CERIAS Tech Report 2004-36

THE SESSION TOKEN PROTOCOL FOR FORENSICS AND TRACEBACK

by Brian Carrier and Clay Shields

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

The Session Token Protocol for Forensics and Traceback

BRIAN CARRIER
Purdue University
and
CLAY SHIELDS
Georgetown University

In this paper we present the Session Token Protocol (STOP), a new protocol that can assist in the forensic analysis of a computer involved in malicious network activity. It has been designed to help automate the process of tracing attackers who log on to a series of hosts to hide their identity. STOP utilizes the Identification Protocol infrastructure, improving both its capabilities and user privacy. On request, the STOP protocol saves user-level and application-level data associated with a particular TCP connection and returns a random token specifically related to that session. The saved data are not revealed to the requester unless the token is returned to the local administrator, who verifies the legitimacy of the need for the release of information. The protocol supports recursive traceback requests to gather information about the entire path of a connection. This allows an incident investigator to trace attackers to their home systems, but does not violate the privacy of normal users. This paper details the new protocol and presents implementation and performance results.

Categories and Subject Descriptors:

General Terms: Design, Performance, Security

Additional Key Words and Phrases: Digital forensics, digital investigations, TCP traceback, privacy, auditing and intrusion detection

1. INTRODUCTION

Many times, attackers log on to a series of compromised hosts before they attack their target. This is shown in Figure 1, where H_i, $0 \leq i \leq n$, is a set of hosts, and there is a connection C_i between hosts H_i and H_{i+1} if there exists an active

Portions of this work appeared in preliminary form elsewhere [Carrier and Shields 2002]. This work was supported by the National Science Foundation under ANI-0296194, and by the Center for Education and Research in Information Assurance and Security (CERIAS).
Authors' addresses: Brian Carrier, Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University, West Lafayette, IN 47907; email: carrier@cerias.purdue.edu; Clay Shields, Department of Computer Science, Georgetown University, Washington, D.C. 20057; email: clay@cs.georgetown.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2004 ACM 1094-9224/04/0800-0333 $5.00

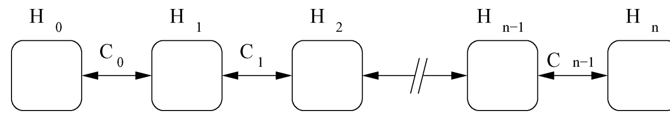

Fig. 1. Connection chain example between H_0 and H_n.

TCP session between them. A *connection chain*, $\mathcal{C}$, between hosts H_0 and H_n is the set of connections C_i, where $0 \leq i < n$.

Use of this technique complicates the determination of the attacker's location, and is commonly called stone stepping [Zhang and Paxson 2000]. The first step in determining an attacker's location is to contact the previous host, if discernible, and ask the administrator to investigate his or her own system. To identify the exact origin of the attack, this process continues recursively across all hosts on the chain. However, complete forensic investigations are expensive, and at any step of the chain, administrator may lack the skill, resources, knowledge, trust, or data to continue the investigation.

In this paper we present a protocol named the Session Token Protocol (STOP) [Carrier and Shields 2002; Carrier 2001], based on the Identification Protocol (IDENT) [Johns 1993], which is designed to help automate forensics investigations that determine attacker location while protecting user privacy. It allows individual hosts or firewalls to request data about inbound and outbound TCP connections. A daemon on the host that receives these requests can save application-level data about the process and user that opened the TCP connection, and can send recursive traceback requests to identify previous hosts. At each stage, only a hashed token is returned so that at no point in the protocol does the requester ever directly learn user or process data. Instead, the requester must contact the system administrator to redeem the token for the saved information. While STOP is limited by deployment issues and limitations in operating systems and will not always identify the complete connection chain, it can significantly reduce the amount of expensive forensic investigation needed to locate the source of an attack.

In Section 2, we describe related traceback work and the original IDENT. Section 3 describes the design goals and specification of the new protocol, along with examples of protocol operation. Section 4 provides a description of the Linux, OpenBSD, and Solaris implementations, and is followed in Section 5 by performance analysis of the implementations and in Section 6 by a description of STOP's limitations.

2. BACKGROUND

There are currently two major categories of network traceback. The one that has received more research attention is the problem of IP traceback, which pursues the goal of locating the source of spoofed IP packets. The relative wealth of research in this area was motivated by the problem of determining the source of denial-of-service (DoS) attacks, which tend to use spoofed packets. The other area of research is locating the source of an attack hidden by the use of a connection chain. The use of TCP connections generally precludes the use of spoofed packets, making the two areas discrete but complimentary.

Section 2.1 briefly describes work on IP traceback, while Sections 2.2 and 2.3 describe the approaches to connection-chain traceback that have been attempted. The IDENT is described in Section 2.4, as STOP is designed to inter-operate and be backwards compatible with it.

2.1 IP Traceback

IP traceback work attempts to locate the source of spoofed IP packets. The ideal solution to locating the source of spoofed IP packets would be to prevent them from being sent, so that the source address was always correct. Universal ingress filtering would accomplish this [Ferguson and Senie 1998, 2000], but such filtering is not as widespread as necessary. Other work has examined placing filters in a few key points in the network to limit IP spoofing [Park and Lee 2001b] or using automated methods to locate the path by which packets are arriving and placing filters along those paths [Yaar et al. 2003; Ioannidis and Bellovin 2002], but any common implementation also lies in the future.

A solution that has been proposed in a number of forms as a solution to distributed denial-of-service (DDoS) attacks is to probabilistically add markings to IP packets [Savage et al. 2000; Song and Perrig 2001; Park and Lee 2001a; Dean et al. 2001; Doeppner et al. 2000; Adler 2002]. Because of the limited space available to insert information into the packet headers, these schemes generally require that a large number of packets be sent along the same set of paths to ensure precision. Other solutions to DDoS attacks rely on routers counting an abnormal number of packets at an interface to trace the source of those packets [Cisco Systems Inc. 2003]. This method is also imprecise and inaccurate without sufficient attack traffic. Another method of tracing continuous streams of attack traffic involves launching counter-attacks, which is of dubious legality [Burch and Cheswick 2000]. Techniques that create overlay networks across which packets can be tracked have also been proposed [Stone 2000; Chang et al. 1999]. All of the solutions are ineffective against reflector attacks [Paxson 2001], in which the attack spoofs the victim's IP address in a request to a noncompromised system, which responds to the victim according to some common protocol, such as DNS.

A variation of a probabilistic approach that can be somewhat effective against reflector attacks is Itrace [Bellovin 2000]. Modifications to this scheme were proposed to improve performance with fewer packets [Mankin et al. 2001], and to allow traceback of reflector attacks by occasionally sending ICMP packets to the source of the selected packet, rather than the destination [Barros 2000]. The probabilistic nature of the scheme still requires a large flow of traffic to allow source identification.

A proposed solution that allows location of a single spoofed packet is called SPIE [Snoeren et al. 2001]. Because the memory of the devices used in SPIE is expensive and limited, the information is only available for a short time before being overwritten, unless it is requested in a timely manner, in which case it is saved.

No solution exists that allows source determination of all reflector attacks, nor one that allows forensic recovery and source location of packets, as would be needed for traceback, however the breadth of research is promising.

2.2 Network-Based Connection-Chain Traceback

Network-based connection-chain traceback uses network traffic analysis to match streams in a connection chain. The earliest work in stream matching proposed using network-monitoring devices to compute *thumbprints* based on the contents of the stream [Staniford-Chen and Heberlein 1995]. Other work also examined matching streams by altering their content [Wang et al. 2001]. Both schemes are easily defeated by hop-by-hop compression or encryption, and would likely be illegal to use without a warrant in the United States [Lee and Shields 2001], as it is against US law for ISPs to examine the contents of packets.

Another method proposed to compare the rate at which the TCP sequence numbers increase in a connection [Yoda and Etoh 2000]. The results indicate that streams in a small data set can be matched effectively, though false positives occur in a larger data set. What is unclear is if the connections that are being matched have experienced any significant congestion effects, as might be expected in an attack chain that traverses a large network like the Internet, or if the streams were subjected to compression at hosts which would alter the byte count significantly on different links, as can occur when using different versions of secure shell. Work has also appeared that uses timing information as an intrusion detection (ID) mechanism to find interactive streams arriving and departing a single domain over a single link [Zhang and Paxson 2000]. The use of timing information is an improvement because it is unaffected by content changes resulting from compression or encryption, as other work has shown that attackers are limited in their ability to effectively disguise such information [Donoho et al. 2002]. The authors also propose an architecture using their ID system as a means of forensic traceback.

While these results show that stream matching is possible in some circumstances, the reliability of this technique has not been demonstrated. The effects of network delay and packet loss have not been investigated, and there has been no work done on determining the error rates, in terms of false matches or missed matches, in a larger network like the Internet. Notice that an attack path may cross the network a number of times, so such effects can become significant.

2.3 Protocol-Based Connection-Chain Traceback

The only published protocol-based solution is the Caller Identification System (Caller ID) [Jung et al. 1993]. While its primary purpose is for authentication, the data it gathers could be used to trace an attacker. The system consists of two pieces of software that must run on each host, the Extended TCP Wrapper (ETCPW) and the Caller Identification Server (CIS).

