A RECURSIVE TCP SESSION TOKEN PROTOCOL FOR USE IN COMPUTER FORENSICS AND TRACEBACK

Brian Carrier

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

A RECURSIVE TCP SESSION TOKEN PROTOCOL FOR USE IN COMPUTER FORENSICS AND TRACEBACK

A Thesis

Submitted to the Faculty

of

Purdue University

by

Brian Carrier

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2001

CERIAS TR 2001-19

ACKNOWLEDGMENTS

I first extend thanks to my thesis committee: Dr. Clay Shields, Dr. Eugene Spafford, and Dr. Jens Palsberg. I would also like to thank CERIAS and all of its members for providing me with an amazing environment, which has allowed me to learn and grow. The technical suggestions and comments from Tom Daniels and Ben Kuperman were always helpful and greatly appreciated.

Thank you to my family for providing me with the opportunities and freedom over the years to follow my dreams and reach my goals.

Lastly, I would like to thank my Lego planes and spaceships, pokemon characters, and Mr. Krups for entertaining and supporting me during the late nights in the office. Thank you Jenny for trying to understand why I was spending so much time with the aforementioned objects and reminding me that there is sometimes a world beyond computer security.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Carrier, Brian D. M.S., Purdue University, May, 2001. A Recursive TCP Session Token Protocol for Use in Computer Forensics and Traceback. Major Professor: Clay Shields.

In this thesis, a new protocol is presented, the Session Token Protocol (STOP) that can assist in the forensic analysis of a computer involved in malicious network activity. It has been designed to trace attackers who log on to a series of hosts to hide their identity. The protocol utilizes the Identification Protocol (ident) infrastructure and improves its capabilities and user's privacy. The STOP protocol saves user- and application-level data associated with a requested TCP connection and returns a random token. The user- and application-level data are not revealed until the token is returned to the local administrator. A trail of tokens can be created by sending a traceback request to the previous host from which the user has connected. The previous host will save the appropriate data, return a token, and send a new traceback request. This allows an incidents investigator to trace attackers to their home systems, but does not violate the privacy of normal users. This thesis also describes how the new protocol was implemented on three platforms.

1. INTRODUCTION

Many times, attackers log on to a series of compromised hosts before they attack their target. This technique complicates the forthcoming investigation and is commonly called stone stepping [ZP00]. The first obvious step when investigating an attacker's path is to contact the previous host, if discernible, and ask the administrator to investigate his or her system. This administrator may lack the resources, knowledge, trust, or data to continue the investigation. In some instances, there is inadequate logging on a system to even identify the previous host.

Attempts have been made to correlate pairs of network connections to identify hosts that are being used in the same attack. These methods collect network-level data related to the attack, but give no insight regarding user- or application-level data. These methods identify a host, but not the user account or software involved. As many operating systems do not log outbound sockets, there could be little record of which user made a network connection to a remote host. Furthermore, because many operating systems do not correlate inbound data flow with outbound data flow, it maybe difficult to determine where the attacker logged on from, even when the user id is known.

In this thesis, a new protocol based on the Identification Protocol (`ident`) [Joh93] is presented, which helps forensics investigations, while protecting the privacy of users. It compensates for the lack of socket logging by giving border gateways the ability to request data about inbound and outbound TCP connections. A daemon that implements this protocol can save application-level data about the process and user that opened the TCP connection, and can send traceback requests to identify previous hosts. At each stage, a hashed token is returned and at no point in the protocol does the requester ever directly learn user or process data. Instead, he or she must redeem the token and prove his or her identity.

This paper will first describe related traceback work and the original `ident` protocol. The design goals and specification of the new protocol will then be given, along with case studies. The details of the Linux, OpenBSD, and Solaris implementations will be given with performance results. It will conclude with operating system design suggestions for making this process easier.

2. PREVIOUS AND RELATED WORK

We will begin with some basic definitions [SCH95]. As shown in Figure 2.1, let H_i, $0 \leq i \leq n$, be a set of hosts, and let there be a connection C_i between hosts H_i and H_{i+1} if there exists an active TCP session between them. A connection chain, $\mathcal{C}$, between hosts H_0 and H_n is the set of connections C_i, where $0 \leq i < n$.

There are currently two major categories of network traceback: IP traceback and connection chain traceback. This paper is concerned with connection chain traceback, but IP traceback will be briefly discussed in Section 2.1 for completeness. Connection chain traceback can be broken up into two categories: network traffic analysis and protocols with which stone-stepping hosts communicate. These two categories will be discussed in Section 2.2 and Section 2.3. The Identification Protocol will be discussed in Section 2.4.

2.1 IP Traceback

IP traceback is concerned with identifying hosts that are sending IP packets with forged source addresses. This scenario is common in denial of service attacks. The work by Savage et al. [SWKA00] and Song and Perrig [SP01] add routing information to the `ID` field in an IP packet [Pos81]. The extra information is added to packets with a certain probability and will help determine the actual path through which the packet traveled. The theory is that when a host is being flooded by forged IP packets,

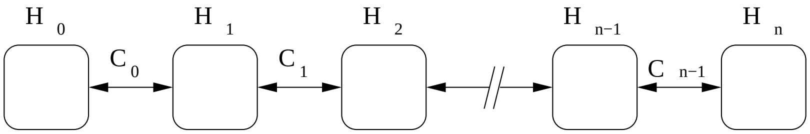

Figure 2.1. Connection chain example between H_0 and H_n

many packets will have routing data in the ID field and the path they took can be determined. The effectiveness of this technique was analyzed by Park and Lee [PL01].

Another method, proposed by Bellovin [Bel00], sends ICMP messages from routers that process an IP packet to the packet's destination on a probabilistic basis. The theory is that if the victim is being flooded with forged IP messages, they can identify the actual path by examining the source and content of the ICMP messages. This technique does not require any modification to the network because the messages can be sent from special network devices and gateways can filter the ICMP messages if they do not use them.

IP traceback identifies the host that is sending forged IP packets, but does not identify the attacker's location. It is likely that the attacker is not sending the forged packets from his or her personal machine and therefore an investigator must use connection chain traceback to identify the attacker's home base. IP traceback attempts to identify the host that is sending forged IP packets and connection chain traceback attempts to identify the hosts that an attacker is using for his or her attack.

2.2 Network-Based Connection Chain Traceback

2.2.1 Holding Intruders Accountable on the Internet

The Staniford-Chen and Heberlein paper [SCH95] was the first to present work in network connection correlation. Their goal was to provide an off-line procedure for identifying connection chains after an attack was detected, yet not modify any network protocols.

Their approach was to make a *thumbprint* for active connections based solely on packet content. Their design placed network sniffers at gateways and monitored all active connections. A thumbprint was calculated for each connection using the character frequency of the packet contents. The size of a thumbprint increased at a rate of 24-bytes per minute.

When an attack was detected, the data from network sniffers was gathered and processed. The processing included statistical analysis that would identify sets of thumbprints that could have been in the same connection chain.

While this method may produce good results for cleartext communication protocols such as telnet or rlogin, it does not work on channels that use encryption or compression. This method does not need accurate or synchronized clocks and therefore scales well because it can use thumbprints from all over the world without having to consider network delay and synchronization.

2.2.2 Detecting Stepping Stones

Yin Zhang and Vern Paxson published a paper on tracing connection chains by using timing analysis [ZP00]. Their design placed a network sniffer at network gateways and recorded the ON and OFF times of all network connections. The OFF time began after a specified idle time of no data flow and the ON time began when data flow occurred. The analysis machine would try to match inbound connections with outbound connections by analyzing the beginning of the ON times, or similarly the end of the OFF times. The beginning of the OFF time was not used because it was too dependent on the available bandwidth of each connection and was therefore not an accurate measurement.

Pairs of connections, C_i and C_j, were eliminated if their ON times were further apart than γ, or if C_i's ON time began before C_j in some instances and C_j's ON time began first in others.

As this approach uses only packet timing, it can be used for encrypted channels, including link level encryption when header data is encrypted. This algorithm can be fooled when an attacker adds more than γ delay between the inbound and outbound connections. This of course is inconvenient for the attacker because his or her data transfer will be slower.

2.2.3 Finding a Connection Chain for Tracing Intruders

Another method was presented by Kunikazu Yoda and Hiroaki Etoh from IBM [YE00]. They traced TCP packets using sequence number and timing analysis, again not relying on packet content in case encryption was used. Their scheme is similar to the one by Staniford-Chen and Heberlein [SCH95] because they record data about connections at a number of network gateways and then try to correlate the results to find pairs of connections.

