Bro: A System for Detecting Network Intruders in Real-Time

Vern Paxson

Lawrence Berkeley National Laboratory, Berkeley, CA*
and
AT&T Center for Internet Research at ICSI, Berkeley, CA
vern@aciri.org

Abstract

We describe Bro, a stand-alone system for detecting network intruders in real-time by passively monitoring a network link over which the intruder's traffic transits. We give an overview of the system's design, which emphasizes high-speed (FDDI-rate) monitoring, real-time notification, clear separation between mechanism and policy, and extensibility. To achieve these ends, Bro is divided into an "event engine" that reduces a kernel-filtered network traffic stream into a series of higher-level events, and a "policy script interpreter" that interprets event handlers written in a specialized language used to express a site's security policy. Event handlers can update state information, synthesize new events, record information to disk, and generate real-time notifications via *syslog*. We also discuss a number of attacks that attempt to subvert passive monitoring systems and defenses against these, and give particulars of how Bro analyzes the six applications integrated into it so far: Finger, FTP, Portmapper, Ident, Telnet and Rlogin. The system is publicly available in source code form.

1 Introduction

With growing Internet connectivity comes growing opportunities for attackers to illicitly access computers over the network. The problem of detecting such attacks is termed *network intrusion detection*, a relatively new area of security research [MHL94]. We can divide these systems into two types, those that rely on audit information gathered by the hosts in the network they are trying to protect, and those that operate "stand-alone" by observing network traffic directly, and passively, using a packet filter. There is also increasing interest in building hybrid systems that combine these two approaches [Ax99].

In this paper we focus on the problem of building stand-alone systems, which we will term "monitors." Though monitors necessarily face the difficulties of more limited information than systems with access to audit trails, monitors also gain the major benefit that they can be added to a network without requiring any changes to the hosts. For our purposes—monitoring a collection of several thousand heterogeneous, diversely-administered hosts—this advantage is immense.

Our monitoring system is called Bro (an Orwellian reminder that monitoring comes hand in hand with the potential for privacy violations). A number of commercial products exist that do what Bro does, generally with much more sophisticated interfaces and management software [In99, To99, Ci99],[1] and larger "attack signature" libraries. To our knowledge, however, there are no detailed accounts in the network security literature of how monitors can be built. Furthermore, monitors can be susceptible to a number of attacks aimed at subverting the monitoring; we believe the attacks we discuss here have not been previously described in the literature. Thus, the contribution of this paper is not at heart a novel idea (though we believed it novel when we undertook the project, in 1995), but rather a detailed overview of some experiences with building such a system.

Prior to developing Bro, we had significant operational experience with a simpler system based on off-line analysis of `tcpdump` [JLM89] trace files. Out of this experience we formulated a number of design goals and requirements:

High-speed, large volume monitoring For our environment, we view the greatest source of threats as external hosts connecting to our hosts over the Internet. Since the network we want to protect has a single link connecting it to the remainder of the Internet (a "DMZ"), we can economically monitor our greatest potential source of attacks by passively watching the DMZ link.

*This paper appears in *Computer Networks*, 31(23–24), pp. 2435–2463, 14 Dec. 1999. This work was supported by the Director, Office of Energy Research, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division of the United States Department of Energy under Contract No. DE-AC03-76SF00098. An earlier version of this paper appeared in the Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998.

[1] Or at least appear, according to their product literature, to do the same things—we do not have direct experience with any of these products.

A somewhat different sort of product, the "Network Flight Recorder," is described in [RLSSLW97], though it is now increasingly used for intrusion detection [Ne99].

However, the link is an FDDI ring, so to monitor it requires a system that can capture traffic at speeds of up to 100 Mbps.

No packet filter drops If an application using a packet filter cannot consume packets as quickly as they arrive on the monitored link, then the filter will buffer the packets for later consumption. However, eventually the filter will run out of buffer, at which point it *drops* any further packets that arrive. From a security monitoring perspective, drops can completely defeat the monitoring, since the missing packets might contain exactly the interesting traffic that identifies a network intruder. Given our first design requirement—high-speed monitoring—then avoiding packet filter drops becomes another strong requirement.

It is sometimes tempting to dismiss a problem such as packet filter drops with an argument that it is unlikely a traffic spike will occur at the same time as an attack happens to be underway. This argument, however, is completely undermined if we assume that an attacker might, in parallel with a break-in attempt, *attack the monitor itself* (see below).

Real-time notification One of our main dissatisfactions with our initial off-line system was the lengthy delay incurred before detecting an attack. If an attack, or an attempted attack, is detected quickly, then it can be much easier to trace back the attacker (for example, by telephoning the site from which they are coming), minimize damage, prevent further break-ins, and initiate full recording of all of the attacker's network activity. Therefore, one of our requirements for Bro was that it detect attacks in real-time. This is not to discount the enormous utility of keeping extensive, permanent logs of network activity for later analysis. Invariably, when we have suffered a break-in, we turn to these logs for retrospective damage assessment, sometimes searching back a number of months.

Mechanism separate from policy Sound software design often stresses constructing a clear separation between mechanism and policy; done properly, this buys both simplicity and flexibility. The problems faced by our system particularly benefit from separating the two: because we have a fairly high volume of traffic to deal with, we need to be able to easily trade-off at different times how we filter, inspect and respond to different types of traffic. If we hardwired these responses into the system, then these changes would be cumbersome (and error-prone) to make.

Extensible Because there are an enormous number of different network attacks, with who knows how many waiting to be discovered, the system clearly must be designed in order to make it easy to add to it knowledge of new types of attacks. In addition, while our system is a research project, it is at the same time a production system that plays a significant role in our daily security operations. Consequently, we need to be able to upgrade it in small, easily debugged increments.

Avoid simple mistakes Of course, we always want to avoid mistakes. However, here we mean that we particularly desire that the way that a site defines its security policy be both clear and as error-free as possible. (For example, we would not consider expressing the policy in C code as meeting these goals.)

The monitor will be attacked We must assume that attackers will (eventually) have full knowledge of the techniques used by the monitor, and access to its source code, and will use this knowledge in attempts to subvert or overwhelm the monitor so that it fails to detect the attacker's break-in activity. This assumption significantly complicates the design of the monitor; but failing to address it is to build a house of cards.

We do, however, allow one further assumption, namely that *the monitor will only be attacked from one end*. That is, given a network connection between hosts A and B, we assume that at most one of A or B has been compromised and might try to attack the monitor, but not both. This assumption greatly aids in dealing with the problem of attacks on the monitor, since it means that *we can trust one of the endpoints* (though we do not know which).

In addition, we note that this second assumption costs us virtually nothing. If, indeed, both A and B have been compromised, then the attacker can establish intricate covert channels between the two. These can be immeasurably hard to detect, depending on how devious the channel is; that our system fails to do so only means we give up on something extremely difficult anyway.

A final important point concerns the broader context for our monitoring system. Our site is engaged in basic, unclassified research. The consequences of a break-in are usually limited to (potentially significant) expenditure in lost time and re-securing the compromised machines, and perhaps a tarnished public image depending on the subsequent actions of the attackers. Thus, while we very much aim to minimize break-in activity, we do not try to achieve “airtight” security. We instead emphasize monitoring over blocking when possible. Obviously, other sites may have quite different security priorities, which we do not claim to address.

In the remainder of this paper we discuss how the design of Bro attempts to meet these goals and constraints. First, in § 2 we give an overview of the structure of the whole system. § 3 presents the specialized `Bro` language used to express a site's security policy. We turn in § 4 to the details of how the system is currently implemented. § 5 discusses attacks on the monitoring system. § 6 looks at the specialized analysis

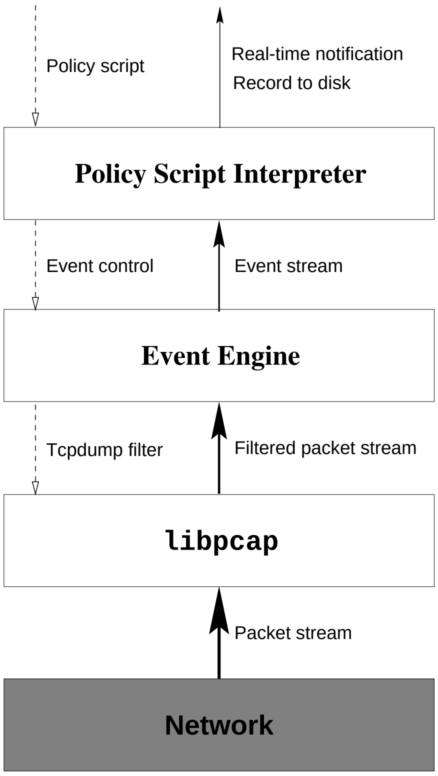

Figure 1: Structure of the Bro system

Bro does for six Internet applications: FTP, Finger, Portmapper, Ident, Telnet and Rlogin. § 7 gives the status of the implementation and our experiences with it, including a brief assessment of its performance. § 8 offers some thoughts on future directions. Finally, an Appendix illustrates how the different elements of the system come together for monitoring Finger traffic.

2 Structure of the system

Bro is conceptually divided into an "event engine" that reduces a stream of (filtered) packets to a stream of higher-level network events, and an interpreter for a specialized language that is used to express a site's security policy. More generally, the system is structured in layers, as shown in Figure 1. The lower-most layers process the greatest volume of data, and hence must limit the work performed to a minimum. As we go higher up through the layers, the data stream diminishes, allowing for more processing per data item. This basic design reflects the need to conserve processing as much as possible, in order to meet the goals of monitoring high-speed, large volume traffic flows without dropping packets.

2.1 `libpcap`

From the perspective of the rest of the system, just above the network itself is `libpcap` [MLJ94], the packet-capture library used by `tcpdump` [JLM89]. Using `libpcap` gains significant advantages: it isolates Bro from details of the network link technology (Ethernet, FDDI, SLIP, etc.); it greatly aids in porting Bro to different Unix variants (which also makes it easier to upgrade to faster hardware as it becomes available); and it means that Bro can also operate on `tcpdump` save files, making off-line development and analysis easy.

Another major advantage of `libpcap` is that if the host operating system provides a sufficiently powerful kernel packet filter, such as BPF [MJ93], then `libpcap` downloads the filter used to reduce the traffic into the kernel. Consequently, rather than having to haul every packet up to user-level merely so the majority can be discarded (if the filter accepts only a small proportion of the traffic), the rejected packets can instead be discarded in the kernel, without suffering a context switch or data copying. Winnowing down the packet stream as soon as possible greatly abets monitoring at high speeds without losing packets.

The key to packet filtering is, of course, judicious selection of which packets to keep and which to discard. For the application protocols that Bro knows about, it captures every packet, so it can analyze how the application is being used. In `tcpdump`'s filtering language, this looks like:

```
port finger or port ftp or tcp port 113 or
port telnet or port login or port 111
```

That is, the filter accepts any TCP packets with a source or destination port of 79 (Finger), 21 (FTP), 113 (Ident), 23 (Telnet), 513 (Rlogin), and any TCP or UDP packets with a source or destination port of 111 (Portmapper). In addition, Bro uses:

```
tcp[13] & 7 != 0
```

to capture any TCP packets with the SYN, FIN, or RST control bits set. These packets delimit the beginning (SYN) and end (FIN or RST) of each TCP connection. Because TCP/IP packet headers contain considerable information about each TCP connection, from just these control packets one can extract connection start time, duration, participating hosts, ports (and hence, generally, the application protocol), and the number of bytes sent in each direction. Thus, by capturing on the order of only 4 packets (the two initial SYN packets exchanged, and the final two FIN packets exchanged), we can determine a great deal about a connection even though we filter out all of its data packets.

The final filter we use is:

```
ip[6:2] & 0x3fff != 0
```

which captures IP fragments, necessary for sound traffic analysis, and also to protect against particular attacks on the monitoring system § 5.3.

