Detecting Backdoors

Yin Zhang and Vern Paxson[*]

Abstract

Backdoors are often installed by attackers who have compromised a system to ease their subsequent return to the system. We consider the problem of identifying a large class of backdoors, namely those providing interactive access on non-standard ports, by passively monitoring a site's Internet access link. We develop a general algorithm for detecting interactive traffic based on packet size and timing characteristics, and a set of protocol-specific algorithms that look for signatures distinctive to particular protocols. We evaluate the algorithms on large Internet access traces and find that they perform quite well. In addition, some of the algorithms are amenable to prefiltering using a stateless packet filter, which yields a major performance increase at little or no loss of accuracy. However, the success of the algorithms is tempered by the discovery that large sites have many users who routinely access what are in fact benign backdoors, such as servers running on non-standard ports not to hide, but for mundane administrative reasons. Hence, backdoor detection also requires a significant policy component for separating allowable backdoor access from surreptitious access.

1 Introduction

A *backdoor* is a mechanism surreptitiously introduced into a computer system to facilitate unauthorized access to the system. While backdoors can be installed for accessing a variety of services, of particular interest for network security are ones that provide interactive access. These are often installed by attackers who have compromised a system to ease their subsequent return to the system.

From a network monitoring perspective, such backdoors frequently run over protocols such as Telnet [PR83a], Rlogin [Ka91], or SSH [YKSRL99]. An example of a non-interactive backdoor would be an unauthorized SMTP server [Po82], say to facilitate relaying email spam; and one somewhat in between would be an FTP [PR85] backdoor used to provide access to illicit content such as pirated software, or a Napster server [NA99] run in violation of a site's policy.

Backdoors are, by design, difficult to detect. A common scheme for masking their presence is to run a server for a standard service such as Telnet, but on an undistinguished port rather than the well-known port associated with the service, or perhaps on a well-known port associated with a *different* service. In this paper we examine the problem of detecting backdoors, particularly interactive ones, by inspecting network traffic using an intrusion detection system (IDS), where we presume that there is a large volume of legitimate traffic which must be distinguished from the illegitimate traffic. To our knowledge, this problem has not been previously addressed in the literature.

Our general approach is to develop a set of algorithms for detecting different types of interactive traffic. These algorithms can then be applied to a traffic stream and whenever they detect interactive traffic using a non-standard service port, we have found some form of backdoor.

The rest of the paper is organized as follows. In § 2, we discuss the design considerations and examine the tradeoffs of different approaches. In § 3, we develop a general algorithm for detecting interactive traffic based on its timing characteristics, and in § 4 we present a number of protocol-specific algorithms. In § 5, we evaluate the algorithms using traces of Internet traffic. We summarize in § 6.

2 Design Space

A basic principle for backdoor detection is to find distinctive features indicative of the activity of interest, be it general interactive access, or use of a specific protocol such as SSH. The more powerful a feature is for distinguishing between genuine instances of the activity and false alarms, the better.

Candidates for such features include the specific contents of the data stream, the size and transmission rate of the packets in the stream, and their timing structure. This last is potentially very powerful for detecting interactive traffic: studies of Internet traffic have found that the interarrivals of user keystrokes have a striking distribution [DJCME92, PF95], namely a Pareto with infinite variance. There is also the possibility that a combination of features will prove to have greater distinctive power than any one feature by itself.

We now turn to a discussion of various tradeoffs that arise when considering how to develop detection algorithms.

2.1 Open vs. evasive attackers

In general, network intrusion detection becomes much more difficult when the attacker actively attempts to evade detection by the monitor [PN98, Pa98]. Much of the difficulty comes

[*]Y. Zhang is with the Computer Science Department, Cornell University, Ithaca, NY. Email: yzhang@cs.cornell.edu. V. Paxson is with the AT&T Center for Internet Research at ICSI, at the International Computer Science Institute in Berkeley, CA, and with the Lawrence Berkeley National Laboratory. Email: vern@aciri.org. This paper appears in the Proceedings of the 9th USENIX Security Symposium, Denver, Colorado, August 2000.

from the ability of attackers to exploit ambiguities in a traffic stream. From a monitoring perspective, heuristics might work well for "open" (non-evasive) attackers, but completely fail in the face of an actively evasive attacker.

While ideally any detection algorithms we develop would of course be resistant to evasive attackers, ensuring such robustness can sometimes be exceedingly difficult, and we proceed here on the assumption that there is utility in "raising the bar" even when a detection algorithm can be defeated by a sufficiently aggressive attacker. We further note that if an attacker fully controls *both* the remote and the local host, and in particular if they are patient and/or able to deploy arbitrary software, then all sorts of devious covert channels become possible[1] [Gl93], and backdoor detection becomes essentially hopeless. We do not attempt to address the problem of detecting covert channels.

Thus, we propose the algorithms in this paper not as solutions, but merely as waystations in the ongoing "arms race" between attackers and intrusion detection. One form of arms race we anticipate is particularly likely is between the developers of Napster [NA99] (and Gnutella [GN00]) and our corresponding detection algorithm. Napster has a history of sites attempting to control its use, and of users attempting to circumvent these restrictions [We00], and our algorithm gives sites a new tool for detecting surreptitious use of Napster.

2.2 Passive vs. active monitoring

One tradeoff is whether we only allow the monitor to perform passive monitoring, or if it can actively inject traffic into the network. Passive monitoring has the advantage that it cannot disturb the normal operation of the network. On the other hand, an active monitor could augment its backdoor detection by trying to connect to suspected backdoors in order to probe the server listening on the port to determine its service. However, doing so could in principle tip off the attacker as to the presence of the monitor and the discovery of the backdoor.

In this paper we confine ourselves to monitors that only use passive monitoring.

2.3 Content vs. timing

A natural approach for detecting connections to command shell servers is to monitor the keystrokes looking for common shell commands. Such a content-based approach has several drawbacks, however:

- Scanning each byte in each incoming packet is very expensive, especially if we must first reassemble TCP streams to defeat the sort of evasions characterized in [Pa98]. The intruder can then overload the monitor by generating a large amount of legitimate traffic.

[1]See [Ra00] for a discussion of experiences with running NFS over email by tunneling IP packets over messages delivered by SMTP.