They use TCP sequence numbers to correlate traffic. When a TCP connection is made, a random initial sequence number is chosen. A sequence number is sent with every TCP packet and is incremented by the amount of data sent in the previous packet. For example, if a packet was 128 bytes long and had a sequence number of 150, the next packet would have a sequence number of 278. Therefore, the sequence numbers of a connection increase at a rate proportional to the number of bytes transmitted and connections in the same connection chain will have sequence numbers that increase at roughly the same rate.

The Yoda and Etoh scheme records the sequence number of each packet and makes a graph with packet time on the X-axis and sequence number on the Y-axis. The graph will increase at a rate proportional to the amount of data transmitted and two connections are in the same chain if their graphs are similar. The X-axis and Y-axis values are shifted to account for network time delay and initial sequence number. The *deviation* is defined as the minimum average difference in the X-axis between two graphs. When this value is small, the two connections are more likely to be in the same chain.

This method does not work when link encryption is used because the TCP header is encrypted. It does works when application level encryption is used, such as with SSH [YKS+93]. This method also does not work when compression is used in some, but not all, connections. In this case, the connections that use compression will have a slower increase in sequence numbers.

2.3 Protocol-Based Connection Chain Traceback

2.3.1 Caller Identification System

The only published protocol-based solution to tracing multiple host connections is the Caller Identification System (Caller ID) [JKS+93]. The Caller ID system is a set of protocols designed to authenticate a user and identify the previous hosts that a user is logged into. Its primary purpose is for authentication, but the data it gathers can also 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 Extended TCP Wrapper 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.

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 extensive overhead because it must save the previous host list for all active connections. This is only practical in an university or large corporate network and not across the entire Internet.

The Caller ID work does not address the problem of mapping an outgoing TCP connection with an incoming connection. The CIS saves the list of previous hosts for a user, but when the user has several active log in sessions, the CIS will not know which host list to send [BDKS00]. This issue is addressed in this thesis.

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

The Caller ID system is a weak form of authentication because it requires one to trust the response of an untrusted system. For example, let an attacker gain *root* access to H_{i-1} and replace the CIS. The attacker then logs into H_i, so CIS_i sends a request to CIS_{i-1}. CIS_{i-1} can fool CIS_i in several ways. The easiest is to say that the user has logged into H_{i-1} locally and therefore the original source of the attack will not be learned. The next easiest is to return a host list of another user. This will certainly pass the authentication process, but lead an investigation to an innocent user.

Another method of fooling this system depends on implementation. The paper says that the host list is verified by checking that a user is running a process on that host. If 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 when verifying the list.

This protocol may add considerable delay to the log in time, especially if a host is unreachable and the CIS must wait to timeout.

2.4 The Identification Protocol

The Identification Protocol (`ident`) [Joh93] is a simple 2-way protocol that was designed to allow a server to identify the client-side user name of a network connection. The `ident` protocol was previously called the Authentication Server Protocol [Joh85] and was later renamed because of its actual functionality. The protocol, as shown in Figure 2.2, 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.

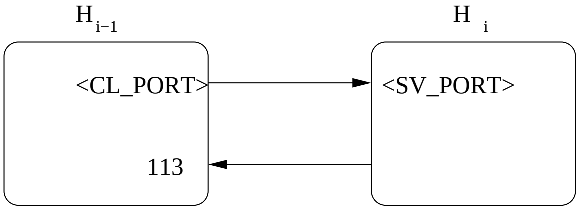

Figure 2.2. Identification Protocol request

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:

   `<CL_PORT>,<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:

   `<CL_PORT>,<SV_PORT>:USERID:UNIX:<USER_ID>`

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

   `<CL_PORT>,<SV_PORT>:ERROR:<ERROR_MSG>`

The grammar for the protocol is given in Figure 2.3.

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.

```
<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)
```

Figure 2.3. Identification Protocol grammar

Electronic Mail: *Sendmail* [sen] sends an `ident` request when it receives mail to trace forged mail. The `ident` response is placed in the email header.

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

Port Filters: Applications such as TCP Wrappers [Ven92] can log and filter network requests based on `ident` replies.

As 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 [Gol96]. 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 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 [Ope00] returns a string of 80 random bits in hexadecimal instead of the user name. The random token 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 [Eri00] 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. Other measures include always returning "OTHER" as the operating system type, returning "UNKNOWN-ERROR" for all types of errors, and returning a default user name instead of errors.

In theory, the ident protocol 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 [Mor98] is an extension to the ident protocol. 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. TCP SESSION TOKEN PROTOCOL

The protocol that is proposed in this thesis, the TCP Session Token Protocol (STOP), provides additional functionality to what is offered by the Identification Protocol (ident) [Joh93]. It can be run on any host with no modification of protocols, network topology, or kernel. It saves user- 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.

Section 3.1 will list the protocol design goals and specifications. Section 3.2 will discuss recursive traceback requests and Section 3.3 will discuss saving user- and application-level data. Section 3.4 will discuss the security of this protocol followed by three case studies in Section 3.5.

3.1 Protocol Design

3.1.1 Design Goals

The original protocol design goals were:

1. Must be backward compatible with the Identification Protocol as specified in RFC1413 [Joh93] because of its widespread usage and implementation.

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- 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 polices.

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

3.1.2 Specification

The `ident` protocol 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 3.1. The request message has the following format:

```
<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 ":"
```

Figure 3.1. Session Token Protocol grammar

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

<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 3.1. <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 or Intrusion Detection Systems (IDS) to compensate for the lack of socket logging on many machines. By utilizing this, gateways can collect tokens on all outbound or inbound connections. To prevent in-

Table 3.1
Session Token Protocol request types

Type	Description
ID	This request has the same behavior as the original ident protocol. The daemon saves the user name in a log file and returns a user 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>.

formation 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 protocol, 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 user 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 Internet Relay Chat (IRC) networks. A daemon that implements this protocol must have the following properties:

- Return a user token for all established outbound connections.
- User 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.
- Return an error for requests of TCP sessions that were not initiated by the local host (i.e. inbound connections).
- Return a user token to all requests that specify the remote IP address of the connection; this includes replacing error messages.
- Process requests in the original RFC 1413 format as `ID` type requests.
- Save additional user- and application-level data when `SV` or `SV_REC` requests are received (see Section 3.3).
- 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 3.2).

- Save tokens from recursive traceback requests with the returned user token. The recursive-based tokens must not be sent to the original requester.

- 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 user token is returned and the event is logged.

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

- Provide an option to return a user token instead of error messages.

- 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.

- 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 Limitations

The protocol described in this thesis has limitations, which will be described in this section. The traceback property only works if every intermediate host is running a STOP daemon. The chain of hosts can be identified to the first host not running the protocol. If every intermediate host in the chain is running the protocol, then the attacker can be identified even when he or she is not.

As will be shown in Section 3.4, the data from a STOP daemon cannot always be trusted. Attackers that gain *root* privileges to a host can replace the daemon with a rogue version. The validity of the daemon and saved data must be determined during an investigation.

When a connection chain is closed before the traceback is complete, the remaining hosts cannot be determined. Most operating systems do not save socket data after it has been closed, so the previous host cannot be identified. A feature to cache socket data would require a kernel modification, which is out of the design scope.

3.2 Traceback Requests

3.2.1 General

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. The only way to compensate for this is to 'walk' up the process tree and save information associated with all open sockets and pipes. Requests are then sent to any host connected via a TCP socket to one of the analyzed processes. See Section 4 for details on how the prototype implementation did this. Buchholz and Shields [BS01] have proposed a better solution to this problem by including the previous host IP address in every process structure, but this solution requires kernel modification.

The user 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 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.

3.2.2 Loop Detection

The traceback requests must contain a random session identifier to prevent cycles and a denial of service 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 seconds was used in the prototype implementation. If the daemon receives a duplicate request for a socket with the same session identifier and from the same host within the specified time, it must not process the request and return a `<user-reply>` type message.

