Bro: An Open Source Network Intrusion Detection System

Robin Sommer
Computer Science Department
TU München, Germany
sommer@in.tum.de

Abstract: Bro is a powerful, but largely unknown open source network intrusion detection system. Based on a sound design, Bro achieves its main goals – separating policy from mechanisms, efficient operation in high-volume networks, and withstanding attacks against itself – by using an event-driven approach. Bro contains several analyzers (e.g. protocol decoders for a variety of network protocols and a signature matching engine), which are by themselves policy-neutral but raise events as an abstraction of the underlying network activity. Based on scripts written in Bro's own powerful scripting language, the user defines event handlers to specify his environment-specific policy.

We give an overview about the design and implementation of Bro, describe our experiences with deploying it in a large-scale research environment, and present some of our extensions.

1 Introduction

Due to the enormous increase in the number of attacks seen on the Internet, *network intrusion detection systems* (NIDSs) are an important part of many security policies. In contrast to a *host-based* intrusion detection system, a NIDS usually taps the network at some central point, monitoring all traffic to detect malicious activity. Traditionally, two approaches to network intrusion detection are differentiated: a system using *anomaly-detection* relies on a definition of *normal* network activity and alerts if traffic deviates from this profile. Deviation is typically measured using statistics. On the other hand, *misuse-detection* systems use a library of specific attack characteristics and generate an alert whenever they locate one of them in the monitored traffic. The draw-back of misuse-detection is the difficulty of recognizing previously unknown attacks. While anomaly-based NIDSs have the potential for this, they often suffer from a high number of false-alarms. The most common form of misuse detection is signature matching: the NIDS identifies an attack by recognizing a specific byte sequence within the network packets. Most commercial systems follow this approach [Jac99] as does the well-known open source software Snort [Roe99].

Taking a closer look at open source NIDSs, we notice that Snort is the only one widely deployed in real networks. It appears that most people are not aware of alternatives. This is interesting for three reasons. First, in other areas of security there usually is a variety of open source software available. Second, Snort shows some technical limitations that indicate that it is not always an optimal solution. Finally, there is at least one other very powerful open source system that is largely unknown: Bro [Pax99, Bro].

In this work, we present an overview of Bro. After discussing its main design goals, we describe Bro's general architecture. We show several examples of the various ways Bro performs its analysis, and discuss some of our own experiences during deploying and actively extending Bro, including some of the problems we encountered. Finally, we present some future work.

Bro originated as a research system. It is designed and developed by Vern Paxson of *ICSI's Center for Internet Research (ICIR), Berkeley*. He started the project in 1995 at *Lawrence Berkeley National Laboratory (LBL)*, and Bro is under active development since then. It was first published in 1998. Major extentions of Bro are contributed by people at the *University of California, Berkeley* and *Princeton University, New Jersey*. Among other locations, Bro is operationally deployed at the *University of California, Berkeley* and at LBL. Our group at TU Munich, Germany (formerly at Saarland University, Germany) has used and extended Bro since 2001. Deploying it in a large-scale research environment at a 622 MBit/s Internet uplink of the *Leibniz Rechenzentrum, Munich*, we are faced with problems which are often underestimated when using NIDSs in smaller networks.

Bro's fundamental design goal is to separate mechanism and policy. While Bro implements very sophisticated state management and protocol analysis, it is by itself *policy-neutral*. The network activity is abstracted into *events* which are passed from Bro's core to an upper layer, the *policy layer*. On this policy layer, the administrator defines environment specific constraints by writing custom scripts in a powerful scripting language. As a consequence of this design, Bro is neither fundamentally anomaly-based nor misuse-based. Instead it supports both approaches and, in fact, the default policy scripts shipped with Bro implement examples of anomaly- as well as misuse-detection. Its integrated signature matcher provides a superset of Snort's capabilites and may even use Snort's signatures by means of a converter (see 3.5).

Like most NIDS, Bro is based on a packet monitor that listens to network traffic at some central point of the network. By closely following the communication between every two connection endpoints, Bro can reconstruct the semantics of the connections. For each of them, Bro keeps comprehensive state information. For example, on the transport layer, Bro reconstructs TCP sessions, which includes reassembling the data stream and handling of retransmissions. On the application layer, it implements a variety of protocol-specific analyzers, e.g. for HTTP, SMTP, DNS and many others. The per-connection state is fully accessible by the policy layer, so policy scripts can base decisions on the semantics and the context of a connection rather than having to inspect individual packets.

Of course, while being very powerful, Bro also has limitations. It needs more time to understand and deploy than a system like Snort. Currently, the latest stable version is quite old while the development version is cutting-edge, but not very well tested yet in operational use. Perhaps most importantly, due to its origins as a research system some parts are quite unpolished. While there is a mailing list for coordinating development efforts, Bro needs more contributors to make it a general production-level system. Furthermore, more feedback is needed about the problems that arise in networks of different scales.