- Many command shells allow the user to define aliases and editing characters, which can easily defeat this approach unless the monitor performs alias and editing expansion of the commands (such as also required for "bottleneck" analysis [LWWWG98]). Note that this problem can arise either inadvertently, because the attacker as a matter of course uses aliases or redefines the editing sequences, or deliberately, when the attacker is attempting to evade detection. The former case may be amenable to heuristic analysis; the latter likely is not.

- The intruder can easily evade the monitor by encrypting their content either through some application-level encryption method, or directly using encrypted protocols such as SSH.

In contrast, timing-based algorithms can be completely unperturbed by the use of encryption. However, timing information can become distorted due to clock skew, propagation delays, loss, and packetization variations. Making timing-based algorithm robust against such noise is challenging.

2.4 Filtering

An important factor for the success of real-time backdoor detection is filtering. The more traffic that can be discarded on a per-packet basis due to patterns in the TCP/IP headers, the better, as this can greatly reduce the processing load on the monitor. As we will see in subsequent sections, filtering can sometimes be highly effective in winnowing down a large traffic stream to just a few packets of interest.

However, there is clearly a tradeoff between reduced system load and lost information. First, if a monitor detects suspicious activity in a filtered stream, often the filtering has removed sufficient accompanying context that it becomes quite difficult to determine if the activity is indeed an attack. In addition, the existence of filtering criteria makes it easier for the attackers to evade detection by manipulating their traffic so that it no longer matches the filtering criteria. For example, an evasion against filtering based on packet size (see below) is to use a Telnet client modified to send a large number of do-nothing Telnet options along with each keystroke or line of input.

In addition, reliance on filtering can significantly magnify the problem of "chaff," i.e., attackers generating bogus traffic that matches the filtering criteria in order to overwhelm the monitor's analysis load, and/or to generate a huge number of false positives, in order to mask a true attack.

Three possible filtering criteria for backdoor detection are:

- *Packet size*. Keystroke packets are quite small. Even when entire lines of input are transferred using "line mode" [Bo90], packet payloads will tend to be much smaller than used for bulk-transfer protocols. Therefore, by filtering packets to only capture small packets, the monitor can significantly reduce its packet capture load.

- *Directionality*. In general, an interactive connection such as Telnet is initiated by the client rather than the server, unless the attacker sets up some sort of *callback* mechanism. This makes it possible to filter connections based on their directionality (inbound vs. outbound). If we are monitoring an Internet access link and are only interested in detecting backdoors at the local site, we can limit our monitoring to just inbound connections, which can significantly reduce the packet capture load (for example, by filtering out outbound Web surfing connections).

  Note that there is also a "cold start" problem when the monitor starts running and needs to analyze an existing traffic stream. In this case, it generally cannot determine whether the traffic was initiated inbound or outbound, and accordingly cannot filter it out.

- *Packet contents*. When we are interested in identifying specific interactive protocols, it is sometimes possible to filter incoming packets based on patterns specific to the protocol. An example is SSH, discussed in § 4.1 below.

2.5 Accuracy

As with intrusion detection in general, we face the problem of *false positives* (non-backdoor connections erroneously flagged as backdoors) and *false negatives* (backdoor connections the monitor fails to detect). The former can make the detection algorithm unusable, because it becomes impossible (or at least too tedious) to examine all of the alerts manually, and attackers can exploit the latter to evade the monitor.

We would of course like to have both the false positive rate and the false negative rate be as low as possible. But particularly for those of our algorithms that are based on overall traffic characteristics rather than sharp signatures, we frequently will have to choose tradeoffs between the two.

2.6 Responsiveness

Another important design parameter is the responsiveness of the detection algorithm. That is, after a backdoor connection starts, how long does it take for the monitor to detect the backdoor? Clearly, it is desirable to detect backdoors as quickly as possible, to enable taking additional actions such as recording related traffic or shutting down the connection. However, in many cases waiting longer allows the monitor to gather more information and consequently can detect backdoors more accurately, resulting in a tradeoff of responsiveness versus accuracy.

Another consideration related to responsiveness concerns the system resources consumed by the detection algorithm. If we want to detect backdoors quickly, then we must take care not to require more resources than the monitor can devote to detection over a short time period. On the other hand, if off-line analysis is sufficient, then we can use more resource-intensive algorithms.

3 A General Algorithm for Detecting Interactive Backdoors

In this section we present a general algorithm for detecting interactive backdoors based on keystroke characteristics. The algorithm incorporates three types of characteristics: directionality, packet sizes, and packet interarrival times. We also find we need to exclude excessively short flows (common in our traces due to the use of scanning by automated monitoring software), which do not provide enough traffic to analyze soundly. The criterion we use is to skip analysis of any flows comprised of fewer than 8 packets or lasting less than 2 seconds, where a flow is one direction of a bidirectional TCP connection.

3.1 Exploiting connection directionality

As noted above, an interactive connection is most likely initiated by the client, unless the server has some callback mechanism. Therefore, when looking for keystrokes we need only consider traffic sent by the initiator of a connection. However, if the monitor doesn't see the establishment of the connection, that is, the connection is a *partial* connection, there is no way to tell who is the actual initiator. In this case, we must consider both flows.

If we are monitoring an access link and are only interested in detecting backdoors within the local site, we can further exploit the connection directionality and ignore all outbound flows, even if the connection is partial.

3.2 Exploiting packet length characteristics

3.2.1 The size of keystroke packets

Keystroke packets are likely to be very small, even if sent in line mode, because most commands are short. To verify this assumption, we analyzed several Internet traffic traces with a total of 2.1 million Telnet and Rlogin client data packets. Of these, 79% carried a single byte, 97% carried 3 bytes or less, and 99.7% carried 20 bytes or less.

For a trace of SSH 1.x and 2.x connections (very heavily skewed towards 1.x), we found that 28% of the 150 K client data packets had length 20 or less. (Note that those SSH connections with predominantly big packets are likely to be file transfers.)

Consequently, we use 20 bytes as our cutoff for "small" packets.

3.2.2 Characterizing the frequency of small packets

Since most keystroke packets are quite small, we can exclude those connections that don't have enough small packets. More specifically, we can devise a metric to measure the frequency of small packets in a connection, which we then use to determine whether we should exclude the connection.