If the procedure described in Section 3.2.1 is used to determine previous hosts, the scenario shown in Figure 3.2 will cause a loop. Let H_2 have a process that runs the following pseudo code:

```
listen (Port$_1$);
connect ($H_3$, Port$_2$);
listen (Port$_3$);
```

Let H_3 have a process that runs the following pseudo code:

```
listen (Port$_2$);
connect ($H_2$, Port$_3$);
```

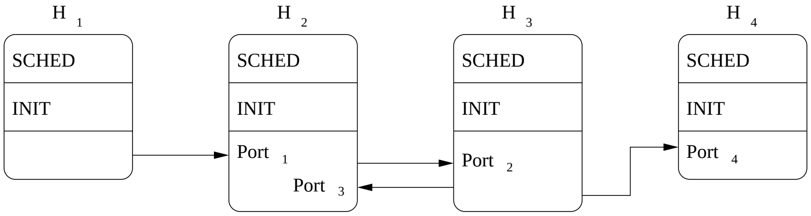

Figure 3.2. Process trees of 4 hosts in a network loop

```
connect ($H_4$, Port$_4$);
```

Let H_4 run the following pseudo code:

```
listen (Port$_4$);
```

Let the attacker connect 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.

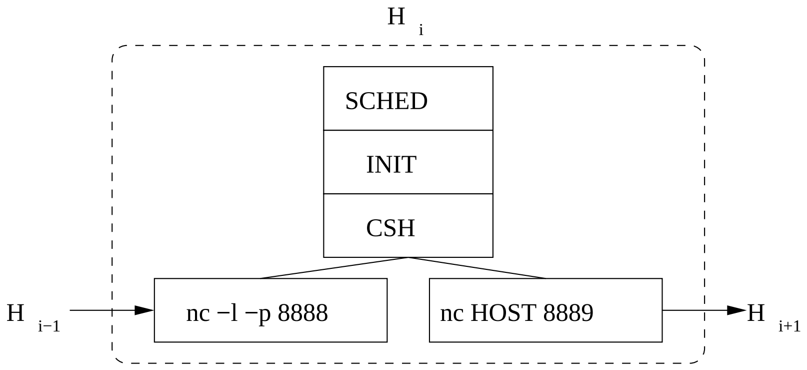

Figure 3.3. Process tree of command: `# nc -l -p 8888 | nc HOST 8889`

3.2.3 Resolving Interprocess Communication

Performing a simple 'walk' up the process tree may not be adequate when tracing malicious users. As shown in Figure 3.3, if an attacker ran the following simple command to 'pass through' host H_i, the daemon could not determine host H_{i-1}.

```
# nc -l -p 8888 | nc <H_{i+1}> 8889
```

This command uses *netcat* [Hob96] 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`

Table 3.2
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

or a local interface. IPC techniques such as shared memory are not addressed in this thesis.

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

3.3.1 General

A distinct feature of this protocol is the ability to save user- 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 3.2 lists data that is 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 request. This data is recommended for completeness, but if storage space is an issue then this data may not be saved.

3.3.2 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. There is little that can be done to prevent this, but measures can be taken to detect it.

The log entries can be protected by sending log entries 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 [SK99] [BY97]. These methods will not prevent the log from being modified, but will identify when an entry has been changed or deleted.

The easiest way to protect the process data files is to generate a one-way hash of them using SHA-1 [SHA93]. 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 taken of the file to verify it is the same value as when it was originally created. This will show when a modification has occurred, but not what was modified.

3.4 Security Analysis of Protocol

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. This section will analyze the effectiveness of the protocol when the daemon of host H_i has been killed or replaced. It is important to remember that the logs of any system that has had *root* access compromised are never fully trusted.

If the attacker kills the daemon, then this is the same situation 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.
- 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.

The above conditions would be identified during a thorough forensic analysis of the system. These scenarios show that this protocol is not a quick fix to the stepping stones scenario and must be used only as a tool, whose data must be verified.

3.5 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.

3.5.1 Simple Process Structure

The most basic traceback scenario is if a user logs into a system, gets a shell, and then logs into another host. An example of the process trees in this scenario are shown in Figure 3.4.

In this example, Alice is logged into H_1. She then uses the ssh protocol to log into H_2 and from there uses the telnet protocol to log into H_3. Let H_3 send a SV_REC request to H_2. A summary of the data saved is shown in Figure 3.5, 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. The full report is given in Appendix B.1.

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:  SV_REC: 92847523456
```

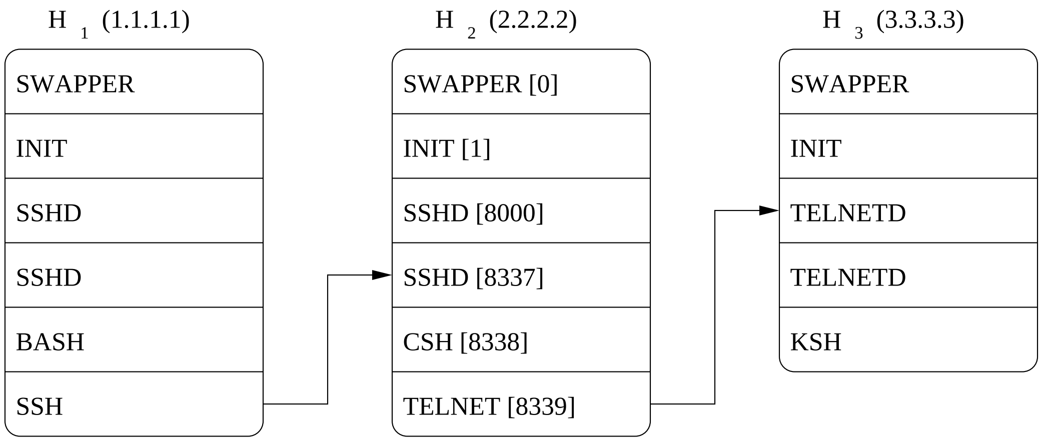

Figure 3.4. Simple process structure process tree

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 or 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 is 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 `1.1.1.1.1`:

```
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
```

Figure 3.5. Simple process structure process data

```
616, 22:  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*, *sshd*, *init*, and *swapper* processes will be analyzed, saved to a file, and a token will be returned to `2.2.2.2`. `1.1.1.1` will send a request to the host that connected to it through the *sshd* process.

3.5.2 Complex Process Structure

A more complex process structure can be found in Figure 3.6. This process structure contains 14 unique processes, three process groups, and six forms of interprocess communication 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 creates 9 children that can communicate via pipes. Process P_4 creates an Internet

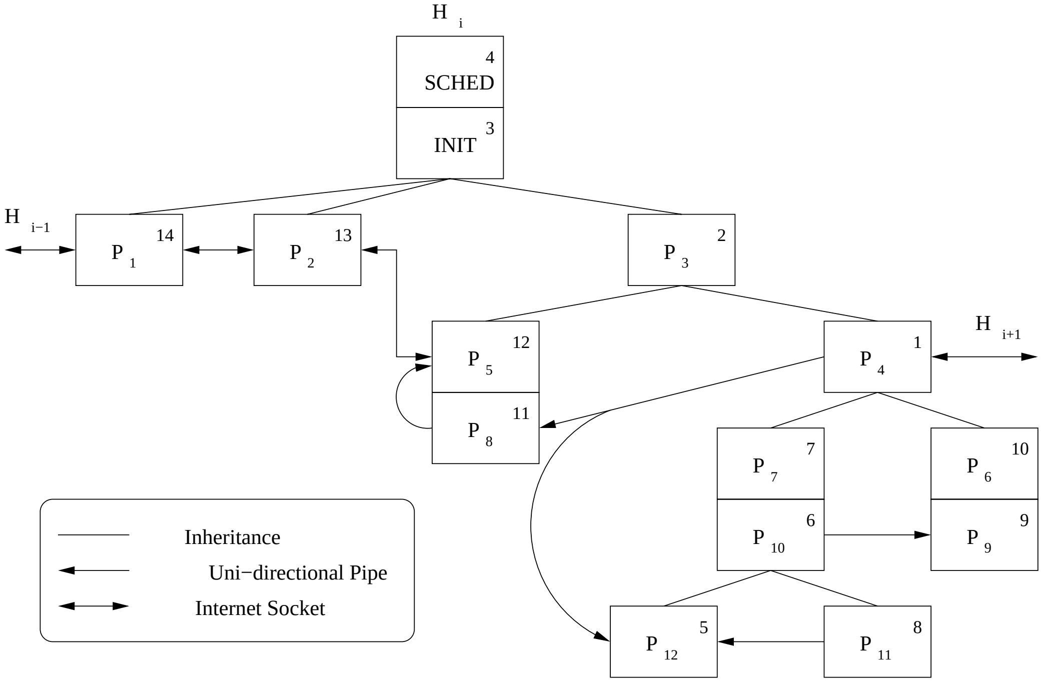

Figure 3.6. Complex process structure process trees

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 3.7. The actual numeric process identifiers have been replaced with the process labels shown in Figure 3.6. The full report can be found in Appendix B.2. As will be shown in Section 4.5, pipes are saved and resolved in OpenBSD by using the kernel memory addresses of `pipe` structures and Internet domain sockets are resolved by comparing the local and remote address and port tuples.

```
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
```

Figure 3.7. Complex process structure process data

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 with a local structure at kernel memory address `0xE07F2600` and remote structure at kernel memory address `0xE08CA780`, in addition to the requested connection. 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.