2 Design and implementation of Bro

Bro consists of three major parts : (i) the packet capture, (ii) the policy-neutral event engine which contains various low-level components like protocol analyzers, and (iii) the policy layer which defines the environment-specific constraints based on scripts supplied by the user. Starting from Bro's design goals, we will discuss each part in turn.

2.1 Design Goals

Bro's design has been guided by a set of goals [Pax99]. The most important ones are:

Separation of mechanism and policy: A network security policy depends heavily on the specific environment. For example, while a public research institute can often provide full Internet access to all internal systems, a corporate network may have to impose severe restrictions. By clearly separating the mechanisms needed to monitor (and potentially to enforce) a particular policy from its specification, we gain both simplicity and flexibility.

Efficient operation suitable for high-speed, large-volume monitoring: With network bandwidths routinely reaching one Gbit/s these days, any NIDS has to be highly efficient to cope with these amounts of data in real-time. Every dropped packet may potentially lead to a missed attack.

Withstanding attacks: While it is usually not the *main* goal of an attacker to compromise the NIDS itself, it is an obvious first step of an attack to disable the NIDS before targeting the real victim. Hence, a NIDS which can be easily turned off is not of much use at all.

2.2 Architecture

The design goals presented above have lead to the layered architecture shown in Figure 1. The task of the lowest layer, the *packet capture*, is to get the packets off the wire. It feeds them into the core of Bro, the *event engine*, which performs low-level processing like state management and protocol analysis. The engine generates a stream of *events* which are passed on to the *policy layer*. The policy layer evaluates the events according to user-supplied scripts.

Events are central to Bro's approach to network intrusion detection. An event is generated for relevant network activitiy, and represents an abstraction of the network packets into higher-level context information. For example, a typical event is `connection_established`. Each time Bro's core encounters a successful three-way-TCP-handshake between two hosts, it sends a `connection_established` event to the policy layer. An event is augmented by additional context, in this case by the involved IP addresses and ser-

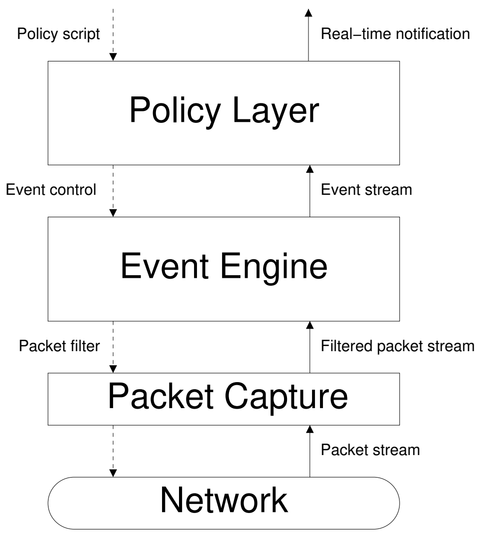

Figure 1: Design of Bro (adopted from [Pax99])

vice ports. On the policy layer, the user can define a corresponding *event handler* which specifies actions to be performed whenever a connection is established. For its operation, the handler does not have to inspect the involved packets itself, but works on the high-level context information.

Bro defines events for most relevant network activity on different levels of abstraction; each event is accompanied by a set of context information. For example, the `http_request` event is raised for each HTTP request performed by a client and passes the requested URI to the event handler. The event handler may then, for instance, inspect the URI for unauthorized accesses. On the other hand, `net_weird` is raised for IP-level obscurities such as corrupted checksums. Its context consists of a descriptive string. There are events for both normal and abnormal network activity: `login_success` signals a successful TELNET login while `login_failure` is raised on failed attempts. It is up to the policy layer to define what is acceptable. Raising an event by itself does not imply any judgement.

The use of events is Bro's main method of coping with a large amount of network traffic in real time. While the work within the event engine has to be performed for every packet, actions on the policy layer are done only per event. Hence, while Bro's core has to be highly efficient in decoding the network packets, event handlers may perform more costly operations. Additionally, they can save state information between multiple invocations. For example, a port scan analyzer may count for each source how many `connection_attempt` events have been seen. In addition, while the event engine is coded in C++ for efficiency, the policy layer interprets scripts during run-time. Although interpreting is expensive in general, this is feasible due to the low volume of the event stream. This provides us with all advantages of a flexible high-level scripting language.

2.2.1 Packet Capture

The lowest layer of Bro is the packet capture which passes the actual network packets to the event engine. For this task, Bro uses the library `libpcap` [Pca] which provides a platform-independent interface to different network link technologies. `libpcap` implements small but powerful filter expressions which can be used to reduce the amount of analyzed packets. For instance, if we do not want to analyze traffic on FTP data channels, we may simply ignore all packets on TCP port 20. Implementing this idea, Bro builds a filter expression dynamically during run-time. The filter selects only those for further analysis packets which are indeed needed by the current policy. As `libpcap` interfaces directly to kernel-level packet filters on many operating systems (like FreeBSD and Linux), it provides efficient access to the needed subset of network packets.