The simplest metric is the ratio of the number of small packets over the total number of packets, for a suitable definition of “small packet,” which per the previous section we define as 20 bytes or less of payload. Unfortunately, this metric doesn’t work well in practice. Although, as stated in the previous section, over 99.7% of keystrokes are very small, such statistics are based on a large number of connections. For a specific connection, we find that the ratio can be as low as 30–40%. Consequently, in order to prevent frequent false negatives, we have to choose a conservative threshold as low as 20–30%. But with such a low threshold, the metrics have little discriminating power and can introduce too many false positives.

To avoid such problems, we devised a metric Γ, defined in terms of S, the number of small packets, N, the total number of packets, and G, the number of gaps between small packets. A gap occurs any time two small packets are separated by at least one large packet. We then evaluate:

$$\Gamma = \frac{S - G - 1}{N}.$$

The intuition behind Γ is that consecutive small packets are strong indicators that a connection has interactive traffic. If the small packets are all spread throughout a connection, then we will have $G = S - 1$, so $\Gamma = 0$. If they are all grouped together, then $G = 0$ and Γ will reflect the relative proportion of small packets in the trace.

In our final algorithm, we set the threshold to $\Gamma = 0.2$.

3.3 Exploiting timing characteristics

As mentioned above, keystroke interarrival times come in a striking Pareto distribution, exhibiting a very broad range [PF95]. We can then exploit the tendency of machine-driven, non-interactive traffic to send packets back-to-back, with a very short interval between them, to discriminate non-interactive traffic from interactive. We do so by examining each pair of back-to-back small packet arrivals and computing the ratio α of how many of these interarrival times fall within the range 10 msec through 2 sec. (We need to take care not to include retransmitted packets in this computation.) The upper bound of 2 sec is fairly arbitrary; using 100 sec does not appreciably change the performance.

We then define a metric α to quantify how often the interarrival between two consecutive small packets falls in this range. In our final algorithm, we set the threshold to $\alpha = 0.2$.

It might appear that the criteria of $\Gamma = 0.2$ and $\alpha = 0.2$ are too lax, and singularly, they are; but jointly, they prove highly effective, as we show in § 5.7.

3.4 Making the algorithm run in real-time

In this section we discuss two considerations in using the algorithm in real-time. First, we observe that we can reduce the packet capture load a great deal by filtering on the data payload length of the packets to only capture small packets. `tcpdump` [JLM91] doesn’t actually have an easy way to specify a particular range of payload sizes, but the following will filter out all packets with more than 20 bytes of payload:

```
# (packet length -
#  ip header length -
#  tcp header length) <= 20.
# That is, data length <= 20.
(ip[2:2] - ((ip[0]&0x0f)<<2) -
 (tcp[12]>>2)) <= 20
```

where the bit-shifting is required to extract the IP and TCP header lengths, which can be variable length due to the presence of IP or TCP options.

Introducing filtering does not affect the evaluation of α for a flow, since α is only computed for packets that are consecutive in the TCP sequence space (§ 3.3). However, we must take care when evaluating Γ, since now that we only see small packets, we can’t accurately tell the total number of packets N transmitted by a given flow. To solve this problem, whenever we see a gap in the sequence number, we estimate the number of missing large packets in the gap as $\lceil \text{gap}/\text{LARGE\_PKT\_SIZE} \rceil$, where LARGE_PKT_SIZE is a guess at the most common size for full-sized packets. This size varies with path characteristics such as the Maximum Transmission Unit, and also depends on the particular TCP implementation, but as a rough approximation we simply use LARGE_PKT_SIZE $= 500$.

The other consideration for real-time detection concerns how quickly the algorithm can determine it has found a backdoor. For off-line analysis, it suffices to check whether a connection has backdoor characteristics when the connection terminates (or when the trace ends), and as we have defined Γ and α above, they are in terms of statistics computed over a connection’s total lifetime.

The simplest way to adapt the algorithm to run in real time is to reevaluate Γ and α on each incoming packet. Alternatively, we can have a timer for each connection and test the connection whenever the timer goes off. Unfortunately, neither approach works well in practice. The major problem is that when we classify a connection as a non-backdoor connection, we can’t just ignore the connection later on, because it’s hard to tell whether the connection is indeed a non-backdoor connection, or instead actually a backdoor connection with a preamble that has non-backdoor characteristics (such as the Telnet option negotiations that precede a Telnet login dialog). Consequently, we have to keep re-testing each non-backdoor connection, which is clearly very expensive.

We address this problem by exponentially backing off the reevaluation timer. We initially choose a small timeout value for the timer (30 seconds). Subsequently, whenever a connection appears to be a non-backdoor, we increase the timeout value by a factor of 1.5, which spreads the computational load over the lifetime of the connection.

4 Special-Purpose Detection Algorithms

In this section we explore algorithms that look for signatures reflecting the use of particular protocols. If we then find servers for those protocols running on ports other than their standard ones, such instances may indicate the presence of a backdoor.

Compared to the general-purpose detection algorithm, special-purpose algorithms can better benefit from protocol-specific information, and hence are likely to be more accurate or more efficient. On the other hand, relying on protocol-specific information can make the algorithm susceptible to evasion, if the attacker can perturb the signature.

There are two major applications for special-purpose detection algorithms. First, they can be used as baseline algorithms to evaluate the performance of the general-purpose algorithm described in § 3, allowing us to understand how much performance we lose by making the algorithm more general (and hence more difficult to evade). Second, the special-purpose algorithms themselves can be used either individually or in combination with the general-purpose algorithm to detect backdoors.

In the rest of this section, we introduce 15 algorithms for detecting various interactive protocols and the like. Based on different design purposes, we can divide these algorithms into the following two classes:

- Optimal algorithms are designed to identify backdoors as accurately as possible, without worrying about efficiency. Such algorithms are intended for use as baseline algorithms and for off-line analysis.
- Efficient algorithms incorporate protocol-specific filtering mechanisms into the optimal algorithms to reduce their expense, at the cost of a degree of accuracy. The tradeoff here varies a great deal—sometimes it is even possible to use a simple packet filter to achieve accuracy in the same league as for much more expensive algorithms (see § 4.1 below)—and the gain is algorithms efficient enough to use for real-time detection.