When using a packet filter, one must also choose a *snapshot length*, which determines how much of each packet should be captured. For example, by default `tcpdump` uses a snapshot length of 68 bytes, which suffices to capture link-layer and TCP/IP headers, but generally discards most of the data in the packet. The smaller the snapshot length, the less data per accepted packet needs to copied up to the user-level by the packet filter, which aids in accelerating packet processing and avoiding loss. On the other hand, to analyze connections at the application level, Bro requires the full data contents of each packet. Consequently, it sets the snapshot length to capture entire packets.

2.2 Event engine

The resulting filtered packet stream is then handed up to the next layer, the Bro "event engine." This layer first performs several integrity checks to assure that the packet headers are well-formed, including verifying the IP header checksum. If these checks fail, then Bro generates an event indicating the problem and discards the packet. It is also at this point that Bro reassembles IP fragments so it can then analyze complete IP datagrams.

If the checks succeed, then the event engine looks up the connection state associated with the tuple of the two IP addresses and the two TCP or UDP port numbers, creating new state if none already exists. It then dispatches the packet to a handler for the corresponding connection (described shortly). Bro maintains a `tcpdump` trace file associated with the traffic it sees. The connection handler indicates upon return whether the engine should record the entire packet to the trace file, just its header, or nothing at all. This triage trades off the completeness of the traffic trace versus its size and time spent generating the trace. Generally, Bro records full packets if it analyzed the entire packet; just the header if it only analyzed the packet for SYN/FIN/RST computations; and skips recording the packet if it did not do any processing on it.

We now give an overview of general processing done for TCP and UDP packets. In both cases, the processing ends with invoking a handler to process the data payload of the packet. For applications known to Bro, this results in further analysis, as discussed in § 6. For other applications, analysis ends at this point.

TCP processing. For each TCP packet, the connection handler (a C++ virtual function) verifies that the entire TCP header is present and validates the TCP checksum over the packet header and payload. If successful, it then tests whether the TCP header includes any of the SYN/FIN/RST control flags, and if so adjusts the connection's state accordingly. Finally, it processes any data acknowledgement present in the header, and then invokes a handler to process the payload data, if any.

Different changes in the connection's state generate different events. When the initial SYN packet requesting a connection is seen, the event engine schedules a timer for T seconds in the future (presently, five minutes); if the timer expires and the connection has not changed state, then the engine generates a `connection_attempt` event. If before that time, however, the other connection endpoint replies with a correct SYN acknowledgement packet, then the engine immediately generates a `connection_established` event, and cancels the connection attempt timer. On the other hand, if the endpoint replies with a RST packet, then the connection attempt has been rejected, and the engine generates `connection_rejected`. Similarly, if a connection terminates via a normal FIN exchange, then the engine generates `connection_finished`. It also generates several other events reflecting more unusual ways in which connections can terminate.

UDP processing. UDP processing is similar but simpler, since there is no connection state, except in one regard. If host A sends a UDP packet to host B with a source port of p_A and a destination port of p_B, then Bro considers A as having initiated a "request" to B, and establishes pseudo-connection state associated with that request. If B subsequently sends a UDP packet to A with a source port of p_B and destination p_A, then Bro considers this packet to reflect a "reply" to the request. The handlers (virtual functions) for the UDP payload data can then readily distinguish between requests and replies for the usual case when UDP traffic follows that pattern. The default handlers for UDP requests and replies simply generate `udp_request` and `udp_reply` events.

2.3 Policy script interpreter

After the event engine has finished processing a packet, it then checks whether the processing generated any events. (These are kept on a FIFO queue.) If so, it processes each event until the queue is empty, as described below. It also checks whether any timer events have expired, and if so processes them, too (see § 4 for more on timer expiration).[2]

A key facet of Bro's design is the clear distinction between the generation of events versus what to do in response to the events. These are shown as separate boxes in Figure 1, and this structure reflects the separation between mechanism and policy discussed in § 1. The "policy script interpreter" executes scripts written in the specialized `Bro` language (detailed in § 3). These scripts specify event handlers, which are essentially identical to Bro functions except that they don't return a value. For each event passed to the interpreter, it retrieves the (semi-)compiled code for the corresponding handler, binds the values of the events to the arguments of the

[2] There is a subtle design decision involved with processing all of the generated events before proceeding to read the next packet. We might be tempted to defer event processing until a period of relatively light activity, to aid the engine with keeping up during periods of heavy load. However, doing so can lead to races: the "event control" arrow in Figure 1 reflects the fact that the policy script can, to a limited degree, manipulate the connection state maintained inside the engine. If event processing is deferred, then such control may happen after the connection state has already been changed due to more recently-received traffic. So, to ensure that event processing always reflects fresh data, and does not inadvertently lead to inconsistent connection state, we process events immediately, before moving on to newly-arrived network traffic.

handler, and interprets the code. This code in turn can execute arbitrary Bro scripting commands, including generating new events, logging real-time notifications (using the Unix *syslog* function), recording data to disk, or modifying internal state for access by subsequently invoked event handlers (or by the event engine itself).

Finally, along with separating mechanism from policy, Bro's emphasis on asynchronous events as the link between the event engine and the policy script interpreter buys a great deal in terms of extensibility. Adding new functionality to Bro generally consists of adding a new protocol analyzer to the event engine and then writing new event handlers for the events generated by the analyzer. Neither the analyzer nor the event handlers tend to have much overlap with existing functionality, so for the most part we can avoid the subtle interactions between loosely coupled modules that can easily lead to maintenance headaches and buggy programs.

3 The `Bro` language

As discussed above, we express security policies in terms of scripts written in the specialized `Bro` language. In this section we give an overview of the language's features. The aim is to convey the flavor of the language, rather than describe it precisely.

Our goal of "avoid simple mistakes" (§ 1), while perhaps sounding trite, in fact heavily influenced the design of the `Bro` language. Because intrusion detection can form a cornerstone of the security measures available to a site, we very much want our policy scripts to behave as expected. From our own experience, a big step towards avoiding surprises is to use a strongly typed language that detects typing inconsistencies at compile-time, and that guarantees that all variable references at run-time will be to valid values. Furthermore, we have come to appreciate the benefits of domain-specific languages, that is, languages tailored for a particular task. Having cobbled together our first monitoring system out of `tcpdump`, `awk`, and shell scripts, we thirsted for ways to deal directly with hostnames, IP addresses, port numbers, and the like, rather than devising ASCII pseudo-equivalents. By making these sorts of entities first-class values in `Bro`, we both increase the ease of expression offered by the language and, due to strong typing, catch errors (such as comparing a port to an IP address) that might otherwise slip by.

3.1 Data types and constants

Atomic types. `Bro` supports several types familiar to users of traditional languages: `bool` for booleans, `int` for integers, `count` for non-negative integers ("unsigned" in C), `double` for double-precision floating point, and `string` for a series of bytes. The first four of these (all but `string`) are termed *arithmetic* types, and mixing them in expressions promotes `bool` to `count`, `count` to `int`, and `int` to `double`.

`Bro` provides `T` and `F` as `bool` constants for true and false; a series of digits for `count` constants; and C-style constants for `double` and `string`.

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

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

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

Similarly, it is important that when Bro logs such strings, or prints them as text to a file, that it expands embedded NULs into visible escape sequences to flag their appearance.

`Bro` also includes a number of non-traditional types, geared towards its specific problem domain. A value of type `time` reflects an absolute time, and `interval` a difference in time. Subtracting two `time` values yields an `interval`; adding or subtracting an `interval` to a `time` yields a `time`; adding two `time` values is an error. There are presently no `time` constants, but `interval` constants can be specified using a numeric (possibly floating-point) value followed by a unit of time, such as "`30 min`" for thirty minutes.

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

There are two forms of `port` constants. The first consists of an unsigned integer followed by either "`/tcp`" or "`/udp`." So, for example, "`80/tcp`" corresponds to TCP port 80 (the HTTP protocol used by the World Wide Web). The second form of constant is specified using a predefined identifier, such as "`http`", equivalent to "`80/tcp`." Originally, we would look up otherwise-undefined identifiers using the *getservbyname* library routine. However, doing so not only runs into difficulties when a single name like "`domain`" has both TCP and UDP definitions, but, more fundamentally, erodes portability because a *getservbyname* service name known on one system might well be missing from another system, rendering invalid any `Bro` scripts written using the service name.

Values of type `port` may be compared for equality or ordering (for example, "`20/tcp < telnet`" yields true), but otherwise cannot be operated on.

Another networking type provided by `Bro` is `addr`, corresponding to an IP address. These are represented internally as unsigned, 32-bit integers, but in Bro scripts the only operations that can be performed on them are comparisons for equality or inequality (also, a built-in function provides masking, as discussed below). Constants of type addr have the familiar "dotted quad" format, $A_1.A_2.A_3.A_4$.

More interesting are *hostname* constants. There is no Bro type corresponding to Internet hostnames, because hostnames can correspond to multiple IP addresses, so one quickly runs into ambiguities if comparing one hostname with another. Bro does, however, support hostnames as constants. Any series of two or more identifiers delimited by dots forms a hostname constant, so, for example, "lbl.gov" and "www.microsoft.com" are both hostname constants (the latter, as of this writing, corresponds to 6 distinct IP addresses). The value of a hostname constant is a list of addr containing one or more elements. These lists cannot be used in Bro expressions; but they play a central role in initializing Bro table's and set's, discussed in § 3.3 below.

Aggregate types. Bro also supports a number of aggregate types. A record is a collection of elements of arbitrary type. For example, the predefined conn_id type, used to hold connection identifiers, is defined in the Bro run-time initialization file as:

```
type conn_id: record {
    orig_h: addr;
    orig_p: port;
    resp_h: addr;
    resp_p: port;
};
```

The orig_h and resp_h elements (or "fields") have type addr and hold the connection originator's and responder's IP addresses. Similarly, orig_p and resp_p hold the originator and responder ports. Record fields are accessed using the "$" operator.

For specifying security policies, a particularly useful Bro type is table. Bro tables have two components, a set of *indices* and a *yield type*. The indices may be of any atomic (non-aggregate) type, and/or any record types that, when (recursively) expanded into all of their elements, are comprised of only atomic types. (Thus, Bro tables provide a form of associative array.) So, for example,

```
table[port] of string
```

can be indexed by a port value, yielding a string, and:

```
table[conn_id] of ftp_session_info
```

is indexed by a conn_id record—or, equivalently, by an addr, a port, another addr, and another port—and yields an ftp_session_info record as a result.

Closely related to table types are set types. These are simply table types that do not yield a value. Their purpose is to maintain collections of tuples, expressed in terms of the set's indices. The examples in § 3.3 clarify how this is useful.

Another aggregate type supported is file. Support for files is presently crude: a script can open files for writing or appending, and can pass the resulting file variable to the print command to specify where it should write, but that is all. Also, these files are simple ASCII. In the future, we plan to extend files to support reading, ASCII parsing, and binary (typed) reading and writing.

Finally, above we alluded to the list type, which holds zero or more instances of a value. Currently, this type is not directly available to the Bro script writer, other than implicitly when using *hostname* constants. Since its present use is primarily internal to the script interpreter (when initializing variables, per § 3.3), we do not describe it further.

Regular expressions. The last built-in Bro type is pattern. Patterns are Unix-style regular expressions; in particular, the syntax used by the *flex* utility [Pa96]. Pattern constants are enclosed by / delimiters. For example:

```
/sync|lp|uucp|operator|ezsetup|4dgifts/
```

is a pattern that matches a number of common default Unix accounts.

Presently, only two operations are allowed on pattern values: assignment, and testing to see whether the pattern value matches a given string (discussed below).