2.2.2 Event Engine

The core of Bro is its event engine. This layer performs all low-level work of analyzing network packets. The event engine gets raw IP packets from the packet capture, sorts them by connection, reassembles TCP data streams, and decodes application layer protocols. Whenever it encounters something potentially relevant to the policy layer, it generates an event. Needing to cope with large amounts of traffic, the event engine has to be highly efficient. Simultaniously, to withstand attacks against Bro itself, it has to be as robust as possible. Hence, it may not make any assumptions about the validity of network packets.

The event engine consists of several *analyzers*. Each of them is responsible for a well-defined task, for example decoding a specific protocol, performing signature-matching, or identifying backdoors. Each may raise a set of analyzer-specific events. Usually, an analyzer is accompanied by a default script which implements some general policy adjustable to the local environment.

The event engine can be divided into four major parts:

State Management: Bro's main data type is a *connection*. Each packet belongs to exactly one connection. For a connection-oriented protocol like TCP, the definition of a connection is obvious. For UDP and ICMP, Bro uses a flow-like abstraction (see [Pax02] for more details).

The most prevalent problem in any connection-based analysis is the difficulty to decide when a connection ends, so that associated state can be released. Protocol analysis is useful in the sense of providing us with good hints: if both endpoints of a TCP connection send a `FIN` packet, it is reasonable to assume that the connection has indeed been shut-down. But it is not sufficient: experience shows that on large networks there are tons of "crud" [Pax99], i.e. network packets which do not conform to any standard. While it is often hard to reason where it originates, it is important to note that most of the time it is not part of an attack. Common explanations are link-level errors and buggy protocol implementations. As usual in networking, the rule "be strict in what you send, liberal in what you accept" applies. Thus, Bro implements a sophisticated state expiration mechanism using various user-definable timeouts.

Transport Layer Analyzers: On the transport layer, Bro analyzes TCP, UDP and ICMP packets. The most interesting is the first one: TCP is a complex protocol involving state machines on both endpoints which determine the current state of the connection. Since the application layer analyzers (see Section 3) need the actual payload streams as sent by the two endpoints, Bro's TCP analyzer closely follows the various state changes, keeps track of acknowledgments, handles retransmissions and much more. The result is a real byte stream of payload data. This is in contrast to Snort's analysis of TCP: Snort does not reconstruct a full byte stream but uses various heuristics to combine several network packets into larger "virtual" packets which are then passed on to the rest of the system. This still leads to discretization problems.

We note that in certain situations reconstructing TCP payload is impossible without using further knowledge of the environment [PN98]. See Section [SP03a] for a discussion of how Bro may tackle some of these problems.

Application Layer Analyzers: The analysis of the application layer data of a connection depends on the particular service. There are analyzers for a wide variety of different protocols, e.g. HTTP, SMTP or DNS. See Section 3 for a more detailed description of some of them. Most of these analyzers do a fairly detailed analysis of the data stream. The SMTP decoder, for example, is able to decode MIME messages and may pass them on to the policy layer.

Infrastructure: The general infrastructure includes components like event and timer management, the script interpreter, and data structures.

2.2.3 Policy Layer

While the event engine itself is policy-neutral, the top layer of Bro defines the environment-specific network security policy. By writing handlers for events that may be raised by the event engine, the user can precisely define the particular constraints in his network. If a security breach is detected, the policy layer generates an alert.

The event handlers are written in Bro's own scripting language (see Figure 2 for a typical implementation of a handler). While providing all expected convenience of a powerful scripting language (like hash tables, dynamic memory management, dynamic typing and regular expressions), it has been designed with network intrusion detection in mind. There are specific data types for IP addresses, port numbers and time intervals. Bro's strings may contain null bytes (in contrast to `C` strings), and host names are implicitly resolved by `DNS`. One important goal of the language is to support the user in avoiding "simple errors" [Pax99]. Consequently, the language uses strong typing and lacks any support for unbounded loops[1].

[1]Event handlers have to finish as quickly as possible, so it is not feasible to use loops that may execute an arbitrary number of times.

```
const sensitive_URIs =  /etc.*\/.*(passwd|shadow|netconfig)/
                      | /\/cgi-bin\/(phf|php\.cgi|test-cgi)/;

const sensitive_post_URIs = /wwwroot/ &redef;

event http_request(c: connection, method: string, URI: string, version: string)
  {
  if ((sensitive_URIs in URI || (method == "POST" && sensitive_post_URIs in URI))
    {
    ALERT([$alert=HTTP_SensitiveURI, $conn=c,
           $msg=fmt("%s %s: %s %s", id_string(c$id), c$addl, method, URI)]);
    }
  }

# Sample output
#
#  HTTP_SensitiveURI 10.0.0.1/1078 > 10.0.0.2/http %1: GET /etc/passwd
```

Figure 2: Sample event handler in Bro's scripting language (adapted from Bro's default policy script `http-request`)

One or more handlers can be associated with an event. These handlers are called successivly if the event is raised. This allows modularizing policy scripts. Each script only defines the handlers it needs. During startup, Bro loads all scripts enabled by the user. If a script needs a particular analyzer, the analyzer is automatically enabled in Bro's event engine. This ensures that only those components of Bro are activated and hence consume resources that are indeed used.