The typical sequence of events is as follows:

(1) A user attempts to log in to H_i from H_{i-1}.
(2) The log in attempt is processed by a daemon on H_i, which passes the information to the local ETCPW application, ETCPW_i.
(3) ETCPW_i sends a request to the local CIS, CIS_i. The request includes the local port, remote port, and remote address.

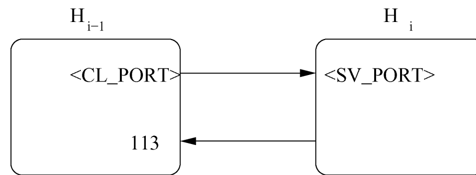

Fig. 2. IDENT request.

(4) CIS_i sends a request to the CIS on H_{i-1}, CIS_{i-1}, which is identified by the data from ETCPW_i. The request includes the remote and local ports numbers.
(5) CIS_{i-1} identifies the user session and returns a list of user id and IP address pairs of the previous hosts through which the user logged in.
(6) CIS_i verifies the list by sending a request to each IP address on it. The request includes only the user id. The remote host will return `yes` if the user has a process running and will return `no` if the user does not.
(7) If any CIS responds with `no` then the user is not allowed to log in to H_i. Otherwise, CIS_i saves the list, to return it when contacted by CIS_{i+1}, and notifies ETCPW_i that the user is authorized.

This system requires every host to run a CIS daemon and requires some overhead because it must save the previous host list for all active connections. This is most practical in a university or large corporate network but not across the entire Internet. This protocol may also add considerable delay to the login time, especially if a host is unreachable and the CIS must wait to timeout.

A problem that the Caller ID does not address is that of mapping a given process with the incoming network connection that was used by the process owner to gain access to the system. Instead, it verifies only that a process with that name is running. Because proper checking is not done, CIS_{i-1} can select random hosts and common users like `nobody` or `root`. Similarly, the CIS could `finger` random hosts for a list of active users and return them in the list. This scenario can be prevented by sending the port numbers of the network connection when verifying the list.

The Caller ID system also does not protect a user's privacy. The system tells every host in the connection chain which other hosts the user is logged in to. This allows hosts to create profiles of their users and maintain a list of other accounts their users hold.

2.4 The Identification Protocol

The IDENT [Johns 1993] is a simple two-way protocol that was designed to allow a server to identify the client-side user name of a network connection. The IDENT was previously called the Authentication Server Protocol [Johns 1985] and was later renamed because of its actual functionality. The protocol, as shown in Figure 2, works as follows:

(1) User U_{i-1} on host H_{i-1} establishes a TCP connection from port `<CL_PORT>` to port `<SV_PORT>` with host H_i.

```
<request> ::= <port-pair> <EOL>
<port-pair> ::= <integer> "," <integer>
<EOL> ::= "015 012" ; CR-LF End of Line Indicator
<reply> ::= <port-pair> ":" <reply-text> <EOL>
<reply-text> ::= <ident-reply> | <error-reply>
<ident-reply> ::= "USERID" ":" <os> ["," <charset>]
          ":" <user-id>
<error-reply> ::= "ERROR" ":" <error-type>
<error-type> ::= "INVALID-PORT" | "UNKNOWN-ERROR" |
          "NO-USER" | "HIDDEN-USER" | <error-token>
<os> ::= "OTHER" | "UNIX" | <token> | as defined in RFC 1340
<charset> ::= "US-ASCII" | as defined in RFC 1340
<user-id> ::= <octet-string>
<token> ::= 1*64<token-characters> ; 1-64 characters
<error-token> ::= "X"1*63<token-characters>
<integer> ::= 1*5<digit> ; 1-5 digits
<digit> ::= [0-9]
<token-characters> ::= All printable ASCII except ":"
<octet-string> ::= 1*512<octet-characters>
<octet-characters> ::= <any octet from 00 to 177 except
          NULL (000), CR (015) and LF (012)
```

Fig. 3. IDENT grammar.

(2) To determine the identity of U_{i-1}, H_i establishes a connection to TCP port 113 on H_{i-1} and sends the following message:

$$< \texttt{CL_PORT} >, < \texttt{SV_PORT} >$$

(3) H_{i-1} determines which, if any, process has a connection from port `<CL_PORT>` to port `<SV_PORT>` using the source IP address of the request.

(4) If the process is found, it returns a message such as:

$$< \texttt{CL_PORT} >, < \texttt{SV_PORT} >: \texttt{USERID} : \texttt{UNIX} :< \texttt{USER_ID} >$$

where `<USER_ID>` is U_{i-1}. In the case of error, the following is sent:

$$< \texttt{CL_PORT} >, < \texttt{SV_PORT} >: \texttt{ERROR} :< \texttt{ERROR_MSG} >$$

The grammar for the protocol is given in Figure 3.

An IDENT daemon comes with most UNIX systems and is used for several applications, including:

Internet Relay Chat (IRC). Resolves a handle to a user name. Many IRC servers require that a client be running an IDENT daemon. Some IRC clients contain IDENT daemons that return false user information.

Electronic Mail. *Sendmail* sends an IDENT request when it receives mail to trace forged mail. The response is placed in the email header.

Anonymous FTP. FTP servers can be configured to use the IDENT to determine the user name of those who use the *anonymous* login.

Port Filters. Applications such as TCP Wrappers [Venema 1992] can log and filter network requests based on IDENT replies.

As the IDENT returns user information to untrusted sources, it is not surprising that it can be used for other purposes besides security-based user identification. Dave Goldsmith showed that RFC 1413 did not specify that the daemon

should only return the identity of connections that originated on the local host [Goldsmith 1996]. By exploiting this, an attacker can learn as what user a service is running. The attacker establishes a connection to the service and sends an IDENT request for the connection. If the IDENT daemon does not distinguish between inbound and outbound connections, it will respond with the user name of the service.

Another undesirable consequence of running an IDENT daemon is that the email addresses can be gathered to create bulk email lists, or spam. This can occur when a user is using the World Wide Web and connects to a web server. Once the TCP connection is established between the HTML browser and the server, the server can query for the user name.

Several IDENT implementations take additional steps to protect user privacy. The daemon that ships with the OpenBSD operating system returns a string of 80 random bits in hexadecimal instead of the user name. The random string can be translated to a user name via log entries after proper identification and need have been presented to the system administrator. Similarly, the *pidentd* IDENT daemon [Eriksson 2000] can return the user name encrypted using DES. When an investigator needs to know the actual user, he or she can send the encrypted string to the system administrator and he or she can decrypt it.

In theory, the IDENT is useful, but in practice it has many shortcomings. These shortcomings are because of issues with trust. This protocol requires a host running it to give sensitive data to an untrusted host. Furthermore, the host receiving the data cannot trust it and therefore should not make any decisions based on it. For these reasons, this protocol provides little benefit and yet leaks private data.

The S/Ident Protocol [Morgan 1998] is an extension to the IDENT. It uses IDENT to provide authentication for application protocols that do not offer it. For example, this could be used by an HTTP server to authenticate a user before a sensitive HTML document is sent. This protocol relies on an authentication infrastructure, such as Kerberos, and therefore is not applicable to our needs.

3. THE SESSION TOKEN PROTOCOL

The protocol that is proposed in this paper, the STOP, provides additional functionality to what is offered by the IDENT [Johns 1993]. STOP operates using TCP to prevent a potential DoS attack. It can be run on any host with no modification of protocols, network topology, or kernel. It saves user-level and application-level data and can send recursive requests to trace connection chains. It can also be run in parallel with network analysis tools like those described in Section 2.2.

3.1 Protocol Design

3.1.1 *Design Goals.* The original protocol design goals were:

(1) Must be backward compatible with the IDENT as specified in RFC 1413 [Johns 1993] because of its widespread usage and implementation.

```
<request> ::= <port-pair> ":" <request-type> [":" <ip>]<EOL>
<port-pair> ::= <integer> "," <integer>
<request-type> ::= "ID" | "ID_REC" ":" <sid> | "SV" |
          "SV_REC" ":" <sid>
<ip> ::= <byte> "." <byte> "." <byte> "." <byte>
<sid> ::= <int>
<EOL> ::= "015 012" ; CR-LF End of Line Indicator
<reply> ::= <port-pair> ":" <reply-text> <EOL>
<reply-text> ::= <ok-reply> | <error-reply>
<ok-reply> ::= "USERID" ":" "OTHER" ["," <charset>]
          ":" <user-token>
<error-reply> ::= "ERROR" ":" <error-type>
<error-type> ::= "INVALID-PORT" | "UNKNOWN-ERROR" |
          "NO-USER" | <error-token>
<charset> ::= "US-ASCII" | as defined in RFC 1340
<user-token> ::= 1*512<token-characters>
<error-token> ::= "X"1*63<token-characters>
<byte> ::= integer values 0 to 2^8 in ASCII
<int> ::= integer values 0 to 2^32 in ASCII
<token-characters> ::= All printable ASCII except ":"
```

Fig. 4. STOP grammar.