The pipe that P_{12} has open 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 open 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 was 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 has open 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.

3.5.3 Reverse Telnet

Reverse telnet [SMK01] 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. Let the server have a Common Gateway Interface (CGI) 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 3.8. 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.

Let the attacker, H_{i-1}, have IP address 1.1.1.1, the compromised server, H_i, have IP address 2.2.2.2, and let there be an IDS system on the victim's network that 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
```

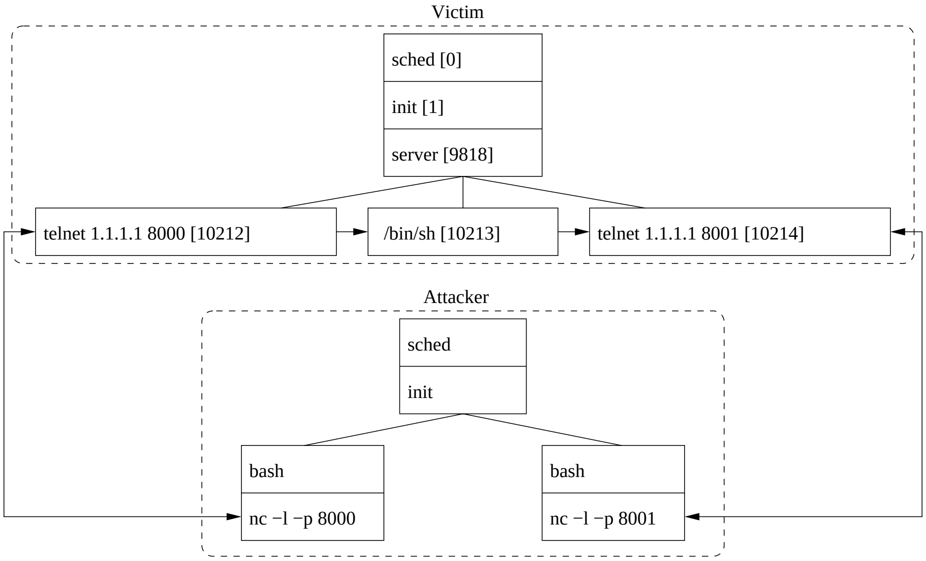

Figure 3.8. Reverse telnet process trees

A summary of the daemon output on a Linux box is shown in Figure 3.9 and the full report is available in Appendix B.3. As will be shown in Section 4.4, pipes are resolved in Linux by using the inode associated with it.

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 is saved and a token is returned to the IDS.

