/64`.

Subnets can be compared for equality (==, `!=`). An "addr" can be checked for inclusion in a subnet using the `in` or `!in` operators.

**enum**

A type allowing the specification of a set of related values that have no further structure. An example declaration:

The last comma after `Blue` is optional. Both the type name `color` and the individual values (`Red`, etc.) have global scope.

Enumerations do not have associated values or ordering. The only operations allowed on enumerations are equality comparisons (==, `!=`) and assignment (=).

**table**

An associate array that maps from one set of values to another. The values being mapped are termed the *index* or *indices* and the result of the mapping is called the *yield*. Indexing into tables is very efficient, and internally it is just a single hash table lookup.

The table declaration syntax is:

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

where *type*^+ is one or more types, separated by commas. The index type cannot be any of the following types: pattern, table, set, vector, file, opaque, any.

Here is an example of declaring a table indexed by "count" values and yielding "string" values:

The yield type can also be more complex:

which declares a table indexed by "count" and yielding another "table" which is indexed by an "addr" and "port" to yield a "string".

One way to initialize a table is by enclosing a set of initializers within braces, for example:

A table constructor can also be used to create a table:

Table constructors can also be explicitly named by a type, which is useful when a more complex index type could otherwise be ambiguous:

Accessing table elements is provided by enclosing index values within square brackets (`[]`), for example:

And membership can be tested with `in` or `!in`:

Add or overwrite individual table elements by assignment:

Remove individual table elements with *`delete`*:

Nothing happens if the element with index value `13` isn't present in the table.

The number of elements in a table can be obtained by placing the table identifier between vertical pipe characters:

See the *`for`* statement for info on how to iterate over the elements in a table.

**`set`**

A set is like a *`table`*, but it is a collection of indices that do not map to any yield value. They are declared with the syntax:

```
set [ type^+ ]
```

where *type*^+ is one or more types separated by commas. The index type cannot be any of the following types: pattern, table, set, vector, file, opaque, any.

Sets can be initialized by listing elements enclosed by curly braces:

A set constructor (equivalent to above example) can also be used to create a set:

Set constructors can also be explicitly named by a type, which is useful when a more complex index type could otherwise be ambiguous:

Set membership is tested with `in` or `!in`:

Elements are added with *`add`*:

Nothing happens if the element with value `22/tcp` was already present in the set.

And removed with *`delete`*:

Nothing happens if the element with value `21/tcp` isn't present in the set.

The number of elements in a set can be obtained by placing the set identifier between vertical pipe characters:

See the *`for`* statement for info on how to iterate over the elements in a set.

**`vector`**

A vector is like a *`table`*, except it's always indexed by a *`count`* (and vector indexing is always zero-based). A vector is declared like:

And can be initialized with the vector constructor:

Vector constructors can also be explicitly named by a type, which is useful for when a more complex yield type could otherwise be ambiguous.

Accessing vector elements is provided by enclosing index values within square brackets (`[]`), for example:

An element can be added to a vector by assigning the value (a value that already exists at that index will be overwritten):

The number of elements in a vector can be obtained by placing the vector identifier between vertical pipe characters:

Vectors of integral types (`int` or `count`) support the pre-increment (`++`) and pre-decrement operators (`--`), which will increment or decrement each element in the vector.

Vectors of arithmetic types (`int`, `count`, or `double`) can be operands of the arithmetic operators (`+`, `-`, `*`, `/`, `%`), but both operands must have the same number of elements (and the modulus operator `%` cannot be used if either operand is a `vector of double`). The resulting vector contains the result of the operation applied to each of the elements in the operand vectors.

Vectors of bool can be operands of the logical "and" (`&&`) and logical "or" (`||`) operators (both operands must have same number of elements). The resulting vector of bool is the logical "and" (or logical "or") of each element of the operand vectors.

See the *`for`* statement for info on how to iterate over the elements in a vector.

**`record`**

A "record" is a collection of values. Each value has a field name and a type. Values do not need to have the same type and the types have no restrictions. Field names must follow the same syntax as regular variable names (except that field names are allowed to be the same as local or global variables). An example record type definition:

Records can be initialized or assigned as a whole in three different ways. When assigning a whole record value, all fields that are not *`&optional`* or have a *`&default`* attribute must be specified. First, there's a constructor syntax:

And the constructor can be explicitly named by type, too, which is arguably more readable:

And the third way is like this:

Access to a record field uses the dollar sign (`$`) operator, and record fields can be assigned with this:

To test if a field that is *`&optional`* has been assigned a value, use the `?$` operator (it returns a *`bool`* value of `T` if the field has been assigned a value, or `F` if not):

**`function`**

Function types in Bro are declared using:

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

where *argument* is a (possibly empty) comma-separated list of arguments, and *type* is an optional return type. For example:

Here `greeting` is an identifier with a certain function type. The function body is not defined yet and `greeting` could even have different function body values at different times. To define a function including a body value, the syntax is like:

Note that in the definition above, it's not necessary for us to have done the first (forward) declaration of `greeting` as a function type, but when it is, the return type and argument list (including the name of each argument) must match exactly.

Here is an example function that takes no parameters and does not return a value:

Function types don't need to have a name and can be assigned anonymously:

And finally, the function can be called like:

Function parameters may specify default values as long as they appear last in the parameter list:

If a function was previously declared with default parameters, the default expressions can be omitted when implementing the function body and they will still be used for function calls that lack those arguments.

And calls to the function may omit the defaults from the argument list:

**event**

Event handlers are nearly identical in both syntax and semantics to a *function*, 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.

Example:

Instead of directly calling an event handler from a script, event handler bodies are executed when they are invoked by one of three different methods:

- 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 flushing the invocation of other event handlers that were queued first.

- With the `event` statement from a script

  Immediately queuing invocation of an event handler occurs like:

  This assumes that `password_exposed` was previously declared as an event handler type with compatible arguments.

- Via the *schedule* expression in a script

  This delays the invocation of event handlers until some time in the future. For example:

Multiple event handler bodies can be defined for the same event handler identifier and the body of each will be executed in turn. Ordering of execution can be influenced with *&priority*.

**hook**

A hook is another flavor of function that shares characteristics of both a *function* and an *event*. They are like events in that many handler bodies can be defined for the same hook identifier and the order of execution can be enforced with *&priority*. They are more like functions in the way they are invoked/called, because, unlike events, their execution is immediate and they do not get scheduled through an event queue. Also, a unique feature of a hook is that a given hook handler body can short-circuit the execution of remaining hook handlers simply by exiting from the body as a result of a *break* statement (as opposed to a *return* or just reaching the end of the body).

A hook type is declared like:

```
hook( argument* )
```

where *argument* is a (possibly empty) comma-separated list of arguments. For example:

Here `myhook` is the hook type identifier and no hook handler bodies have been defined for it yet. To define some hook handler bodies the syntax looks like:

Note that the first (forward) declaration of `myhook` as a hook type isn't strictly required. Argument types must match for all hook handlers and any forward declaration of a given hook.

To invoke immediate execution of all hook handler bodies, they are called similarly to a function, except preceded by the `hook` keyword:

or

And the output would look like:

```
priority 10 myhook handler, hi
break out of myhook handling, bye
```

Note how the modification to arguments can be seen by remaining hook handlers.

The return value of a hook call is an implicit *bool* value with `T` meaning that all handlers for the hook were executed and `F` meaning that only some of the handlers may have executed due to one handler body exiting as a result of a `break` statement.

**file**

Bro supports writing to files, but not reading from them (to read from files see the *Input Framework*). Files can be opened using either the `open` or `open_for_append` built-in functions, and closed using the `close` built-in function. For example, declare, open, and write to a file and finally close it like:

Writing to files like this for logging usually isn't recommended, for better logging support see *Logging Framework*.

**opaque**

A data type whose actual representation/implementation is intentionally hidden, but whose values may be passed to certain built-in functions that can actually access the internal/hidden resources. Opaque types are differentiated from each other by qualifying them like "opaque of md5" or "opaque of sha1".

An example use of this type is the set of built-in functions which perform hashing:

Here the opaque type is used to provide a handle to a particular resource which is calculating an MD5 hash incrementally over time, but the details of that resource aren't relevant, it's only necessary to have a handle as a way of identifying it and distinguishing it from other such resources.

**any**

Used to bypass strong typing. For example, a function can take an argument of type `any` when it may be of different types. The only operation allowed on a variable of type `any` is assignment.

Note that users aren't expected to use this type. It's provided mainly for use by some built-in functions and scripts included with Bro.

**void**

An internal Bro type (i.e., "void" is not a reserved keyword in the Bro scripting language) representing the absence of a return type for a function.

### 3.2.3 Attributes

The Bro scripting language supports the following attributes.

| Name | Description |
|---|---|
| *&redef* | Redefine a global constant or extend a type. |
| *&priority* | Specify priority for event handler or hook. |
| *&log* | Mark a record field as to be written to a log. |
| *&optional* | Allow a record field value to be missing. |
| *&default* | Specify a default value. |
| *&add_func* | Specify a function to call for each "redef +=". |
| *&delete_func* | Same as "&add_func", except for "redef -=". |
| *&expire_func* | Specify a function to call when container element expires. |
| *&read_expire* | Specify a read timeout interval. |
| *&write_expire* | Specify a write timeout interval. |
| *&create_expire* | Specify a creation timeout interval. |
| *&synchronized* | Synchronize a variable across nodes. |
| *&persistent* | Make a variable persistent (written to disk). |
| *&rotate_interval* | Rotate a file after specified interval. |
| *&rotate_size* | Rotate a file after specified file size. |
| *&encrypt* | Encrypt a file when writing to disk. |
| *&raw_output* | Open file in raw mode (chars. are not escaped). |
| *&mergeable* | Prefer set union for synchronized state. |
| *&error_handler* | Used internally for reporter framework events. |
| *&type_column* | Used by input framework for "port" type. |
| *&deprecated* | Marks an identifier as deprecated. |

Here is a more detailed explanation of each attribute:

**&redef**

Allows use of a *redef* to redefine initial values of global variables (i.e., variables declared either *global* or *const*). Example:

```
const clever = T &redef;
global cache_size = 256 &redef;
```

Note that a variable declared "global" can also have its value changed with assignment statements (doesn't matter if it has the "&redef" attribute or not).

**&priority**

Specifies the execution priority (as a signed integer) of a hook or event handler. Higher values are executed before lower ones. The default value is 0. Example:

```
event bro_init() &priority=10
{
    print "high priority";
}
```

**&log**

Writes a *record* field to the associated log stream.

**&optional**

Allows a record field value to be missing (i.e., neither initialized nor ever assigned a value).

In this example, the record could be instantiated with either "myrec($a=127.0.0.1)" or "myrec($a=127.0.0.1, $b=80/tcp)":

```
type myrec: record { a: addr; b: port &optional; };
```

The `?$` operator can be used to check if a record field has a value or not (it returns a `bool` value of `T` if the field has a value, and `F` if not).

**&default**

Specifies a default value for a record field, container element, or a function/hook/event parameter.

In this example, the record could be instantiated with either "myrec($a=5, $c=3.14)" or "myrec($a=5, $b=53/udp, $c=3.14)":

```
type myrec: record { a: count; b: port &default=80/tcp; c: double; };
```

In this example, the table will return the string `"foo"` for any attempted access to a non-existing index:

```
global mytable: table[count] of string &default="foo";
```

When used with function/hook/event parameters, all of the parameters with the "&default" attribute must come after all other parameters. For example, the following function could be called either as "myfunc(5)" or as "myfunc(5, 53/udp)":

```
function myfunc(a: count, b: port &default=80/tcp)
{
    print a, b;
}
```

**&add_func**

Can be applied to an identifier with &redef to specify a function to be called any time a "redef <id> += ..." declaration is parsed. The function takes two arguments of the same type as the identifier, the first being the old value of the variable and the second being the new value given after the "+=" operator in the "redef" declaration. The return value of the function will be the actual new value of the variable after the "redef" declaration is parsed.

**&delete_func**

Same as *&add_func*, except for *redef* declarations that use the "-=" operator.

**&expire_func**

Called right before a container element expires. The function's first parameter is of the same type of the container and the second parameter the same type of the container's index. The return value is an *interval* indicating the amount of additional time to wait before expiring the container element at the given index (which will trigger another execution of this function).

**&read_expire**

Specifies a read expiration timeout for container elements. That is, the element expires after the given amount of time since the last time it has been read. Note that a write also counts as a read.

**&write_expire**

Specifies a write expiration timeout for container elements. That is, the element expires after the given amount of time since the last time it has been written.

**&create_expire**

Specifies a creation expiration timeout for container elements. That is, the element expires after the given amount of time since it has been inserted into the container, regardless of any reads or writes.

**&synchronized**

Synchronizes variable accesses across nodes. The value of a `&synchronized` variable is automatically propagated to all peers when it changes.

**&persistent**

Makes a variable persistent, i.e., its value is written to disk (per default at shutdown time).

**&rotate_interval**

Rotates a file after a specified interval.

Note: This attribute is deprecated and will be removed in a future release.

**&rotate_size**
Rotates a file after it has reached a given size in bytes.

Note: This attribute is deprecated and will be removed in a future release.

**&encrypt**
Encrypts files right before writing them to disk.

Note: This attribute is deprecated and will be removed in a future release.

**&raw_output**
Opens a file in raw mode, i.e., non-ASCII characters are not escaped.

**&mergeable**
Prefers merging sets on assignment for synchronized state. This attribute is used in conjunction with *&synchronized* container types: when the same container is updated at two peers with different values, the propagation of the state causes a race condition, where the last update succeeds. This can cause inconsistencies and can be avoided by unifying the two sets, rather than merely overwriting the old value.

**&error_handler**
Internally set on the events that are associated with the reporter framework: `reporter_info`, `reporter_warning`, and `reporter_error`. It prevents any handlers of those events from being able to generate reporter messages that go through any of those events (i.e., it prevents an infinite event recursion). Instead, such nested reporter messages are output to stderr.

**&type_column**
Used by the input framework. It can be used on columns of type *port* (such a column only contains the port number) and specifies the name of an additional column in the input file which specifies the protocol of the port (tcp/udp/icmp).

In the following example, the input file would contain four columns named “ip”, “srcp”, “proto”, and “msg”:

```
type Idx: record {
    ip: addr;
};


type Val: record {
    srcp: port &type_column = "proto";
    msg: string;
};
```

**&deprecated**
The associated identifier is marked as deprecated and will be removed in a future version of Bro. Look in the NEWS file for more instructions to migrate code that uses deprecated functionality.

### 3.2.4 Declarations and Statements

The Bro scripting language supports the following declarations and statements.

## Declarations

| Name | Description |
|---|---|
| *module* | Change the current module |
| *export* | Export identifiers from the current module |
| *global* | Declare a global variable |
| *const* | Declare a constant |
| *type* | Declare a user-defined type |
| *redef* | Redefine a global value or extend a user-defined type |
| *function/event/hook* | Declare a function, event handler, or hook |

## Statements

| Name | Description |
|---|---|
| *local* | Declare a local variable |
| *add*, *delete* | Add or delete elements |
| *print* | Print to stdout or a file |
| *for*, *while*, *next*, *break* | Loop over each element in a container object (`for`), or as long as a condition evaluates to true (`while`). |
| *if* | Evaluate boolean expression and if true, execute a statement |
| *switch*, *break*, *fallthrough* | Evaluate expression and execute statement with a matching value |
| *when* | Asynchronous execution |
| *event*, *schedule* | Invoke or schedule an event handler |
| *return* | Return from function, hook, or event handler |

## Declarations

Declarations cannot occur within a function, hook, or event handler.

Declarations must appear before any statements (except those statements that are in a function, hook, or event handler) in the concatenation of all loaded Bro scripts.

**module**

The "module" keyword is used to change the current module. This affects the scope of any subsequently declared global identifiers.

Example:

```
module mymodule;
```

If a global identifier is declared after a "module" declaration, then its scope ends at the end of the current Bro script or at the next "module" declaration, whichever comes first. However, if a global identifier is declared after a "module" declaration, but inside an *export* block, then its scope ends at the end of the last loaded Bro script, but it must be referenced using the namespace operator (`::`) in other modules.

There can be any number of "module" declarations in a Bro script. The same "module" declaration can appear in any number of different Bro scripts.

**export**

An "export" block contains one or more declarations (no statements are allowed in an "export" block) that the current module is exporting. This enables these global identifiers to be visible in other modules (but not prior to their declaration) via the namespace operator (`::`). See the *module* keyword for a more detailed explanation.

Example:

```
export {
    redef enum Log::ID += { LOG };

    type Info: record {
        ts: time &log;
        uid: string &log;
    };

    const conntime = 30sec &redef;
}
```

Note that the braces in an "export" block are always required (they do not indicate a compound statement). Also, no semicolon is needed to terminate an "export" block.

**global**

Variables declared with the "global" keyword will be global.

If a type is not specified, then an initializer is required so that the type can be inferred. Likewise, if an initializer is not supplied, then the type must be specified. In some cases, when the type cannot be correctly inferred, the type must be specified even when an initializer is present. Example:

```
global pi = 3.14;
global hosts: set[addr];
global ciphers: table[string] of string = table();
```

Variable declarations outside of any function, hook, or event handler are required to use this keyword (unless they are declared with the *const* keyword instead).

Definitions of functions, hooks, and event handlers are not allowed to use the "global" keyword. However, function declarations (i.e., no function body is provided) can use the "global" keyword.

The scope of a global variable begins where the declaration is located, and extends through all remaining Bro scripts that are loaded (however, see the *module* keyword for an explanation of how modules change the visibility of global identifiers).

**const**

A variable declared with the "const" keyword will be constant.

Variables declared as constant are required to be initialized at the time of declaration. Normally, the type is inferred from the initializer, but the type can be explicitly specified. Example:

```
const pi = 3.14;
const ssh_port: port = 22/tcp;
```

The value of a constant cannot be changed. The only exception is if the variable is a global constant and has the *&redef* attribute, but even then its value can be changed only with a *redef*.

The scope of a constant is local if the declaration is in a function, hook, or event handler, and global otherwise.

Note that the "const" keyword cannot be used with either the "local" or "global" keywords (i.e., "const" replaces "local" and "global").

**type**

The "type" keyword is used to declare a user-defined type. The name of this new type has global scope and can be used anywhere a built-in type name can occur.

The "type" keyword is most commonly used when defining a *record* or an *enum*, but is also useful when dealing with more complex types.

Example:

```
type mytype: table[count] of table[addr, port] of string;
global myvar: mytype;
```

**redef**

There are three ways that "redef" can be used: to change the value of a global variable (but only if it has the *&redef* attribute), to extend a record type or enum type, or to specify a new event handler body that replaces all those that were previously defined.

If you're using "redef" to change a global variable (defined using either *const* or *global*), then the variable that you want to change must have the *&redef* attribute. If the variable you're changing is a table, set, or pattern, you can use += to add new elements, or you can use = to specify a new value (all previous contents of the object are removed). If the variable you're changing is a set or table, then you can use the -= operator to remove the specified elements (nothing happens for specified elements that don't exist). If the variable you are changing is not a table, set, or pattern, then you must use the = operator.

Examples:

```
redef pi = 3.14;
```

If you're using "redef" to extend a record or enum, then you must use the += assignment operator. For an enum, you can add more enumeration constants, and for a record you can add more record fields (however, each record field in the "redef" must have either the *&optional* or *&default* attribute).

Examples:

```
redef enum color += { Blue, Red };
redef record MyRecord += { n2:int &optional; s2:string &optional; };
```

If you're using "redef" to specify a new event handler body that replaces all those that were previously defined (i.e., any subsequently defined event handler body will not be affected by this "redef"), then the syntax is the same as a regular event handler definition except for the presence of the "redef" keyword.

Example:

```
redef event myevent(s:string) { print "Redefined", s; }
```

**function/event/hook** For details on how to declare a *function*, *event* handler, or *hook*, see the documentation for those types.

## Statements

Statements (except those contained within a function, hook, or event handler) can appear only after all global declarations in the concatenation of all loaded Bro scripts.

Each statement in a Bro script must be terminated with a semicolon (with a few exceptions noted below). An individual statement can span multiple lines.

Here are the statements that the Bro scripting language supports.

**add**

The "add" statement is used to add an element to a *set*. Nothing happens if the specified element already exists in the set.

Example:

```
local myset: set[string];
add myset["test"];
```

**break**

The "break" statement is used to break out of a *switch*, *for*, or *while* statement.

**delete**

The "delete" statement is used to remove an element from a *set* or *table*, or to remove a value from a *record* field that has the *&optional* attribute. When attempting to remove an element from a set or table, nothing happens if the specified index does not exist. When attempting to remove a value from an "&optional" record field, nothing happens if that field doesn't have a value.

Example:

```
local myset = set("this", "test");
local mytable = table(["key1"] = 80/tcp, ["key2"] = 53/udp);
local myrec = MyRecordType($a = 1, $b = 2);

delete myset["test"];
delete mytable["key1"];

# In this example, "b" must have the "&optional" attribute
delete myrec$b;
```

**event**

The "event" statement immediately queues invocation of an event handler.

Example:

```
event myevent("test", 5);
```

**fallthrough**

The "fallthrough" statement can be used as the last statement in a "case" block to indicate that execution should continue into the next "case" or "default" label.

For an example, see the *switch* statement.

**for**

A "for" loop iterates over each element in a string, set, vector, or table and executes a statement for each iteration (note that the order in which the loop iterates over the elements in a set or a table is nondeterministic). However, no loop iterations occur if the string, set, vector, or table is empty.

For each iteration of the loop, a loop variable will be assigned to an element if the expression evaluates to a string or set, or an index if the expression evaluates to a vector or table. Then the statement is executed.

If the expression is a table or a set with more than one index, then the loop variable must be specified as a comma-separated list of different loop variables (one for each index), enclosed in brackets.

Note that the loop variable in a "for" statement is not allowed to be a global variable, and it does not need to be declared prior to the "for" statement. The type will be inferred from the elements of the expression.

Currently, modifying a container's membership while iterating over it may result in undefined behavior, so do not add or remove elements inside the loop.

A *break* statement will immediately terminate the "for" loop, and a *next* statement will skip to the next loop iteration.

Example:

```
local myset = set(80/tcp, 81/tcp);
local mytable = table([10.0.0.1, 80/tcp]="s1", [10.0.0.2, 81/tcp]="s2");

for (p in myset)
    print p;
```

```
for ([i,j] in mytable) {
    if (mytable[i,j] == "done")
        break;
    if (mytable[i,j] == "skip")
        next;
    print i,j;
}
```

**if**

Evaluates a given expression, which must yield a *bool* value. If true, then a specified statement is executed. If false, then the statement is not executed. Example:

```
if ( x == 2 ) print "x is 2";
```

However, if the expression evaluates to false and if an "else" is provided, then the statement following the "else" is executed. Example:

```
if ( x == 2 )
    print "x is 2";
else
    print "x is not 2";
```

**local**

A variable declared with the "local" keyword will be local. If a type is not specified, then an initializer is required so that the type can be inferred. Likewise, if an initializer is not supplied, then the type must be specified.

Examples:

```
local x1 = 5.7;
local x2: double;
local x3: double = 5.7;
```

Variable declarations inside a function, hook, or event handler are required to use this keyword (the only two exceptions are variables declared with *const*, and variables implicitly declared in a *for* statement).

The scope of a local variable starts at the location where it is declared and persists to the end of the function, hook, or event handler in which it is declared (this is true even if the local variable was declared within a *compound statement* or is the loop variable in a "for" statement).

**next**

The "next" statement can only appear within a *for* or *while* loop. It causes execution to skip to the next iteration.

**print**

The "print" statement takes a comma-separated list of one or more expressions. Each expression in the list is evaluated and then converted to a string. Then each string is printed, with each string separated by a comma in the output.

Examples:

```
print 3.14;
print "Results", x, y;
```

By default, the "print" statement writes to the standard output (stdout). However, if the first expression is of type *file*, then "print" writes to that file.

If a string contains non-printable characters (i.e., byte values that are not in the range 32 - 126), then the "print" statement converts each non-printable character to an escape sequence before it is printed.

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

**return**

The "return" statement immediately exits the current function, hook, or event handler. For a function, the specified expression (if any) is evaluated and returned. A "return" statement in a hook or event handler cannot return a value because event handlers and hooks do not have return types.

Examples:

```
function my_func(): string
{
    return "done";
}

event my_event(n: count)
{
    if ( n == 0 ) return;

    print n;
}
```

There is a special form of the "return" statement that is only allowed in functions. Syntactically, it looks like a *when* statement immediately preceded by the "return" keyword. This form of the "return" statement is used to specify a function that delays its result (such a function can only be called in the expression of a *when* statement). The function returns at the time the "when" statement's condition becomes true, and the function returns the value that the "when" statement's body returns (or if the condition does not become true within the specified timeout interval, then the function returns the value that the "timeout" block returns).

Example:

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

function a() : count
      {
      # This delays until condition becomes true.
      return when ( "a" in X )
            {
            return X["a"];
            }
      timeout 30 sec
            {
            return 0;
            }
      }

event bro_init()
      {
      # Installs a trigger which fires if a() returns 42.
      when ( a() == 42 )
          print "expected result";

      print "Waiting for a() to return...";
      X["a"] = 42;
      }
```

**schedule**

The "schedule" statement is used to raise a specified event with specified parameters at a later time specified as an *interval*.

Example:

```
schedule 30sec { myevent(x, y, z) };
```

Note that the braces are always required (they do not indicate a *compound statement*).

Note that "schedule" is actually an expression that returns a value of type "timer", but in practice the return value is not used.

**switch**

A "switch" statement evaluates a given expression and jumps to the first "case" label which contains a matching value (the result of the expression must be type-compatible with all of the values in all of the "case" labels). If there is no matching value, then execution jumps to the "default" label instead, and if there is no "default" label then execution jumps out of the "switch" block.

Here is an example (assuming that "get_day_of_week" is a function that returns a string):

```
switch get_day_of_week()
    {
    case "Sa", "Su":
        print "weekend";
        fallthrough;
    case "Mo", "Tu", "We", "Th", "Fr":
        print "valid result";
        break;
    default:
        print "invalid result";
        break;
    }
```

A "switch" block can have any number of "case" labels, and one optional "default" label.

A "case" label can have a comma-separated list of more than one value. A value in a "case" label can be an expression, but it must be a constant expression (i.e., the expression can consist only of constants).

Each "case" and the "default" block must end with either a `break`, `fallthrough`, or `return` statement (although "return" is allowed only if the "switch" statement is inside a function, hook, or event handler).

Note that the braces in a "switch" statement are always required (these do not indicate the presence of a *compound statement*), and that no semicolon is needed at the end of a "switch" statement.

**when**

Evaluates a given expression, which must result in a value of type `bool`. When the value of the expression becomes available and if the result is true, then a specified statement is executed.

In the following example, if the expression evaluates to true, then the "print" statement is executed:

```
when ( (local x = foo()) && x == 42 )
    print x;
```

However, if a timeout is specified, and if the expression does not evaluate to true within the specified timeout interval, then the statement following the "timeout" keyword is executed:

```
when ( (local x = foo()) && x == 42 )
    print x;
timeout 5sec {
    print "timeout";
}
```

Note that when a timeout is specified the braces are always required (these do not indicate a *compound statement*).

The expression in a "when" statement can contain a declaration of a local variable but only if the declaration is written in the form "local *var* = *init*" (example: "local x = myfunction()"). This form of a local declaration is actually an expression, the result of which is always a boolean true value.

The expression in a "when" statement can contain an asynchronous function call such as `lookup_hostname` (in fact, this is the only place such a function can be called), but it can also contain an ordinary function call. When an asynchronous function call is in the expression, then Bro will continue processing statements in the script following the "when" statement, and when the result of the function call is available Bro will finish evaluating the expression in the "when" statement. See the *return* statement for an explanation of how to create an asynchronous function in a Bro script.

**while**

A "while" loop iterates over a body statement as long as a given condition remains true.

A *break* statement can be used at any time to immediately terminate the "while" loop, and a *next* statement can be used to skip to the next loop iteration.

Example:

```
local i = 0;

while ( i < 5 )
    print ++i;

while ( some_cond() )
    {
    local finish_up = F;

    if ( skip_ahead() )
        next;

    if ( finish_up )
        break;

    }
```

**compound statement** A compound statement is created by wrapping zero or more statements in braces `{ }`. Individual statements inside the braces need to be terminated by a semicolon, but a semicolon is not needed at the end (outside of the braces) of a compound statement.

A compound statement is required in order to execute more than one statement in the body of a *for*, *while*, *if*, or *when* statement.

Example:

```
if ( x == 2 ) {
    print "x is 2";
    ++x;
}
```

Note that there are other places in the Bro scripting language that use braces, but that do not indicate the presence of a compound statement (these are noted in the documentation).

**null statement** The null statement (executing it has no effect) consists of just a semicolon. This might be useful during testing or debugging a Bro script in places where a statement is required, but it is probably not useful otherwise.

Example:

```
if ( x == 2 )
    ;
```

## 3.2.5 Directives

The Bro scripting language supports a number of directives that can affect which scripts will be loaded or which lines in a script will be executed. Directives are evaluated before script execution begins.

**@DEBUG**
TODO

**@DIR**
Expands to the directory pathname where the current script is located.

Example:

```
print "Directory:", @DIR;
```

**@FILENAME**
Expands to the filename of the current script.

Example:

```
print "File:", @FILENAME;
```

**@load**
Loads the specified Bro script, specified as the relative pathname of the file (relative to one of the directories in Bro's file search path). If the Bro script filename ends with ".bro", then you don't need to specify the file extension. The filename cannot contain any whitespace.

In this example, Bro will try to load a script "policy/misc/capture-loss.bro" by looking in each directory in the file search path (the file search path can be changed by setting the BROPATH environment variable):

```
@load policy/misc/capture-loss
```

If you specify the name of a directory instead of a filename, then Bro will try to load a file in that directory called "__load__.bro" (presumably that file will contain additional "@load" directives).

In this example, Bro will try to load a file "tuning/defaults/__load__.bro" by looking in each directory in the file search path:

```
@load tuning/defaults
```

The purpose of this directive is to ensure that all script dependencies are satisfied, and to avoid having to list every needed Bro script on the command-line. Bro keeps track of which scripts have been loaded, so it is not an error to load a script more than once (once a script has been loaded, any subsequent "@load" directives for that script are ignored).

**@load-plugin**
Activate a dynamic plugin with the specified plugin name. The specified plugin must be located in Bro's plugin search path. Example:

```
@load-plugin Demo::Rot13
```

By default, Bro will automatically activate all dynamic plugins found in the plugin search path (the search path can be changed by setting the environment variable BRO_PLUGIN_PATH to a colon-separated list of directories). However, in bare mode ("bro -b"), dynamic plugins can be activated only by using "@load-plugin", or by specifying the full plugin name on the Bro command-line (e.g., “bro Demo::Rot13”), or by setting the environment variable BRO_PLUGIN_ACTIVATE to a comma-separated list of plugin names.

**@load-sigs**

This works similarly to “@load”, except that in this case the filename represents a signature file (not a Bro script). If the signature filename ends with ”.sig”, then you don’t need to specify the file extension in the “@load-sigs” directive. The filename cannot contain any whitespace.

In this example, Bro will try to load a signature file “base/protocols/ssl/dpd.sig”:

```
@load-sigs base/protocols/ssl/dpd
```

The format for a signature file is explained in the documentation for the Signature Framework.

**@unload**

This specifies a Bro script that we don’t want to load (so a subsequent attempt to load the specified script will be skipped). However, if the specified script has already been loaded, then this directive has no affect.

In the following example, if the “policy/misc/capture-loss.bro” script has not been loaded yet, then Bro will not load it:

```
@unload policy/misc/capture-loss
```

**@prefixes**

Specifies a filename prefix to use when looking for script files to load automatically. The prefix cannot contain any whitespace.

In the following example, the prefix “cluster” is used and all prefixes that were previously specified are not used:

```
@prefixes = cluster
```

In the following example, the prefix “cluster-manager” is used in addition to any previously-specified prefixes:

```
@prefixes += cluster-manager
```

The way this works is that after Bro parses all script files, then for each loaded script Bro will take the absolute path of the script and then it removes the portion of the directory path that is in Bro’s file search path. Then it replaces each “/” character with a period ”.” and then prepends the prefix (specified in the “@prefixes” directive) followed by a period. The resulting filename is searched for in each directory in Bro’s file search path. If a matching file is found, then the file is automatically loaded.

For example, if a script called “local.bro” has been loaded, and a prefix of “test” was specified, then Bro will look for a file named “test.local.bro” in each directory of Bro’s file search path.

An alternative way to specify prefixes is to use the “-p” Bro command-line option.

**@if**

The specified expression must evaluate to type *bool*. If the value is true, then the following script lines (up to the next “@else” or “@endif”) are available to be executed.

Example:

```
@if ( ver == 2 )
    print "version 2 detected";
@endif
```

**@ifdef**

This works like “@if”, except that the result is true if the specified identifier is defined.

Example:

```
@ifdef ( pi )
    print "pi is defined";
@endif
```

**@ifndef**

This works exactly like "@ifdef", except that the result is true if the specified identifier is not defined.

Example:

```
@ifndef ( pi )
    print "pi is not defined";
@endif
```

**@else**

This directive is optional after an "@if", "@ifdef", or "@ifndef". If present, it provides an else clause.

Example:

```
@ifdef ( pi )
    print "pi is defined";
@else
    print "pi is not defined";
@endif
```

**@endif**

This directive is required to terminate each "@if", "@ifdef", or "@ifndef".

### 3.2.6 Log Files

Listed below are the log files generated by Bro, including a brief description of the log file and links to descriptions of the fields for each log type.

### Network Protocols

| Log File | Description | Field Descriptions |
|---|---|---|
| conn.log | TCP/UDP/ICMP connections | `Conn::Info` |
| dhcp.log | DHCP leases | `DHCP::Info` |
| dnp3.log | DNP3 requests and replies | `DNP3::Info` |
| dns.log | DNS activity | `DNS::Info` |
| ftp.log | FTP activity | `FTP::Info` |
| http.log | HTTP requests and replies | `HTTP::Info` |
| irc.log | IRC commands and responses | `IRC::Info` |
| kerberos.log | Kerberos | `KRB::Info` |
| modbus.log | Modbus commands and responses | `Modbus::Info` |
| modbus_register_change.log | Tracks changes to Modbus holding registers | `Modbus::MemmapInfo` |
| mysql.log | MySQL | `MySQL::Info` |
| radius.log | RADIUS authentication attempts | `RADIUS::Info` |
| rdp.log | RDP | `RDP::Info` |
| rfb.log | Remote Framebuffer (RFB) | `RFB::Info` |
| sip.log | SIP | `SIP::Info` |
| smtp.log | SMTP transactions | `SMTP::Info` |
| snmp.log | SNMP messages | `SNMP::Info` |
| socks.log | SOCKS proxy requests | `SOCKS::Info` |
| ssh.log | SSH connections | `SSH::Info` |
| ssl.log | SSL/TLS handshake info | `SSL::Info` |
| syslog.log | Syslog messages | `Syslog::Info` |
| tunnel.log | Tunneling protocol events | `Tunnel::Info` |

### Files

| Log File | Description | Field Descriptions |
|---|---|---|
| files.log | File analysis results | `Files::Info` |
| pe.log | Portable Executable (PE) | `PE::Info` |
| x509.log | X.509 certificate info | `X509::Info` |

### NetControl

| Log File | Description | Field Descriptions |
|---|---|---|
| netcontrol.log | NetControl actions | `NetControl::Info` |
| netcontrol_drop.log | NetControl actions | `NetControl::DropInfo` |
| netcontrol_shunt.log | NetControl shunt actions | `NetControl::ShuntInfo` |
| netcontrol_catch_release.log | NetControl catch and release actions | `NetControl::CatchReleaseInfo` |
| openflow.log | OpenFlow debug log | `OpenFlow::Info` |

### Detection

| Log File | Description | Field Descriptions |
|---|---|---|
| intel.log | Intelligence data matches | `Intel::Info` |
| notice.log | Bro notices | `Notice::Info` |
| notice_alarm.log | The alarm stream | `Notice::ACTION_ALARM` |
| signatures.log | Signature matches | `Signatures::Info` |
| traceroute.log | Traceroute detection | `Traceroute::Info` |

#### Network Observations

| Log File | Description | Field Descriptions |
|---|---|---|
| known_certs.log | SSL certificates | `Known::CertsInfo` |
| known_devices.log | MAC addresses of devices on the network | `Known::DevicesInfo` |
| known_hosts.log | Hosts that have completed TCP handshakes | `Known::HostsInfo` |
| known_modbus.log | Modbus masters and slaves | `Known::ModbusInfo` |
| known_services.log | Services running on hosts | `Known::ServicesInfo` |
| software.log | Software being used on the network | `Software::Info` |

#### Miscellaneous

| Log File | Description | Field Descriptions |
|---|---|---|
| barnyard2.log | Alerts received from Barnyard2 | `Barnyard2::Info` |
| dpd.log | Dynamic protocol detection failures | `DPD::Info` |
| unified2.log | Interprets Snort's unified output | `Unified2::Info` |
| weird.log | Unexpected network-level activity | `Weird::Info` |

#### Bro Diagnostics

| Log File | Description | Field Descriptions |
|---|---|---|
| capture_loss.log | Packet loss rate | `CaptureLoss::Info` |
| cluster.log | Bro cluster messages | `Cluster::Info` |
| communication.log | Communication events between Bro or Broccoli instances | `Communication::Info` |
| loaded_scripts.log | Shows all scripts loaded by Bro | `LoadedScripts::Info` |
| packet_filter.log | List packet filters that were applied | `PacketFilter::Info` |
| prof.log | Profiling statistics (to create this log, load policy/misc/profiling.bro) | N/A |
| reporter.log | Internal error/warning/info messages | `Reporter::Info` |
| stats.log | Memory/event/packet/lag statistics | `Stats::Info` |
| stderr.log | Captures standard error when Bro is started from BroControl | N/A |
| stdout.log | Captures standard output when Bro is started from BroControl | N/A |

### 3.2.7 Notices

See the Bro Notice Index.

### 3.2.8 Protocol Analyzers

**Contents**

- *Bro::BitTorrent*
- *Bro::ConnSize*
- *Bro::DCE_RPC*
- *Bro::DHCP*
- *Bro::DNP3*
- *Bro::DNS*
- *Bro::File*
- *Bro::Finger*
- *Bro::FTP*
- *Bro::Gnutella*
- *Bro::GTPv1*
- *Bro::HTTP*
- *Bro::ICMP*
- *Bro::Ident*
- *Bro::InterConn*
- *Bro::IRC*
- *Bro::KRB*
- *Bro::Login*
- *Bro::MIME*
- *Bro::Modbus*
- *Bro::MySQL*
- *Bro::NCP*
- *Bro::NetBIOS*
- *Bro::NTP*
- *Bro::PIA*
- *Bro::POP3*
- *Bro::RADIUS*
- *Bro::RDP*
- *Bro::RPC*
- *Bro::SIP*
- *Bro::SNMP*
- *Bro::SMB*
- *Bro::SMTP*
- *Bro::SOCKS*
- *Bro::SSH*

**Analyzer::Tag**

**Type** *enum*

**Analyzer::ANALYZER_AYIYA**

**Analyzer::ANALYZER_BACKDOOR**

**Analyzer::ANALYZER_BITTORRENT**

**Analyzer::ANALYZER_BITTORRENTTRACKER**

**Analyzer::ANALYZER_CONNSIZE**

**Analyzer::ANALYZER_DCE_RPC**

**Analyzer::ANALYZER_CONTENTS_DCE_RPC**

**Analyzer::ANALYZER_DHCP**

**Analyzer::ANALYZER_DNP3_TCP**

**Analyzer::ANALYZER_DNP3_UDP**

**Analyzer::ANALYZER_DNS**

**Analyzer::ANALYZER_CONTENTS_DNS**

**Analyzer::ANALYZER_FTP_DATA**

**Analyzer::ANALYZER_IRC_DATA**

**Analyzer::ANALYZER_FINGER**

**Analyzer::ANALYZER_FTP**

**Analyzer::ANALYZER_FTP_ADAT**

**Analyzer::ANALYZER_GNUTELLA**

**Analyzer::ANALYZER_GTPV1**

**Analyzer::ANALYZER_HTTP**

**Analyzer::ANALYZER_ICMP**

**Analyzer::ANALYZER_IDENT**

**Analyzer::ANALYZER_INTERCONN**

**Analyzer::ANALYZER_IRC**

**Analyzer::ANALYZER_KRB**

**Analyzer::ANALYZER_KRB_TCP**

**Analyzer::ANALYZER_TELNET**

**Analyzer::ANALYZER_RSH**

**Analyzer::ANALYZER_RLOGIN**

**Analyzer::ANALYZER_NVT**

**Analyzer::ANALYZER_LOGIN**

**Analyzer::ANALYZER_CONTENTS_RSH**

**Analyzer::ANALYZER_CONTENTS_RLOGIN**

**Analyzer::ANALYZER_MODBUS**

**Analyzer::ANALYZER_MYSQL**

**Analyzer::ANALYZER_NCP**

**Analyzer::ANALYZER_CONTENTS_NCP**

**Analyzer::ANALYZER_NETBIOSSSN**

**Analyzer::ANALYZER_CONTENTS_NETBIOSSSN**

**Analyzer::ANALYZER_NTP**

**Analyzer::ANALYZER_PIA_TCP**

**Analyzer::ANALYZER_PIA_UDP**

**Analyzer::ANALYZER_POP3**

**Analyzer::ANALYZER_RADIUS**

**Analyzer::ANALYZER_RDP**

**Analyzer::ANALYZER_NFS**

**Analyzer::ANALYZER_PORTMAPPER**

**Analyzer::ANALYZER_CONTENTS_RPC**

**Analyzer::ANALYZER_CONTENTS_NFS**

**Analyzer::ANALYZER_SIP**

**Analyzer::ANALYZER_SNMP**

**Analyzer::ANALYZER_SMB**

**Analyzer::ANALYZER_CONTENTS_SMB**

**Analyzer::ANALYZER_SMTP**

**Analyzer::ANALYZER_SOCKS**

**Analyzer::ANALYZER_SSH**

**Analyzer::ANALYZER_SSL**

**Analyzer::ANALYZER_DTLS**

**Analyzer::ANALYZER_STEPPINGSTONE**

**Analyzer::ANALYZER_SYSLOG**

**Analyzer::ANALYZER_TCP**

**Analyzer::ANALYZER_TCPSTATS**

**`Analyzer::ANALYZER_CONTENTLINE`**

**`Analyzer::ANALYZER_CONTENTS`**

**`Analyzer::ANALYZER_TEREDO`**

**`Analyzer::ANALYZER_UDP`**

**`Analyzer::ANALYZER_ZIP`**

## Bro::ARP

ARP Parsing

### Components

### Events

**`arp_request`**

**Type** *`event`* (mac_src: *`string`*, mac_dst: *`string`*, SPA: *`addr`*, SHA: *`string`*, TPA: *`addr`*, THA: *`string`*)

Generated for ARP requests.

See Wikipedia for more information about the ARP protocol.

**Mac_src** The request's source MAC address.

**Mac_dst** The request's destination MAC address.

**SPA** The sender protocol address.

**SHA** The sender hardware address.

**TPA** The target protocol address.

**THA** The target hardware address.

See also:

**`arp_reply`**

**Type** *`event`* (mac_src: *`string`*, mac_dst: *`string`*, SPA: *`addr`*, SHA: *`string`*, TPA: *`addr`*, THA: *`string`*)

Generated for ARP replies.

See Wikipedia for more information about the ARP protocol.

**Mac_src** The reply's source MAC address.

**Mac_dst** The reply's destination MAC address.

**SPA** The sender protocol address.

**SHA** The sender hardware address.

**TPA** The target protocol address.

**THA** The target hardware address.

See also:

**`bad_arp`**

**Type** *event* (SPA: *addr*, SHA: *string*, TPA: *addr*, THA: *string*, explanation: *string*)

Generated for ARP packets that Bro cannot interpret. Examples are packets with non-standard hardware address formats or hardware addresses that do not match the originator of the packet.

**SPA** The sender protocol address.

**SHA** The sender hardware address.

**TPA** The target protocol address.

**THA** The target hardware address.

**Explanation** A short description of why the ARP packet is considered "bad".

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

## Bro::AYIYA

AYIYA Analyzer

### Components

*Analyzer::ANALYZER_AYIYA*

## Bro::BackDoor

Backdoor Analyzer deprecated

### Components

*Analyzer::ANALYZER_BACKDOOR*

### Events

**`backdoor_stats`**

**Type** *event* (c: `connection`, os: `backdoor_endp_stats`, rs: `backdoor_endp_stats`)

Deprecated. Will be removed.

**`backdoor_remove_conn`**

**Type** *event* (c: `connection`)

Deprecated. Will be removed.

**`ftp_signature_found`**

**Type** *event* (c: `connection`)

Deprecated. Will be removed.

**gnutella_signature_found**

**Type** *event* (c: connection)

Deprecated. Will be removed.

**http_signature_found**

**Type** *event* (c: connection)

Deprecated. Will be removed.

**irc_signature_found**

**Type** *event* (c: connection)

Deprecated. Will be removed.

**telnet_signature_found**

**Type** *event* (c: connection, is_orig: *bool*, len: *count*)

Deprecated. Will be removed.

**ssh_signature_found**

**Type** *event* (c: connection, is_orig: *bool*)

Deprecated. Will be removed.

**rlogin_signature_found**

**Type** *event* (c: connection, is_orig: *bool*, num_null: *count*, len: *count*)

Deprecated. Will be removed.

**smtp_signature_found**

**Type** *event* (c: connection)

Deprecated. Will be removed.

**http_proxy_signature_found**

**Type** *event* (c: connection)

Deprecated. Will be removed.

### Bro::BitTorrent

BitTorrent Analyzer

#### Components

*Analyzer::ANALYZER_BITTORRENT*

*Analyzer::ANALYZER_BITTORRENTTRACKER*

### Events

**`bittorrent_peer_handshake`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, reserved: *`string`*, info_hash: *`string`*, peer_id: *`string`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_keep_alive`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_choke`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_unchoke`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_interested`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_not_interested`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_have`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, piece_index: *`count`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_bitfield`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, bitfield: *`string`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_request`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, index: *`count`*, begin: *`count`*, length: *`count`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_piece`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, index: *`count`*, begin: *`count`*, piece_length: *`count`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_cancel`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, index: *`count`*, begin: *`count`*, length: *`count`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_port`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, listen_port: *`port`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_unknown`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, message_id: *`count`*, data: *`string`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**`bittorrent_peer_weird`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, msg: *`string`*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**bt_tracker_request**

**Type** *event* (c: connection, uri: *string*, headers: bt_tracker_headers)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**bt_tracker_response**

**Type** *event* (c: connection, status: *count*, headers: bt_tracker_headers, peers: bittorrent_peer_set, benc: bittorrent_benc_dir)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**bt_tracker_response_not_ok**

**Type** *event* (c: connection, status: *count*, headers: bt_tracker_headers)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

**bt_tracker_weird**

**Type** *event* (c: connection, is_orig: *bool*, msg: *string*)

TODO.

See Wikipedia for more information about the BitTorrent protocol.

See also:

## Bro::ConnSize

Connection size analyzer

### Components

*Analyzer::ANALYZER_CONNSIZE*

### Events

**conn_bytes_threshold_crossed**

**Type** *event* (c: connection, threshold: *count*, is_orig: *bool*)

Generated for a connection that crossed a set byte threshold. Note that this is a low level event that should usually be avoided for user code. Use ConnThreshold::bytes_threshold_crossed instead.

**C** the connection

**Threshold** the threshold that was set

**Is_orig** true if the threshold was crossed by the originator of the connection

See also:

**`conn_packets_threshold_crossed`**

**Type** *`event`* (c: `connection`, threshold: *`count`*, is_orig: *`bool`*)

Generated for a connection that crossed a set packet threshold. Note that this is a low level event that should usually be avoided for user code. Use ConnThreshold::bytes_threshold_crossed instead.

**C** the connection

**Threshold** the threshold that was set

**Is_orig** true if the threshold was crossed by the originator of the connection

See also:

### Functions

**`set_current_conn_bytes_threshold`**

**Type** *`function`* (cid: `conn_id`, threshold: *`count`*, is_orig: *`bool`*) : *`bool`*

Sets the current byte threshold for connection sizes, overwriting any potential old threshold. Be aware that in nearly any case you will want to use the high level API instead (ConnThreshold::set_bytes_threshold).

**Cid** The connection id.

**Threshold** Threshold in bytes.

**Is_orig** If true, threshold is set for bytes from originator, otherwhise for bytes from responder.

See also:

**`set_current_conn_packets_threshold`**

**Type** *`function`* (cid: `conn_id`, threshold: *`count`*, is_orig: *`bool`*) : *`bool`*

Sets a threshold for connection packets, overwtiting any potential old thresholds. Be aware that in nearly any case you will want to use the high level API instead (ConnThreshold::set_packets_threshold).

**Cid** The connection id.

**Threshold** Threshold in packets.

**Is_orig** If true, threshold is set for packets from originator, otherwhise for packets from responder.

See also:

**`get_current_conn_bytes_threshold`**

**Type** *`function`* (cid: `conn_id`, is_orig: *`bool`*) : *`count`*

Gets the current byte threshold size for a connection.

**Cid** The connection id.

**Is_orig** If true, threshold of originator, otherwhise threshold of responder.

**Returns** 0 if no threshold is set or the threshold in bytes

See also:

**`get_current_conn_packets_threshold`**

> **Type** *`function`* (cid: `conn_id`, is_orig: *`bool`*) : *`count`*

Gets the current packet threshold size for a connection.

> **Cid** The connection id.
>
> **Is_orig** If true, threshold of originator, otherwise threshold of responder.
>
> **Returns** 0 if no threshold is set or the threshold in packets

See also:

### Bro::DCE_RPC

DCE-RPC analyzer

### Components

*`Analyzer::ANALYZER_CONTENTS_DCE_RPC`*

*`Analyzer::ANALYZER_DCE_RPC`*

### Events

**`dce_rpc_message`**

> **Type** *`event`* (c: `connection`, is_orig: *`bool`*, ptype: `dce_rpc_ptype`, msg: *`string`*)

TODO.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`dce_rpc_bind`**

> **Type** *`event`* (c: `connection`, uuid: *`string`*)

TODO.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`dce_rpc_request`**

> **Type** *`event`* (c: `connection`, opnum: *`count`*, stub: *`string`*)

TODO.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**dce_rpc_response**

**Type** *event* (c: `connection`, opnum: *count*, stub: *string*)

TODO.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**epm_map_response**

**Type** *event* (c: `connection`, uuid: *string*, p: *port*, h: *addr*)

TODO.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

## Bro::DHCP

DHCP analyzer

### Components

*Analyzer::ANALYZER_DHCP*

### Events

**dhcp_discover**

**Type** *event* (c: `connection`, msg: `dhcp_msg`, req_addr: *addr*, host_name: *string*)

Generated for DHCP messages of type *DHCPDISCOVER* (client broadcast to locate available servers).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Req_addr** The specific address requested by the client.

**Host_name** The value of the host name option, if specified by the client.

See also:

---

**Note:** Bro does not support broadcast packets (as used by the DHCP protocol). It treats broadcast addresses just like any other and associates packets into transport-level flows in the same way as usual.

---

### `dhcp_offer`

**Type** *`event`* (c: `connection`, msg: `dhcp_msg`, mask: *`addr`*, router: `dhcp_router_list`, lease: *`interval`*, serv_addr: *`addr`*, host_name: *`string`*)

Generated for DHCP messages of type *DHCPOFFER* (server to client in response to DHCPDISCOVER with offer of configuration parameters).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Mask** The subnet mask specified by the message.

**Router** The list of routers specified by the message.

**Lease** The least interval specified by the message.

**Serv_addr** The server address specified by the message.

**Host_name** Optional host name value. May differ from the host name requested from the client.

See also:

---

**Note:** Bro does not support broadcast packets (as used by the DHCP protocol). It treats broadcast addresses just like any other and associates packets into transport-level flows in the same way as usual.

---

### `dhcp_request`

**Type** *`event`* (c: `connection`, msg: `dhcp_msg`, req_addr: *`addr`*, serv_addr: *`addr`*, host_name: *`string`*)

Generated for DHCP messages of type *DHCPREQUEST* (Client message to servers either (a) requesting offered parameters from one server and implicitly declining offers from all others, (b) confirming correctness of previously allocated address after, e.g., system reboot, or (c) extending the lease on a particular network address.)

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Req_addr** The client address specified by the message.

**Serv_addr** The server address specified by the message.

**Host_name** The value of the host name option, if specified by the client.

See also:

**Note:** Bro does not support broadcast packets (as used by the DHCP protocol). It treats broadcast addresses just like any other and associates packets into transport-level flows in the same way as usual.

**dhcp_decline**

**Type** *event* (c: `connection`, msg: `dhcp_msg`, host_name: *string*)

Generated for DHCP messages of type *DHCPDECLINE* (Client to server indicating network address is already in use).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Host_name** Optional host name value.

See also:

**Note:** Bro does not support broadcast packets (as used by the DHCP protocol). It treats broadcast addresses just like any other and associates packets into transport-level flows in the same way as usual.

**dhcp_ack**

**Type** *event* (c: `connection`, msg: `dhcp_msg`, mask: *addr*, router: `dhcp_router_list`, lease: *interval*, serv_addr: *addr*, host_name: *string*)

Generated for DHCP messages of type *DHCPACK* (Server to client with configuration parameters, including committed network address).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Mask** The subnet mask specified by the message.

**Router** The list of routers specified by the message.

**Lease** The least interval specified by the message.

**Serv_addr** The server address specified by the message.

**Host_name** Optional host name value. May differ from the host name requested from the client.

See also:

**dhcp_nak**

**Type** *event* (c: `connection`, msg: `dhcp_msg`, host_name: *string*)

Generated for DHCP messages of type *DHCPNAK* (Server to client indicating client's notion of network address is incorrect (e.g., client has moved to new subnet) or client's lease has expired).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Host_name** Optional host name value.

See also:

**Note:** Bro does not support broadcast packets (as used by the DHCP protocol). It treats broadcast addresses just like any other and associates packets into transport-level flows in the same way as usual.

**`dhcp_release`**

**Type** *`event`* (c: `connection`, msg: `dhcp_msg`, host_name: *`string`*)

Generated for DHCP messages of type *DHCPRELEASE* (Client to server relinquishing network address and cancelling remaining lease).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Host_name** The value of the host name option, if specified by the client.

See also:

**`dhcp_inform`**

**Type** *`event`* (c: `connection`, msg: `dhcp_msg`, host_name: *`string`*)

Generated for DHCP messages of type *DHCPINFORM* (Client to server, asking only for local configuration parameters; client already has externally configured network address).

**C** The connection record describing the underlying UDP flow.

**Msg** The parsed type-independent part of the DHCP message.

**Host_name** The value of the host name option, if specified by the client.

See also:

**Note:** Bro does not support broadcast packets (as used by the DHCP protocol). It treats broadcast addresses just like any other and associates packets into transport-level flows in the same way as usual.

### Bro::DNP3

DNP3 UDP/TCP analyzers

#### Components

*`Analyzer::ANALYZER_DNP3_TCP`*

*`Analyzer::ANALYZER_DNP3_UDP`*

#### Events

**`dnp3_application_request_header`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, application: *`count`*, fc: *`count`*)

Generated for a DNP3 request header.

**C** The connection the DNP3 communication is part of.

**Is_orig** True if this reflects originator-side activity.

**Fc** function code.

**`dnp3_application_response_header`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, application: *`count`*, fc: *`count`*, iin: *`count`*)

Generated for a DNP3 response header.

**C** The connection the DNP3 communication is part of.

**Is_orig** True if this reflects originator-side activity.

**Fc** function code.

**Iin** internal indication number.

**`dnp3_object_header`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, obj_type: *`count`*, qua_field: *`count`*, number: *`count`*, rf_low: *`count`*, rf_high: *`count`*)

Generated for the object header found in both DNP3 requests and responses.

**C** The connection the DNP3 communication is part of.

**Is_orig** True if this reflects originator-side activity.

**Obj_type** type of object, which is classified based on an 8-bit group number and an 8-bit variation number.

**Qua_field** qualifier field.

**Number** TODO.

**Rf_low** the structure of the range field depends on the qualified field. In some cases, the range field contains only one logic part, e.g., number of objects, so only *rf_low* contains useful values.

**Rf_high** in some cases, the range field contains two logic parts, e.g., start index and stop index, so *rf_low* contains the start index while *rf_high* contains the stop index.

**`dnp3_object_prefix`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix_value: *`count`*)

Generated for the prefix before a DNP3 object. The structure and the meaning of the prefix are defined by the qualifier field.

**C** The connection the DNP3 communication is part of.

**Is_orig** True if this reflects originator-side activity.

**Prefix_value** The prefix.

**`dnp3_header_block`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, start: *`count`*, len: *`count`*, ctrl: *`count`*, dest_addr: *`count`*, src_addr: *`count`*)

Generated for an additional header that the DNP3 analyzer passes to the script-level. This header mimics the DNP3 transport-layer yet is only passed once for each sequence of DNP3 records (which are otherwise reassembled and treated as a single entity).

**C** The connection the DNP3 communication is part of.

**Is_orig** True if this reflects originator-side activity.

**Start** the first two bytes of the DNP3 Pseudo Link Layer; its value is fixed as 0x0564.

**Len** the "length" field in the DNP3 Pseudo Link Layer.

**Ctrl** the "control" field in the DNP3 Pseudo Link Layer.

**Dest_addr** the "destination" field in the DNP3 Pseudo Link Layer.

**Src_addr** the "source" field in the DNP3 Pseudo Link Layer.

**`dnp3_response_data_object`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, data_value: *`count`*)

Generated for a DNP3 "Response_Data_Object". The "Response_Data_Object" contains two parts: object prefix and object data. In most cases, object data are defined by new record types. But in a few cases, object data are directly basic types, such as int16, or int8; thus we use an additional *data_value* to record the values of those object data.

**C** The connection the DNP3 communication is part of.

**Is_orig** True if this reflects originator-side activity.

**Data_value** The value for those objects that carry their information here directly.

**`dnp3_attribute_common`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, data_type_code: *`count`*, leng: *`count`*, attribute_obj: *`string`*)

Generated for DNP3 attributes.

**`dnp3_crob`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, control_code: *`count`*, count8: *`count`*, on_time: *`count`*, off_time: *`count`*, status_code: *`count`*)

Generated for DNP3 objects with the group number 12 and variation number 1

**CROB** control relay output block

**`dnp3_pcb`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, control_code: *`count`*, count8: *`count`*, on_time: *`count`*, off_time: *`count`*, status_code: *`count`*)

Generated for DNP3 objects with the group number 12 and variation number 2

**PCB** Pattern Control Block

**`dnp3_counter_32wFlag`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, flag: *`count`*, count_value: *`count`*)

Generated for DNP3 objects with the group number 20 and variation number 1 counter 32 bit with flag

**`dnp3_counter_16wFlag`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, flag: *`count`*, count_value: *`count`*)

Generated for DNP3 objects with the group number 20 and variation number 2 counter 16 bit with flag

**`dnp3_counter_32woFlag`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, count_value: *`count`*)

Generated for DNP3 objects with the group number 20 and variation number 5 counter 32 bit without flag

**`dnp3_counter_16woFlag`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, count_value: *`count`*)

Generated for DNP3 objects with the group number 20 and variation number 6 counter 16 bit without flag

**dnp3_frozen_counter_32wFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, count_value: *count*)

Generated for DNP3 objects with the group number 21 and variation number 1 frozen counter 32 bit with flag

**dnp3_frozen_counter_16wFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, count_value: *count*)

Generated for DNP3 objects with the group number 21 and variation number 2 frozen counter 16 bit with flag

**dnp3_frozen_counter_32wFlagTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, count_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 21 and variation number 5 frozen counter 32 bit with flag and time

**dnp3_frozen_counter_16wFlagTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, count_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 21 and variation number 6 frozen counter 16 bit with flag and time

**dnp3_frozen_counter_32woFlag**

**Type** *event* (c: connection, is_orig: *bool*, count_value: *count*)

Generated for DNP3 objects with the group number 21 and variation number 9 frozen counter 32 bit without flag

**dnp3_frozen_counter_16woFlag**

**Type** *event* (c: connection, is_orig: *bool*, count_value: *count*)

Generated for DNP3 objects with the group number 21 and variation number 10 frozen counter 16 bit without flag

**dnp3_analog_input_32wFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*)

Generated for DNP3 objects with the group number 30 and variation number 1 analog input 32 bit with flag

**dnp3_analog_input_16wFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*)

Generated for DNP3 objects with the group number 30 and variation number 2 analog input 16 bit with flag

**dnp3_analog_input_32woFlag**

**Type** *event* (c: connection, is_orig: *bool*, value: *count*)

Generated for DNP3 objects with the group number 30 and variation number 3 analog input 32 bit without flag

**dnp3_analog_input_16woFlag**

**Type** *event* (c: connection, is_orig: *bool*, value: *count*)

Generated for DNP3 objects with the group number 30 and variation number 4 analog input 16 bit without flag

**dnp3_analog_input_SPwFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*)

Generated for DNP3 objects with the group number 30 and variation number 5 analog input single precision, float point with flag

**dnp3_analog_input_DPwFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value_low: *count*, value_high: *count*)

Generated for DNP3 objects with the group number 30 and variation number 6 analog input double precision, float point with flag

**dnp3_frozen_analog_input_32wFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*)

Generated for DNP3 objects with the group number 31 and variation number 1 frozen analog input 32 bit with flag

**dnp3_frozen_analog_input_16wFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*)

Generated for DNP3 objects with the group number 31 and variation number 2 frozen analog input 16 bit with flag

**dnp3_frozen_analog_input_32wTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 31 and variation number 3 frozen analog input 32 bit with time-of-freeze

**dnp3_frozen_analog_input_16wTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 31 and variation number 4 frozen analog input 16 bit with time-of-freeze

**dnp3_frozen_analog_input_32woFlag**

**Type** *event* (c: connection, is_orig: *bool*, frozen_value: *count*)

Generated for DNP3 objects with the group number 31 and variation number 5 frozen analog input 32 bit without flag

**dnp3_frozen_analog_input_16woFlag**

**Type** *event* (c: connection, is_orig: *bool*, frozen_value: *count*)

Generated for DNP3 objects with the group number 31 and variation number 6 frozen analog input 16 bit without flag

**dnp3_frozen_analog_input_SPwFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*)

Generated for DNP3 objects with the group number 31 and variation number 7 frozen analog input single-precision, float point with flag

**dnp3_frozen_analog_input_DPwFlag**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value_low: *count*, frozen_value_high: *count*)

Generated for DNP3 objects with the group number 31 and variation number 8 frozen analog input double-precision, float point with flag

**dnp3_analog_input_event_32woTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*)

Generated for DNP3 objects with the group number 32 and variation number 1 analog input event 32 bit without time

**dnp3_analog_input_event_16woTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*)

Generated for DNP3 objects with the group number 32 and variation number 2 analog input event 16 bit without time

**dnp3_analog_input_event_32wTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 32 and variation number 3 analog input event 32 bit with time

**dnp3_analog_input_event_16wTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 32 and variation number 4 analog input event 16 bit with time

**dnp3_analog_input_event_SPwoTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*)

Generated for DNP3 objects with the group number 32 and variation number 5 analog input event single-precision float point without time

**dnp3_analog_input_event_DPwoTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value_low: *count*, value_high: *count*)

Generated for DNP3 objects with the group number 32 and variation number 6 analog input event double-precision float point without time

**dnp3_analog_input_event_SPwTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 32 and variation number 7 analog input event single-precision float point with time

**dnp3_analog_input_event_DPwTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, value_low: *count*, value_high: *count*, time48: *count*)

Generated for DNP3 objects with the group number 32 and variation number 8 analog input event double-precisiion float point with time

**dnp3_frozen_analog_input_event_32woTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*)

Generated for DNP3 objects with the group number 33 and variation number 1 frozen analog input event 32 bit without time

**dnp3_frozen_analog_input_event_16woTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*)

Generated for DNP3 objects with the group number 33 and variation number 2 frozen analog input event 16 bit without time

**dnp3_frozen_analog_input_event_32wTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 33 and variation number 3 frozen analog input event 32 bit with time

**dnp3_frozen_analog_input_event_16wTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 33 and variation number 4 frozen analog input event 16 bit with time

**dnp3_frozen_analog_input_event_SPwoTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*)

Generated for DNP3 objects with the group number 33 and variation number 5 frozen analog input event single-precision float point without time

**dnp3_frozen_analog_input_event_DPwoTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value_low: *count*, frozen_value_high: *count*)

Generated for DNP3 objects with the group number 33 and variation number 6 frozen analog input event double-precision float point without time

**dnp3_frozen_analog_input_event_SPwTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value: *count*, time48: *count*)

Generated for DNP3 objects with the group number 33 and variation number 7 frozen analog input event single-precision float point with time

**dnp3_frozen_analog_input_event_DPwTime**

**Type** *event* (c: connection, is_orig: *bool*, flag: *count*, frozen_value_low: *count*, frozen_value_high: *count*, time48: *count*)

Generated for DNP3 objects with the group number 34 and variation number 8 frozen analog input event double-precision float point with time

**dnp3_file_transport**

**Type** *event* (c: connection, is_orig: *bool*, file_handle: *count*, block_num: *count*, file_data: *string*)

g70

**dnp3_debug_byte**

**Type** *event* (c: connection, is_orig: *bool*, debug: *string*)

Debugging event generated by the DNP3 analyzer. The “Debug_Byte” binpac unit generates this for unknown “cases”. The user can use it to debug the byte string to check what caused the malformed network packets.

### Bro::DNS

DNS analyzer

#### Components

*Analyzer::ANALYZER_CONTENTS_DNS*

*Analyzer::ANALYZER_DNS*

#### Events

**dns_message**

**Type** *event* (c: `connection`, is_orig: *bool*, msg: `dns_msg`, len: *count*)

Generated for all DNS messages.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Is_orig** True if the message was sent by the originator of the connection.

**Msg** The parsed DNS message header.

**Len** The length of the message's raw representation (i.e., the DNS payload).

See also:

**dns_request**

**Type** *event* (c: `connection`, msg: `dns_msg`, query: *string*, qtype: *count*, qclass: *count*)

Generated for DNS requests. For requests with multiple queries, this event is raised once for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Query** The queried name.

**Qtype** The queried resource record type.

**Qclass** The queried resource record class.

See also:

**dns_rejected**

**Type** *event* (c: `connection`, msg: `dns_msg`, query: *string*, qtype: *count*, qclass: *count*)

Generated for DNS replies that reject a query. This event is raised if a DNS reply indicates failure because it does not pass on any answers to a query. Note that all of the event's parameters are parsed out of the reply; there's no stateful correlation with the query.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Query** The queried name.

**Qtype** The queried resource record type.

**Qclass** The queried resource record class.

See also:

**`dns_query_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, query: *`string`*, qtype: *`count`*, qclass: *`count`*)

Generated for each entry in the Question section of a DNS reply.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Query** The queried name.

**Qtype** The queried resource record type.

**Qclass** The queried resource record class.

See also:

**`dns_A_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, a: *`addr`*)

Generated for DNS replies of type *A*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**A** The address returned by the reply.

See also:

**`dns_AAAA_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, a: *`addr`*)

Generated for DNS replies of type *AAAA*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**A** The address returned by the reply.

See also:

**dns_A6_reply**

**Type** `event` (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, a: `addr`)

Generated for DNS replies of type *A6*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**A** The address returned by the reply.

See also:

**dns_NS_reply**

**Type** `event` (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, name: `string`)

Generated for DNS replies of type *NS*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Name** The name returned by the reply.

See also:

**dns_CNAME_reply**

**Type** `event` (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, name: `string`)

Generated for DNS replies of type *CNAME*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Name** The name returned by the reply.

See also:

**dns_PTR_reply**

**Type** `event` (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, name: `string`)

Generated for DNS replies of type *PTR*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Name** The name returned by the reply.

See also:

**`dns_SOA_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, soa: `dns_soa`)

Generated for DNS replies of type *CNAME*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Soa** The parsed SOA value.

See also:

**`dns_WKS_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`)

Generated for DNS replies of type *WKS*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

See also:

**`dns_HINFO_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`)

Generated for DNS replies of type *HINFO*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

See also:

**`dns_MX_reply`**

**Type** *event* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, name: *string*, preference: *count*)

Generated for DNS replies of type *MX*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Name** The name returned by the reply.

**Preference** The preference for *name* specified by the reply.

See also:

**`dns_TXT_reply`**

**Type** *event* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, strs: `string_vec`)

Generated for DNS replies of type *TXT*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Strs** The textual information returned by the reply.

See also:

**`dns_SRV_reply`**

**Type** *event* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`, target: *string*, priority: *count*, weight: *count*, p: *count*)

Generated for DNS replies of type *SRV*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

**Target** Target of the SRV response – the canonical hostname of the machine providing the service, ending in a dot.

**Priority** Priority of the SRV response – the priority of the target host, lower value means more preferred.

**Weight** Weight of the SRV response – a relative weight for records with the same priority, higher value means more preferred.

**P** Port of the SRV response – the TCP or UDP port on which the service is to be found.

See also:

**`dns_unknown_reply`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_answer`)

Generated on DNS reply resource records when the type of record is not one that Bro knows how to parse and generate another more specific event.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The type-independent part of the parsed answer record.

See also:

**`dns_EDNS_addl`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_edns_additional`)

Generated for DNS replies of type *EDNS*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The parsed EDNS reply.

See also:

**`dns_TSIG_addl`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`, ans: `dns_tsig_additional`)

Generated for DNS replies of type *TSIG*. For replies with multiple answers, an individual event of the corresponding type is raised for each.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

**Ans** The parsed TSIG reply.

See also:

**`dns_end`**

**Type** *`event`* (c: `connection`, msg: `dns_msg`)

Generated at the end of processing a DNS packet. This event is the last `dns_*` event that will be raised for a DNS query/reply and signals that all resource records have been passed on.

See Wikipedia for more information about the DNS protocol. Bro analyzes both UDP and TCP DNS sessions.

**C** The connection, which may be UDP or TCP depending on the type of the transport-layer session being analyzed.

**Msg** The parsed DNS message header.

See also:

**`dns_full_request`**

**Type** *`event`* ()

Deprecated. Will be removed.

**Todo**

Unclear what this event is for; it's never raised. We should just remove it.

**`non_dns_request`**

**Type** *`event`* (c: `connection`, msg: *`string`*)

**Msg** The raw DNS payload.

**Note:** This event is deprecated and superseded by Bro's dynamic protocol detection framework.

## Bro::File

Generic file analyzer

### Components

*`Analyzer::ANALYZER_FTP_DATA`*

*`Analyzer::ANALYZER_IRC_DATA`*

### Events

**`file_transferred`**

**Type** *`event`* (c: `connection`, prefix: *`string`*, descr: *`string`*, mime_type: *`string`*)

Generated when a TCP connection associated w/ file data transfer is seen (e.g. as happens w/ FTP or IRC).

**C** The connection over which file data is transferred.

**Prefix** Up to 1024 bytes of the file data.

**Descr** Deprecated/unused argument.

**Mime_type** MIME type of the file or "<unknown>" if no file magic signatures matched.

## Bro::Finger

Finger analyzer

### Components

*`Analyzer::ANALYZER_FINGER`*

### Events

**`finger_request`**

**Type** *`event`* (c: `connection`, full: *`bool`*, username: *`string`*, hostname: *`string`*)

Generated for Finger requests.

See Wikipedia for more information about the Finger protocol.

**C** The connection.

**Full** True if verbose information is requested (`/W` switch).

**Username** The request's user name.

**Hostname** The request's host name.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`finger_reply`**

**Type** *`event`* (c: `connection`, reply_line: *`string`*)

Generated for Finger replies.

See Wikipedia for more information about the Finger protocol.

**C** The connection.

**Reply_line** The reply as returned by the server

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### Bro::FTP

FTP analyzer

### Components

*`Analyzer::ANALYZER_FTP`*

*`Analyzer::ANALYZER_FTP_ADAT`*

### Types

**`ftp_port`**

**Type** *`record`*

**h:** ***`addr`*** The host's address.

**p:** ***`port`*** The host's port.

**valid:** ***`bool`*** True if format was right. Only then are *h* and *p* valid.

A parsed host/port combination describing server endpoint for an upcoming data transfer.

See also:

### Events

**`ftp_request`**

**Type** *`event`* (c: `connection`, command: *`string`*, arg: *`string`*)

Generated for client-side FTP commands.

See Wikipedia for more information about the FTP protocol.

**C** The connection.

**Command** The FTP command issued by the client (without any arguments).

**Arg** The arguments going with the command.

See also:

**`ftp_reply`**

**Type** *`event`* (c: `connection`, code: *`count`*, msg: *`string`*, cont_resp: *`bool`*)

Generated for server-side FTP replies.

See Wikipedia for more information about the FTP protocol.

**C** The connection.

**Code** The numerical response code the server responded with.

**Msg** The textual message of the response.

**Cont_resp** True if the reply line is tagged as being continued to the next line. If so, further events will be raised and a handler may want to reassemble the pieces before processing the response any further.

See also:

### Functions

**`parse_ftp_port`**

**Type** *`function`* (s: *`string`*) : *`ftp_port`*

Converts a string representation of the FTP PORT command to an *`ftp_port`*.

**S** The string of the FTP PORT command, e.g., `"10,0,0,1,4,31"`.

**Returns** The FTP PORT, e.g., `[h=10.0.0.1,p=1055/tcp,valid=T]`.

See also:

**parse_eftp_port**

**Type** *function* (s: *string*) : *ftp_port*

Converts a string representation of the FTP EPRT command (see **RFC 2428**) to an *ftp_port*. The format is `"EPRT<space><d><net-prt><d><net-addr><d><tcp-port><d>"`, where `<d>` is a delimiter in the ASCII range 33-126 (usually `|`).

**S** The string of the FTP EPRT command, e.g., `"|1|10.0.0.1|1055|"`.

**Returns** The FTP PORT, e.g., `[h=10.0.0.1,p=1055/tcp,valid=T]`.

See also:

**parse_ftp_pasv**

**Type** *function* (str: *string*) : *ftp_port*

Converts the result of the FTP PASV command to an *ftp_port*.

**Str** The string containing the result of the FTP PASV command.

**Returns** The FTP PORT, e.g., `[h=10.0.0.1,p=1055/tcp,valid=T]`.

See also:

**parse_ftp_epsv**

**Type** *function* (str: *string*) : *ftp_port*

Converts the result of the FTP EPSV command (see **RFC 2428**) to an *ftp_port*. The format is `"<text> (<d><d><d><tcp-port><d>)"`, where `<d>` is a delimiter in the ASCII range 33-126 (usually `|`).

**Str** The string containing the result of the FTP EPSV command.

**Returns** The FTP PORT, e.g., `[h=10.0.0.1,p=1055/tcp,valid=T]`.

See also:

**fmt_ftp_port**

**Type** *function* (a: *addr*, p: *port*) : *string*

Formats an IP address and TCP port as an FTP PORT command. For example, `10.0.0.1` and `1055/tcp` yields `"10,0,0,1,4,31"`.

**A** The IP address.

**P** The TCP port.

**Returns** The FTP PORT string.

See also:

## Bro::Gnutella

Gnutella analyzer

## Components

*Analyzer::ANALYZER_GNUTELLA*

### Events

**gnutella_text_msg**

**Type** *event* (c: connection, orig: *bool*, headers: *string*)

TODO.

See Wikipedia for more information about the Gnutella protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**gnutella_binary_msg**

**Type** *event* (c: connection, orig: *bool*, msg_type: *count*, ttl: *count*, hops: *count*, msg_len: *count*, payload: *string*, payload_len: *count*, trunc: *bool*, complete: *bool*)

TODO.

See Wikipedia for more information about the Gnutella protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**gnutella_partial_binary_msg**

**Type** *event* (c: connection, orig: *bool*, msg: *string*, len: *count*)

TODO.

See Wikipedia for more information about the Gnutella protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**gnutella_establish**

**Type** *event* (c: connection)

TODO.

See Wikipedia for more information about the Gnutella protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**gnutella_not_establish**

**Type** *event* (c: `connection`)

TODO.

See Wikipedia for more information about the Gnutella protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**gnutella_http_notify**

**Type** *event* (c: `connection`)

TODO.

See Wikipedia for more information about the Gnutella protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### Bro::GTPv1

GTPv1 analyzer

#### Components

*Analyzer::ANALYZER_GTPV1*

#### Events

**gtpv1_message**

**Type** *event* (c: `connection`, hdr: `gtpv1_hdr`)

Generated for any GTP message with a GTPv1 header.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**`gtpv1_g_pdu_packet`**

**Type** *`event`* (outer: `connection`, inner_gtp: `gtpv1_hdr`, inner_ip: `pkt_hdr`)

Generated for GTPv1 G-PDU packets. That is, packets with a UDP payload that includes a GTP header followed by an IPv4 or IPv6 packet.

**Outer** The GTP outer tunnel connection.

**Inner_gtp** The GTP header.

**Inner_ip** The inner IP and transport layer packet headers.

**Note:** Since this event may be raised on a per-packet basis, handling it may become particularly expensive for real-time analysis.

**`gtpv1_create_pdp_ctx_request`**

**Type** *`event`* (c: `connection`, hdr: `gtpv1_hdr`, elements: `gtp_create_pdp_ctx_request_elements`)

Generated for GTPv1-C Create PDP Context Request messages.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**Elements** The set of Information Elements comprising the message.

**`gtpv1_create_pdp_ctx_response`**

**Type** *`event`* (c: `connection`, hdr: `gtpv1_hdr`, elements: `gtp_create_pdp_ctx_response_elements`)

Generated for GTPv1-C Create PDP Context Response messages.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**Elements** The set of Information Elements comprising the message.

**`gtpv1_update_pdp_ctx_request`**

**Type** *`event`* (c: `connection`, hdr: `gtpv1_hdr`, elements: `gtp_update_pdp_ctx_request_elements`)

Generated for GTPv1-C Update PDP Context Request messages.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**Elements** The set of Information Elements comprising the message.

**`gtpv1_update_pdp_ctx_response`**

**Type** *`event`* (c: `connection`, hdr: `gtpv1_hdr`, elements: `gtp_update_pdp_ctx_response_elements`)

Generated for GTPv1-C Update PDP Context Response messages.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**Elements** The set of Information Elements comprising the message.

**gtpv1_delete_pdp_ctx_request**

**Type** *event* (c: `connection`, hdr: `gtpv1_hdr`, elements: `gtp_delete_pdp_ctx_request_elements`)

Generated for GTPv1-C Delete PDP Context Request messages.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**Elements** The set of Information Elements comprising the message.

**gtpv1_delete_pdp_ctx_response**

**Type** *event* (c: `connection`, hdr: `gtpv1_hdr`, elements: `gtp_delete_pdp_ctx_response_elements`)

Generated for GTPv1-C Delete PDP Context Response messages.

**C** The connection over which the message is sent.

**Hdr** The GTPv1 header.

**Elements** The set of Information Elements comprising the message.

### Bro::HTTP

HTTP analyzer

#### Components

*Analyzer::ANALYZER_HTTP*

#### Events

**http_request**

**Type** *event* (c: `connection`, method: *string*, original_URI: *string*, unescaped_URI: *string*, version: *string*)

Generated for HTTP requests. Bro supports persistent and pipelined HTTP sessions and raises corresponding events as it parses client/server dialogues. This event is generated as soon as a request's initial line has been parsed, and before any *http_header* events are raised.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Method** The HTTP method extracted from the request (e.g., `GET`, `POST`).

**Original_URI** The unprocessed URI as specified in the request.

**Unescaped_URI** The URI with all percent-encodings decoded.

**Version** The version number specified in the request (e.g., `1.1`).

See also:

**`http_reply`**

**Type** *`event`* (c: `connection`, version: *`string`*, code: *`count`*, reason: *`string`*)

Generated for HTTP replies. Bro supports persistent and pipelined HTTP sessions and raises corresponding events as it parses client/server dialogues. This event is generated as soon as a reply's initial line has been parsed, and before any *`http_header`* events are raised.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Version** The version number specified in the reply (e.g., `1.1`).

**Code** The numerical response code returned by the server.

**Reason** The textual description returned by the server along with *code*.

See also:

**`http_header`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, name: *`string`*, value: *`string`*)

Generated for HTTP headers. Bro supports persistent and pipelined HTTP sessions and raises corresponding events as it parses client/server dialogues.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the header was sent by the originator of the TCP connection.

**Name** The name of the header.

**Value** The value of the header.

See also:

**Note:** This event is also raised for headers found in nested body entities.

**`http_all_headers`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, hlist: `mime_header_list`)

Generated for HTTP headers, passing on all headers of an HTTP message at once. Bro supports persistent and pipelined HTTP sessions and raises corresponding events as it parses client/server dialogues.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the header was sent by the originator of the TCP connection.

**Hlist** A *table* containing all headers extracted from the current entity. The table is indexed by the position of the header (1 for the first, 2 for the second, etc.).

See also:

**Note:** This event is also raised for headers found in nested body entities.

**`http_begin_entity`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

Generated when starting to parse an HTTP body entity. This event is generated at least once for each non-empty (client or server) HTTP body; and potentially more than once if the body contains further nested MIME entities. Bro raises this event just before it starts parsing each entity's content.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the entity was sent by the originator of the TCP connection.

See also:

`http_end_entity`

**Type** *event* (c: `connection`, is_orig: *bool*)

Generated when finishing parsing an HTTP body entity. This event is generated at least once for each non-empty (client or server) HTTP body; and potentially more than once if the body contains further nested MIME entities. Bro raises this event at the point when it has finished parsing an entity's content.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the entity was sent by the originator of the TCP connection.

See also:

`http_entity_data`

**Type** *event* (c: `connection`, is_orig: *bool*, length: *count*, data: *string*)

Generated when parsing an HTTP body entity, passing on the data. This event can potentially be raised many times for each entity, each time passing a chunk of the data of not further defined size.

A common idiom for using this event is to first *reassemble* the data at the scripting layer by concatenating it to a successively growing string; and only perform further content analysis once the corresponding *http_end_entity* event has been raised. Note, however, that doing so can be quite expensive for HTTP tranders. At the very least, one should impose an upper size limit on how much data is being buffered.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the entity was sent by the originator of the TCP connection.

**Length** The length of *data*.

**Data** One chunk of raw entity data.

See also:

`http_content_type`

**Type** *event* (c: `connection`, is_orig: *bool*, ty: *string*, subty: *string*)

Generated for reporting an HTTP body's content type. This event is generated at the end of parsing an HTTP header, passing on the MIME type as specified by the `Content-Type` header. If that header is missing, this event is still raised with a default value of `text/plain`.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the entity was sent by the originator of the TCP connection.

**Ty** The main type.

**Subty** The subtype.

See also:

**Note:** This event is also raised for headers found in nested body entities.

### `http_message_done`

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, stat: `http_message_stat`)

Generated once at the end of parsing an HTTP message. Bro supports persistent and pipelined HTTP sessions and raises corresponding events as it parses client/server dialogues. A "message" is one top-level HTTP entity, such as a complete request or reply. Each message can have further nested sub-entities inside. This event is raised once all sub-entities belonging to a top-level message have been processed (and their corresponding `http_entity_*` events generated).

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Is_orig** True if the entity was sent by the originator of the TCP connection.

**Stat** Further meta information about the message.

See also:

### `http_event`

**Type** *`event`* (c: `connection`, event_type: *`string`*, detail: *`string`*)

Generated for errors found when decoding HTTP requests or replies.

See Wikipedia for more information about the HTTP protocol.

**C** The connection.

**Event_type** A string describing the general category of the problem found (e.g., `illegal format`).

**Detail** Further more detailed description of the error.

See also:

### `http_stats`

**Type** *`event`* (c: `connection`, stats: `http_stats_rec`)

Generated at the end of an HTTP session to report statistics about it. This event is raised after all of an HTTP session's requests and replies have been fully processed.

**C** The connection.

**Stats** Statistics summarizing HTTP-level properties of the finished connection.

See also:

## Functions

### `skip_http_entity_data`

**Type** *`function`* (c: `connection`, is_orig: *`bool`*) : *`any`*

Skips the data of the HTTP entity.

**C** The HTTP connection.

**Is_orig** If true, the client data is skipped, and the server data otherwise.

See also:

**`unescape_URI`**

**Type** *`function`* (URI: *`string`*) : *`string`*

Unescapes all characters in a URI (decode every `%xx` group).

**URI** The URI to unescape.

**Returns** The unescaped URI with all `%xx` groups decoded.

**Note:** Unescaping reserved characters may cause loss of information. **RFC 2396**: A URI is always in an "escaped" form, since escaping or unescaping a completed URI might change its semantics. Normally, the only time escape encodings can safely be made is when the URI is being created from its component parts.

### Bro::ICMP

ICMP analyzer

#### Components

*`Analyzer::ANALYZER_ICMP`*

#### Events

**`icmp_sent`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`)

Generated for all ICMP messages that are not handled separately with dedicated ICMP events. Bro's ICMP analyzer handles a number of ICMP messages directly with dedicated events. This event acts as a fallback for those it doesn't.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

See also:

**`icmp_sent_payload`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, payload: *`string`*)

The same as *`icmp_sent`* except containing the ICMP payload.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Payload** The payload of the ICMP message.

See also:

**`icmp_echo_request`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, id: *`count`*, seq: *`count`*, payload: *`string`*)

Generated for ICMP *echo request* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Id** The *echo request* identifier.

**Seq** The *echo request* sequence number.

**Payload** The message-specific data of the packet payload, i.e., everything after the first 8 bytes of the ICMP header.

See also:

**`icmp_echo_reply`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, id: *`count`*, seq: *`count`*, payload: *`string`*)

Generated for ICMP *echo reply* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Id** The *echo reply* identifier.

**Seq** The *echo reply* sequence number.

**Payload** The message-specific data of the packet payload, i.e., everything after the first 8 bytes of the ICMP header.

See also:

**`icmp_error_message`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, code: *`count`*, context: `icmp_context`)

Generated for all ICMPv6 error messages that are not handled separately with dedicated events. Bro's ICMP analyzer handles a number of ICMP error messages directly with dedicated events. This event acts as a fallback for those it doesn't.

See Wikipedia for more information about the ICMPv6 protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Code** The ICMP code of the error message.

**Context** A record with specifics of the original packet that the message refers to.

See also:

**`icmp_unreachable`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, code: *`count`*, context: `icmp_context`)

Generated for ICMP *destination unreachable* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Code** The ICMP code of the *unreachable* message.

**Context** A record with specifics of the original packet that the message refers to. *Unreachable* messages should include the original IP header from the packet that triggered them, and Bro parses that into the *context* structure. Note that if the *unreachable* includes only a partial IP header for some reason, no fields of *context* will be filled out.

See also:

**`icmp_packet_too_big`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, code: *`count`*, context: `icmp_context`)

Generated for ICMPv6 *packet too big* messages.

See Wikipedia for more information about the ICMPv6 protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Code** The ICMP code of the *too big* message.

**Context** A record with specifics of the original packet that the message refers to. *Too big* messages should include the original IP header from the packet that triggered them, and Bro parses that into the *context* structure. Note that if the *too big* includes only a partial IP header for some reason, no fields of *context* will be filled out.

See also:

**`icmp_time_exceeded`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, code: *`count`*, context: `icmp_context`)

Generated for ICMP *time exceeded* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Code** The ICMP code of the *exceeded* message.

**Context** A record with specifics of the original packet that the message refers to. *Unreachable* messages should include the original IP header from the packet that triggered them, and Bro parses that into the *context* structure. Note that if the *exceeded* includes only a partial IP header for some reason, no fields of *context* will be filled out.

See also:

**`icmp_parameter_problem`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, code: *`count`*, context: `icmp_context`)

Generated for ICMPv6 *parameter problem* messages.

See Wikipedia for more information about the ICMPv6 protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Code** The ICMP code of the *parameter problem* message.

**Context** A record with specifics of the original packet that the message refers to. *Parameter problem* messages should include the original IP header from the packet that triggered them, and Bro parses that into the *context* structure. Note that if the *parameter problem* includes only a partial IP header for some reason, no fields of *context* will be filled out.

See also:

`icmp_router_solicitation`

**Type** *event* (c: `connection`, icmp: `icmp_conn`, options: `icmp6_nd_options`)

Generated for ICMP *router solicitation* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Options** Any Neighbor Discovery options included with message (**RFC 4861**).

See also:

`icmp_router_advertisement`

**Type** *event* (c: `connection`, icmp: `icmp_conn`, cur_hop_limit: *count*, managed: *bool*, other: *bool*, home_agent: *bool*, pref: *count*, proxy: *bool*, rsv: *count*, router_lifetime: *interval*, reachable_time: *interval*, retrans_timer: *interval*, options: `icmp6_nd_options`)

Generated for ICMP *router advertisement* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Cur_hop_limit** The default value that should be placed in Hop Count field for outgoing IP packets.

**Managed** Managed address configuration flag, **RFC 4861**.

**Other** Other stateful configuration flag, **RFC 4861**.

**Home_agent** Mobile IPv6 home agent flag, **RFC 3775**.

**Pref** Router selection preferences, **RFC 4191**.

**Proxy** Neighbor discovery proxy flag, **RFC 4389**.

**Rsv** Remaining two reserved bits of router advertisement flags.

**Router_lifetime** How long this router should be used as a default router.

**Reachable_time** How long a neighbor should be considered reachable.

**Retrans_timer** How long a host should wait before retransmitting.

**Options** Any Neighbor Discovery options included with message (**RFC 4861**).

See also:

`icmp_neighbor_solicitation`

**Type** *event* (c: `connection`, icmp: `icmp_conn`, tgt: *addr*, options: `icmp6_nd_options`)

Generated for ICMP *neighbor solicitation* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Tgt** The IP address of the target of the solicitation.

**Options** Any Neighbor Discovery options included with message (**RFC 4861**).

See also:

**`icmp_neighbor_advertisement`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, router: *`bool`*, solicited: *`bool`*, override: *`bool`*, tgt: *`addr`*, options: `icmp6_nd_options`)

Generated for ICMP *neighbor advertisement* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Router** Flag indicating the sender is a router.

**Solicited** Flag indicating advertisement is in response to a solicitation.

**Override** Flag indicating advertisement should override existing caches.

**Tgt** the Target Address in the soliciting message or the address whose link-layer address has changed for unsolicited adverts.

**Options** Any Neighbor Discovery options included with message (**RFC 4861**).

See also:

**`icmp_redirect`**

**Type** *`event`* (c: `connection`, icmp: `icmp_conn`, tgt: *`addr`*, dest: *`addr`*, options: `icmp6_nd_options`)

Generated for ICMP *redirect* messages.

See Wikipedia for more information about the ICMP protocol.

**C** The connection record for the corresponding ICMP flow.

**Icmp** Additional ICMP-specific information augmenting the standard connection record *c*.

**Tgt** The address that is supposed to be a better first hop to use for ICMP Destination Address.

**Dest** The address of the destination which is redirected to the target.

**Options** Any Neighbor Discovery options included with message (**RFC 4861**).

See also:

### Bro::Ident

Ident analyzer

### Components

*`Analyzer::ANALYZER_IDENT`*

### Events

**ident_request**

**Type** *event* (c: connection, lport: *port*, rport: *port*)

Generated for Ident requests.

See Wikipedia for more information about the Ident protocol.

**C** The connection.

**Lport** The request's local port.

**Rport** The request's remote port.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**ident_reply**

**Type** *event* (c: connection, lport: *port*, rport: *port*, user_id: *string*, system: *string*)

Generated for Ident replies.

See Wikipedia for more information about the Ident protocol.

**C** The connection.

**Lport** The corresponding request's local port.

**Rport** The corresponding request's remote port.

**User_id** The user id returned by the reply.

**System** The operating system returned by the reply.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**ident_error**

**Type** *event* (c: connection, lport: *port*, rport: *port*, line: *string*)

Generated for Ident error replies.

See Wikipedia for more information about the Ident protocol.

**C** The connection.

**Lport** The corresponding request's local port.

**Rport** The corresponding request's remote port.

**Line** The error description returned by the reply.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

## Bro::InterConn

InterConn analyzer deprecated

### Components

*`Analyzer::ANALYZER_INTERCONN`*

### Events

**`interconn_stats`**

**Type** *`event`* (c: `connection`, os: `interconn_endp_stats`, rs: `interconn_endp_stats`)

Deprecated. Will be removed.

**`interconn_remove_conn`**

**Type** *`event`* (c: `connection`)

Deprecated. Will be removed.

## Bro::IRC

IRC analyzer

### Components

*`Analyzer::ANALYZER_IRC`*

### Events

**`irc_request`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, command: *`string`*, arguments: *`string`*)

Generated for all client-side IRC commands.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** Always true.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Command** The command.

**Arguments** The arguments for the command.

See also:

**Note:** This event is generated only for messages that originate at the client-side. Commands coming in from remote trigger the `irc_message` event instead.

**irc_reply**

**Type** `event` (c: `connection`, is_orig: `bool`, prefix: `string`, code: `count`, params: `string`)

Generated for all IRC replies. IRC replies are sent in response to a request and come with a reply code.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the reply. IRC uses the prefix to indicate the true origin of a message.

**Code** The reply code, as specified by the protocol.

**Params** The reply's parameters.

See also:

**irc_message**

**Type** `event` (c: `connection`, is_orig: `bool`, prefix: `string`, command: `string`, message: `string`)

Generated for IRC commands forwarded from the server to the client.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** Always false.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Command** The command.

**Message** TODO.

See also:

**Note:** This event is generated only for messages that are forwarded by the server to the client. Commands coming from client trigger the `irc_request` event instead.

**irc_quit_message**

**Type** `event` (c: `connection`, is_orig: `bool`, nick: `string`, message: `string`)

Generated for IRC messages of type *quit*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Nick** The nickname coming with the message.

**Message** The text included with the message.

See also:

**`irc_privmsg_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, source: *`string`*, target: *`string`*, message: *`string`*)

Generated for IRC messages of type *privmsg*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Source** The source of the private communication.

**Target** The target of the private communication.

**Message** The text of communication.

See also:

**`irc_notice_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, source: *`string`*, target: *`string`*, message: *`string`*)

Generated for IRC messages of type *notice*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Source** The source of the private communication.

**Target** The target of the private communication.

**Message** The text of communication.

See also:

**`irc_squery_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, source: *`string`*, target: *`string`*, message: *`string`*)

Generated for IRC messages of type *squery*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Source** The source of the private communication.

**Target** The target of the private communication.

**Message** The text of communication.

See also:

**`irc_join_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, info_list: `irc_join_list`)

Generated for IRC messages of type *join*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Info_list** The user information coming with the command.

See also:

**`irc_part_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, nick: *`string`*, chans: `string_set`, message: *`string`*)

Generated for IRC messages of type *part*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Nick** The nickname coming with the message.

**Chans** The set of channels affected.

**Message** The text coming with the message.

See also:

**`irc_nick_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, who: *`string`*, newnick: *`string`*)

Generated for IRC messages of type *nick*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Who** The user changing its nickname.

**Newnick** The new nickname.

See also:

**irc_invalid_nick**

**Type** *event* (c: `connection`, is_orig: *bool*)

Generated when a server rejects an IRC nickname.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

See also:

**irc_network_info**

**Type** *event* (c: `connection`, is_orig: *bool*, users: *count*, services: *count*, servers: *count*)

Generated for an IRC reply of type *luserclient*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Users** The number of users as returned in the reply.

**Services** The number of services as returned in the reply.

**Servers** The number of servers as returned in the reply.

See also:

**irc_server_info**

**Type** *event* (c: `connection`, is_orig: *bool*, users: *count*, services: *count*, servers: *count*)

Generated for an IRC reply of type *luserme*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Users** The number of users as returned in the reply.

**Services** The number of services as returned in the reply.

**Servers** The number of servers as returned in the reply.

See also:

**irc_channel_info**

**Type** *event* (c: `connection`, is_orig: *bool*, chans: *count*)

Generated for an IRC reply of type *luserchannels*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Chans** The number of channels as returned in the reply.

See also:

**irc_who_line**

**Type** *event* (c: `connection`, is_orig: *bool*, target_nick: *string*, channel: *string*, user: *string*, host: *string*, server: *string*, nick: *string*, params: *string*, hops: *count*, real_name: *string*)

Generated for an IRC reply of type *whoreply*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Target_nick** The target nickname.

**Channel** The channel.

**User** The user.

**Host** The host.

**Server** The server.

**Nick** The nickname.

**Params** The parameters.

**Hops** The hop count.

**Real_name** The real name.

See also:

**`irc_names_info`**

**Type** *event* (c: `connection`, is_orig: *bool*, c_type: *string*, channel: *string*, users: `string_set`)

Generated for an IRC reply of type *namereply*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**C_type** The channel type.

**Channel** The channel.

**Users** The set of users.

See also:

**`irc_whois_operator_line`**

**Type** *event* (c: `connection`, is_orig: *bool*, nick: *string*)

Generated for an IRC reply of type *whoisoperator*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Nick** The nickname specified in the reply.

See also:

**`irc_whois_channel_line`**

**Type** *event* (c: `connection`, is_orig: *bool*, nick: *string*, chans: `string_set`)

Generated for an IRC reply of type *whoischannels*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Nick** The nickname specified in the reply.

**Chans** The set of channels returned.

See also:

**`irc_whois_user_line`**

**Type** *event* (c: `connection`, is_orig: *bool*, nick: *string*, user: *string*, host: *string*, real_name: *string*)

Generated for an IRC reply of type *whoisuser*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Nick** The nickname specified in the reply.

**User** The user name specified in the reply.

**Host** The host name specified in the reply.

**Real_name** The real name specified in the reply.

See also:

**`irc_oper_response`**

**Type** *event* (c: `connection`, is_orig: *bool*, got_oper: *bool*)

Generated for IRC replies of type *youreoper* and *nooperhost*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Got_oper** True if the *oper* command was executed successfully (*youreport*) and false otherwise (*nooperhost*).

See also:

**`irc_global_users`**

**Type** *event* (c: `connection`, is_orig: *bool*, prefix: *string*, msg: *string*)

Generated for an IRC reply of type *globalusers*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Msg** The message coming with the reply.

See also:

`irc_channel_topic`

**Type** *event* (c: `connection`, is_orig: *bool*, channel: *string*, topic: *string*)

Generated for an IRC reply of type *topic*.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Channel** The channel name specified in the reply.

**Topic** The topic specified in the reply.

See also:

`irc_who_message`

**Type** *event* (c: `connection`, is_orig: *bool*, mask: *string*, oper: *bool*)

Generated for IRC messages of type *who*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Mask** The mask specified in the message.

**Oper** True if the operator flag was set.

See also:

`irc_whois_message`

**Type** *event* (c: `connection`, is_orig: *bool*, server: *string*, users: *string*)

Generated for IRC messages of type *whois*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Server** TODO.

**Users** TODO.

See also:

`irc_oper_message`

**Type** *event* (c: `connection`, is_orig: *bool*, user: *string*, password: *string*)

Generated for IRC messages of type *oper*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**User** The user specified in the message.

**Password** The password specified in the message.

See also:

**`irc_kick_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, chans: *`string`*, users: *`string`*, comment: *`string`*)

Generated for IRC messages of type *kick*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Chans** The channels specified in the message.

**Users** The users specified in the message.

**Comment** The comment specified in the message.

See also:

**`irc_error_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, message: *`string`*)

Generated for IRC messages of type *error*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Message** The textual description specified in the message.

See also:

**`irc_invite_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, nickname: *`string`*, channel: *`string`*)

Generated for IRC messages of type *invite*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Nickname** The nickname specified in the message.

**Channel** The channel specified in the message.

See also:

**`irc_mode_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, params: *`string`*)

Generated for IRC messages of type *mode*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Params** The parameters coming with the message.

See also:

**`irc_squit_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, server: *`string`*, message: *`string`*)

Generated for IRC messages of type *squit*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Server** The server specified in the message.

**Message** The textual description specified in the message.

See also:

**`irc_dcc_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, prefix: *`string`*, target: *`string`*, dcc_type: *`string`*, argument: *`string`*, address: *`addr`*, dest_port: *`count`*, size: *`count`*)

Generated for IRC messages of type *dcc*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Prefix** The optional prefix coming with the command. IRC uses the prefix to indicate the true origin of a message.

**Target** The target specified in the message.

**Dcc_type** The DCC type specified in the message.

**Argument** The argument specified in the message.

**Address** The address specified in the message.

**Dest_port** The destination port specified in the message.

**Size** The size specified in the message.

See also:

**`irc_user_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, user: *`string`*, host: *`string`*, server: *`string`*, real_name: *`string`*)

Generated for IRC messages of type *user*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**User** The user specified in the message.

**Host** The host name specified in the message.

**Server** The server name specified in the message.

**Real_name** The real name specified in the message.

See also:

**`irc_password_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, password: *`string`*)

Generated for IRC messages of type *password*. This event is generated for messages coming from both the client and the server.

See Wikipedia for more information about the IRC protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Password** The password specified in the message.

See also:

### Bro::KRB

Kerberos analyzer

#### Components

*`Analyzer::ANALYZER_KRB`*

*`Analyzer::ANALYZER_KRB_TCP`*

## Types

**KRB::Error_Msg**

**Type** *record*

**pvno:** *count* Protocol version number (5 for KRB5)

**msg_type:** *count* The message type (30 for ERROR_MSG)

**client_time:** *time &optional* Current time on the client

**server_time:** *time* Current time on the server

**error_code:** *count* The specific error code

**client_realm:** *string &optional* Realm of the ticket

**client_name:** *string &optional* Name on the ticket

**service_realm:** *string* Realm of the service

**service_name:** *string* Name of the service

**error_text:** *string &optional* Additional text to explain the error

**pa_data:** *vector* **of** *KRB::Type_Value &optional* Optional pre-authentication data

The data from the ERROR_MSG message. See **RFC 4120**.

**KRB::SAFE_Msg**

**Type** *record*

**pvno:** *count* Protocol version number (5 for KRB5)

**msg_type:** *count* The message type (20 for SAFE_MSG)

**data:** *string* The application-specific data that is being passed from the sender to the reciever

**timestamp:** *time &optional* Current time from the sender of the message

**seq:** *count &optional* Sequence number used to detect replays

**sender:** *KRB::Host_Address &optional* Sender address

**recipient:** *KRB::Host_Address &optional* Recipient address

The data from the SAFE message. See **RFC 4120**.

**KRB::KDC_Options**

**Type** *record*

**forwardable:** *bool* The ticket to be issued should have its forwardable flag set.

**forwarded:** *bool* A (TGT) request for forwarding.

**proxiable:** *bool* The ticket to be issued should have its proxiable flag set.

**proxy:** *bool* A request for a proxy.

**allow_postdate:** *bool* The ticket to be issued should have its may-postdate flag set.

**postdated:** *bool* A request for a postdated ticket.

**renewable:** *bool* The ticket to be issued should have its renewable flag set.

**opt_hardware_auth:** *bool* Reserved for opt_hardware_auth

**disable_transited_check:** *`bool`* Request that the KDC not check the transited field of a TGT against the policy of the local realm before it will issue derivative tickets based on the TGT.

**renewable_ok:** *`bool`* If a ticket with the requested lifetime cannot be issued, a renewable ticket is acceptable

**enc_tkt_in_skey:** *`bool`* The ticket for the end server is to be encrypted in the session key from the additional TGT provided

**renew:** *`bool`* The request is for a renewal

**validate:** *`bool`* The request is to validate a postdated ticket.

KDC Options. See **RFC 4120**

**`KRB::AP_Options`**

**Type** *`record`*

**use_session_key:** *`bool`* Indicates that user-to-user-authentication is in use

**mutual_required:** *`bool`* Mutual authentication is required

AP Options. See **RFC 4120**

**`KRB::Type_Value`**

**Type** *`record`*

**data_type:** *`count`* The data type

**val:** *`string`* The data value

Used in a few places in the Kerberos analyzer for elements that have a type and a string value.

**`KRB::Ticket`**

**Type** *`record`*

**pvno:** *`count`* Protocol version number (5 for KRB5)

**realm:** *`string`* Realm

**service_name:** *`string`* Name of the service

**cipher:** *`count`* Cipher the ticket was encrypted with

A Kerberos ticket. See **RFC 4120**.

**`KRB::Ticket_Vector`**

**Type** *`vector`* of *`KRB::Ticket`*

**`KRB::Host_Address`**

**Type** *`record`*

**ip:** *`addr &log &optional`* IPv4 or IPv6 address

**netbios:** *`string &log &optional`* NetBIOS address

**unknown:** *`KRB::Type_Value &optional`* Some other type that we don’t support yet

A Kerberos host address See **RFC 4120**.

**`KRB::KDC_Request`**

**Type** *`record`*

**pvno:** *`count`* Protocol version number (5 for KRB5)

**msg_type:** *count* The message type (10 for AS_REQ, 12 for TGS_REQ)

**pa_data:** *vector* **of** *KRB::Type_Value &optional* Optional pre-authentication data

**kdc_options:** *KRB::KDC_Options* Options specified in the request

**client_name:** *string &optional* Name on the ticket

**service_realm:** *string* Realm of the service

**service_name:** *string &optional* Name of the service

**from:** *time &optional* Time the ticket is good from

**till:** *time* Time the ticket is good till

**rtime:** *time &optional* The requested renew-till time

**nonce:** *count* A random nonce generated by the client

**encryption_types:** *vector* **of** *count* The desired encryption algorithms, in order of preference

**host_addrs:** *vector* **of** *KRB::Host_Address &optional* Any additional addresses the ticket should be valid for

**additional_tickets:** *vector* **of** *KRB::Ticket &optional* Additional tickets may be included for certain transactions

The data from the AS_REQ and TGS_REQ messages. See **RFC 4120**.

**`KRB::KDC_Response`**

**Type** *record*

**pvno:** *count* Protocol version number (5 for KRB5)

**msg_type:** *count* The message type (11 for AS_REP, 13 for TGS_REP)

**pa_data:** *vector* **of** *KRB::Type_Value &optional* Optional pre-authentication data

**client_realm:** *string &optional* Realm on the ticket

**client_name:** *string* Name on the service

**ticket:** *KRB::Ticket* The ticket that was issued

The data from the AS_REQ and TGS_REQ messages. See **RFC 4120**.

## Events

**`krb_as_request`**

**Type** *event* (c: `connection`, msg: *KRB::KDC_Request*)

A Kerberos 5 `Authentication Server (AS) Request` as defined in **RFC 4120**. The AS request contains a username of the client requesting authentication, and returns an AS reply with an encrypted Ticket Granting Ticket (TGT) for that user. The TGT can then be used to request further tickets for other services.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Msg** A Kerberos KDC request message data structure.

See also:

**krb_as_response**

**Type** *event* (c: connection, msg: *KRB::KDC_Response*)

A Kerberos 5 `Authentication Server (AS) Response` as defined in **RFC 4120**. Following the AS request for a user, an AS reply contains an encrypted Ticket Granting Ticket (TGT) for that user. The TGT can then be used to request further tickets for other services.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Msg** A Kerberos KDC reply message data structure.

See also:

**krb_tgs_request**

**Type** *event* (c: connection, msg: *KRB::KDC_Request*)

A Kerberos 5 `Ticket Granting Service (TGS) Request` as defined in **RFC 4120**. Following the Authentication Server exchange, if successful, the client now has a Ticket Granting Ticket (TGT). To authenticate to a Kerberized service, the client requests a Service Ticket, which will be returned in the TGS reply.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Msg** A Kerberos KDC request message data structure.

See also:

**krb_tgs_response**

**Type** *event* (c: connection, msg: *KRB::KDC_Response*)

A Kerberos 5 `Ticket Granting Service (TGS) Response` as defined in **RFC 4120**. This message returns a Service Ticket to the client, which is encrypted with the service's long-term key, and which the client can use to authenticate to that service.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Msg** A Kerberos KDC reply message data structure.

See also:

**krb_ap_request**

**Type** *event* (c: connection, ticket: *KRB::Ticket*, opts: *KRB::AP_Options*)

A Kerberos 5 `Authentication Header (AP) Request` as defined in **RFC 4120**. This message contains authentication information that should be part of the first message in an authenticated transaction.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Ticket** The Kerberos ticket being used for authentication.

**Opts** A Kerberos AP options data structure.

See also:

**krb_ap_response**

**Type** *event* (c: connection)

A Kerberos 5 `Authentication Header (AP) Response` as defined in **RFC 4120**. This is used if mutual authentication is desired. All of the interesting information in here is encrypted, so the event doesn't have much useful data, but it's provided in case it's important to know that this message was sent.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

See also:

**`krb_priv`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*)

A Kerberos 5 `Private Message` as defined in **RFC 4120**. This is a private (encrypted) application message, so the event doesn't have much useful data, but it's provided in case it's important to know that this message was sent.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Is_orig** Whether the originator of the connection sent this message.

See also:

**`krb_safe`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, msg: *`KRB::SAFE_Msg`*)

A Kerberos 5 `Safe Message` as defined in **RFC 4120**. This is a safe (checksummed) application message.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Is_orig** Whether the originator of the connection sent this message.

**Msg** A Kerberos SAFE message data structure.

See also:

**`krb_cred`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, tickets: *`KRB::Ticket_Vector`*)

A Kerberos 5 `Credential Message` as defined in **RFC 4120**. This is a private (encrypted) message to forward credentials.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Is_orig** Whether the originator of the connection sent this message.

**Tickets** Tickets obtained from the KDC that are being forwarded.

See also:

**`krb_error`**

**Type** *`event`* (c: `connection`, msg: *`KRB::Error_Msg`*)

A Kerberos 5 `Error Message` as defined in **RFC 4120**.

See Wikipedia for more information about the Kerberos protocol.

**C** The connection over which this Kerberos message was sent.

**Msg** A Kerberos error message data structure.

See also:

### Bro::Login

Telnet/Rsh/Rlogin analyzers

#### Components

*Analyzer::ANALYZER_CONTENTS_RLOGIN*

*Analyzer::ANALYZER_CONTENTS_RSH*

*Analyzer::ANALYZER_LOGIN*

*Analyzer::ANALYZER_NVT*

*Analyzer::ANALYZER_RLOGIN*

*Analyzer::ANALYZER_RSH*

*Analyzer::ANALYZER_TELNET*

#### Events

**rsh_request**

**Type** *event* (c: `connection`, client_user: *string*, server_user: *string*, line: *string*, new_session: *bool*)

Generated for client side commands on an RSH connection.

See **RFC 1258** for more information about the Rlogin/Rsh protocol.

**C** The connection.

**Client_user** The client-side user name as sent in the initial protocol handshake.

**Server_user** The server-side user name as sent in the initial protocol handshake.

**Line** The command line sent in the request.

**New_session** True if this is the first command of the Rsh session.

See also:

**Note:** For historical reasons, these events are separate from the `login_` events. Ideally, they would all be handled uniquely.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**rsh_reply**

**Type** *event* (c: `connection`, client_user: *string*, server_user: *string*, line: *string*)

Generated for client side commands on an RSH connection.

See **RFC 1258** for more information about the Rlogin/Rsh protocol.

**C** The connection.

**Client_user** The client-side user name as sent in the initial protocol handshake.

**Server_user** The server-side user name as sent in the initial protocol handshake.

**Line** The command line sent in the request.

See also:

**Note:** For historical reasons, these events are separate from the `login_` events. Ideally, they would all be handled uniquely.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`login_failure`**

**Type** *`event`* (c: `connection`, user: *`string`*, client_user: *`string`*, password: *`string`*, line: *`string`*)

Generated for Telnet/Rlogin login failures. The *login* analyzer inspects Telnet/Rlogin sessions to heuristically extract username and password information as well as the text returned by the login server. This event is raised if a login attempt appears to have been unsuccessful.

**C** The connection.

**User** The user name tried.

**Client_user** 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).

**Password** The password tried.

**Line** The line of text that led the analyzer to conclude that the authentication had failed.

See also:

**Note:** The login analyzer depends on a set of script-level variables that need to be configured with patterns identifying login attempts. This configuration has not yet been ported over from Bro 1.5 to Bro 2.x, and the analyzer is therefore not directly usable at the moment.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

**`login_success`**

**Type** *event* (c: `connection`, user: *string*, client_user: *string*, password: *string*, line: *string*)

Generated for successful Telnet/Rlogin logins. The *login* analyzer inspects Telnet/Rlogin sessions to heuristically extract username and password information as well as the text returned by the login server. This event is raised if a login attempt appears to have been successful.

**C** The connection.

**User** The user name used.

**Client_user** 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).

**Password** The password used.

**Line** The line of text that led the analyzer to conclude that the authentication had succeeded.

See also:

**Note:** The login analyzer depends on a set of script-level variables that need to be configured with patterns identifying login attempts. This configuration has not yet been ported over from Bro 1.5 to Bro 2.x, and the analyzer is therefore not directly usable at the moment.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

**`login_input_line`**

**Type** *event* (c: `connection`, line: *string*)

Generated for lines of input on Telnet/Rlogin sessions. The line will have control characters (such as in-band Telnet options) removed.

**C** The connection.

**Line** The input line.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

**`login_output_line`**

**Type** *event* (c: `connection`, line: *string*)

Generated for lines of output on Telnet/Rlogin sessions. The line will have control characters (such as in-band Telnet options) removed.

**C** The connection.

**Line** The ouput line.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

**`login_confused`**

**Type** *`event`* (c: `connection`, msg: *`string`*, line: *`string`*)

Generated when tracking of Telnet/Rlogin authentication failed. As Bro's *login* analyzer uses a number of heuristics to extract authentication information, it may become confused. If it can no longer correctly track the authentication dialog, it raises this event.

**C** The connection.

**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)

**Line** The line of text that caused the heuristics to conclude they were confused.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

**`login_confused_text`**

**Type** *`event`* (c: `connection`, line: *`string`*)

Generated after getting confused while tracking a Telnet/Rlogin authentication dialog. The *login* analyzer generates this even for every line of user input after it has reported *`login_confused`* for a connection.

**C** The connection.

**Line** The line the user typed.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

**`login_terminal`**

**Type** *`event`* (c: `connection`, terminal: *`string`*)

Generated for clients transmitting a terminal type in a Telnet session. This information is extracted out of environment variables sent as Telnet options.

**C** The connection.

**Terminal** The TERM value transmitted.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

### `login_display`

**Type** *`event`* (c: `connection`, display: *`string`*)

Generated for clients transmitting an X11 DISPLAY in a Telnet session. This information is extracted out of environment variables sent as Telnet options.

**C** The connection.

**Display** The DISPLAY transmitted.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

### `authentication_accepted`

**Type** *`event`* (name: *`string`*, c: `connection`)

Generated when a Telnet authentication has been successful. The Telnet protocol 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.

See Wikipedia for more information about the Telnet protocol.

**Name** The authenticated name.

**C** The connection.

See also:

---

**Note:** This event inspects the corresponding Telnet option while *`login_success`* heuristically determines success by watching session data.

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

### `authentication_rejected`

**Type** *`event`* (name: *`string`*, c: `connection`)

Generated when a Telnet authentication has been unsuccessful. The Telnet protocol includes options for negotiating authentication. When such an option is sent from client to server and the server replies that it did not accept the authentication, then the event engine generates this event.

See Wikipedia for more information about the Telnet protocol.

**Name** The attempted authentication name.

**C** The connection.

See also:

**Note:** This event inspects the corresponding Telnet option while *`login_success`* heuristically determines failure by watching session data.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

**`authentication_skipped`**

**Type** *`event`* (c: `connection`)

Generated for Telnet/Rlogin sessions when a pattern match indicates that no authentication is performed.

See Wikipedia for more information about the Telnet protocol.

**C** The connection.

See also:

**Note:** The login analyzer depends on a set of script-level variables that need to be configured with patterns identifying activity. This configuration has not yet been ported over from Bro 1.5 to Bro 2.x, and the analyzer is therefore not directly usable at the moment.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

**`login_prompt`**

**Type** *`event`* (c: `connection`, prompt: *`string`*)

Generated for clients transmitting a terminal prompt in a Telnet session. This information is extracted out of environment variables sent as Telnet options.

See Wikipedia for more information about the Telnet protocol.

**C** The connection.

**Prompt** The TTYPROMPT transmitted.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

**`activating_encryption`**

**Type** *`event`* (c: `connection`)

Generated for Telnet sessions when encryption is activated. The Telnet protocol includes options for negotiating encryption. When such a series of options is successfully negotiated, the event engine generates this event.

See Wikipedia for more information about the Telnet protocol.

**C** The connection.

See also:

**`inconsistent_option`**

**Type** *`event`* (c: `connection`)

Generated for an inconsistent Telnet option. Telnet options are specified by the client and server stating which options they are willing to support vs. which they are 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 this event.

See Wikipedia for more information about the Telnet protocol.

**C** The connection.

See also:

**`bad_option`**

**Type** *`event`* (c: `connection`)

Generated for an ill-formed or unrecognized Telnet option.

See Wikipedia for more information about the Telnet protocol.

**C** The connection.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

**`bad_option_termination`**

**Type** *`event`* (c: `connection`)

Generated for a Telnet option that's incorrectly terminated.

See Wikipedia for more information about the Telnet protocol.

**C** The connection.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

### Functions

#### `get_login_state`

**Type** *`function`* (cid: `conn_id`) : *`count`*

Returns the state of the given login (Telnet or Rlogin) connection.

**Cid** The connection ID.

**Returns** False if the connection is not active or is not tagged as a login analyzer. Otherwise the function returns the state, which can be 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.

See also:

#### `set_login_state`

**Type** *`function`* (cid: `conn_id`, new_state: *`count`*) : *`bool`*

Sets the login state of a connection with a login analyzer.

**Cid** The connection ID.

**New_state** The new state of the login analyzer. See *`get_login_state`* for possible values.

**Returns** Returns false if *cid* is not an active connection or is not tagged as a login analyzer, and true otherwise.

See also:

### Bro::MIME

MIME parsing

### Components

### Events

#### `mime_begin_entity`

**Type** *event* (c: `connection`)

Generated when starting to parse an email MIME entity. MIME is a protocol-independent data format for encoding text and files, along with corresponding metadata, for transmission. Bro raises this event when it begins parsing a MIME entity extracted from an email protocol.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

See also:

**Note:** Bro also extracts MIME entities from HTTP sessions. For those, however, it raises *http_begin_entity* instead.

**mime_end_entity**

**Type** *event* (c: `connection`)

Generated when finishing parsing an email MIME entity. MIME is a protocol-independent data format for encoding text and files, along with corresponding metadata, for transmission. Bro raises this event when it finished parsing a MIME entity extracted from an email protocol.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

See also:

**Note:** Bro also extracts MIME entities from HTTP sessions. For those, however, it raises *http_end_entity* instead.

**mime_one_header**

**Type** *event* (c: `connection`, h: `mime_header_rec`)

Generated for individual MIME headers extracted from email MIME entities. MIME is a protocol-independent data format for encoding text and files, along with corresponding metadata, for transmission.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**H** The parsed MIME header.

See also:

**Note:** Bro also extracts MIME headers from HTTP sessions. For those, however, it raises *http_header* instead.

**mime_all_headers**

**Type** *event* (c: `connection`, hlist: `mime_header_list`)

Generated for MIME headers extracted from email MIME entities, passing all headers at once. MIME is a protocol-independent data format for encoding text and files, along with corresponding metadata, for transmission.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**Hlist** A *table* containing all headers extracted from the current entity. The table is indexed by the position of the header (1 for the first, 2 for the second, etc.).

See also:

---

**Note:** Bro also extracts MIME headers from HTTP sessions. For those, however, it raises *`http_header`* instead.

---

**`mime_segment_data`**

**Type** *`event`* (c: `connection`, length: *`count`*, data: *`string`*)

Generated for chunks of decoded MIME data from email MIME entities. MIME is a protocol-independent data format for encoding text and files, along with corresponding metadata, for transmission. As Bro parses the data of an entity, it raises a sequence of these events, each coming as soon as a new chunk of data is available. In contrast, there is also *`mime_entity_data`*, which passes all of an entities data at once in a single block. While the latter is more convenient to handle, `mime_segment_data` is more efficient as Bro does not need to buffer the data. Thus, if possible, this event should be preferred.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**Length** The length of *data*.

**Data** The raw data of one segment of the current entity.

See also:

---

**Note:** Bro also extracts MIME data from HTTP sessions. For those, however, it raises *`http_entity_data`* (sic!) instead.

---

**`mime_entity_data`**

**Type** *`event`* (c: `connection`, length: *`count`*, data: *`string`*)

Generated for data decoded from an email MIME entity. This event delivers the complete content of a single MIME entity. In contrast, there is also *`mime_segment_data`*, which passes on a sequence of data chunks as they come in. While `mime_entity_data` is more convenient to handle, `mime_segment_data` is more efficient as Bro does not need to buffer the data. Thus, if possible, the latter should be preferred.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**Length** The length of *data*.

**Data** The raw data of the complete entity.

See also:

**Note:** While Bro also decodes MIME entities extracted from HTTP sessions, there's no corresponding event for that currently.

**`mime_all_data`**

**Type** *`event`* (c: `connection`, length: *`count`*, data: *`string`*)

Generated for passing on all data decoded from a single email MIME message. If an email message has more than one MIME entity, this event combines all their data into a single value for analysis. Note that because of the potentially significant buffering necessary, using this event can be expensive.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**Length** The length of *data*.

**Data** The raw data of all MIME entities concatenated.

See also:

**Note:** While Bro also decodes MIME entities extracted from HTTP sessions, there's no corresponding event for that currently.

**`mime_event`**

**Type** *`event`* (c: `connection`, event_type: *`string`*, detail: *`string`*)

Generated for errors found when decoding email MIME entities.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**Event_type** A string describing the general category of the problem found (e.g., `illegal format`).

**Detail** Further more detailed description of the error.

See also:

**Note:** Bro also extracts MIME headers from HTTP sessions. For those, however, it raises *`http_event`* instead.

**`mime_content_hash`**

**Type** *`event`* (c: `connection`, content_len: *`count`*, hash_value: *`string`*)

Generated for decoded MIME entities extracted from email messages, passing on their MD5 checksums. Bro computes the MD5 over the complete decoded data of each MIME entity.

Bro's MIME analyzer for emails currently supports SMTP and POP3. See Wikipedia for more information about MIME.

**C** The connection.

**Content_len** The length of the entity being hashed.

**Hash_value** The MD5 hash.

See also:

**Note:** While Bro also decodes MIME entities extracted from HTTP sessions, there's no corresponding event for that currently.

## Bro::Modbus

Modbus analyzer

### Components

*Analyzer::ANALYZER_MODBUS*

### Events

**modbus_message**

**Type** *event* (c: `connection`, headers: `ModbusHeaders`, is_orig: *bool*)

Generated for any Modbus message regardless if the particular function is further supported or not.

**C** The connection.

**Headers** The headers for the modbus function.

**Is_orig** True if the event is raised for the originator side.

**modbus_exception**

**Type** *event* (c: `connection`, headers: `ModbusHeaders`, code: *count*)

Generated for any Modbus exception message.

**C** The connection.

**Headers** The headers for the modbus function.

**Code** The exception code.

**modbus_read_coils_request**

**Type** *event* (c: `connection`, headers: `ModbusHeaders`, start_address: *count*, quantity: *count*)

Generated for a Modbus read coils request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first coil to be read.

**Quantity** The number of coils to be read.

**modbus_read_coils_response**

**Type** *event* (c: `connection`, headers: `ModbusHeaders`, coils: `ModbusCoils`)

Generated for a Modbus read coils response.

**C** The connection.

**Headers** The headers for the modbus function.

**Coils** The coil values returned from the device.

**`modbus_read_discrete_inputs_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, quantity: *`count`*)

Generated for a Modbus read discrete inputs request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first coil to be read.

**Quantity** The number of coils to be read.

**`modbus_read_discrete_inputs_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, coils: `ModbusCoils`)

Generated for a Modbus read discrete inputs response.