While it is expected that policy scripts are written by the user, there are nevertheless quite a few of them shipped with Bro. These default scripts already perform a wide range of analyses and are easily customizable.

2.3 Robustness

One of Bro's design goals is to withstand attacks against the NIDS itself. For this, Bro makes two basic assumptions:

- Bro's source code is known to the attacker but the policy scripts are not.
- For each connection at most one of its endpoints is compromised.

The first assumption is similiar in spirit to *Kerckhoff's principle*: the mechanism is public but its parametrization is secret. Hence, the attacker may know what kind of low-level processing the NIDS is performing, but he cannot predict how the system will actually react to a particular input. This considerably complicates attacks which, for example, try to overload the system by exhausting its resources.

The second assumption implies that the attacker can only modify the packets on one side of a bi-directional connection. In particular, this allows the reliable reconstruction of TCP streams as it helps to limit the amount of buffered data without risking a misinterpretation of the stream. This assumption is reasonable since as soon as an attacker controls both endpoints we are lost anyway: By using any valid connection between the two systems, he can set up a covert channel, without virtually any chance of being detected.

Even if these two assumptions hold, attacks against the system may still be successful. However, this is not a problem that is specific to Bro but applies to all NIDSs. First, as long as a NIDS stores state about network traffic, it may be possible to overload it by carefully crafting traffic. Second, there are evasion methods which cannot be solved in general (see [PN98]). Consequently, although Bro cannot resist all attacks, it tries hard to at least notice them. Network packets are untrusted, they are carefully examined, and all pecularities are reported. Similarly, no assumption about the validity of the data is made during the application-layer protocol analysis. If processing of a single packet consumes too much cpu time, Bro's *watchdog* raises an alert, assuming that the delay is caused by exhausted resources or a coding error.

3 Analyzers

Bro's most important components are the *analyzers*. Among others, there are analyzers for application-layer decoding, anomaly-detection, signature matching and connection analysis. We will briefly describe several of them in this section. New analyzers are easy to add.

The term "analyzer" is only loosely defined and refers to part of Bro that performs some specific task. Most ananalyzers consist of a low-level part located inside Bro's event engine, and an accompanying default policy script. The latter can be customized by the user. Sometimes, however, the analyzer is just a policy script implementing some event handlers. For example, the scan detector is realized only on the script layer, relying on the general connection setup and tear-down events.

3.1 Application-layer Analyzers

3.1.1 HTTP

One of the most powerful analyzers is the HTTP decoder. It decodes both sides of a HTTP connection: the client side as well as the server side. For the former, the event engine generates an event for every request, supplying information about the connection, the URI, the HTTP method and the version. Additionally, a separate event is raised for every HTTP header line. On the server's side, the reply is parsed and dissected into its parts: header lines and MIME encoded components. The analyzer is capable to parse HTTP 1.0 and 1.1 messages with persistent connections and pipelining.

The default policy scripts log each request and the corresponding reply (Figure 3 shows a sample of its output). In addition, they may alert on accesses to a customizable list of "hot" URIs.

```
15:18:10 %1 start 10.0.0.1 > 66.35.250.150
15:18:10 %2 start 10.0.0.1 > 216.239.51.101
15:18:10 %3 start 10.0.0.1 > 66.35.250.123
15:18:10 %2 GET /ca/show_ads.js (200 "OK" [" 2294", 2294])
15:18:10 %1 GET / (200 "OK" [" 10401", 10401])
15:18:11 %3 GET /Slashdot/pc.gif?index,1053349807784 (200 "OK" [" 42", 42])
15:18:11 %4 start 10.0.0.1 > 66.35.250.110
15:18:11 %2 GET /search?[...]q=http%3A//slashdot.org/ (200 "OK" ["", 1417])
15:18:11 %4 GET /banner/goog5025en.gif?1053349808023 (200 "OK" [" 42", 42])
15:18:11 %4 GET /title.gif (200 "OK" [" 3473", 3473])
15:18:11 %4 GET /topics/topicmatrix.gif (200 "OK" [" 1564", 1564])
15:18:11 %4 GET /topics/topicslashback.gif (200 "OK" [" 2141", 2141])
15:18:11 %4 GET /topics/topicmovies.gif (200 "OK" [" 1476", 1476])
15:18:12 %4 GET /topics/topicwireless.gif (200 "OK" [" 1045", 1045])
15:18:12 %4 GET /topics/topichardware.gif (200 "OK" [" 1185", 1185])
15:18:12 %4 GET /slc.gif (200 "OK" [" 122", 122])
[...]
```

Figure 3: Output of HTTP analyzer when accessing `http://slashdot.org`

3.1.2 FTP

The FTP analyzer parses the control channel of FTP sessions and provides access to authorization information, commands, parameters and server replies. As it parses the negotiation of data channels, the default policy script notices if privileged ports are used or if a data channel is opened by a source which has not negotiated it. File accesses are logged, and accesses to a list of sensitive files are reported.