3.2 Operators

Bro provides a number of C-like operators (+, -, *, /, %, !, &&, ||, ?:, relationals like <=) with which we assume the reader is familiar, and will not detail here. Assignment is done using =, table and set indexing with [], and function invocation and event generation with (). Numeric variables can be incremented and decremented using ++ and --. Record fields are accessed using $, to avoid ambiguity with *hostname* constants. Assignment of aggregate values is *shallow*—the newly-assigned variable refers to the same aggregate value as the right-hand side of the assignment expression. This choice was made to facilitate performance; we have not yet been bitten by the semantics (which differ from C). We may in the future add a copy operator to construct "deep" copies.

From the perspective of C, the only novel operators are in and !in. These infix operators yield bool values depending on whether or not a given index is in a given table or set. For example, if sensitive_services is a set indexed by a single port, then

```
23/tcp in sensitive_services
```

returns true if the set has an element corresponding to an index of TCP port 23, false if it does not have such an element. Similarly, if RPC_okay is a set (or table) indexed by a source address, a destination address, and an RPC service number (a count), then

```
[src_addr, dst_addr, serv] in RPC_okay
```

yields true if the given ordered triple is present as an index into RPC_okay. The !in operator simply returns the boolean negation of the in operator.

Presently, indexing a table or set with a value that does not correspond to one of its elements leads to a run-time error, so such operations need to be preceded by `in` tests. We find this not entirely satisfying, and plan to add a mechanism for optionally specifying the action to take in such cases on a per-table basis.

Another use of the `in` and `!in` operators is for regular-expression pattern matching. For example,

```
filename in /rootkit-1\.[5-8]/
```

yields true if the value of the expression `filename` (which must have type `string`) matches any of `rootkit-1.5`, `rootkit-1.6`, `rootkit-1.7`, or `rootkit-1.8`.

Finally, `Bro` includes a number of predefined functions to perform operations not directly available in the language. Some of the more interesting: `fmt` provides *sprintf*-style formatting for use in printing or manipulating strings; `edit` returns a copy of a string that has been edited using the given editing characters (currently it only knows about single-character deletions); `mask_addr` takes an `addr` and returns another `addr` corresponding to its top n bits; `open` and `close` manipulate `files`; `network_time` returns the timestamp of the most recently received packet; `getenv` provides access to environment variables; `skip_further_processing` marks a connection as not requiring any further analysis; `set_record_packets` instructs the event engine whether or not to record any of a connection's future packets (though SYN/FIN/RST are always recorded); `set_contents_file` specifies a file to which Bro records the connection's reassembled byte stream; `system` executes a string as a Unix shell command; and `parse_ftp_port` takes an FTP "PORT" command and returns a `record` with the corresponding `addr` and `port`.

3.3 Variables

`Bro` supports two levels of scoping: local to a function or event handler, and global to the entire `Bro` script. Experience has already shown that we would benefit by adding a third, intermediate level of scoping, perhaps as part of a "module" or "object" facility, or even as simple as C's `static` scoping. Local variables are declared using the keyword `local`, and the declarations must come inside the body of a function or event handler. There is no requirement to declare variables at the beginning of the function. The scope of the variable ranges from the point of declaration to the end of the body. Global variables are declared using the keyword `global` and the declarations must come outside of any function bodies. For either type of declaration, the keyword can be replaced instead by `const`, which indicates that the variable's value is constant and cannot be changed.

Syntactically, a variable declaration looks like:

```
{class} {identifier} [':' {type}] ['=' {init}]
```

That is, a class (`local` or `global` scope, or the `const` qualifier), the name of the variable, an optional type, and an optional initialization value. One of the latter two must be specified. If both are, then naturally the type of the initialization much agree with the specified type. If only a type is given, then the variable is marked as not having a value yet; attempting to access its value before first setting it results in a run-time error.

If only an initializer is specified, then Bro infers the variable's type from the form of the initializer. This proves quite convenient, as does the ease with which complex tables and sets can be initialized. For example,

```
const IRC = { 6666/tcp, 6667/tcp, 6668/tcp };
```

infers a type of `set[port]` for `IRC`, while:

```
const ftp_serv = { ftp.lbl.gov, www.lbl.gov };
```

infers a type of `set[addr]` for `ftp_serv`, and initializes it to consist of the IP addresses for `ftp.lbl.gov` and `www.lbl.gov`, which, as noted above, may encompass more than two addresses. Bro infers compound indices by use of `[]` notation:

```
const allowed_services = {
   [ftp.lbl.gov, ftp], [ftp.lbl.gov, smtp],
   [ftp.lbl.gov, ident], [ftp.lbl.gov, 20/tcp],
   [www.lbl.gov, ftp], [www.lbl.gov, smtp],
   [www.lbl.gov, ident], [www.lbl.gov, 20/tcp],
   [nntp.lbl.gov, nntp]
};
```

results in `allowed_services` having type `set[addr, port]`. Here again, the *hostname* constants may result in more than one IP address. Any time Bro encounters a `list` of values in an initialization, it replicates the corresponding index. Furthermore, one can explicitly introduce lists in initializers by enclosing a series of values (with compatible types) in `[]`'s, so the above could be written:

```
const allowed_services: set[addr, port] = {
   [ftp.lbl.gov, [ftp, smtp, ident, 20/tcp]],
   [www.lbl.gov, [ftp, smtp, ident, 20/tcp]],
   [nntp.lbl.gov, nntp]
};
```

The only cost of such an initialization is that Bro's algorithm for inferring the variable's type from its initializer currently gets confused by these embedded lists, so the type now needs to be explicitly supplied, as shown.

In addition, any previously-defined global variable can be used in the initialization of a subsequent global variable. If the variable used in this fashion is a `set`, then its indices are expanded as if enclosed in their own list. So the above could be further simplified to:

```
const allowed_services: set[addr, port] = {
   [ftp_serv, [ftp, smtp, ident, 20/tcp]],
   [nntp.lbl.gov, nntp]
};
```

Initializing `table` values looks very similar, with the difference that a `table` initializer includes a *yield* value, too. For example:

```
global port_names = {
   [7/tcp] = "echo",
   [9/tcp] = "discard",
   [11/tcp] = "systat",
   ...
};
```

which infers a type of `table[port] of string`.

We find that these forms of initialization shorthand are much more than syntactic sugar. Because they allow us to define large tables in a succinct fashion, by referring to previously-defined objects and by concisely capturing forms of replication in the table, we can specify intricate policy relationships in a fashion that's both easy to write and easy to verify. Certainly, we would prefer the final definition of `allowed_services` above to any of its predecessors, in terms of knowing exactly what the set consists of.

Along with clarity and conciseness, another important advantage of `Bro`'s emphasis on tables and sets is speed. Consider the common problem of attempting to determine whether access is allowed to service `S` of host `H`. Rather than using (conceptually):

```
if ( H == ftp.lbl.gov || H == www.lbl.gov )
    if ( S == ftp || S == smtp || ... )
else if ( H == nntp.lbl.gov )
    if ( S == nntp )
...
```

we can simply use:

```
if ( [S, H] in allowed_services )
     ... it's okay ...
```

The `in` operation translates into a single hash table lookup, avoiding the cascaded `if`'s and clearly showing the intent of the test.

3.4 Statements

`Bro` currently supports only a modest group of statements, which we have so far found sufficient. Along with C-style `if` and `return` and expression evaluation, other statements are: `print` a list of expressions to a `file` (*stdout* by default); `log` a list of expressions; `add` an element to a `set`; `delete` an element from a `set` or a `table`; and `event`, which generates a new event.

In particular, the language does not support looping using a `for`-style construct. We are wary of loops in event handlers because they can lead to arbitrarily large processing delays, which in turn could lead to packet filter drops. We wanted to see whether we could still adequately express security policies in `Bro` without resorting to loops; if so, then we have some confidence that every event is handled quickly. So far, this experiment has been successful. Looping is still possible via recursion (either functions calling themselves, or event handlers generating their own events), but we have not found a need to resort to it.

Like in C, we can group sets of statements into *blocks* by enclosing them within {}'s. Function definitions look like:

```
function endpoint_id(h: addr, p: port): string
   {
   if ( p in port_names )
      return fmt("%s/%s", h, port_names[p]);
   else
      return fmt("%s/%d", h, p);
   }
```

Event handler definitions look the same except that `function` is replaced by `event` and they cannot specify a return type. See Appendix A for an example.

Functions are invoked the usual way, as expressions specified by the function's name followed by its arguments enclosed within parentheses. Events are generated in a similar fashion, except using the keyword `event` before the handler's name and argument list. Since events do not return values (they can't, since they are processed asynchronously), event generation is a statement in `Bro` and not an expression.

`Bro` also allows "global" statements that are not part of a function or event handler definition. These are executed after parsing the full script, and can of course invoke functions or generate events. The event engine also generates events during different phases of its operation: `bro_init` when it is about to begin operation, `bro_done` when it is about to terminate, and `bro_signal` when it receives a Unix signal.

One difference between defining functions and defining event handlers is that `Bro` allows multiple, different definitions for a given event handler. Whenever an event is generated, each instance of a handler is invoked in turn (in the order they appear in the script). So, for example, different (conceptual) modules can each define `bro_init` handlers to take care of their initialization. We find this considerably simplifies the task of creating modular sets of event handlers, but we anticipate requiring greater control in the future over the exact order in which `Bro` invokes multiple handlers.

4 Implementation issues

We implemented the Bro event engine and script interpreter in C++, currently about 27,000 lines. In this section we discuss some of the significant implementation decisions and tradeoffs. We defer to § 5 discussion of how Bro defends against attacks on the monitoring system, and postpone application-specific issues until § 6, as that discussion benefits from notions developed in § 5.

Use of C++. Our use of C++ was motivated by our successful experience with using it for implementing another event-oriented script interpreter, the Glish "software bus" [PS93]. For Bro, this has been a clear success. Class hierarchies map well to protocol layers, which then simplifies extending the event engine and script interpreter. We have not perceived any performance problems related to the choice of

C++; the choice of interpreting versus compiling (see below) is clearly a more dominant effect.

Single-threaded design. Since event handling lies at the heart of the system, it is natural to consider a multi-threaded design, with one thread per active event handler. We have so far resisted this approach, because of concerns that it could lead to subtle race conditions in `Bro` scripts.

An important consequence of a single-threaded design is that the system must be careful before initiating any activity that may potentially block waiting for a resource, leading to packet filter drops as the engine fails to consume incoming traffic. A particular concern is performing Domain Name System (DNS) lookups, which can take many seconds to complete or time out. Currently, Bro only performs such lookups when parsing its input file, but we want in the future to be able to make address and hostname translations on the fly, both to generate clearer messages, and to detect certain types of attacks. Consequently, Bro includes customized non-blocking DNS routines that perform DNS lookups asynchronously.

We may yet adopt a multi-threaded design. A more likely possibility is evolving Bro towards a distributed design, in which loosely-coupled, multiple Bro's on separate hosts monitor the same network link. Each Bro would watch a different type of traffic (e.g., HTTP or NFS) and communicate only at a high level, to convey current threat information.[3] A further extension of this notion is a more general distributed design, in which multiple Bro's watch multiple links, partitioning the monitoring workload; and also interacting with host-based agents. Others have recently also begun pursuing distributed architectures [Ci99, In99].

Managing timers. Bro uses numerous timers internally for operations such as timing out a connection establishment attempt. It sometimes has thousands of timers pending at a given moment. Consequently, it is important that timers be very lightweight: quick to set and to expire. Our initial implementation used a single priority heap, which we found attractive since insert and delete operations both require only $O(\log(N))$ time if the heap contains N elements. However, we found that when the heap grows quite large—such as during a hostile port scan that creates hundreds of new connections each second—then this overhead becomes significant. Consequently, we perceived a need to redesign timers to bring the overhead closer to $O(1)$. To achieve this, Bro now uses "calendar queues" instead [Br88].