(2) Must not release any user, application, or system data until proper credentials have been provided to an administrator.
(3) Must provide a mechanism to request that a daemon implementing this protocol save additional user-level and application-level data.
(4) Must provide a mechanism such that the protocol can trace a user's path through previous hosts.
(5) Must not release any data to eavesdroppers that they could not have determined from other traffic on the network segment.
(6) Must be configurable to comply with the system security and privacy policies.
(7) Should be efficient and not add considerable load to the daemon host or delay to the requester.
(8) Should allow a host that is not on the connection chain to make requests on behalf of a host.

The standard IDENT satisfies goals 1 and 7. Some implementations satisfy goals 2, 5, and 6 by returning random strings instead of user names and returning "OTHER" instead of the actual operating system. The IDENT offers nothing similar to goal 3, 4, or 8.

3.1.2 *Specification.* The IDENT satisfied many of the design goals and was used as a basis for the additional features. The new protocol modifies the request message to provide more options and modifies the response message to protect privacy. The new grammar can be found in Figure 4. The request message has the following format:

```
< CL_PORT >, < SV_PORT >:< REQ_TYPE > [:< SID >][:< CL_IP >]
```

Table I. STOP Request Types

Type	Description
ID	This request has the same behavior as the original IDENT. The daemon saves the user name in a log file and returns a random token.
ID_REC	This request will cause the daemon to log the user name and return a token. The daemon then sends ID_REC requests to the host from which the user logged in. This option requires a random session identifier, <SID>, to identify cycles in the traceback.
SV	This request will cause the daemon to not only log the user name , but also save data associated with the process that opened <CL_PORT>.
SV_REC	This request saves the same information as SV and also has the traceback property as described with ID_REC. This type also requires a session identifier, <SID>.

<CL_PORT> and <SV_PORT> are the TCP ports of the requested connection and the <REQ_TYPE> entry specifies the request type. Its values are given in Table I. <CL_IP> is an optional IP address in the standard X.X.X.X format that can be used as the remote address, instead of the address of the host that connected to the daemon. This is intended to be used by gateways, firewalls, or Intrusion Detection Systems (IDS). By using this, gateways can collect tokens on all outbound or inbound connections. To prevent information gathering by attackers, no error messages will be returned when <CL_IP> is specified in the request.

The protocol uses the same response messages as the IDENT, with three exceptions. "OTHER" is always returned as the operating system type to satisfy design goals 2 and 5 and because the operating system value is not required to identify a session. The second exception is that "HIDDEN-USER" is no longer required as an error message. The original intent of this message was to allow users to specify that their user name not be sent to other systems. This protocol only returns random tokens and therefore does not need this error type. The last change is that only printable ASCII is allowed in the user token. The original protocol allowed the return token to be any octet value except NULL, CR, and LF. This protocol returns random tokens that will be later redeemed for actual data, and it will be easier if tokens are generated using only printable ASCII.

This protocol returns a random token instead of a user name, because of the second design goal. In some implementations, the user may 'opt-in' to have his or her user name sent, to satisfy the requirements by some IRC networks.
A daemon that implements this protocol must have the following properties:

(1) Return a random token for all established outbound connections.
(2) Random tokens need not be cryptographically random, but must not contain any obvious values related to the request, such as UID, time, or IP address. The tokens must also be the same length for all request types and responses.
(3) Return an error for requests of TCP sessions that were not initiated by the local host (i.e, inbound connections).
(4) Return a random token to all requests that specify the remote IP address of the connection; this includes replacing error messages.
(5) Process requests in the original RFC 1413 format as ID type requests.

(6) Save additional user-level and application-level data when SV or SV_REC requests are received (see Section 3.3).
(7) Send requests with the same type and session identifier to the hosts that a user logged in from when ID_REC or SV_REC requests are received (see Section 3.1).
(8) Save tokens from recursive traceback requests with the returned random token. The recursive-based tokens must not be sent to the original requester.
(9) Do not process more than one request of type ID_REC or SV_REC from the same host with the same session identifier for a specified number of seconds, 120 for example. If a second request is received within the specified number of seconds of the first, a random token is returned and the event is logged.

A daemon that implements this protocol should have the following properties:

(1) Provide an option to return a random token instead of error messages.
(2) Provide an user-based option to return the actual user name instead of a token for an ID type request. All other request types must return a token.
(3) Provide options for what user, application, and host data to save on behalf of SV and SV_REC requests to satisfy policies or resources such as disk space.

3.1.3 *Traceback Requests.* The ID_REC and SV_REC request types allow tokens to be generated along an entire path of hosts. Unfortunately, the standard UNIX environment saves little about the previous host address. The only records of the previous host are typically entries in *wtmp*, or *utmp* files, which on many systems are host names truncated to 16 characters. Furthermore, there is not always a clear correlation between a process and a specific login. To solve the problem without modifying the kernel, the process and its parent processes are analyzed and requests are sent to any host connected to them via a TCP socket. The details of doing this are not outlined here, but are in Carrier [2001]. Buchholz and Shields [2002] have proposed a better solution to this problem by including the previous host IP address in every process structure.

The random token should be sent back to the requester before the recursive requests are sent. This is so the requester does not have to wait for all responses to be received. When the responses from the previous host are received, they should be saved with the original token. If any of the responses are sent to the requester, then the daemon would be violating design goal 2 because the requester would learn that the previous host is not the end of the chain.

3.1.4 *Loop Detection.* Recursive traceback requests must contain a random session identifier to prevent cycles and a DoS situation. The daemon must keep track of the ID_REC and SV_REC requests that it has seen within a specified number of seconds. The number of seconds should be chosen such that it is larger than the time required to trace a connection and smaller than the expected cycle time of the 32-bit random session identifier. 120 s was used in the prototype implementation. If the daemon receives a duplicate request for a TCP session with the same session identifier and from the same host within the specified time, it must not process the request and return a <user-token>

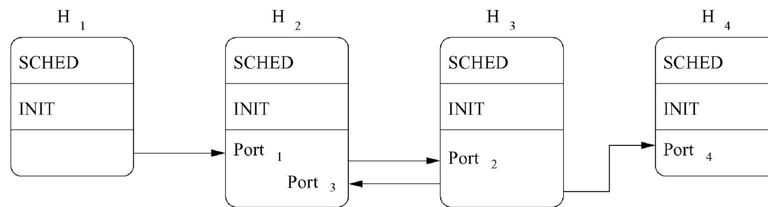

Fig. 5. Process trees of four hosts in a network loop.

type message. The daemon uses all bits of the source and destination addresses and ports identify duplicate requests.

If recursive requests are sent to determine previous hosts along the connection chain, the scenario shown in Figure 5 will cause a loop. H_2 has a process that runs the following pseudo code:

```
listen (Port1);
connect (H3, Port2);
listen (Port3);
```

H_3 has a process that runs the following pseudo code:

```
listen (Port2);
connect (H2, Port3);
connect (H4, Port4);
```

H_4 runs the following pseudo code:

```
listen (Port4);
```

The attacker connects to port Port_1 on H_2 from H_1. This will cause H_2 to connect to H_3, who will then connect back to H_2 and then connect to H_4. Now, let the following events occur:

(1) H_4 sends a SV_REC request to H_3 with random session identifier SID.
(2) H_3 sends a SV_REC request with identifier SID to H_2 because of the inbound connection to Port_2.
(3) H_2 sends SV_REC requests with identifier SID to H_1 for the connection to Port_1 and to H_3 for the connection to Port_3.
(4) H_3 sends a SV_REC request with identifier SID to H_2 for the connection to Port_2. The loop is not detected on H_3 because it has not previously seen a request from H_2.
(5) H_2 notices that it has already processed a request from H_3 with identifier SID and does not send any more requests.

3.2 Resolving Interprocess Communication

Performing a simple 'walk' up the process tree may not be adequate when tracing malicious users. As shown in Figure 6, if an attacker ran the following simple command to 'pass through' host H_i, the daemon could not determine

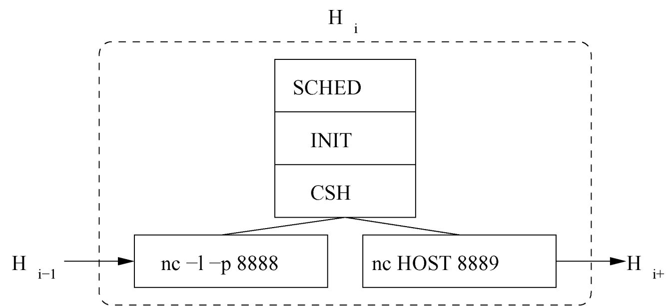

Fig. 6. Process tree of command: `# nc -l -p 8888 | nc HOST 8889`.

host H_{i-1}:

$$\# \texttt{nc} \; -\texttt{l} \; -\texttt{p} \; 8888 \,|\, \texttt{nc} \; < \texttt{H}_{i+1} > \; \texttt{8889}$$

This command uses *netcat* [Hobbit 1996] to listen on port 8888 of host H_i and pipes data received on that port to another *netcat* process that sends the data to port 8889 on host H_{i+1}. When the daemon 'walks' up the process that connects to H_{i+1} it does not encounter any other sockets. Therefore, if H_{i+1} sent a request of type `SV_REC` the daemon would not be able to send a recursive request. By resolving the pipe and determining which process was at the other end of the pipe, it is able to determine the identity of H_{i-1}.