**C** The connection.

**Headers** The headers for the modbus function.

**Coils** The coil values returned from the device.

**`modbus_read_holding_registers_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, quantity: *`count`*)

Generated for a Modbus read holding registers request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first register to be read.

**Quantity** The number of registers to be read.

**`modbus_read_holding_registers_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, registers: `ModbusRegisters`)

Generated for a Modbus read holding registers response.

**C** The connection.

**Headers** The headers for the modbus function.

**Registers** The register values returned from the device.

**`modbus_read_input_registers_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, quantity: *`count`*)

Generated for a Modbus read input registers request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first register to be read.

**Quantity** The number of registers to be read.

**`modbus_read_input_registers_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, registers: `ModbusRegisters`)

Generated for a Modbus read input registers response.

**C** The connection.

**Headers** The headers for the modbus function.

**Registers** The register values returned from the device.

**`modbus_write_single_coil_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, address: *`count`*, value: *`bool`*)

Generated for a Modbus write single coil request.

**C** The connection.

**Headers** The headers for the modbus function.

**Address** The memory address of the coil to be written.

**Value** The value to be written to the coil.

**`modbus_write_single_coil_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, address: *`count`*, value: *`bool`*)

Generated for a Modbus write single coil response.

**C** The connection.

**Headers** The headers for the modbus function.

**Address** The memory address of the coil that was written.

**Value** The value that was written to the coil.

**`modbus_write_single_register_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, address: *`count`*, value: *`count`*)

Generated for a Modbus write single register request.

**C** The connection.

**Headers** The headers for the modbus function.

**Address** The memory address of the register to be written.

**Value** The value to be written to the register.

**`modbus_write_single_register_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, address: *`count`*, value: *`count`*)

Generated for a Modbus write single register response.

**C** The connection.

**Headers** The headers for the modbus function.

**Address** The memory address of the register that was written.

**Value** The value that was written to the register.

**`modbus_write_multiple_coils_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, coils: `ModbusCoils`)

Generated for a Modbus write multiple coils request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first coil to be written.

**Coils** The values to be written to the coils.

**`modbus_write_multiple_coils_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, quantity: *`count`*)

Generated for a Modbus write multiple coils response.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first coil that was written.

**Quantity** The quantity of coils that were written.

**`modbus_write_multiple_registers_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, registers: `ModbusRegisters`)

Generated for a Modbus write multiple registers request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first register to be written.

**Registers** The values to be written to the registers.

**`modbus_write_multiple_registers_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*, quantity: *`count`*)

Generated for a Modbus write multiple registers response.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The memory address of the first register that was written.

**Quantity** The quantity of registers that were written.

**`modbus_read_file_record_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`)

Generated for a Modbus read file record request.

**C** The connection.

**Headers** The headers for the modbus function.

**`modbus_read_file_record_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`)

Generated for a Modbus read file record response.

**C** The connection.

**Headers** The headers for the modbus function.

**`modbus_write_file_record_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`)

Generated for a Modbus write file record request.

**C** The connection.

**Headers** The headers for the modbus function.

**`modbus_write_file_record_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`)

Generated for a Modbus write file record response.

**C** The connection.

**Headers** The headers for the modbus function.

**`modbus_mask_write_register_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, address: *`count`*, and_mask: *`count`*, or_mask: *`count`*)

Generated for a Modbus mask write register request.

**C** The connection.

**Headers** The headers for the modbus function.

**Address** The memory address of the register where the masks should be applied.

**And_mask** The value of the logical AND mask to apply to the register.

**Or_mask** The value of the logical OR mask to apply to the register.

**`modbus_mask_write_register_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, address: *`count`*, and_mask: *`count`*, or_mask: *`count`*)

Generated for a Modbus mask write register request.

**C** The connection.

**Headers** The headers for the modbus function.

**Address** The memory address of the register where the masks were applied.

**And_mask** The value of the logical AND mask applied register.

**Or_mask** The value of the logical OR mask applied to the register.

**`modbus_read_write_multiple_registers_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, read_start_address: *`count`*, read_quantity: *`count`*, write_start_address: *`count`*, write_registers: `ModbusRegisters`)

Generated for a Modbus read/write multiple registers request.

**C** The connection.

**Headers** The headers for the modbus function.

**Read_start_address** The memory address of the first register to be read.

**Read_quantity** The number of registers to read.

**Write_start_address** The memory address of the first register to be written.

**Write_registers** The values to be written to the registers.

**`modbus_read_write_multiple_registers_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, written_registers: `ModbusRegisters`)

Generated for a Modbus read/write multiple registers response.

**C** The connection.

**Headers** The headers for the modbus function.

**Written_registers** The register values read from the registers specified in the request.

**`modbus_read_fifo_queue_request`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, start_address: *`count`*)

Generated for a Modbus read FIFO queue request.

**C** The connection.

**Headers** The headers for the modbus function.

**Start_address** The address of the FIFO queue to read.

**`modbus_read_fifo_queue_response`**

**Type** *`event`* (c: `connection`, headers: `ModbusHeaders`, fifos: `ModbusRegisters`)

Generated for a Modbus read FIFO queue response.

**C** The connection.

**Headers** The headers for the modbus function.

**Fifos** The register values read from the FIFO queue on the device.

## Bro::MySQL

MySQL analyzer

### Components

*`Analyzer::ANALYZER_MYSQL`*

### Events

**`mysql_command_request`**

**Type** *`event`* (c: `connection`, command: *`count`*, arg: *`string`*)

Generated for a command request from a MySQL client.

See the MySQL documentation for more information about the MySQL protocol.

**C** The connection.

**Command** The numerical code of the command issued.

**Arg** The argument for the command (empty string if not provided).

See also:

**`mysql_error`**

**Type** *`event`* (c: `connection`, code: *`count`*, msg: *`string`*)

Generated for an unsuccessful MySQL response.

See the MySQL documentation for more information about the MySQL protocol.

**C** The connection.

**Code** The error code.

**Msg** Any extra details about the error (empty string if not provided).

See also:

**`mysql_ok`**

**Type** *`event`* (c: `connection`, affected_rows: *`count`*)

Generated for a successful MySQL response.

See the MySQL documentation for more information about the MySQL protocol.

**C** The connection.

**Affected_rows** The number of rows that were affected.

See also:

**`mysql_server_version`**

**Type** *`event`* (c: `connection`, ver: *`string`*)

Generated for the initial server handshake packet, which includes the MySQL server version.

See the MySQL documentation for more information about the MySQL protocol.

**C** The connection.

**Ver** The server version string.

See also:

**`mysql_handshake`**

**Type** *`event`* (c: `connection`, username: *`string`*)

Generated for a client handshake response packet, which includes the username the client is attempting to connect as.

See the MySQL documentation for more information about the MySQL protocol.

**C** The connection.

**Username** The username supplied by the client

See also:

### Bro::NCP

NCP analyzer

#### Components

*Analyzer::ANALYZER_CONTENTS_NCP*

*Analyzer::ANALYZER_NCP*

#### Events

**ncp_request**

**Type** *event* (c: `connection`, frame_type: *count*, length: *count*, func: *count*)

Generated for NCP requests (Netware Core Protocol).

See Wikipedia for more information about the NCP protocol.

**C** The connection.

**Frame_type** The frame type, as specified by the protocol.

**Length** The length of the request body, excluding the frame header.

**Func** The requested function, as specified by the protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**ncp_reply**

**Type** *event* (c: `connection`, frame_type: *count*, length: *count*, req_frame: *count*, req_func: *count*, completion_code: *count*)

Generated for NCP replies (Netware Core Protocol).

See Wikipedia for more information about the NCP protocol.

**C** The connection.

**Frame_type** The frame type, as specified by the protocol.

**Length** The length of the request body, excluding the frame header.

**Req_frame** The frame type from the corresponding request.

**Req_func** The function code from the corresponding request.

**Completion_code** The reply's completion code, as specified by the protocol.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### Bro::NetBIOS

NetBIOS analyzer support

#### Components

*Analyzer::ANALYZER_CONTENTS_NETBIOSSSN*

*Analyzer::ANALYZER_NETBIOSSSN*

#### Events

**`netbios_session_message`**

**Type** *event* (c: `connection`, is_orig: *bool*, msg_type: *count*, data_len: *count*)

Generated for all NetBIOS SSN and DGM messages. Bro's NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Is_orig** True if the message was sent by the originator of the connection.

**Msg_type** The general type of message, as defined in Section 4.3.1 of **RFC 1002**.

**Data_len** The length of the message's payload.

See also:

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`netbios_session_request`**

**Type** *event* (c: `connection`, msg: *string*)

Generated for NetBIOS messages of type *session request*. Bro's NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Msg** The raw payload of the message sent, excluding the common NetBIOS header.

See also:

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### `netbios_session_accepted`

**Type** `event` (c: `connection`, msg: `string`)

Generated for NetBIOS messages of type *positive session response*. Bro's NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Msg** The raw payload of the message sent, excluding the common NetBIOS header.

See also:

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### `netbios_session_rejected`

**Type** `event` (c: `connection`, msg: `string`)

Generated for NetBIOS messages of type *negative session response*. Bro's NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Msg** The raw payload of the message sent, excluding the common NetBIOS header.

See also:

---

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**`netbios_session_raw_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, msg: *`string`*)

Generated for NetBIOS messages of type *session message* that are not carrying an SMB payload.

NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Is_orig** True if the message was sent by the originator of the connection.

**Msg** The raw payload of the message sent, excluding the common NetBIOS header (i.e., the `user_data`).

See also:

---

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

---

**Todo**

This is an oddly named event. In fact, it's probably an odd event to have to begin with.

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**`netbios_session_ret_arg_resp`**

**Type** *`event`* (c: `connection`, msg: *`string`*)

Generated for NetBIOS messages of type *retarget response*. Bro's NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Msg** The raw payload of the message sent, excluding the common NetBIOS header.

See also:

---

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

---

**Todo**

This is an oddly named event.

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**`netbios_session_keepalive`**

**Type** *`event`* (c: `connection`, msg: *`string`*)

Generated for NetBIOS messages of type *keep-alive*. Bro's NetBIOS analyzer processes the NetBIOS session service running on TCP port 139, and (despite its name!) the NetBIOS datagram service on UDP port 138.

See Wikipedia for more information about NetBIOS. **RFC 1002** describes the packet format for NetBIOS over TCP/IP, which Bro parses.

**C** The connection, which may be TCP or UDP, depending on the type of the NetBIOS session.

**Msg** The raw payload of the message sent, excluding the common NetBIOS header.

See also:

---

**Note:** These days, NetBIOS is primarily used as a transport mechanism for SMB/CIFS. Bro's SMB analyzer parses both SMB-over-NetBIOS and SMB-over-TCP on port 445.

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

### Functions

**decode_netbios_name**

**Type** *function* (name: *string*) : *string*

Decode a NetBIOS name. See http://support.microsoft.com/kb/194203.

**Name** The encoded NetBIOS name, e.g., "FEEIEFCAEOEFFEECEJEPFDCAEOEBENEF".

**Returns** The decoded NetBIOS name, e.g., "THE NETBIOS NAME".

See also:

**decode_netbios_name_type**

**Type** *function* (name: *string*) : *count*

Converts a NetBIOS name type to its corresponding numeric value. See http://support.microsoft.com/kb/163409.

**Name** The NetBIOS name type.

**Returns** The numeric value of *name*.

See also:

## Bro::NTP

NTP analyzer

### Components

*Analyzer::ANALYZER_NTP*

### Events

**ntp_message**

**Type** *event* (u: connection, msg: ntp_msg, excess: *string*)

Generated for all NTP messages. Different from many other of Bro's events, this one is generated for both client-side and server-side messages.

See Wikipedia for more information about the NTP protocol.

**U** The connection record describing the corresponding UDP flow.

**Msg** The parsed NTP message.

**Excess** The raw bytes of any optional parts of the NTP packet. Bro does not further parse any optional fields.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### Bro::PIA

Analyzers implementing Dynamic Protocol

#### Components

*Analyzer::ANALYZER_PIA_TCP*

*Analyzer::ANALYZER_PIA_UDP*

### Bro::POP3

POP3 analyzer

#### Components

*Analyzer::ANALYZER_POP3*

#### Events

**pop3_request**

**Type** *event* (c: connection, is_orig: *bool*, command: *string*, arg: *string*)

Generated for client-side commands on POP3 connections.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Command** The command sent.

**Arg** The argument to the command.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**pop3_reply**

**Type** *event* (c: connection, is_orig: *bool*, cmd: *string*, msg: *string*)

Generated for server-side replies to commands on POP3 connections.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

**Is_orig** True if the command was sent by the originator of the TCP connection.

**Cmd** The success indicator sent by the server. This corresponds to the first token on the line sent, and should be either `OK` or `ERR`.

**Msg** The textual description the server sent along with *cmd*.

See also:

---

**Todo**

This event is receiving odd parameters, should unify.

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

### `pop3_data`

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, data: *`string`*)

Generated for server-side multi-line responses on POP3 connections. POP3 connections use multi-line responses to send bulk data, such as the actual mails. This event is generated once for each line that's part of such a response.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

**Is_orig** True if the data was sent by the originator of the TCP connection.

**Data** The data sent.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

### `pop3_unexpected`

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, msg: *`string`*, detail: *`string`*)

Generated for errors encountered on POP3 sessions. If the POP3 analyzer finds state transitions that do not conform to the protocol specification, or other situations it can't handle, it raises this event.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

**Is_orig** True if the data was sent by the originator of the TCP connection.

**Msg** A textual description of the situation.

**Detail** The input that triggered the event.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`pop3_starttls`**

**Type** *`event`* (c: `connection`)

Generated when a POP3 connection goes encrypted. While POP3 is by default a clear-text protocol, extensions exist to switch to encryption. This event is generated if that happens and the analyzer then stops processing the connection.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`pop3_login_success`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, user: *`string`*, password: *`string`*)

Generated for successful authentications on POP3 connections.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

**Is_orig** Always false.

**User** The user name used for authentication. The event is only generated if a non-empty user name was used.

**Password** The password used for authentication.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`pop3_login_failure`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, user: *`string`*, password: *`string`*)

Generated for unsuccessful authentications on POP3 connections.

See Wikipedia for more information about the POP3 protocol.

**C** The connection.

**Is_orig** Always false.

**User** The user name attempted for authentication. The event is only generated if a non-empty user name was used.

**Password** The password attempted for authentication.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

## Bro::RADIUS

RADIUS analyzer

### Components

*`Analyzer::ANALYZER_RADIUS`*

### Types

**`RADIUS::AttributeList`**

**Type** *`vector`* of *`string`*

**`RADIUS::Attributes`**

**Type** *`table`* [*`count`*] of *`RADIUS::AttributeList`*

**`RADIUS::Message`**

**Type** *`record`*

**code:** ***`count`*** The type of message (Access-Request, Access-Accept, etc.).

**trans_id:** ***`count`*** The transaction ID.

**authenticator:** ***`string`*** The "authenticator" string.

**attributes:** ***`RADIUS::Attributes &optional`*** Any attributes.

### Events

**`radius_message`**

**Type** *`event`* (c: `connection`, result: *`RADIUS::Message`*)

Generated for RADIUS messages.

See Wikipedia for more information about RADIUS.

**C** The connection.

**Result** A record containing fields parsed from a RADIUS packet.

**`radius_attribute`**

**Type** *`event`* (c: `connection`, attr_type: *`count`*, value: *`string`*)

Generated for each RADIUS attribute.

See Wikipedia for more information about RADIUS.

**C** The connection.

**Attr_type** The value of the code field (1 == User-Name, 2 == User-Password, etc.).

**Value** The data/value bound to the attribute.

## Bro::RDP

RDP analyzer

### Components

*`Analyzer::ANALYZER_RDP`*

### Types

**`RDP::EarlyCapabilityFlags`**

**Type** *`record`*

support_err_info_pdu: *`bool`*

want_32bpp_session: *`bool`*

support_statusinfo_pdu: *`bool`*

strong_asymmetric_keys: *`bool`*

support_monitor_layout_pdu: *`bool`*

support_netchar_autodetect: *`bool`*

support_dynvc_gfx_protocol: *`bool`*

support_dynamic_time_zone: *`bool`*

support_heartbeat_pdu: *`bool`*

**`RDP::ClientCoreData`**

**Type** *`record`*

version_major: *`count`*

version_minor: *`count`*

desktop_width: *`count`*

desktop_height: *count*

color_depth: *count*

sas_sequence: *count*

keyboard_layout: *count*

client_build: *count*

client_name: *string*

keyboard_type: *count*

keyboard_sub: *count*

keyboard_function_key: *count*

ime_file_name: *string*

post_beta2_color_depth: *count &optional*

client_product_id: *string &optional*

serial_number: *count &optional*

high_color_depth: *count &optional*

supported_color_depths: *count &optional*

ec_flags: *RDP::EarlyCapabilityFlags &optional*

dig_product_id: *string &optional*

### Events

**rdp_connect_request**

**Type** *event* (c: `connection`, cookie: *string*)

Generated for X.224 client requests.

**C** The connection record for the underlying transport-layer session/flow.

**Cookie** The cookie included in the request.

**rdp_negotiation_response**

**Type** *event* (c: `connection`, security_protocol: *count*)

Generated for RDP Negotiation Response messages.

**C** The connection record for the underlying transport-layer session/flow.

**Security_protocol** The security protocol selected by the server.

**rdp_negotiation_failure**

**Type** *event* (c: `connection`, failure_code: *count*)

Generated for RDP Negotiation Failure messages.

**C** The connection record for the underlying transport-layer session/flow.

**Failure_code** The failure code sent by the server.

**rdp_client_core_data**

**Type** *event* (c: `connection`, data: *RDP::ClientCoreData*)

Generated for MCS client requests.

**C** The connection record for the underlying transport-layer session/flow.

**Data** The data contained in the client core data structure.

**rdp_gcc_server_create_response**

**Type** *event* (c: connection, result: *count*)

Generated for MCS server responses.

**C** The connection record for the underlying transport-layer session/flow.

**Result** The 8-bit integer representing the GCC Conference Create Response result.

**rdp_server_security**

**Type** *event* (c: connection, encryption_method: *count*, encryption_level: *count*)

Generated for MCS server responses.

**C** The connection record for the underlying transport-layer session/flow.

**Encryption_method** The 32-bit integer representing the encryption method used in the connection.

**Encryption_level** The 32-bit integer representing the encryption level used in the connection.

**rdp_server_certificate**

**Type** *event* (c: connection, cert_type: *count*, permanently_issued: *bool*)

Generated for a server certificate section. If multiple X.509 certificates are included in chain, this event will still only be generated a single time.

**C** The connection record for the underlying transport-layer session/flow.

**Cert_type** Indicates the type of certificate.

**Permanently_issued** Value will be true is the certificate(s) is permanent on the server.

**rdp_begin_encryption**

**Type** *event* (c: connection, security_protocol: *count*)

Generated when an RDP session becomes encrypted.

**C** The connection record for the underlying transport-layer session/flow.

**Security_protocol** The security protocol being used for the session.

### Bro::RPC

Analyzers for RPC-based protocols

#### Components

*Analyzer::ANALYZER_CONTENTS_NFS*

*Analyzer::ANALYZER_CONTENTS_RPC*

*Analyzer::ANALYZER_NFS*

*Analyzer::ANALYZER_PORTMAPPER*

## Events

**`nfs_proc_null`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`)

Generated for NFSv3 request/reply dialogues of type *null*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_getattr`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, fh: *`string`*, attrs: `NFS3::fattr_t`)

Generated for NFSv3 request/reply dialogues of type *getattr*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Fh** TODO.

**Attrs** The attributes returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_lookup`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::diropargs_t`, rep: `NFS3::lookup_reply_t`)

Generated for NFSv3 request/reply dialogues of type *lookup*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** The arguments passed in the request.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**nfs_proc_read**

**Type** *event* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::readargs_t`, rep: `NFS3::read_reply_t`)

Generated for NFSv3 request/reply dialogues of type *read*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** The arguments passed in the request.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**nfs_proc_readlink**

**Type** *event* (c: `connection`, info: `NFS3::info_t`, fh: *string*, rep: `NFS3::readlink_reply_t`)

Generated for NFSv3 request/reply dialogues of type *readlink*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Fh** The file handle passed in the request.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_write`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::writeargs_t`, rep: `NFS3::write_reply_t`)

Generated for NFSv3 request/reply dialogues of type *write*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** TODO.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_create`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::diropargs_t`, rep: `NFS3::newobj_reply_t`)

Generated for NFSv3 request/reply dialogues of type *create*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** TODO.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_mkdir`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::diropargs_t`, rep: `NFS3::newobj_reply_t`)

Generated for NFSv3 request/reply dialogues of type *mkdir*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** TODO.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_remove`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::diropargs_t`, rep: `NFS3::delobj_reply_t`)

Generated for NFSv3 request/reply dialogues of type *remove*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** TODO.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`nfs_proc_rmdir`**

**Type** *`event`* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::diropargs_t`, rep: `NFS3::delobj_reply_t`)

Generated for NFSv3 request/reply dialogues of type *rmdir*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** TODO.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**nfs_proc_readdir**

**Type** *event* (c: `connection`, info: `NFS3::info_t`, req: `NFS3::readdirargs_t`, rep: `NFS3::readdir_reply_t`)

Generated for NFSv3 request/reply dialogues of type *readdir*. The event is generated once we have either seen both the request and its corresponding reply, or an unanswered request has timed out.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Req** TODO.

**Rep** The response returned in the reply. The values may not be valid if the request was unsuccessful.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**nfs_proc_not_implemented**

**Type** *event* (c: `connection`, info: `NFS3::info_t`, proc: `NFS3::proc_t`)

Generated for NFSv3 request/reply dialogues of a type that Bro's NFSv3 analyzer does not implement.

NFS is a service running on top of RPC. See Wikipedia for more information about the service.

**C** The RPC connection.

**Info** Reports the status of the dialogue, along with some meta information.

**Proc** The procedure called that Bro does not implement.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**nfs_reply_status**

**Type** *event* (n: `connection`, info: `NFS3::info_t`)

Generated for each NFSv3 reply message received, reporting just the status included.

**N** The connection.

**Info** Reports the status included in the reply.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**pm_request_null**

**Type** *event* (r: `connection`)

Generated for Portmapper requests of type *null*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**pm_request_set**

**Type** *event* (r: `connection`, m: `pm_mapping`, success: *bool*)

Generated for Portmapper request/reply dialogues of type *set*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**M** The argument to the request.

**Success** True if the request was successful, according to the corresponding reply. If no reply was seen, this will be false once the request times out.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**pm_request_unset**

**Type** *event* (r: `connection`, m: `pm_mapping`, success: *bool*)

Generated for Portmapper request/reply dialogues of type *unset*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**M** The argument to the request.

**Success** True if the request was successful, according to the corresponding reply. If no reply was seen, this will be false once the request times out.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**`pm_request_getport`**

**Type** *`event`* (r: `connection`, pr: `pm_port_request`, p: *`port`*)

Generated for Portmapper request/reply dialogues of type *getport*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Pr** The argument to the request.

**P** The port returned by the server.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**`pm_request_dump`**

**Type** *`event`* (r: `connection`, m: `pm_mappings`)

Generated for Portmapper request/reply dialogues of type *dump*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**M** The mappings returned by the server.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**pm_request_callit**

**Type** *event* (r: `connection`, call: `pm_callit_request`, p: *port*)

Generated for Portmapper request/reply dialogues of type *callit*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Call** The argument to the request.

**P** The port value returned by the call.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**pm_attempt_null**

**Type** *event* (r: `connection`, status: `rpc_status`)

Generated for failed Portmapper requests of type *null*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

---

**pm_attempt_set**

**Type** *event* (r: `connection`, status: `rpc_status`, m: `pm_mapping`)

Generated for failed Portmapper requests of type *set*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

**M** The argument to the original request.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**pm_attempt_unset**

**Type** *event* (r: `connection`, status: `rpc_status`, m: `pm_mapping`)

Generated for failed Portmapper requests of type *unset*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

**M** The argument to the original request.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**pm_attempt_getport**

**Type** *event* (r: `connection`, status: `rpc_status`, pr: `pm_port_request`)

Generated for failed Portmapper requests of type *getport*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

**Pr** The argument to the original request.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**pm_attempt_dump**

**Type** *event* (r: `connection`, status: `rpc_status`)

Generated for failed Portmapper requests of type *dump*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**pm_attempt_callit**

**Type** *event* (r: `connection`, status: `rpc_status`, call: `pm_callit_request`)

Generated for failed Portmapper requests of type *callit*.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

**Call** The argument to the original request.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**pm_bad_port**

**Type** *event* (r: `connection`, bad_p: *count*)

Generated for Portmapper requests or replies that include 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 >= 65536. If so, this event is generated.

Portmapper is a service running on top of RPC. See Wikipedia for more information about the service.

**R** The RPC connection.

**Bad_p** The invalid port value.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**rpc_dialogue**

**Type** *event* (c: `connection`, prog: *count*, ver: *count*, proc: *count*, status: `rpc_status`, start_time: *time*, call_len: *count*, reply_len: *count*)

Generated for RPC request/reply *pairs*. The RPC analyzer associates request and reply by their transaction identifiers and raises this event once both have been seen. If there's not a reply, this event will still be generated eventually on timeout. In that case, *status* will be set to `RPC_TIMEOUT`.

See Wikipedia for more information about the ONC RPC protocol.

> **C** The connection.
>
> **Prog** The remote program to call.
>
> **Ver** The version of the remote program to call.
>
> **Proc** The procedure of the remote program to call.
>
> **Status** The status of the reply, which should be one of the index values of `RPC_status`.
>
> **Start_time** The time when the *call* was seen.
>
> **Call_len** The size of the *call_body* PDU.
>
> **Reply_len** The size of the *reply_body* PDU.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

### `rpc_call`

> **Type** *`event`* (c: `connection`, xid: *`count`*, prog: *`count`*, ver: *`count`*, proc: *`count`*, call_len: *`count`*)

Generated for RPC *call* messages.

See Wikipedia for more information about the ONC RPC protocol.

> **C** The connection.
>
> **Xid** The transaction identifier allowing to match requests with replies.
>
> **Prog** The remote program to call.
>
> **Ver** The version of the remote program to call.
>
> **Proc** The procedure of the remote program to call.
>
> **Call_len** The size of the *call_body* PDU.

See also:

---

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

---

### `rpc_reply`

> **Type** *`event`* (c: `connection`, xid: *`count`*, status: `rpc_status`, reply_len: *`count`*)

Generated for RPC *reply* messages.

See Wikipedia for more information about the ONC RPC protocol.

> **C** The connection.

**Xid** The transaction identifier allowing to match requests with replies.

**Status** The status of the reply, which should be one of the index values of `RPC_status`.

**Reply_len** The size of the *reply_body* PDU.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to add a call to `Analyzer::register_for_ports` or a DPD payload signature.

### Bro::SIP

SIP analyzer UDP-only

#### Components

*`Analyzer::ANALYZER_SIP`*

#### Events

**`sip_request`**

**Type** *`event`* (c: `connection`, method: *`string`*, original_URI: *`string`*, version: *`string`*)

Generated for SIP (Session Initiation Protocol) requests, used in Voice over IP (VoIP).

This event is generated as soon as a request's initial line has been parsed.

See Wikipedia for more information about the SIP protocol.

**C** The connection.

**Method** The SIP method extracted from the request (e.g., `REGISTER`, `NOTIFY`).

**Original_URI** The unprocessed URI as specified in the request.

**Version** The version number specified in the request (e.g., `2.0`).

See also:

**`sip_reply`**

**Type** *`event`* (c: `connection`, version: *`string`*, code: *`count`*, reason: *`string`*)

Generated for SIP replies, used in Voice over IP (VoIP).

This event is generated as soon as a reply's initial line has been parsed.

See Wikipedia for more information about the SIP protocol.

**C** The connection.

**Version** The SIP version in use.

**Code** The response code.

**Reason** Textual details for the response code.

See also:

**sip_header**

**Type** *event* (c: connection, is_orig: *bool*, name: *string*, value: *string*)

Generated for each SIP header.

See Wikipedia for more information about the SIP protocol.

**C** The connection.

**Is_orig** Whether the header came from the originator.

**Name** Header name.

**Value** Header value.

See also:

**sip_all_headers**

**Type** *event* (c: connection, is_orig: *bool*, hlist: mime_header_list)

Generated once for all SIP headers from the originator or responder.

See Wikipedia for more information about the SIP protocol.

**C** The connection.

**Is_orig** Whether the headers came from the originator.

**Hlist** All the headers, and their values

See also:

**sip_begin_entity**

**Type** *event* (c: connection, is_orig: *bool*)

Generated at the beginning of a SIP message.

This event is generated as soon as a message's initial line has been parsed.

See Wikipedia for more information about the SIP protocol.

**C** The connection.

**Is_orig** Whether the message came from the originator.

See also:

**sip_end_entity**

**Type** *event* (c: connection, is_orig: *bool*)

Generated at the end of a SIP message.

See Wikipedia for more information about the SIP protocol.

**C** The connection.

**Is_orig** Whether the message came from the originator.

See also:

### Bro::SNMP

SNMP analyzer

### Components

*Analyzer::ANALYZER_SNMP*

### Types

**SNMP::Header**

**Type** *record*

version: *count*

**v1: *SNMP::HeaderV1 &optional*** Set when `version` is 0.

**v2: *SNMP::HeaderV2 &optional*** Set when `version` is 1.

**v3: *SNMP::HeaderV3 &optional*** Set when `version` is 3.

A generic SNMP header data structure that may include data from any version of SNMP. The value of the `version` field determines what header field is initialized.

**SNMP::HeaderV1**

**Type** *record*

community: *string*

The top-level message data structure of an SNMPv1 datagram, not including the PDU data. See **RFC 1157**.

**SNMP::HeaderV2**

**Type** *record*

community: *string*

The top-level message data structure of an SNMPv2 datagram, not including the PDU data. See **RFC 1901**.

**SNMP::HeaderV3**

**Type** *record*

id: *count*

max_size: *count*

flags: *count*

auth_flag: *bool*

priv_flag: *bool*

reportable_flag: *bool*

security_model: *count*

security_params: *string*

pdu_context: *SNMP::ScopedPDU_Context &optional*

The top-level message data structure of an SNMPv3 datagram, not including the PDU data. See **RFC 3412**.

**SNMP::PDU**

**Type** *record*

request_id: *int*

error_status: *int*

error_index: *int*

bindings: *SNMP::Bindings*

A PDU data structure from either **RFC 1157** or **RFC 3416**.

**SNMP::TrapPDU**

**Type** *record*

enterprise: *string*

agent: *addr*

generic_trap: *int*

specific_trap: *int*

time_stamp: *count*

bindings: *SNMP::Bindings*

A Trap-PDU data structure from **RFC 1157**.

**SNMP::BulkPDU**

**Type** *record*

request_id: *int*

non_repeaters: *count*

max_repititions: *count*

bindings: *SNMP::Bindings*

A BulkPDU data structure from **RFC 3416**.

**SNMP::ScopedPDU_Context**

**Type** *record*

engine_id: *string*

name: *string*

The ScopedPduData data structure of an SNMPv3 datagram, not including the PDU data (i.e. just the "context" fields). See **RFC 3412**.

**SNMP::ObjectValue**

**Type** *record*

tag: *count*

oid: *string &optional*

signed: *int &optional*

unsigned: *count &optional*

address: *addr &optional*

octets: *string &optional*

A generic SNMP object value, that may include any of the valid ObjectSyntax values from **RFC 1155** or **RFC 3416**. The value is decoded whenever possible and assigned to the appropriate field, which can be determined from the value of the tag field. For tags that can't be mapped to an appropriate type, the octets field holds the BER encoded ASN.1 content if there is any (though, octets is may also be used for other tags such as OCTET STRINGS or Opaque). Null values will only have their corresponding tag value set.

**SNMP::Binding**

**Type** *record*

oid: *string*

value: *SNMP::ObjectValue*

The VarBind data structure from either **RFC 1157** or **RFC 3416**, which maps an Object Identifier to a value.

**SNMP::Bindings**

**Type** *vector* of *SNMP::Binding*

A VarBindList data structure from either **RFC 1157** or **RFC 3416**. A sequences of *SNMP::Binding*, which maps an OIDs to values.

## Events

**snmp_get_request**

**Type** *event* (c: connection, is_orig: *bool*, header: *SNMP::Header*, pdu: *SNMP::PDU*)

An SNMP GetRequest-PDU message from either **RFC 1157** or **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**snmp_get_next_request**

**Type** *event* (c: connection, is_orig: *bool*, header: *SNMP::Header*, pdu: *SNMP::PDU*)

An SNMP GetNextRequest-PDU message from either **RFC 1157** or **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**snmp_response**

**Type** *event* (c: connection, is_orig: *bool*, header: *SNMP::Header*, pdu: *SNMP::PDU*)

An SNMP GetResponse-PDU message from **RFC 1157** or a Response-PDU from **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**snmp_set_request**

**Type** *event* (c: connection, is_orig: *bool*, header: *SNMP::Header*, pdu: *SNMP::PDU*)

An SNMP `SetRequest-PDU` message from either **RFC 1157** or **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**`snmp_trap`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, header: *`SNMP::Header`*, pdu: *`SNMP::TrapPDU`*)

An SNMP `Trap-PDU` message from **RFC 1157**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**`snmp_get_bulk_request`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, header: *`SNMP::Header`*, pdu: *`SNMP::BulkPDU`*)

An SNMP `GetBulkRequest-PDU` message from **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**`snmp_inform_request`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, header: *`SNMP::Header`*, pdu: *`SNMP::PDU`*)

An SNMP `InformRequest-PDU` message from **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**`snmp_trapV2`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, header: *`SNMP::Header`*, pdu: *`SNMP::PDU`*)

An SNMP `SNMPv2-Trap-PDU` message from **RFC 1157**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**snmp_report**

**Type** *event* (c: `connection`, is_orig: *bool*, header: *SNMP::Header*, pdu: *SNMP::PDU*)

An SNMP `Report-PDU` message from **RFC 3416**.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Pdu** An SNMP PDU data structure.

**snmp_unknown_pdu**

**Type** *event* (c: `connection`, is_orig: *bool*, header: *SNMP::Header*, tag: *count*)

An SNMP PDU message of unknown type.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Tag** The tag of the unknown SNMP PDU.

**snmp_unknown_scoped_pdu**

**Type** *event* (c: `connection`, is_orig: *bool*, header: *SNMP::Header*, tag: *count*)

An SNMPv3 `ScopedPDUData` of unknown type (neither plaintext or an encrypted PDU was in the datagram).

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**Tag** The tag of the unknown SNMP PDU scope.

**snmp_encrypted_pdu**

**Type** *event* (c: `connection`, is_orig: *bool*, header: *SNMP::Header*)

An SNMPv3 encrypted PDU message.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Header** SNMP version-dependent data that precedes PDU data in the top-level SNMP message structure.

**snmp_unknown_header_version**

**Type** *event* (c: `connection`, is_orig: *bool*, version: *count*)

A datagram with an unknown SNMP version.

**C** The connection over which the SNMP datagram is sent.

**Is_orig** The endpoint which sent the SNMP datagram.

**Version** The value of the unknown SNMP version.

### Bro::SMB

SMB analyzer

#### Components

*Analyzer::ANALYZER_CONTENTS_SMB*

*Analyzer::ANALYZER_SMB*

#### Events

**smb_message**

**Type** *event* (c: `connection`, hdr: `smb_hdr`, is_orig: *bool*, cmd: *string*, body_length: *count*, body: *string*)

Generated for all SMB/CIFS messages.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Is_orig** True if the message was sent by the originator of the underlying transport-level connection.

**Cmd** A string mnemonic of the SMB command code.

**Body_length** The length of the SMB message body, i.e. the data starting after the SMB header.

**Body** The raw SMB message body, i.e., the data starting after the SMB header.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**smb_com_tree_connect_andx**

**Type** *event* (c: `connection`, hdr: `smb_hdr`, path: *string*, service: *string*)

Generated for SMB/CIFS messages of type *tree connect andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Path** The `path` attribute specified in the message.

**Service** The `service` attribute specified in the message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_tree_disconnect`**

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`)

Generated for SMB/CIFS messages of type *tree disconnect*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_nt_create_andx`**

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`, name: *`string`*)

Generated for SMB/CIFS messages of type *nt create andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Name** The `name` attribute specified in the message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_transaction`**

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`, trans: `smb_trans`, data: `smb_trans_data`, is_orig: *`bool`*)

Generated for SMB/CIFS messages of type *nt transaction*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Trans** The parsed transaction header.

**Data** The raw transaction data.

**Is_orig** True if the message was sent by the originator of the connection.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### smb_com_transaction2

**Type** *event* (c: `connection`, hdr: `smb_hdr`, trans: `smb_trans`, data: `smb_trans_data`, is_orig: *bool*)

Generated for SMB/CIFS messages of type *nt transaction 2*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Trans** The parsed transaction header.

**Data** The raw transaction data.

**Is_orig** True if the message was sent by the originator of the connection.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### smb_com_trans_mailslot

**Type** *event* (c: `connection`, hdr: `smb_hdr`, trans: `smb_trans`, data: `smb_trans_data`, is_orig: *bool*)

Generated for SMB/CIFS messages of type *transaction mailslot*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Trans** The parsed transaction header.

**Data** The raw transaction data.

**Is_orig** True if the message was sent by the originator of the connection.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### `smb_com_trans_rap`

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`, trans: `smb_trans`, data: `smb_trans_data`, is_orig: *`bool`*)

Generated for SMB/CIFS messages of type *transaction rap*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Trans** The parsed transaction header.

**Data** The raw transaction data.

**Is_orig** True if the message was sent by the originator of the connection.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### `smb_com_trans_pipe`

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`, trans: `smb_trans`, data: `smb_trans_data`, is_orig: *`bool`*)

Generated for SMB/CIFS messages of type *transaction pipe*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Trans** The parsed transaction header.

**Data** The raw transaction data.

**Is_orig** True if the message was sent by the originator of the connection.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_read_andx`**

**Type** `event` (c: `connection`, hdr: `smb_hdr`, data: `string`)

Generated for SMB/CIFS messages of type *read andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Data** Always empty.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_write_andx`**

**Type** `event` (c: `connection`, hdr: `smb_hdr`, data: `string`)

Generated for SMB/CIFS messages of type *read andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Data** Always empty.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_get_dfs_referral`**

**Type** `event` (c: `connection`, hdr: `smb_hdr`, max_referral_level: `count`, file_name: `string`)

Generated for SMB/CIFS messages of type *get dfs referral*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Max_referral_level** The `max_referral_level` attribute specified in the message.

**File_name** The `filene_name` attribute specified in the message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_negotiate`**

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`)

Generated for SMB/CIFS messages of type *negotiate*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**`smb_com_negotiate_response`**

**Type** *`event`* (c: `connection`, hdr: `smb_hdr`, dialect_index: *`count`*)

Generated for SMB/CIFS messages of type *negotiate response*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Dialect_index** The `dialect` indicated in the message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**smb_com_setup_andx**

**Type** *event* (c: `connection`, hdr: `smb_hdr`)

Generated for SMB/CIFS messages of type *setup andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**smb_com_generic_andx**

**Type** *event* (c: `connection`, hdr: `smb_hdr`)

Generated for SMB/CIFS messages of type *generic andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**smb_com_close**

**Type** *event* (c: `connection`, hdr: `smb_hdr`)

Generated for SMB/CIFS messages of type *close*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**smb_com_logoff_andx**

**Type** *event* (c: connection, hdr: smb_hdr)

Generated for SMB/CIFS messages of type *logoff andx*.

See Wikipedia for more information about the SMB/CIFS protocol. Bro's SMB/CIFS analyzer parses both SMB-over-NetBIOS on ports 138/139 and SMB-over-TCP on port 445.

**C** The connection.

**Hdr** The parsed header of the SMB message.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

**smb_error**

**Type** *event* (c: connection, hdr: smb_hdr, cmd: *count*, cmd_str: *string*, data: *string*)

Generated for SMB/CIFS messages that indicate an error. This event is triggered by an SMB header including a status that signals an error.

**C** The connection.

**Hdr** The parsed header of the SMB message.

**Cmd** The SMB command code.

**Cmd_str** A string mnemonic of the SMB command code.

**Data** The raw SMB message body, i.e., the data starting after the SMB header.

See also:

**Todo**

Bro's current default configuration does not activate the protocol analyzer that generates this event; the corresponding script has not yet been ported to Bro 2.x. To still enable this event, one needs to register a port for it or add a DPD payload signature.

### Bro::SMTP

SMTP analyzer

### Components

*Analyzer::ANALYZER_SMTP*

### Events

**smtp_request**

**Type** *event* (c: `connection`, is_orig: *bool*, command: *string*, arg: *string*)

Generated for client-side SMTP commands.

See Wikipedia for more information about the SMTP protocol.

**C** The connection.

**Is_orig** True if the sender of the command is the originator of the TCP connection. Note that this is not redundant: the SMTP `TURN` command allows client and server to flip roles on established SMTP sessions, and hence a "request" might still come from the TCP-level responder. In practice, however, that will rarely happen as TURN is considered insecure and rarely used.

**Command** The request's command, without any arguments.

**Arg** The request command's arguments.

See also:

**Note:** Bro does not support the newer ETRN extension yet.

**smtp_reply**

**Type** *event* (c: `connection`, is_orig: *bool*, code: *count*, cmd: *string*, msg: *string*, cont_resp: *bool*)

Generated for server-side SMTP commands.

See Wikipedia for more information about the SMTP protocol.

**C** The connection.

**Is_orig** True if the sender of the command is the originator of the TCP connection. Note that this is not redundant: the SMTP `TURN` command allows client and server to flip roles on established SMTP sessions, and hence a "reply" might still come from the TCP-level originator. In practice, however, that will rarely happen as TURN is considered insecure and rarely used.

**Code** The reply's numerical code.

**Cmd** TODO.

**Msg** The reply's textual description.

**Cont_resp** True if the reply line is tagged as being continued to the next line. If so, further events will be raised and a handler may want to reassemble the pieces before processing the response any further.

See also:

**Note:** Bro doesn't support the newer ETRN extension yet.

**smtp_data**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, data: *`string`*)

Generated for DATA transmitted on SMTP sessions. This event is raised for subsequent chunks of raw data following the `DATA` SMTP command until the corresponding end marker `.` is seen. A handler may want to reassemble the pieces as they come in if stream-analysis is required.

See Wikipedia for more information about the SMTP protocol.

**C** The connection.

**Is_orig** True if the sender of the data is the originator of the TCP connection.

**Data** The raw data. Note that the size of each chunk is undefined and depends on specifics of the underlying TCP connection.

See also:

**Note:** This event receives the unprocessed raw data. There is a separate set of `mime_*` events that strip out the outer MIME-layer of emails and provide structured access to their content.

**`smtp_unexpected`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, msg: *`string`*, detail: *`string`*)

Generated for unexpected activity on SMTP sessions. The SMTP analyzer tracks the state of SMTP sessions and reports commands and other activity with this event that it sees even though it would not expect so at the current point of the communication.

See Wikipedia for more information about the SMTP protocol.

**C** The connection.

**Is_orig** True if the sender of the unexpected activity is the originator of the TCP connection.

**Msg** A descriptive message of what was unexpected.

**Detail** The actual SMTP line triggering the event.

See also:

**`smtp_starttls`**

**Type** *`event`* (c: `connection`)

Generated if a connection switched to using TLS using STARTTLS. After this event no more SMTP events will be raised for the connection. See the SSL analyzer for related SSL events, which will now be generated.

**C** The connection.

### Functions

**`skip_smtp_data`**

**Type** *`function`* (c: `connection`) : *any*

Skips SMTP data until the next email in a connection.

**C** The SMTP connection.

See also:

### Bro::SOCKS

SOCKS analyzer

#### Components

*Analyzer::ANALYZER_SOCKS*

#### Events

**socks_request**

> **Type** *event* (c: `connection`, version: *count*, request_type: *count*, sa: `SOCKS::Address`, p: *port*, user: *string*)

Generated when a SOCKS request is analyzed.

> **C** The parent connection of the proxy.
>
> **Version** The version of SOCKS this message used.
>
> **Request_type** The type of the request.
>
> **Sa** Address that the tunneled traffic should be sent to.
>
> **P** The destination port for the proxied traffic.
>
> **User** Username given for the SOCKS connection. This is not yet implemented for SOCKSv5.

**socks_reply**

> **Type** *event* (c: `connection`, version: *count*, reply: *count*, sa: `SOCKS::Address`, p: *port*)

Generated when a SOCKS reply is analyzed.

> **C** The parent connection of the proxy.
>
> **Version** The version of SOCKS this message used.
>
> **Reply** The status reply from the server.
>
> **Sa** The address that the server sent the traffic to.
>
> **P** The destination port for the proxied traffic.

**socks_login_userpass_request**

> **Type** *event* (c: `connection`, user: *string*, password: *string*)

Generated when a SOCKS client performs username and password based login.

> **C** The parent connection of the proxy.
>
> **User** The given username.
>
> **Password** The given password.

**socks_login_userpass_reply**

> **Type** *event* (c: `connection`, code: *count*)

Generated when a SOCKS server replies to a username/password login attempt.

> **C** The parent connection of the proxy.

**Code** The response code for the attempted login.

### Bro::SSH

Secure Shell analyzer

#### Components

*`Analyzer::ANALYZER_SSH`*

#### Types

**`SSH::Algorithm_Prefs`**

**Type** *`record`*

**client_to_server:** ***`vector`*** **of** ***`string &optional`*** The algorithm preferences for client to server communication

**server_to_client:** ***`vector`*** **of** ***`string &optional`*** The algorithm preferences for server to client communication

The client and server each have some preferences for the algorithms used in each direction.

**`SSH::Capabilities`**

**Type** *`record`*

**kex_algorithms:** **`string_vec`** Key exchange algorithms

**server_host_key_algorithms:** **`string_vec`** The algorithms supported for the server host key

**encryption_algorithms:** ***`SSH::Algorithm_Prefs`*** Symmetric encryption algorithm preferences

**mac_algorithms:** ***`SSH::Algorithm_Prefs`*** Symmetric MAC algorithm preferences

**compression_algorithms:** ***`SSH::Algorithm_Prefs`*** Compression algorithm preferences

**languages:** ***`SSH::Algorithm_Prefs &optional`*** Language preferences

**is_server:** ***`bool`*** Are these the capabilities of the server?

This record lists the preferences of an SSH endpoint for algorithm selection. During the initial SSH (Secure Shell) key exchange, each endpoint lists the algorithms that it supports, in order of preference. See **RFC 4253#section-7.1** for details.

#### Events

**`ssh_server_version`**

**Type** *`event`* (c: `connection`, version: *`string`*)

An SSH Protocol Version Exchange message from the server. This contains an identification string that's used for version identification. See **RFC 4253#section-4.2** for details.

**C** The connection over which the message was sent.

**Version** The identification string

See also:

**`ssh_client_version`**

**Type** *`event`* (c: `connection`, version: *`string`*)

An SSH Protocol Version Exchange message from the client. This contains an identification string that's used for version identification. See **RFC 4253#section-4.2** for details.

**C** The connection over which the message was sent.

**Version** The identification string

See also:

**`ssh_auth_successful`**

**Type** *`event`* (c: `connection`, auth_method_none: *`bool`*)

This event is generated when an SSH connection was determined to have had a successful authentication. This determination is based on packet size analysis, and errs on the side of caution - that is, if there's any doubt about the authentication success, this event is *not* raised.

**C** The connection over which the SSH connection took place.

**Auth_method_none** This is true if the analyzer detected a successful connection before any authentication challenge. The SSH protocol provides a mechanism for unauthenticated access, which some servers support.

See also:

**`ssh_auth_failed`**

**Type** *`event`* (c: `connection`)

This event is generated when an SSH connection was determined to have had a failed authentication. This determination is based on packet size analysis, and errs on the side of caution - that is, if there's any doubt about the authentication failure, this event is *not* raised.

**C** The connection over which the SSH connection took place.

See also:

**`ssh_capabilities`**

**Type** *`event`* (c: `connection`, cookie: *`string`*, capabilities: *`SSH::Capabilities`*)

During the initial SSH key exchange, each endpoint lists the algorithms that it supports, in order of preference. This event is generated for each endpoint, when the SSH_MSG_KEXINIT message is seen. See **RFC 4253#section-7.1** for details.

**C** The connection over which the SSH connection took place.

**Cookie** The SSH_MSG_KEXINIT cookie - a random value generated by the sender.

**Capabilities** The list of algorithms and languages that the sender advertises support for, in order of preference.

See also:

**`ssh2_server_host_key`**

**Type** *`event`* (c: `connection`, key: *`string`*)

During the SSH key exchange, the server supplies its public host key. This event is generated when the appropriate key exchange message is seen for SSH2.

**C** The connection over which the SSH connection took place.

**Key** The server's public host key. Note that this is the public key itself, and not just the fingerprint or hash.

See also:

**`ssh1_server_host_key`**

**Type** *`event`* (c: `connection`, p: *`string`*, e: *`string`*)

During the SSH key exchange, the server supplies its public host key. This event is generated when the appropriate key exchange message is seen for SSH1.

**C** The connection over which the SSH connection took place.

**P** The prime for the server's public host key.

**E** The exponent for the serer's public host key.

See also:

**`ssh_encrypted_packet`**

**Type** *`event`* (c: `connection`, orig: *`bool`*, len: *`count`*)

This event is generated when an SSH encrypted packet is seen. This event is not handled by default, but is provided for heuristic analysis scripts. Note that you have to set `SSH::skip_processing_after_detection` to false to use this event. This carries a performance penalty.

**C** The connection over which the SSH connection took place.

**Orig** Whether the packet was sent by the originator of the TCP connection.

**Len** The length of the SSH payload, in bytes. Note that this ignores reassembly, as this is unknown.

See also:

**`ssh2_dh_server_params`**

**Type** *`event`* (c: `connection`, p: *`string`*, q: *`string`*)

Generated if the connection uses a Diffie-Hellman Group Exchange key exchange method. This event contains the server DH parameters, which are sent in the SSH_MSG_KEY_DH_GEX_GROUP message as defined in **RFC 4419#section-3**.

**C** The connection.

**P** The DH prime modulus.

**Q** The DH generator.

See also:

**`ssh2_gss_error`**

**Type** *`event`* (c: `connection`, major_status: *`count`*, minor_status: *`count`*, err_msg: *`string`*)

In the event of a GSS-API error on the server, the server MAY send send an error message with some additional details. This event is generated when such an error message is seen. For more information, see **RFC 4462#section-2.1**.

**C** The connection.

**Major_status** GSS-API major status code.

**Minor_status** GSS-API minor status code.

**Err_msg** Detailed human-readable error message

See also:

**`ssh2_ecc_key`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, q: *`string`*)

The ECDH (Elliptic Curve Diffie-Hellman) and ECMQV (Elliptic Curve Menezes-Qu-Vanstone) key exchange algorithms use two ephemeral key pairs to generate a shared secret. This event is generated when either the client's or server's ephemeral public key is seen. For more information, see: **RFC 5656#section-4**.

**C** The connection

**Is_orig** Did this message come from the originator?

**Q** The ephemeral public key

See also:

## Bro::SSL

SSL/TLS and DTLS analyzers

### Components

*`Analyzer::ANALYZER_DTLS`*

*`Analyzer::ANALYZER_SSL`*

### Events

**`ssl_client_hello`**

**Type** *`event`* (c: `connection`, version: *`count`*, possible_ts: *`time`*, client_random: *`string`*, session_id: *`string`*, ciphers: `index_vec`)

Generated for an SSL/TLS client's initial *hello* message. SSL/TLS sessions start with an unencrypted handshake, and Bro extracts as much information out of that as it can. This event provides access to the initial information sent by the client.

See Wikipedia for more information about the SSL/TLS protocol.

**C** The connection.

**Version** The protocol version as extracted from the client's message. The values are standardized as part of the SSL/TLS protocol. The `SSL::version_strings` table maps them to descriptive names.

**Possible_ts** The current time as sent by the client. Note that SSL/TLS does not require clocks to be set correctly, so treat with care.

**Session_id** The session ID sent by the client (if any).

**Client_random** The random value sent by the client. For version 2 connections, the client challenge is returned.

**Ciphers** The list of ciphers the client offered to use. The values are standardized as part of the SSL/TLS protocol. The `SSL::cipher_desc` table maps them to descriptive names.

See also:

**`ssl_server_hello`**

**Type** *event* (c: `connection`, version: *count*, possible_ts: *time*, server_random: *string*, session_id: *string*, cipher: *count*, comp_method: *count*)

Generated for an SSL/TLS server's initial *hello* message. SSL/TLS sessions start with an unencrypted handshake, and Bro extracts as much information out of that as it can. This event provides access to the initial information sent by the client.

See Wikipedia for more information about the SSL/TLS protocol.

**C** The connection.

**Version** The protocol version as extracted from the server's message. The values are standardized as part of the SSL/TLS protocol. The `SSL::version_strings` table maps them to descriptive names.

**Possible_ts** The current time as sent by the server. Note that SSL/TLS does not require clocks to be set correctly, so treat with care.

**Session_id** The session ID as sent back by the server (if any).

**Server_random** The random value sent by the server. For version 2 connections, the connection-id is returned.

**Cipher** The cipher chosen by the server. The values are standardized as part of the SSL/TLS protocol. The `SSL::cipher_desc` table maps them to descriptive names.

**Comp_method** The compression method chosen by the client. The values are standardized as part of the SSL/TLS protocol.

See also:

**`ssl_extension`**

**Type** *event* (c: `connection`, is_orig: *bool*, code: *count*, val: *string*)

Generated for SSL/TLS extensions seen in an initial handshake. SSL/TLS sessions start with an unencrypted handshake, and Bro extracts as much information out of that as it can. This event provides access to any extensions either side sends as part of an extended *hello* message.

Note that Bro offers more specialized events for a few extensions.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Code** The numerical code of the extension. The values are standardized as part of the SSL/TLS protocol. The `SSL::extensions` table maps them to descriptive names.

**Val** The raw extension value that was sent in the message.

See also:

**`ssl_extension_elliptic_curves`**

**Type** *event* (c: `connection`, is_orig: *bool*, curves: `index_vec`)

Generated for an SSL/TLS Elliptic Curves extension. This TLS extension is defined in **RFC 4492** and sent by the client in the initial handshake. It gives the list of elliptic curves supported by the client.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Curves** List of supported elliptic curves.

See also:

**`ssl_extension_ec_point_formats`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, point_formats: `index_vec`)

Generated for an SSL/TLS Supported Point Formats extension. This TLS extension is defined in **RFC 4492** and sent by the client and/or server in the initial handshake. It gives the list of elliptic curve point formats supported by the client.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Point_formats** List of supported point formats.

See also:

**`ssl_server_curve`**

**Type** *`event`* (c: `connection`, curve: *`count`*)

Generated if a named curve is chosen by the server for an SSL/TLS connection. The curve is sent by the server in the ServerKeyExchange message as defined in **RFC 4492**, in case an ECDH or ECDHE cipher suite is chosen.

**C** The connection.

**Curve** The curve.

See also:

**`ssl_dh_server_params`**

**Type** *`event`* (c: `connection`, p: *`string`*, q: *`string`*, Ys: *`string`*)

Generated if a server uses a DH-anon or DHE cipher suite. This event contains the server DH parameters, which are sent in the ServerKeyExchange message as defined in **RFC 5246**.

**C** The connection.

**P** The DH prime modulus.

**Q** The DH generator.

**Ys** The server's DH public key.

See also:

**`ssl_extension_application_layer_protocol_negotiation`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, protocols: `string_vec`)

Generated for an SSL/TLS Application-Layer Protocol Negotiation extension. This TLS extension is defined in draft-ietf-tls-applayerprotoneg and sent in the initial handshake. It contains the list of client supported application protocols by the client or the server, respectively.

At the moment it is mostly used to negotiate the use of SPDY / HTTP2-drafts.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Protocols** List of supported application layer protocols.

See also:

**`ssl_extension_server_name`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, names: `string_vec`)

Generated for an SSL/TLS Server Name extension. This SSL/TLS extension is defined in **RFC 3546** and sent by the client in the initial handshake. It contains the name of the server it is contacting. This information can be used by the server to choose the correct certificate for the host the client wants to contact.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Names** A list of server names (DNS hostnames).

See also:

**`ssl_established`**

**Type** *`event`* (c: `connection`)

Generated at the end of an SSL/TLS handshake. SSL/TLS sessions start with an unencrypted handshake, and Bro extracts as much information out of that as it can. This event signals the time when an SSL/TLS has finished the handshake and its endpoints consider it as fully established. Typically, everything from now on will be encrypted.

See Wikipedia for more information about the SSL/TLS protocol.

**C** The connection.

See also:

**`ssl_alert`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, level: *`count`*, desc: *`count`*)

Generated for SSL/TLS alert records. SSL/TLS sessions start with an unencrypted handshake, and Bro extracts as much information out of that as it can. If during that handshake, an endpoint encounters a fatal error, it sends an *alert* record, that in turn triggers this event. After an *alert*, any endpoint may close the connection immediately.

See Wikipedia for more information about the SSL/TLS protocol.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Level** The severity level, as sent in the *alert*. The values are defined as part of the SSL/TLS protocol.

**Desc** A numerical value identifying the cause of the *alert*. The values are defined as part of the SSL/TLS protocol.

See also:

**`ssl_session_ticket_handshake`**

**Type** *`event`* (c: `connection`, ticket_lifetime_hint: *`count`*, ticket: *`string`*)

Generated for SSL/TLS handshake messages that are a part of the stateless-server session resumption mechanism. SSL/TLS sessions start with an unencrypted handshake, and Bro extracts as much information out of that as it can. This event is raised when an SSL/TLS server passes a session ticket to the client that can later be used for resuming the session. The mechanism is described in **RFC 4507**.

See Wikipedia for more information about the SSL/TLS protocol.

**C** The connection.

**Ticket_lifetime_hint** A hint from the server about how long the ticket should be stored by the client.

**Ticket** The raw ticket data.

See also:

**`ssl_heartbeat`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, length: *`count`*, heartbeat_type: *`count`*, payload_length: *`count`*, payload: *`string`*)

Generated for SSL/TLS heartbeat messages that are sent before session encryption starts. Generally heartbeat messages should rarely be seen in normal TLS traffic. Heartbeats are described in **RFC 6520**.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Length** length of the entire heartbeat message.

**Heartbeat_type** type of the heartbeat message. Per RFC, 1 = request, 2 = response.

**Payload_length** length of the payload of the heartbeat message, according to packet field.

**Payload** payload contained in the heartbeat message. Size can differ from payload_length, if payload_length and actual packet length disagree.

See also:

**`ssl_encrypted_data`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, content_type: *`count`*, length: *`count`*)

Generated for SSL/TLS messages that are sent after session encryption started.

Note that `SSL::disable_analyzer_after_detection` has to be changed from its default to false for this event to be generated.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Content_type** message type as reported by TLS session layer.

**Length** length of the entire heartbeat message.

See also:

**`ssl_stapled_ocsp`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, response: *`string`*)

This event contains the OCSP response contained in a Certificate Status Request message, when the client requested OCSP stapling and the server supports it. See description in **RFC 6066**.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Response** OCSP data.

**`ssl_handshake_message`**

**Type** *`event`* (c: `connection`, is_orig: *`bool`*, msg_type: *`count`*, length: *`count`*)

This event is raised for each unencrypted SSL/TLS handshake message.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

**Msg_type** Type of the handshake message that was seen.

**Length** Length of the handshake message that was seen.

See also:

**ssl_change_cipher_spec**

**Type** *event* (c: connection, is_orig: *bool*)

This event is raised when a SSL/TLS ChangeCipherSpec message is encountered before encryption begins. Traffic will be encrypted following this message.

**C** The connection.

**Is_orig** True if event is raised for originator side of the connection.

See also:

### Bro::SteppingStone

Stepping stone analyzer

### Components

*Analyzer::ANALYZER_STEPPINGSTONE*

### Events

**stp_create_endp**

**Type** *event* (c: connection, e: *int*, is_orig: *bool*)

Deprecated. Will be removed.

**stp_resume_endp**

**Type** *event* (e: *int*)

Event internal to the stepping stone detector.

**stp_correlate_pair**

**Type** *event* (e1: *int*, e2: *int*)

Event internal to the stepping stone detector.

**stp_remove_pair**

**Type** *event* (e1: *int*, e2: *int*)

Event internal to the stepping stone detector.

**stp_remove_endp**

**Type** *event* (e: *int*)

Event internal to the stepping stone detector.

### Bro::Syslog

Syslog analyzer UDP-only

### Components

*`Analyzer::ANALYZER_SYSLOG`*

### Events

**`syslog_message`**

**Type** *`event`* (c: `connection`, facility: *`count`*, severity: *`count`*, msg: *`string`*)

Generated for monitored Syslog messages.

See Wikipedia for more information about the Syslog protocol.

**C** The connection record for the underlying transport-layer session/flow.

**Facility** The "facility" included in the message.

**Severity** The "severity" included in the message.

**Msg** The message logged.

**Note:** Bro currently parses only UDP syslog traffic. Support for TCP syslog will be added soon.

## Bro::TCP

TCP analyzer

### Components

*`Analyzer::ANALYZER_CONTENTLINE`*

*`Analyzer::ANALYZER_CONTENTS`*

*`Analyzer::ANALYZER_TCP`*

*`Analyzer::ANALYZER_TCPSTATS`*

### Events

**`new_connection_contents`**

**Type** *`event`* (c: `connection`)

Generated when reassembly starts for a TCP connection. This event is raised at the moment when Bro's TCP analyzer enables stream reassembly for a connection.

**C** The connection.

See also:

**`connection_attempt`**

**Type** *`event`* (c: `connection`)

Generated for an unsuccessful connection attempt. This event is raised when an originator unsuccessfully attempted to establish a connection. "Unsuccessful" is defined as at least `tcp_attempt_delay` seconds having elapsed since the originator first sent a connection establishment packet to the destination without seeing a reply.

**C** The connection.

See also:

**`connection_established`**

**Type** *`event`* (c: `connection`)

Generated when seeing a SYN-ACK packet from the responder in a TCP handshake. An associated SYN packet was not seen from the originator side if its state is not set to `TCP_ESTABLISHED`. The final ACK of the handshake in response to SYN-ACK may or may not occur later, one way to tell is to check the *history* field of `connection` to see if the originator sent an ACK, indicated by 'A' in the history string.

**C** The connection.

See also:

**`partial_connection`**

**Type** *`event`* (c: `connection`)

Generated for a new active TCP connection if Bro did not see the initial handshake. This event is raised when Bro has observed traffic from each endpoint, but the activity did not begin with the usual connection establishment.

**C** The connection.

See also:

**`connection_partial_close`**

**Type** *`event`* (c: `connection`)

Generated when a previously inactive endpoint attempts to close a TCP connection via a normal FIN handshake or an abort RST sequence. When the endpoint sent one of these packets, Bro waits `tcp_partial_close_delay` prior to generating the event, to give the other endpoint a chance to close the connection normally.

**C** The connection.

See also:

**`connection_finished`**

**Type** *`event`* (c: `connection`)

Generated for a TCP connection that finished normally. The event is raised when a regular FIN handshake from both endpoints was observed.

**C** The connection.

See also:

**`connection_half_finished`**

**Type** *`event`* (c: `connection`)

Generated when one endpoint of a TCP connection attempted to gracefully close the connection, but the other endpoint is in the TCP_INACTIVE state. This can happen due to split routing, in which Bro only sees one side of a connection.

**C** The connection.

See also:

**connection_rejected**

**Type** *event* (c: connection)

Generated for a rejected TCP connection. This event is raised when an originator attempted to setup a TCP connection but the responder replied with a RST packet denying it.

**C** The connection.

See also:

**Note:** If the responder does not respond at all, *connection_attempt* is raised instead. If the responder initially accepts the connection but aborts it later, Bro first generates *connection_established* and then *connection_reset*.

**connection_reset**

**Type** *event* (c: connection)

Generated when an endpoint aborted a TCP connection. The event is raised when one endpoint of an established TCP connection aborted by sending a RST packet.

**C** The connection.

See also:

**connection_pending**

**Type** *event* (c: connection)

Generated for each still-open TCP connection when Bro terminates.

**C** The connection.

See also:

**connection_SYN_packet**

**Type** *event* (c: connection, pkt: SYN_packet)

Generated for a SYN packet. Bro raises this event for every SYN packet seen by its TCP analyzer.

**C** The connection.

**Pkt** Information extracted from the SYN packet.

See also:

**Note:** This event has quite low-level semantics and can potentially be expensive to generate. It should only be used if one really needs the specific information passed into the handler via the `pkt` argument. If not, handling one of the other `connection_*` events is typically the better approach.

**connection_first_ACK**

**Type** *event* (c: connection)

Generated for the first ACK packet seen for a TCP connection from its *originator*.

**C** The connection.

See also:

**Note:** This event has quite low-level semantics and should be used only rarely.

**connection_EOF**

**Type** *event* (c: `connection`, is_orig: *bool*)

Generated at the end of reassembled TCP connections. The TCP reassembler raised the event once for each endpoint of a connection when it finished reassembling the corresponding side of the communication.

**C** The connection.

**Is_orig** True if the event is raised for the originator side.

See also:

**tcp_packet**

**Type** *event* (c: `connection`, is_orig: *bool*, flags: *string*, seq: *count*, ack: *count*, len: *count*, payload: *string*)

Generated for every TCP packet. This is a very low-level and expensive event that should be avoided when at all possible. It's usually infeasible to handle when processing even medium volumes of traffic in real-time. It's slightly better than `new_packet` because it affects only TCP, but not much. That said, if you work from a trace and want to do some packet-level analysis, it may come in handy.

**C** The connection the packet is part of.

**Is_orig** True if the packet was sent by the connection's originator.

**Flags** A string with the packet's TCP flags. In the string, each character corresponds to one set flag, as follows: `S` -> SYN; `F` -> FIN; `R` -> RST; `A` -> ACK; `P` -> PUSH.

**Seq** The packet's relative TCP sequence number.

**Ack** If the ACK flag is set for the packet, the packet's relative ACK number, else zero.

**Len** The length of the TCP payload, as specified in the packet header.

**Payload** The raw TCP payload. Note that this may be shorter than *len* if the packet was not fully captured.

See also:

**tcp_option**

**Type** *event* (c: `connection`, is_orig: *bool*, opt: *count*, optlen: *count*)

Generated for each option found in a TCP header. Like many of the `tcp_*` events, this is a very low-level event and potentially expensive as it may be raised very often.

**C** The connection the packet is part of.

**Is_orig** True if the packet was sent by the connection's originator.

**Opt** The numerical option number, as found in the TCP header.

**Optlen** The length of the options value.

See also:

**Note:** There is currently no way to get the actual option value, if any.

**tcp_contents**

**Type** *event* (c: `connection`, is_orig: *bool*, seq: *count*, contents: *string*)

Generated for each chunk of reassembled TCP payload. When content delivery is enabled for a TCP connection (via `tcp_content_delivery_ports_orig`, `tcp_content_delivery_ports_resp`, `tcp_content_deliver_all_orig`, `tcp_content_deliver_all_resp`), this event is raised for each chunk of in-order payload reconstructed from the packet stream. Note that this event is potentially expensive if many connections carry significant amounts of data as then all that data needs to be passed on to the scripting layer.

**C** The connection the payload is part of.

**Is_orig** True if the packet was sent by the connection's originator.

**Seq** The sequence number corresponding to the first byte of the payload chunk.

**Contents** The raw payload, which will be non-empty.

See also:

**Note:** The payload received by this event is the same that is also passed into application-layer protocol analyzers internally. Subsequent invocations of this event for the same connection receive non-overlapping in-order chunks of its TCP payload stream. It is however undefined what size each chunk has; while Bro passes the data on as soon as possible, specifics depend on network-level effects such as latency, acknowledgements, reordering, etc.

**`tcp_rexmit`**

**Type** *event* (c: `connection`, is_orig: *bool*, seq: *count*, len: *count*, data_in_flight: *count*, window: *count*)

TODO.

**`contents_file_write_failure`**

**Type** *event* (c: `connection`, is_orig: *bool*, msg: *string*)

Generated when failing to write contents of a TCP stream to a file.

**C** The connection whose contents are being recorded.

**Is_orig** Which side of the connection encountered a failure to write.

**Msg** A reason or description for the failure.

See also:

## Functions

**`get_orig_seq`**

**Type** *function* (cid: `conn_id`) : *count*

Get the originator sequence number of a 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).

**Cid** The connection ID.

**Returns** The highest sequence number sent by a connection's originator, or 0 if *cid* does not point to an active TCP connection.

See also:

**`get_resp_seq`**

**Type** *function* (cid: `conn_id`) : *count*

Get the responder sequence number of a 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).

**Cid** The connection ID.

**Returns** The highest sequence number sent by a connection's responder, or 0 if *cid* does not point to an active TCP connection.

See also:

**`get_gap_summary`**

**Type** *function* () : `gap_info`

Returns statistics about TCP gaps.

**Returns** A record with TCP gap statistics.

See also:

**`set_contents_file`**

**Type** *function* (cid: `conn_id`, direction: *count*, f: *file*) : *bool*

Associates a file handle with a connection for writing TCP byte stream contents.

**Cid** The connection ID.

**Direction** Controls what sides of the connection to record. The argument can take one of the four values:

- `CONTENTS_NONE`: Stop recording the connection's content.
- `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. Results in the two directions being intermixed in the file, in the order the data was seen by Bro.

**F** The file handle of the file to write the contents to.

**Returns** Returns false if *cid* does not point to an active connection, and true otherwise.

**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; this can happen, e.g., due to an `ack_above_hole` event.

See also:

**`get_contents_file`**

**Type** *function* (cid: `conn_id`, direction: *count*) : *file*

Returns the file handle of the contents file of a connection.

**Cid** The connection ID.

**Direction** Controls what sides of the connection to record. See *set_contents_file* for possible values.

**Returns** The *file* handle for the contents file of the connection identified by *cid*. If the connection exists but there is no contents file for *direction*, then the function generates an error and returns a file handle to `stderr`.

See also:

### Bro::Teredo

Teredo analyzer

#### Components

*Analyzer::ANALYZER_TEREDO*

#### Events

**teredo_packet**

**Type** *event* (outer: connection, inner: teredo_hdr)

Generated for any IPv6 packet encapsulated in a Teredo tunnel. See **RFC 4380** for more information about the Teredo protocol.

**Outer** The Teredo tunnel connection.

**Inner** The Teredo-encapsulated IPv6 packet header and transport header.

See also:

**Note:** Since this event may be raised on a per-packet basis, handling it may become particularly expensive for real-time analysis.

**teredo_authentication**

**Type** *event* (outer: connection, inner: teredo_hdr)

Generated for IPv6 packets encapsulated in a Teredo tunnel that use the Teredo authentication encapsulation method. See **RFC 4380** for more information about the Teredo protocol.

**Outer** The Teredo tunnel connection.

**Inner** The Teredo-encapsulated IPv6 packet header and transport header.

See also:

**Note:** Since this event may be raised on a per-packet basis, handling it may become particularly expensive for real-time analysis.

**teredo_origin_indication**

**Type** *event* (outer: connection, inner: teredo_hdr)

Generated for IPv6 packets encapsulated in a Teredo tunnel that use the Teredo origin indication encapsulation method. See **RFC 4380** for more information about the Teredo protocol.

**Outer** The Teredo tunnel connection.

**Inner** The Teredo-encapsulated IPv6 packet header and transport header.

See also:

**Note:** Since this event may be raised on a per-packet basis, handling it may become particularly expensive for real-time analysis.

**`teredo_bubble`**

**Type** *`event`* (outer: `connection`, inner: `teredo_hdr`)

Generated for Teredo bubble packets. That is, IPv6 packets encapsulated in a Teredo tunnel that have a Next Header value of `IPPROTO_NONE`. See **RFC 4380** for more information about the Teredo protocol.

**Outer** The Teredo tunnel connection.

**Inner** The Teredo-encapsulated IPv6 packet header and transport header.

See also:

**Note:** Since this event may be raised on a per-packet basis, handling it may become particularly expensive for real-time analysis.

### Bro::UDP

UDP Analyzer

#### Components

*`Analyzer::ANALYZER_UDP`*

#### Events

**`udp_request`**

**Type** *`event`* (u: `connection`)

Generated for each packet sent by a UDP flow's originator. This a potentially expensive event due to the volume of UDP traffic and should be used with care.

**U** The connection record for the corresponding UDP flow.

See also:

**`udp_reply`**

**Type** *`event`* (u: `connection`)

Generated for each packet sent by a UDP flow's responder. This a potentially expensive event due to the volume of UDP traffic and should be used with care.

**U** The connection record for the corresponding UDP flow.

See also:

**`udp_contents`**

**Type** *`event`* (u: `connection`, is_orig: *`bool`*, contents: *`string`*)

Generated for UDP packets to pass on their payload. As the number of UDP packets can be very large, this event is normally raised only for those on ports configured in `udp_content_delivery_ports_orig` (for packets sent by the flow's originator) or `udp_content_delivery_ports_resp` (for packets sent by the flow's responder). However, delivery can be enabled for all UDP request and reply packets by setting `udp_content_deliver_all_orig` or `udp_content_deliver_all_resp`, respectively. Note that this event is also raised for all matching UDP packets, including empty ones.

**U** The connection record for the corresponding UDP flow.

**Is_orig** True if the event is raised for the originator side.

**Contents** TODO.

See also:

#### Bro::ZIP

Generic ZIP support analyzer

#### Components

*`Analyzer::ANALYZER_ZIP`*

### 3.2.9 File Analyzers

**Contents**

**`Files::Tag`**

**Type** *`enum`*

**`Files::ANALYZER_DATA_EVENT`**

**`Files::ANALYZER_EXTRACT`**

**`Files::ANALYZER_MD5`**

**`Files::ANALYZER_SHA1`**

**`Files::ANALYZER_SHA256`**

**`Files::ANALYZER_PE`**

**`Files::ANALYZER_UNIFIED2`**

**`Files::ANALYZER_X509`**

### Bro::FileDataEvent

Delivers file content

#### Components

*Files::ANALYZER_DATA_EVENT*

### Bro::FileExtract

Extract file content

#### Components

*Files::ANALYZER_EXTRACT*

#### Events

**file_extraction_limit**

**Type** *event* (f: `fa_file`, args: *any*, limit: *count*, len: *count*)

This event is generated when a file extraction analyzer is about to exceed the maximum permitted file size allowed by the *extract_limit* field of `Files::AnalyzerArgs`. The analyzer is automatically removed from file *f*.

**F** The file.

**Args** Arguments that identify a particular file extraction analyzer. This is only provided to be able to pass along to `FileExtract::set_limit`.

**Limit** The limit, in bytes, the extracted file is about to breach.

**Len** The length of the file chunk about to be written.

See also:

#### Functions

**FileExtract::__set_limit**

**Type** *function* (file_id: *string*, args: *any*, n: *count*) : *bool*

`FileExtract::set_limit`.

### Bro::FileHash

Hash file content

### Components

*Files::ANALYZER_MD5*

*Files::ANALYZER_SHA1*

*Files::ANALYZER_SHA256*

### Events

**file_hash**

**Type** *event* (f: fa_file, kind: *string*, hash: *string*)

This event is generated each time file analysis generates a digest of the file contents.

**F** The file.

**Kind** The type of digest algorithm.

**Hash** The result of the hashing.

See also:

## Bro::PE

Portable Executable analyzer

### Components

*Files::ANALYZER_PE*

### Events

**pe_dos_header**

**Type** *event* (f: fa_file, h: PE::DOSHeader)

A PE (Portable Executable) file DOS header was parsed. This is the top-level header and contains information like the size of the file, initial value of registers, etc.

**F** The file.

**H** The parsed DOS header information.

See also:

**pe_dos_code**

**Type** *event* (f: fa_file, code: *string*)

A PE file DOS stub was parsed. The stub is a valid application that runs under MS-DOS, by default to inform the user that the program can't be run in DOS mode.

**F** The file.

**Code** The DOS stub

See also:

**`pe_file_header`**

**Type** *`event`* (f: `fa_file`, h: `PE::FileHeader`)

A PE file file header was parsed. This header contains information like the target machine, the timestamp when the file was created, the number of sections, and pointers to other parts of the file.

**F** The file.

**H** The parsed file header information.

See also:

**`pe_optional_header`**

**Type** *`event`* (f: `fa_file`, h: `PE::OptionalHeader`)

A PE file optional header was parsed. This header is required for executable files, but not for object files. It contains information like OS requirements to execute the file, the original entry point address, and information needed to load the file into memory.

**F** The file.

**H** The parsed optional header information.

See also:

**`pe_section_header`**

**Type** *`event`* (f: `fa_file`, h: `PE::SectionHeader`)

A PE file section header was parsed. This header contains information like the section name, size, address, and characteristics.

**F** The file.

**H** The parsed section header information.

See also:

## Bro::Unified2

Analyze Unified2 alert files.

### Components

*`Files::ANALYZER_UNIFIED2`*

### Types

**`Unified2::IDSEvent`**

**Type** *`record`*

sensor_id: *`count`*

event_id: *`count`*

ts: *`time`*

signature_id: *`count`*

generator_id: *`count`*

signature_revision: *count*

classification_id: *count*

priority_id: *count*

src_ip: *addr*

dst_ip: *addr*

src_p: *port*

dst_p: *port*

impact_flag: *count*

impact: *count*

blocked: *count*

**mpls_label:** ***count &optional*** Not available in "legacy" IDS events.

**vlan_id:** ***count &optional*** Not available in "legacy" IDS events.

**packet_action:** ***count &optional*** Only available in "legacy" IDS events.

**`Unified2::Packet`**

**Type** *record*

sensor_id: *count*

event_id: *count*

event_second: *count*

packet_ts: *time*

link_type: *count*

data: *string*

## Events

**`unified2_event`**

**Type** *event* (f: `fa_file`, ev: *Unified2::IDSEvent*)

Abstract all of the various Unified2 event formats into a single event.

**F** The file.

**Ev** TODO.

**`unified2_packet`**

**Type** *event* (f: `fa_file`, pkt: *Unified2::Packet*)

The Unified2 packet format event.

**F** The file.

**Pkt** TODO.

## Bro::X509

X509 analyzer

### Components

*Files::ANALYZER_X509*

### Types

**X509::Certificate**

**Type** *record*

**version:** ***count &log*** Version number.

**serial:** ***string &log*** Serial number.

**subject:** ***string &log*** Subject.

**issuer:** ***string &log*** Issuer.

**cn:** ***string &optional*** Last (most specific) common name.

**not_valid_before:** ***time &log*** Timestamp before when certificate is not valid.

**not_valid_after:** ***time &log*** Timestamp after when certificate is not valid.

**key_alg:** ***string &log*** Name of the key algorithm

**sig_alg:** ***string &log*** Name of the signature algorithm

**key_type:** ***string &optional &log*** Key type, if key parseable by openssl (either rsa, dsa or ec)

**key_length:** ***count &optional &log*** Key length in bits

**exponent:** ***string &optional &log*** Exponent, if RSA-certificate

**curve:** ***string &optional &log*** Curve, if EC-certificate

**X509::Extension**

**Type** *record*

**name:** ***string*** Long name of extension. oid if name not known

**short_name:** ***string &optional*** Short name of extension if known

**oid:** ***string*** Oid of extension

**critical:** ***bool*** True if extension is critical

**value:** ***string*** Extension content parsed to string for known extensions. Raw data otherwise.

**X509::BasicConstraints**

**Type** *record*

**ca:** ***bool &log*** CA flag set?

**path_len:** ***count &optional &log*** Maximum path length

**Attributes** *&log*

**X509::SubjectAlternativeName**

**Type** *record*

**dns:** **string_vec** ***&optional &log*** List of DNS entries in SAN

**uri:** **string_vec** ***&optional &log*** List of URI entries in SAN

**email:** `string_vec` *`&optional`* *`&log`* List of email entries in SAN

**ip:** `addr_vec` *`&optional`* *`&log`* List of IP entries in SAN

**other_fields:** *`bool`* True if the certificate contained other, not recognized or parsed name fields

**`X509::Result`**

**Type** *`record`*

**result:** *`int`* OpenSSL result code

**result_string:** *`string`* Result as string

**chain_certs:** *`vector`* **of** *`opaque`* **of x509** *`&optional`* References to the final certificate chain, if verification successful. End-host certificate is first.

Result of an X509 certificate chain verification

## Events

**`x509_certificate`**

**Type** *`event`* (f: `fa_file`, cert_ref: *`opaque`* of x509, cert: *`X509::Certificate`*)

Generated for encountered X509 certificates, e.g., in the clear SSL/TLS connection handshake.

See Wikipedia for more information about the X.509 format.

**F** The file.

**Cert_ref** An opaque pointer to the underlying OpenSSL data structure of the certificate.

**Cert** The parsed certificate information.

See also:

**`x509_extension`**

**Type** *`event`* (f: `fa_file`, ext: *`X509::Extension`*)

Generated for X509 extensions seen in a certificate.

See Wikipedia for more information about the X.509 format.

**F** The file.

**Ext** The parsed extension.

See also:

**`x509_ext_basic_constraints`**

**Type** *`event`* (f: `fa_file`, ext: *`X509::BasicConstraints`*)

Generated for the X509 basic constraints extension seen in a certificate. This extension can be used to identify the subject of a certificate as a CA.

**F** The file.

**Ext** The parsed basic constraints extension.

See also:

**`x509_ext_subject_alternative_name`**

**Type** *`event`* (f: `fa_file`, ext: *`X509::SubjectAlternativeName`*)

Generated for the X509 subject alternative name extension seen in a certificate. This extension can be used to allow additional entities to be bound to the subject of the certificate. Usually it is used to specify one or multiple DNS names for which a certificate is valid.

**F** The file.

**Ext** The parsed subject alternative name extension.

See also:

## Functions

### `x509_parse`

**Type** *`function`* (cert: *`opaque`* of x509) : *`X509::Certificate`*

Parses a certificate into an X509::Certificate structure.

**Cert** The X509 certificate opaque handle.

**Returns** A X509::Certificate structure.

See also:

### `x509_get_certificate_string`

**Type** *`function`* (cert: *`opaque`* of x509, pem: *`bool`* *`&default`* = `F` *`&optional`*) : *`string`*

Returns the string form of a certificate.

**Cert** The X509 certificate opaque handle.

**Pem** A boolean that specifies if the certificate is returned in pem-form (true), or as the raw ASN1 encoded binary (false).

**Returns** X509 certificate as a string.

See also:

### `x509_ocsp_verify`

**Type** *`function`* (certs: `x509_opaque_vector`, ocsp_reply: *`string`*, root_certs: `table_string_of_string`, verify_time: *`time`* *`&default`* = `0.0` *`&optional`*) : *`X509::Result`*

Verifies an OCSP reply.

**Certs** Specifies the certificate chain to use. Server certificate first.

**Ocsp_reply** the ocsp reply to validate.

**Root_certs** A list of root certificates to validate the certificate chain.

**Verify_time** Time for the validity check of the certificates.

**Returns** A record of type X509::Result containing the result code of the verify operation.

See also:

### `x509_verify`

**Type** *`function`* (certs: `x509_opaque_vector`, root_certs: `table_string_of_string`, verify_time: *`time`* *`&default`* = `0.0` *`&optional`*) : *`X509::Result`*

Verifies a certificate.

**Certs** Specifies a certificate chain that is being used to validate the given certificate against the root store given in *root_certs*. The host certificate has to be at index 0.

**Root_certs** A list of root certificates to validate the certificate chain.

**Verify_time** Time for the validity check of the certificates.

**Returns** A record of type X509::Result containing the result code of the verify operation. In case of success also returns the full certificate chain.

See also:

### 3.2.10 Bro Package Index

Bro has the following script packages (e.g. collections of related scripts in a common directory). If the package directory contains a `__load__.bro` script, it supports being loaded in mass as a whole directory for convenience.

Packages/scripts in the `base/` directory are all loaded by default, while ones in `policy/` provide functionality and customization options that are more appropriate for users to decide whether they'd like to load it or not.

base/frameworks/broker

The Broker communication framework facilitates connecting to remote Bro instances to share state and transfer events.

base/frameworks/logging

The logging framework provides a flexible key-value based logging interface.

base/frameworks/logging/postprocessors

Support for postprocessors in the logging framework.

base/frameworks/input

The input framework provides a way to read previously stored data either as an event stream or into a Bro table.

base/frameworks/analyzer

The analyzer framework allows to dynamically enable or disable Bro's protocol analyzers, as well as to manage the well-known ports which automatically activate a particular analyzer for new connections.

base/frameworks/files

The file analysis framework provides an interface for driving the analysis of files, possibly independent of any network protocol over which they're transported.

base/frameworks/files/magic

base/bif

base/bif/plugins

base/frameworks/reporter

This framework is intended to create an output and filtering path for internally generated messages/warnings/errors.

base/frameworks/notice

The notice framework enables Bro to "notice" things which are odd or potentially bad, leaving it to the local configuration to define which of them are actionable. This decoupling of detection and reporting allows Bro to be customized to the different needs that sites have.

base/frameworks/cluster

The cluster framework provides for establishing and controlling a cluster of Bro instances.

base/frameworks/control

The control framework provides the foundation for providing “commands” that can be taken remotely at runtime to modify a running Bro instance or collect information from the running instance.

base/frameworks/dpd

The DPD (dynamic protocol detection) activates port-independent protocol detection and selectively disables analyzers if protocol violations occur.

base/frameworks/signatures

The signature framework provides for doing low-level pattern matching. While signatures are not Bro’s preferred detection tool, they sometimes come in handy and are closer to what many people are familiar with from using other NIDS.

base/frameworks/packet-filter

The packet filter framework supports how Bro sets its BPF capture filter.

base/frameworks/software

The software framework provides infrastructure for maintaining a table of software versions seen on the network. The version parsing itself is carried out by external protocol-specific scripts that feed into this framework.

base/frameworks/communication

The communication framework facilitates connecting to remote Bro or Broccoli instances to share state and transfer events.

base/frameworks/intel

The intelligence framework provides a way to store and query intelligence data (such as IP addresses or strings). Metadata can also be associated with the intelligence.

base/frameworks/sumstats

The summary statistics framework provides a way to summarize large streams of data into simple reduced measurements.

base/frameworks/sumstats/plugins

Plugins for the summary statistics framework.

base/frameworks/tunnels

The tunnels framework handles the tracking/logging of tunnels (e.g. Teredo, AYIYA, or IP-in-IP such as 6to4 where “IP” is either IPv4 or IPv6).

base/protocols/conn

Support for connection (TCP, UDP, or ICMP) analysis.

base/protocols/dhcp

Support for Dynamic Host Configuration Protocol (DHCP) analysis.

base/protocols/dnp3

Support for Distributed Network Protocol (DNP3) analysis.

base/protocols/dns

Support for Domain Name System (DNS) protocol analysis.

base/protocols/ftp

Support for File Transfer Protocol (FTP) analysis.

base/protocols/ssl

Support for Secure Sockets Layer (SSL) protocol analysis.

base/files/x509

Support for X509 certificates with the file analysis framework.

base/files/hash

Support for file hashes with the file analysis framework.

base/protocols/http

Support for Hypertext Transfer Protocol (HTTP) analysis.

base/protocols/irc

Support for Internet Relay Chat (IRC) protocol analysis.

base/protocols/krb

Support for Kerberos protocol analysis.

base/protocols/modbus

Support for Modbus protocol analysis.

base/protocols/mysql

Support for MySQL protocol analysis.

base/protocols/pop3

Support for POP3 (Post Office Protocol) protocol analysis.

base/protocols/radius

Support for RADIUS protocol analysis.

base/protocols/rdp

Support for Remote Desktop Protocol (RDP) analysis.

base/protocols/sip

Support for Session Initiation Protocol (SIP) analysis.

base/protocols/snmp

Support for Simple Network Management Protocol (SNMP) analysis.

base/protocols/smtp

Support for Simple Mail Transfer Protocol (SMTP) analysis.

base/protocols/socks

Support for Socket Secure (SOCKS) protocol analysis.

base/protocols/ssh

Support for SSH protocol analysis.

base/protocols/syslog

Support for Syslog protocol analysis.

base/protocols/tunnels

base/files/pe

Support for Portable Executable (PE) file analysis.

base/files/extract

Support for extracing files with the file analysis framework.

base/files/unified2

Support for Unified2 files in the file analysis framework.

broxygen

This package is loaded during the process which automatically generates reference documentation for all Bro scripts (i.e. "Broxygen"). Its only purpose is to provide an easy way to load all known Bro scripts plus any extra scripts needed or used by the documentation process.

policy/frameworks/intel/seen

Scripts that send data to the intelligence framework.

policy/integration/barnyard2

Integration with Barnyard2.

policy/integration/collective-intel

The scripts in this module are for deeper integration with the Collective Intelligence Framework (CIF) since Bro's Intel framework doesn't natively behave the same as CIF nor does it store and maintain the same data in all cases.

policy/misc/app-stats

AppStats collects information about web applications in use on the network.

policy/misc/app-stats/plugins

Plugins for AppStats.

policy/misc/detect-traceroute

Detect hosts that are running traceroute.

policy/tuning

Miscellaneous tuning parameters.

policy/tuning/defaults

Sets various defaults, and prints warning messages to stdout under certain conditions.

### 3.2.11 Bro Script Index

## 3.3 Subcomponents

The following are snapshots of documentation for components that come with this version of Bro (2.4). Since they can also be used independently, see the download page for documentation of any current, independent component releases.

### 3.3.1 BinPAC

BinPAC is a high level language for describing protocol parsers and generates C++ code. It is currently maintained and distributed with the Bro Network Security Monitor distribution, however, the generated parsers may be used with other programs besides Bro.

**Contents**

### Download

You can find the latest BinPAC release for download at http://www.bro.org/download.

BinPAC's git repository is located at git://git.bro.org/binpac.git. You can browse the repository here.

This document describes BinPAC 0.44-24. See the `CHANGES` file for version history.

### Prerequisites

BinPAC relies on the following libraries and tools, which need to be installed before you begin:

- **Flex (Fast Lexical Analyzer)** Flex is already installed on most systems, so with luck you can skip having to install it yourself.
- **Bison (GNU Parser Generator)** Bison is also already installed on many system.
- **CMake 2.6.3 or greater** CMake is a cross-platform, open-source build system, typically not installed by default. See http://www.cmake.org for more information regarding CMake and the installation steps below for how to use it to build this distribution. CMake generates native Makefiles that depend on GNU Make by default

### Installation

To build and install into `/usr/local`:

```
./configure
cd build
make
make install
```

This will perform an out-of-source build into the build directory using the default build options and then install the binpac binary into `/usr/local/bin`.

You can specify a different installation directory with:

```
./configure --prefix=<dir>
```

Run `./configure --help` for more options.

### Glossary and Convention

To make this document easier to read, the following are the glossary and convention used.

- PAC grammar - .pac file written by user.
- PAC source - _pac.cc file generated by binpac

- PAC header - _pac.h file generated by binpac
- Analyzer - Protocol decoder generated by compiling PAC grammar
- Field - a member of a record
- Primary field - member of a record as direct result of parsing
- Derivative field - member of a record evaluated through post processing

### BinPAC Language Reference

BinPAC language consists of:

- analyzer
- type - data structure like definition describing parsing unit. Types can built on each other to form more complex type similar to yacc productions.
- flow - "flow" defines how data will be fed into the analyzer and the top level parsing unit.
- Keywords
- Built-in macros

### Defining an analyzer

There are two components to an analyzer definition: the top level context and the connection definition.

#### Context Definition

Each analyzer requires a top level context defined by the following syntax:

Typically top level context contains pointer to top level analyzer and connection definition like below:

#### Connection Definition

A "connection" defines the entry point into the analyzer. It consists of two "flow" definitions, an "upflow" and a "downflow".

Example:

### type

A "type" is the basic building block of binpac-generated parser, and describes the structure of a byte segment. Each non-primitive "type" generates a C++ class that can independently parse the structure which it describes.

Syntax:

Example:

PAC grammar:

```
type myType = record {
   data:uint8;
};
```

PAC header:

```
class myType{
public:
   myType();
   ~myType();
   int Parse(const_byteptr const t_begin_of_data, const_byteptr const t_end_of_data);
   uint8 data() const  { return data_; }
protected:
   uint8 data_;
};
```

## Primitives

Primitive type can be treated as #define in C language. They are embedded into other type which reference them but do not generate any parsing code of their own. Available primitive types are:

- int8
- int16
- int32
- uint8
- uint16
- uint32
- Regular expression ( `type HTTP_URI = RE/[[:alnum:][:punct:]]+/;` )
- bytestring

Examples:

is equivalent to:

(Note: this behavior may change in future versions of binpac.)

## record

A "record" composes primitive type(s) and other record(s) to create new "type". This new "type" in turn can be used as part of parent type or directly for parsing.

Example:

## case

The "case" compositor allows switching between different parsing methods.

A "case" supports an optional "default" label to denote none of the above labels are matched. If no fields follow a given label, a user can specify an arbitrary field name with the "empty" type. See the following example.

Note that only one field is allowed after a given label. If multiple fields are to be specified, they should be packed in another "record" type first. The other usages of *case* are described later.

### array

A type can be defined as a sequence of "single-type elements". By default, array type continue parsing for the array element in an infinite loop. Or an array size can be specified to control the number of match. &until can be also conditionally end parsing:

Array can also be used directly inside of "record". For example:

### flow

A "flow" defines how data is fed into the analyzer. It also maintains custom state information declared by *%member*. flow is configured by specifiying type of data unit.

Syntax:

When "flow" is added to top level context analyzer, it enables use of &online and &length in "record" type. flow buffers data when there is not enough to evaluate the record and dispatchs data for evaluation when the threshold is reached.

### flowunit

When flowunit is used, the analyzer uses flow buffer to handle incremental input and provide support for &oneline/&length. For further detail on this, see *Buffering*.

### datagram

Opposite to flowunit, by declaring data unit as datagram, flow buffer is opted out. This results in faster parsing but no incremental input or buffering support.

### Byte Ordering and Alignment

### Byte Ordering

### Byte Alignment

### Functions

User can define functions in binpac. Function can be declared using one of the three ways:

### PAC with embedded body

PAC style function prototype and embed the body using %{ %}:

```
function print_stuff(value :const_bytestring):bool
%{
   printf("Value [%s]\n", std_str(value).c_str());
%}
```

### PAC with PAC-case body

Pac style function with a case body, this type of declaration is useful for extending later by casefunc:

```
function RPC_Service(prog: uint32, vers: uint32): EnumRPCService =
   case prog of {
       default -> RPC_SERVICE_UNKNOWN;
   };
```

### Inlined by %code

Function can be completely inlined by using %code:

```
%code{
EnumRPCService RPC_Service(const RPC_Call* call)
   {
   return call ? call->service() : RPC_SERVICE_UNKNOWN;
   }
%}
```

### Extending

PAC code can be extended by using “refine”. This is useful for code reusing and splitting functionality for parallel development.

### Extending record

Record can be extended to add addtional attribute(s) by using “refine typeattr”. One of the typical use is to add &let for split protocol parsing from protocol analysis.

### Extending type case

### Extending function case

Function which is declared as a PAC case can be extended by adding additional case into the switch.

### Extending connection

Connection can be extended to add functions and members. Example:

```
refine connection RPC_Conn += {
   function ProcessPortmapReply(results: PortmapResults): bool
       %{
       %}
};
```

### State Management

State is maintained by extending parsing class by declaring derivative. State lasts until the top level parsing unit (flowunit/datagram is destroyed).

## Keywords

### Source code embedding

C++ code can be embedded within the .pac file using the following directives. These code will be copied into the final generated code.

- %header{...%}

  Code to be inserted in binpac generated header file.

- %code{...%}

  Code to be inserted at the beginning of binpac generated C++ file.

- %member{...%}

  Add additional member(s) to connection (?) and flow class.

- %init{...%}

  Code to be inserted in flow constructor.

- %cleanup{...%}

  Code to be inserted in flow destructor.

### Embedded pac primitive

- ${
- $set{
- $type{
- $typeof{
- $const_def{

## Condition checking

### &until

"&until" is used in conjunction with array declaration. It specifies exit condition for array parsing.

### &requires

Process data dependencies before evaluating field.

Example: typically, derivative field is evaluated after primary field. However "&requires" is used to force evaluate of length before msg_body.

#### &if

Evaluate field only if condition is met.

#### case

There are two uses to the "case" keyword.

- As part of record field. In this scenario, it allow alternative methods to parse a field. Example:

```
type RPC_Reply(msg: RPC_Message) = record {
  stat:       uint32;
  reply:      case stat of {
      MSG_ACCEPTED -> areply:  RPC_AcceptedReply(call);
      MSG_DENIED   -> rreply:  RPC_RejectedReply(call);
  };
} &let {
  call: RPC_Call = context.connection.FindCall(msg.xid);
  success: bool = (stat == MSG_ACCEPTED && areply.stat == SUCCESS);
};
```

- As function definition. Example:

```
function RPC_Service(prog: uint32, vers: uint32): EnumRPCService =
    case prog of {
            default -> RPC_SERVICE_UNKNOWN;
    };
```

Note that one can "refine" both types of cases:

### Built-in macros

#### $input

This macro refers to the data that was passed into the ParseBuffer function. When $input is used, binpac generate a const_bytestring which contains the start and end pointer of the input.

PAC grammar:

```
&until($input.length()==0);
```

PAC source:

```
const_bytestring t_val__elem_input(t_begin_of_data, t_end_of_data);
if (  ( t_val__elem_input.length() == 0 )   )
```

#### $element

$element provides access to entry of the array type. Following are the ways which $element can be used.

- Current element. Check on the value of the most recently parsed entry. This would get executed after each time an entry is parsed. Example:

```
type SMB_ascii_string       = uint8[] &until($element == 0);
```

- Current element's field. Example:

```
type DNS_label(msg: DNS_message) = record {
   length:      uint8;
   data:        case label_type of {
       0 ->     label:  bytestring &length = length;
       3 ->     ptr_lo: uint8;
   };
} &let {
   label_type: uint8 = length >> 6;
   last:       bool  = (length == 0) || (label_type == 3);
};
type DNS_name(msg: DNS_message) = record {
   labels:      DNS_label(msg)[] &until($element.last);
};
```

#### $context

This macro refers to the Analyzer context class (Context<Name> class gets generated from analyzer <Name> with-context {}). Using this macro, users can gain access to the "flow" object and "analyzer" object.

### Other keywords

#### &transient

Do not create copy of the bytestring

#### &let

Adds derivative field to a record

#### let

Declares global value. If the user does not specify a type, the compiler will assume the "int" type.

PAC grammar:

```
let myValue:uint8=10;
```

PAC source:

```
uint8 const myValue = 10;
```

PAC header:

```
extern uint8 const myValue;
```

### &restofdata

Grab the rest of the data available in the FlowBuffer.

PAC grammar:

```
onebyte: uint8;
value: bytestring &restofdata &transient;
```

PAC source:

```
// Parse "onebyte"
onebyte_ = *((uint8 const *) (t_begin_of_data));
// Parse "value"
int t_value_string_length;
t_value_string_length = (t_end_of_data) - ((t_begin_of_data + 1));
int t_value__size;
t_value__size = t_value_string_length;
value_.init((t_begin_of_data + 1), t_value_string_length);
```

### &length

Length can appear in two different contexts: as property of a field or as property of a record. Examples: &length as field property:

```
protocol    : bytestring &length = 4;
```

translates into:

```
const_byteptr t_end_of_data = t_begin_of_data + 4;
int t_protocol_string_length;
t_protocol_string_length = 4;
int t_protocol__size;
t_protocol__size = t_protocol_string_length;
protocol_.init(t_begin_of_data, t_protocol_string_length);
```

### &check

Check a condition and raise exception if not met.

### &chunked and $chunk

When parsing a long field with variable length, “chunked” can be used to improve performance. However, chunked field are not buffered across packet. Data for the chunk in the current packet can be access by using “$chunk”.

### &exportsourcedata

Data matched for a particular type, the data matched can be retained by using “&exportsourcedata”.

.pac file

_pac.h

_pac.cc

Source data can be used within the type that match it or at the parent type.

translates into

### &refcount

### withinput

### Parsing Methodology

### Buffering

binpac supports incremental input to deal with packet fragmentation. This is done via use of FlowBuffer class and maintaining buffering/parsing states.

#### FlowBuffer Class

FlowBuffer provides two mode of buffering: line and frame. Line mode is useful for parsing line based language like HTTP. Frame mode is best for fixed length message. Buffering mode can be switched during parsing and is done transparently to the grammar writer.

At compile time binpac calculates number of bytes required to evaluate each field. During run time, data is buffered up in FlowBuffer until there is enough to evaluate the "record". To optimize the buffering process, if FlowBuffer has enough data to evaluate on the first NewData, it would only mark the start and end pointer instead of copying.

- void **NewMessage**();
  - Advances the orig_data_begin_ pointer depend on current mode_. Moves by 1/2 characters in LINE_MODE, by frame_length_ in FRAME_MODE and nothing in UNKNOWN_MODE (default mode).
  - Set buffer_n_ to 0
  - Reset message_complete_
- void **NewLine**();
  - Reset frame_length_ and chunked_, set mode_ to LINE_MODE
- void **NewFrame**(int frame_length, bool chunked_);
- void **GrowFrame**(int new_frame_length);
- void **AppendToBuffer**(const_byteptr data, int len);
  - Reallocate buffer_ to add new data then copy data
- void **ExpandBuffer**(int length);
  - Reallocate buffer_ to new size if new size is bigger than current size.
  - Set minimum size to 512 (optimization?)
- void **MarkOrCopyLine**();
  - Seek current input for end of line (CR/LF/CRLF depend on line break mode). If found append found data to buffer if one is already created or mark (set frame_length_) if one is not created (to minimize copying). If end of line is not found, append partial data till end of input to buffer. Buffer is created if one is not there.

- const_byteptr **begin**()/**end**()
  - Returns buffer_ and buffer_n_ if a buffer exist, otherwise orig_data_begin_ and orig_data_begin_ + frame_length_.

### Parsing States

- buffering_state_ - each parsing class contains a flag indicating whether there are enough data buffered to evaluate the next block.
- parsing_state_ - each parsing class which consists of multiple parsing data unit (line/frames) has this flag indicating the parsing stage. Each time new data comes in, it invokes parsing function and switch on parsing_state to determine which sub parser to use next.

### Regular Expression

### Evaluation Order

### Running Binpac-generated Analyzer Standalone

To run binpac-generated code independent of Bro. Regex library must be substituted. Below is one way of doing it. Use the following three header files.

#### RE.h

#### bro_dummy.h

#### binpac_pcre.h

#### main.cc

In your main source, add this dummy stub.

### Q & A

- Does &oneline only work when "flow" is used?

  Yes. binpac uses the flowunit definition in "flow" to figure out which types require buffering. For those that do, the parse function is:

  And the code of flow_buffer_t provides the functionality of buffering up to one line. That's why &oneline is only active when "flow" is used and the type requires buffering.

  In certain cases we would want to use &oneline even if the type does not require buffering, binpac currently does not provide such functionality.
- How would incremental input work in the case of regex?

  A regex should not take incremental input. (The binpac compiler will complain when that happens.) It should always appear below some type that has either &length=... or &oneline.
- What is the role of Context_<Name> class (generated by analyzer <Name> withcontext)?

- What is the difference between ‘’withcontext” and w/o ‘’withcontext’‘?

  withcontext should always be there. It’s fine to have an empty context.

- Elaborate on $context and how it is related to “withcontext”.

  A “context” parameter is passed to every type. It provides a vehicle to pass something to every type without adding a parameter to every type. In that sense, it’s optional. It exists for convenience.

- Example usage of composite type array.

  Please see HTTP_Headers in http-protocol.pac in the Bro source code.

- Clarification on “connection” keyword (binpac paper).
- Need a new way to attach hook additional code to each class beside &let.
- &transient, how is this different from declaring anonymous field? and currently it doesn’t seem to do much
- Detail on the globals ($context, $element, $input...etc)
- How does BinPAC work with dynamic protocol detection?

  Well, you can use the code in DNS-binpac.cc as a reference. First, create a pointer to the connection. (See the example in DNS-binpac.cc)

  Pass the data received from “DeliverPacket” or “DeliverStream” to “interp->NewData()”. (Again, see the example in DNS-binpac.cc)

- Explanation of &withinput
- Difference between using flow and not using flow (binpac generates Parse method instead of ParseBuffer)
- &check currently working?
- Difference between flowunit and datagram, datagram and &oneline, &length?
- Go over TODO list in binpac release
- How would input get handle/buffered when length is not known (chunked)
- More feature multi byte character? utf16 utf32 etc.

## TODO List

### New Features

- Provides a method to match simple ascii text.
- Allows use fixed length array in addition to vector.

### Bugs

### Small clean-ups

- Remove anonymous field bytestring assignment.
- Redundant overflow checking/more efficient fixed length text copying.

#### Warning/Errors

Things that compiler should flag out at code generation time

- Give warning when &transient is used on none bytestring
- Give warning when &oneline, &length is used and flowunit is not.
- Warning when more than one "connection" is defined

## 3.3.2 Broccoli: The Bro Client Communications Library

Broccoli is the "Bro client communications library". It allows you to create client sensors for the Bro intrusion detection system. Broccoli can speak a good subset of the Bro communication protocol, in particular, it can receive Bro IDs, send and receive Bro events, and send and receive event requests to/from peering Bros. You can currently create and receive values of pure types like integers, counters, timestamps, IP addresses, port numbers, booleans, and strings.

#### Download

You can find the latest Broccoli release for download at http://www.bro.org/download.

Broccoli's git repository is located at git://git.bro.org/broccoli. You can browse the repository here.

This document describes Broccoli 1.97-14. See the `CHANGES` file for version history.

#### Installation

The Broccoli library has been tested on Linux, the BSDs, and Solaris. A Windows build has not currently been tried but is part of our future plans. If you succeed in building Broccoli on other platforms, let us know!

#### Prerequisites

Broccoli relies on the following libraries and tools, which need to be installed before you begin:

**Flex (Fast Lexical Analyzer)** Flex is already installed on most systems, so with luck you can skip having to install it yourself.

**Bison (GNU Parser Generator)** This comes with many systems, but if you get errors compiling parse.y, you will need to install it.

**OpenSSL headers and libraries** For encrypted communication. These are likely installed, though some platforms may require installation of a 'devel' package for the headers.

**CMake 2.6.3 or greater** CMake is a cross-platform, open-source build system, typically not installed by default. See http://www.cmake.org for more information regarding CMake and the installation steps below for how to use it to build this distribution. CMake generates native Makefiles that depend on GNU Make by default.

Broccoli can also make use of some optional libraries if they are found at installation time:

**Libpcap headers and libraries** Network traffic capture library

### Installation

To build and install into `/usr/local`:

```
./configure
make
make install
```

This will perform an out-of-source build into the build directory using the default build options and then install libraries into `/usr/local/lib`.

You can specify a different installation directory with:

```
./configure --prefix=<dir>
```

Or control the python bindings install destination more precisely with:

```
./configure --python-install-dir=<dir>
```

Run `./configure --help` for more options.

Further notable configure options:

**`--enable-debug`** This one enables lots of debugging output. Be sure to disable this when using the library in a production environment! The output could easily end up in undersired places when the stdout of the program you've instrumented is used in other ways.

**`--with-configfile=FILE`** Broccoli can read key/value pairs from a config file. By default it is located in the etc directory of the installation root (exception: when using `--prefix=/usr`, `/etc` is used instead of /usr/etc). The default config file name is broccoli.conf. Using `--with-configfile`, you can override the location and name of the config file.

To use the library in other programs & configure scripts, use the `broccoli-config` script. It gives you the necessary configuration flags and linker flags for your system, see `--cflags` and `--libs`.

The API is contained in broccoli.h and pretty well documented. A few usage examples can be found in the test directory, in particular, the `broping` tool can be used to test event transmission and reception. Have a look at the policy file `broping.bro` for the events that need to be defined at the peering Bro. Try `broping -h` for a look at the available options.

Broccoli knows two kinds of version numbers: the release version number (as in "broccoli-x.y.tar.gz", or as shipped with Bro) and the shared library API version number (as in libbroccoli.so.3.0.0). The former relates to changes in the tree, the latter to compatibility changes in the API.

Comments, feedback and patches are appreciated; please check the Bro website.

### Documentation

Please see the Broccoli User Manual and the Broccoli API Reference.

## 3.3.3 Broccoli: The Bro Client Communications Library

This page documents Broccoli, the Bro client communications library. It allows you to create client sensors for the Bro intrusion detection system. Broccoli can speak a good subset of the Bro communication protocol, in particular, it can receive Bro IDs, send and receive Bro events, and send and receive event requests to/from peering Bros.

**Contents**

## Introduction

### What is Broccoli?

Broccoli is the BRO Client COmmunications LIbrary. It allows you to write applications that speak the communication protocol of the Bro intrusion detection system.

Broccoli is free software under terms of the BSD license as given in the COPYING file distributed with its source code.

In this document, we assume that you are familiar with the basic concepts of Bro, so please first review the documentation/publications available from the Bro website if necessary.

Feedback, patches and bug reports are all welcome, please see http://www.bro.org/community for instructions on how to participate in the Bro community.

### Why do I care?

Having a single IDS on your network is good, but things become a lot more interesting when you can communicate information among multiple vantage points in your network. Bro agents can communicate with other Bro agents, sending and receiving events and other state information. In the Bro context this is particularly interesting because it means that you can build sophisticated policy-controlled distributed event management systems.

Broccoli enters the picture when it comes to integrating components that are not Bro agents themselves. Broccoli lets you create applications that can speak the Bro communication protocol. You can compose, send, request, and receive events. You can register your own event handlers. You can talk to other Broccoli applications or Bro agents – Bro agents cannot tell whether they are talking to another Bro or a Broccoli application. Broccoli allows you to integrate applications of your choosing into a distributed policy-controlled event management system. Broccoli is intended to be portable: it should build on Linux, the BSDs, Solaris, and Windows (in the MinGW environment).

Unlike other distributed IDSs, Bro does not assume a strict sensor-manager hierarchy in the information flow. Instead, Bro agents can request delivery of arbitrary *events* from other instances. When an event is triggered in a Bro agent, it checks whether any connected agents have requested notification of this event, and sends a *copy* of the event, including the *event arguments*. Recall that in Bro, an event handler is essentially a function defined in the Bro language, and an event materializes through invocation of an event handler. Each remote agent can define its own event handlers.

Broccoli applications will typically do one or more of the following:

- *Configuration/Management Tasks:* the Broccoli application is used to configure remotely running Bros without the need for a restart.
- *Interfacing with other Systems:* the Broccoli application is used to convert Bro events to other alert/notice formats, or into syslogd entries.
- *Host-based Sensor Feeds into Bro:* the Broccoli application reports events based on host-based activity generated in kernel space or user space applications.

### Installing Broccoli

The installation process will hopefully be painless: Broccoli is installed from source using the usual `./configure <options> && make && make install` routine after extraction of the tarball.

Some relevant configuration options to pass to configure are:

- `--prefix=<DIR>`: sets the installation root to DIR. The default is to install below `/usr/local`.
- `--enable-debug`: enables debugging output. Please refer to the *Configuring Debugging Output* section for details on configuring and using debugging output.

- `--with-configfile=<FILE>`: use FILE as location of configuration file. See the section on *Configuration Files* for more on this.
- `--with-openssl=<DIR>`: use the OpenSSL installation below DIR.

After installation, you'll find the library in shared and static versions in `<prefix>/lib`, the header file for compilation in `<prefix>/include`.

## Using Broccoli

### Obtaining information about your build using `broccoli-config`

Similarly to many other software packages, the Broccoli distribution provides a script that you can use to obtain details about your Broccoli setup. The script currently provides the following flags:

- `--build` prints the name of the machine the build was made on, when, and whether debugging support was enabled or not.
- `--prefix` prints the directory in the filesystem below which Broccoli was installed.
- `--version` prints the version of the distribution you have installed.
- `--libs` prints the flags to pass to the linker in order to link in the Broccoli library.
- `--cflags` prints the flags to pass to the compiler in order to properly include Broccoli's header file.
- `--config` prints the location of the system-wide config file your installation will use.

The `--cflags` and `--libs` flags are the suggested way of obtaining the necessary information for integrating Broccoli into your build environment. It is generally recommended to use `broccoli-config` for this purpose, rather than, say, develop new **autoconf** tests. If you use the **autoconf/automake** tools, we recommend something along the following lines for your `configure` script:

```
dnl ##################################################
dnl # Check for Broccoli
dnl ##################################################
AC_ARG_WITH(broccoli-config,
    AC_HELP_STRING(\[--with-broccoli-config=FILE], \[Use given broccoli-config]),
    [ brocfg="$withval" ],
    [ AC_PATH_GENERIC(broccoli,,
        brocfg="broccoli-config",
        AC_MSG_ERROR(Cannot find Broccoli: Is broccoli-config in path? Use more␣
↪fertilizer?)) ])

broccoli_libs=`$brocfg --libs`
broccoli_cflags=`$brocfg --cflags`
AC_SUBST(broccoli_libs)
AC_SUBST(broccoli_cflags)``
```

You can then use the compiler/linker flags in your Makefile.in/ams by substituting in the values accordingly, which might look as follows:

```
CFLAGS = -W -Wall -g -DFOOBAR @broccoli_cflags@
LDFLAGS = -L/usr/lib/foobar @broccoli_libs@
```

### Suggestions for instrumenting applications

Often you will want to make existing applications Bro-aware, that is, *instrument* them so that they can send and receive Bro events at appropriate moments in the execution flow. This will involve modifying an existing code tree, so care needs to be taken to avoid unwanted side effects. By protecting the instrumented code with `#ifdef`/`#endif` statements you can still build the original application, using the instrumented source tree. The `broccoli-config` script helps you in doing so because it already adds `-DBROCCOLI` to the compiler flags reported when run with the `--cflags` option:

So simply surround all inserted code with a preprocessor check for `BROCCOLI` and you will be able to build the original application as soon as `BROCCOLI` is not defined.

### The Broccoli API

Time for some code. In the code snippets below we will introduce variables whenever context requires them and not necessarily when C requires them. In order to make the API known, include `broccoli.h`:

**Note:** *Broccoli's memory management philosophy:*

Broccoli generally does not release objects you allocate. The approach taken is "you clean up what you allocate."

### Initialization

Broccoli requires global initialization before most of its other functions can be used. Generally, the way to initialize Broccoli is as follows:

The argument to `bro_init()` provides optional initialization context, and may be kept `NULL` for normal use. If required, you may allocate a `BroCtx` structure locally, initialize it using `bro_ctx_init()`, fill in additional values as required and subsequently pass it to `bro_init()`:

**Note:** The `BroCtx` structure currently contains a set of five different callback function pointers. These are *required* for thread-safe operation of OpenSSL (Broccoli itself is thread-safe). If you intend to use Broccoli in a multithreaded environment, you need to implement functions and register them via the `BroCtx` structure. The O'Reilly book "Network Security with OpenSSL" by Viega et al. shows how to implement these callbacks.

**Warning:** You *must* call `bro_init()` at the start of your application. Undefined behavior may result if you don't.

### Data types in Broccoli

Broccoli declares a number of data types in `broccoli.h` that you should know about. The more complex ones are kept opaque, while you do get access to the fields in the simpler ones. The full list is as follows:

- Simple signed and unsigned types: int, uint, uint16, uint32, uint64 and uchar.
- Connection handles: BroConn, kept opaque.
- Bro events: BroEvent, kept opaque.
- Buffer objects: BroBuf, kept opaque. See also *Using Dynamic Buffers*.

- Ports: BroPort for network ports, defined as follows:
- Records: BroRecord, kept opaque. See also *Handling Records*.
- Strings (character and binary): BroString, defined as follows:
- BroStrings are mostly kept transparent for convenience; please have a look at the *Broccoli API Reference*.
- Tables: BroTable, kept opaque. See also *Handling Tables*.
- Sets: BroSet, kept opaque. See also *Handling Sets*.
- IP Address: BroAddr, defined as follows:

  Both IPv4 and IPv6 addresses are supported, with the former occupying only the first 4 bytes of the `addr` array.
- Subnets: BroSubnet, defined as follows:

## Managing Connections

You can use Broccoli to establish a connection to a remote Bro, or to create a Broccoli-enabled server application that other Bros will connect to (this means that in principle, you can also use Broccoli purely as middleware and have multiple Broccoli applications communicate directly).

In order to establish a connection to a remote Bro, you first obtain a connection handle. You then use this connection handle to request events, connect to the remote Bro, send events, etc. Connection handles are pointers to `BroConn` structures, which are kept opaque. Use `bro_conn_new()` or `bro_conn_new_str()` to obtain a handle, depending on what parameters are more convenient for you: the former accepts the IP address and port number as separate numerical arguments, the latter uses a single string to encode both, in “hostname:port” format.

To write a Broccoli-enabled server, you first need to implement the usual `socket()` / `bind()` / `listen()` / `accept()` routine. Once you have obtained a file descriptor for the new connection from `accept()`, you pass it to the third function that returns a `BroConn` handle, `bro_conn_new_socket()`. The rest of the connection handling then proceeds as in the client scenario.

All three calls accept additional flags for fine-tuning connection behaviour. These flags are:

- `BRO_CFLAG_NONE`: no functionality. Use when no flags are desired.
- `BRO_CFLAG_RECONNECT`: When using this option, Broccoli will attempt to reconnect to the peer transparently after losing connectivity. Essentially whenever you try to read from or write to the peer and its connection has broke down, a full reconnect including complete handshaking is attempted. You can check whether the connection to a peer is alive at any time using `bro_conn_alive()`.
- `BRO_CFLAG_ALWAYS_QUEUE`: When using this option, Broccoli will queue any events you send for later transmission when a connection is currently down. Without using this flag, any events you attempt to send while a connection is down get dropped on the floor. Note that Broccoli maintains a maximum queue size per connection so if you attempt to send lots of events while the connection is down, the oldest events may start to get dropped nonetheless. Again, you can check whether the connection is currently okay by using `bro_conn_alive()`.
- `BRO_CFLAG_DONTCACHE`: When using this option, Broccoli will ask the peer not to use caching on the objects it sends to us. This is the default, and the flag need not normally be used. It is kept to maintain backward compatibility.
- `BRO_CFLAG_CACHE`: When using this option, Broccoli will ask the peer to use caching on the objects it sends to us. Caching is normally disabled.
- `BRO_CFLAG_YIELD`: When using this option, `bro_conn_process_input()` processes at most one event at a time and then returns.

By obtaining a connection handle, you do not also establish a connection right away. This is done using `bro_conn_connect()`. The main reason for this is to allow you to subscribe to events (using `bro_event_registry_add()`, see *Receiving Events*) before establishing the connection. Upon returning from `bro_conn_connect()` you are guaranteed to receive all instances of the event types you have requested, while later on during the connection some time may elapse between the issuing of a request for events and the processing of that request at the remote end. Connections are established via TCP, optionally using SSL encryption. See "*Configuring Encrypted Communication*", for more information on setting up encryption. The port numbers Bro agents and Broccoli applications listen on can vary from peer to peer.

Finally, `bro_conn_delete()` terminates a connection and releases all resources associated with it. You can create as many connections as you like, to one or more peers. You can obtain the file descriptor of a connection using `bro_conn_get_fd()`:

Or simply use the string-based version:

### Connection Classes

When you want to establish connections from multiple Broccoli applications with different purposes, the peer needs a means to understand what kind of application each connection belongs to. The real meaning of "kind of application" here is "sets of event types to request", because depending on the class of an application, the peer will likely want to receive different types of events.

Broccoli lets you set the class of a connection using `bro_conn_set_class()`. When using this feature, you need to call that function before issuing a `bro_conn_connect()` since the class of a connection is determined at connection startup:

If your peer is a Bro node, you need to match the chosen connection class in the remote Bro's `Communication::nodes` configuration. See *Configuring event reception in Bro scripts*, for how to do this. Finally, in order to obtain the class of a connection as indicated by the remote side, use `bro_conn_get_peer_class()`.

### Composing and sending events

In order to send an event to the remote Bro agent, you first create an empty event structure with the name of the event, then add parameters to pass to the event handler at the remote agent, and then send off the event.

Let's assume we want to request a report of all connections a remote Bro currently keeps state for that match a given destination port and host name and that have amassed more than a certain number of bytes. The idea is to send an event to the remote Bro that contains the query, identifiable through a request ID, and have the remote Bro answer us with `remote_conn` events containing the information we asked for. The definition of our requesting event could look as follows in the Bro policy:

First, create a new event:

Now we need to add parameters to the event. The sequence and types must match the event handler declaration – check the Bro policy to make sure they match. The function to use for adding parameter values is `bro_event_add_val()`. All values are passed as *pointer arguments* and are copied internally, so the object you're pointing to stays unmodified at all times. You clean up what you allocate. In order to indicate the type of the value passed into the function, you need to pass a numerical type identifier along as well. *Table-1* lists the value types that Broccoli supports along with the type identifier and data structures to point to.

### Types, type tags, and data structures for event parameters in Broccoli

| Type | Type tag | Data type pointed to |
|---|---|---|
| Boolean | `BRO_TYPE_BOOL` | `int` |
| Integer value | `BRO_TYPE_INT` | `int64` |
| Counter (nonnegative integers) | `BRO_TYPE_COUNT` | `uint64` |
| Enums (enumerated values) | `BRO_TYPE_ENUM` | `uint64` (see also description of `bro_event_add_val()`'s `type_name` argument) |
| Floating-point number | `BRO_TYPE_DOUBLE` | `double` |
| Timestamp | `BRO_TYPE_TIME` | `double` (see also `bro_util_timeval_to_double()` and `bro_util_current_time()`) |
| Time interval | `BRO_TYPE_INTERVAL` | `double` |
| Strings (text and binary) | `BRO_TYPE_STRING` | `BroString` (see also family of `bro_string_xxx()` functions) |
| Network ports | `BRO_TYPE_PORT` | `BroPort`, with the port number in host byte order |
| IPv4/IPv6 address | `BRO_TYPE_IPADDR` | `BroAddr`, with the `addr` member in network byte order and `size` member indicating the address family and number of 4-byte words of `addr` that are occupied (1 for IPv4 and 4 for IPv6) |
| IPv4/IPv6 subnet | `BRO_TYPE_SUBNET` | `BroSubnet`, with the `sn_net` member in network byte order |
| Record | `BRO_TYPE_RECORD` | `BroRecord` (see also the family of `bro_record_xxx()` functions and their explanation below) |
| Table | `BRO_TYPE_TABLE` | `BroTable` (see also the family of `bro_table_xxx()` functions and their explanation below) |
| Set | `BRO_TYPE_SET` | `BroSet` (see also the family of `bro_set_xxx()` functions and their explanation below) |

Knowing these, we can now compose a `request_connections` event:

The third argument to `bro_event_add_val()` lets you specify a specialization of the types listed in *Table-1*. This is generally not necessary except for one situation: when using `BRO_TYPE_ENUM`. You currently cannot define a Bro-level enum type in Broccoli, and thus when sending an enum value, you have to specify the type of the enum along with the value. For example, in order to add an instance of enum `transport_proto` defined in Bro's `init-bare.bro`, you would use:

to get the equivalent of "udp" on the remote side. The same system is used to point out type names when calling `bro_event_set_val()`, `bro_record_add_val()`, `bro_record_set_nth_val()`, and `bro_record_set_named_val()`.

All that's left to do now is to send off the event. For this, use `bro_event_send()` and pass it the connection handle and the event. The function returns `TRUE` when the event could be sent right away or if it was queued for later delivery. `FALSE` is returned on error. If the event gets queued, this does not indicate an error – likely the connection was just not ready to send the event at this point. Whenever you call `bro_event_send()`, Broccoli attempts to send as much of an existing event queue as possible. Again, the event is copied internally to make it easier for you to send the same event repeatedly. You clean up what you allocate:

Two other functions may be useful to you: `bro_event_queue_length()` tells you how many events are currently queued, and `bro_event_queue_flush()` attempts to flush the current event queue and returns the number of events that do remain in the queue after the flush.

**Note:** you do not normally need to call this function, queue flushing is attempted every time you send an event.

### Receiving Events

Receiving events is a little more work because you need to

1. tell Broccoli what to do when requested events arrive,
2. let the remote Bro agent know that you would like to receive those events,
3. find a spot in the code path suitable for extracting and processing arriving events.

Each of these steps is explained in the following sections.

### Implementing event callbacks

When Broccoli receives an event, it tries to dispatch the event to callbacks registered for that event type. The place where callbacks get registered is called the callback registry. Any callbacks registered for the arriving event's name are invoked with the parameters shipped with the event. There are two styles of argument passing to the event callbacks. Which one is better suited depends on your application.

### Expanded Argument Passing

Each event argument is passed via a pointer to the callback. This makes best sense when you know the type of the event and of its arguments, because it provides you immediate access to arguments as when using a normal C function.

In order to register a callback with expanded argument passing, use `bro_event_registry_add()` and pass it the connection handle, the name of the event for which you register the callback, the callback itself that matches the signature of the `BroEventFunc` type, and any user data (or `NULL`) you want to see passed to the callback on each invocation. The callback's type is defined rather generically as follows:

It requires a connection handle as its first argument and a pointer to user-provided callback data as the second argument. Broccoli will pass the connection handle of the connection on which the event arrived through to the callback. `BroEventFunc`'s are variadic, because each callback you provide is directly invoked with pointers to the parameters of the event, in a format directly usable in C. All you need to know is what type to point to in order to receive the parameters in the right layout. Refer to *Table-1* again for a summary of those types. Record types are more involved and are addressed in more detail in *Handling Records*.

**Note:** Note that *all* parameters are passed to the callback as pointers, even elementary types such as `int` that would normally be passed directly. Also note that Broccoli manages the lifecycle of event parameters and therefore you do *not* have to clean them up inside the event handler.

Continuing our example, we will want to process the connection reports that contain the responses to our `report_conns` event. Let's assume those look as follows:

The reply events contain the request ID so we can associate requests with replies, and a connection record (defined in `init-bare.bro` in Bro). (It'd be nicer to report all replies in a single event but we'll ignore that for now.) For this event, our callback would look like this:

Once more, you clean up what you allocate, and since you never allocated the space these arguments point to, you also don't clean them up. Finally, we register the callback using `bro_event_registry_add()`:

In this case we have no additional data to be passed into the callback, so we use `NULL` for the last argument. If you have multiple events you are interested in, register each one in this fashion.

### Compact Argument Passing

This is designed for situations when you have to determine how to handle different types of events at runtime, for example when writing language bindings or when implementing generic event handlers for multiple event types. The callback is passed a connection handle and the user data as above but is only passed one additional pointer, a BroEvMeta structure. This structure contains all metadata about the event, including its name, timestamp (in UTC) of creation, number of arguments, the arguments' types (via type tags as listed in *Table-1*), and the arguments themselves.

In order to register a callback with compact argument passing, use `bro_event_registry_add_compact()` and pass it similar arguments as you'd use with `bro_event_registry_add()`. The callback's type is defined as follows:

**Note:** As before, Broccoli manages the lifecycle of event parameters. You do not have to clean up the BroEvMeta structure or any of its contents.

Below is sample code for extracting the arguments from the BroEvMeta structure, using our running example. This is still written with the assumption that we know the types of the arguments, but note that this is not a requirement for this style of callback:

Finally, register the callback using `bro_event_registry_add_compact()`:

### Requesting event delivery

At this point, Broccoli knows what to do with the requested events upon arrival. What's left to do is to let the remote Bro know that you would like to receive the events for which you registered. If you haven't yet called `bro_conn_connect()`, then there is nothing to do, since that function will request the registered events anyway. Once connected, you can still request events. To do so, call `bro_event_registry_request()`:

This mechanism also implies that no unrequested events will be delivered to us (and if that happened for whatever reason, the event would simply be dropped on the floor).

**Note:** At the moment you cannot unrequest events, nor can you request events based on predicates on the values of the events' arguments.

### Reading events from the connection handle

At this point the remote Bro will start sending you the requested events once they are triggered. What is left to do is to read the arriving events from the connection and trigger dispatching them to the registered callbacks.

If you are writing a new Bro-enabled application, this is easy, and you can choose among two approaches: polling explicitly via Broccoli's API, or using `select()` on the file handle associated with a BroConn. The former case is particularly straightforward; all you need to do is call `bro_conn_process_input()`, which will go off and check if any events have arrived and if so, dispatch them accordingly. This function does not block – if no events have arrived, then the call will return immediately. For more fine-grained control over your I/O handling, you will probably want to use `bro_conn_get_fd()` to obtain the file descriptor of your connection and then incorporate that in your standard `FD_SET`/`select()` code. Once you have determined that data in fact are ready to be read from the obtained file descriptor, you can then try another `bro_conn_process_input()` this time knowing that it'll find something to dispatch.

As a side note, if you don't process arriving events frequently enough, then TCP's flow control will start to slow down the sender until eventually events will queue up and be dropped at the sending end.

### Handling Records

Broccoli supports record structures, i.e., types that pack a set of values together, placing each value into its own field. In Broccoli, the way you handle records is somewhat similar to events: after creating an empty record (of opaque type `BroRecord`), you can iteratively add fields and values to it. The main difference is that you must specify a field name with the value; each value in a record can be identified both by position (a numerical index starting from zero), and by field name. You can retrieve vals in a record by field index or field name. You can also reassign values. There is no explicit, IDL-style definition of record types. You define the type of a record implicitly by the sequence of field names and the sequence of the types of the values you put into the record.

Note that all fields in a record must be assigned before it can be shipped.

The API for record composition consists of `bro_record_new()`, `bro_record_free()`, `bro_record_add_val()`, `bro_record_set_nth_val()`, and `bro_record_set_named_val()`.

On records that use field names, the names of individual fields can be extracted using `bro_record_get_nth_name()`. Extracting values from a record is done using `bro_record_get_nth_val()` and `bro_record_get_named_val()`. The former allows numerical indexing of the fields in the record, the latter provides name-based lookups. Both need to be passed the record you want to extract a value from, the index or name of the field, and either a pointer to an int holding a BRO_TYPE_xxx value (see again *Table-1* for a summary of those types) or `NULL`. The pointer, if not `NULL`, serves two purposes: type checking and type retrieval. Type checking is performed if the value of the int upon calling the functions is not BRO_TYPE_UNKNOWN. The type tag of the requested record field then has to match the type tag stored in the int, otherwise `NULL` is returned. If the int stores BRO_TYPE_UNKNOWN upon calling, no type-checking is performed. In *both* cases, the *actual* type of the requested record field is returned in the int pointed to upon return from the function. Since you have no guarantees of the type of the value upon return if you pass `NULL` as the int pointer, this is a bad idea and either BRO_TYPE_UNKNOWN or another type value should always be used.

For example, you could extract the value of the record field "label", which we assume should be a string, in the following ways:

Record fields can be records, for example in the case of Bro's standard connection record type. In this case, in order to get to a nested record, you use `BRO_TYPE_RECORD`:

### Handling Tables

Broccoli supports Bro-style tables, i.e., associative containers that map instances of a key type to an instance of a value type. A given key can only ever point to a single value. The key type can be *composite*, i.e., it may consist of an ordered sequence of different types, or it can be *direct*, i.e., consisting of a single type (such as an integer, a string, or a record).

The API for table manipulation consists of `bro_table_new()` `bro_table_free()`, `bro_table_insert()`, `bro_table_find()`, `bro_table_get_size()`, `bro_table_get_types()`, and `bro_table_foreach()`.

Tables are handled similarly to records in that typing is determined dynamically by the initial key/value pair inserted. The resulting types can be obtained via `bro_table_get_types()`. Should the types not have been determined yet, `BRO_TYPE_UNKNOWN` will result. Also, as with records, values inserted into the table are copied internally, and the ones passed to the insertion functions remain unaffected.

In contrast to records, table entries can be iterated. By passing a function of signature `BroTableCallback()` and a pointer to data of your choosing, `bro_table_foreach()` will invoke the given function for each key/value pair stored in the table. Return `TRUE` to keep the iteration going, or `FALSE` to stop it.

---

**Note:** The main thing to know about Broccoli's tables is how to use composite key types. To avoid additional API calls, you may treat composite key types exactly as records, though you do not need to use field names when assigning elements to individual fields. So in order to insert a key/value pair, you create a record with the needed items assigned to its slots, and use this record as the key object. In order to differentiate composite index types from direct ones consisting of a single record, use `BRO_TYPE_LIST` as the type of the record, as opposed to `BRO_TYPE_RECORD`. Broccoli will then know to interpret the record as an ordered sequence of items making up a composite element, not a regular record.

---

`brotable.c` in the `test/` subdirectory of the Broccoli tree contains an extensive example of using tables with composite as well as direct indexing types.

### Handling Sets

Sets are essentially tables with void value types. The API for set manipulation consists of `bro_set_new()`, `bro_set_free()`, `bro_set_insert()`, `bro_set_find()`, `bro_set_get_size()`, `bro_set_get_type()`, and `bro_set_foreach()`.

### Associating data with connections

You will often find that you would like to connect data with a `BroConn`. Broccoli provides an API that lets you associate data items with a connection handle through a string-based key-value registry. The functions of interest are `bro_conn_data_set()`, `bro_conn_data_get()`, and `bro_conn_data_del()`. You need to provide a string identifier for a data item and can then use that string to register, look up, and remove the associated data item. Note that there is currently no mechanism to trigger a destructor function for registered data items when the Bro connection is terminated. You therefore need to make sure that all data items that you do not have pointers to via some other means are properly released before calling `bro_disconnect()`.

### Configuration Files

Imagine you have instrumented the mother of all server applications. Building it takes forever, and every now and then you need to change some of the parameters that your Broccoli code uses, such as the host names of the Bro agents to talk to. To allow you to do this quickly, Broccoli comes with support for configuration files. All you need to do is change the settings in the file and restart the application (we're considering adding support for volatile configuration items that are read from the file every time they are requested).

A configuration is read from a single configuration file. This file can be read from different locations. Broccoli searches in this order for the config file:

- The location specified by the `BROCCOLI_CONFIG_FILE` environment variable.
- A per-user configuration file stored in `~/.broccoli.conf`.
- The system-wide configuration file. You can obtain the location of this config file by running `broccoli-config --config`.

---

**Note:** `BROCCOLI_CONFIG_FILE` or `~/.broccoli.conf` will only be used if it is a regular file, not executable, and neither group nor others have any permissions on the file. That is, the file's permissions must look like

`-rw-------` *or* `-r--------`.

In the configuration file, a `#` anywhere starts a comment that runs to the end of the line. Configuration items are specified as key-value pairs:

```
# This is the Broccoli system-wide configuration file.
#
# Entries are of the form <identifier> <value>, where the
# identifier is a sequence of letters, and value can be a string
# (including whitespace), and floating point or integer numbers.
# Comments start with a "#" and go to the end of the line. For
# boolean values, you may also use "yes", "on", "true", "no",
# "off", or "false".  Strings may contain whitespace, but need
# to be surrounded by double quotes '"'.
#
# Examples:
#
Foo/PeerName          mybro.securesite.com
Foo/PortNum           123
Bar/SomeFloat         1.23443543
Bar/SomeLongStr       "Hello World"
```

You can also have multiple sections in your configuration. Your application can select a section as the current one, and queries for configuration settings will then only be answered with values specified in that section. A section is started by putting its name (no whitespace please) between square brackets. Configuration items positioned before the first section title are in the default domain and will be used by default:

```
# This section contains all settings for myapp.
[ myapp ]
```

You can name identifiers any way you like, but to keep things organized it is recommended to keep a namespace hierarchy similar to the file system. In the code, you can query configuration items using `bro_conf_get_str()`, `bro_conf_get_int()`, and `bro_conf_get_dbl()`. You can switch between sections using `bro_conf_set_domain()`.

### Using Dynamic Buffers

Broccoli provides an API for dynamically allocatable, growable, shrinkable, and consumable buffers with `BroBuf`. You may or may not find this useful – Broccoli mainly provides this feature in `broccoli.h` because these buffers are used internally anyway and because they are a typical case of something that people implement themselves over and over again, for example to collect a set of data before sending it through a file descriptor, etc.

The buffers work as follows. The structure implementing a buffer is called `BroBuf`, and is initialized to a default size when created via `bro_buf_new()` and released using `bro_buf_free()`. Each `BroBuf` has a content pointer that points to an arbitrary location between the start of the buffer and the first byte after the last byte currently used in the buffer (see `buf_off` in the illustration below). The content pointer can seek to arbitrary locations, and data can be copied from and into the buffer, adjusting the content pointer accordingly. You can repeatedly append data to the end of the buffer's used contents using `bro_buf_append()`.

```
<---------------- allocated buffer space ------------>
<======== used buffer space ========>                ^
^              ^                    ^                |
|              |                    |                |
buf            buf_ptr              buf_off          buf_len
```

Have a look at the following functions for the details: `bro_buf_new()`, `bro_buf_free()`, `bro_buf_append()`, `bro_buf_consume()`, `bro_buf_reset()`, `bro_buf_get()`, `bro_buf_get_end()`, `bro_buf_get_size()`, `bro_buf_get_used_size()`, `bro_buf_ptr_get()`, `bro_buf_ptr_tell()`, `bro_buf_ptr_seek()`, `bro_buf_ptr_check()`, and `bro_buf_ptr_read()`.

## Configuring Encrypted Communication

Encrypted communication between Bro peers takes place over an SSL connection in which both endpoints of the connection are authenticated. This requires at least some PKI in the form of a certificate authority (CA) which you use to issue and sign certificates for your Bro peers. To facilitate the SSL setup, each peer requires three documents: a certificate signed by the CA and containing the public key, the corresponding private key, and a copy of the CA's certificate.

The OpenSSL command line tool `openssl` can be used to create all files necessary, but its unstructured arguments and poor documentation make it a pain to use and waste lots of people a lot of time [1]. For an alternative tool to create SSL certificates for secure Bro/Broccoli communication, see the `create-cert` tool available at ftp://ee.lbl.gov/create-cert.tar.gz.

In order to enable encrypted communication for your Broccoli application, you need to put the CA certificate and the peer certificate in the `/broccoli/ca_cert` and `/broccoli/host_cert` keys, respectively, in the configuration file. Optionally, you can store the private key in a separate file specified by `/broccoli/host_key`. To quickly enable/disable a certificate configuration, the `/broccoli/use_ssl` key can be used.

**Note:** *This is where you configure whether to use encrypted or unencrypted connections.*

If the `/broccoli/use_ssl` key is present and set to one of "yes", "true", "on", or 1, then SSL will be used and an incorrect or missing certificate configuration will cause connection attempts to fail. If the key's value is one of "no", "false", "off", or 0, then in no case will SSL be used and connections will always be cleartext.

If the `/broccoli/use_ssl` key is *not* present, then SSL will be used if a certificate configuration is found, and invalid certificates will cause the connection to fail. If no certificates are configured, cleartext connections will be used.

In no case does an SSL-enabled setup ever fall back to a cleartext one.

```
/broccoli/use_ssl          yes
/broccoli/ca_cert          <path>/ca_cert.pem
/broccoli/host_cert        <path>/bro_cert.pem
/broccoli/host_key         <path>/bro_cert.key
```

In a Bro policy, you need to load the `frameworks/communication/listen.bro` script and redef `Communication::listen_ssl=T`, `ssl_ca_certificate`, and `ssl_private_key`, defined in `init-bare.bro`:

By default, you will be prompted for the passphrase for the private key matching the public key in your agent's certificate. Depending on your application's user interface and deployment, this may be inappropriate. You can store the passphrase in the config file as well, using the following identifier:

```
/broccoli/host_pass        foobar
```

[1] In other documents and books on OpenSSL you will find this expressed more politely, using terms such as "daunting to the uninitiated", "challenging", "complex", "intimidating".

**Warning:** *Make sure that access to your configuration is restricted.*

If you provide the passphrase this way, it is obviously essential to have restrictive permissions on the configuration file. Broccoli partially enforces this. Please refer to the section on *Configuration Files* for details.

### Configuring event reception in Bro scripts

Before a remote Bro will accept your connection and your events, it needs to have its policy configured accordingly:

1. Load `frameworks/communication/listen`, and redef the boolean variable `Communication::listen_ssl` depending on whether you want to have encrypted or cleartext communication. Obviously, encrypting the event exchange is recommended and cleartext should only be used for early experimental setups. See below for details on how to set up encrypted communication via SSL.
2. You need to find a port to use for the Bros and Broccoli applications that will listen for connections. Every such agent can use a different port, though default ports are provided in the Bro policies. To change the port the Bro agent will be listening on from its default, redefine the `Communication::listen_port`. Have a look at these policies as well as `base/frameworks/communication/main.bro` for the default values. Here is the policy for the unencrypted case:

   Including the settings for the cryptographic files introduced in the previous section, here is the encrypted one:
3. The policy controlling which peers a Bro agent will communicate with and how this communication will happen are defined in the `Communication::nodes` table defined in `base/frameworks/communication/main.bro`. This table contains entries of type `Node`, whose members mostly provide default values so you do not need to define everything. You need to come up with a tag for the connection under which it can be found in the table (a creative one would be "broccoli"), the IP address of the peer, the pattern of names of the events the Bro will accept from you, whether you want Bro to connect to your machine on startup or not, if so, a port to connect to (default is `Communication::default_port` also defined in `base/frameworks/communication/main.bro`), a retry timeout, whether to use SSL, and the class of a connection as set on the Broccoli side via `bro_conn_set_class()`.

   An example could look as follows:

   This example is taken from `broping.bro`, the policy the remote Bro must run when you want to use the `broping` tool explained in the section on *test programs* below. It will allow an agent on the local host to connect and send "ping" events. Our Bro will not attempt to connect, and incoming connections will be expected in cleartext.

### Configuring Debugging Output

If your Broccoli installation was configured with `--enable-debug`, Broccoli will report two kinds of debugging information:

1. function call traces and
2. individual debugging messages.

Both are enabled by default, but can be adjusted in two ways.

- In the configuration file: in the appropriate section of the configuration file, you can set the keys `/broccoli/debug_messages` and `/broccoli/debug_calltrace` to `on`/`off` to enable/disable the corresponding output.
- In code: you can set the variables `bro_debug_calltrace` and `bro_debug_messages` to 1/0 at any time to enable/disable the corresponding output.

By default, debugging output is inactive (even with debug support compiled in). You need to enable it explicitly either in your code by assigning 1 to `bro_debug_calltrace` and `bro_debug_messages` or by enabling it in the configuration file.

### Test programs

The Broccoli distribution comes with a few small test programs, located in the `test/` directory of the tree. The most notable one is `broping` [2], a mini-version of ping. It sends "ping" events to a remote Bro agent, expecting "pong" events in return. It operates in two flavours: one uses atomic types for sending information across, and the other one uses records. The Bro agent you want to ping needs to run either the `broping.bro` or `broping-record.bro` policies. You can find these in the `test/` directory of the source tree, and in `<prefix>/share/broccoli` in the installed version. `broping.bro` is shown below. By default, pinging a Bro on the same machine is configured. If you want your Bro to be pinged from another machine, you need to update the `Communication::nodes` variable accordingly:

`broping` sends ping events to Bro. Bro accepts those because they are configured accordingly in the nodes table. As shown in the policy, ping events trigger pong events, and `broccoli` requests delivery of all pong events back to it. When running `broping`, you'll see something like this:

### Notes

### Broccoli API Reference

The API documentation describes Broccoli's public C interface.

## 3.3.4 Python Bindings for Broccoli

This Python module provides bindings for Broccoli, Bro's client communication library. In general, the bindings provide the same functionality as Broccoli's C API.

**Contents**

[2] Pronunciation is said to be somewhere on the continuum between "brooping" and "burping".

### Download

You can find the latest Broccoli-Python release for download at http://www.bro.org/download.

Broccoli-Python's git repository is located at git://git.bro.org/broccoli-python.git. You can browse the repository here.

This document describes Broccoli-Python 0.59. See the `CHANGES` file for version history.

### Installation

Installation of the Python module is pretty straight-forward:

```
./configure
make
python setup.py install
```

Try the following to test the installation. If you do not see any error message, everything should be fine:

```
python -c "import broccoli"
```

### Usage

The following examples demonstrate how to send and receive Bro events in Python.

The main challenge when using Broccoli from Python is dealing with the data types of Bro event parameters as there is no one-to-one mapping between Bro's types and Python's types. The Python modules automatically maps between those types which both systems provide (such as strings) and provides a set of wrapper classes for Bro types which do not have a direct Python equivalent (such as IP addresses).

#### Connecting to Bro

The following code sets up a connection from Python to a remote Bro instance (or another Broccoli) and provides a connection handle for further communication:

```
from broccoli import *
bc = Connection("127.0.0.1:47758")
```

An `IOError` will be raised if the connection cannot be established.

#### Sending Events

Once you have a connection handle `bc` set up as shown above, you can start sending events:

```
bc.send("foo", 5, "attack!")
```

This sends an event called `foo` with two parameters, `5` and `attack!`. Broccoli operates asynchronously, i.e., events scheduled with `send()` are not always sent out immediately but might be queued for later transmission. To ensure that all events get out (and incoming events are processed, see below), you need to call `bc.processInput()` regularly.

## Data Types

In the example above, the types of the event parameters are automatically derived from the corresponding Python types: the first parameter (`5`) has the Bro type `int` and the second one (`attack!`) has Bro type `string`.

For types which do not have a Python equivalent, the `broccoli` module provides wrapper classes which have the same names as the corresponding Bro types. For example, to send an event called `bar` with one `addr` argument and one `count` argument, you can write:

```
bc.send("bar", addr("192.168.1.1"), count(42))
```

The following table summarizes the available atomic types and their usage.

| Bro Type | Python Type | Example |
|---|---|---|
| addr | | `addr("192.168.1.1")` |
| bool | bool | `True` |
| count | | `count(42)` |
| double | float | `3.14` |
| enum | | Type currently not supported |
| int | int | `5` |
| interval | | `interval(60)` |
| net | | Type currently not supported |
| port | | `port("80/tcp")` |
| string | string | `"attack!"` |
| subnet | | `subnet("192.168.1.0/24")` |
| time | | `time(1111111111.0)` |

The `broccoli` module also supports sending Bro records as event parameters. To send a record, you first define a record type. For example, a Bro record type:

```
type my_record: record {
    a: int;
    b: addr;
    c: subnet;
};
```

turns into Python as:

```
my_record = record_type("a", "b", "c")
```

As the example shows, Python only needs to know the attribute names but not their types. The types are derived automatically in the same way as discussed above for atomic event parameters.

Now you can instantiate a record instance of the newly defined type and send it out:

```
rec = record(my_record)
rec.a = 5
rec.b = addr("192.168.1.1")
rec.c = subnet("192.168.1.0/24")
bc.send("my_event", rec)
```

**Note:** The Python module does not support nested records at this time.

### Receiving Events

To receive events, you define a callback function having the same name as the event and mark it with the `event` decorator:

```
@event
def foo(arg1, arg2):
    print arg1, arg2
```

Once you start calling `bc.processInput()` regularly (see above), each received `foo` event will trigger the call-back function.

By default, the event's arguments are always passed in with built-in Python types. For Bro types which do not have a direct Python equivalent (see table above), a substitute built-in type is used which corresponds to the type the wrapper class' constructor expects (see the examples in the table). For example, Bro type `addr` is passed in as a string and Bro type `time` is passed in as a float.

Alternatively, you can define a _typed_ prototype for the event. If you do so, arguments will first be type-checked and then passed to the call-back with the specified type (which means instances of the wrapper classes for non-Python types). Example:

```
@event(count, addr)
def bar(arg1, arg2):
    print arg1, arg2
```

Here, `arg1` will be an instance of the `count` wrapper class and `arg2` will be an instance of the `addr` wrapper class.

Protoyping works similarly with built-in Python types:

```
@event(int, string):
def foo(arg1, arg2):
    print arg1, arg2
```

In general, the prototype specifies the types in which the callback wants to receive the arguments. This actually provides support for simple type casts as some types support conversion to into something different. If for instance the event source sends an event with a single port argument, `@event(port)` will pass the port as an instance of the `port` wrapper class; `@event(string)` will pass it as a string (e.g., `"80/tcp"`); and `@event(int)` will pass it as an integer without protocol information (e.g., just `80`). If an argument cannot be converted into the specified type, a `TypeError` will be raised.

To receive an event with a record parameter, the record type first needs to be defined, as described above. Then the type can be used with the `@event` decorator in the same way as atomic types:

```
my_record = record_type("a", "b", "c")
@event(my_record)
def my_event(rec):
    print rec.a, rec.b, rec.c
```

### Helper Functions

The `broccoli` module provides one helper function: `current_time()` returns the current time as a float which, if necessary, can be wrapped into a `time` parameter (i.e., `time(current_time())`

### Examples

There are some example scripts in the `tests/` subdirectory of the `broccoli-python` repository here:

- `broping.py` is a (simplified) Python version of Broccoli's test program `broping`. Start Bro with `broping.bro`.
- `broping-record.py` is a Python version of Broccoli's `broping` for records. Start Bro with `broping-record.bro`.
- `test.py` is a very ugly but comprehensive regression test and part of the communication test-suite. Start Bro with `test.bro`.

## 3.3.5 Ruby Bindings for Broccoli

This is the broccoli-ruby extension for Ruby which provides access to the Broccoli API. Broccoli is a library for communicating with the Bro Intrusion Detection System.

### Download

You can find the latest Broccoli-Ruby release for download at http://www.bro.org/download.

Broccoli-Ruby's git repository is located at git://git.bro.org/broccoli-ruby.git. You can browse the repository here.

This document describes Broccoli-Ruby 1.58. See the `CHANGES` file for version history.

### Installation

To install the extension:

1. Make sure that the `broccoli-config` binary is in your path. (`export PATH=/usr/local/bro/bin:$PATH`)
2. Run `sudo ruby setup.rb`.

To install the extension as a gem (suggested):

1. Install rubygems.
2. Make sure that the `broccoli-config` binary is in your path. (`export PATH=/usr/local/bro/bin:$PATH`)
3. Run, `sudo gem install rbroccoli`.

### Usage

There aren't really any useful docs yet. Your best bet currently is to read through the examples.

One thing I should mention however is that I haven't done any optimization yet. You may find that if you write code that is going to be sending or receiving extremely large numbers of events, that it won't run fast enough and will begin to fall behind the Bro server. The dns_requests.rb example is a good performance test if your Bro server is sitting on a network with many dns lookups.

### Contact

If you have a question/comment/patch, see the Bro contact page.

### 3.3.6 Broker: Bro's Messaging Library

The Broker library implements Bro's high-level communication patterns:

- remote logging
- remote printing
- remote events
- distributed data stores

Logging, printing, and events all follow a pub/sub communication model between Broker endpoints that are directly peered with each other. An endpoint also has the option of subscribing to its own messages. Subscriptions are matched prefix-wise and endpoints have the capability of fine-tuning the subscription topic strings they wish to advertise to peers as well as the messages they wish to send to them.

The distributed data store functionality allows a master data store associated with one Broker endpoint to be cloned at peer endpoints which may then perform lightweight, local queries against the clone, which automatically stays synchronized with the master store. Clones cannot modify their content directly, instead they send modifications to the centralized master store which applies them and then broadcasts them to all clones.

Applications which integrate the Broker library may communicate with each other using the above-mentioned patterns which are common to Bro.

#### Dependencies

Compiling Broker requires the following libraries/tools to already be installed:

A C++11 capable compiler (GCC 4.8+ or Clang 3.3+)

**CAF (C++ Actor Framework) version 0.14+** https://github.com/actor-framework/actor-framework

**CMake 2.8+** CMake is a cross-platform, open-source build system, typically not installed by default. See http://www.cmake.org for more information regarding CMake and the installation steps below for how to use it to build this distribution. CMake generates native Makefiles that depend on GNU Make by default.

And optionally:

**SWIG 3.0.3+, http://www.swig.org/** SWIG is used to optionally build Python bindings for Broker.

#### Compiling/Installing

To compile and install in to `/usr/local`:

```
./configure
make
make install
```

See `./configure --help` for more advanced configuration options.

#### Documentation

Please see the Broker User Manual for more documentation and examples of how to use the library.

## 3.3.7 Broker: Bro's Messaging Library

### Introduction

The Broker library is the successor to the Broccoli library. One of the main differences is that while Broccoli is a reimplementation of a communication protocol built in to Bro (i.e. Bro does *not* link against libbroccoli), Broker implements a standalone protocol and data model that Bro uses directly (i.e. Bro *does* link against libbroker). All applications that integrate the Broker library share the same capabilities for exchanging information with each other, so let's go through some examples of what you can do.

### Creating Endpoints and Peering Them Together

Connecting endpoints together is simple, just give them descriptive names and request they peer with each other.

Endpoints don't have to live in the same process, instead they can talk over TCP.

The initiator of a remote peering will automatically try reconnecting if it cannot initially reach the remote side or if it ever becomes disconnected. The frequency at which it retries can be passed as the third parameter to the `peer()` call.

### Sending Messages

Now that endpoints are connected, they can talk to each other.

Any endpoints connected to "first_endpoint" will receive the message if they have advertised interest (covered below) in any prefix of "my_topic".

Broker messages are a sequence of data items and a data item is a variant type whose storage may hold one of several data types. In the above code, the message contains the string, "hi", and an unsigned 64-bit integer, 42. For all the possible data types allowed in Broker data/messages see the data API reference.

The third paramenter to the `send()` method are flags which tune the behavior of how the message is to be sent. By default, messages are allowed to be sent to the sending endpoint itself or to peer endpoints if they advertise interest in any prefix of the message's topic string. There's another flag, `UNSOLICITED` that can be used to send messages to peers even if they have not advertised interest in any prefix of the message's topic string.

### Receiving Messages

To receive messages from other endpoints, one needs to create a message queue which advertises interest in a topic prefix and then extract messages from the queue.

Here, the message queue is attached to the "second_endpoint" endpoint and subscribes to all messages that get sent with a topic string prefixed by "my". Message subscriptions are prefix-based, e.g. the empty string will match messages sent with any topic and "a" will receive messages sent with either topic "alice" or "amy" but not "bob".

Broker's queue structures expose a file descriptor that signals read-readiness when the queue has content that can be retrieved. It can be used to integrate into event loops as usual.

Alternatively, there is a `need_pop()` method which blocks until at least one item is available in the queue. This is mostly for convenience, use with caution.

Either pop method retrieves all contents that have been received by the queue up to that point in time.

#### Monitor Connection Status

By default, Broker endpoints have queues attached to them which can be monitored to check the status of connections with peer endpoints.

Applications should periodically check connection status queues for updates.

#### Tuning Access Control

By default, Broker endpoints do not restrict the message topics that it sends to peers and do not restrict what message queue topics and data store identifiers get advertised to peers. This is the default `AUTO_PUBLISH | AUTO_ADVERTISE` flags argument to the `endpoint` constructor.

If not using the `AUTO_PUBLISH` flag, one can use an endpoint's `publish()` and `unpublish()` methods to manipulate the set of message topics (must match exactly) that are allowed to be sent to peer endpoints. These settings take precedence over the per-message `PEERS` flag supplied to `send()`.

If not using the `AUTO_ADVERTISE` flag, one can use an endpoint's `advertise()` and `unadvertise()` to manipulate the set of topic prefixes that are allowed to be advertised to peers. If an endpoint does not advertise a topic prefix, then the only way peers can send messages to it is via the `UNSOLICITED` flag to `send()` and choosing a topic with a matching prefix (i.e. full topic may be longer than receivers prefix, just the prefix needs to match).

#### Distributed Data Stores

There are three flavors of key-value data store interfaces: master, clone, and frontend.

A frontend is the common interface to query and modify data stores. That is, a clone is a specific type of frontend and a master is also a specific type of frontend, but a standalone frontend can also exist to e.g. query and modify the contents of a remote master store without actually "owning" any of the contents itself.

A master data store attached with one Broker endpoint can be cloned at peer endpoints which may then perform lightweight, local queries against the clone, which automatically stays synchronized with the master store. Clones cannot modify their content directly, instead they send modifications to the centralized master store which applies them and then broadcasts them to all clones.

Master and clone stores get to choose what type of storage backend to use. E.g. In-memory versus SQLite for persistence. Note that if clones are used, data store sizes should still be able to fit within memory regardless of the storage backend as a single snapshot of the master store is sent in a single chunk to initialize the clone.

Data stores also support expiration on a per-key basis either using an absolute point in time or a relative amount of time since the entry's last modification time.

See the unit tests in `tests/test_store*` and the `store/` API reference for more examples and details.

### 3.3.8 BroControl

This document summarizes installation and use of *BroControl*, Bro's interactive shell for operating Bro installations. *BroControl* has two modes of operation: a *stand-alone* mode for managing a traditional, single-system Bro setup; and a *cluster* mode for maintaining a multi-system setup of coordinated Bro instances load-balancing the work across a set of independent machines. Once installed, the operation is pretty similar for both types; just keep in mind that if this document refers to "nodes" and you're in a stand-alone setup, there is only a single one and no worker/proxies.

**Contents**

### Download

You can find the latest BroControl release for download at http://www.bro.org/download.

BroControl's git repository is located at git://git.bro.org/broctl. You can browse the repository here.

This document describes BroControl 1.4-150. See the `CHANGES` file for version history.

### Prerequisites

Running BroControl requires the following prerequisites:

- A Unix system. FreeBSD, Linux, and Mac OS X are supported and should work out of the box. Other Unix systems will quite likely require some tweaking. Note that in a cluster setup, all systems must be running exactly the *same* operating system.
- A version of *Python* >= 2.6 (on FreeBSD, the package "py27-sqlite3" must also be installed).
- A *bash* (note in particular, that on FreeBSD, *bash* is not installed by default).
- If *sendmail* is installed (for a cluster setup, it is needed on the manager only), then BroControl can send mail. Otherwise, BroControl will not attempt to send mail.

- If *gdb* is installed and if Bro crashes, then BroControl can include a backtrace in its crash report. That can be helpful for debugging problems with Bro.

For a cluster setup that spans more than one machine, there are additional requirements:

- Every host in the cluster must have *rsync* installed.
- The manager host must have *ssh* installed, and every other host in the cluster must have *sshd* installed and running.
- Decide which user account will be running BroControl, and then make sure this user account is set up on all hosts in your cluster. Also make sure this user can `ssh` from the manager host to each of the other hosts in your cluster, and this must work without being prompted for a password/passphrase (one way to accomplish this is to use ssh public key authentication).

If you're using a load-balancing method such as PF_RING, then there is additional software to install (for details, see the *Cluster Configuration* documentation).

### Installation

Follow the directions to install Bro and BroControl according to the instructions in the *Installing Bro* documentation. Note that if you are planning to run Bro in a cluster configuration, then you need to install Bro and BroControl only on the manager host (the BroControl *install* or *deploy* commands will install Bro and all required scripts to the other hosts in your cluster).

### Using BroControl as an unprivileged user

If you decide to run BroControl as an unprivileged user, there are a few issues that you may encounter.

If you installed Bro and BroControl as the "root" user, then you will need to adjust the ownership or permissions of the "logs" and "spool" directories (and everything in those directories) so that the user running BroControl has write permission.

If you're using a cluster setup that spans multiple machines, and if your BroControl `install` or `deploy` commands fail with a permission denied error, then it's most likely due to the user running BroControl not having permission to create the install prefix directory (by default, this is `/usr/local/bro`) on each remote machine. A simple workaround is to login to each machine in your cluster and manually create the install prefix directory and then set ownership or permissions of this directory so that the user who will run BroControl has write access to it.

Finally, on the worker nodes (or the standalone node), Bro must have access to the target network interface in promiscuous mode. If Bro doesn't have the necessary permissions, then it will fail almost immediately upon startup. A workaround for this is provided in the Bro FAQ.

### Configuration

Before actually running BroControl, you first need to edit the `broctl.cfg`, `node.cfg`, and `networks.cfg` files.

In the `broctl.cfg` file, you should review the BroControl options and make sure they are set correctly for your environment. Most options have default values that are reasonable for most users (the *MailTo* option is probably the one that you will most likely want to change), but for a description of every BroControl option, see the *Option Reference* section below.

Next, edit the `node.cfg` file and specify the nodes that you will be running. For a description of every option available for nodes, see the *Node* section below. If you will be using a standalone configuration then most likely the only option you need to change is the `interface`. If you will be using a cluster configuration, there is a *Cluster Configuration* guide that provides examples and additional information.

Finally, edit the `networks.cfg` file and list each network (see the examples in the file for the format to use) that is considered local to the monitored environment.

### Basic Usage

BroControl is an interactive interface for managing a Bro installation which allows you to, e.g., start/stop the monitoring or update its configuration.

BroControl is started with the `broctl` script and then expects commands on its command-line (alternatively, `broctl` can also be started with a single command directly on the shell's command line, such as `broctl help`):

```
> broctl
Welcome to BroControl x.y

Type "help" for help.