A related issue with managing timers concerns exactly when to expire timers. Bro derives its notion of time from the timestamps provided by `libpcap` with each packet it delivers. Whenever this clock advances to a time later than the first element on the timer queue, Bro begins removing timers from the queue and processing their expiration, continuing until the queue is empty or its first element has a timestamp later than the current time. This approach is flawed, however, because in some situations—such as port scans—the event engine may find it needs to expire hundreds of timers that have suddenly become due, because the clock has advanced by a large amount due to a lull in incoming traffic. We avoid incurring a large processing spike in this situation by placing an upper limit k on the number of timers expired for any single advance of the clock. Doing so trades off timer exactness for spreading out load. Since we do not perceive a requirement for precise timers, this is an acceptable compromise.

Implementing regular expressions. Bro uses a custom regular-expression matching library, rather than reusing an existing one, for two reasons. First, we were unable to locate a high performance regular expression library with a redistribution license we found acceptable. In addition, intrusion detection pattern-matching differs from more typical text matching in two ways.

First, we want the ability to match text piecemeal, so we can feed the matcher new chunks of text as they arrive, without having to construct a copy of the entire string to match. Second, we anticipate matching sets of patterns and wanting to know which subset were matched by a given set of text, and for performance reasons we want to do the match with a single finite automaton rather than trying each pattern sequentially.

Since we had experience writing a high performance regular expression compiler [Pa96], and one that already supported the second of the above requirements, we decided to take that compiler and reimplement it in C++ to fit into Bro. Doing so was actually considerably easier than anticipated, and the only remaining piece for supporting the above requirements now is the corresponding Bro interpreter modifications.

One final facet of implementing regular expressions concerns caching: we employ a large number of patterns in our analysis (particularly for scanning interactive sessions, as discussed in § 6.5). These can take a large amount of CPU time (minutes) to compile, which is problematic when we want to start up the monitor quickly. Consequently, Bro maintains a cache of previously-compiled regular expressions, and if called upon to compile one that is already in the cache, simply loads the compiled version, taking very little time.

Interpreting vs. compiling. Presently, Bro interprets the policy script: that is, it parses the script into a tree of C++ objects that reflect an abstract syntax tree (AST), and then executes portions of the tree as needed by invoking a virtual evaluation method at the root of a given subtree. This method in turn recursively invokes evaluation methods on its children.

Such a design has the virtues of simplicity and ease of debugging, but comes at the cost of considerable overhead. From its inception, we intended `Bro` to readily admit compilation to a low-level virtual machine. Execution profiles of the current implementation indicate that the interpretive

[3] Some systems, such as DIDS and CSM, orchestrate multiple monitors watching multiple network links, in order to track users as they move from machine to machine [MHL94, WFP96]. These differ from what we envision for Bro in that they require each host in the network to run a monitor.

overhead is indeed significant, so we anticipate developing a compiler and optimizer. (The current interpreter does some simple constant folding and peephole optimization when building the AST, but no more.)

Using an interpreter also inadvertently introduced an implementation problem. By structuring the interpreter such that it recursively invokes virtual evaluation methods on the AST, we wind up intricately tying the `Bro` evaluation stack with the C++ run-time stack. Consequently, we cannot easily bundle up a `Bro` function's execution state into a closure to execute at some later point in time. Yet we would like to have this functionality, so `Bro` scripts have timers available to them; the semantics of these timers are to execute a block of statements when a timer expires, including access to the local variables of the function or event handler scheduling the timer. Therefore, adding timers to `Bro` will require at a minimum implementing an execution stack for `Bro` scripts separate from that of the interpreter.

Checkpointing. We run Bro continuously to monitor our DMZ network. However, we need to periodically checkpoint its operation, both to reclaim memory tied up in remembering state for long-dormant connections (because we don't yet have timers in the scripting language; see above), and to collect a snapshot for archiving and off-line analysis (discussed below).

Checkpointing is currently a three-stage process. First, we run a new instance of Bro that parses the policy script and resolves all of the DNS names in it. Because we have non-blocking DNS routines, Bro can perform a large number of lookups in parallel, as well as timing out lookup attempts whenever it chooses. For each lookup, it compares the results with any it may have previously cached and generates corresponding events (mapping valid, mapping unverified if it had to time out the lookup, or mapping changed). It then updates the DNS cache file and exits.

In the second stage, we run another instance of Bro, this time specifying that it should only consult the DNS cache and not perform lookups. Because it works directly out of the cache, it starts very quickly. After waiting a short interval, we then send a signal to the long-running Bro telling it to terminate. When it exits, the checkpointing is complete.

We find the checkpointing deficient in two ways. First, it would be simpler to coordinate a checkpoint if a new instance of Bro could directly signal an old instance to announce that it is ready to take over monitoring. Second, and more important, currently no state survives the checkpointing. In particular, if the older Bro has identified some suspect activity and is watching it particularly closely (say, by recording all of its packets), this information is lost when the new Bro takes over. Clearly, we need to fix this.

Off-line analysis. As mentioned above, one reason for checkpointing the system is to facilitate off-line analysis. The first step of this analysis is to copy the `libpcap` save file and any files generated by the policy script to an analysis machine. Our policy script generates six such files: a summary of all connection activity, including starting time, duration, size in each direction, protocol, IP addresses, connection state, and any additional information (such as username, when identified); a summary of the network interface and packet filter statistics; a list of all generated log messages; summaries of Finger and FTP commands; and a list of all unusual networking events.

Regarding this last, the event engine identifies more than 70 different types of unusual behavior, such as incorrect connection initiations and terminations, checksum errors, packet length mismatches, and protocol violations. For each, it generates a `conn_weird` or `net_weird` event, identifying the behavior with a predefined string. Our policy script uses a `table[string] of count` to map these strings to one of "ignore," "file," "log always," "log once per connection," and "log once per originating source address," meaning ignore the behavior entirely, record it to the anomaly file, log it (real-time notification) and record it to the file, and log it but only the first time it occurs for the given connection or the given source address. Some anomalies prove surprisingly common, and on a typical day the anomaly file contains several thousand entries, even though our script suppresses duplicate messages. (See § 7.3 below for further discussion of anomalies.)

All of the copied files thus form an archival record of the day's traffic. We keep these files indefinitely. They can prove invaluable when we discover a break-in that first occurred weeks or months in the past. In addition, once we have identified an attacking site, we can run it through the archive to find any other hosts it may have attacked that the monitoring failed to detect (for example, the attacker has obtained a list of passwords using a password-sniffer).

Finally, the off-line analysis generates a traffic summary highlighting the busiest hosts and giving the volume (number of connections and bytes transferred) due to different applications. As of this writing, on a typical day our site engages in about 1,200,000 connections transferring 40 GB of data. The great majority (75–80%) of the connections are HTTP; the highest byte volume comes from HTTP, FTP data, and sometimes the NFS network file system.

5 Attacks on the monitor

In this section we discuss the difficult problem of defending the monitor against attacks upon itself. We defer discussion of Bro's application-specific processing until after this section, because elements of that processing reflect attempts to defeat the types of attacks we describe here.

As discussed in § 1, we assume that such attackers have full access to the monitor's algorithms and source code; but also that they have control over only one of the two connection endpoints. In addition, we assume that the cracker does *not* have access to the `Bro` policy script, which each site will have customized, and should keep well protected.

While previous work has addressed the general problem of testing intrusion detection systems [PZCMO96], this work has focused on correctness of the system in terms of whether it does indeed recognize the attacks claimed. To our knowledge, the first discussion in the literature specifically aimed at the problem of attackers subverting a network intrusion detection system was the concurrent publication of the earlier version of this paper [Pa98] and that of Ptacek and Newsham [PN98].

The second of these is the more thorough, being completely devoted to the topic. The authors consider three types of attacks, "insertion," in which the attacker attempts to mislead the monitor into accepting traffic that the destination end-system rejects; "evasion," in which the monitor fails to accept traffic that the end-system does in fact accept; and denial-of-service, in which the attacker attempts to exploit a monitor's proactive mechanisms (such as terminating connections belonging to an apparent attack) in order to disrupt legitimate uses of the network.

For our purposes, however, we use a different attack taxonomy, because we focus on designing monitors to resist these attacks. We classify network monitor attacks into three categories: *overload*, *crash*, and *subterfuge*. The remainder of this section defines each category and briefly discusses the degree to which Bro meets that class of threat.

5.1 Overload attacks

We term an attack as an *overload* if the goal of the attack is to overburden the monitor to the point where it fails to keep up with the data stream it must process. The attack has two phases, the first in which the attacker drives the monitor to the point of overload, and the second in which the attacker attempts a network intrusion. The monitor would ordinarily detect this second phase, but fails to do so—or at least fails to do so with some non-negligible probability—because it is no longer tracking all of the data necessary to detect every current threat.

It is this last consideration, that the attack might still be detected because the monitor was not sufficiently overwhelmed, that complicates the use of overload attacks; so, in turn, this provides a defensive strategy, namely to leave some doubt as to the exact power and typical load of the monitor.

Another defensive strategy is for the monitor to *shed load* when it becomes unduly stressed (see [CT94] for a discussion of shedding load in a different context). For example, the monitor might decide to cease to capture HTTP packets, as these form a high proportion of the traffic. Of course, if the attacker knows the form of load-shedding used by the monitor, then they can exploit its consequent blindness and launch a now-undetected attack.

For Bro in particular, to develop an overload attack one might begin by inspecting Figure 1 to see how to increase the data flow. One step is to send packets that match the packet filter; another, packet streams that in turn generate events; and a third, events that lead to logging or recording to disk.

The first of these is particularly easy, because the `libpcap` filter used by Bro is fixed. One defense against it is to use a hardware platform with sufficient processing power to keep up with a high volume of filtered traffic, and it was this consideration that lead to our elaborating the goal of "no packet filter drops" in § 1. The second level of attack, causing the engine to generate a large volume of events, is a bit more difficult to achieve because Bro events are designed to be lightweight. It is only the events for which the policy specifies quite a bit of work that provide much leverage for an attack at this level, and we do *not* assume that the attacker has access to the policy scripts. This same consideration makes an attack at the final level—elevating the logging or recording rate—difficult, because the attacker does not necessarily know which events lead to logging.

Finally, to help defend against overload attacks, the event engine periodically generates a `net_stats_update` event. The value of this event gives the number of packets received, the number dropped by the packet filter due to insufficient buffer, and the number reported dropped by the network interface because the kernel failed to consume them quickly enough. Thus, `Bro` scripts at least have some basic information available to them to determine whether the monitor is becoming overloaded.

5.2 Crash attacks

Crash attacks aim to knock the monitor completely out of action by causing it to either fault or run out of resources. As with an overload attack, the crash attack has two phases, the first during which the attacker crashes the monitor, and the second during which they then proceed with an intrusion.

Crash attacks can be much more subtle than overload attacks, though. By careful source code analysis, it may be possible to find a series of packets, or even just one, that, when received by the monitor, causes it to fault due to a coding error. The effect can be immediate and violent.

We can perhaps defend against this form of crash attack by careful coding and testing. Another type of crash attack, harder to defend against, is one that causes the monitor to exhaust its available resources: dynamic memory or disk space. Even if the monitor has no memory leaks, it still needs to maintain state for any active traffic. Therefore, one attack is to create traffic that consumes a large amount of state. When `Bro` supports timers for policy scripts, this attack will become more difficult, because it will be harder to predict the necessary level of bogus traffic. Attacks on disk space are likewise difficult, unless one knows the available disk capacity. In addition, the monitor might continue to run even with no disk space available, sacrificing an archival record but still producing real-time notifications, so a disk space attack might fail to mask a follow-on attack.