Table 1 summarizes the algorithms discussed in the rest of this section.

Backdoor type	Optimal algorithm	Efficient algorithm
SSH	**ssh-sig**, **ssh-len**	**ssh-sig-filter**
Rlogin	**rlogin-sig**	**rlogin-sig-filter**
Telnet	**telnet-sig**	**telnet-sig-filter**
FTP	**ftp-sig**	**ftp-sig-filter**
Root prompt	**root-sig**	**root-sig-filter**
Napster	**napster-sig**	**napster-sig-filter**
Gnutella	**gnutella-sig**	**gnutella-sig-filter**

Table 1: Summary of the special-purpose backdoor detection algorithms.

4.1 SSH

Secure Shell (SSH) encrypts transmitted content with strong cryptography. It is increasingly used for both interactive and bulk transfer traffic. While all in all its deployment represents a major advance for Internet security, it presents significant difficulties for content-based intrusion detection precisely because it renders the monitor blind to the specifics of each connection. It is thus particularly attractive for backdoor use.

Our first algorithm for detecting SSH, **ssh-sig**, uses the SSH version string as the signature for SSH. When an SSH connection has been established, both sides send an identifying string of the form “SSH-protoversion-softwareversion comments”, followed by carriage-return and newline (ASCII 13 and 10, respectively) [YKSRL99]. The maximum length of the string is 255 characters, including the carriage-return/newline. Version strings contain only printable characters, not including space or “`-`”.

Currently, the SSH protocol version is either “1.x” or “2.x”. Therefore, it suffices for **ssh-sig** to look for text “SSH-1.” or “SSH-2.” at the beginning of the first data packet sent in each direction of a connection.

We can replace **ssh-sig** with the following tcpdump filter (denoted as **ssh-sig-filter**) for very efficient detection:

```
# 1st 4 bytes are 'SSH-' and
# bytes 5 and 6 are '1.' or '2.'
tcp[(tcp[12]>>2):4] = 0x5353482D and
(tcp[((tcp[12]>>2)+4):2] = 0x312E or
 tcp[((tcp[12]>>2)+4):2] = 0x322E)
```

Our second detection algorithm, **ssh-len**, uses an implicit signature, the packet length, to detect SSH sessions. According to the SSH specification, SSH 1.x will (in the absence of TCP repacketization) generate packet payload sizes of the form $8k + 4$, that is, 4 more than a multiple of 8. SSH 2.x will generate payload sizes of length at least 16, and also a multiple of the cipher block size, which is a multiple of 8 for all of the ciphers of which we are aware. Therefore, for SSH, either most packets will have length $8k+4$, or most will have length $8k$. One deviation occurs with the initial version exchange, which does not conform with these rules.

In light of this pattern, **ssh-len** detects SSH as follows:

1. First test for an interactive connection using the timing-based algorithm (§ 3). If it is interactive, go to the next step, otherwise stop.
2. If the proportion of packets with length $8k + 4$ or the number of packets with length $8k$ exceeds a threshold, classify the connection as SSH.

We need to be careful when choosing the threshold, because packet retransmission and fragmentation can sometimes distort such characteristics. In our current implementation, we set the threshold to 75%.

4.2 Rlogin

Upon connection establishment, an Rlogin client sends four NUL-terminated strings to the server in the following format [Ka91]:

> `<NUL>`
> *client-user-name*`<NUL>`
> *server-user-name*`<NUL>`
> *terminal-type/speed*`<NUL>`

The server then returns a zero byte (NUL) to indicate that it has received these strings and is now in data transfer mode. Algorithm **rlogin-sig** attempts to detect Rlogin sessions using this negotiation as a signature. It first applies the following analysis to a connection:

- For the flow towards the initiator of a connection, check if the first byte is a NUL.
- For the flow sent by the initiator, keep testing each byte until one of the following events happens:
  - A gap in sequence number occurs;
  - four NUL's have been seen;
  - an empty string or a non-7-bit-ASCII byte is seen; or
  - the number of bytes we examined reaches a maximum bound (128 in the current algorithm).

If the above terminates by finding four NUL's, then we check to see whether the flow in the other direction begins with a non-NUL byte, or whether we found any empty strings or non-7-bit-ASCII bytes. If neither of these last two hold, then the connection is classified as an Rlogin connection.

We can combine **rlogin-sig** with the following tcpdump filter, resulting in a more efficient algorithm **rlogin-sig-filter**:

```
# last byte is 0 and data len != 0 and
# data length <= 128
(tcp[(ip[2:2]-((ip[0]&0x0f)<<2))-1] = 0)
and ((ip[2:2]-((ip[0]&0x0f)<<2)-
     (tcp[12]>>2)) != 0)
and ((ip[2:2]-((ip[0]&0x0f)<<2)-
     (tcp[12]>>2)) <= 128)
```

Note that **rlogin-sig** tests for whether the *last* byte in the packet is NUL, rather than the first byte. This is necessary because we find that clients tend to send their first NUL in its own packet, and the remainder of the prolog information in a second packet.

4.3 Telnet

The Telnet protocol [PR83a] includes a quite general mechanism for negotiating options [PR83b]. Since most Telnet sessions begin with a series of option negotiations, we can attempt to detect these, which have a distinct pattern, taking one of the following four 3-byte formats:

> `IAC WILL` *option-code*
> `IAC WON'T` *option-code*
> `IAC DO` *option-code*
> `IAC DON'T` *option-code*

The code values for `WILL`, `WON'T`, `DO`, `DON'T`, and `IAC` are 251, 252, 253, 254, and 255 respectively. Note that some options have parameters, and so can be longer than the above three bytes.

telnet-sig tests the first two bytes of each incoming packet to see if they match the beginning of any of the above. If a connection doesn't involve any option negotiation, we classify it as a non-Telnet connection. Otherwise, we test the following additional conditions:

- At least 75% of the bytes are 7-bit-ASCII.
- At least 50% of the lines are not longer than 80 bytes.

These aid in weeding out binary traffic that happens to match the option patterns above.