It is therefore important that the daemon resolve as many types of Interprocess Communication (IPC) as possible. This includes pipes, local domain sockets (also called UNIX domain sockets), and Internet domain sockets connected to `localhost` or a local interface. IPC techniques such as shared memory are not addressed in this paper.

The processes that are identified from resolving IPC must have their process tree expanded and their sockets and pipes resolved. This continues until all sockets and pipes have been resolved.

3.3 Saving User and Application Data

A distinct feature of this protocol is the ability to save user-level and application-level state data. This functionality is achieved by sending an `SV` or `SV_REC` request to the daemon. Upon receiving this request, the daemon will save additional data to a file in a directory such as */var/stop*.

Table II lists data that are important to save. The first column lists those variables that should be saved for every process that is analyzed. This includes the process with the socket open and the parents of that process as the tree is 'walked'. These values could give investigators information regarding the type of software that was being used in the attack. Furthermore, the values listed are easy to determine. Some data, such as open files, could be useful to an investigator but is expensive to save because the daemon would have to translate an inode number to an actual file name. The second column lists variables that should be saved with every request. It includes data that can help an investigator verify what operating system was used and who made the

Table II. User, Application, and System State Variables

Per Process	Per Request
Process name	Host name
Process identifier (PID)	Boot time
Parent PID	OS/version/kernel
Real and effective UID	Address of requesting host
Start time	Address and port of remote end of socket
Terminal device	Address and port of local end of socket
Priority	Type of request
Open sockets and pipes	Entries in *utmp* for users in report

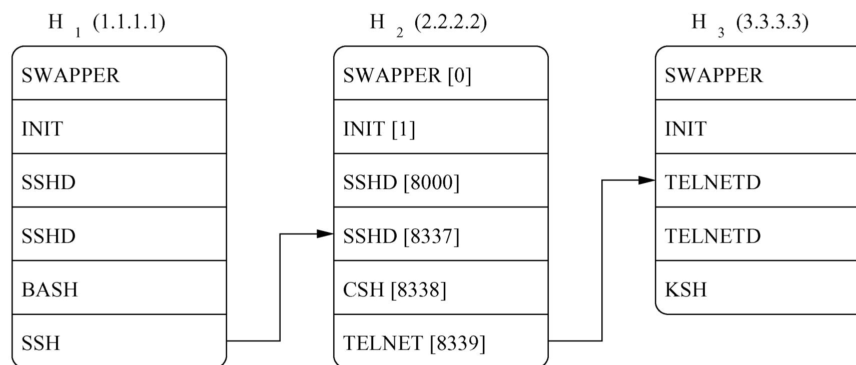

Fig. 7. Simple process structure process tree.

request. This data is recommended for completeness, but if storage space is an issue then this data may not be saved.

3.4 Case Studies

This section provides three examples of process trees that could exist and be analyzed by a daemon that implements this protocol. Each example provides an explanation and a possible procedure that the daemon could use to resolve the process structure. The first example is a simple and common scenario, the second is much more complex and unlikely to typically occur, and the third is a method used by attackers to control a compromised system.

3.5 Simple Process Structure

The most basic traceback scenario is if a user logs in to a system, gets a shell, and then logs in to another host. An example of the process trees in this scenario is shown in Figure 7.

In this example, Alice is logged in to H_1. She then uses the *ssh* protocol to log in to H_2 and from there uses the telnet protocol to log in to H_3. Using STOP, H_3 sends a `SV_REC` request to H_2. A summary of the data saved is shown in Figure 8, where H_{i-1} has IP address `1.1.1.1`, H_i has IP address `2.2.2.2`, and H_{i+1} has IP address `3.3.3.3`.

```
Primary Processes
1:  telnet [8339] parent:  8338
   Sockets:
      INET_TCP: 2.2.2.2:968 -> 3.3.3.3:23
2:  csh [8338] parent:  8337
3:  sshd [8337] parent:  8000
   Sockets:
      INET_TCP: 2.2.2.2:22 <- 1.1.1.1:616
4:  sshd [8000] parent:  1
   Sockets:
      INET_TCP: localhost:22 <- any:0
5:  init [1] parent:  0
6:  swapper [0] parent:  N/A
```

Fig. 8. Simple process structure process data.

The request for the socket between `2.2.2.2` port 968 and `3.3.3.3` port 23 is sent from `3.3.3.3` to `2.2.2.2` with session id 92847523456:

$$968, 23 : \text{SV_REC} : 92847523456$$

The daemon on `2.2.2.2` identifies that process 8339 has that Internet socket open. The other file descriptors are analyzed, but there are no other open sockets or pipes. The parent of 8339 is identified as 8338, *csh*, and its file descriptors are also analyzed. It is found to have no open sockets or pipes. The parent of *csh* is the SSH daemon child process, 8337. It is found to have an Internet domain socket from `1.1.1.1` using local port 22 and remote port 616. The SSH daemon parent process, 8000, is analyzed and found to have an Internet domain socket listening on port 22 with no connections. The parent of *sshd* is *init* and neither it nor its parent, *swapper*, have any open sockets or pipes.

The list of processes and file descriptors are analyzed for Internet sockets to `localhost`, local sockets, or pipes. None of these exist. The data are saved to a file, the SHA-1 hash of the file is calculated and returned to the requester, and the list is once again analyzed for Internet domain sockets. Process 8337 has an Internet domain socket with host `1.1.1.1`, so the following message is sent to `1.1.1.1`:

$$616, 22 : \text{SV_REC} : 92847523456$$

The daemon on `1.1.1.1` will process the request and identify the *ssh* process as having the socket open. The *ssh*, *bash*, *sshd*, *init*, and *swapper* processes will be analyzed, saved to a file, and a token will be returned to `2.2.2.2`. The host with the address `1.1.1.1` will send a request to the host that connected to it through the *sshd* process.

3.6 Complex Process Structure

A more complex process structure can be found in Figure 9. This process structure contains 14 unique processes, three process groups, and six forms of IPC that must be resolved. This structure starts as only process P_1 listening on an Internet domain socket. When it receives a connection, it spawns off process P_2 into its own process group and they connect with an Internet domain socket. Process P_3 is then spawned, an Internet domain connection is made, and P_3

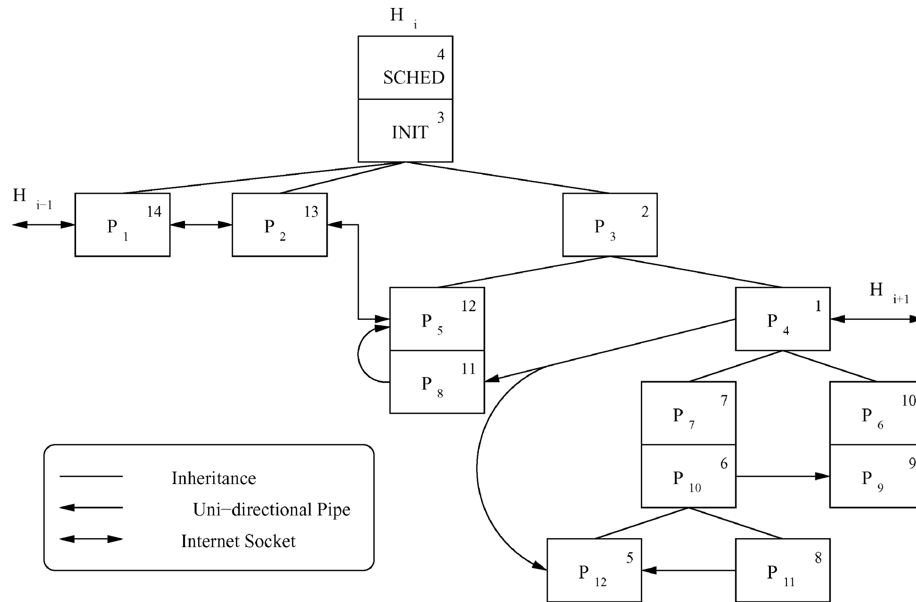

Fig. 9. Complex process structure process trees.

creates nine children that can communicate via pipes. Process P_4 creates an Internet domain socket and connects to H_{i+1} and a one-way communication path between P_4 and P_1 exists (through P_8, P_5, and P_2).

If this structure was running on a system, the STOP request would come from H_{i+1} for the connection to process P_4 and eventually resolve to process P_1. A summary of the state data on an OpenBSD system from this lookup is given in Figure 10. The actual numeric process identifiers have been replaced with the process labels shown in Figure 9. Pipes are saved and resolved in OpenBSD by using the kernel memory addresses of `pipe` data structures and Internet domain sockets are resolved by comparing the local and remote address and port tuples.

When a STOP request is sent from H_{i+1}, P_4 is identified as having the socket to H_{i+1} open. It is analyzed and found to have a pipe whose data structure at the local end is at kernel memory address `0xE07F2600` and the data structure at the remote end is at kernel memory address `0xE08CA780`. The parent of P_4, P_3, is analyzed but does not contain open sockets or pipes. The *init* process and *swapper* processes are analyzed next, but neither have open sockets or pipes.

The pipe on P_4 is resolved by looking for processes with a local pipe structure at `0xE08CA780`. P_{12} and P_8 are both found with a pipe structure at this kernel memory address.