```
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
```

Figure 3.9. Reverse telnet process data

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. For example, Linux uses the process pseudo file system and Solaris uses stream-based sockets.

The functions that were written for the implementation can be broken up into two categories: the external interface and the internal interface. Within each of these categories, there are general functions that are common to all platforms and others that are platform-specific. An overview of the implementation, including runtime options, is given in Section 4.1. The external interface is described in Section 4.2 and the internal interface is described in Section 4.3. The implementation details for Linux, OpenBSD, and Solaris are presented in Sections 4.4, 4.5, and 4.6, respectively. Section 4.7 concludes with the measured performance results.

4.1 Overview

4.1.1 Description

The main design goals when implementing this protocol were for it to be secure, portable across multiple platforms, and able to be easily turned into a library. A library was an important goal so that it would be easy for existing `ident` programs to be modified to handle the new features.

Table 4.1
External and internal interface functions

External Interface		
Platform	**Function**	**Description**
General	`parse_req()`	Parses the request string
	`print_procdata()`	Prints process data
	`send_reqs()`	Sends recursive traceback requests
Specific	`process_req()`	Processes request

Internal Interface		
Platform	**Function**	**Description**
General	`resolve_lcl()`	Identifies IPC methods to resolve
Specific	`walk_ptree()`	Saves user and application data
	`find_proc()`	Identifies processes based on file descriptors

These goals were met by creating internal and external interfaces. Table 4.1 gives the functions and actions that each interface provides. The table also defines which actions are platform-specific and which are general. The external interface is used as the library API. It defines data structures and functions that all platforms use when interacting with the protocol. The internal interface defines a set of functions that are called by the external interface functions and allow for generic code to be reused. In general, the platform-specific code gathers process data and generic code identifies IPC methods to resolve, prints process data to files, and parses request strings.

The daemon was based on the *oidentd* `ident` daemon [McC00] 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 user 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. This 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 Interprocess Communication 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 user 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. Sample reports can be found in Appendix B.

For request types ID_REC and SV_REC, the process data is 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 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.

4.1.2 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 one process with the specified socket. 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 interprocess communication. 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.

4.1.3 Limitations

This implementation has limitations that are because of the types of platforms used, assumptions that made the program faster, and the scope of the project. The Linux implementation does not resolve local domain sockets because the process file system does not show which sockets are connected. None of the implementations resolve any System V forms of IPC, such as shared memory, because of the scope of the project. As mentioned in Section 4.1.2, files are not resolved and only one process is identified when searching for a socket.

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.

4.2 External Interface

This section describes the external interface for the protocol implementation. The general flow of execution will be described first, followed by the data structures and functions that are used. A detailed specification can be found in Appendix A.

4.2.1 External Interface Layout

The external interface was organized for the following sequence of events:

1. Request is received as an ASCII string.

2. The request string is parsed by `parse_req()` and a `reqtype` structure is filled with the request details.

3. The `reqtype` structure is passed to `process_req()` and a list of processes associated with the requested TCP connection are returned in a `procdata` structure.

4. If the request was for type `ID` or `ID_REC`, then a token is returned to the requester and the UID is saved.

5. If the request was for type `SV` or `SV_REC`, then the process data is saved using `print_procdata()` and the hash of the data is returned to the requester.

6. If the request was for type `ID_REC` or `SV_REC`, then `send_reqs()` is called to send requests to previous hosts.

4.2.2 External Interface Data Structures

Three data structures were used in the external interface. One was used to store the request details, and two were used to store process and file descriptor data. The structure fields can be found in Figure 4.1 and the specific types are specified in Appendix A.1.

reqtype

The `reqtype` data structure contains the details of the connection request. The actual request is sent to the daemon as an ASCII string. As will be shown in Section 4.2.3, the string is passed to the `parse_req()` function, which fills the appropriate

reqtype	procdata	fddata
type	parent	next
lcladdr	name	type
lclport	pid	lcladdr
remaddr	ppid	remaddr
remport	ptype	lclport
reqaddr	ruid	remport
sessid	euid	protocol
ruid	rgid	dir
pid	egid	lclunix
	dev	remunix
	prio	
	start	
	fd	

Figure 4.1. External interface data structures

fields of the `reqtype` structure. The fields of this structure can be found in Figure 4.1.

This structure is passed as an argument to all interface functions. Its contents include the IP address and port number of the remote host and local interface. There is also a field for the IP address of the host that requested the connection, because this could be different then the remote host if the optional IP address is given in the request. The type of request, such as `ID` or `SV_REC`, and the session identifier are also saved in this structure.

The `reqtype` structure also contains fields for process id and user id. These are for the respective values of the requested process. This was done as an optimization for the basic ID request so it does not have to allocate the larger structures that are described next. The reqtype structure has a size of 32 bytes.

procdata

As described in Section 3.3, a daemon that implements this protocol analyzes and saves state data about processes and file descriptors for ID_REC, SV, and SV_REC type requests. The platform-specific process data is copied to the generic procdata structure. The fields of the structure are found in Figure 4.1. This structure is a linked list, the head of which is the process that had the requested socket open. The next entry in the list is either the parent process or a process that was identified by resolving a method of IPC. The ptype field distinguishes between these two types of processes. Processes that were in the original process tree are marked as PRIMARY and those that were resolved are marked as SECONDARY.

This structure contains basic process data such as process name, PID, real UID, effective UID, real GID, effective GID, start time, terminal device, and priority. As described next, file descriptors are stored in a linked list of fddata structures. The pointer of the file descriptor list is found in fd.

The contents of this structure are filled by the walk_ptree() function. The procdata list is used when sending traceback requests and printing process details to a log file. The structure is 52 bytes long before the process name array is allocated.

fddata

The fddata structure holds data on open file descriptors; its fields can be found in Figure 4.1. The structure forms a linked list and the head is pointed to by the procdata structure. This structure contains data about all sockets and pipes. The type field differentiates among them.

The structure contains enough data to identify the local and remote ends of the communication channel. The type of data stored depends on both the file descriptor type and operating system. Therefore, the exact content will be explained in the platform-specific sections. All data is stored in host architecture byte order.

For all types of file descriptors, the direction (dir) and protocol type (protocol) are saved. The direction is set to either INBOUND, OUTBOUND, or UNKNOWN. The direction field is useful when sending out traceback requests and resolving local connections. The protocol type is set to either STREAM, DATA_GRAM, or RAW. For local domain sockets, the bounded paths are saved as ASCII strings. The fddata structure has a size of 32 bytes before the local domain strings are allocated.

4.2.3 External Interface Functions

This section describes functions that are part of the protocol external interface. Some functions are platform-specific and their details will be provided in the platform-specific sections. The functions are presented in the order of execution. A specification for the functions, including argument types, can be found in Appendix A.2.

parse_req

The protocol request is received from the network as an ASCII string. The request string is passed to the `parse_req()` function, which parses it and fills in the appropriate fields in the `reqtype` structure (Section 4.2.2). This function identifies the port numbers, remote address, request type, and session identifier from the string. After this function, the `reqtype` structure will be used when referring to the details of the request.

process_req

After the request string has been parsed by `parse_req()`, the resulting `reqtype` structure is passed to the `process_req()` function to collect process data. This function returns a `procdata` list that contains data on primary and secondary processes.

The details of this function are platform-specific, to optimize the performance. In general, the `process_req()` function has three main steps. The first is to identify the UID and PID of the requested process. If the request was an `ID` request, then there is nothing left to do. If it was not, then `process_req()` must save data by walking up the process tree using the `walk_ptree()` function. The third step is to resolve all local communication methods by using the `resolve_lcl()` function.

print_procdata

The `print_procdata()` function takes the `reqtype` structure and the `procdata` list as arguments. It prints data depending on the status of the `printflags` global variable. This variable is defined at execution time based on a configuration file. By default, all data is printed, but because of disk size constraints or privacy policies the amount can be restricted.

This function prints all variables defined in Section 3.3.1. Samples are given in Appendix B.

send_reqs

The `send_reqs()` function takes a `reqtype` structure and `procdata` list as arguments and searches the `procdata` list for TCP Internet domain sockets whose direction is either `INBOUND` or `UNKNOWN` and whose local and remote addresses are different. When one is found, a request is created using the session identifier and type found in `reqtype`. The request is sent to the remote end of the connection. The returned tokens are logged to a file.

4.3 Internal Interface

This section describes the internal interface for the `STOP` protocol. The functions and data structures presented here are called by the external interface functions.

4.3.1 Internal Interface Data Structures

This section describes a data structure that is used when resolving methods of IPC. The specification can be found in Appendix A.3.

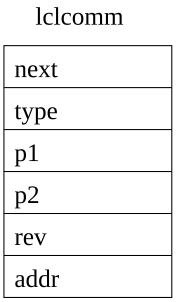

Figure 4.2. Internal interface data structures

lclcomm

The `lclcomm` structure is a small linked list. It is used within the `resolve_lcl()` function to keep a sorted list of local connections to resolve. It contains six fields and has a size of 14 bytes. The fields can be found in Figure 4.2. As will be seen, this structure allowed the daemon to easily determine when it did not need to resolve a connection because the remote end had already been analyzed.

The `lclcomm` structures are stored in a list sorted according to two fields, `p1` and `p2`, where `p1 < p2`. The list is first sorted by increasing values of `p1` and if multiple structures exist with the same `p1` value then the `p2` value is compared.

The contents of `p1` and `p2` are platform-specific. In general, they contain port numbers for Internet domain sockets and kernel memory addresses or inode values for pipes and local domain sockets. The local port or memory address should be placed in `p1` and the remote port or memory address should be placed in `p2`. If `p2 > p1`, then they are reversed and the `rev` field is set to 1. Otherwise, `rev` is set to 0. The `type` field identifies the type of connection.

The address of Internet domain sockets is copied into the addr field. Only sockets to localhost (127.0.0.1) or those that have the same local and remote IP addresses are resolved. In both cases, only one address is needed. For local domain sockets and pipes, the contents of lcladdr and remaddr in the fddata structure are copied into p1 and p2, with rev set accordingly.

Two structures are *opposites* if they have equal p1 and p2 values, but opposite rev values. For example, if there is a socket to localhost, the lclcomm structures that are created for each end of the socket are *opposites* because they contain the same port numbers, but have a different perspective about which is local and remote.

4.3.2 Internal Interface Functions

This implementation defined three functions in the internal interface. This section will describe what they do. The formal specification can be found in Appendix A.4.

walk_ptree

The walk_ptree() function is platform-specific and takes as arguments a PID and a list of processes that have already been seen. The function enters data into the procdata structures (Section 4.2.2) by analyzing processes starting with the PID argument. After a process has been analyzed, the parent is analyzed and the procedure is repeated until process 0 is reached or until a process that has already been analyzed is reached. The latter case is determined by using the list of seen processes that was passed as an argument.

This function is originally called from within process_req() and is later called from within resolve_lcl() to analyze the identified secondary processes.

resolve_lcl

The resolve_lcl() function is responsible for resolving forms of Interprocess Communication. It takes the procdata list from walk_ptree(), identifies local socket and pipe connections, and calls find_proc() to identify the process on the other end. After the process has been identified, walk_ptree() is called to gather data on the new process tree. The new process tree is analyzed for additional IPC connections and the procedure continues until all IPC connections have been resolved.

To do this efficiently, the lclcomm data structure is used. The resolve_lcl() function maintains a seen list and a todo list of lclcomm structures. The todo list contains sockets or pipes that need to be resolved and the seen list contains all local sockets and pipes that have been added to the todo list. After a connection is resolved, it is removed from the todo, but not the seen list.

This function searches a procdata list for Internet domain sockets to localhost (127.0.0.1), Internet domain sockets with the same local and remote IP addresses, local domain sockets, and pipes. When one of these connections is found, data about the connection is placed in a lclcomm structure, as described in Section 4.3.1.

If the socket or pipe is not in the seen list, it is added to both the seen and todo lists. If it is in the seen list, then the todo list is examined. If the todo list already contains this connection, then the new one is ignored. If the opposite connection is in todo (equal values of p1 and p2 but opposite values of rev) and the connection is for a socket, then we remove the opposite from the todo list and add the new connection to the seen list. This case occurs when we have already analyzed both ends of the socket. Opposite ends of pipes are not removed from the todo list because there may exist several processes at each end.

find_proc

The find_proc() function is platform-specific. It takes a (void *) pointer, a type identifier, and an empty array for PIDs as arguments. The contents of the pointer are based on the type identifier and platform.

In general, this function will cycle through all processes on the system and identify the ones with a file descriptor trait specified by the search type and the value in the (void *) pointer. Each process that is found with the trait is added to the PID array that was passed as an argument. Details are given in the platform-specific sections.

4.4 Linux Implementation

The protocol presented in this thesis was implemented on the Debian Linux 2.2 kernel [lin00]. The Linux platform is distinctive from OpenBSD and Solaris because it includes a process pseudo file system from which state data is gathered. This section will describe the location of process data in Linux, followed by the implementation details.

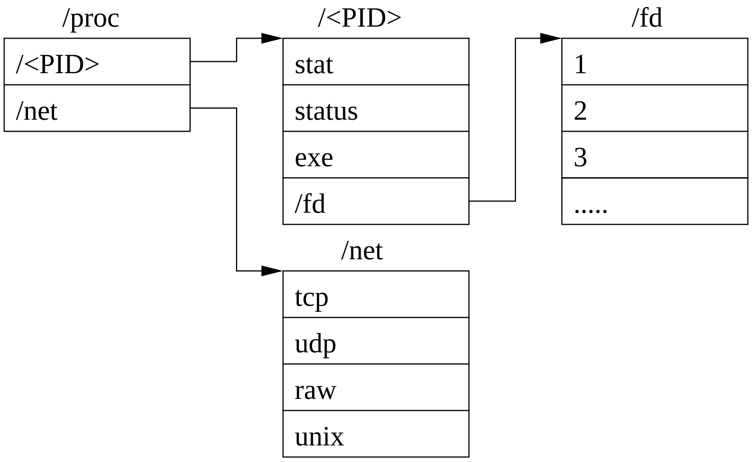

Figure 4.3. Linux proc file system layout

4.4.1 Process Structure

The Linux operating system provides access to process data through a pseudo file system, `procfs`. The `procfs` is an abstraction into a file-like format of the kernel memory-based process structures. It provides a uniform interface to process details.

As shown in Figure 4.3, the `/proc/` directory is the mounting point for the `procfs` and contains files and directories for process and system data. Process related data is stored in a directory named by the process identification number (PID). System directories and files have names such as `net/`, `sys/`, `meminfo`, and `version`.

The process directories contain files with environment settings, process status, memory maps, and the original command line. The `stat` and `status` files contain, among other data, the parent PID, tty device, start time, scheduling priority, and real and effective user and group identifiers.