Bro provides two features to aid with defending against crash attacks. First, the event engine maintains a "watchdog" timer that expires every T seconds. (This timer is not a Bro internal timer, but rather a Unix "alarm.") Upon expiration, the watchdog handler checks to see whether the event engine has failed to finish processing the packet (and subsequent events) it was working on T seconds before. If so, then the watchdog presumes that the engine is in some sort of processing jam (perhaps due to a coding error, perhaps due to excessive time spent managing overburdened resources), and terminates the monitor process (first logging this fact, of course, and generating a core image for later analysis).

This feature might not seem particularly useful, except for the fact that it is coupled with a second feature: the script that runs Bro also detects if it ever unduly exits, and, if so, logs this fact and executes a copy of `tcpdump` that records the same traffic that the monitor would have captured. Thus, crash attacks are (1) logged, and (2) do not allow a subsequent intrusion attempt to go unrecorded, only to evade real-time detection. However, there is a window of opportunity between the time when the Bro monitor crashes and when `tcpdump` runs. If an attacker can predict exactly when this window occurs, then they can still evade detection. But determining the window is difficult without knowledge of the exact configuration of the monitoring system. One way of closing this window is to employ a second, "shadow" monitoring machine that simply records to disk the same traffic as the Bro monitor inspects.

5.3 Subterfuge attacks

In a *subterfuge* attack, an attacker attempts to mislead the monitor as to the meaning of the traffic it analyzes. These attacks are particularly difficult to defend against, because (1) unlike overload and crash attacks, if successful they do not leave any traces that they have occurred, and (2) the attacks can be quite subtle. Access to the monitor's source code particularly aids with devising subterfuge attacks.

We briefly discussed an example of a subterfuge attack in § 3.1, in which the attacker sends text with an embedded NUL in the hope that the monitor will miss the text after the NUL. Another form of subterfuge attack is using fragmented IP datagrams in an attempt to elude monitors that fail to reassemble IP fragments (an attack well-known to the firewall community, and one we have increasingly detected in our ongoing operation of Bro). The key principle is to find a traffic pattern interpreted by the monitor in a different fashion than by the receiving endpoint, and then to leverage this into an insertion or evasion attack, as discussed above.

To thwart subterfuge attacks, as we developed Bro we attempted at each stage to analyze the explicit and implicit assumptions made by the system, and how, by violating them, an attack might successfully elude detection. This can be a difficult process, though, and we make no claims to have found them all! In the remainder of this section, we focus on subterfuge attacks on the integrity of the byte stream monitored for a TCP connection. Then, in § 6.5, we look at subterfuge attacks aimed at hiding keywords in interactive text.

To analyze a TCP connection at the application level requires extracting the payload data from each TCP packet and reassembling it into its proper sequence. We now consider a spectrum of approaches to this problem, ranging from simplest and easiest to defeat, to increasingly resilient.

Scanning the data in individual packets without remembering any connection state, while easiest, obviously suffers from major problems: any time the text of interest happens to straddle the boundary between the end of one packet and the beginning of the next, the text will go unobserved. Such a split can happen simply by accident, and certainly by malicious intent.

Some systems address this problem by remembering previously-seen text up to a certain degree (perhaps from the beginning of the current line). This approach fails as soon as a sequence "hole" appears: that is, any time a packet is missing—due to loss or out-of-order delivery—then the resulting discontinuity in the data stream again can mask the presence of key text that is only partially present.

The next step is to fully reassemble the TCP data stream, based on the sequence numbers associated with each packet. Doing so requires maintaining a list of contiguous data blocks received so far, and fitting the data from new packets into the blocks, merging now-adjacent blocks when possible. At any given moment, one can then scan the text from the beginning of the connection to the highest in-sequence byte received.

Unless we are careful, even keeping track of non-contiguous data blocks does not suffice to prevent a TCP subterfuge attack. The key observation is that an attacker can manipulate the packets their TCP sends so that the monitor sees a particular packet, but the endpoint does not. One way of doing so is to transmit the packet with an invalid TCP checksum. (This particular attack can be dealt with by checksumming every packet, and discarding those that fail; a monitor needs to do this anyway so that it correctly tracks the endpoint's state in the presence of honest data corruption errors, which are not particularly rare [Pa97a].) Another way is to launch the packet with an IP "Time To Live" (TTL) field sufficient to carry the packet past the monitoring point, but insufficient to carry it all the way to the endpoint. (If the site has a complex topology, it may be difficult for the monitor to detect this attack.) A third way becomes possible if the final path to the attacked endpoint happens to have a smaller Maximum Transmission Unit (MTU) than the Internet path from the attacker's host to the monitoring point. The attacker then sends a packet with a size exceeding this MTU and with the IP "Don't Fragment" header bit set. This packet will then transit past the monitoring point, but be discarded by the router at the point where the MTU narrows.

By manipulating packets in this fashion, an attacker can send innocuous text for the benefit of the monitor, such as "`USER nice`", and then retransmit (using the same sequence numbers) attack text ("`USER root`"), this time allowing the packets to traverse all the way to the endpoint. If the monitor simply discards retransmitted data without inspecting it, then it will mistakenly believe that the endpoint received the innocuous text, and fail to detect the attack.

Figure 2 illustrates this attack. Here, the attacker sends the text "`USER`" with an initial TTL of 20 hops, covering se-

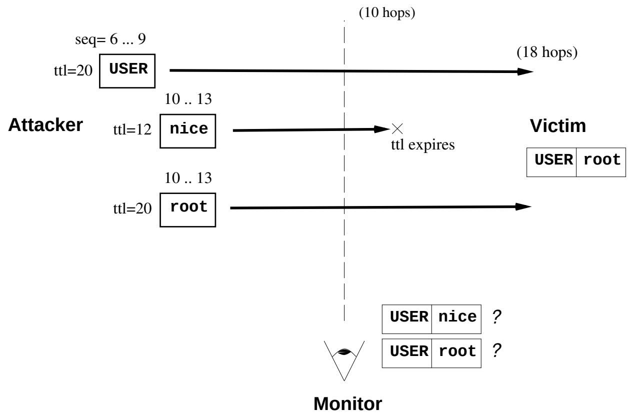

Figure 2: A TTL-based evasion attack on an intrusion detection system

quence numbers 6 through 9 in the TCP data stream. It is 18 hops to the victim and 10 hops to the monitor, so both see this text and accept it. The attacker next transmits the text "`nice`" covering the next consecutive span of the sequence space, 10 through 13, but with an initial TTL of only 12, which suffices for the packet to travel past the monitor, but not all the way to the victim. Hence, the monitor sees this text but the victim does not. The attacker the sends the text "`root`" with the same sequence numbers as "`nice`", but this time with enough TTL to reach the victim. The victim will thus only see the text "`USER`" followed by "`root`", while the monitor will see two versions of the text for sequence numbers 10 through 13, and will have to decide which to assume was also received by the victim (if, indeed, it even detects that the data stream includes an inconsistency, which requires extra work on the monitor's part). While in this case by inspecting the TTLs it *may* be able to determine which of the two versions the victim will have seen, there are many other ways (window checks, the MTU attack above, checksums, acknowledgement sequence number checks) of subtly affecting header fields such that the victim will reject one or the other of the two versions. Fundamentally, the monitor cannot confidently know which of the two versions to accept.

A partial defense against this attack is that when we observe a retransmitted packet (one with data that wholly or partially overlaps previously-seen data), we compare it with any data it overlaps, and sound an alarm (or, for Bro, generate an event) if they disagree. A properly-functioning TCP will always retransmit the same data as originally sent, so any disagreement is either due to a broken TCP, undetected data corruption (i.e., corruption the checksum fails to catch), or an attack.

We have argued that the monitor must retain a record of previously transmitted data, both in-sequence and out-of-sequence. The question now arises as to how long the monitor must keep this data around. If it keeps it for the lifetime of the connection, then it may require prodigious amounts of memory any time it happens upon a particularly large connection; these are not infrequent [Pa94]. We instead would like to discard data blocks as soon as possible, to reclaim the associated memory. Clearly, we cannot safely discard blocks above a sequencing hole, as we then lose the opportunity to scan the text that crosses from the sequence hole into the block. But we would like to determine when it is safe to discard in-sequence data.

Here we can make use of our assumption that the attacker controls only one of the connection endpoints. Suppose the stream of interest flows from host A to host B. If the attacker controls B, then they are unable to manipulate the data packets in a subterfuge attack, so we can safely discard the data once it is in-sequence and we have had an opportunity to analyze it. On the other hand, if they control A, then, from our assumption, any traffic we see from B reflects the correct functioning of its TCP (this assumes that we use anti-spoofing filters so that the attacker cannot forge bogus traffic purportedly coming from B). In particular, we can trust that if we see an acknowledgement from B for sequence number n, then indeed B has received all data in sequence up to n. At this point, B's TCP will deliver, or has already delivered, this data to the application running on B. In particular, B's TCP cannot accept any retransmitted data below sequence n, as it has already indicated it has no more interest in such data. Therefore, when the monitor sees an acknowledgement for n, it can safely release any memory associated with data up to sequence n.

While this defense works for detecting this general class of insertion attacks, it suffers from false positives, as discussed in § 7.3 below.

Finally, we note a general defense against certain types of subterfuge attacks, which we term "bifurcating analysis." The idea is that when the monitor cannot determine how an endpoint will interpret some network traffic (such as whether it will accept `USER nice` or `USER root`), it forms multiple threads of analysis, examining each of the possibilities. We note one example of doing so in § 6.5 below in our discussion of analyzing Telnet and Rlogin traffic.

6 Application-specific processing

We finish our overview of Bro with a discussion of the additional processing it does for the six applications it currently knows about: Finger, FTP, Portmapper, Ident, Telnet and Rlogin. Admittedly these are just a small portion of the different Internet applications used in attacks, and Bro's effectiveness will benefit greatly as more are added. Fortunately, we have in general found that the system meets our goal of extensibility (§ 1), and adding new applications to Bro is—other than the sometimes major headache of robustly interpreting the application protocol itself—quite straightforward, a matter of deriving a C++ class to analyze each connection's traffic, and devising a set of events corresponding to significant elements of the application.

6.1 Finger

The first of the applications is the Finger "User Information" service [Zi91]. Structurally, Finger is very simple: the connection originator sends a single line, terminated by a carriage-return line-feed, specifying the user for which they request information. An optional flag requests "full" (verbose) output. The responder returns whatever information it deems appropriate in multiple lines of text, after which it closes the connection.

Bro generates a `finger_request` event whenever it monitors a complete Finger request. A handler for this event looks like:

```
event finger_request(c: connection,
                     user: string, full: bool)
```

Our site's policy for Finger requests includes testing for possible buffer-overflow attacks and checking the user against a list of sensitive user ID's, such as privileged accounts. See Appendix A for a discussion of how the Finger analysis is integrated into Bro.

Bro generates an analogous `finger_reply` event:

```
event finger_reply(c: connection,
                   reply_line: string)
```

for each line of the reply from the Finger server.

A final note: if the event engine finds that the policy script does not define a `finger_request` or `finger_reply` handler, then it does not bother creating Finger-specific analyzers for new Finger connections. In general, the event engine tries to determine as early as possible whether the user has defined a particular handler, and, if not, avoids undertaking the work associated with generating the corresponding event.

6.2 FTP

The File Transfer Protocol [PR85] is much more complex than the Finger protocol; it also, however, is highly structured and easy to parse, so interpreting an FTP dialog is straight-forward.

For FTP requests, Bro parses each line sent by the connection originator into a command (first word) and an argument (the remainder), splitting the two at the first instance of whitespace it finds, and converting the command to uppercase (to circumvent problems such as a policy script testing for "store file" commands as `STOR` or `stor`, and an attacker instead sending `stOR`, which the remote FTP server will happily accept). It then generates an `ftp_request` event with these and the corresponding connection as arguments.