3.1.3 SMTP

SMTP sessions are fully parsed, too. The analyzer follows the protocol commands and checks for various inconsistencies. For transfered messages, it extracts the envelope information as well as the RFC822 mail itself. The latter may be decoded into all of its MIME parts. The analyzer even detects relays by correlating mails using `MD5` hashes.

3.1.4 DNS

The DNS analyzer is one of the few analyzers decoding UDP data. A DNS lookup usually consists of a at least one reply and one response. The analyzer correlates requests with replies, providing both questions and answers to event handlers. The default script handles all types of DNS messages, logs various pieces of information (see Figure 4), and checks for inconsistencies such as PTR scans.

```
Mon May 19 16:00:20 2003 #1 10.0.0.1/34252 > 131.159.14.1/dns start
#1 10.0.0.1 ?A www.net.in.tum.de = www.net.informatik.tu-muenchen.de \
                <TTL = 68214.000000>
#1 10.0.0.1 = 131.159.15.242 <TTL = 86400.000000>
#1 10.0.0.1 <auth NS> = dns1.net.informatik.tu-muenchen.de <TTL = 86400.000000>
#1 10.0.0.1 <auth NS> = tuminfo1.informatik.tu-muenchen.de <TTL = 86400.000000>
#1 10.0.0.1 <addl A> = 131.159.14.1 <TTL = 86400.000000>
#1 10.0.0.1 <addl A> = 131.159.0.1 <TTL = 67342.000000>
#1 finish
```

Figure 4: Output of DNS analyzer when looking up `http://www.net.in.tum.de`

3.2 Scan Detection

Prior to an actual attack, attackers often scan potential victims to identify the services they are running. Using Bro's connection-tracking, it is trivial to identify scans: For *vertical* scans, we simply count the number of (successful or unsuccessful) connection attempts to different ports on each host, and for *horizontal* scans, we count the number of connection attempts to each port within a set of hosts. One of Bro's default policy script implements this approach. Figure 5 shows an example.

```
14:27:42 PortScan 10.0.0.1 has scanned 1000 ports of 192.168.0.1
14:32:06 AddressScan 10.1.0.6 has scanned 100 hosts (http)
14:33:13 AddressScan 10.0.2.7 has scanned 100 hosts (http)
14:34:21 AddressScan 10.3.0.4 has scanned 100 hosts (http)
14:37:06 AddressScan 10.0.4.9 has scanned 100 hosts (http)
14:37:28 AddressScan 10.5.0.1 has scanned 100 hosts (netbios-ssn)
14:37:56 AddressScan 10.0.6.4 has scanned 100 hosts (http)
14:38:16 AddressScan 10.7.0.5 has scanned 500 hosts (netbios-ssn)
14:43:37 AddressScan 10.0.8.6 has scanned 100 hosts (1214/tcp)
```

Figure 5: Output of scan analyzer

3.3 Stepping Stones

A *stepping stone* is an intermediary host that someone uses to access a system: instead of connecting from host A to B directly, he may connect to C first, and then from C to B. In general, using a stepping stone is not illegitimate; sometimes it is even neccessary. But often stepping stones are used by attackers to hide their origin. Therefore, it is valuable to identify them.

Bro contains an analyzer that correlates the timing of packets on incoming connections with those on outgoing connections to identify stepping stones (see [ZP00] for a full description of the algorithm).

3.4 SYN-Floods

One of the most common denial-of-service attacks is flooding a victim with bogus connection requests by sending millions of `SYN` packets. While some operating systems already defend themselves against resource exhaustion (for example by using "`SYN` cookies" [Ber96]), such an attack has the potential to eventually fill up the network link. For a connection-tracking NIDS, another problem arises: storing state for each of the connection attempts would fill up the system's memory quite quickly. But, considering a single `SYN` at a time, it is hard to decide whether a connection request is bogus or not.

There exists an experimental `SYN`-flood analyzer for Bro to cope with this situation. While by default Bro creates state for each `SYN` packet, the analyzer counts the number of connection attempts to each host. If a count reaches a certain threshold, Bro starts to sample the packets to this destination aggressively, ignoring most of them. If the flood ends, sampling is turned off again.

3.5 Signatures

Most NIDSs match a large set of *signatures* against the network traffic. Here, a signature is basically a pattern of bytes that the NIDS tries to locate in the payload of network packets. As soon as a match is found, the system generates an alert. To limit the scope of a signature, it can usually be restricted to a subset of packets by defining conditions the packet header has to fulfill (for example aiming at a certain port or a destination). One of the most prominent of such systems is Snort which ships with a default set of roughly 1300 signatures; Figure 6 (a) shows one of them.

Bro's general approach to intrusion detection has a much broader scope than traditional signature-matching. Nevertheless, it still contains a signature engine providing a functionality that is similiar to that known from other systems. While Bro implements its own flexible signature language, there exists a converter which directly translates Snort's signatures into Bro's syntax. Figure 6 (b) shows the translation of a signature.