We can combine the following packet filter with **telnet-sig** to form a more efficient algorithm, **telnet-sig-filter**:

```
# 1st byte is <IAC> (0xff),
# 2nd byte is <251> - <254>
(tcp[(tcp[12]>>2):2] > 0xfffa) and
(tcp[(tcp[12]>>2):2] < 0xffff)
```

4.4 FTP

In this section we look at a somewhat different form of interactive protocol, the user control portion of the FTP file transfer protocol [PR85]. FTP is a request/reply protocol in which requests are sent in single, usually short, lines of ASCII text, and replies have a similar structure, but can be longer and multiline. Some FTP requests are sent in response to user activity, and accordingly have interactive-like timing. Others are generated mechanically by the FTP client, and arrive closely spaced.

Replies sent by FTP servers start with a status code (a number), followed by any accompanying text. For a day's worth of FTP activity between the Lawrence Berkeley National Laboratory and the rest of the Internet (7,229 connections), the distribution of the code in the first reply returned by the server is: code `220` ("ready for new user") seen 6,685 times; code `421` ("service not available") seen 535 times; code `226` ("closing data connection") seen 7 times; codes `426` ("connection closed") and `200` ("command okay") each seen once; no other codes seen.

Of these, if we miss a server that returns `421` we haven't actually missed anything significant, since the service is not available. All that really matters is detecting `220`, though we can include `421`, too, without too much extra effort.

For FTP server replies, the fourth byte is either a blank or a hyphen, the latter indicating a multi-line reply. Therefore, the **ftp-sig** algorithm looks in the first four bytes for either `220` or 421, followed by either a blank or a hyphen, as a signature for an FTP connection.

We can also compose **ftp-sig-filter**:

```
# 1st three bytes are ’220’,
# 4th byte is blank or hyphen
tcp[(tcp[12]>>2):4] = 0x3232302d or
tcp[(tcp[12]>>2):4] = 0x32323020
```

with a similar filter for 421.

One difficulty with this approach is that the same sort of status codes are used by the popular SMTP mail transfer protocol [Po82]. Code 220 corresponds to “service ready” and 421 to “service not available,” just as it does for FTP. This means that our algorithms for detecting FTP backdoors should work just as well for SMTP backdoors (which can actually be beneficial), which in § 5.5 we explore further.

4.5 Root Backdoor

From operational experience we have found that one particular type of backdoor installed by attackers is a Unix root shell, and the connection to it may not involve any Telnet option negotiation. For these, often the server starts by sending a packet with a payload of exactly two bytes: “#<*blank*>”, which corresponds to one of the forms of a Unix root shell prompt. This gives us a simple algorithm, **root-sig**, which attempts to detect root backdoors by looking for the two bytes in the first packet sent by the server side of a connection, and the corresponding **root-sig-filter**:

```
# look for ’# ’ in a packet with
# exactly 2 bytes of payload
tcp[(tcp[12]>>2):2] = 0x2320 and
(ip[2:2] - ((ip[0]&0x0f)<<2) -
(tcp[12]>>2)) == 2
```

which, given its conceptual simplicity, works surprisingly well (see § 5.6 below).

4.6 Napster

Napster is a distributed system by which users can share copies of music that has been digitized in MP3 format [NA99]. Users run a client that connects to napster.com servers for purposes of publishing the MP3’s that the user has made available to the public, and for searching for particular MP3’s available elsewhere in the distributed database. The server redirects the client to other clients that have the desired MP3 available, and the client then makes a direct connection to the source of the MP3, bypassing the server at this point.

Napster has proven controversial because often the MP3 trading is in violation of copyright laws, and also because MP3’s tend to be large files, so the enthusiasm of a site’s Napster users can consume considerable resources [NA00, Ha00]. Therefore, sites make efforts to control Napster traffic, for example by removing connectivity to the napster.com servers. Napster users have taken counter-measures to circumvent such blocking [We00], including configuring Napster servers to use non-standard ports for their communications. Open-source Napster clients are also available [GN99, ON00a], which will aid Napster users in modifying the client’s behavior to better circumvent detection.

Detecting Napster traffic is thus in many ways similar to detecting other backdoors, even though in this case the traffic does not reflect a security access violation, but rather a policy violation (authorization rather than authentication).

We focused on the problem of detecting the communication directly between Napster clients (used to transfer the actual MP3’s). One thought was to develop a generic MP3 detector, though our preliminary work on this has shown the problem to be somewhat difficult, as the format has a short, binary header that does not suggest a simple, distinct pattern to look for [Bo00].

The Napster client communication, however, has a quite distinctive signature [ON00b]. The communication begins with the string SEND or GET followed immediately by the name of the item (no intervening whitespace). Furthermore, we have found that the SEND or GET directive is sent by the Napster client in its own packet,[2] so our current version of **napster-sig** simply looks for either of these strings sent in their own packet and occurring at the beginning of a connection. **napster-sig-filter** does the same, but without the beginning-of-a-connection context:

```
# look for "SEND" or "GET" in a
# packet by itself (so payload of
# 4 or 3 bytes, respectively)
((ip[2:2] - ((ip[0]&0x0f)<<2) -
  (tcp[12]>>2)) = 4 and
 tcp[(tcp[12]>>2):4] = 0x53454e44) or
((ip[2:2] - ((ip[0]&0x0f)<<2) -
  (tcp[12]>>2)) = 3 and
 tcp[(tcp[12]>>2):2] = 0x4745 and
 tcp[(tcp[12]>>2)+2]=0x54)
```

4.7 Gnutella

Gnutella is a distribution system similar in spirit to Napster [GN00]. Its distinctive features are that it is fully open source, it can be used to exchange arbitrary files and not just MP3’s (although there are now Napster add-ons for doing this, too), and it has no centralized component—Gnutella clients simply need to know the name of another Gnutella client and they can participate in the distribution network. Consequently, Gnutella is likely to prove harder for sites to control than Napster.

In its current form, however, Gnutella is very easy to detect. Each Gnutella session begins with the connecting client transmitting:

[2] Clearly, this is very easy for the Napster client to change, and the corresponding change to make to our detector is looking for the absence of whitespace following the directive, which will address mistaking Napster GET’s for those used by HTTP.

```
    GNUTELLA CONNECT/<version><NL><NL>
```

and receiving in reply:

```
    GNUTELLA OK<NL><NL>
```

where <*NL*> is the newline character (ASCII 10).