Process P_{12} is analyzed and a new pipe with local address `0xE08CAE80` and remote address `0xE0801880` is identified. The parent of P_{12}, P_{10}, is analyzed and a pipe with local address `0xE0801C00` and remote address `0xE0801400` is found. The parent of P_{10}, P_7, is analyzed, but does not contain any open sockets or pipes. The parent of P_7, P_4, has already been analyzed.

```
Primary Processes
1:  resolve [P4] parent:  P3
   Sockets:
      INET_TCP: 2.2.2.2:8526 -> 3.3.3.3:9010
   Pipes:
      E07F2600 -> E08CA780
2:  resolve [P3] parent:  1
3:  init [1] parent:  0
4:  swapper [0] parent:  N/A
Resolved Processes
5:  resolve [P12] parent:  P10
   Pipes:
      E08CA780 -> E07F2600
      E08CAE80 -> E0801880
6:  resolve [P10] parent:  P7
   Pipes:
      E0801C00 -> E0801400
7:  resolve [P7] parent:  P4
8:  resolve [P11] parent:  P10
   Pipes:
      E0801880 -> E08CAE80
9:  resolve [P9] parent:  P6
   Pipes:
      E0801400 -> E0801C00
10:  resolve [P6] parent:  P4
11:  resolve [P8] parent:  P5
   Pipes:
      E08CA780 -> E07F2600
      E08CA380 -> E07E8080
12:  resolve [P5] parent:  P3
   Sockets:
   INET_TCP: localhost:8012 <- any
   INET_TCP: 127.0.0.1:8012 <- 127.0.0.1:32145
   Pipes:
      E07E8080 -> E08CA380
13:  resolve [P2] parent:  1
   Sockets:
      INET_TCP: localhost:8011 <- any
      INET_TCP: 127.0.0.1:8011 <- 127.0.0.1:39352
      INET_TCP: 127.0.0.1:32145 -> 127.0.0.1:8012
14:  resolve [P1] parent:  1
   Sockets:
      INET_TCP: localhost:8010 <- any
      INET_TCP: 2.2.2.2:8010 <- 1.1.1.1:1874
      INET_TCP: 127.0.0.1:39352 -> 127.0.0.1:8011
```

Fig. 10. Complex process structure process data.

The pipe that P_{12} has opened is resolved to P_{11}, which contains no additional file descriptors. The parent of P_{11}, P_{10}, has already been analyzed.

The pipe that P_{10} has opened is resolved to P_9. It is analyzed as is the parent process, P_6. Neither of them have additional open sockets or pipes and the parent of P_6, P_4, is the original process.

P_8 is analyzed next, because of the pipe with P_4, and a new pipe is found with local address 0xE08CA380 and remote address 0xE07E8080. The parent of

P_8, P_5, is analyzed and found to have the same pipe open. It also has an Internet domain socket on port 8012, which is connected to localhost. The parent of P_5, P_3, has already been analyzed. The pipe that P_8 and P_5 have opened is searched for, but no other processes are identified.

The Internet domain socket connection on P_5 to localhost was resolved to process P_2, which also had an Internet domain socket on port 8011 to localhost. The parent of P_2 is *init*. The TCP connection on P_2 is resolved to P_1, which is found to have an Internet domain socket on port 8010 to host 1.1.1.1. The parent of P_1 is also *init*. At this point, all IPC methods have been resolved.

If the original request had type ID_REC or SV_REC, then a traceback request would have been sent to 1.1.1.1 because of the connection with P_1.

3.7 Reverse Telnet

Reverse telnet [Scambray et al. 2001] is a technique that an attacker can use to execute commands on a compromised system behind a restrictive firewall. For example, a firewall may allow only port 80 and STOP traffic to the HTTP server. The server has a Common Gateway Interface script with a vulnerability such that attackers can execute an arbitrary command. To gain control of the host, the attacker must either kill the HTTP server and replace it with a shell listening on port 80, or get the host to make an outbound connection to his or her machine.

The reverse telnet technique creates two one-way communication channels, both of which start on the compromised host and connect to the attacker's host. The attacker first executes the following *netcat* command on his or her machine, H_{i-1} :

```
# nc -l -p 8000
```

and in a different terminal:

```
# nc -l -p 8001
```

The attacker exploits the server vulnerability such that the server executes the following command:

/bin/telnet H_{i-1} 8000 | /bin/sh |/bin/telnet H_{i-1} 8001

A figure of this can be seen in Figure 11. The attacker is running two *netcat* servers that are listening on ports 8000 and 8001 for connections. The command that is run on the compromised host uses two telnet sessions to connect to the two netcat servers. The firewall will not block the connections because they are outbound. The data received on the compromised server from the telnet connection to port 8000 is passed to /bin/sh through a pipe. The output from /bin/sh is then passed via pipe to the second telnet session, which sends the data to port 8001 on the attacker's system. The result of this is that the attacker can type commands in the window with the server listening on port 8000, they will be executed by the /bin/sh process, and the output will be sent to the other window with the second netcat server.

The attacker, H_{i-1}, has IP address 1.1.1.1, the compromised server, H_i, has IP address 2.2.2.2, and there is an IDS system on the victim's network that

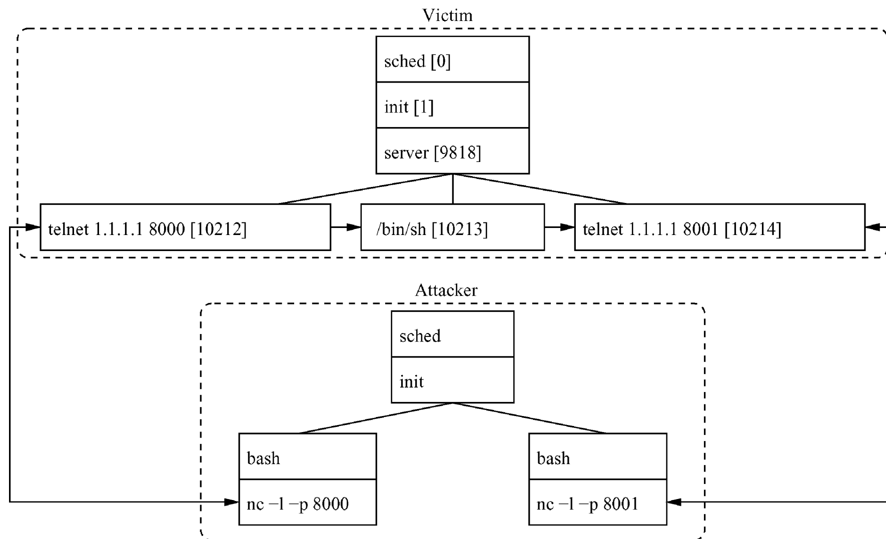

Fig. 11. Reverse telnet process trees.

```
Primary Processes
1:  telnet [10212] parent:  9818
   Sockets:
      INET_TCP: 2.2.2.2:1885 <> 1.1.1.1:8000
   Pipes:
      0 -> 38AB64
2:  server [9818] parent:  1
   Sockets:
      INET_TCP: 2.2.2.2:80 <> any
3:  init [1] parent:  0
4:  sched [0] parent:  N/A
Resolved Processes
5:  sh [10213] parent:  9818
   Pipes:
      0 -> 38AB64
      0 -> 38AB65
6:  telnet [10214] parent:  9818
   Sockets:
      INET_TCP: 2.2.2.2:1886 <> 1.1.1.1:8001
   Pipes:
      0 -> 38AB65
```

Fig. 12. Reverse telnet process data.

sends a request to the victim. There are two possible requests, the port 8000 connection will be examined first. The IDS system sends the following request to the daemon on 2.2.2.2:

1885, 8000 : SV_REC : 1485730682 : 1.1.1.1

A summary of the daemon output on a Linux box is shown in Figure 12. The daemon determines that process 10212 has the socket open and also identifies a pipe with inode 0x38AB64. The parent process is the vulnerable server,

which has an Internet socket open on port 80. The parent of the server is *init* and *sched*, which do not have any sockets or pipes. When the pipe with inode 0x38AB64 is resolved, the */bin/sh* process is identified, which also has a pipe with inode 0x38AB65. The parent of the */bin/sh* process is the server program, which has already been seen. The 0x38AB65 pipe is resolved to process 10214. Process 10214 is analyzed and found to have an Internet socket to host 1.1.1.1. The state data are saved and a token are returned to the IDS.

The daemon will analyze the process data for Internet domain sockets to which to send requests. The daemon cannot determine socket direction, because it is running on a Linux system, and will send a request for the connection in process 10214 to 1.1.1.1.

If the machine that the attacker is using is also running a STOP daemon, it may not process the request because it is an inbound connection. If the daemon cannot determine direction, then the *netcat* session will be saved and a request will be sent to the previous host.

If a request was sent for the connection to port 8001, then the same process data would have been saved, but in the opposite order.

4. IMPLEMENTATION

The protocol as described in Section 3 was implemented on three operating systems: Linux, OpenBSD, and Solaris. Each platform had a trait that made the implementation distinct. The Linux implementation used the process pseudofile system to extract detailed information about processes. The OpenBSD implementation used the KVM library and read the process table and file descriptor tables directly from kernel memory. The Solaris implementation also used the KVM library to read kernel memory, but Solaris uses a stream design for its in-memory network structures. This design makes it more complex to gather information than the OpenBSD design. Space limitations preclude an in-depth discussion of these differences, but they are available elsewhere [Carrier 2001].