FTP replies begin with a status code (a number), followed by any accompanying text. Replies also can indicate whether they continue to another line. Accordingly, for each line of reply the event engine generates an `ftp_reply` with the code, the text, a flag indicating continuation, and the corresponding connection as arguments.

As far as the event engine is concerned, that's it—100 lines of straight-forward C++. What is interesting about FTP is that all the remaining work can be done in `Bro` (about 400 lines for our site). The `ftp_request` handler keeps track of distinct FTP sessions, pulls out usernames to test against a list of sensitive ID's (and to annotate the connection's general summary), and, for any FTP request that manipulates a file, checks for access to sensitive files. Some of these checks depend on context; for example, a guest (or "anonymous") user should not attempt to manipulate user-configuration files, while for other users doing so is fine.

One subtlety in the FTP analysis is being careful to maintain a notion of "current requests awaiting replies," rather than just "the most recently seen request." Doing so circumvents an attack in which the attacker pipelines multiple requests—rather than issuing a single request at a time and awaiting its response—and confuses the monitor as to which replies go with which requests.

A final analysis step for `ftp_request` events is to parse any `PORT` request to extract the hostname and TCP port associated with an upcoming transfer. (The FTP protocol uses multiple TCP connections, one for the control information such as user requests, and others, dynamically created, for each data transfer.) This is an important step, because it enables the script to tell which subsequent connections belong to this FTP session and which do not. A site's policy might allow FTP access to particular servers, but any other access to those servers merits an alarm; but without parsing the `PORT` request, it can be impossible to distinguish a legitimate FTP data transfer connection from an illicit, non-FTP connection. Consequently, the script keeps track of pending data transfer connections, and when it encounters them, marks them as `ftp-data` applications, even if they do not use the well-known port associated with such transfers (the standard does not require them to do so).

We also note that, in addition to correctly identifying FTP-related traffic, parsing `PORT` requests makes it possible to detect "FTP bounce" attacks. In these attacks, a malicious FTP client instructs an FTP server to open a data transfer connection not back to it, but to a third, victim site. The client can thus manipulate the server into uploading data to an arbitrary service on the victim site, or to effectively port-scan the victim site (which the client does by using multiple bogus `PORT` requests and observing the completion status of subsequent data-transfer requests). Our script flags `PORT` requests that attempt any redirection of the data transfer connection. Interestingly, we added this check mostly because it was easy to do so; months later, we monitored the first of several subsequent FTP bounce attacks. This form of serendipitous discovery of an unanticipated type of attack argues for employing a general principle of "sanity checking" the monitored traffic as much as possible. For a difficulty with this principle, however, see § 7.3.

For `ftp_reply` events, most of the work is simply formatting a succinct one-line summary of the request and its result for recording in the FTP activity log. In addition, an

FTP `PASV` request has a structure similar to a `PORT` request, except that the FTP server instead of the client determines the specifics of the subsequent data transfer connection. Consequently our script subjects `PASV` replies to the same analysis as `PORT` requests. Finally, there is nothing to prevent a *different* remote host from connecting to the data transfer port offered by a server via a `PASV` reply. It may be hard to see why this might actually occur, but putting in a test for it is simple (unfortunately, there are some false alarms due to multi-homed clients; we use heuristics to reduce these); and, indeed, several months after adding it, it triggered, due to an attacker using 3-way FTP as (evidently) a way to disguise their trail, another serendipitous result of the sanity-checking principle.

6.3 Portmapper

Many services based on Remote Procedure Call (RPC; defined in [Sr95a]) do not listen for requests on a "well-known" port, but rather pick an arbitrary port when initialized. They then register this port with a Portmapper service running on the same machine. Only the Portmapper needs to run on a well-known port; when clients want access to the service, they first contact the Portmapper, and it tells them which port they should then contact in order to reach the service. This second port may be for TCP or UDP access (depending on which of these the client requests from the Portmapper).

Thus, by monitoring Portmapper traffic, we can detect any attempted access to a number of sensitive RPC services, such as NFS and YP, except in cases where the attacker learns the port for those services some other way (e.g., port-scanning).

The Portmapper service is itself built on top of RPC, which in turn uses the XDR External Data Representation Standard [Sr95b]. Furthermore, one can use RPC on top of either TCP or UDP, and typically the Portmapper listens on both a well-known TCP port and a well-known UDP port (both are port 111). Consequently, adding Portmapper analysis to Bro required adding a generic RPC analyzer, TCP- and UDP-specific analyzers to unwrap the different ways in which RPCs are embedded in TCP and UDP packets, an XDR analyzer, and a Portmapper-specific analyzer.

This last generates six pairs of events, one for each request and reply for the six actions the Portmapper supports: a null call; add a binding between a service and a port; remove a binding; look up a binding; dump the entire table of bindings; and both look up a service and call it directly without requiring a second connection. (This last is a monitoring headache because it means any RPC service can potentially be accessed directly through a Portmapper connection.)

Our policy script for Portmapper traffic again is fairly large, more than 300 lines. Most of this concerns what Portmapper requests we allow between which pairs of hosts, particularly for NFS access.

6.4 Ident

The Identification Protocol ("ident") is used to query hosts for the user identity associated with an active connection [S-J93]. The request is of the form "*remote-port* **:** *local-port*". If host A sends such a request to the ident server on host B, then the request is asking for the identification of the user on host B who has a connection from host B's *remote-port* to host A's *local-port*. The reply identifies the operating system, perhaps a language encoding, and a username (or a "cookie" that does not directly reveal the username but can be used subsequently by an administrator of host B to identify the user).

Bro generates three events, `ident_request`, which identifies the *remote-port* and *local-port* in a request, `ident_reply`, which includes the username and the operating system, and `ident_error`, for when the remote server declares that the ident request was invalid. Our site's policy scripts check the username against a list of sensitive user ID's (such as "`rewt`", a name commonly used for backdoor "root" accounts), and annotates the corresponding connection record with the username.

6.5 Telnet and Rlogin

The final applications currently built into Bro are Telnet and Rlogin, services for remote interactive access [PR83a, Ka91]. There are several significant difficulties with monitoring interactive traffic. The first is that, unlike FTP, Telnet and Rlogin traffic is virtually unstructured. There are no nice "`USER xyz`" directives that make it trivial to identify the account associated with the activity; instead, one must employ a series of heuristics. (The Rlogin protocol includes a mechanism for specifying an initial username, but does not include a mechanism for indicating that the username was rejected, so the situation is virtually identical to that for Telnet in which the initial name is presumably the first text typed by the user.) This problem makes interactive traffic particularly susceptible to subterfuge attacks, since if the heuristics have holes, an attacker can slip through them undetected.

There are two parts to the analysis: determining usernames in a robust fashion, and scanning interactive sessions for strings reflecting questionable activity. We discuss each in turn. Because of the close similarities between analyzing Telnet and Rlogin sessions, Bro combines them into a generic "Login" analyzer, which is the term we use for both in the remainder of the section.

Recognizing the authentication dialog. The first facet of analyzing Login activity is to accurately track the initial authentication dialog and extract from it the usernames associated with both login failures and successes. Initially we attempted to build a state machine that would track the various authentication steps: waiting for the username, scanning the login prompt (this comes after the username, since the processing is line-oriented, and the full, newline-terminated prompt line does not appear until after the username has been entered), waiting for the password, scanning the password prompt, and then looking for an indication that the password was rejected (in which case the process repeats) or accepted. This approach, though, founders on the great variety of authentication dialogs used by different operating systems, some of which sometimes do not prompt for passwords, or re-prompt for passwords rather than login names after a password failure, or utilize two steps of password authentication, or extract usernames from environment variables, and so on. We instead use a simpler approach, based on associating particular strings (such as “Password:”) with particular information, and not attempting to track the authentication states explicitly. It works well, although not perfectly, and its workings are certainly easier to follow.

The Login analyzer generates `login_success` upon determining that a user has successfully authenticated, `login_failure` when a user's attempt to authenticate fails, `authentication_skipped` if it recognizes the authentication dialog as one specified by the policy script as not requiring further analysis, and `login_confused` if the analyzer becomes confused regarding the authentication dialog. (This last could, for example, trigger full-packet recording of the subsequent session, for later manual analysis.)

Type-ahead. A basic difficulty that complicates the analysis is type-ahead. We cannot rely on the most-recently entered string as corresponding to the current prompt line. Instead, we keep track of user input lines separately, and consume them as we observe different prompts. For example, if the analyzer scans “Password:”, then it associates with the prompt the first unread line in the user type-ahead buffer, and consumes that line. The hazard of this approach is if the login server ever flushes the type-ahead buffer (due to part of its authentication dialog, or upon an explicit signal from the user), then if the monitor misses this fact it will become out of sync. This opens the monitor to a subterfuge attack, in which an attacker passes off an innocuous string as a username, and the policy script in turn fails to recognize that the attacker in fact has authenticated as a privileged user. One fix to this problem—reflecting a strategy we adopt for the more general “keystroke editing” problem discussed below—is to test *both* usernames and passwords against any list of sensitive usernames, an example of the “bifurcation” approach discussed in § 5.3 above.

Unless we are careful, type-ahead also opens the door to another subterfuge attack. For example, an attacker can type-ahead the string “Password:”, which, when echoed by the login server, would be interpreted by the analyzer as corresponding to a password prompt, when in fact the dialog is in a different state. The analyzer defends against these attacks by checking each typed-ahead string against the different dialog strings it knows about, generating `possible_login_ploy` upon a match.

Keystroke editing. Usernames can also become disguised due to use of keystroke editing. For example, we would like to recognize that “rb<*DEL*>oot” does indeed correspond to a username of `root`, assuming that <*DEL*> is the single-character deletion operator. We find this assumption, however, problematic, since some systems use <*DEL*> and others use <*BS*>. We address this problem by applying both forms of editing to usernames, yielding possibly three different strings, each of which the script then assesses in turn. So, for example, the string “rob<*DEL*><*BS*><*BS*>ot” is tested both directly, as “ro<*BS*><*BS*>ot”, and as “root”. This is another example of using bifurcation to address analysis ambiguities.

Editing is not limited to deleting individual characters, however. Some systems support deleting entire words or lines; others allow access to previously-typed lines using an escape sequence. Word and line deletion do not allow an attacker to hide their username, if tests for sensitive usernames check for any embedded occurrence of the username within the input text. “History” access to previous text is more problematic; presently, the analyzer recognizes one operating system that supports this (VMS) and, for it only, expands the escape sequence into the text of the previous line.

Telnet options. The Telnet protocol supports a rich, complex mechanism for exchanging options between the client and server [PR83b] (there are more than 50 RFCs discussing different Telnet options). Unhappily, we cannot ignore the possible presence of these options in our analysis, because an attacker can embed one in the middle of text they transmit in order to disguise their intent—for example, “ro<*option*>ot”. The Telnet server will dutifully strip out the option before passing along the remaining text to the authentication system. We must do the same. On the other hand, parsing options also yields some benefits: we can detect connections that successfully negotiate to encrypt the data session, and skip subsequent analysis (rather than generating `login_confused` events), as well as analyzing options used for authentication (for example, Kerberos) and to transmit the user's environment variables (some systems use `$USER` as the default username during subsequent authentication).

Scanning the session contents. The last form of Login analysis, and in our experience far and away the most powerful for detecting break-ins, is looking at the contents of the lines sent by the user (`login_input_line` events) and by the remote server (`login_output_line`).

For input lines, some of the patterns we search for are the string “`eggdrop`” (an Internet Relay Chat tool that many attackers install upon a break-in), “`loadmodule`” and “`/bin/eject`” (used in buffer overflow attacks), and access to hidden directories with names like “`...`”. For output lines, we look for “`ls`” output showing setuid-root versions of command-line interpreters like *csh*, and strings like “`Jumping to address`” and “`Log started at`” which correspond to popular buffer-overflow and password sniffer tools, respectively.