Accordingly, **gnutella-sig** looks for the string "GNUTELLA<*blank*>" at the beginning of a connection.

The corresponding **gnutella-sig-filter** is:

```
  # look for "GNUTELLA " as first
  # 9 characters of payload
  tcp[(tcp[12]>>2):4] = 0x474e5554 and
  tcp[(4+(tcp[12]>>2)):4] = 0x454c4c41
  and tcp[8+(tcp[12]>>2)] = 0x20
```

5 Performance evaluation

In this section we evaluate the algorithms developed in § 3 and § 4. The evaluations were done by adding implementations of the algorithms to the Bro intrusion detection system [Pa98].

Our general framework for evaluation is as follows. To assess an algorithm's accuracy, we first run it against known interactive traffic of the particular type it is supposed to detect (Telnet, Rlogin, SSH; or, for the general algorithm, a combination of Telnet and Rlogin, since SSH traffic is sometimes bulk-transfer) and analyze how often it fails to flag a connection in the trace as interactive. This evaluates the *false negative* rate. We then run the algorithm against packet traces of a site's Internet traffic (these have high-volume protocols such as HTTP, NFS, and X11 removed, because otherwise we could not capture the traces reliably) to see which connections they mark as interactive, and then manually assess whether the connection does indeed appear to be interactive. This evaluates the *false positive* rate.

Note, we do not assess the Napster and Gnutella detectors, as the traces we use here were captured before those applications existed. However, our informal assessment based on correlating traffic to known Napster and Gnutella ports and services is that they work very well.

5.1 Trace description

We used four traces to evaluate the performance of the algorithms:

- `ssh.trace` (194MB, 380K packets, 905 connections), a half-hour snapshot of all the SSH connections seen late at night on the Internet access link (DMZ) of the University of California at Berkeley (UCB).

- `lbnl.mix1.trace` (54MB, 134K packets, 4.6K connections) and `lbnl.mix2.trace` (421MB, 863K packets, 14.7K connections). Each trace contains one hour of aggregate traffic collected at the DMZ of the Lawrence Berkeley National Laboratory (LBNL), the first in the middle of the night, the second in the middle of the afternoon. The traces have had high volume protocols (HTTP, SSH, NFS, X11, NNTP, FTP data) filtered out.

  Note that we might well apply such filtering for operational use, too, deciding to trade off missing backdoors on those ports for the reduced packet capture load.

- `lbnl.inter.trace` (389MB, 3.5M packets, 5.5K connections), one day's worth of Telnet and Rlogin traffic collected at LBNL.

5.2 Performance of SSH algorithms

We ran **ssh-sig** on trace `ssh.trace` to evaluate its false negative ratio. Clearly, **ssh-sig** only works when the beginning of a connection is present. Altogether, there are 546 complete SSH connections in `ssh.trace`, none of which is missed by **ssh-sig**. This demonstrates that the false negative ratio of **ssh-sig** is extremely low, which is to be expected since the presence of the signature is required by the specification.

We then ran **ssh-sig** on `lbnl.mix1.trace`, `lbnl.mix2.trace` and `lbnl.inter.trace` to evaluate its false positive ratio. Among the 16,938 complete non-SSH connections, none is mis-classified as SSH by **ssh-sig**. Therefore, the false positive ratio of **ssh-sig** is close to 0.

ssh-sig-filter has exactly the same good performance on the traces we have, which is not surprising, as the only apparent opportunity for error is unusual packetization splitting the SSH version text across multiple packets. In addition, the filtering gain is tremendous, because only those packets that contain the SSH version string need to be further processed. For `ssh.trace`, the algorithm needs only inspect 111 KB of packets rather than the 194 MB present in the entire trace.

The major limitation of **ssh-sig** and **ssh-sig-filter** is that they only work when the beginning of an SSH connection is present.

Since SSH can be used for both interactive traffic and bulk transfer, it is difficult to soundly evaluate the false negative ratio of **ssh-len**, which is designed to detect *interactive* SSH backdoors. Consequently, we only evaluate the false positive ratio here.

Again, we ran **ssh-len** on the three traces without ssh connections: `lbnl.mix1.trace`, `lbnl.mix2.trace` and `lbnl.inter.trace`. Among the 16,938 non-SSH connections, only 5 are classified as SSH by **ssh-len**, yielding a very low false positive rate.

Compared with **ssh-sig** and **ssh-sig-filter**, **ssh-len** does not require the presence of the beginning of a connection. However, it is less robust for SSH 1.x over highly lossy links, where two SSH blocks of length $8k + 4$ could be coalesced due to packet retransmission, resulting in a single packet of $8(k_1 + k_2 + 1)$ bytes. Consequently, we only use **ssh-len** when the beginning of a connection is missing.

5.3 Performance of Rlogin algorithms

Altogether there are 175 complete Rlogin connections in the traces, none of which is missed by **rlogin-sig**.

We begin with evaluating the false positive ratio of **rlogin-sig**. In the four traces, altogether there are 17,306 non-rlogin connections, none of which is mis-classified as an Rlogin connection. This means **rlogin-sig** also has an extremely low false positive ratio.

After adding filtering into **rlogin-sig**, we found that the false negative ratio remains the same (0/175). Meanwhile, the increase in the false positive ratio is marginal: altogether there are 4 out of 17,306 non-Rlogin connections that are mis-classified as Rlogin connections by **rlogin-sig-filter**.

The filtering gain of **rlogin-sig-filter** is significant. Among the 1 GB data we have in the four traces, only 16 MB data needs to be processed by **rlogin-sig**.

The major limitation of **rlogin-sig** and **rlogin-sig-filter** is similar to **ssh-sig**—they only work when the beginning of a connection is seen by the monitor.

5.4 Performance of Telnet algorithms

Again, we first evaluate the false negative ratio of algorithm **telnet-sig**. Unfortunately, it turns out that many Telnet connections in our traces are very short. For such short connections, **telnet-sig** fails because the connections do not include option negotiations. On the other hand, if a connection is that short, even if it is indeed a backdoor, it is not likely to cause significant damage.