The STOP daemon was based on the *oidentd* IDENT daemon [McCabe 2000] and had several run-time options including:

—Always return random tokens instead of errors.
—Always return "UNKNOWN-ERROR" for all error types.
—Select what state data to save for SV and SV_REC requests.
—Allow users to 'opt-in' to releasing their user name.
—Restrict the number of active lookups to limit the amount of resources the daemon takes.

If users are allowed to 'opt-in' to their user name being released, then they can create a file called ~/.ident that contains a list of hosts to which their user name can be sent. All other hosts are sent a random token.

The implementation resolves all pipes, local domain sockets, and Internet domain sockets to localhost or ones that have the same local and remote IP addresses. The implementation also assumes that only one process has a socket open, but that many processes would have a pipe open. The reason for this is

that Internet sockets are traditionally used for communication between hosts, but pipes are always used for IPC and are more likely to be used by more than one process. IPC methods such as shared memory were not resolved.

For request types SV and SV_REC, the state data was stored in a file. The SHA-1 hash of the data was computed and sent to the requester as the random token. The SHA-1 hash is sent as the token to detect any tampering the attacker may do to the data file. Our implementation saved all variables mentioned in Section 3.3. For a typical process tree with six processes, the output file was roughly 1600 bytes. If the tokens are saved to a small disk, an attacker could cause the drive to fill with token files before the actual attack. The data files could also be compressed to roughly 700 bytes.

For request types ID_REC and SV_REC, the process data are analyzed for open Internet domain stream sockets. We tried to limit ourselves to sending requests for inbound sockets only, but this was unsuccessful. One reason it was unsuccessful is that only OpenBSD socket structures save data about direction. When the direction is known, then requests are only sent to inbound sockets, but when direction is not known requests are sent to all sockets.

Cycles among recursive requests are detected by keeping a hash table of ID_REC and SV_REC requests. The hash function uses bits from the random session id, remote address, remote port, and local port.

4.1 Assumptions

Several assumptions were made while implementing the STOP protocol. It is assumed that only one process has a socket open, but that many processes may have a pipe open. This makes the typical scenario faster, because the program will stop searching after identifying a process that has the socket in the initial request and when resolving connections to the local host. Pipes are used only for IPC, while Internet domain sockets are primarily used for communication between hosts. A child process may have the same socket as its parent, but usually one of them closes it after the child is created. The implementation can be easily modified if this assumption is found to be invalid. This implementation did not take advantage of the reference count value of a socket or pipe, which could be used to identify the number of processes to search for.

An attacker could exploit this assumption by creating two processes with the same socket and letting the child process create its own process group. If the daemon resolves the socket to the child process, its parent is *init* and the previous host will not be determined. This will only work if the daemon analyzes the child process before the parent, which could be difficult for the attacker to ensure.

This implementation also assumes that files will not be used for IPC. It is possible for communication to be performed using this method, but files are typically used for storage and not as a communication channel. This assumption was made to make the typical scenario more efficient. If this assumption is shown to be invalid, the reference count could be used to identify files that are open by more than one process.

When sending recursive traceback requests, it is assumed that the services that accept incoming network connections and provide a method to make outbound network connections are creating child processes for each inbound connection. It is further assumed that the parent is closing its copy of the socket. When these assumptions are not true, the STOP daemon may send recursive traceback requests to every host that is connected to the service. This could generate an avalanche effect of traceback requests. These assumptions are made to simplify the traceback process. Otherwise, a file descriptor flow analysis must be performed to identify an inbound socket that can communicate with the requested outbound socket.

Finally, this implementation does not include a daemon for devices that do Network Address Translation (NAT). A NAT device that runs this protocol would return a token for a STOP request, identify the internal host that has the requested connection, and send a request to it. The response from the internal host would be logged with the token that was sent to the original request. This implementation was out of the scope of this project.

5. PERFORMANCE

The Linux and OpenBSD systems that were used to implement this protocol have identical hardware and were tested for performance results. The systems had 600 MHz Intel Pentium III processors and 128 MB of RAM. Though our Solaris implementation was successful, we did not take performance measurements because we lacked a modern Solaris system in our testing facility, and felt that the results on the old systems available would not provide an accurate representation of the implementation. We would expect the Solaris performance to be similar to that of the BSD system given the same hardware.

The daemon was first tested to determine how long requests would take to complete. This was performed in several environments and the results are given in Section 5.1. The system impact was also tested to determine how much a system's performance would be impacted by running this daemon. These results are found in Section 5.2.

5.1 Request Processing Times

The implementation code was tested to determine how long a request would take to complete. To simulate an actual daemon, the public interface was used. The test program created a child process, waited for it to finish, and repeated for a specified number of times. Each child process parsed a request string and processed it. The time it took to process the specified number of requests was recorded and divided by the number of requests to determine the average lookup time. The number of lookups was varied depending on the environment so that each test took around 90 min to complete.

This procedure was performed on two process structures, one simple and one complex. The results are given in the following two sections.

5.1.1 *Simple Process Structure.* The test program was first run on a simple process tree that contained six unique processes and no forms of IPC. It is the

Table III. Average Lookup Time (ms) for Six Unique Processes

Platform	ID	SV	SV with file
Linux	0.533	5.718	8.243
OpenBSD	0.803	2.421	7.871

Table IV. Average Lookup Time (ms) for 14 Unique Processes

Platform	SV	SV with 100 Processes
Linux	63.354	224.589
OpenBSD	10.256	32.059

process tree shown in Figure 7. It is assumed that a STOP daemon would process this type of structure the most frequently.

Table III shows the average number of milliseconds per lookup from the tests. The first data column shows the lookup time for an `ID` type request. As described in Section 3.1, an `ID` type request is equivalent to the traditional IDENT request. This was run to compare how much longer a new `SV` type request takes over the original IDENT lookup. The results show that Linux is the most efficient at determining the UID of a socket. This operation was performed in Linux by parsing the `/proc/net/tcp` file and in OpenBSD by using the `sysctl()` system call. Linux adds entries to the `/proc/net/tcp` file as a stack and the most recent socket is on top. Typically, a request will be made for the socket shortly after it is opened and it will therefore be one of the first entries in the file. The OpenBSD `sysctl()` function uses a hash table to find the protocol control block entry for the socket.

The second and third data columns contain the times for performing a `SV` type request. The third column includes the time to save the data to a file, while the second does not. As described in Section 3.1, a `SV` type request saves state data for the process tree that has the requested socket open. From the second data column, it is clear that it is faster to directly access kernel memory in OpenBSD than by searching and parsing the `/proc/` files in Linux. OpenBSD has a 201% increase in lookup time between a traditional `ID` request and the new `SV` request and Linux has nearly a 973% increase in lookup time. On average, Linux spends 136% more time performing an `SV` lookup than OpenBSD does. This is because OpenBSD can do more in kernel space and Linux must do file IO and use `scanf()` to determine process data. When both platforms write the process data to file, Linux takes only slightly longer.

5.1.2 *Complex Process Structure.* The test program was then run on the 14-process structure described in Section 3.6 and shown in Figure 9. This structure resolves to 14 unique processes, three process groups, and contains six instances of IPC to resolve using pipes and Internet domain sockets. This structure is not typical and is used as an extreme example.

The testing program performed lookups on the socket from process P_4 on a system with no other users and the results can be found in data column one of Table IV. These results show that the OpenBSD lookup time for the

Table V. System Performance Data

Platform	Requests Per Minute						
	6	20	60	120	600	3000	6000
Linux	99.88%	99.74%	99.21%	98.45%	92.99%	75.85%	64.08%
OpenBSD	99.99%	99.90%	99.61%	99.17%	96.11%	86.96%	80.80%

14-processes structure is 324% longer than for the six process structure. Linux had a 1008% increase over the six process structure and was 518% longer than OpenBSD.

To simulate a loaded system, the tests were repeated with the addition of 100 processes that had two open pipe descriptors, one open file descriptor, standard input, standard output, and standard error open. Therefore, each lookup had to examine 600 additional file descriptors when resolving pipes. The testing program was run again and the results can be found in the second data column. This shows that the average OpenBSD lookup had a 213% increase with the 100 additional processes, Linux had a 254% increase, and Linux took 600% longer than OpenBSD.

While these increases sound substantial, they are for an nontypical example. As will be shown next, the system impact of processing requests is minimal.

5.2 System Performance

The system impact was measured to identify how much system performance would be impacted by running this daemon versus not running the daemon. To perform this test, a memory intensive program was written where each round took roughly 10 min to run with no load. The program creates an array of 1,000,000 floating point entries. It then performs a series of floating point calculations on elements within the array. To cause nonsequential memory accesses, operations are performed on random elements in the array.

The execution time of the benchmark program was measured on the OpenBSD and Linux systems with no other processes running to get a base time. The test program used in the previous lookup tests was modified such that it slept for a specified number of seconds between lookups. The test program and benchmark program were then run simultaneously and timed. The benchmark base time was divided by the execution time to calculate the performance impact. The impact percentage was compared with the number of lookups per minute being performed. This benchmark was designed to measure both the CPU and memory access load.