6.6 Scan detection

We finish with a discussion of detecting port and address scanning. While not, strictly speaking, a form of application-specific processing, we have deferred discussion until now so we can refer to the previously-developed concepts of Bro language mechanisms and attacks on the monitor.

Scan detection is all done at the policy script level, so sites may of course tailor the detection however they wish. However, the basic approach we use is to maintain pairs of tables. For detecting address scanning, the first of the pair of tables, `distinct_peers`, is a `table[addr, addr] of bool`. We index it using the source and destination address of each newly-attempted connection. If the pair of addresses is not in the table, then we add them to the table, and increment `num_distinct_peers`, a corresponding `table[addr] of count`. This second table keeps track for each source address the number of distinct destination addresses to which it has attempted to connect. As that number crosses different thresholds, the script generates a series of real-time notifications indicating that an address scan is underway. It can of course take additional action, too, such as invoking via `system()` a script that removes the attacking site's connectivity to the local site (§ 8).

We detect port scanning in a similar fashion, using `distinct_ports`, a `table[addr, port] of bool` indexed by source address and destination port number, and a companion table `num_distinct_ports`, and again generate notifications as the distinct port count for a given address crosses different thresholds.

Note that this approach does not have any restrictions on the *order* in which addresses or ports are scanned, nor any particular requirements for how quickly they are scanned. By removing these sorts of restrictions, we can detect not only simple brute-force scans, but also some forms of "stealth" scanning, in which the scan is done slowly across a randomized list of addresses.

There are two problems with the approach, however. First, while the above steps do indeed detect scanning activities, they also generate false hits, because some services naturally result in a single source contacting multiple destination addresses (for example, a single client surfing multiple Web servers), or contacting multiple ports on the same remote host (an FTP server running on a non-standard port, so Bro does not know to track its PORT/PASV directives in order to associate connections on ephemeral ports with the FTP session). We can generally deal with this problem, however, by introducing some additional policy elements in our script, such as a list of services which we should ignore when updating the tables to reflect newly attempted connections.

The second difficulty concerns consumption of memory. Depending on a site's traffic patterns, the scan-detection tables can grow quite large. They can especially grow large if an attacker deliberately targets them as a way to attempt to compromise the monitor via an overload attack. One solution for addressing this problem would be to introduce the notion of associating timers with table elements. With such a mechanism, we could, for example, over time remove elements from `distinct_peers` and `num_distinct_peers`. Doing so, however, trades off recovering resources (and thus impairing an attacker's ability to launch an overload attack) with failing to detect slow stealth scans.

See § 7.1 below for a brief discussion of our experiences with scan-detection.

7 Status and Experiences

Bro has operated continuously since April 1996 as an integral part of our site's security system. It initially included only general TCP/IP analysis; as time permitted, we added the additional modules discussed in § 6, and we plan to add many more. In this section we sketch its current status and our experiences with operating it.

7.1 Implementation status

Presently, the implementation is about 27,000 lines of C++ and another 3,200 lines of `Bro` (about 2,700 lines of which are "boilerplate" not specific to our site). It runs under Digital Unix, FreeBSD, Linux, and Solaris operating systems. We use the *autoconf* auto-configuration tool as our main mechanism for abetting portability.

Bro is publicly available in source-code form (see *http://www-nrg.ee.lbl.gov/bro-info.html* for release information), though the current release is of "alpha" quality and includes only very limited documentation.

We hope that it will both benefit the community and in turn benefit from community efforts to enhance it. We have set up a mailing list for discussion—see the above Web page for subscription information.

In our on-going operations, Bro generates about 85 MB of connection summaries each day, and around 40 real-time notifications, though this figure varies greatly. While most of the notifications are innocuous (and if we were not also developers of the system, we would suppress these), we not infrequently also detect break-in attempts, and we average 4–5 address and port scans each day. Operation of the system has resulted so far in 4,000 email messages, 150 incident reports filed with CIAC and CERT, a number of accounts deactivated by other sites, and a couple incidents involving law enforcement.

7.2 Performance

The system generally operates without incurring any packet drops. The FDDI ring it runs on is fairly heavily utilized: a January, 1999 trace of a 14:30-15:30 busy hour reflects a traffic level of 11,900 packets/sec (34 Mbps) sustained for the full hour, with peaks of 18,000 packets/sec. However, the packet filter discards a great deal of this, both due to filtering primarily on SYN, FIN, or RST control bits, and because only about 20% of the traffic belongs to networks that we routinely monitor (the link is shared with a large neighbor institution).

To test the system under stress, we ran it for a 40 minute period without the "interesting networks" filter, resulting in a much higher fraction of traffic accepted by the packet filter. During this period, the filter accepted an average of 730 packets/sec, with peaks over 1,200 packets/sec, and without dropping any packets. The monitor system uses stripped disks and large BPF packet buffers [RLSSLW97] to improve performance.

7.3 Crud seen on a DMZ

An important and sobering aspect of our operational experience with Bro was the realization of how frequently, when monitoring a large volume of network traffic, legitimate (i.e., non-attacking) traffic exhibits abnormal behavior. We have observed all of the following:

- "Storms" of 10,000 FIN or RST packets, in which due to a protocol implementation error two hosts exchange FIN or RST packets extremely rapidly.
- Storms due to foggy days.[4]
- "Private" Internet addresses [Re96] leaking out into the public Internet. These addresses are inherently unroutable, and should never be used by a public Internet connection.
- SYN packets with the "Urgent" bit set. For SYN packets, setting "urgent" does not make any sense, since the connection is not yet established and hence cannot possibly have urgent data to send. Such packets are problematic, however, because some firewalls and monitors that are not carefully coded look for the beginning of connections to be indicated by the TCP "flags" field being equal to the SYN flag, rather than simply having the SYN flag set. When the Urgent bit is set, the field is no longer *equal* to the SYN flag.
- TCPs that when retransmitting data can send different data for the same sequence numbers as they sent the first time.
- TCPs that sometimes acknowledge receipt of data never sent.
- IP fragments in which the initial fragment is very small and the final fragment is large. Such fragments can be used to attempt to circumvent firewalls and monitors that do not do fragment reassembly.
- Fragments with the "Don't Fragment" bit set. While allowed by the IP standard, it is difficult to envision a situation in which such fragments can be legitimately constructed, yet we do indeed see them on clearly innocuous traffic.
- Overlapping fragments, in which the end of the first fragment is common with the beginning of the second. Such fragments are also used for "teardrop" denial-of-service attacks.
- Overlapping fragments for which the two fragments disagree on the contents of the overlapped region.

We recount these pathologies not simply because it is somewhat fascinating to see what a broad range of behavior we can observe in real network traffic; but also for the important reason that *many of these pathologies look very similar to genuine attacks*. Thus, the diversity of legitimate network traffic, including the implementation errors sometimes reflected within it, leads to a very real problem for intrusion detection, namely discerning in some circumstances between a true attack versus an innocuous implementation error. For example, it can be extremely difficult to discern between the "`USER nice`" / "`USER root`" subterfuge attack discussed in § 5.3, and a broken TCP implementation that sometimes retransmits different text than it originally sent. More generally, we cannot rely on "clearly" broken protocol behavior as definitely indicating an attack—it very well may simply reflect the operation of an incorrect implementation of that protocol.

We finish our discussion by noting a situation that does not reflect a protocol implementation error, but rather a common real-world problem, one that greatly complicates monitoring.

If ever a site's network topology includes multiple paths from the site to the remainder of the Internet, then the monitor may observe only one direction of a connection, because the traffic for the other direction transits an alternate route. We term this situation "split routing." (In the Internet at large, asymmetric routing is quite common, and so there are numerous monitoring points that suffer from split routing [Pa97b]. Individual sites, however, often have full control over whether they have multiple Internet connections. Some pursue multiple connections in order to provide redundancy in their connectivity to protect against occasional outages.)

Split routing can, of course, lead to the monitor missing attacks entirely because it never sees the traffic corresponding to the attack. Even if a site runs multiple monitors, one per Internet link, a subtle problem remains: the split routing can defeat precautions taken by the monitor because it can no longer assume that it sees traffic from at least one trustworthy endpoint for each connection. So, for example, the monitor loses the ability to determine when it can safely discard in-sequence data. Consequently, unless the multiple monitors communicate with one another concerning connection state, an attacker who discovers a split-route can exploit it to elude the monitor.

[4] One of the routers on our DMZ has a microwave link to a peer on the other side of San Francisco Bay. On foggy days, this link sometimes "flaps," leading to routing loops on the DMZ in which sets of packets enter routing loops and cross the DMZ 10's or 100's of times, until their TTLs expire.

Fortunately, split routing is at least easy to detect, because the monitor observes a connection transmitting unidirectional traffic without having first completed the initial three-way SYN handshake. Whenever Bro detects split routing, it generates an event announcing the problem.

8 Future directions

In addition to developing more application analysis modules, we see a number of avenues for future work. As discussed above, compiling `Bro` scripts and, especially, devising mechanisms to distribute monitoring across multiple hosts offer the promise of increasing monitoring performance. We are also very interested in extending BPF to better support monitoring, such as adding lookup tables and variable-length snapshots.

Another interesting direction is adding "teeth" to the monitoring in the form of actively terminating misbehaving connections by sending RST packets to their endpoints, or communicating with intermediary routers, as some commercially available monitors already do. We have implemented both of these for Bro and are now experimenting with their effectiveness. The ability to ask a router to drop traffic involving a particular address has already proven extremely useful, as it greatly limits the information that attackers can gather by scanning our site; once Bro recognizes a scan, it instructs the border router to drop any further traffic involving the given site. Some open issues with this form of reaction are the impact on router performance as the number of such filters increases, and attackers forging traffic from remote sites to mislead Bro into dropping them, as a form of denial-of-service attack.

More generally, however, we have found our fairly in-depth consideration of the problem of attacks on monitors (§ 5) sobering. Some forms of subterfuge attacks are extremely difficult to defend against, and we believe it is inevitable that attackers will devise and share toolkits for launching such attacks. This in turn suggests three important areas for research into intrusion detection: *(i)* further exploring the notion of "bifurcating analysis" discussed in § 5.3; *(ii)* studying the notion of traffic "normalizers" that remove ambiguities from traffic streams (one such normalizer is an "in-the-loop" monitor, one that must approve the forwarding of any packet it receives); and *(iii)* integrating into the system monitor "sensors" that run on the end hosts. Such sensors can analyze network traffic at a sufficiently high layer in their host's network stack where ambiguities about how the traffic will be interpreted have already been resolved. Our near-term research is focussing on the second of these, exploring the issues associated with building traffic normalizers.

9 Acknowledgements

We gratefully acknowledge Digital Equipment Corporation's Western Research Laboratory for contributing the Alpha system that made developing and operating Bro at high speeds possible. I would particularly like to thank Jeff Mogul, who was instrumental in arranging this through WRL's External Research Program.

Many thanks, too, to Craig Leres. Bro has benefited greatly from many discussions with him. Craig also wrote the calendar queue and non-blocking DNS routines discussed in § 4. Along with Craig Leres, I'd like to acknowledge the on-going feedback I receive from Craig Lant and Partha Banerjee on the daily operation of Bro, and their efforts at analyzing security incidents detected by Bro.

My appreciation to Scott Denton, John Antonishek, and many others for alpha-testing Bro and contributing portability fixes and other enhancements.

Finally, this work would not have been possible without the support and enthusiasm of Mark Rosenberg, Van Jacobson, Jim Rothfuss, Stu Loken and Dave Stevens—much appreciated!