To make the evaluation meaningful, we only consider those connections satisfying:

- the client sends at least two lines of data;
- the server sends at least one line of data; and
- the duration of the connection is at least 1 second.

After eliminating connections not satisfying these requirements, 1,526 Telnet connections remain, 18 of which are missed by **telnet-sig**. Further inspection shows that 17 out of the 18 involve the same public library catalog server, which performs passwordless logins without any option negotiation.

We further find that of the 12,708 non-Telnet connections in the traces, none is mis-classified as Telnet connections. This demonstrates that **telnet-sig** has a very low false positive ratio.

After adding filtering into **telnet-sig** to form algorithm **telnet-sig-filter**, the false positive and false negative ratios are unaffected for the traces we have studied. The filtering gain, however, is significant: **telnet-sig-filter** has to process less than 1.5 MB out of over 1 GB of packet data.

The major limitation of **telnet-sig** and **telnet-sig-filter** is similar to **ssh-sig** and **rlogin-sig**—they only work when the connection as seen by the monitor includes option negotiations, which tends to only occur at the beginning of a connection.

5.5 Performance of FTP algorithms

As noted in § 4.4, our FTP detection algorithm will also detect SMTP, so here we note this limitation and then treat the two protocols together.

We have altogether 5,629 FTP/SMTP sessions in which the server sent at least 4 bytes of data. Of these, 29 are missed by **ftp-sig**. Further inspection shows that these connections are almost all partial connections for which the initial dialog (which is far and away the most likely place for our signature to trigger) is missing. This demonstrates that **ftp-sig** has a low false negative ratio.

Among 20,135 non-FTP/SMTP connections, only one is classified as FTP/SMTP. Further inspection shows that this is actually an FTP server running via WinSock—so there is no false positive after all!

After adding filtering, **ftp-sig-filter** enjoys the same accuracy, as well as a terrific filtering gain: only 1.2 MB out of over 1 GB data need be processed by **ftp-sig-filter**.

Again, the limitation for **ftp-sig** and **ftp-sig-filter** is that, except for rare exceptions, they only work when the beginning of a connection is seen by the monitor.

5.6 Root shell algorithms

As far as we can tell, our traces do not include any root shells, so we cannot soundly evaluate the performance of **root-sig** and **root-sig-filter**. But see the next section for preliminary experiences indicating that they (**root-sig-filter**, in particular) are quite powerful.

5.7 Performance of the general detection algorithm

To assess the false negative ratio of the algorithm, we ran it on trace `lbnl.inter.trace`, which consists only of Telnet and Rlogin connections. Among the 150 complete Rlogin connections, 26 are missed by the algorithm. Further inspection shows that 23 are excessively short (less than 2 seconds in duration, or only one command executed), and the other 3 are user login failures. Among all 1,450 Telnet connections that are not excessively short, 22 are missed by the timing-based algorithm. Therefore, the false negative ratio is at least comparable to **telnet-sig**. Further inspection shows that the algorithm found all 18 connections missed by the **telnet-sig**, but 22 connections detected by **telnet-sig** are missed by the timing-based algorithm.

To evaluate the false positive ratio of the algorithm, we ran the algorithm on `lbnl.mix1.trace` and `lbnl.mix2.trace` with all the Telnet/Rlogin/FTP/SSH/SMTP connections filtered out. Among over 12,000 connections, the timing-based algorithm reported 57 backdoors. Further inspection shows that 45 are IMAP [Cr94] and POP [MR96] mail servers used interactively, and therefore are not in fact false positives.[3]

[3]The algorithm has also detected interactive SMTP sessions, nominally a

5.8 Experience with production use

We only recently begun operational deployment of the backdoor detection algorithms for production use on the LBNL DMZ. One of the most surprising (and, in retrospect, obvious) findings has been the large number of legitimate backdoors.

For example, when analyzing 20 minutes of traffic from the UCB DMZ (comprising 4.9 GB of data after filtering out the high volume traffic), the protocol-specific algorithms report 334 backdoors on non-standard ports. Of these, 326 are FTP servers on non-standard ports, 7 are interactive games, and the remaining one is a library card catalog server. In contrast, the timing-based algorithm reports 220 backdoors. From visual inspections of 75 of these, we found: 17 are interactive AOL sessions, 19 are interactive games, 14 are chat sessions, 3 are card catalog servers, 7 are FTP sessions, and we were unable to identify the other 15.

Running on the live traffic stream, the SSH detection algorithms have turned up SSH servers running on port 80 (nominally HTTP—the server ran on that port to provide tunneling through firewalls); port 110 (nominally POP); port 32 (used to run an older version of SSH than the one on port 22, due to compatibility problems); ports 44320–44327 (a NAT server with SSH access to the collection of hosts behind it via a number of different ports); as well as a host of variants of 22 (222, 922, 2222, . . .).

For production use it is unsafe to filter out the high-volume protocols. Running the signature-based tcpdump filters on full traffic streams does not present any performance problems when using a kernel-based packet filter, as the filters are highly selective. For the other protocol-specific detectors, it appears we can also run them on good-sized full traffic streams, as running all of them against a 10 GB trace only takes about 20 CPU minutes on a 400 MHz Pentium II.

We run all of the protocol-specific detectors daily against traces of LBNL traffic other than the high-volume ports. (We will shortly be configuring our monitor to run them in real-time.) We currently run with a set of five filters to remove legitimate backdoors: the NAT front-end mentioned above; two hosts that run a document upload service that triggers `ftp-sig` (the protocol is not FTP or SMTP, but has a similar structure); a host that runs a service on TCP port 497 that involves an exchange that looks like Telnet option negotiation (but isn't); and a popular FTP server that sometimes serves files with binary data that looks like embedded Telnet options.

The Napster and Gnutella detectors have become important tools in enforcing LBNL's appropriate use policy, and, for example, have detected a remote Napster server running on port 21 (FTP) in an apparent attempt to hide or circumvent a firewall.