Table V shows the performance percentages that were found by running the daemon at intervals of 6, 20, 60, 120, 600, 3000, and 6000 lookups per minute. Each lookup was a `SV` type request on a six-process basic process tree with the output printed to a file. Figure 13 shows these values in a graph.

These data show that the daemon does not pose a significant threat to system performance under typical operation. For a reference value, the average number of logins per minute was calculated from the main student computer at Purdue University. The computer, `expert.cc.purdue.edu`, is run by the Purdue University Computing Center and all graduate and undergraduate students are

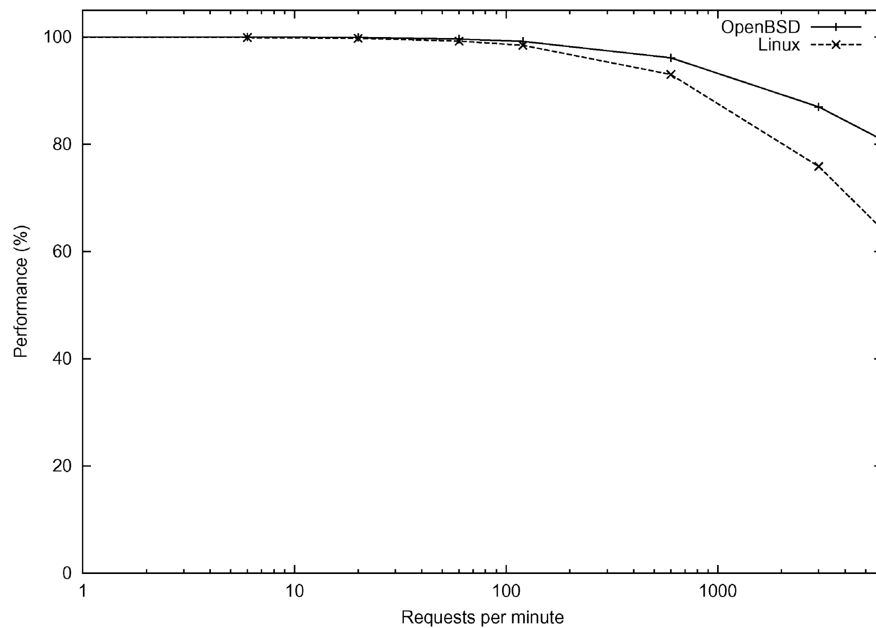

Fig. 13. Performance impact graph.

given an account on it. Over a seven-hour period, there were 2499 logins, or almost six per minute. If we use this value as an upper bound for the number of requests a host like `expert` would receive a minute, the daemon impact would be negligible. The upper bound is the extreme case that every user logged in to another system after logging in to `expert`. Few users do this on a regular basis.

Clearly, resolving processes is an expensive operation, but the complex structure as shown in Section 5.1 is not typical. The lookup times with the additional 100 processes shows that on a multiple user system, significant time could be spent on these lookups and possibly result in a DoS scenario if enough requests are being serviced. As shown in Section 5.2, a system can process 100 requests per second for basic process trees and only see an 20% or 36% performance decrease. This high number of requests will most likely only occur when the host is under attack. The daemon restricted the number of active lookups to prevent it from consuming all of the system resources. The data presented here also show that by only using a process pseudofile system, as Linux does, the daemon does not scale as well.

6. LIMITATIONS

6.1 Partial Deployment

Traceback will occur across the entire length of the connection chain only if every intermediate host is running a STOP daemon, otherwise the chain of hosts will be identified to the first host not running the protocol. If every intermediate

host in the chain is running the protocol, then the attacker's IP address can be identified even if the attacker is not running it.

When a connection chain is closed before the traceback is complete, the remaining hosts cannot be determined, and only a partial result will be available. Most operating systems do not save a socket's data after it has been closed, so the previous host cannot be identified. A feature to cache socket data would require a kernel modification, and has been proposed in other work [Buchholz and Shields 2002].

While a trace of the entire connection chain would be optimal, partial results are still potentially useful. STOP is not intended to be a complete traceback solution in and of itself. Instead, it is designed to make it easier for a forensic investigator to complete an investigation of the connection chain. At each host running STOP, the investigator may only need to verify the integrity of the STOP daemon and its records (as described below) instead of performing a complete analysis of the system. Because forensic investigation is an expensive process, this will reduce the cost and improve the efficiency of investigation.

STOP can be useful in making investigations of internal systems more efficient, even if it only achieves limited deployment. Organizations, like governmental agencies and large corporations, could have a policy that requires STOP to be run on internal networks, and perhaps could mandate that incoming connections to their domain honor STOP requests. These organizations, which currently have internal networks and intranets consisting of many thousands of systems, could improve their capability for response and decrease their investigative costs even if STOP was not adopted across the whole Internet.

6.2 Covert Channels

Covert channels have been a concern in operating systems research for many years [Gligor 1993], though the results of that research have yet to be applied to the general-purpose operating systems currently in use. STOP is designed to trace internal data flow across most of the standard IPC mechanisms, as described in Section 3.2. It is possible for an attacker to create an environment where processes communicate using other forms of IPC. For example, one process may open a file, write data to it, and close it. Another process may open the same file, read the data from it, and close it. This is a simple form of a covert storage channel and one that will halt an automated traceback process using STOP. It may be impossible to prevent or detect all covert channels in a general-purpose operating system, though recent work has proposed adding mechanisms to the kernel to trace the flow of information through a system [Buchholz and Shields 2002].

If a covert channel exists that STOP does not detect, then a partial traceback will exist to the host. In addition, STOP will have identified at least one process that was involved in the covert channel and may provide clues to help the investigation locate the channel. In the previous example, the STOP daemon would identify one of the applications that was using the file for communications. Looking for similar applications may identify the other process, logs, and

STOP information relating to where the user connected from. More complicated covert channels might leave more evidence for the investigator.

6.3 Effects on Network Devices

While STOP is an end-to-end protocol, it can have an impact on a variety of network devices that forward traffic within the network.

A device that might be impacted by STOP is a proxy. If the proxy only forwards connections and does not allow the user to run processes, it is very easy to design a STOP proxy that stores all user information and tokens itself and which does not need to parse any operating system data structures to match incoming and outgoing connections. Such a STOP proxy could be made highly efficient.

If the proxy is a firewall of the bastion host type, all connections that cross it will require a login, and a STOP daemon will need to run on that firewall. If users cannot run arbitrary processes on the host but only connect out, the more efficient proxy described above can be used. Otherwise, the host will need to process the STOP requests like other systems do. Servicing a large number of STOP requests can cause a load on the system and that excessive load might cause a DoS situation. However, as we show through measurement of our implementation, the overhead does not seem excessive enough that this will be a more effective attack than attacking many other services that will likely be available on the host. This is particularly true because the lookup process is brief if the connection being requested does not exist, and replies are rate limited for existing connections.

If a packet-filtering firewall is being used, it will have to be configured to allow STOP requests through transparently. Therefore, the system may have some increased load from forwarding the STOP traffic, though the extra load from forwarding STOP traffic would likely be negligible given the relative proportion of interactive connections to other types of traffic in the network. The possibility would exist for an attacker to map active IP addresses behind the firewall by attempting to connect to the STOP TCP port, 113. This is no different than the many other services that network mapping can target, and the same defenses can be used. The simplest is to configure the firewall as a NAT device, which would require a daemon on the firewall that can translate STOP requests as described in Section 4.1. Like the proxy daemon, this daemon would not need to analyze the local processes and could be very efficient. Some packet-filtering firewalls examine traffic and change their rules in response to what they observe on the network. Such a firewall could be programmed to only pass STOP requests associated with an active connection, further reducing the overhead and the ability of attackers to use STOP for network mapping.

Another problem exists in that some network devices, such as routers and switches, allow remote logins and can be used as stepping stones themselves. To allow STOP to conduct traceback across these devices, the daemon would need to be ported to them. Unfortunately, these devices have limited processing and storage resources, but there are several possibilities for addressing these limitations. To reduce CPU load, the lookup mechanism could be simplified to

reflect that there are fewer mechanisms for IPC and fewer user processes. To reduce the storage requirements, the data could be sent to a remote log server. In general, these devices present a significant impediment to any forensic investigation because of their heavy load and natural limitations.

6.4 Compromised STOP Daemons

This protocol may not trace every connection chain, because the daemon can be killed on any system for which the attacker has gained *root* privileges. It is important to remember that the logs of any system that has had *root* access compromised cannot be fully trusted.

If the attacker kills the daemon, the situation is the same as though the host was never running it. Therefore, H_{i+1} will have a log message indicating that H_i has rejected the network connection and the attacker's path can be traced back to only H_i.

If the attacker replaces the daemon with a rogue version, several situations can occur:

—The daemon could not save any data. This is the same as if it were not running and the path would be known to H_i, though a forensic examination might reveal H_{i-1}, and the STOP trace could continue from there.

—The daemon could not send recursive requests, which would cause the path to also end at H_i if it does not save the previous host data or at H_{i-1} if it does save the previous host data.

—The daemon could save false application and traceback data. For example, the daemon could pick another user session at random, and claim that it was the attacker's session. This scenario could lead an investigator away from the true path, but the compromised host would be investigated for malicious activity.

Because the data from a STOP daemon cannot always be trusted, the validity of the daemon and saved data must be determined during an investigation.