[BroControl] >
```

As the message says, type *help* to see a list of all commands. We will now briefly summarize the most important commands. A full reference follows *Command Reference*.

The *config* command gives a list of all BroControl options with their current values. This can be useful when troubleshooting a problem to check if an option has the expected value.

If this is the first time you are running BroControl, then the first command you must run is the BroControl *deploy* command. The "deploy" command will make sure all of the files needed by BroControl and Bro are brought up-to-date based on the configuration specified in the `broctl.cfg`, `node.cfg`, and `networks.cfg` files. It will also check if there are any syntax errors in your Bro policy scripts. For a cluster setup it will copy all of the required scripts and executables to all the other hosts in your cluster. Then it will successively start the logger, manager, proxies, and workers (for a standalone configuration, only one Bro instance will be started).

The *status* command can be used to check that all nodes are "running". If any nodes have a status of "crashed", then use the *diag* command to see diagnostic information (you can specify the name of a crashed node as an argument to the diag command to show diagnostics for only that one node).

To stop the monitoring, issue the *stop* command. After all nodes have stopped, the *status* command should show all nodes as "stopped". *exit* leaves the shell (you can also exit BroControl while Bro nodes are running).

Whenever the BroControl or Bro configuration is modified in any way, including changes to configuration files and site-specific policy scripts or upgrading to a new version of Bro, *deploy* must be run (deploy will check all policy scripts, install all needed files, and restart Bro). No changes will take effect until *deploy* is run.

The BroControl *cron* command performs housekeeping tasks, such as checking whether Bro is running or not (and starting or stopping to match the expected state, as needed), checking if there is sufficient free disk space, etc. This command is intended to be run from a cron job, rather than interactively by a user. To setup a cron job that runs once every five minutes, insert the following entry into the crontab of the user running BroControl (change the path to the actual location of broctl on your system):

```
0-59/5 * * * * /usr/local/bro/bin/broctl cron
```

If the `"broctl cron disable"` command is run, then broctl cron will be disabled (i.e., broctl cron won't do anything) until the `"broctl cron enable"` command is run. To check the status at any time, run `"broctl cron ?"`.

### Log Files

While Bro is running you can find the current set of (aggregated) logs in `logs/current` (which is a symlink to the corresponding spool directory). In a cluster setup, logs are written on the logger host (however, if there is no logger defined in your node.cfg, then the manager writes logs).

Bro logs are automatically rotated once per hour by default, or whenever Bro is stopped. A rotated log is renamed to contain a timestamp in the filename. For example, the `conn.log` might be renamed to `conn.2015-01-20-15-23-42.log`.

Immediately after a log is rotated, it is archived automatically. When a log is archived, it is moved to a subdirectory of `logs/` named by date (such as `logs/2015-01-20`), then it is renamed again, and gzipped. For example, a rotated log file named `conn.2015-01-20-15-23-42.log` might be archived to `logs/2015-01-20/conn.15:48:23-16:00:00.log.gz`. If the archival was successful, then the original (rotated) log file is removed.

If, for some reason, a rotated log file cannot be archived then it will be left in the node's working directory. Next time when BroControl either stops Bro or tries to restart a crashed Bro, it will try to archive such log files again. If this attempt fails, then an email is sent which contains the name of a directory where any such unarchived logs can be found.

### Files created only when using BroControl

When BroControl starts Bro it creates two files "stdout.log" and "stderr.log", which just capture stdout and stderr from Bro. Although these are not actually Bro logs, they might contain useful error or diagnostic information.

Also, whenever logs are rotated, a connection summary report is generated if the trace-summary tool is installed. Although these are not actually Bro logs, they follow the same filename convention as other Bro logs and they have the filename prefix "conn-summary". To prevent these files from being created, set the value of the *TraceSummary* option to an empty string.

### Site-specific Customization

You'll most likely want to adapt the Bro policy to the local environment and generally site-specific tuning requires writing local policy scripts.

Sample local policy scripts (which you can edit) are located in `share/bro/site`. In the stand-alone setup, a single file called `local.bro` gets loaded automatically. In the cluster setup, the same `local.bro` gets loaded, followed by one of three other files: `local-manager.bro`, `local-worker.bro`, and `local-proxy.bro` are loaded by the manager, workers, and proxy, respectively.

The recommended way to modify the policy is to use only "@load" directives in the `local.bro` scripts. For example, you can add a "@load" directive to load a Bro policy script that is included with Bro but is not loaded by default. You can also create custom site-specific policy scripts in the same directory as the `local.bro` scripts, and "@load" them from one of the `local.bro` scripts. For example, you could create your own Bro script `mypolicy.bro` in the `share/bro/site` directory, and then add a line "@load mypolicy" (without the quotes) to the `local.bro` script. Note that in a cluster setup, notice filtering should be done only on the manager.

After creating or modifying your local policy scripts, you must install them by using the BroControl "install" or "deploy" command. Next, you can use the BroControl "scripts" command to verify that your new scripts will be loaded when you start Bro.

If you want to change which local policy scripts are loaded by the nodes, you can set *SitePolicyStandalone* for all Bro instances, *SitePolicyManager* for the manager, and *SitePolicyWorker* for the workers. To change the directory where local policy scripts are located, set the option *SitePolicyPath* to a different path. These options can be changed in the `broctl.cfg` file.

### Load Order of Scripts

When writing custom site-specific policy scripts, it can sometimes be useful to know in which order the scripts are loaded (the BroControl "scripts" command shows the load order of every script loaded by Bro). For example, if more than one script sets a value for the same global variable, then the value that takes effect is the one set by the last such script loaded.

When BroControl starts Bro, the first script loaded is init-bare.bro, followed by init-default.bro (keep in mind that each of these scripts loads many other scripts).

Next, the local.bro script is loaded. This provides for a common set of loaded scripts for all nodes.

Next, the "broctl" script package is loaded. This consists of some standard settings that BroControl needs.

In a cluster setup, one of the following scripts are loaded: local-manager.bro, local-proxy.bro, or local-worker.bro.

The next scripts loaded are `local-networks.bro` and `broctl-config.bro`. These scripts are automatically generated by BroControl based on the contents of the `networks.cfg` and `broctl.cfg` files.

The last scripts loaded are any node-specific scripts specified with the option `aux_scripts` in `node.cfg`. This option can be used to load additional scripts to individual nodes only. For example, one could add a script `experimental.bro` to a single worker for trying out new experimental code.

### Mails

There are several situations when BroControl sends mail to the address given in *MailTo* (note that BroControl will not be able to send any mail when the value of the *SendMail* option is an empty string):

1. When the `broctl cron` command runs it performs various tasks (such as checking available disk space, expiring old log files, etc.). If any problems occur, a mail will be sent containing a list of those issues. Setting `MailHostUpDown=0` will disable some of this output. Also, setting `StatsLogEnable=0` will disable some functionality involving writing to stats.log (which could also reduce the amount of email).
2. When BroControl tries to start or stop (via any of these commands: start, stop, restart, deploy, or cron) a node that has crashed, a crash report is mailed (one for each crashed node). The crash report is essentially just the output of the `broctl diag` command. When `broctl cron` is run with the `--no-watch` option, then it will not attempt to start or stop Bro.
3. When BroControl stops Bro or restarts a crashed Bro, if any log files could not be archived, then an email will be sent. This can be disabled by setting `MailArchiveLogFail=0`.
4. If trace-summary is installed, a traffic summary is mailed each rotation interval. This can be disabled by setting `MailConnectionSummary=0`.

### Command Reference

The following summary lists all commands supported by BroControl. All commands may be either entered interactively or specified on the shell's command line. If not specified otherwise, commands taking *[<nodes>]* as arguments apply their action either to the given set of nodes, to the manager node if "manager" is given, to all proxy nodes if "proxies" is given, to all worker nodes if "workers" is given, or to all nodes if none are given.

***capstats [<nodes>] [<interval>]*** Determines the current load on the network interfaces monitored by each of the given worker nodes. The load is measured over the specified interval (in seconds), or by default over 10 seconds. This command uses the *capstats* tool, which is installed along with `broctl`.

***check [<nodes>]*** Verifies a modified configuration in terms of syntactical correctness (most importantly correct syntax in policy scripts). This command should be executed for each configuration change *before install* is used to put the change into place. The `check` command uses the policy files as found in *SitePolicyPath* to make sure they compile correctly. If they do, *install* will then copy them over to an internal place from where the nodes will read them at the next *start*. This approach ensures that new errors in a policy script will not affect currently running nodes, even when one or more of them needs to be restarted.

***cleanup [–all] [<nodes>]*** Clears the nodes' spool directories (if they are not running currently). This implies that their persistent state is flushed. Nodes that were crashed are reset into *stopped* state. If `--all` is specified, this command also removes the content of the node's *TmpDir*, in particular deleteing any data potentially saved there for reference from previous crashes. Generally, if you want to reset the installation back into a clean state, you can first *stop* all nodes, then execute `cleanup --all`, and finally *start* all nodes again.

***config*** Prints all configuration options with their current values.

***cron [enable|disable|?] | [–no-watch]*** This command has two modes of operation. Without arguments (or just `--no-watch`), it performs a set of maintenance tasks, including the logging of various statistical information, expiring old log files, checking for dead hosts, and restarting nodes which terminated unexpectedly (the latter can be suppressed with the `--no-watch` option if no auto-restart is desired). This mode is intended to be executed regularly via *cron*, as described in the installation instructions. While not intended for interactive use, no harm will be caused by executing the command manually: all the maintenance tasks will then just be performed one more time.

The second mode is for interactive usage and determines if the regular tasks are indeed performed when `broctl cron` is executed. In other words, even with `broctl cron` in your crontab, you can still temporarily disable it by running `cron disable`, and then later reenable with `cron enable`. This can be helpful while working, e.g., on the BroControl configuration and `cron` would interfere with that. `cron ?` can be used to query the current state.

***deploy*** Checks for errors in Bro policy scripts, then does an install followed by a restart on all nodes. This command should be run after any changes to Bro policy scripts or the broctl configuration, and after Bro is upgraded or even just recompiled.

This command is equivalent to running the *check*, *install*, and *restart* commands, in that order.

***df [<nodes>]*** Reports the amount of disk space available on the nodes. Shows only paths relevant to the broctl installation.

***diag [<nodes>]*** If a node has terminated unexpectedly, this command prints a (somewhat cryptic) summary of its final state including excerpts of any stdout/stderr output, resource usage, and also a stack backtrace if a core dump is found. The same information is sent out via mail when a node is found to have crashed (the "crash report"). While the information is mainly intended for debugging, it can also help to find misconfigurations (which are usually, but not always, caught by the *check* command).

***exec <command line>*** Executes the given Unix shell command line on all hosts configured to run at least one Bro instance. This is handy to quickly perform an action across all systems.

***exit*** Terminates the shell.

***help*** Prints a brief summary of all commands understood by the shell.

***install [–local]*** Reinstalls on all nodes (unless the `--local` option is given, in which case nothing will be propagated to other nodes), including all configuration files and local policy scripts. Usually all nodes should be reinstalled at the same time, as any inconsistencies between them will lead to strange effects. This command must be executed after *all* changes to any part of the broctl configuration (and after upgrading to a new version of Bro or BroControl), otherwise the modifications will not take effect. Before executing `install`, it is recommended to verify the configuration with *check*.

***netstats [<nodes>]*** Queries each of the nodes for their current counts of captured and dropped packets.

***nodes*** Prints a list of all configured nodes.

***peerstatus [<nodes>]*** Primarily for debugging, `peerstatus` reports statistics about the network connections cluster nodes are using to communicate with other nodes.

***print <id> [<nodes>]*** Reports the *current* live value of the given Bro script ID on all of the specified nodes (which obviously must be running). This can for example be useful to (1) check that policy scripts are working as expected, or (2) confirm that configuration changes have in fact been applied. Note that IDs defined inside a Bro namespace must be prefixed with `<namespace>::` (e.g., `print HTTP::mime_types_extensions` to print the corresponding table from `file-ident.bro`).

***process <trace> [options] [– <scripts>]*** Runs Bro offline on a given trace file using the same configuration as when running live. It does, however, use the potentially not-yet-installed policy files in *SitePolicyPath* and disables log rotation. Additional Bro command line flags and scripts can be given (each argument after a `--` argument is interpreted as a script).

Upon completion, the command prints a path where the log files can be found. Subsequent runs of this command may delete these logs.

In cluster mode, Bro is run with *both* manager and worker scripts loaded into a single instance. While that doesn't fully reproduce the live setup, it is often sufficient for debugging analysis scripts.

***quit*** Terminates the shell.

***restart [–clean] [<nodes>]*** Restarts the given nodes, or all nodes if none are specified. The effect is the same as first executing *stop* followed by a *start*, giving the same nodes in both cases.

If `--clean` is given, the installation is reset into a clean state before restarting. More precisely, a `restart --clean` turns into the command sequence *stop*, *cleanup*, *check*, *install*, and *start*.

***scripts [-c] [<nodes>]*** Primarily for debugging Bro configurations, the `scripts` command lists all the Bro scripts loaded by each of the nodes in the order they will be parsed by the node at startup. If `-c` is given, the command operates as *check* does: it reads the policy files from their *original* location, not the copies installed by *install*. The latter option is useful to check a not yet installed configuration.

***start [<nodes>]*** Starts the given nodes, or all nodes if none are specified. Nodes already running are left untouched.

***status [<nodes>]*** Prints the current status of the given nodes.

For each node, the information shown includes the node's name and type, the host where the node will run, the status, the PID, and the date/time when the node was started. The status column will usually show a status of either "stopped" or "running". A status of "crashed" means that BroControl verified that Bro is no longer running, but was expected to be running.

***stop [<nodes>]*** Stops the given nodes, or all nodes if none are specified. Nodes not running are left untouched.

***top [<nodes>]*** For each of the nodes, prints the status of the two Bro processes (parent process and child process) in a *top*-like format, including CPU usage and memory consumption. If executed interactively, the display is updated frequently until key `q` is pressed. If invoked non-interactively, the status is printed only once.

***update [<nodes>]*** After a change to Bro policy scripts, this command updates the Bro processes on the given nodes *while they are running* (i.e., without requiring a *restart*). However, such dynamic updates work only for a *subset* of Bro's full configuration. The following changes can be applied on the fly: The value of all const variables defined with the `&redef` attribute can be changed. More extensive script changes are not possible during runtime and always require a restart; if you change more than just the values of `&redef`-able consts and still issue `update`, the results are undefined and can lead to crashes. Also note that before running `update`, you still need to do an *install* (preferably after *check*), as otherwise `update` will not see the changes and it will resend the old configuration.

## Option Reference

This section summarizes the options that can be set in `broctl.cfg` for customizing the behavior of BroControl (the option names are case-insensitive). Usually, one only needs to change the "user options", which are listed first. The "internal options" are, as the name suggests, primarily used internally and set automatically. They are documented here only for reference.

### User Options

***BroArgs* (string, default _empty_)** Additional arguments to pass to Bro on the command-line (e.g. broargs=-f "tcp port 80").

***BroPort* (int, default 47760)** The TCP port number that Bro will listen on. For a cluster configuration, each node in the cluster will automatically be assigned a subsequent port to listen on.

***CommTimeout* (int, default 10)** The number of seconds to wait before assuming Broccoli communication events have timed out.

***CommandTimeout* (int, default 60)** The number of seconds to wait for a command to return results.

***CompressCmd* (string, default "gzip -9")** If archived logs will be compressed, the command to use for that. The specified command must compress its standard input to standard output.

***CompressExtension* (string, default "gz")** If archived logs will be compressed, the file extension to use on compressed log files. When specifying a file extension, don't include the period character (e.g., specify 'gz' instead of '.gz').

***CompressLogs* (bool, default 1)** True to compress archived log files.

***CronCmd* (string, default _empty_)** A custom command to run everytime the cron command has finished.

***Debug* (bool, default 0)** Enable extensive debugging output in spool/debug.log.

***Env_Vars* (string, default _empty_)** A comma-separated list of environment variables (e.g. env_vars=VAR1=123, VAR2=456) to set on all nodes immediately before starting Bro. Node-specific values (specified in the node configuration file) override these global values.

***HaveNFS* (bool, default 0)** True if shared files are mounted across all nodes via NFS (see the *FAQ*).

***IPv6Comm* (bool, default 1)** Enable IPv6 communication between cluster nodes (and also between them and BroControl). This overrides the Bro script variable Communication::listen_ipv6.

***KeepLogs* (string, default _empty_)** A space-separated list of filename shell patterns of expired log files to keep (empty string means don't keep any expired log files). The filename shell patterns are not regular expressions and do not include any directories. For example, specifying 'conn.* dns*' will prevent any expired log files with filenames starting with 'conn.' or 'dns' from being removed. Finally, note that this option is ignored if log files never expire.

***LogDir* (string, default "${BroBase}/logs")** Directory for archived log files.

***LogExpireInterval* (string, default "0")** Time interval that archived log files are kept (a value of 0 means log files never expire). The time interval is expressed as an integer followed by one of the following time units: day, hr, min.

***LogRotationInterval* (int, default 3600)** The frequency of log rotation in seconds for the manager/standalone node (zero to disable rotation). This overrides the Bro script variable Log::default_rotation_interval.

***MailAlarmsInterval* (int, default 86400)** The frequency (in seconds) of sending alarm summary mails (zero to disable). This overrides the Bro script variable Log::default_mail_alarms_interval.

***MailAlarmsTo* (string, default "${MailTo}")** Destination address for alarm summary mails. Default is to use the same address as MailTo. This overrides the Bro script variable Notice::mail_dest_pretty_printed.

***MailArchiveLogFail* (bool, default 1)** True to enable sending mail when log files fail to be archived.

***MailConnectionSummary* (bool, default 1)** True to mail connection summary reports each log rotation interval (if false, then connection summary reports will still be generated and archived, but they will not be mailed). However, this option has no effect if the trace-summary script is not available.

***MailFrom* (string, default "Big Brother <bro@localhost>")** Originator address for mails. This overrides the Bro script variable Notice::mail_from.

***MailHostUpDown*** **(bool, default 1)** True to enable sending mail when broctl cron notices the availability of a host in the cluster to have changed.

***MailReplyTo*** **(string, default _empty_)** Reply-to address for broctl-generated mails.

***MailSubjectPrefix*** **(string, default "[Bro]")** General Subject prefix for mails. This overrides the Bro script variable Notice::mail_subject_prefix.

***MailTo*** **(string, default "<user>")** Destination address for non-alarm mails. This overrides the Bro script variable Notice::mail_dest.

***MakeArchiveName*** **(string, default "${BroBase}/share/broctl/scripts/make-archive-name")** Script to generate filenames for archived log files.

***MemLimit*** **(string, default "unlimited")** Maximum amount of memory for Bro processes to use (in KB, or the string 'unlimited').

***MinDiskSpace*** **(int, default 5)** Percentage of minimum disk space available before warning is mailed.

***PFRINGClusterID*** **(int, default 0)** If PF_RING flow-based load balancing is desired, this is where the PF_RING cluster id is defined. The default value is configuration-dependent and determined automatically by CMake at configure-time based upon whether PF_RING's enhanced libpcap is available. Bro must be linked with PF_RING's libpcap wrapper for this option to work.

***PFRINGClusterType*** **(string, default "4-tuple")** If PF_RING flow-based load balancing is desired, this is where the PF_RING cluster type is defined. Allowed values are: 2-tuple, 4-tuple, 5-tuple, tcp-5-tuple, 6-tuple, or round-robin. Bro must be linked with PF_RING's libpcap wrapper and PFRINGClusterID must be non-zero for this option to work.

***PFRINGFirstAppInstance*** **(int, default 0)** The first application instance for a PF_RING dnacluster interface to use. Broctl will start at this application instance number and increment for each new process running on that DNA cluster. Bro must be linked with PF_RING's libpcap wrapper, PFRINGClusterID must be non-zero, and you must be using PF_RING+DNA and libzero for this option to work.

***Prefixes*** **(string, default "local")** Additional script prefixes for Bro, separated by colons. Use this instead of @prefix.

***SaveTraces*** **(bool, default 0)** True to let backends capture short-term traces via '-w'. These are not archived but might be helpful for debugging.

***SendMail*** **(string, default "@SENDMAIL@")** Location of the sendmail binary. Make this string blank to prevent email from being sent. The default value is configuration-dependent and determined automatically by CMake at configure-time. This overrides the Bro script variable Notice::sendmail.

***SitePluginPath*** **(string, default _empty_)** Directories to search for custom plugins (i.e., plugins that are not included with broctl), separated by colons.

***SitePolicyManager*** **(string, default "local-manager.bro")** Space-separated list of local policy files for manager.

***SitePolicyPath*** **(string, default "${PolicyDir}/site")** Directories to search for local (i.e., site-specific) policy files, separated by colons. For each such directory, all files and subdirectories are copied to PolicyDirSiteInstall during broctl 'install' or 'deploy' (however, if the same file or subdirectory is found in more than one such directory, then only the first one encountered will be used).

***SitePolicyStandalone*** **(string, default "local.bro")** Space-separated list of local policy files for all Bro instances.

***SitePolicyWorker*** **(string, default "local-worker.bro")** Space-separated list of local policy files for workers.

***StatsLogEnable*** **(bool, default 1)** True to enable BroControl to write statistics to the stats.log file.

***StatsLogExpireInterval*** **(int, default 0)** Number of days entries in the stats.log file are kept (zero means never expire).

***StatusCmdShowAll*** **(bool, default 0)** True to have the status command show all output, or False to show only some of the output (peer information will not be collected or shown, so the command will run faster).

***StopTimeout* (int, default 60)** The number of seconds to wait before sending a SIGKILL to a node which was previously issued the 'stop' command but did not terminate gracefully.

***TimeFmt* (string, default "%d %b %H:%M:%S")** Format string to print date/time specifications (see 'man strftime').

***TimeMachineHost* (string, default _empty_)** If the manager should connect to a Time Machine, the address of the host it is running on.

***TimeMachinePort* (string, default "47757/tcp")** If the manager should connect to a Time Machine, the port it is running on (in Bro syntax, e.g., 47757/tcp).

***ZoneID* (string, default _empty_)** If the host running BroControl is managing a cluster comprised of nodes with non-global IPv6 addresses, this option indicates what **RFC 4007** zone_id to append to node addresses when communicating with them.

## Internal Options

***BinDir* (string, default "${BroBase}/bin")** Directory for executable files.

***Bro* (string, default "${BinDir}/bro")** Path to Bro binary.

***BroBase* (string, default _empty_)** Base path of broctl installation on all nodes.

***CapstatsPath* (string, default "${bindir}/capstats")** Path to capstats binary; empty if not available.

***CfgDir* (string, default "${BroBase}/etc")** Directory for configuration files.

***DebugLog* (string, default "${SpoolDir}/debug.log")** Log file for debugging information.

***HelperDir* (string, default "${BroBase}/share/broctl/scripts/helpers")** Directory for broctl helper scripts.

***LibDir* (string, default "${BroBase}/lib")** Directory for library files.

***LibDirInternal* (string, default "${BroBase}/lib/broctl")** Directory for broctl-specific library files.

***LocalNetsCfg* (string, default "${CfgDir}/networks.cfg")** File defining the local networks.

***LockFile* (string, default "${SpoolDir}/lock")** Lock file preventing concurrent shell operations.

***LogExpireMinutes* (int, default 0)** Time interval (in minutes) that archived log files are kept (0 means they never expire). Users should never modify this value (see the LogExpireInterval option).

***NodeCfg* (string, default "${CfgDir}/node.cfg")** Node configuration file.

***OS* (string, default _empty_)** Name of operating system as reported by uname.

***PcapBufsize* (int, default 128)** Number of Mbytes to provide as buffer space when capturing from live interfaces via libpcap.

***PcapSnaplen* (int, default 8192)** Number of bytes per packet to capture from live interfaces via libpcap.

***PluginBroDir* (string, default "${BroBase}/lib/bro/plugins")** Directory where Bro plugins are located. BroControl will search this directory tree for broctl plugins that are provided by any Bro plugin.

***PluginDir* (string, default "${LibDirInternal}/plugins")** Directory where standard broctl plugins are located.

***PolicyDir* (string, default "${BroScriptDir}")** Directory for standard policy files.

***PolicyDirSiteInstall* (string, default "${SpoolDir}/installed-scripts-do-not-touch/site")** Directory where the shell copies local (i.e., site-specific) policy scripts when installing.

***PolicyDirSiteInstallAuto* (string, default "${SpoolDir}/installed-scripts-do-not-touch/auto")** Directory where the shell copies auto-generated local policy scripts when installing.

***PostProcDir* (string, default "${BroBase}/share/broctl/scripts/postprocessors")** Directory for log postprocessors.

***ScriptsDir* (string, default "${BroBase}/share/broctl/scripts")** Directory for executable scripts shipping as part of broctl.

***SpoolDir* (string, default "${BroBase}/spool")** Directory for run-time data.

***StandAlone* (bool, default 0)** True if running in stand-alone mode (see elsewhere).

***StateFile* (string, default "${SpoolDir}/state.db")** File storing the current broctl state.

***StaticDir* (string, default "${BroBase}/share/broctl")** Directory for static, arch-independent files.

***StatsDir* (string, default "${LogDir}/stats")** Directory where statistics are kept.

***StatsLog* (string, default "${SpoolDir}/stats.log")** Log file for statistics.

***Time* (string, default _empty_)** Path to time binary.

***TmpDir* (string, default "${SpoolDir}/tmp")** Directory for temporary data.

***TmpExecDir* (string, default "${SpoolDir}/tmp")** Directory where binaries are copied before execution. This option is ignored if HaveNFS is 0.

***TraceSummary* (string, default "${bindir}/trace-summary")** Path to trace-summary script (empty if not available). Make this string blank to disable the connection summary reports.

***Version* (string, default _empty_)** Version of the broctl.

## Plugins

BroControl provides a plugin interface to extend its functionality. A plugin is written in Python and can do any, or all, of the following:

- Perform actions before or after any of the standard BroControl commands is executed. When running before the actual command, it can filter which nodes to operate or stop the execution altogether. When running after the command, it gets access to the commands success status on a per-node basis (where applicable).
- Add custom commands to BroControl.
- Add custom options to BroControl defined in `broctl.cfg`.
- Add custom keys to nodes defined in `node.cfg`.

A plugin is written by deriving a new class from BroControl class *Plugin*. The Python script with the new plugin is then copied into a plugin directory searched by BroControl at startup. By default, BroControl searches `<prefix>/lib/broctl/plugins`; additional directories may be configured by setting the *SitePluginPath* option. Note that any plugin script must end in `*.py` to be found. BroControl comes with some example plugins that can be used as a starting point; see the `<prefix>/lib/broctl/plugins` directory.

In the following, we document the API that is available to plugins. A plugin must be derived from the *Plugin* class, and can use its methods as well as those of the *Node* class.

### Class `Plugin`

**class Plugin** The class `Plugin` is the base class for all BroControl plugins.

The class has a number of methods for plugins to override, and every plugin must at least override `name()` and `pluginVersion()`.

For each BroControl command `foo`, there are two methods, `cmd_foo_pre` and `cmd_foo_post`, that are called just before the command is executed and just after it has finished, respectively. The arguments these methods receive correspond to their command-line parameters, and are further documented below.

The `cmd_<XXX>_pre` methods have the ability to prevent the command's execution, either completely or partially for those commands that take nodes as parameters. In the latter case, the method receives a list of nodes that the command is to be run on, and it can filter that list and returns modified version of nodes to actually use. The standard case would be returning simply the unmodified `nodes` parameter. To completely block the command's execution, return an empty list. To just not execute the command for a subset, remove the affected ones. For commands that do not receive nodes as arguments, the return value is interpreted as boolean indicating whether command execution should proceed (True) or not (False).

The `cmd_<XXX>_post` methods likewise receive the commands arguments as their parameter, as documented below. For commands taking nodes, the list corresponds to those nodes for which the command was actually executed (i.e., after any `cmd_<XXX>_pre` filtering).

Note that if a plugin prevents a command from executing either completely or partially, it should report its reason via the `message()` or `error()` methods.

If multiple plugins hook into the same command, all their `cmd_<XXX>_{pre,post}` are executed in undefined order. The command is executed on the intersection of all `cmd_<XXX>_pre` results.

Finally, note that the `restart` command is just a combination of other commands and thus their callbacks are run in addition to the callbacks for `restart`. **debug** (self, msg)

Logs a debug message in BroControl's debug log if enabled.

**error** (self, msg)

Reports an error to the user.

**execute** (self, node, cmd)

Executes a command on the host for the given *node* of type *Node*. Returns a tuple `(success,output)` in which `success` is True if the command ran successfully and `output` is the combined stdout/stderr output.

**executeParallel** (self, cmds)

Executes a set of commands in parallel on multiple hosts. `cmds` is a list of tuples `(node,cmd)`, in which the *node* is a *Node* instance and *cmd* is a string with the command to execute for it. The method returns a list of tuples `(node,success,output)`, in which `success` is True if the command ran successfully and `output` is the combined stdout/stderr output for the corresponding `node`.

**getGlobalOption** (self, name)

Returns the value of the global BroControl option or state attribute *name*. If the user has not set the options, its default value is returned. See the output of `broctl config` for a complete list.

**getOption** (self, name)

Returns the value of one of the plugin's options, *name*.

An option has a default value (see *options()*), which can be overridden by a user in `broctl.cfg`. An option's value cannot be changed by the plugin.

**getState** (self, name)

Returns the current value of one of the plugin's state variables, *name*. The returned value will always be a string. If it has not yet been set, an empty string will be returned.

Different from options, state variables can be set by the plugin. They are persistent across restarts.

Note that a plugin cannot query any global BroControl state variables.

**hosts** (self, nodes)

Returns a list of *Node* objects which is a subset of the list in *nodes*, such that only one node per host will be chosen. If *nodes* is empty, then the returned list will be a subset of the entire list of configured nodes.

**message** (self, msg)

Reports a message to the user.

**nodes** (self)

Returns a list of all configured *Node* objects.

**parseNodes** (self, names)

Returns a tuple which contains two lists. The first list is a list of *Node* objects for a string of space-separated node names. If a name does not correspond to a known node, then the name is added to the second list in the returned tuple.

**setState** (self, name, value)

Sets one of the plugin's state variables, *name*, to *value*. *value* must be a string. The change is permanent and will be recorded to disk.

Note that a plugin cannot change any global BroControl state variables.

**broProcessDied** (self, node)

Called when BroControl finds the Bro process for *Node node* to have terminated unexpectedly. This method will be called just before BroControl prepares the node's "crash report" and before it cleans up the node's spool directory.

This method can be overridden by derived classes. The default implementation does nothing.

**broctl_config** (self)

Returns a string containing Bro script code that should be written to the dynamically generated Bro script named "broctl-config.bro". This provides a way for plugins to easily add Bro script code that depends on broctl settings.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_capstats_post** (self, nodes, interval)

Called just after the `capstats` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_capstats_pre** (self, nodes, interval)

Called just before the `capstats` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command. *interval* is an integer with the measurement interval in seconds.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_check_post** (self, results)

Called just after the `check` command has finished. It receives the list of 2-tuples `(node,bool)` indicating the nodes the command was executed for, along with their success status.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_check_pre** (self, nodes)

Called just before the `check` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_cleanup_post** (self, nodes, all)

Called just after the `cleanup` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_cleanup_pre** (self, nodes, all)

Called just before the `cleanup` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command. *all* is boolean indicating whether the `--all` argument has been given.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_config_post** (self)

Called just after the `config` command has finished.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_config_pre** (self)

Called just before the `config` command is run. Returns a boolean indicating whether or not the command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_cron_post** (self, arg, watch)

Called just after the `cron` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_cron_pre** (self, arg, watch)

Called just before the `cron` command is run. *arg* is an empty string if the command is executed without arguments. Otherwise, it is one of the strings: `enable`, `disable`, `?`. *watch* is a boolean indicating whether the `cron` command should restart abnormally terminated Bro processes; it's only valid if *arg* is empty.

Returns a boolean indicating whether or not the `cron` command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_custom** (self, cmd, args, cmdout)

Called when a command defined by the `commands` method is executed. *cmd* is the command (without the plugin's prefix), and *args* is a single string with all arguments. It returns a CmdResult object containing the command results.

If the arguments are actually node names, `parseNodes` can be used to get the *Node* objects.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_deploy_post** (self)

Called just after the `deploy` command has finished.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_deploy_pre** (self)

Called just before the `deploy` command is run. Returns a boolean indicating whether or not the command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_df_post** (self, nodes)

Called just after the `df` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_df_pre** (self, nodes)

Called just before the `df` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_diag_post** (self, nodes)

Called just after the `diag` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_diag_pre** (self, nodes)

Called just before the `diag` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_exec_post** (self, cmdline)

Called just after the `exec` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_exec_pre** (self, cmdline)

Called just before the `exec` command is run. *cmdline* is a string with the command line to execute.

Returns a boolean indicating whether or not the `exec` command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_install_post** (self)

Called just after the `install` command has finished.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_install_pre** (self)

Called just before the `install` command is run. Returns a boolean indicating whether or not the command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_netstats_post** (self, nodes)

Called just after the `netstats` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_netstats_pre** (self, nodes)

Called just before the `netstats` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_nodes_post** (self)

Called just after the `nodes` command has finished.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_nodes_pre** (self)

Called just before the `nodes` command is run. Returns a boolean indicating whether or not the command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_peerstatus_post** (self, nodes)

Called just after the `peerstatus` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_peerstatus_pre** (self, nodes)

Called just before the `peerstatus` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_print_post** (self, nodes, id)

Called just after the `print` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_print_pre** (self, nodes, id)

Called just before the `print` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command. *id* is a string with the name of the ID to be printed.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_process_post** (self, trace, options, scripts, success)

Called just after the `process` command has finished. Arguments are as with the `pre` method, plus an additional boolean *success* indicating whether Bro terminated normally.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_process_pre** (self, trace, options, scripts)

Called just before the `process` command is run. It receives the *trace* to read from as a string, a list of additional Bro *options*, and a list of additional Bro *scripts*.

Returns a boolean indicating whether or not the `process` command should run.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_restart_post** (self, nodes)

Called just after the `restart` command has finished. It receives a list of *nodes* indicating the nodes on which the command was executed.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_restart_pre** (self, nodes, clean)

Called just before the `restart` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command. *clean* is boolean indicating whether the `--clean` argument has been given.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_scripts_post** (self, nodes, check)

Called just after the `scripts` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_scripts_pre** (self, nodes, check)

Called just before the `scripts` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command. *check* is boolean indicating whether the `-c` option was given.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_start_post** (self, results)

Called just after the `start` command has finished. It receives the list of 2-tuples `(node,bool)` indicating the nodes the command was executed for, along with their success status.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_start_pre** (self, nodes)

Called just before the `start` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_status_post** (self, nodes)

Called just after the `status` command has finished. Arguments are as with the `pre` method.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_status_pre** (self, nodes)

Called just before the `status` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_stop_post** (self, results)

Called just after the `stop` command has finished. It receives the list of 2-tuples `(node,bool)` indicating the nodes the command was executed for, along with their success status.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_stop_pre** (self, nodes)

Called just before the `stop` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_top_post** (self, nodes)

Called just after the `top` command has finished. Arguments are as with the `pre` method. Note that when `top` is run interactively to auto-refresh continuously, this method will be called once after each update.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_top_pre** (self, nodes)

Called just before the `top` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command. Note that when `top` is run interactively to auto-refresh continuously, this method will be called once before each update.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_update_post** (self, results)

Called just after the `update` command has finished. It receives the list of 2-tuples `(node,bool)` indicating the nodes the command was executed for, along with their success status.

This method can be overridden by derived classes. The default implementation does nothing.

**cmd_update_pre** (self, nodes)

Called just before the `update` command is run. It receives the list of nodes, and returns the list of nodes that should proceed with the command.

This method can be overridden by derived classes. The default implementation does nothing.

**commands** (self)

Returns a set of custom commands provided by the plugin.

The return value is a list of 3-tuples each having the following elements:

**`command`** A string with the command's name. Note that the command name exposed to the user will be prefixed with the plugin's prefix as returned by *prefix()* (e.g., `myplugin.mycommand`).

**`arguments`** A string describing the command's arguments in a textual form suitable for use in the `help` command summary (e.g., `[<nodes>]` for a command taking an optional list of nodes). Empty if no arguments are expected.

**`description`** A string with a description of the command's semantics suitable for use in the `help` command summary.

This method can be overridden by derived classes. The implementation must not call the parent class' implementation. The default implementation returns an empty list.

**done** (self)

Called once just before BroControl terminates. This method can do any cleanup the plugin may require.

This method can be overridden by derived classes. The default implementation does nothing.

**hostStatusChanged** (self, host, status)

Called when BroControl's `cron` command finds the availability of a cluster system to have changed. Initially, all systems are assumed to be up and running. Once BroControl notices that a system isn't responding (defined as not accepting SSH sessions), it calls this method, passing in a string with the name of the *host* and a boolean *status* set to False. Once the host becomes available again, the method will be called again for the same host with *status* now set to True.

Note that BroControl's `cron` tracks a host's availability across execution, so if the next time it's run the host is still down, this method will not be called again.

This method can be overridden by derived classes. The default implementation does nothing.

**init** (self)

Called once just before BroControl starts executing any commands. This method can do any initialization that the plugin may require.

Note that when this method executes, BroControl guarantees that all internals are fully set up (e.g., user-defined options are available). This may not be the case when the class `__init__` method runs.

Returns a boolean, indicating whether the plugin should be used. If it returns `False`, the plugin will be removed and no other methods called.

This method can be overridden by derived classes. The default implementation always returns True.

**name** (self)

Returns a string with a descriptive name for the plugin (e.g., `"TestPlugin"`). The name must not contain any whitespace.

This method must be overridden by derived classes. The implementation must not call the parent class' implementation.

**nodeKeys** (self)

Returns a list of names of custom keys (the value of a key can be specified in `node.cfg` for any node defined there). The value for a key will be available from the *Node* object as attribute `<prefix>_<key>` (e.g., `node.myplugin_mykey`). If not set, the attribute will be set to an empty string.

This method can be overridden by derived classes. The implementation must not call the parent class' implementation. The default implementation returns an empty list.

**options** (self)

Returns a set of local configuration options provided by the plugin.

The return value is a list of 4-tuples each having the following elements:

**`name`** A string with name of the option (e.g., `Path`). Option names are case-insensitive. Note that the option name exposed to the user will be prefixed with your plugin's prefix as returned by *prefix()* (e.g., `myplugin.Path`).

**`type`** A string with type of the option, which must be one of `"bool"`, `"string"`, or `"int"`.

**`default`** The option's default value. Note that this value must be enclosed in quotes if the type is "string", and must not be enclosed in quotes if the type is not "string".

**`description`** A string with a description of the option semantics.

This method can be overridden by derived classes. The implementation must not call the parent class' implementation. The default implementation returns an empty list.

**pluginVersion** (self)

Returns an integer with a version number for the plugin. Plugins should increase their version number with any significant change.

This method must be overridden by derived classes. The implementation must not call the parent class' implementation.

**prefix** (self)

Returns a string with a prefix for the plugin's options and commands names (e.g., "myplugin").

This method can be overridden by derived classes. The implementation must not call the parent class' implementation. The default implementation returns a lower-cased version of *name()*.

## Class `Node`

**class Node** Class representing one node of the BroControl maintained setup. In standalone mode, there's always exactly one node of type `standalone`. In a cluster setup, there is zero or one of type `logger`, exactly one of type `manager`, one or more of type `proxy`, and zero or more of type `worker`. The manager will handle writing logs if there is no logger defined in a cluster.

A `Node` object has a number of keys with values that are set via the `node.cfg` file and can be accessed directly (from a plugin) via corresponding Python attributes (e.g., `node.name`):

**`name`** **(string)** The name of the node, which corresponds to the `[<name>]` section in `node.cfg`.

**`type`** **(string)** The type of the node. In a standalone configuration, the only allowed type is `standalone`. In a cluster configuration, the type must be one of: `logger`, `manager`, `proxy`, or `worker`.

**`env_vars`** **(string)** A comma-separated list of environment variables to set when running Bro (e.g., `env_vars=VAR1=1,VAR2=2`). These node-specific values override any global values specified in the `broctl.cfg` file.

**`host`** **(string)** The hostname (or the IP address) of the system the node is running on.

**`interface`** **(string)** The network interface for Bro to use; empty if not set.

**`lb_procs`** **(integer)** The number of clustered Bro workers you'd like to start up. This number must be greater than zero.

**`lb_method`** **(string)** The load balancing method to distribute packets to all of the processes. This must be one of: `pf_ring`, `myricom`, `custom`, or `interfaces`.

**`lb_interfaces`** **(string)** If the load balancing method is `interfaces`, then this is a comma-separated list of network interface names to use.

**`pin_cpus`** **(string)** A comma-separated list of CPU numbers to which the node's Bro processes will be pinned (if not specified, then CPU pinning will not be used for this node). This option is only supported on Linux and FreeBSD (it is ignored on all other platforms). CPU numbering starts at zero (e.g., the only valid CPU numbers for a machine with one dual-core processor would be 0 and 1). If the length of this list does not match the number of Bro processes for this node, then some CPUs could have zero (if too many CPU numbers are specified) or more than one (if not enough CPU numbers are specified) Bro processes pinned to them. Only the specified CPU numbers will be used, regardless of whether additional CPU cores exist.

**`aux_scripts`** **(string)** Any node-specific Bro script configured for this node.

**`zone_id`** **(string)** If BroControl is managing a cluster comprised of nodes using non-global IPv6 addresses, then this configures the **RFC 4007** `zone_id` string that the node associates with the common zone that all cluster nodes are a part of. This identifier may differ between nodes.

Any attribute that is not defined in `node.cfg` will be empty.

In addition, plugins can override *Plugin.nodeKeys* to define their own node keys, which can then be likewise set in `node.cfg`. The key names will be prepended with the plugin's *Plugin.prefix* (e.g., for the plugin `test`, the node key `foo` is set by adding `test.foo=value` to `node.cfg`).

Finally, a Node object has the following methods that can be called from a plugin: **cwd** (self)

Returns a string with the node's working directory.

**describe** (self)

Returns an extended string representation of the node including all its keys with values (sorted by key).

**getPID** (self)

Returns the process ID of the node's Bro process if running, and None otherwise.

**getPort** (self)

Returns an integer with the port that this node's communication system is listening on for incoming connections, or -1 if no such port has been set yet.

**hasCrashed** (self)

Returns True if the node's Bro process has exited abnormally.

### Questions and Answers

***Can I use an NFS-mounted partition as the cluster's base directory to avoid the ''rsync''ing?*** Yes. *BroBase* can be on an NFS partition. Configure and install the shell as usual with `--prefix=<BroBase>`. Then add `HaveNFS=1` and `SpoolDir=<spath>` to `broctl.cfg`, where `<spath>` is a path on the local disks of the nodes; `<spath>` will be used for all non-shared data (make sure that the parent directory exists and is writable on all nodes!). Then run `make install` again. Finally, you can remove `<BroBase>/spool` (or link it to <spath>). In addition, you might want to keep the log files locally on the nodes as well by setting *LogDir* to a non-NFS directory. (Only the manager's logs will be kept permanently, the logs of workers/proxies are discarded upon rotation.)

***What do I need to do when something in the Bro distribution changes?*** After pulling from the main Bro git repository, just re-run `make install` inside your build directory. It will reinstall all the files from the distribution that are not up-to-date. Then do `broctl deploy` to make sure everything gets pushed out.

***Can I change the naming scheme that BroControl uses for archived log files?*** Yes, set *MakeArchiveName* to a script that outputs the desired destination file name for an archived log file. The default script for that task is `<BroBase>/share/broctl/scripts/make-archive-name`, which you can use as a template for creating your own version. See the beginning of that script for instructions.

***Can BroControl manage a cluster of nodes over non-global IPv6 scope (e.g. link-local)?*** Yes, set `ZoneID` in `broctl.cfg` to the zone identifier that the BroControl node uses to identify the scope zone (the `ifconfig` command output is usually helpful, if it doesn't show the zone identifier appended to the address with a '%' character, then it may just be the interface name). Then in `node.cfg`, add a `zone_id` key to each node section representing that particular node's zone identifier and set the `host` key to the IPv6 address assigned to the node within the scope zone. Most nodes probably have the same `zone_id`, but may not if their interface configuration differs. See **RFC 4007** for more information on IPv6 scoped addresses and zones.

## 3.3.9 Bro Auxiliary Programs

**Contents**

**Version** 0.35-27

Handy auxiliary programs related to the use of the Bro Network Security Monitor (http://www.bro.org).

### Installation

Installation is simple and standard:

```
./configure
make
make install
```

### adtrace

The “adtrace” utility is used to compute the network address that compose the internal and extern nets that bro is monitoring. This program just reads a pcap (tcpdump) file and writes out the src MAC, dst MAC, src IP, dst IP for each packet seen in the file.

### bro-cut

The “bro-cut” utility reads ASCII Bro logs on standard input and outputs them to standard output with only the specified columns (the column names can be found in each log file in the “#fields” header line). The specified order of the column names determines the output order of the columns (i.e., “bro-cut” can reorder the columns). If no column names are specified, then “bro-cut” simply outputs all columns.

There are several command-line options available to modify the output (run “bro-cut -h” to see a list of all options). For example, there are options to convert timestamps into human-readable format, and options to specify whether or not to include the format header lines in the output (by default, they’re not included). For example, the following command will output the three specified columns from conn.log with the timestamps from the “ts” column being converted to human-readable format:

```
cat conn.log | bro-cut -d ts id.orig_h id.orig_p
```

The “bro-cut” utility can read the concatenation of one or more uncompressed ASCII log files produced by Bro version 2.0 or newer, as long as each log file contains format header lines (i.e., lines at the beginning of the file starting with “#”). So, for example, “bro-cut” cannot read a Bro log file in the JSON format. To read a compressed log file, a tool such as “zcat” must be used to uncompress the file. For example, “bro-cut” can read a group of compressed conn.log files with a command like this:

```
zcat conn.*.log.gz | bro-cut
```

### devel-tools

A set of scripts used commonly for Bro development. Note that none of these scripts are installed by ‘make install’.

**extract-conn-by-uid** Extracts a connection from a trace file based on its UID found in Bro’s conn.log

**gen-mozilla-ca-list.rb** Generates list of Mozilla SSL root certificates in a format readable by Bro.

**update-changes** A script to maintain the CHANGES and VERSION files.

**git-show-fastpath** Show commits to the fastpath branch not yet merged into master.

**cpu-bench-with-trace** Run a number of Bro benchmarks on a trace file.

### rst

The “rst” utility can be invoked by a Bro script to terminate an established TCP connection by forging RST tear-down packets.

### 3.3.10 BTest - A Simple Driver for Basic Unit Tests

The `btest` is a simple framework for writing unit tests. Freely borrowing some ideas from other packages, it's main objective is to provide an easy-to-use, straightforward driver for a suite of shell-based tests. Each test consists of a set of command lines that will be executed, and success is determined based on their exit codes. `btest` comes with some additional tools that can be used within such tests to compare output against a previously established baseline.

**Contents**

#### Download

You can find the latest BTest release for download at http://www.bro.org/download.

BTest's git repository is located at git://git.bro.org/btest.git. You can browse the repository here.

This document describes BTest 0.54-65. See the `CHANGES` file for version history.

## Prerequisites

BTest has the following prerequisites:

- Python version >= 2.6.
- Bash (note that on FreeBSD, bash is not installed by default).

BTest has the following optional prerequisites to enable additional functionality:

- Sphinx.
- perf (Linux only). Note that on Debian/Ubuntu, you also need to install the "linux-tools" package.

## Installation

Installation is simple and standard:

```
tar xzvf btest-*.tar.gz
cd btest-*
python setup.py install
```

This will install a few scripts: `btest` is the main driver program, and there are a number of further helper scripts that we discuss below (including `btest-diff`, which is a tool for comparing output to a previously established baseline).

## Writing a Simple Test

In the most simple case, `btest` simply executes a set of command lines, each of which must be prefixed with `@TEST-EXEC:`

```
> cat examples/t1
@TEST-EXEC: echo "Foo" | grep -q Foo
@TEST-EXEC: test -d .
> btest examples/t1
examples.t1 ... ok
```

The test passes as both command lines return success. If one of them didn't, that would be reported:

```
> cat examples/t2
@TEST-EXEC: echo "Foo" | grep -q Foo
@TEST-EXEC: test -d DOESNOTEXIST
> btest examples/t2
examples.t2 ... failed
```

Usually you will just run all tests found in a directory:

```
> btest examples
examples.t1 ... ok
examples.t2 ... failed
1 test failed
```

Why do we need the `@TEST-EXEC:` prefixes? Because the file containing the test can simultaneously act as *its input*. Let's say we want to verify a shell script:

```
> cat examples/t3.sh
# @TEST-EXEC: sh %INPUT
ls /etc | grep -q passwd
> btest examples/t3.sh
examples.t3 ... ok
```

Here, `btest` is executing (something similar to) `sh examples/t3.sh`, and then checks the return value as usual. The example also shows that the `@TEST-EXEC` prefix can appear anywhere, in particular inside the comment section of another language.

Now, let's say we want to check the output of a program, making sure that it matches what we expect. For that, we first add a command line to the test that produces the output we want to check, and then run `btest-diff` to make sure it matches a previously recorded baseline. `btest-diff` is itself just a script that returns success if the output is as expected, and failure otherwise. In the following example, we use an awk script as a fancy way to print all file names starting with a dot in the user's home directory. We write that list into a file called `dots` and then check whether its content matches what we know from last time:

```
> cat examples/t4.awk
# @TEST-EXEC: ls -a $HOME | awk -f %INPUT >dots
# @TEST-EXEC: btest-diff dots
/^\.+/ { print $1 }
```

Note that each test gets its own little sandbox directory when run, so by creating a file like `dots`, you aren't cluttering up anything.

The first time we run this test, we need to record a baseline:

```
> btest -U examples/t4.awk
```

Now, `btest-diff` has remembered what the `dots` file should look like:

```
> btest examples/t4.awk
examples.t4 ... ok
> touch ~/.NEWDOTFILE
> btest examples/t4.awk
examples.t4 ... failed
1 test failed
```

If we want to see what exactly the unexpected change is that was introduced to `dots`, there's a *diff* mode for that:

```
> btest -d examples/t4.awk
examples.t4 ... failed
% 'btest-diff dots' failed unexpectedly (exit code 1)
% cat .diag
== File ===============================
[... current dots file ...]
== Diff ===============================
--- /Users/robin/work/binpacpp/btest/Baseline/examples.t4/dots
2010-10-28 20:11:11.000000000 -0700
+++ dots      2010-10-28 20:12:30.000000000 -0700
@@ -4,6 +4,7 @@
.CFUserTextEncoding
.DS_Store
.MacOSX
+.NEWDOTFILE
.Rhistory
.Trash
.Xauthority
```

```
========================================

% cat .stderr
[... if any of the commands had printed something to stderr, that would follow here ..
↪.]
```

Once we delete the new file, we are fine again:

```
> rm ~/.NEWDOTFILE
> btest -d examples/t4.awk
examples.t4 ... ok
```

That's already the main functionality that the `btest` package provides. In the following, we describe a number of further options extending/modifying this basic approach.

## Reference

### Command Line Usage

`btest` must be started with a list of tests and/or directories given on the command line. In the latter case, the default is to recursively scan the directories and assume all files found to be tests to perform. It is however possible to exclude specific files and directories by specifying a suitable *configuration file*.

`btest` returns exit code 0 if all tests have successfully passed, and 1 otherwise.

`btest` accepts the following options:

**-a ALTERNATIVE, --alternative=ALTERNATIVE** Activates an *alternative* configuration defined in the configuration file. Multiple alternatives can be given as a comma-separated list (in this case, all specified tests are run once for each specified alternative). If `ALTERNATIVE` is `-` that refers to running with the standard setup, which can be used to run tests both with and without alternatives by giving both.

**-b, --brief** Does not output *anything* for tests which pass. If all tests pass, there will not be any output at all except final summary information.

**-c CONFIG, --config=CONFIG** Specifies an alternative *configuration file* to use. If not specified, the default is to use a file called `btest.cfg` if found in the current directory. An alternative way to specify a different config file is with the `BTEST_CFG` environment variable (however, the command-line option overrides `BTEST_CFG`).

**-d, --diagnostics** Reports diagnostics for all failed tests. The diagnostics include the command line that failed, its output to standard error, and potential additional information recorded by the command line for diagnostic purposes (see *@TEST-EXEC* below). In the case of `btest-diff`, the latter is the `diff` between baseline and actual output.

**-D, --diagnostics-all** Reports diagnostics for all tests, including those which pass.

**-f DIAGFILE, --file-diagnostics=DIAGFILE** Writes diagnostics for all failed tests into the given file. If the file already exists, it will be overwritten.

**-g GROUPS, --groups=GROUPS** Runs only tests assigned to the given test groups, see *@TEST-GROUP*. Multiple groups can be given as a comma-separated list. Specifying groups with a leading `-` leads to all tests to run that are *not* not part of them. Specifying a sole `-` as a group name selects all tests that do not belong to any group. (Note that if you combine these variants to create ambigious situations, it's left undefined which tests will end up running).

**-j THREADS, --jobs=THREADS** Runs up to the given number of tests in parallel. If no number is given, BTest substitutes the number of available CPU cores as reported by the OS.

By default, BTest assumes that all tests can be executed concurrently without further constraints. One can however ensure serialization of subsets by assigning them to the same serialization set, see *@TEST-SERIALIZE*.

**-q, --quiet** Suppress information output other than about failed tests. If all tests pass, there will not be any output at all.

**-r, --rerun** Runs only tests that failed last time. After each execution (except when updating baselines), BTest generates a state file that records the tests that have failed. Using this option on the next run then reads that file back in and limits execution to those tests found in there.

**-t, --tmp-keep** Does not delete any temporary files created for running the tests (including their outputs). By default, the temporary files for a test will be located in `.tmp/<test>/`, where `<test>` is the relative path of the test file with all slashes replaced with dots and the file extension removed (e.g., the files for `example/t3.sh` will be in `.tmp/example.t3`).

**-T, --update-times** Record new *timing* baselines for the current host for tests that have *@TEST-MEASURE-TIME*. Tests are run as normal except that the timing measurements are recorded as the new baseline instead of being compared to a previous baseline.

**-U, --update-baseline** Records a new baseline for all `btest-diff` commands found in any of the specified tests. To do this, all tests are run as normal except that when `btest-diff` is executed, it does not compute a diff but instead considers the given file to be authoritative and records it as the version to compare with in future runs.

**-u, --update-interactive** Each time a `btest-diff` command fails in any tests that are run, btest will stop and ask whether or not the user wants to record a new baseline.

**-v, --verbose** Shows all test command lines as they are executed.

**-w, --wait** Interactively waits for `<enter>` after showing diagnostics for a test.

**-x FILE, --xml=FILE** Records test results in JUnit XML format to the given file. If the file exists already, it is overwritten.

## Configuration

Specifics of `btest`'s execution can be tuned with a configuration file, which by default is `btest.cfg` if that's found in the current directory. It can alternatively be specified with the `--config` command line option, or a `BTEST_CFG` environment variable. The configuration file is "INI-style", and an example comes with the distribution, see `btest.cfg.example`. A configuration file has one main section, `btest`, that defines most options; as well as an optional section for defining *environment variables* and further optional sections for defining *alternatives*.

Note that all paths specified in the configuration file are relative to `btest`'s *base directory*. The base directory is either the one where the configuration file is located if such is given/found, or the current working directory if not. One can also override it explicitly by setting the environment variable `BTEST_TEST_BASE`. When setting values for configuration options, the absolute path to the base directory is available by using the macro `%(testbase)s` (the weird syntax is due to Python's `ConfigParser` class).

Furthermore, all values can use standard "backtick-syntax" to include the output of external commands (e.g., xyz='echo test'). Note that the backtick expansion is performed after any `%(..)` have already been replaced (including within the backticks).

## Options

The following options can be set in the `btest` section of the configuration file:

**`TestDirs`** A space-separated list of directories to search for tests. If defined, one doesn't need to specify any tests on the command line.

**`TmpDir`** A directory where to create temporary files when running tests. By default, this is set to `%(testbase)s/.tmp`.

**`BaselineDir`** A directory where to store the baseline files for `btest-diff` (note that the actual baseline files will be in test-specific subdirectories of this directory). By default, this is set to `%(testbase)s/Baseline`.

**`IgnoreDirs`** A space-separated list of relative directory names to ignore when scanning test directories recursively. Default is empty.

An alternative way to ignore a directory is placing a file `.btest-ignore` in it.

**`IgnoreFiles`** A space-separated list of filename globs matching files to ignore when scanning given test directories recursively. Default is empty.

An alternative way to ignore a file is by placing `@TEST-IGNORE` in it.

**`StateFile`** The name of the state file to record the names of failing tests. Default is `.btest.failed.dat`.

**`Initializer`** An executable that will be executed before each test. It runs in the same directory as the test itself will and receives the name of the test as its parameter. The return value indicates whether the test should continue; if false, the test will be considered failed. By default, there's no initializer set.

**`Finalizer`** An executable that will be executed each time any test has successfully run. It runs in the same directory as the test itself and receives the name of the test as its parameter. The return value indicates whether the test should indeed be considered successful. By default, there's no finalizer set.

**`PartFinalizer`** An executable that will be executed each time a test *part* has successfully run. This operates similarly to `Finalizer` except that it runs after each test part rather than only at completion of the full test. See *parts* for more about test parts.

**`CommandPrefix`** Changes the naming of all `btest` commands by replacing the `@TEST-` prefix with a custom string. For example, with `CommandPrefix=$TEST-`, the `@TEST-EXEC` command becomes `$TEST-EXEC`.

**`TimingBaselineDir`** A directory where to store the host-specific *timing* baseline files. By default, this is set to `%(testbase)s/Baseline/_Timing`.

**`TimingDeltaPerc`** A value defining the *timing* deviation percentage that's tolerated for a test before it's considered failed. Default is 1.0 (which means a 1.0% deviation is tolerated by default).

**`PerfPath`** Specifies a path to the `perf` tool, which is used on Linux to measure the execution times of tests. By default, BTest searches for `perf` in `PATH`.

### Environment Variables

A special section `environment` defines environment variables that will be propagated to all tests:

```
[environment]
CFLAGS=-O3
PATH=%(testbase)s/bin:%(default_path)s
```

Note how `PATH` can be adjusted to include local scripts: the example above prefixes it with a local `bin/` directory inside the base directory, using the predefined `default_path` macro to refer to the `PATH` as it is set by default.

Furthermore, by setting `PATH` to include the `btest` distribution directory, one could skip the installation of the `btest` package.

### Alternatives

BTest can run a set of tests with different settings than it would normally use by specifying an *alternative* configuration. Currently, three things can be adjusted:

- Further environment variables can be set that will then be available to all the commands that a test executes.
- *Filters* can modify an input file before a test uses it.
- *Substitutions* can modify command lines executed as part of a test.

We discuss the three separately in the following. All of them are defined by adding sections `[<type>-<name>]` where `<type>` corresponds to the type of adjustment being made and `<name>` is the name of the alternative. Once at least one section is defined for a name, that alternative can be enabled by BTest's `--alternative` flag.

### Environment Variables

An alternative can add further environment variables by defining an `[environment-<name>]` section:

```
[environment-myalternative]
CFLAGS=-O3
```

Running `btest` with `--alternative=myalternative` will now make the `CFLAGS` environment variable available to all commands executed.

As a special case, one can override `BTEST_TEST_BASE` inside an alternative's environment section, and it will not only be passed on to child processes, but also apply to the `btest` process itself. That way, one can switch to a different test base directory for an alternative.

### Filters

Filters are a transparent way to adapt the input to a specific test command before it is executed. A filter is defined by adding a section `[filter-<name>]` to the configuration file. This section must have exactly one entry, and the name of that entry is interpreted as the name of a command whose input is to be filtered. The value of that entry is the name of a filter script that will be run with two arguments representing input and output files, respectively. Example:

```
[filter-myalternative]
cat=%(testbase)s/bin/filter-cat
```

Once the filter is activated by running `btest` with `--alternative=myalternative`, every time a `@TEST-EXEC: cat %INPUT` is found, `btest` will first execute (something similar to) `%(testbase)s/bin/filter-cat %INPUT out.tmp`, and then subsequently `cat out.tmp` (i.e., the original command but with the filtered output). In the simplest case, the filter could be a no-op in the form `cp $1 $2`.

**Note:** There are a few limitations to the filter concept currently:

- Filters are *always* fed with `%INPUT` as their first argument. We should add a way to filter other files as well.
- Filtered commands are only recognized if they are directly starting the command line. For example, `@TEST-EXEC: ls | cat >outout` would not trigger the example filter above.
- Filters are only executed for `@TEST-EXEC`, not for `@TEST-EXEC-FAIL`.

### Substitutions

Substitutions are similar to filters, yet they do not adapt the input but the command line being executed. A substitution is defined by adding a section `[substitution-<name>]` to the configuration file. For each entry in this section, the entry's name specifies the command that is to be replaced with something else given as its value. Example:

```
[substitution-myalternative]
gcc=gcc -O2
```

Once the substitution is activated by running `btest` with `--alternative=myalternative`, every time a `@TEST-EXEC` executes `gcc`, that is replaced with `gcc -O2`. The replacement is simple string substitution so it works not only with commands but anything found on the command line; it however only replaces full words, not subparts of words.

### Writing Tests

`btest` scans a test file for lines containing keywords that trigger certain functionality. Currently, the following keywords are supported:

**`@TEST-EXEC: <cmdline>`** Executes the given command line and aborts the test if it returns an error code other than zero. The `<cmdline>` is passed to the shell and thus can be a pipeline, use redirection, and any environment variables specified in `<cmdline>` will be expanded, etc.

When running a test, the current working directory for all command lines will be set to a temporary sandbox (and will be deleted later).

There are two macros that can be used in `<cmdline>`: `%INPUT` will be replaced with the full pathname of the file defining the test (this file is in a temporary sandbox directory and is a copy of the original test file); and `%DIR` will be replaced with the full pathname of the directory where the test file is located (note that this is the directory where the original test file is located, not the directory where the `%INPUT` file is located). The latter can be used to reference further files also located there.

In addition to environment variables defined in the configuration file, there are further ones that are passed into the commands:

**`TEST_DIAGNOSTICS`** A file where further diagnostic information can be saved in case a command fails (this is also where `btest-diff` stores its diff). If this file exists, then the `--diagnostics-all` or `--diagnostics` options will show this file (for the latter option, only if a command fails).

**TEST_MODE** This is normally set to `TEST`, but will be `UPDATE` if `btest` is run with `--update-baseline`, or `UPDATE_INTERACTIVE` if run with `--update-interactive`.

**TEST_BASELINE** The name of a directory where the command can save permanent information across `btest` runs. (This is where `btest-diff` stores its baseline in `UPDATE` mode.)

**TEST_NAME** The name of the currently executing test.

**TEST_VERBOSE** The path of a file where the test can record further information about its execution that will be included with btest's `--verbose` output. This is for further tracking the execution of commands and should generally generate output that follows a line-based structure.

**TEST_BASE** The btest base directory, i.e., the directory where `btest.cfg` is located.

**TEST_PART** The test part number (see *parts* for more about test parts).

**Note:** If a command returns the special exit code 100, the test is considered failed, however subsequent test commands within the current test are still run. `btest-diff` uses this special exit code to indicate that no baseline has yet been established.

If a command returns the special exit code 200, the test is considered failed and all further tests are aborted. `btest-diff` uses this special exit code when btest is run with the `--update-interactive` option and the user chooses to abort the tests when prompted to record a new baseline.

**@TEST-EXEC-FAIL: <cmdline>** Like `@TEST-EXEC`, except that this expects the command to *fail*, i.e., the test is aborted when the return code is zero.

**@TEST-REQUIRES: <cmdline>** Defines a condition that must be met for the test to be executed. The given command line will be run before any of the actual test commands, and it must return success for the test to continue. If it does not return success, the rest of the test will be skipped but doing so will not be considered a failure of the test. This allows to write conditional tests that may not always make sense to run, depending on whether external constraints are satisfied or not (say, whether a particular library is available). Multiple requirements may be specified and then all must be met for the test to continue.

**@TEST-ALTERNATIVE: <alternative>** Runs this test only for the given alternative (see *alternative*). If `<alternative>` is `default`, the test executes when BTest runs with no alternative given (which however is the default anyway).

**@TEST-NOT-ALTERNATIVE: <alternative>** Ignores this test for the given alternative (see *alternative*). If `<alternative>` is `default`, the test is ignored if BTest runs with no alternative given.

**@TEST-COPY-FILE: <file>** Copy the given file into the test's directory before the test is run. If `<file>` is a relative path, it's interpreted relative to the BTest's base directory. Environment variables in `<file>` will be replaced if enclosed in `${..}`. This command can be given multiple times.

**@TEST-START-NEXT** This is a short-cut for defining multiple test inputs in the same file, all executing with the same command lines. When `@TEST-START-NEXT` is encountered, the test file is initially considered to end at that point, and all `@TEST-EXEC-*` are run with an `%INPUT` truncated accordingly. Afterwards, a *new* `%INPUT` is created with everything *following* the `@TEST-START-NEXT` marker, and the *same* commands are run again (further `@TEST-EXEC-*` will be ignored). The effect is that a single file can actually define two tests, and the `btest` output will enumerate them:

```
> cat examples/t5.sh
# @TEST-EXEC: cat %INPUT | wc -c >output
# @TEST-EXEC: btest-diff output

This is the first test input in this file.
```

```
# @TEST-START-NEXT

... and the second.

> ./btest -D examples/t5.sh
examples.t5 ... ok
  % cat .diag
  == File ===============================
  119
  [...]

examples.t5-2 ... ok
  % cat .diag
  == File ===============================
  22
  [...]
```

Multiple `@TEST-START-NEXT` can be used to create more than two tests per file.

**`@TEST-START-FILE <file>`** This is used to include an additional input file for a test right inside the test file. All lines following the keyword line will be written into the given file until a line containing `@TEST-END-FILE` is found. The lines containing `@TEST-START-FILE` and `@TEST-END-FILE`, and all lines in between, will be removed from the test's %INPUT. Example:

```
> cat examples/t6.sh
# @TEST-EXEC: awk -f %INPUT <foo.dat >output
# @TEST-EXEC: btest-diff output

    { lines += 1; }
END { print lines; }

@TEST-START-FILE foo.dat
1
2
3
@TEST-END-FILE

> btest -D examples/t6.sh
examples.t6 ... ok
  % cat .diag
  == File ===============================
  3
```

Multiple such files can be defined within a single test.

Note that this is only one way to use further input files. Another is to store a file in the same directory as the test itself, making sure it's ignored via `IgnoreFiles`, and then refer to it via `%DIR/<name>`.

**`@TEST-IGNORE`** This is used to indicate that this file should be skipped (i.e., no test commands in this file will be executed). An alternative way to ignore files is by using the `IgnoreFiles` option in the btest configuration file.

**`@TEST-GROUP: <group>`** Assigns the test to a group of name `<group>`. By using option `-g` one can limit execution to all tests that belong to a given group (or a set of groups).

**`@TEST-SERIALIZE: <set>`** When using option `-j` to parallelize execution, all tests that specify the same serialization set are guaranteed to run sequentially. `<set>` is an arbitrary user-chosen string.

**`@TEST-KNOWN-FAILURE`** Marks a test as known to currently fail. This only changes BTest's output, which upon failure will indicate that that is expected; it won't change the test's processing otherwise. The keyword doesn't take any arguments but one could add a descriptive text, as in

```
.. @TEST-KNOWN-FAILURE: We know this fails because ....
```

**@TEST-MEASURE-TIME** Measures execution time for this test and compares it to a previously established *timing* baseline. If it deviates significantly, the test will be considered failed.

### Splitting Tests into Parts

One can split a single test across multiple files by adding a numerical `#<n>` postfix to their names, where each `<n>` represents a separate part of the test. `btest` will combine all of a test's parts in numerical order and execute them subsequently within the same sandbox. Example:

```
> cat examples/t7.sh#1
# @TEST-EXEC: echo Part 1 - %INPUT >>output

> cat examples/t7.sh#2
# @TEST-EXEC: echo Part 2 - %INPUT >>output

> cat examples/t7.sh#3
# @TEST-EXEC: btest-diff output

> btest -D examples/t7.sh
examples.t7 ... ok
% cat .diag
== File ===============================
Part 1 - /Users/robin/bro/docs/aux/btest/.tmp/examples.t7/t7.sh#1
Part 2 - /Users/robin/bro/docs/aux/btest/.tmp/examples.t7/t7.sh#2
```

Note how `output` contains the output of both `t7.sh#1` and `t7.sh#2`, however in each case `%INPUT` refers to the corresponding part. For the first part of a test, one can also omit the `#1` postfix in the filename.

### Canonifying Diffs

`btest-diff` has the capability to filter its input through an additional script before it compares the current version with the baseline. This can be useful if certain elements in an output are *expected* to change (e.g., timestamps). The filter can then remove/replace these with something consistent. To enable such canonification, set the environment variable `TEST_DIFF_CANONIFIER` to a script reading the original version from stdin and writing the canonified version to stdout. Note that both baseline and current output are passed through the filter before their differences are computed.

### Running Processes in the Background

Sometimes processes need to be spawned in the background for a test, in particular if multiple processes need to cooperate in some fashion. `btest` comes with two helper scripts to make life easier in such a situation:

**btest-bg-run <tag> <cmdline>** This is a script that runs `<cmdline>` in the background, i.e., it's like using `cmdline &` in a shell script. Test execution continues immediately with the next command. Note that the spawned command is *not* run in the current directory, but instead in a newly created sub-directory called `<tag>`. This allows spawning multiple instances of the same process without needing to worry about conflicting outputs. If you want to access a command's output later, like with `btest-diff`, use `<tag>/foo.log` to access it.

**btest-bg-wait [-k] <timeout>** This script waits for all processes previously spawned via `btest-bg-run` to finish. If any of them exits with a non-zero return code, `btest-bg-wait` does so as well, indicating a failed test. `<timeout>` is mandatory and gives the maximum number of seconds to wait for any of the processes to terminate. If any process hasn't done so when the timeout expires, it will be killed and the test is considered to be failed as long as `-k` is not given. If `-k` is given, pending processes are still killed but the test continues normally, i.e., non-termination is not considered a failure in this case. This script also collects the processes' stdout and stderr outputs for diagnostics output.

### Timing Execution

`btest` can time execution of tests and report significant deviations from past runs. As execution time is inherently system-specific it keeps separate per-host timing baselines for that. Furthermore, as time measurements tend to make sense only for individual, usually longer running tests, they are activated on per test basis by adding a *@TEST-MEASURE-TIME* directive. The test will then execute as usual yet also record the duration for which it executes. After the timing baselines are created (with the `--update-times` option), further runs on the same host will compare their times against that baseline and declare a test failed if it deviates by more than, by default, 1%. (To tune the behaviour, look at the `Timing*` *options*.) If a test requests measurement but BTest can't find a timing baseline or the necessary tools to perform timing measurements, then it will ignore the request.

As timing for a test can deviate quite a bit even on the same host, BTest does not actually measure *time* but the number of CPU instructions that a test executes, which tends to be more stable. That however requires the right tools to be in place. On Linux, BTest leverages perf. By default, BTest will search for `perf` in the `PATH`; you can specify a different path to the binary by setting `PerfPath` in `btest.cfg`.

### Integration with Sphinx

`btest` comes with an extension module for the documentation framework Sphinx. The extension module provides two new directives called `btest` and `btest-include`. The `btest` directive allows writing a test directly inside a Sphinx document, and then the output from the test's command is included in the generated documentation. The `btest-include` directive allows for literal text from another file to be included in the generated documentation. The tests from both directives can also be run externally and will catch if any changes to the included content occur. The following walks through setting this up.

### Configuration

First, you need to tell Sphinx a base directory for the `btest` configuration as well as a directory in there where to store tests it extracts from the Sphinx documentation. Typically, you'd just create a new subdirectory `tests` in the Sphinx project for the `btest` setup and then store the tests in there in, e.g., `doc/`:

```
cd <sphinx-root>
mkdir tests
mkdir tests/doc
```

Then add the following to your Sphinx `conf.py`:

```
extensions += ["btest-sphinx"]
btest_base="tests"         # Relative to Sphinx-root.
btest_tests="doc"          # Relative to btest_base.
```

Next, create a `btest.cfg` in `tests/` as usual and add `doc/` to the `TestDirs` option. Also, add a finalizer to `btest.cfg`:

```
[btest]
...
PartFinalizer=btest-diff-rst
```

### Including a Test into a Sphinx Document

The btest extension provides a new directive to include a test inside a Sphinx document:

```
.. btest:: <test-name>

    <test content>
```

Here, <test-name> is a custom name for the test; it will be stored in btest_tests under that name (with a file extension of .btest). <test content> is just a standard test as you would normally put into one of the TestDirs. Example:

```
.. btest:: just-a-test

    @TEST-EXEC: expr 2 + 2
```

When you now run Sphinx, it will (1) store the test content into tests/doc/just-a-test.btest (assuming the above path layout), and (2) execute the test by running btest on it. You can then run btest manually in tests/ as well and it will execute the test just as it would in a standard setup. If a test fails when Sphinx runs it, there will be a corresponding error and include the diagnostic output into the document.

By default, nothing else will be included into the generated documentation, i.e., the above test will just turn into an empty text block. However, btest comes with a set of scripts that you can use to specify content to be included. As a simple example, btest-rst-cmd <cmdline> will execute a command and (if it succeeds) include both the command line and the standard output into the documentation. Example:

```
.. btest:: another-test

    @TEST-EXEC: btest-rst-cmd echo Hello, world!
```

When running Sphinx, this will render as:

```
# echo Hello, world!
Hello, world!
```

The same <test-name> can be used multiple times, in which case each entry will become one part of a joint test. btest will execute all parts subsequently within a single sandbox, and earlier results will thus be available to later parts.

When running btest manually in tests/, the PartFinalizer we added to btest.cfg (see above) compares the generated reST code with a previously established baseline, just like btest-diff does with files. To establish the initial baseline, run btest -u, like you would with btest-diff.

### Scripts

The following Sphinx support scripts come with btest:

`btest-rst-cmd [options] <cmdline>`

By default, this executes <cmdline> and includes both the command line itself and its standard output into the generated documentation (but only if the command line succeeds). See above for an example.

This script provides the following options:

| | |
|---|---|
| **-c ALTERNATIVE_CMDLINE** | Show `ALTERNATIVE_CMDLINE` in the generated documentation instead of the one actually executed. (It still runs the `<cmdline>` given outside the option.) |
| **-d** | Do not actually execute `<cmdline>`; just format it for the generated documentation and include no further output. |
| **-f FILTER_CMD** | Pipe the command line's output through `FILTER_CMD` before including. If `-r` is given, it filters the file's content instead of stdout. |
| **-o** | Do not include the executed command into the generated documentation, just its output. |
| **-r FILE** | Insert `FILE` into output instead of stdout. The `FILE` must be created by a previous `@TEST-EXEC` or `@TEST-COPY-FILE`. |
| **-n N** | Include only `N` lines of output, adding a `[...]` marker if there's more. |

`btest-rst-include [options] <file>`

Includes `<file>` inside a code block. The `<file>` must be created by a previous `@TEST-EXEC` or `@TEST-COPY-FILE`.

This script provides the following options:

| | |
|---|---|
| **-n N** | Include only `N` lines of output, adding a `[...]` marker if there's more. |

`btest-rst-pipe <cmdline>`

Executes `<cmdline>`, includes its standard output inside a code block (but only if the command line succeeds). Note that this script does not include the command line itself into the code block, just the output.

**Note:** All these scripts can be run directly from the command line to show the reST code they generate.

**Note:** `btest-rst-cmd` can do everything the other scripts provide if you give it the right options. In fact, the other scripts are provided just for convenience and leverage `btest-rst-cmd` internally.

### Including Literal Text

The `btest` Sphinx extension module also provides a directive `btest-include` that functions like `literalinclude` (including all its options) but also creates a test checking the included content for changes. As one further extension, the directive expands environment variables of the form `${var}` in its argument. Example:

```
.. btest-include:: ${var}/path/to/file
```

When you now run Sphinx, it will automatically generate a test file in the directory specified by the `btest_tests` variable in the Sphinx `conf.py` configuration file. In this example, the filename would be `include-path_to_file.btest` (it automatically adds a prefix of "include-" and a file extension of ".btest"). When you run the tests externally, the tests generated by the `btest-include` directive will check if any of the included content has changed (you'll first need to run `btest -u` to establish the initial baseline).

#### License

btest is open-source under a BSD licence.

### 3.3.11 capstats - A tool to get some NIC statistics.

capstats is a small tool to collect statistics on the current load of a network interface, using either libpcap or the native interface for Endace's. It reports statistics per time interval and/or for the tool's total run-time.

#### Download

You can find the latest capstats release for download at http://www.bro.org/download.

Capstats's git repository is located at git://git.bro.org/capstats.git. You can browse the repository here.

This document describes capstats 0.22. See the `CHANGES` file for version history.

#### Output

Here's an example output with output in one-second intervals until `CTRL-C` is hit:

Each line starts with a timestamp and the other fields are:

**pkts** Absolute number of packets seen by `capstats` during interval.

**kpps** Number of thousands of packets per second.

**kbytes** Absolute number of KBytes during interval.

**mbps** Mbits/sec.

**nic_pkts** Number of packets as reported by `libpcap`'s `pcap_stats()` (may not match **pkts**)

**nic_drops** Number of packet drops as reported by `libpcap`'s `pcap_stats()`.

**u** Number of UDP packets.

**t** Number of TCP packets.

**i** Number of ICMP packets.

**o** Number of IP packets with protocol other than TCP, UDP, and ICMP.

**nonip** Number of non-IP packets.

#### Options

A list of all options:

```
capstats [Options] -i interface

   -i| --interface <interface>    Listen on interface
   -d| --dag                      Use native DAG API
   -f| --filter <filter>          BPF filter
   -I| --interval <secs>          Stats logging interval
   -l| --syslog                   Use syslog rather than print to stderr
   -n| --number <count>           Stop after outputting <number> intervals
   -N| --select                   Use select() for live pcap (for testing only)
   -p| --payload <n>              Verifies that packets' payloads consist
                                  entirely of bytes of the given value.
   -q| --quiet <count>            Suppress output, exit code indicates >= count
                                  packets received.
   -S| --size <size>              Verify packets to have given <size>
   -s| --snaplen <size>           Use pcap snaplen <size>
   -v| --version                  Print version and exit
   -w| --write <filename>         Write packets to file
```

### Installation

`capstats` has been tested on Linux, FreeBSD, and MacOS. Please see the `INSTALL` file for installation instructions.

## 3.3.12 PySubnetTree - A Python Module for CIDR Lookups

The PySubnetTree package provides a Python data structure `SubnetTree` which maps subnets given in CIDR notation (incl. corresponding IPv6 versions) to Python objects. Lookups are performed by longest-prefix matching.

### Download

You can find the latest PySubnetTree release for download at http://www.bro.org/download.

PySubnetTree's git repository is located at git://git.bro.org/pysubnettree.git. You can browse the repository here.

This document describes PySubnetTree 0.24-7. See the `CHANGES` file for version history.

### Example

A simple example which associates CIDR prefixes with strings:

```
>>> import SubnetTree
>>> t = SubnetTree.SubnetTree()
>>> t["10.1.0.0/16"] = "Network 1"
>>> t["10.1.42.0/24"] = "Network 1, Subnet 42"
>>> print("10.1.42.1" in t)
True
>>> print(t["10.1.42.1"])
Network 1, Subnet 42
>>> print(t["10.1.43.1"])
Network 1
>>> print("10.20.1.1" in t)
False
>>> try:
...     print(t["10.20.1.1"])
... except KeyError as err:
```

```
...     print("Error: %s not found" % err)
Error: '10.20.1.1' not found
```

PySubnetTree also supports IPv6 addresses and prefixes:

```
>>> import SubnetTree
>>> t = SubnetTree.SubnetTree()
>>> t["2001:db8::/32"] = "Company 1"
>>> t["2001:db8:4000::/48"] = "Company 1, Site 1"
>>> t["2001:db8:4000:abcd::"]
Company 1, Site 1
>>> t["2001:db8:fe:1234::"]
Company 1
```

By default, CIDR prefixes and IP addresses are given as strings. Alternatively, a `SubnetTree` object can be switched into *binary mode*, in which single addresses are passed in the form of packed binary strings as, e.g., returned by socket.inet_aton:

```
>>> t.get_binary_lookup_mode()
False
>>> t.set_binary_lookup_mode(True)
>>> t.get_binary_lookup_mode()
True
>>> import socket
>>> print(t[socket.inet_aton("10.1.42.1")])
Network 1, Subnet 42
```

A SubnetTree also provides methods `insert(prefix,object=None)` for insertion of prefixes (`object` can be skipped to use the tree like a set), and `remove(prefix)` for removing entries (`remove` performs an _exact_ match rather than longest-prefix).

Internally, the CIDR prefixes of a `SubnetTree` are managed by a Patricia tree data structure and lookups are therefore efficient even with a large number of prefixes.

PySubnetTree comes with a BSD license.

#### Prerequisites

This package requires Python 2.4 or newer.

#### Installation

Installation is pretty simple:

```
> python setup.py install
```

### 3.3.13 trace-summary - Generating network traffic summaries

`trace-summary` is a Python script that generates break-downs of network traffic, including lists of the top hosts, protocols, ports, etc. Optionally, it can generate output separately for incoming vs. outgoing traffic, per subnet, and per time-interval.

### Download

You can find the latest trace-summary release for download at http://www.bro.org/download.

trace-summary's git repository is located at git://git.bro.org/trace-summary.git. You can browse the repository here.

This document describes trace-summary 0.84-16. See the `CHANGES` file for version history.

### Overview

The `trace-summary` script reads both packet traces in libpcap format and connection logs produced by the Bro network intrusion detection system (for the latter, it supports both 1.x and 2.x output formats).

Here are two example outputs in the most basic form (note that IP addresses are 'anonymized'). The first is from a packet trace and the second from a Bro connection log:

```
>== Total === 2005-01-06-14-23-33 - 2005-01-06-15-23-43
  - Bytes 918.3m - Payload 846.3m - Pkts 1.8m - Frags   0.9% - MBit/s      1.9 -
    Ports        | Sources                   | Destinations              | Protocols |
    80     33.8% | 131.243.89.214       8.5% | 131.243.89.214       7.7% | 6   76.0% |
    22     16.7% | 128.3.2.102          6.2% | 128.3.2.102          5.4% | 17  23.3% |
    11001  12.4% | 204.116.120.26       4.8% | 131.243.89.4         4.8% | 1    0.5% |
    2049   10.7% | 128.3.161.32         3.6% | 131.243.88.227       3.6% |           |
    1023   10.6% | 131.243.89.4         3.5% | 204.116.120.26       3.4% |           |
    993     8.2% | 128.3.164.194        2.7% | 131.243.89.64        3.1% |           |
    1049    8.1% | 128.3.164.15         2.4% | 128.3.164.229        2.9% |           |
    524     6.6% | 128.55.82.146        2.4% | 131.243.89.155       2.5% |           |
    33305   4.5% | 131.243.88.227       2.3% | 128.3.161.32         2.3% |           |
    1085    3.7% | 131.243.89.155       2.3% | 128.55.82.146        2.1% |           |


>== Total === 2005-01-06-14-23-33 - 2005-01-06-15-23-42
  - Connections 43.4k - Payload 398.4m -
    Ports        | Sources                   | Destinations              | Services   ␣
↪        | Protocols | States        |
    80     21.7% | 207.240.215.71       3.0% | 239.255.255.253      8.0% | other      ␣
↪  51.0% | 17  55.8% | S0      46.2% |
    427    13.0% | 131.243.91.71        2.2% | 131.243.91.255       4.0% | http       ␣
↪  21.7% | 6   36.4% | SF      30.1% |
    443     3.8% | 128.3.161.76         1.7% | 131.243.89.138       2.1% | i-echo     ␣
↪   7.3% | 1    7.7% | OTH      7.8% |
    138     3.7% | 131.243.90.138       1.6% | 255.255.255.255      1.7% | https      ␣
↪   3.8% |           | RSTO     5.8% |
    515     2.4% | 131.243.88.159       1.6% | 128.3.97.204         1.5% | nb-dgm     ␣
↪   3.7% |           | SHR      4.4% |
    11001   2.3% | 131.243.88.202       1.4% | 131.243.88.107       1.1% | printer    ␣
↪   2.4% |           | REJ      3.0% |
    53      1.9% | 131.243.89.250       1.4% | 117.72.94.10         1.1% | dns        ␣
↪   1.9% |           | S1       1.0% |
    161     1.6% | 131.243.89.80        1.3% | 131.243.88.64        1.1% | snmp       ␣
↪   1.6% |           | RSTR     0.9% |
    137     1.4% | 131.243.90.52        1.3% | 131.243.88.159       1.1% | nb-ns      ␣
↪   1.4% |           | SH       0.3% |
    2222    1.1% | 128.3.161.252        1.2% | 131.243.91.92        1.1% | ntp        ␣
↪   1.0% |           | RSTRH    0.2% |
```

### Prerequisites

- This script requires Python 2.6 or newer.
- The pysubnettree Python module.
- Eddie Kohler's ipsumdump if using `trace-summary` with packet traces (versus Bro connection logs)

### Installation

Simply copy the script into some directory which is in your `PATH`.

### Usage

The general usage is:

```
trace-summary [options] [input-file]
```

Per default, it assumes the `input-file` to be a `libpcap` trace file. If it is a Bro connection log, use `-c`. If `input-file` is not given, the script reads from stdin. It writes its output to stdout.

### Options

The most important options are summarized below. Run `trace-summary --help` to see the full list including some more esoteric ones.

**-c** Input is a Bro connection log instead of a `libpcap` trace file.

**-b** Counts all percentages in bytes rather than number of packets/connections.

**-E <file>** Gives a file which contains a list of networks to ignore for the analysis. The file must contain one network per line, where each network is of the CIDR form `a.b.c.d/mask` (including the corresponding syntax for IPv6 prefixes, e.g., `1:2:3:4::/64`). Empty lines and lines starting with a "#" are ignored.

**-i <duration>** Creates totals for each time interval of the given length (default is seconds; add "`m`" for minutes and "`h`" for hours). Use `-v` if you also want to see the breakdowns for each interval.

**-l <file>** Generates separate summaries for incoming and outgoing traffic. `<file>` is a file which contains a list of networks to be considered local. Format as for `-E`.

**-n <n>** Show top n entries in each break-down. Default is 10.

**-r** Resolves hostnames in the output.

**-s <n>** Gives the sample factor if the input has been sampled.

**-S <n>** Sample input with the given factor; less accurate but faster and saves memory.

**-m** Does skip memory-expensive statistics.

**-v** Generates full break-downs for each time interval. Requires `-i`.

The Broccoli API Reference may also be of interest.

CHAPTER

FOUR

# DEVELOPMENT

## 4.1 Writing Bro Plugins

Bro internally provides a plugin API that enables extending the system dynamically, without modifying the core code base. That way custom code remains self-contained and can be maintained, compiled, and installed independently. Currently, plugins can add the following functionality to Bro:

- Bro scripts.
- Builtin functions/events/types for the scripting language.
- Protocol analyzers.
- File analyzers.
- Packet sources and packet dumpers.
- Logging framework backends.
- Input framework readers.

A plugin's functionality is available to the user just as if Bro had the corresponding code built-in. Indeed, internally many of Bro's pieces are structured as plugins as well, they are just statically compiled into the binary rather than loaded dynamically at runtime.

### 4.1.1 Quick Start

Writing a basic plugin is quite straight-forward as long as one follows a few conventions. In the following we create a simple example plugin that adds a new built-in function (bif) to Bro: we'll add `rot13(s: string) : string`, a function that rotates every character in a string by 13 places.

Generally, a plugin comes in the form of a directory following a certain structure. To get started, Bro's distribution provides a helper script `aux/bro-aux/plugin-support/init-plugin` that creates a skeleton plugin that can then be customized. Let's use that:

```
# init-plugin ./rot13-plugin Demo Rot13
```

As you can see, the script takes three arguments. The first is a directory inside which the plugin skeleton will be created. The second is the namespace the plugin will live in, and the third is a descriptive name for the plugin itself relative to the namespace. Bro uses the combination of namespace and name to identify a plugin. The namespace serves to avoid naming conflicts between plugins written by independent developers; pick, e.g., the name of your organisation. The namespace `Bro` is reserved for functionality distributed by the Bro Project. In our example, the plugin will be called `Demo::Rot13`.

The `init-plugin` script puts a number of files in place. The full layout is described later. For now, all we need is `src/rot13.bif`. It's initially empty, but we'll add our new bif there as follows:

```
# cat src/rot13.bif
module Demo;

function rot13%(s: string%) : string
    %{
    char* rot13 = copy_string(s->CheckString());

    for ( char* p = rot13; *p; p++ )
        {
        char b = islower(*p) ? 'a' : 'A';
        *p  = (*p - b + 13) % 26 + b;
        }

    BroString* bs = new BroString(1, reinterpret_cast<byte_vec>(rot13),
                                  strlen(rot13));
    return new StringVal(bs);
    %}
```

The syntax of this file is just like any other `*.bif` file; we won't go into it here.

Now we can already compile our plugin, we just need to tell the configure script (that `init-plugin` created) where the Bro source tree is located (Bro needs to have been built there first):

```
# cd rot13-plugin
# ./configure --bro-dist=/path/to/bro/dist && make
[... cmake output ...]
```

This builds the plugin in a subdirectory `build/`. In fact, that subdirectory *becomes* the plugin: when `make` finishes, `build/` has everything it needs for Bro to recognize it as a dynamic plugin.

Let's try that. Once we point Bro to the `build/` directory, it will pull in our new plugin automatically, as we can check with the `-N` option:

```
# export BRO_PLUGIN_PATH=/path/to/rot13-plugin/build
# bro -N
[...]
Demo::Rot13 - <Insert description> (dynamic, version 0.1)
[...]
```

That looks quite good, except for the dummy description that we should replace with something nicer so that users will know what our plugin is about. We do this by editing the `config.description` line in `src/Plugin.cc`, like this:

```
[...]
plugin::Configuration Plugin::Configure()
    {
    plugin::Configuration config;
    config.name = "Demo::Rot13";
    config.description = "Caesar cipher rotating a string's characters by 13 places.";
    config.version.major = 0;
    config.version.minor = 1;
    return config;
    }
[...]
```

Now rebuild and verify that the description is visible:

```
# make
[...]
# bro -N | grep Rot13
Demo::Rot13 - Caesar cipher rotating a string's characters by 13 places. (dynamic,␣
↪version 0.1)
```

Bro can also show us what exactly the plugin provides with the more verbose option `-NN`:

```
# bro -NN
[...]
Demo::Rot13 - Caesar cipher rotating a string's characters by 13 places. (dynamic,␣
↪version 0.1)
    [Function] Demo::rot13
[...]
```

There's our function. Now let's use it:

```
# bro -e 'print Demo::rot13("Hello")'
Uryyb
```

It works. We next install the plugin along with Bro itself, so that it will find it directly without needing the `BRO_PLUGIN_PATH` environment variable. If we first unset the variable, the function will no longer be available:

```
# unset BRO_PLUGIN_PATH
# bro -e 'print Demo::rot13("Hello")'
error in <command line>, line 1: unknown identifier Demo::rot13, at or near "Demo::
↪rot13"
```

Once we install it, it works again:

```
# make install
# bro -e 'print Demo::rot13("Hello")'
Uryyb
```

The installed version went into `<bro-install-prefix>/lib/bro/plugins/Demo_Rot13`.

One can distribute the plugin independently of Bro for others to use. To distribute in source form, just remove the `build/` directory (`make distclean` does that) and then tar up the whole `rot13-plugin/` directory. Others then follow the same process as above after unpacking.

To distribute the plugin in binary form, the build process conveniently creates a corresponding tarball in `build/dist/`. In this case, it's called `Demo_Rot13-0.1.tar.gz`, with the version number coming out of the `VERSION` file that `init-plugin` put into place. The binary tarball has everything needed to run the plugin, but no further source files. Optionally, one can include further files by specifying them in the plugin's `CMakeLists.txt` through the `bro_plugin_dist_files` macro; the skeleton does that for `README`, `VERSION`, `CHANGES`, and `COPYING`. To use the plugin through the binary tarball, just unpack it into `<bro-install-prefix>/lib/bro/plugins/`. Alternatively, if you unpack it in another location, then you need to point `BRO_PLUGIN_PATH` there.

Before distributing your plugin, you should edit some of the meta files that `init-plugin` puts in place. Edit `README` and `VERSION`, and update `CHANGES` when you make changes. Also put a license file in place as `COPYING`; if BSD is fine, you will find a template in `COPYING.edit-me`.

### 4.1.2 Plugin Directory Layout

A plugin's directory needs to follow a set of conventions so that Bro (1) recognizes it as a plugin, and (2) knows what to load. While `init-plugin` takes care of most of this, the following is the full story. We'll use `<base>` to represent a plugin's top-level directory. With the skeleton, `<base>` corresponds to `build/`.

**`<base>/__bro_plugin__`** A file that marks a directory as containing a Bro plugin. The file must exist, and its content must consist of a single line with the qualified name of the plugin (e.g., "Demo::Rot13").

**`<base>/lib/<plugin-name>.<os>-<arch>.so`** The shared library containing the plugin's compiled code. Bro will load this in dynamically at run-time if OS and architecture match the current platform.

**`scripts/`** A directory with the plugin's custom Bro scripts. When the plugin gets activated, this directory will be automatically added to `BROPATH`, so that any scripts/modules inside can be "@load"ed.

**`scripts/__load__.bro`** A Bro script that will be loaded when the plugin gets activated. When this script executes, any BiF elements that the plugin defines will already be available. See below for more information on activating plugins.

**`scripts/__preload__.bro`** A Bro script that will be loaded when the plugin gets activated, but before any BiF elements become available. See below for more information on activating plugins.

**`lib/bif/`** Directory with auto-generated Bro scripts that declare the plugin's bif elements. The files here are produced by `bifcl`.

Any other files in `<base>` are ignored by Bro.

By convention, a plugin should put its custom scripts into sub folders of `scripts/`, i.e., `scripts/<plugin-namespace>/<plugin-name>/<script>.bro` to avoid conflicts. As usual, you can then put a `__load__.bro` in there as well so that, e.g., `@load Demo/Rot13` could load a whole module in the form of multiple individual scripts.

Note that in addition to the paths above, the `init-plugin` helper puts some more files and directories in place that help with development and installation (e.g., `CMakeLists.txt`, `Makefile`, and source code in `src/`). However, all these do not have a special meaning for Bro at runtime and aren't necessary for a plugin to function.

### 4.1.3 `init-plugin`

`init-plugin` puts a basic plugin structure in place that follows the above layout and augments it with a CMake build and installation system. Plugins with this structure can be used both directly out of their source directory (after `make` and setting Bro's `BRO_PLUGIN_PATH`), and when installed alongside Bro (after `make install`).

`make install` copies over the `lib` and `scripts` directories, as well as the `__bro_plugin__` magic file and any further distribution files specified in `CMakeLists.txt` (e.g., README, VERSION). You can find a full list of files installed in `build/MANIFEST`. Behind the scenes, `make install` really just unpacks the binary tarball from `build/dist` into the destination directory.

`init-plugin` will never overwrite existing files. If its target directory already exists, it will by default decline to do anything. You can run it with `-u` instead to update an existing plugin, however it will never overwrite any existing files; it will only put in place files it doesn't find yet. To revert a file back to what `init-plugin` created originally, delete it first and then rerun with `-u`.

`init-plugin` puts a `configure` script in place that wraps `cmake` with a more familiar configure-style configuration. By default, the script provides two options for specifying paths to the Bro source (`--bro-dist`) and to the plugin's installation directory (`--install-root`). To extend `configure` with plugin-specific options (such as search paths for its dependencies) don't edit the script directly but instead extend `configure.plugin`, which `configure` includes. That way you will be able to more easily update `configure` in the future when the distribution version changes. In `configure.plugin` you can use the predefined shell function `append_cache_entry` to seed values into the CMake cache; see the installed skeleton version and existing plugins for examples.

### 4.1.4 Activating a Plugin

A plugin needs to be *activated* to make it available to the user. Activating a plugin will:

1. Load the dynamic module
2. Make any bif items available
3. Add the `scripts/` directory to `BROPATH`
4. Load `scripts/__preload__.bro`
5. Make BiF elements available to scripts.
6. Load `scripts/__load__.bro`

By default, Bro will automatically activate all dynamic plugins found in its search path `BRO_PLUGIN_PATH`. However, in bare mode (`bro -b`), no dynamic plugins will be activated by default; instead the user can selectively enable individual plugins in scriptland using the `@load-plugin <qualified-plugin-name>` directive (e.g., `@load-plugin Demo::Rot13`). Alternatively, one can activate a plugin from the command-line by specifying its full name (`Demo::Rot13`), or set the environment variable `BRO_PLUGIN_ACTIVATE` to a list of comma(!)-separated names of plugins to unconditionally activate, even in bare mode.

`bro -N` shows activated plugins separately from found but not yet activated plugins. Note that plugins compiled statically into Bro are always activated, and hence show up as such even in bare mode.

### 4.1.5 Plugin Components

The following subsections detail providing individual types of functionality via plugins. Note that a single plugin can provide more than one component type. For example, a plugin could provide multiple protocol analyzers at once; or both a logging backend and input reader at the same time.

**Todo**

These subsections are mostly missing right now, as much of their content isn't actually plugin-specific, but concerns generally writing such functionality for Bro. The best way to get started right now is to look at existing code implementing similar functionality, either as a plugin or inside Bro proper. Also, for each component type there's a unit test in `testing/btest/plugins` creating a basic plugin skeleton with a corresponding component.

#### Bro Scripts

Scripts are easy: just put them into `scripts/`, as described above. The CMake infrastructure will automatically install them, as well include them into the source and binary plugin distributions.

#### Builtin Language Elements

**Functions** TODO

**Events** TODO

**Types** TODO

#### Protocol Analyzers

TODO.

#### File Analyzers

TODO.

#### Logging Writer

TODO.

#### Input Reader

TODO.

#### Packet Sources

TODO.

#### Packet Dumpers

TODO.

### 4.1.6 Hooks

TODO.

### 4.1.7 Testing Plugins

A plugin should come with a test suite to exercise its functionality. The `init-plugin` script puts in place a basic *BTest* setup to start with. Initially, it comes with a single test that just checks that Bro loads the plugin correctly. It won't have a baseline yet, so let's get that in place:

```
# cd tests
# btest -d
[  0%] rot13.show-plugin ... failed
% 'btest-diff output' failed unexpectedly (exit code 100)
% cat .diag
== File ===============================
Demo::Rot13 - Caesar cipher rotating a string's characters by 13 places. (dynamic,␣
↪version 0.1)
    [Function] Demo::rot13

== Error ===============================
test-diff: no baseline found.
=======================================

# btest -U
all 1 tests successful

# cd ..
# make test
make -C tests
make[1]: Entering directory `tests'
```

```
all 1 tests successful
make[1]: Leaving directory `tests'
```

Now let's add a custom test that ensures that our bif works correctly:

```
# cd tests
# cat >rot13/bif-rot13.bro

# @TEST-EXEC: bro %INPUT >output
# @TEST-EXEC: btest-diff output

event bro_init()
    {
    print Demo::rot13("Hello");
    }
```

Check the output:

```
# btest -d rot13/bif-rot13.bro
[  0%] rot13.bif-rot13 ... failed
% 'btest-diff output' failed unexpectedly (exit code 100)
% cat .diag
== File ===============================
Uryyb
== Error ===============================
test-diff: no baseline found.
=======================================

% cat .stderr

1 of 1 test failed
```

Install the baseline:

```
# btest -U rot13/bif-rot13.bro
all 1 tests successful
```

Run the test-suite:

```
# btest
all 2 tests successful
```

### 4.1.8 Debugging Plugins

If your plugin isn't loading as expected, Bro's debugging facilities can help illuminate what's going on. To enable, recompile Bro with debugging support (`./configure --enable-debug`), and afterwards rebuild your plugin as well. If you then run Bro with `-B plugins`, it will produce a file `debug.log` that records details about the process for searching, loading, and activating plugins.

To generate your own debugging output from inside your plugin, you can add a custom debug stream by using the `PLUGIN_DBG_LOG(<plugin>,<args>)` macro (defined in `DebugLogger.h`), where `<plugin>` is the `Plugin` instance and `<args>` are printf-style arguments, just as with Bro's standard debugging macros (grep for `DBG_LOG` in Bro's `src/` to see examples). At runtime, you can then activate your plugin's debugging output with `-B plugin-<name>`, where `<name>` is the name of the plugin as returned by its `Configure()` method, yet with the namespace-separator `::` replaced with a simple dash. Example: If the plugin is called `Demo::Rot13`, use

`-B plugin-Demo-Rot13`. As usual, the debugging output will be recorded to `debug.log` if Bro's compiled in debug mode.

### 4.1.9 Documenting Plugins

**Todo**

Integrate all this with Broxygen.

- General Index
- search