But Bro's signatures go well beyond the capabilities of other NIDSs. The most prevalent problem in signature-based intrusion detection are *false positives*. For some attacks (like buffer overflow exploits using a well-known sequence of machine code), defining byte-level signatures is straight-forward. But most attacks are not easily identifiable by just looking at byte sequences (see for example the one described in [Cve]). Often, this leads to very broad signatures, raising a high number of false alarms.

To enhance the quality of signature alerts, Bro makes use of *context*. There are two kinds of context which help to reduce the number of false positives: first, still on the byte-level, Bro provides full regular expressions instead of fixed patterns. Second, on a higher level, a byte-level match is augmented by semantic context made available by the protocol analysis and the scripting language. Instead of immediately generating an alert, a match just raises an event. By making use of Bro's state information inside an event handler, it is possible to eliminate irrelevant matches. For instance, Bro has the ability to identify the software

```
alert tcp any any -> [a.b.0.0/16,c.d.e.0/24] 80
  ( msg:"WEB-ATTACKS conf/httpd.conf attempt";
    nocase; sid:1373; flow:to_server,established;
    content:"conf/httpd.conf"; [...] )
```

(a) Snort

```
signature sid-1373 {
  ip-proto == tcp
  dst-ip == a.b.0.0/16,c.d.e.0/24
  dst-port == 80
  # The payload below is actually generated in a
  # case-insensitive format, which we omit here
  # for clarity.
  payload /.*conf\/httpd\.conf/
  tcp-state established,originator
  event "WEB-ATTACKS conf/httpd.conf attempt"
}
```

(b) Bro

Figure 6: Converting a Snort signature into Bro's language ([SP03b])

used by an `HTTP` server. If we are only interested in potentially successful attacks, we can simply ignore all signature matches for `IIS` exploits if the attacked server is running an `Apache` Web server. Other uses of context include *request/reply signatures* (we only report an alert if signature A matches on the client side and signature B matches on the server side of the same connection), and the identification of exploit scans (similar to port scans, we count how many matches a certain source triggers inside our network).

3.6 Connection Summaries

One of Bro's most useful analyzers is the connection analyzer. It is a bit unusual in that it by itself does not generate any alerts. Rather, for each connection, the analyzer reports a one-line summary (see Figure 7) containing a time-stamp, duration, service, transfered bytes (in terms of application-level throughput), source and destination IP, and a summary of the termination status. The latter indicates, for example, if the connection has been properly initiated and shut-down, or whether one side canceled the connection by sending a `RESET` packet. Internally, these summaries are generated by a full `TCP` state machine closely following the state transitions for each endpoint.

Having connection summaries, we can retrospectively analyze the network traffic in great detail without having to store packet traces. Due to the data volume, storing traces is often infeasible even for medium-scale links. Therefore, even if the NIDS does not report any malicious activity, we may nevertheless manually post-process the connection summaries once new information about an attack is available. In the past, this has proved to be very useful (see Section 4).

```
13:33:53 0.412065 http 391 4608 10.54.63.72 10.138.158.250 SF L
13:33:53 0.166779 http 282 5391 10.190.205.95 10.60.86.182 RSTO L
13:33:52 ? other-4662 ? ? 10.213.195.83 10.252.68.126 S0 L
13:33:52 ? pop-3 ? ? 10.199.178.2 10.210.72.62 S0 X
13:33:53 0.000368953 http ? ? 10.54.16.233 10.142.182.194 REJ X
13:33:52 0.49111 http 547 3518 10.80.43.190 10.114.206.95 SF L
13:33:53 0.247598 http 588 363 10.54.102.137 10.8.90.9 SF L
13:33:53 0.0862861 http 606 3659 10.54.49.247 10.58.190.28 SF L
13:33:53 0.152313 http 481 489 10.54.49.247 10.101.238.240 SF L
13:33:52 ? other-2500 ? ? 10.213.181.159 10.226.36.221 S0 L
13:33:53 0.107428 http ? ? 10.80.43.190 10.147.32.236 REJ L
13:33:52 1.15567 http 933 1760 10.80.43.210 10.173.203.23 SF L
13:33:52 ? http ? ? 10.80.38.149 10.170.171.85 S0 L
13:33:52 ? other-4662 ? ? 10.190.49.197 10.235.19.39 S0 L
13:33:52 ? other-4662 ? ? 10.190.49.197 10.212.169.68 S0 L
13:33:52 ? other-4662 ? ? 10.190.49.197 10.104.191.96 S0 L
```

Figure 7: Connection Summaries

4 Experiences in a High-Volume Environment

We have been using Bro in large-scale network environments for roughly two years now. First, we installed it at the 122 Mbit/s Internet uplink of the Saarland University, Germany. In September 2002, we installed Bro at the Leibniz-Rechenzentrum (LRZ), Munich. The LRZ provides Internet access to most non-commercial research facilities in the area of Munich, among them Technical University (TU), Ludwig-Maximilian-University (LMU) and several Max-Planck- and Fraunhofer-Institutes. The LRZ is connected to the Internet by a 622 Mbit/s link to the DFN (Deutsches Forschungsnetz), the German research network. This link transfers 1-2 TB of data each day on average. A large fraction of the volume is due to the domain `*.leo.org`, which is located at the TU and provides services like FTP-, IRC- and Webservers to people around the world (`ftp.leo.org` transfers 0.5-1 TB a day; `dict.leo.org`, an English-German dictionary, serves 2-2.5 million HTTP requests per day). At both locations, we are running Bro on FreeBSD systems with two AMD CPUs and one/two GB of memory respectively.