6.5 Integrity of Saved Data

If the attacker has gained *root* access to the system, he or she can easily modify or delete the process state files and log file entries. It is far outside the scope of this work to prevent this, but measures can be taken to detect it.

The log entries can be protected by sending them to a log server. An attacker must gain *root* access to the log server to modify the logs. An alternative is to use cryptography to detect when a log entry has been modified [Schneier and Kelsey 1999; Bellare and Yee 1997]. These methods will not prevent the log from being modified, but will identify when an entry has been changed or deleted. Using write-once media will protect the log entries from remote intrusion and require an attacker to be present to remove the information.

The easiest way to protect the process data files is to generate a one-way hash of them using SHA-1 [SRI International 1995]. The hash value is returned to the requester as the token. When the requester redeems his or her token for the data file, the hash can again be calculated for the file to verify it is the same

value as when it was originally created. This will show that a modification has occurred, but not what was modified.

7. CONCLUSION

The STOP provides data that are commonly missing during forensic investigations. It provides a record of socket activity and allows an attacker who is using a series of hosts to be traced. By returning only random tokens, a user's privacy is protected, though other systems cannot rely on replies for authentication.

We have implemented the protocol for several different operating systems and shown that it is effective in saving data about a network session and tracing connection chains. Performance analysis shows that overhead is very low in the most common case, and that it is not prohibitive in other cases. STOP can be used in parallel with other traceback techniques such as network traffic analysis to provide application-level data to investigators.

This protocol is most effective when many hosts are running it. While it could be used for tracing TCP chains across the Internet it is more likely to be useful in more constrained environments in which there are enforceable policies that require the STOP daemon to be run. This could be a single network or an intranet, as the ability to make requests on behalf of other machines provides border gateways and ID systems with a method to request data on suspicious inbound and outbound traffic.

STOP is the first protocol that addresses in implementation the problem of correlating incoming network connections with outgoing ones in existing operating systems, and allows it to be saved in a privacy-preserving manner. While it is clear that STOP will not solve the problem of TCP connection-chain traceback in all situations, it is a further step towards a solution.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thorough reviews and constructive criticism.

REFERENCES

ADLER, M. 2002. Tradeoffs in probabilistic packet marking for IP traceback. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*.

BARROS, C. 2000. [LONG] A Proposal for ICMP Traceback Messages. Available at `http://www.research.att.com/lists/ietf-itrace/2000/09/msg00044.html`.

BELLARE, M. AND YEE, B. 1997. Forward integrity for secure audit logs. Tech. rep. (Nov.), Computer Science and Engineering Department, University of California at San Diego.

BELLOVIN, S. 2000. ICMP Traceback Messages. Tech. rep., draft-bellovin-itrace-00.txt (Mar.), IETF Internet draft.

BUCHHOLZ, F. AND SHIELDS, C. 2002. Providing process origin information to aid in network traceback. In *Proceedings of the 2002 USENIX Annual Technical Conference*.

BURCH, H. AND CHESWICK, B. 2000. Tracing anonymous packets to their approximate source. In *Proceedings of the 14th Conference on Systems Administration (LISA-2000)*, New Orleans, LA.

CARRIER, B. 2001. A Recursive Session Token Protocol For Use in Forensics and TCP Traceback. Master's thesis, CERIAS, Purdue University.

CARRIER, B. AND SHIELDS, C. 2002. A recursive session token protocol for use in computer forensics and TCP traceback. In *Proceedings of the IEEE Infocomm 2002*.

Chang, H. Y., Narayan, R., Wu, S. F., Vetter, B. M., Wang, X., Brown, M., Yuill, J. J., Sargor, C., Jou, F., and Gong., F. 1999. Deciduous: Decentralized source identification for network-based intrusions. In *Proceedings of International Symposium on Integrated Network Management*. 701–714.

Cisco Systems Inc. 2003. Characterizing and Tracing Packet Floods Using Cisco Routers. Available at `http://www.cisco.com/warp/public/707/22.html`.

Computer Science Laboratory, SRI International, Menlo 1995. Secure hash standard. Federal Information Processing Standards Publication (FIPS PUB) 180-1.

Dean, D., Franklin, M., and Stubblefield, A. 2001. An algebraic approach to IP traceback. In *Proceedings of the 2001 Network and Distributed System Security Symposium*, San Diego, CA.

Doeppner, T. W., Klein, P. N., and Koyfman, A. 2000. Using router stamping to identify the source of IP packets. In *7th ACM Conference on Computer and Communications Security*, Athens, Greece. 184–189.

Donoho, D., Flesia, A. G., Shankar, U., Paxson, V., Coit, J., and Staniford, S. 2002. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *Recent Advances in Intrusion Detection—Proceedings of the 5th International Symposium (RAID 2002)* Zurich, Switzerland, A. Wespi, G. Vigna, and L. Deri, Eds., *Lecture Notes in Computer Science*, vol. 2516. Springer-Verlag, Berlin, Germany.

Eriksson, P. 2000. pidentd ident daemon v3.0.12. Available at `http://www2.lysator.liu.se/~pen/pidentd/`.

Ferguson, P. and Senie, D. 1998. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical Report RFC 2267 (Jan.), Internet Society. Available at `ftp://ftp.isi.edu/in-notes/rfc2267.txt`.

Ferguson, P. and Senie, D. 2000. Network Ingress Filtering: Defeating Denial of Service Attack s which employ IP Source Address Spoofing. Technical Report RFC 2827 (May), Internet Society.

Gligor, V. 1993. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030 (Nov.), National Computer Security Center, Ft. George G. Meade, Maryland. Approved for public release: distribution unlimited.

Goldsmith, D. 1996. ident-scan. Email post to Bugtraq Mailing List. Available at `http://lists.insecure.org/bugtraq/1996/Feb/0024.html`.

Hobbit. 1996. netcat v1.10. Available at http://www.l0pht.com/weld/netcat/.

Ioannidis, J. and Bellovin, S. M. 2002. Implementing pushback: Router-based defense against DDoS attacks. In *Proceedings of Network and Distributed System Security Symposium*. The Internet Society.

Johns, M. S. 1985. Authentication server. RFC 931, TPSC.

Johns, M. S. 1993. Identification protocol. RFC 1413, US Department of Defense.

Jung, H. T., Kim, H. L., Seo, Y. M., Choe, G., Min, S. L., Kim, C. S., and Koh, K. 1993. Caller identification system in the Internet environment. In *UNIX Security Symposium IV Proceedings*. 69–78.

Lee, S. C. and Shields, C. 2001. Tracing the source of network attack: A technical, legal, and societal problem. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY.

Mankin, A., Massey, D., Wu, C. L., Wu, S. F., and Zhang, L. 2001. On design and evaluation of intention-driven ICMP traceback. In *Proceedings of the IEEE International Conference on Computer Communication and Networks*.

McCabe, R. 2000. oidentd ident daemon v1.7.1. Available at `http://ojnk.sourceforge.net/`.

Morgan, R. 1998. Sident: Security extensions for the ident protocol. Available at `http://globecom.net/ietf/draft/draft-morgan-ident-ext-04.html`.

Park, K. and Lee, H. 2001a. On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack. In *Proceedings IEEE INFOCOM 2001*. 338–347.

Park, K. and Lee, H. 2001b. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA.

Paxson, V. 2001. An analysis of using reflectors for distributed denial-of-service attacks. *ACM Comput. Commun. Rev. (CCR) 31*, 3 (July), 1–10.

SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. 2000. Practical network support for IP traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*.

SCAMBRAY, J., MCCLURE, S., AND KURTZ, G. 2001. *Hacking Exposed*, 2 ed. McGraw Hill, Osborne, 319–321.

SCHNEIER, B. AND KELSEY, J.. 1999. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur. 2*, 2, 159–176.

SNOEREN, A. C., PARTRIDGE, C., SANCHEZ, L. A., JONES, C. E., TCHAKOUNTIO, F., STRAYER, W. T., AND KENT, S. T. 2001. Hash-Based IP Traceback. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA.

SONG, D. X. AND PERRIG, A. 2001. Advanced and authenticated marking schemes for IP traceback. In *Proceedings of the IEEE Infocomm 2001*.

STANIFORD-CHEN, S. AND HEBERLEIN, L. T. 1995. Holding intruders accountable on the Internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy,* Oakland, CA. 39–49.

STONE, R. 2000. CenterTrack: An IP overlay network for tracking DoS floods. In *Proceedings of the 9th USENIX Security Symposium,* Denver, CO.

VENEMA, W. 1992. TCP wrapper: Network monitoring, access control, and booby traps. In *Proceedings USENIX UNIX Security Symposium III*.

WANG, X., REEVES, D., WU, S., AND YUILL, J. 2001. Sleepy watermark tracing: An active network-based intrusion response framework. In *Proceedings of the IFIP Conference on Security*.

YAAR, A., PERRIG, A., AND SONG, D. 2003. Pi: A path identification mechanism to defend against DDoS attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*.

YODA, K. AND ETOH, H. 2000. Finding a connection chain for tracing intruders. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*.

ZHANG, Y. AND PAXSON, V. 2000. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO.

Received July 2003; revised December 2003, March 2004, and May 2004; accepted May 2004