A Example: tracking Finger traffic

In this appendix we give an overview of how the different elements of Bro come together for monitoring Finger traffic. For the event engine, we have a C++ class `FingerConn`, derived from the general-purpose `TCP_Connection` class. When Bro encounters a new connection with service port 79, it instantiates a corresponding `FingerConn` object, instead of a `TCP_Connection` object as it would for an unrecognized port.

`FingerConn` redefines the virtual function `BuildEndpoints`, which is invoked when a connection object is first created:

```
void FingerConn::BuildEndpoints()
   {
   resp = new TCP_EndpointLine(this, 1, 0, 1);
   orig = new TCP_EndpointLine(this, 0, 0, 1);
   }
```

Here, `resp`, corresponding to the responder (Finger server) side of the connection, is initialized to an ordinary `TCP_Endpoint` object, because Bro does not (presently) look inside Finger replies. But `orig`, the Finger client side, and `resp`, the responder (Finger server) side of the connection are both initialized to `TCP_EndpointLine` objects, which means Bro will track the contents of each side of the connection, and, furthermore, deliver the contents in a line-oriented fashion to `FingerConn`'s virtual `NewLine` function:

```
int FingerConn::NewLine(TCP_Endpoint* /* s */,
                   double /* t */, char* line)
   {
   line = skip_whitespace(line);

   // Check for /W.
   int is_long = (line[0] == '/' &&
                 toupper(line[1]) == 'W');
   if ( is_long )
```

```
      line = skip_whitespace(line+2);

   val_list* vl = new val_list;
   vl->append(BuildConnVal());
   vl->append(new StringVal(line));
   vl->append(new Val(is_long, TYPE_BOOL));

   mgr.QueueEvent(finger_request, vl);
   return 0;
   }
```

(For brevity, we show `NewLine` only for the `finger_request` case.) `NewLine` skips whitespace in the request, scans it for the "`/W`" indicator (which requests verbose Finger output), and moves past it if present. It then creates a `val_list` object, which holds a list of generic Bro `Val` objects. The first of these is assigned to a generic connection-identifier value (see below); the second, to a Bro `string` containing the Finger request, and the third to a `bool` indicating whether the request was verbose or not. The penultimate line queues a new `finger_request` event with the corresponding list of values as arguments; finally, `return 0` indicates that the `FingerConn` is all done with the memory associated with `line` (since `new StringVal(line)` made a copy of it), so that memory can be reclaimed by the caller.

The connection identifier discussed above is defined in Bro as a "`connection`" record:

```
type endpoint: record {
   size: count; state: count;
};
type connection: record {
   id: conn_id;
   orig: endpoint; resp: endpoint;
   start_time: time;
   duration: interval;
   service: string;
      # if empty, service not yet determined
   addl: string;
   hot: count;
      # how hot; 0 = don't know or not hot
};
```

The `id` field is a `conn_id` record, discussed in § 3.1. `orig` and `resp` correspond to the connection originator and responder, each a Bro `endpoint` record consisting of `size` (the number of bytes transferred by that endpoint so far) and `state`, the endpoint's TCP state (e.g., SYN sent, established, closed). This latter would be better expressed using an enumerated type (rather than a `count`), which we may add to Bro in the future.

The `start_time` field reflects when the connection's first packet was seen, and `duration` how long the connection has existed. `service` corresponds to the name of the service, or an empty string if it has not been identified. By convention, `addl` holds additional information associated with the connection; better than a `string` here would be some sort of union or generic type, if Bro supported such. Finally, by convention the policy script increments `hot` whenever it finds something potentially suspicious about the connection.

Here is the corresponding policy script:

```
global hot_names = { "root", "lp", "uucp" };
global finger_log =
   open(getenv("BRO_ID") == "" ?
        "finger.log" :
        fmt("finger.%s", getenv("BRO_ID")));

event finger_request(c:connection,
                     request: string,
                     full: bool)
   {
   if ( byte_len(request) > 80 ) {
      request = fmt("%s...",
                    sub_bytes(request, 1, 80));
      ++c$hot;
   }
   if ( request in hot_names )
      ++c$hot;

   local req = request == "" ?
      "ANY" : fmt("\"%s\"", request);
   if ( c$addl != "" )
      # This is an additional request.
      req = fmt("(%s)", req);
   if ( full )
      req = fmt("%s (/W)", req);

   local msg = fmt("%s > %s %s",
                   c$id$orig_h,
                   c$id$resp_h,
                   req);
   if ( c$hot > 0 )
      log fmt("finger: %s", msg);
   print finger_log,
         fmt("%.6f %s", c$start_time, msg);

   c$addl = c$addl == "" ?
            req : fmt("*%s, %s", c$addl, req);
   }
```

The global `hot_names` is a Bro `set` of `string`. In the next line, `finger_log` is initialized to a Bro `file`, either named "finger.log", or, if the `BRO_ID` environment variable is set, to a name derived from it using the built-in `fmt` function.

The `finger_request` event handler follows. It takes three arguments, corresponding to the values added to the `val_list` above. It first checks whether the request is excessively long, and, if so, truncates it and increments the `hot` field of the connection's information record. (The Bro built-in functions used here are named in terms of "bytes" rather than "string" because they make no assumptions about NUL-termination of their arguments; in particular, `byte_len` returns the length of its argument including a final NUL byte, if present.)

Next, the script checks whether the request corresponds to any of the entries in the `hot_names` set. If so, it again marks the connection as "hot."

We then initialize the local variable `req` to a quoted version of the request; or, if the request was empty (which in the Finger protocol indicates a request type of "ANY"), then it is changed to "ANY".

The event handler stores the Finger request in the connection record's `addl` field (see below), so the next line checks to see whether this field already contains a request. If so, then we are seeing multiple requests for a single Finger connection. This is not allowed by the Finger protocol, but that doesn't mean we won't see them! In particular, we might imagine a subterfuge attack in which an attacker queries an innocuous name in their first request, and a sensitive name in their second, and depending on how the finger server is written, it may well respond to both.[5] This script will still catch such use, since it fully processes each request; but it needs to be careful to keep the global state corresponding to the connection (in the `addl` field) complete. To do so, it marks additional requests by enclosing them in parentheses, and also prepends an asterisk to the entire `addl` field for each additional request, so that in later visual inspection of the Finger logs these requests immediately stand out.

The `msg` local variable holds the basic description of the Finger request. The `fmt` function knows to format the IP addresses `c$id$orig_h` and `c$id$resp_h` as "dotted quads."

Next, if the connection has been marked as "hot" (either just previously, or perhaps by a completely different event handler), then the script generates a real-time notification. In any case, it also records the request to the `finger_log` file. Finally, it updates the `addl` field to reflect the request (and to flag multiple requests, as discussed above).

Entries in the log file look like:

```
880988813.752829 171.64.15.68 >
                 128.3.253.104 "feng"
880991121.364126 131.243.168.28 >
                 130.132.143.23 "anlin"
880997120.932007 192.84.144.6 >
                 128.3.32.16 ALL
881000846.603872 128.3.9.45 >
                 146.165.7.14 ALL (/W)
881001601.958411 152.66.83.11 >
                 128.3.13.76 "davfor"
```

(though without the lines split after the ">").

The real-time notifications look quite similar, with the keyword "`finger:`" added to avoid ambiguity with other types of real-time notification.

References

[Ax99] AXENT Technologies, *Intruder Alert*, http://www.axent.com/product/smsbu/ITA/, 1999.

[Br88] R. Brown, "Calendar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, 31(10), pp. 1220-1227, Oct. 1988.

[Ci99] Cisco Systems, *NetRanger*, http://www.cisco.com/warp/public/cc/cisco/mkt/security/nranger/index.html, 1999.

[5] We do indeed see occasional multiple requests. So far, they have all appeared fully innocuous.

[CT94] C. Compton and D. Tennenhouse, "Collaborative Load Shedding for Media-Based Applications," *Proc. International Conference on Multimedia Computing and Systems*, Boston, MA, May. 1994.

[In99] Internet Security Systems, Inc., *RealSecure*™, http://www.iss.net/prod/rs.php3, 1999.

[JLM89] V. Jacobson, C. Leres, and S. McCanne, `tcpdump`, available via anonymous ftp to ftp.ee.lbl.gov, Jun. 1989.

[Ka91] B. Kantor, "BSD Rlogin," RFC 1282, Network Information Center, SRI International, Menlo Park, CA, Dec. 1991.

[MJ93] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Proc. 1993 Winter USENIX Conference*, San Diego, CA.

[MLJ94] S. McCanne, C. Leres and V. Jacobson, `libpcap`, available via anonymous ftp to ftp.ee.lbl.gov, 1994.

[MHL94] B. Mukherjee, L. Heberlein, and K. Levitt, "Network Intrusion Detection," *IEEE Network*, 8(3), pp. 26-41, May/Jun. 1994.

[Ne99] Network Flight Recorder, Inc., *Network Flight Recorder*, http://www.nfr.com, 1999.

[PS93] V. Paxson and C. Saltmarsh, "Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems," *Proc. 1993 Winter USENIX Conference*, San Diego, CA.

[Pa94] V. Paxson, "Empirically-Derived Analytic Models of Wide-Area TCP Connections," *IEEE/ACM Transactions on Networking*, 2(4), pp. 316-336, Aug. 1994.

[Pa96] V. Paxson, `flex`, available via anonymous ftp to ftp.ee.lbl.gov, Sep. 1996.

[Pa97a] V. Paxson, "End-to-End Internet Packet Dynamics," *Proc. SIGCOMM '97*, Cannes, France, Sep. 1997.

[Pa97b] V. Paxson, "End-to-End Routing Behavior in the Internet," *IEEE/ACM Transactions on Networking*, 5(5), pp. 601-615, Oct. 1997.

[Pa98] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," Proc. 7th USENIX Security Symposium, Jan. 1998.

[PR83a] J. Postel and J. Reynolds, "Telnet Protocol Specification," RFC 854, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PR83b] J. Postel and J. Reynolds, “Telnet Option Specifications,” RFC 855, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PR85] J. Postel and J. Reynolds, “File Transfer Protocol (FTP),” RFC 959, Network Information Center, SRI International, Menlo Park, CA, Oct. 1985.

[PN98] T. Ptacek and T. Newsham, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection,” Secure Networks, Inc., http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps, Jan. 1998.

[PZCMO96] N. Puketza, K. Zhang, M. Chung, B. Mukherjee and R. Olsson, “A Methodology for Testing Intrusion Detection Systems,” *IEEE Transactions on Software Engineering*, 22(10), pp. 719-729, Oct. 1996.

[RLSSLW97] M. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth and E. Wall, “Implementing a generalized tool for network monitoring,” *Proc. LISA '97*, USENIX 11th Systems Administration Conference, San Diego, Oct. 1997.

[Re96] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, “Address Allocation for Private Internets,” RFC 1918, Feb. 1996.

[Sr95a] R. Srinivasan, “RPC: Remote Procedure Call Protocol Specification Version 2,” RFC 1831, DDN Network Information Center, Aug. 1995.

[Sr95b] R. Srinivasan, “XDR: External Data Representation Standard,” RFC 1832, DDN Network Information Center, Aug. 1995.

[S-J93] M. St. Johns, “Identification Protocol,” RFC 1413, Network Information Center, SRI International, Menlo Park, CA, Feb. 1993.

[To99] Touch Technologies, Inc., *INTOUCH INSA*, http://www.ttisms.com/tti/nsa_www.html, 1999.

[WFP96] G. White, E. Fisch and U. Pooch, “Cooperating Security Managers: A Peer-Based Intrusion Detection System,” *IEEE Network*, 10(1), pp. 20-23, Jan./Feb. 1994.

[Zi91] D. Zimmerman, “The Finger User Information Protocol,” RFC 1288, Network Information Center, SRI International, Menlo Park, CA, Dec. 1991.