As a research group, our main objective is to gain experience with network intrusion detection in high-volume environments and to improve the current state-of-the-art technology. Due to its flexibility, Bro turned out to be an ideal NIDS for research purposes. Often, to test a new idea is suffices to write a new policy script. If this insufficient, we can easily extend Bro's open-source core. We are closely collaborating with Bro's main developer, and a significant part of our work has already been incorporated into the distribution. Note that we are not running Bro operationally in 24/7 production mode. We are concentrating on research, improving the system rather than actually chasing attackers.

Considering the enormous amount of data at our points of operation, Bro works remarkably well. On the 622 Mbit/s link of the LRZ, even a simple packet sniffer just writing all packets to disk is usually not able keep up with the volume. While Bro does not analyze every packet (but just the ones needed as specified in its configuration), it performs a significant amount of work (like fully decoding HTTP sessions). This confirms that decreasing the high-volume packet stream to a low-volume event stream is indeed feasible.

Nevertheless, depending on the time of day and what analyzers are activated, Bro may not be able to keep up with the traffic without dropping packets. For example, enabling server-side HTTP decoding at noon for all subnets served by the LRZ turns out to be infeasible. Yet, analyzing only the client-side works fine of most the time (in particular this means that Bro can extract requested URIs). We are currently experimenting with load-level mechanisms, which selectively disable certain analyzers based on the current network load. Similarily, we can reduce the load by temporarily ignoring certain subnets. For example, at the LRZ we can exclude `*.leo.org`.

One of our most prevalent problems is the large amount of memory Bro needs. Keeping per-connection state is naturally expensive if you regularly encounter 300k-600k connections per hour. Thus, one emphasis is to improve the state-management by expiring old state more aggressively, possibly without opening doors for attackers. This is particular important during peak loads. While most network technology can be designed for the average case, a NIDS has to work reliably even in the worst case - which can be intentionally provoked by an attacker. An extreme example are denial-of-service attacks. `irc.leo.org` is a regular victim of `SYN`-Floods. During a typical attack, we see 2-3 million `SYN` packets per minute. Bro would normally create state for each `SYN`, which would exhaust the available memory in a couple of minutes. Therefore, we implemented the `SYN`-Flood analyzer (see Section 3.4). After noticing the flood, it starts to sample the packets to the victim.

Comparing to Snort, we see that Bro takes a completely different approach to network intrusion detection. Nevertheless, we implemented a signature analyzer (see Section 3.5), which provides functionality similar to Snort. It even leverages Snort's comprehensive signature library. While evaluating Bro's signatures, we encountered several limitations of Snort. For instance, its protocol decoders are very limited: the TCP stream reassemblation is incomplete, and the HTTP decoder only considers requests at the start of a packet. The latter is not sufficient for pipelined connections as used in HTTP `1.1`. Bro's protocol decoding is far more sophisticated. Another problem of Snort is its signature library: while it is very comprehensive, the quality of many of the signatures is quite low. This leads to many false alarms. For some attacks, there are several signatures which differ in their quality. For some signatures, their meta-information is wrong. Of course, if we just convert Snort's signatures into Bro's language, we have to live with some of the limitations. Therefore, in the future we hope to build a set of high-quality signatures which take advantage of Bro's features. But while we may be able to jump start this, we will need the help of the community to achieve a comprehensive signature archive.

One aspect of using Bro is that it needs to be taylored to the local environment. While systems like Snort basically just need to know the local IP addresses and perhaps which part of the signature library we want to use, we have to tell Bro more about our network to use it effectively. Most of the default policy scripts need to be adjusted, and we have to write own scripts which define our specific security policy. While ability adds flexibility, it requires time.

The more we tell Bro about our network the better it can identify malicious activity. This raises the question if a location like the uplink of the LRZ is indeed a good place to deploy Bro. On the internal side of the router, there are many subnets of varying size, all administrated independently with different security policies (if any at all). As a consequence, it is impossible to tell Bro exactly what is indeed accepted activity. On the other hand, there are attacks which are easiest to recognize at this level; consider attacks which are not aimed at a particular host but scan a large range of the IP space for potential victims ("script kiddies" and worms come to mind). So, the answer probably lies in between: to get the best of both the worlds, we would like to install Bro at several points located on different levels of the network: from a local subnet of workstations to the backbone. An interesting question is how to coordinate and correlate different instances of a NIDS. This is part of our ongoing research.