The root backdoor filter, **root-sig-filter**, has uncovered root backdoors running on UCB traffic. However, these have not been in the form originally intended (in which the connection begins directly with "#<*blank*>"), which we know from experience are a rare, albeit striking, signature. Instead, because the filter version of the algorithm detects "#<*blank*>" *anywhere* in a connection, providing it is sent as a prompt (by itself with no newline), **root-sig-filter** is quite powerful at detecting both some transitions to root via the Unix *su* command, and sessions for which the prompt seen after the login prolog is indeed "#<*blank*>".

Part of the appeal of **root-sig-filter** is that it generates very few candidate connections, so even though its false hit rate on general traffic is fairly high, the connections it flags are not burdensome to check, and it is an exceptionally cheap algorithm in terms of computation.

We do not yet run the general algorithm operationally. As discussed above, it detects large numbers of interactive services, requiring time-consuming effort contacting the managers for the various machines to determine that in fact the backdoors are legitimate. But the potential of the approach seems clear already.

non-interactive protocol.

6 Summary

The problem of finding a backdoor connection in a flood of otherwise legitimate network traffic initially appears daunting. But because interactive traffic has characteristics quite different from most machine-driven traffic (smaller packet sizes, longer idle periods), it is possible to search efficiently for such traffic. We have presented a general algorithm for doing so, and also protocol-specific algorithms that look for signatures particular to different protocols, both of which we implemented in the Bro intrusion detection system.

One unexpected benefit of developing the protocol-specific algorithms was to realize how it is frequently possible to fingerprint a particular application protocol by unique or nearly unique text it includes. This lead to the developement of successful algorithms for Napster and Gnutella, which can be important to detect given that their use sometimes violates a site's policy, and that their users often attempt to evade detection.

The algorithms are frequently amenable to prefiltering in which a stateless packet filter discards nearly all of the traffic stream before it is even considered by the algorithm. Such filtering yields major performance increases in terms of reduced CPU processing, for little or sometimes no decrease in accuracy. A related line of future work that may prove fruitful is to explore the possibility of combining the general algorithm with the protocol-specific algorithms, which is likely to yield better accuracy.

While the algorithms work very well, a major stumbling block we failed to anticipate is the large number of legitimate "backdoors" that users routinely access. These are not backdoors in the surreptitious sense, but only in the more general sense of standard protocols being run on non-standard ports. We have recently begun using the algorithms operationally, which will necessitate both the development of refined security policies addressing the many legitimate backdoors, and honing our algorithms as a mechanistic way to eliminate certain classes of benign backdoors. But even given these hurdles, we find the utility of the detection algorithms clear and compelling, and a natural next step is to now investigate their application to detecting custom backdoor protocols such as LOKI [da97] and Back Orifice [CERT98].

7 Acknowledgments

We would like to thank Ken Lindahl and Cliff Frost for their greatly appreciated help with gaining research access to UCB's traffic, and Tara Whalen and the anonymous reviewers for their feedback on the work and its presentation.

References

[Bo90] D. Borman, "Telnet Linemode Option," RFC 1184, Network Information Center, SRI International, Menlo Park, CA, Oct. 1990.

[Bo00] G. Bouvigne, "MPEG Audio Layer I/II/III frame header," http://www.mp3-tech.org/programmer/frame_header.html, 2000.

[CERT98] CERT Vulnerability Note VN-98.07, http://www.cert.org/vul_notes/VN-98.07.backorifice.html, Oct 1998.

[Cr94] M. Crispin, "Internet Message Access Protocol - Version 4," RFC 1730, Network Information Center, DDN Network Information Center, Dec. 1994.

[da97] daemon9 `route@infonexus.com`, "LOKI2 (the implementation)," *Phrack Magazine*, 7(51), Sep.01, 1997. http://www.infowar.com/iwftp/phrack/Phrack51/P51-06.txt .

[DJCME92] P. Danzig, S. Jamin, R. Cáceres, D. Mitzel, and D. Estrin, "An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations," *Internetworking: Research and Experience*, 3(1), pp. 1-26, 1992.

[Gl93] V. Gligor, "A Guide to Understanding Covert Channel Analysis of Trusted Systems," NCSC-TG-030, version 1, http://www.radium.ncsc.mil/tpep/lib-rary/rainbow/NCSC-TG-030.html, National Computer Security Center, Nov. 1993.

[GN99] Gnapster, http://www.faradic.net/j̃asta/gnapster.html, 1999.

[GN00] Gnutella, http://gnutella.wego.com, 2000.

[Ha00] J. Harrow, "The Consumer Internet Steamroller," *The Rapidly Changing Face of Computing*, http://www.compaq.com/rcfoc/20000417.html#_Toc480185377, April, 2000.

[JLM91] V. Jacobson, C. Leres, and S. McCanne, "tcpdump," ftp://ftp.ee.lbl.gov/tcpdump.tar.Z, 1991.

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

[LWWWG98] R. Lippmann, D. Wyschogrod, S. Webster, D. Weber, and S. Gorton, "Using Bottleneck Verification to Find Novel New Attacks with a Low False Alarm Rate," Proc. Recent Advances in Intrusion Detection, Sept. 1998; http://www.zurich.ibm.com/˜dac/Prog_RAID98/Talks.html#Lippmann_21 .

[MR96] J. Myers and M. Rose, "Post Office Protocol - Version 3," RFC 1939, Network Information Center, DDN Network Information Center, May 1996.

[NA99] Napster, http://www.napster.com, 1999.

[NA00] Napster (Press Room), http://www.napster.com/press.html, 2000.

[ON00b] "Napster protocol specification," http://opennap.sourceforge.net/napster.txt, June 2000.

[ON00a] OpenNap, http://opennap.sourceforge.net, 2000.

[PF95] V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking*, 3(3), pp. 226-244, June 1995.

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

[Po82] J. Postel, "Simple Mail Transfer Protocol," RFC 821, Network Information Center, SRI International, Menlo Park, CA, Aug. 1982.

[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.

[Ra00] M. Ranum. "RE: Bypassing firewall," mailing list firewall-wizards@nfr.net, Feb. 1, 2000.

[We00] D. Weekly, "How to get around a Napster blockade," http://david.weekly.org/code/napster-proxy.php3, 2000.

[YKSRL99] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen, "SSH Transport Layer Protocol," Internet Draft, draft-ietf-secsh-transport-07.txt, May 2000.