```
# ls -l /proc/213/fd/
lrwx------ 1 root root 64 Apr 1 10:45 0 -> /dev/pts/5
lrwx------ 1 root root 64 Apr 1 10:45 1 -> /dev/pts/5
lrwx------ 1 root root 64 Apr 1 10:45 2 -> /dev/pts/5
lr-x------ 1 root root 64 Apr 1 10:45 3 -> pipe:[3703552]
lrwx------ 1 root root 64 Apr 1 10:45 4 -> socket:[3703580]
l-wx------ 1 root root 64 Apr 1 10:45 5 -> /root/prog/main.c
```

Figure 4.4. Contents of an `fd` directory in Linux

A subdirectory, `fd/`, contains symbolic links to open file descriptors. The links are named using the file descriptor indices. For example, `/proc/213/fd/0` is a symbolic link to the standard input for process 213. A sample listing of an `fd` directory can be found in Figure 4.4. The `pipe:[3703552]` entry shows that the process has a pipe with inode number 3703552 opened. In this example, the reading end is open and the writing end is closed. If another process has the writing end open, it will have an `fd` entry with the same inode, but with write permissions. The `socket:[3703580]` entry shows that the process has a socket with inode number 3703580, but does not identify whether it is an Internet domain or local domain socket. As will be shown, this can be determined by looking in the `/proc/net/` files for inode 3703580. The last entry shows that the `/root/prog/main.c` file is open for writing.

The details of a socket inode can be found in the `/proc/net/tcp`, `/proc/net/udp`, `/proc/net/raw`, and `/proc/net/unix` files. The first three files contain entries for every TCP, UDP, and RAW Internet domain socket, with the local and remote addresses and ports, inode number, and UID. The `/proc/net/unix` file contains an entry for every local domain socket with the inode, bound file system path, and state

Table 4.2
Linux data collection functions

Function	Description
`find_raw()`	Searches `/proc/net/raw` for a socket with specific addresses and ports. Returns the socket inode and UID.
`find_tcp()`	Searches `/proc/net/tcp` for a socket with specific addresses and ports. Returns the socket inode and UID.
`find_udp()`	Searches `/proc/net/udp` for a socket with specific addresses and ports. Returns the socket inode and UID.
`read_fd()`	Reads the `fd` directory entries and calls `save_pipe()` or `save_socket()` accordingly.
`read_name()`	Parses `exe` file in process directory and adds process name to `procdata` structure.
`read_stat()`	Parses `stat` file in process directory and adds the parent PID, device, priority, and start time to the `procdata` structure.
`read_status()`	Parses `status` file in process directory and adds the real and effective UID and real and effective GID to the `procdata` structure.
`save_pipe()`	Saves pipe related data in an `fddata` structure.
`save_socket`	Searches the `/proc/net/`{`tcp, udp, raw, unix`} files for an inode entry and saves the appropriate data in an `fddata` structure.

data. Unfortunately, it does not show the connection status, so we are not able to determine which local domain sockets are connected.

4.4.2 Data Collection

Process state data in Linux was collected by parsing files in the process pseudo file system. To make the program simple, functions were created for each file that needed to be parsed. Table 4.2 shows the functions that were added to the Linux implementation. The daemon needs to read the contents of all `/proc/` directories, and therefore needs *root* permissions.

process_req

The process_req() function calls the find_tcp() function for ID type requests to determine the UID of the requested socket, because the UID is saved in /proc/net/tcp. For non-ID type requests, process_req() calls find_proc() to determine the PID of the process with the requested socket. The PID is passed to walk_ptree() to save the process state and resolve_lcl() is called to resolve any local IPC methods.

walk_ptree

The walk_ptree() function is passed the PID of a process and a list of already seen processes. It saves state data about the process and its parent processes. Brief descriptions of the functions called can be found in Table 4.2.

The process name is saved using read_name() and the user and group identifiers are saved using read_status(). The parent PID, device, priority, and start time are saved using read_stat() and lastly, the read_fd() function is used to save data about open file descriptors. As shown in Table 4.3, the associated inode for local domain sockets and pipes is saved in the remaddr field of the fddata structure. The lcladdr field is set to 0.

find_proc

The find_proc() function searches all process subdirectories in the /proc/ directory for a specific file descriptor inode. Table 4.4 shows the data type and contents of the void pointer.

Table 4.3
Linux `fddata` address values

	fddata	
	lcladdr	**remaddr**
Internet domain socket	IP address	IP address
Local domain socket	0	inode number
pipe	0	inode number

Table 4.4
Linux `find_proc()` argument type

	Data type	**Contents**
Internet domain socket	`reqtype`	Connection tuples
Local domain socket	`uint32_t`	inode number
pipe	`uint32_t`	inode number

When the function is called to find the PID for a socket, the inode of the socket is first determined by calling `find_tcp()`, `find_udp()`, or `find_raw()`. Each `fd` subdirectory is then searched for an entry, such as `socket:[3703580]`. This is clearly an expensive operation because of the number of context switches that must occur while opening each process directory and reading the link of each file descriptor. We can optimize the process slightly because we already know the UID of the process. Before we read any file descriptors, the `status` file is first parsed to check if this process has the same UID as the one we are looking for.

When `find_proc()` is called to find processes with a pipe open, the function is passed an inode value. The function searches all processes for a file descriptor entry with the appropriate link, such as `pipe:[3703552]`.

4.4.3 Conclusion

The Linux implementation was the only one that utilized a process file system. Solaris also has procfs, but did not contain the required data. Unlike Solaris, Linux does not provide access to kernel memory, besides directly reading /dev/kmem. This technique is clearly not portable because Linux is not distributed with the header files for process structures unless the kernel source code is installed.

The Linux implementation was the easiest to develop, debug, and understand. The functions are simple because they typically only open a file for reading and use fscanf() to extract the data. The /proc/net/ files are also convenient because they list the UID of the socket. To prevent enumeration methods as described by Goldsmith [Gol96], the files should also contain the socket direction.

Though the procfs provides a convenient interface to most process data, it did not contain all of the data in which we were interested. The largest shortcoming is that the connection between local domain sockets is not shown. Therefore, an attacker can stop the traceback by running two processes that communicate via a local domain socket. Another useful feature would have been a method to easily group sets of processes by UID. This would have saved time when searching for sockets because the UID comparison could have been done in kernel memory instead of user memory.

4.5 OpenBSD Implementation

The protocol was implemented on a 4.4 BSD kernel using the OpenBSD 2.8 platform [Ope00]. The OpenBSD platform was distinctive from the other platforms because it provided an interface to kernel memory and its sockets were implemented

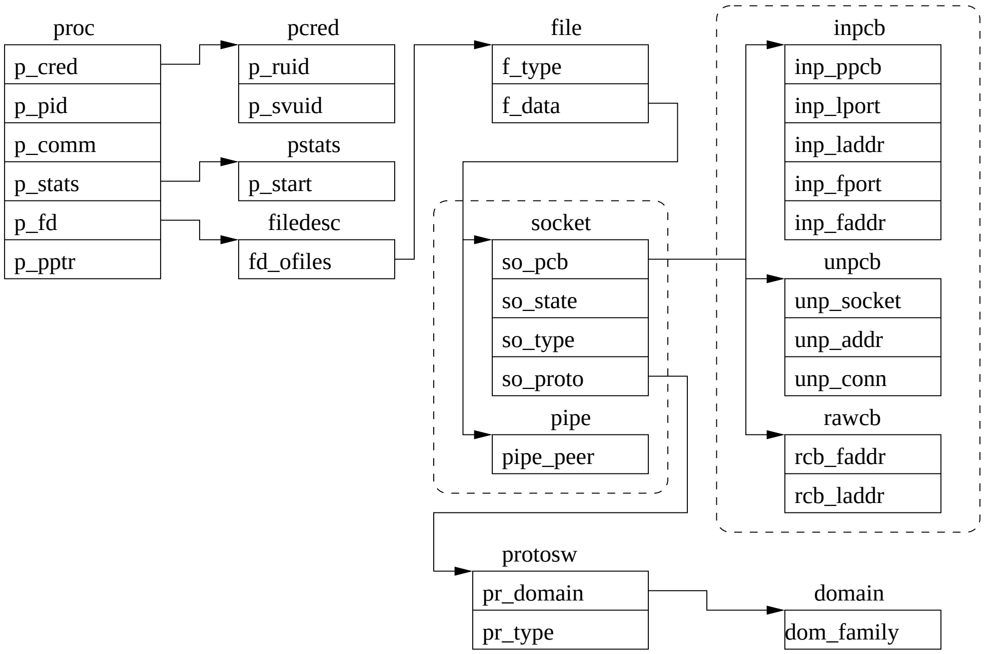

Figure 4.5. OpenBSD process structure

in a non-stream based stack. The first part of this section will give the process organization of OpenBSD, followed by data gathering methods.

4.5.1 Process Structure

Processes

Processes in OpenBSD are organized by `proc` structures in kernel memory. The `proc` structures are dynamically created and each contain data for a specific process. Figure 4.5 shows the structures and fields of the `proc` structure of interest. The `proc` structure contains pointers to the parent process (`p_pptr`), child processes, and sibling processes. The structure also contains the process name (`p_comm`) and PID (`p_pid`).