Bro's connection summaries (see Section 3.6) proved highly valuable in March 2003 at Saarland University. Attackers compromised several machines on the campus by exploiting a bug contained in certain `SSH` server versions [Ssh]. A few days later, the compromised systems started to participate in distributed denial-of-service attacks. While the network administrators were able to identify some of the hosts by observing enormous amounts of transfered data, it was hard to tell if there were more of them, just waiting to be activated. Knowing some of the victims, we used Bro's connection summaries to identify the SSH sessions which carried the initial exploits. We then checked for connections to other internal hosts showing similar characteristics. In fact, we found several more of them and all proved to be compromised.

Finally, used Bro in areas of research other than network security, too. Its powerful protocol decoding can be effectivly used for a large range of traffic analysis tasks. For example, to identify certain characteristics of Web sessions, we instrumented Bro to log the relevant parts of the HTTP communication by simply writing a policy script. Similarly, connection summaries are an invaluable source for traffic statistics. Of late, Bro even contains a framework for anonymizing packet traces. Such traces may eventually be collected in an archive of openly-available real-world traces.

5 Future Work

The protocol analyzers are Bro's heart, and we are in the process of implementing some more of them. Currently, we are working on a SSL/TLS analyzer which will provide access to the certificates exchanged by the parties. Other candidates are POP, IMAP, IRC, NETBIOS/SMB (some work has already been done on this), SNMP, and the various peer-to-peer protocols.

Making Bro more robust during floods and flash-crowds is rather important in high-volume environments. Possible approaches are further optimization of the state-management, and a more systematic integration of packet sampling. To avoid opening doors to attackers, this needs to be done on a sound theoretical base.

Bro already contains preliminary support for *vulnerability profiles* which is the set of software a particular host is running. If Bro knows the profile of a host – either by manual configuration or, more interestingly, by deducing it from the traffic – it can use this profile to estimate the severity of an attack. Currently, Bro is able to identify HTTP and SSH software, but we will further extend this to include more protocols. We plan to design an unified framework on the script level, and to integrate passive OS fingerprinting techniques.

Most ambitiously, we would like to see several Bro systems running at different locations of the network to cooperate. To achieve this, Bro needs a communication interface and new techniques for globally correlating local events. Additionally, Bro could include more data sources in its analysis. For example, while we may not have access to the packets of a network, we may be able to get NetFlow [Cis] records from a router. Theoretically, it should be possible to extend some of Bro's parts to work on flow information.

Independent of all these enhancements, Bro needs some work to make it a production-level system effectively deployable by people who are not familiar with its internals. Reading Bro's mailing list, there are some issues unfamiliar users regularly stumble across. Among them are a lack of introductory documentation[2], portability problems, and configuration difficulties. These problems are mainly due to the limited amount of resources available for a research system. To promote, maintain, and extend Bro, any help from the community is greatly appreciated. Similarily, any feedback about deploying Bro is welcome.

6 Conclusion

We present Bro, a powerful but still largely unknown open source network intrusion detection system. It is designed with three main goals in mind: efficiency, robustness, and separation of policy from mechanisms. To achieve these, a wide variety of protocol-specific analyzers decode network packets. They raise events for relevant activity which are processed by user-supplied handlers. Thus, Bro abstracts from a high-volume traffic stream to a low-volume stream of events suitable for more cpu-intensive processing.

We are successfully deploying Bro in a large-scale network environment. Closely collaborating with Bro's main developer, several of our extensions have already been incorporated into the distribution. Nevertheless, Bro is still a research system. Contributors are needed to make it an production-ready system.

7 Acknowledgments

We would like to thank the Leibniz-Rechenzentrum, Munich, Germany, and the Rechnerbetriebsgruppe of the Saarland University, Saarbrücken, Germany, for enabling our experiments with Bro and their help in understanding the network environments. We are in debt to Vern Paxson for his continuous support of our work with Bro. Finally, we thank Vinay Aggarwal, Holger Dreger and Anja Feldmann for their valuable suggestions on how to improve this presentation.

[2] Bro is accompanied by a rather extensive reference manual, although some parts of it are a bit outdated.

References

[Ber96] Daniel J. Bernstein. SYN cookies. http://cr.yp.to/syncookies.html, 1996.

[Bro] Bro. http://www.icir.org/vern/bro-info.html.

[Cis] Cisco Systems Inc. NetFlow Services and Applications - White paper. http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.h%tm.

[Cve] CVE-2000-0778. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0778.

[Jac99] Kathleen Jackson. Intrusion Detection System Product Survey. Technical Report LA-UR-99-3883, Los Alamos National Laboratory, June 1999.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[Pax02] Vern Paxson. *The Bro 0.8 User Manual*, 2002.

[Pca] libpcap. http://www.tcpdump.org.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Conference on Systems Administration (LISA-99)*, pages 229–238. USENIX Association, November 1999.

[SP03a] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy (to appear)*, 2003.

[SP03b] Robin Sommer and Vern Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. *In submission*, 2003.

[Ssh] Vulnerability Note VU#945216. http://www.kb.cert.org/vuls/id/945216.

[ZP00] Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *Proceedings of the 9th USENIX Security Symposium (SECURITY-00)*, pages 171–184. The USENIX Association, August 2000.