The `p_cred` field in `proc` is a pointer to a `pcred` structure for the process credentials. This structure contains the processes UID, GID, and a pointer to the user credentials structure, ucred. The ucred structure contains the effective user and group ids. The pstats structure contains the process start time.

File Descriptors

The p_fd field in proc is a pointer to a filedesc structure, which manages the file descriptors for all open files, sockets, and pipes. Among other fields, it contains an array of file structures for open descriptors.

The file structure is a generic structure for vnodes, sockets, and pipes. The structure contains general file related data such as a pointer to user credentials, pointers to file operation functions, and the address of a file-type specific structure (f_data). The type of the specific structure is identified by the f_type field and can point to either a vnode, pipe, or socket structure.

The vnode structure is used when regular files are open by a process. It provides an abstraction to the specific file system being used. The pipe structure is used to represent a one-way pipe. It contains fields for buffers, state data, and a pointer to the pipe structure on the remote end. We can easily determine when two processes are connected via a pipe if one process's pipe structure is at the same address as the other's remote pipe pointer.

Sockets

The f_data field points to a socket structure for all socket types. It contains fields for socket state, creator UID, and data buffers. The OpenBSD socket structure is

Table 4.5
OpenBSD protocol control block pointer types

Domain	**stream**	**datagram**	**raw**
Internet	`inpcb`	`inpcb`	`rawcb`
Local	`unpcb`	`unpcb`	N/A

the only one found in this research that had a state bit for direction. The socket is outbound when the `SS_CONNECTOUT` bit is set. The `so_type` field identifies the socket as stream, datagram, or raw. The `socket` structure also contains pointers to the `protosw` protocol switch structure and a protocol control block.

The protocol switch table, `protosw`, is used for communicating between protocol layers and with the system. It contains pointers to specific protocol functions that pass information to the layers above and below. It also contains fields that specify the protocol type, protocol number, flag options, and a pointer to a `domain` structure.

The `domain` structures in OpenBSD form a list where each domain supported by the system has an entry. The structure contains domain-specific method pointers and a field that specifies the domain type. Examples of domain types include *Internet* and *local*.

The protocol control block pointer in the `socket` structure points to an object whose type is based on the protocol and domain type. Therefore, we must first examine the `protosw` and `domain` structures before this field. Table 4.5 shows the object type based on protocol and domain.

The `inpcb` structure is a generic Internet domain protocol control block and is stored in a hash table. This structure contains the local and remote ports and addresses, routing data, and a pointer to the socket structure. Each `inpcb` structure contains a pointer to a specific protocol control block (`inp_ppcb`), based on the socket type. For example, when the socket type is stream, the `inp_ppcb` structure points to a `tcpcb` structure.

The `rawcb` structure is the protocol control block used for raw sockets. It contains fields for the doubly linked list it is stored in, a pointer to the original socket, and data structures for the remote and local addresses.

The `unpcb` structure is the protocol control block used for local domain sockets. This structure contains a pointer to the original socket, a pointer to a `vnode` structure if the socket is bound to a file location, and structures for buffer data. If the socket is connected, this structure also contains a pointer to the protocol control block of the remote socket, `unp_conn`.

4.5.2 Data Collection

Process data was collected by reading data structures directly from kernel memory. OpenBSD provides uniform access to kernel memory using the `KVM` library. The library must first be opened using `kvm_open()` and then data is transfered from kernel memory to user memory using `kvm_read()`. `kvm_getprocs()` returns a list of processes that match a certain property. For example, it can return sets of processes with a specific UID, process group ID, or all processes. Table 4.6 shows the functions that were written for the OpenBSD implementation.

Table 4.6
OpenBSD data collection functions

Function	**Description**
`find_tcp()`	Calls `sysctl()` system call to determine the UID of a TCP socket
`save_pipe()`	Saves `pipe` structure addresses in `fddata` structure
`save_socket()`	Saves appropriate address data for local and Internet domain sockets in a `fddata` structure.

OpenBSD contains a `sysctl()` system call for determining the UID of a TCP socket. The *Management Information Base* (MIB) string for the system call is {`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_IDENT`} and a `tcp_ident_mapping` structure is passed as the argument. The system call returns the UID of the socket, but not the PID.

process_req

The implementation of `process_req()` in OpenBSD first calls the `find_tcp()` function. The `find_tcp()` function takes a `reqtype` structure as an argument and places the socket data into the `tcp_ident_mapping` structure. It then calls the `sysctl()` system call to identify the UID of the socket.

If the request type was `ID`, then `process_req()` returns. For non-`ID` type requests, `kernel_init()` is called to open the `KVM` library file descriptor. The `kvm_open()` function needs direct access to `/dev/mem` and therefore needs *root* privilege. To handle this, the `kernel_init()` function changes from effective user *nobody* to *root* before it calls `kvm_open()`, and then resumes as user *nobody*.

The find_proc() function is used to find the PID of the requested socket and walk_ptree() is called to save the process tree starting at the PID. The process_req() function concludes by calling resolve_lcl() to resolve all pipes and sockets.

walk_ptree

The walk_ptree() function is passed a PID and a procdata list of processes that have already been seen. The PID is passed to kvm_getprocs() to read the associated proc structure from kernel memory. The pointers in the structure can no longer be used in the traditional manner, because they point to addresses in kernel memory. Instead, kvm_read() is used to copy data from the kernel memory address to a buffer in user memory.

For each process, the PID and process name are determined directly from the proc structure. The real and effective UID are determined from the pcred structure and the process start time is determined from the pstats structure.

The file descriptors are analyzed next by reading the filedesc structure and the list of file structure pointers from kernel memory. The list of pointers is cycled and each file structure is read from kernel memory. Using the f_type field we can determine if the descriptor is from a file, pipe, or socket. For pipes, save_pipe() is called and save_socket() is called for a socket. Table 4.7 contains the content that was saved in the address fields of the fddata structures.

The save_pipe() function is passed the file structure as an argument, and it reads the pipe structure from kernel memory using the f_data pointer. As mentioned

Table 4.7
OpenBSD `fddata` address values

	fddata	
	lcladdr	**remaddr**
Internet domain socket	IP address	IP address
Local domain socket	`socket` address	`socket` address
pipe	`pipe` address	`pipe` address

in Section 4.5.1, the `pipe` structure contains a pointer to the `pipe` structure on the remote end. The address of the local `pipe` structure is saved in the `lcladdr` field of `fddata` and the address of the remote `pipe` structure is saved in the `remaddr` field. When we need to resolve the pipe, we will search processes for a `pipe` structure at the remote address.

The `save_socket()` function is more complex because of the different combinations of protocol types and domains. The `socket` structure is read from the `f_data` pointer and the protocol switch table is read from the `so_proto` pointer in `socket`. The `domain` structure is read from the `pr_domain` field in the protocol switch table.

When an Internet domain socket is found, the corresponding `inpcb` or `rawcb` structure is read from kernel memory using the `so_pcb` pointer in `socket`. The local and remote addresses and ports are copied to the `fddata` structure and stored in host order.

The direction of stream sockets are determined by using the `SS_CONNECTOUT` flag and the direction of datagram and raw sockets is determined based on the existence of a remote address. If a remote address and port exist, then the socket is outbound.

Otherwise, it is set to inbound. This is because a listening datagram or raw socket does not have state and therefore does not save data about who connects to it. Therefore, only outbound sockets will have the remote address field filled.

When a local domain (Unix domain) socket is found, the `unpcb` structure is read from memory. The `lcladdr` field of `fddata` is filled with the kernel memory address of the `socket` structure. If the socket is bound, then the path is read from the `mbuf` structure pointed to by `unp_addr` and stored in the `lclunix` field of `fddata`. If the socket is connected, the remote `unpcb` structure is read from `unp_conn`. The kernel memory address of the remote socket structure is saved in the `remaddr` field of `fddata` and the remote bounded path is saved in `remunix`. When we are resolving the socket, we will search for a `socket` structure at the address saved in `remaddr`.

find_proc

As mentioned in Section 4.2.3, `find_proc()` finds the processes that have a specific socket or pipe open. It uses the `kvm_getprocs()` function to get a list of processes to search. The `kvm_getprocs()` function can either return all processes or only processes with a specified UID. To narrow the search for TCP sockets, the UID is determined by calling the `find_tcp()` function. Table 4.8 shows the data type and content of the void pointer argument.

For each process returned by `kvm_getprocs()`, the `filedesc` structure and list of `file` pointers are read from kernel memory. Each `file` structure in the list is then processed. Based on the `f_type` field we can quickly ignore files of a type different

Table 4.8
OpenBSD find_proc() argument type

	Data type	**Contents**
Internet domain socket	reqtype	Connection tuples
Local domain socket	uint32_t	socket address
pipe	uint32_t	pipe address

than what we are looking for. When searching for a local domain socket or pipe, the address of the socket or pipe structure is compared with the address passed as an argument. When searching for an Internet domain socket, the inpcb protocol control block is read from kernel memory and the addresses and ports are compared with those passed as an argument in the reqtype structure.

When the appropriate process is found, it is placed in the PID array. The function returns after finding one socket process, but continues to search when looking for pipes.

4.5.3 Conclusion

Accessing kernel memory as we have done here does not guarantee valid data. For example, between the time that we read the list of file pointers to when we read the final file structure from memory, the file could close and the data would not be valid. The process pseudo-file system that Linux offers can guarantee valid data, but is much slower.

This procedure would be easier if each file descriptor stored a list of processes that have the file open. The list could be updated when the reference count is updated and we would no longer have to search through all the file descriptors for the PID.

The layout of the OpenBSD file descriptors was much more intuitive than was seen with Solaris. OpenBSD provides clear pointers to the Protocol Control Blocks, while Solaris requires the daemon to traverse the stream stack and then access the Protocol Control Blocks. OpenBSD must also be noticed for its `sysctl()` call to identify the UID of a socket and the direction bit in the socket structure. These features were written for the original `ident` protocol and were easily adapted to this protocol.

4.6 Solaris Implementation

This protocol was also implemented on the Solaris 2.7 operating system [sol]. The Solaris platform is distinctive because it offers an interface to kernel memory and uses a streams-based socket approach. The first part of this section will describe the process structure, and then the implementation details will be provided.

4.6.1 Process Structure

Processes

Processes in Solaris are organized in the traditional UNIX fashion, with `proc` structures in the process table. The process table is in kernel memory and is therefore not directly available from a user process. Figure 4.6 shows the fields and structures in which we are interested.

Solaris, like Linux, provides a process pseudo file system, but it could not be utilized for this program. The Solaris process file system is not as advanced as Linux and does not contain data about file descriptors. Therefore, it did not meet the needs of this program and was not used even for basic data such as parent PID and UID.

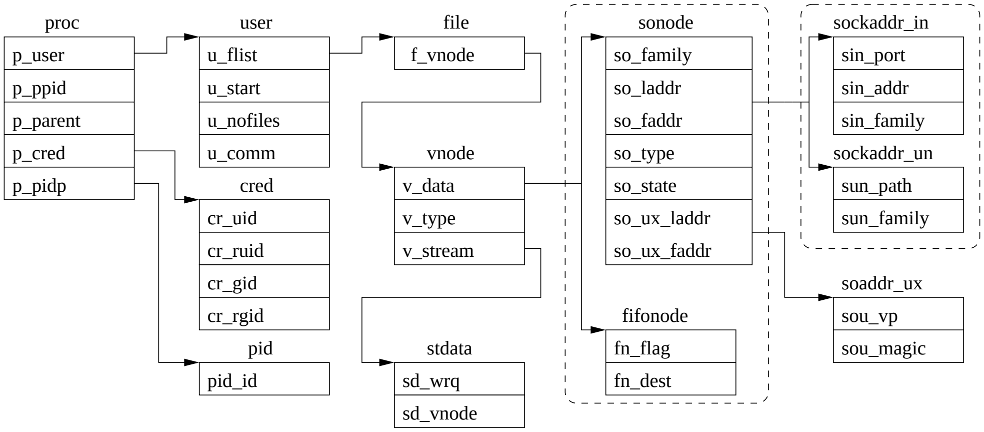

Figure 4.6. Solaris process structure

The `p_cred` field in the `proc` structure points to a `cred` structure that contains the process credentials, including the effective and real user and group identifiers. The `p_pidp` field points to a `pid` structure that contains the process identifier (PID) and process group information. The `p_ppid` field contains the PID of the parent process and `p_parent` is a pointer to the parent `proc` structure.

The `p_user` field points to the `user` structure that contains the process name (`u_comm`), start time (`u_start`), and a list of open files (`u_flist`). The `u_flist` field points to an array of `file` structures with size `u_nofiles`. A `file` structure is allocated for every open file descriptor and contains a pointer to the `vnode` structure (`f_vnode`), user credentials (`f_cred`), and the current read and write offsets (`f_offset`).

File Descriptors

A vnode structure is allocated for every open file descriptor to act as a generic interface to all file system types, pipes, and sockets. It contains pointers to specific functions, but we are only concerned with the data it stores. The v_type field identifies what type of file descriptor is open, for example regular file, socket, directory, or FIFO (pipe). The v_stream field points to the head of the associated stream and the v_data field points to a structure that is specific to the type of open file: a fifonode structure for pipes, a sonode structure for sockets, and an inode structure for regular files.

The fifonode structure is used for two-way FIFO channels and one-way pipe channels. The exact type can be determined using the fn_flag field. This structure contains fields for the buffers, reference counts, and the fn_dest field is a pointer to the vnode at the end of the pipe. This was used to identify the processes that had the remote end of the pipe open.

When the file descriptor is for a socket, the v_data field points to a sonode structure. The so_family field identifies the socket as local domain (Unix domain) or Internet domain. The so_type field identifies the socket as stream, datagram, or raw. The sonode structure contains fields for local and remote address structures. These fields are: so_laddr, so_faddr, so_ux_laddr, and so_ux_faddr and they point to sockaddr_in, sockaddr_un, or soaddr_ux structures. These fields should contain all of the data we need, but this is not the case because the data is not always accurate and in some cases not even valid. To get accurate data, we must examine the stream queues and the associated protocol control blocks.

Streams

Streams are used to create a bi-directional communication path between networking layers. Each layer represents a different protocol, for example one layer could be TCP and the next layer could be IP. Figure 4.7 shows a stream example for an Internet domain socket. Each layer communicates with the layers above and below by using two uni-directional communication channels using `queue` structures. The head of the stream is an `stdata` structure. It contains many administrative fields and we are most interested in the `sd_wrq` and `sd_vnode` fields. The `sd_wrq` field is a pointer to the head of the write queue, or the downward uni-directional path. Data is received into the `stdata` structure from below using buffers that are allocated in this structure. The `sd_vnode` field points back to the `vnode` structure.

The `queue` structure contains four fields of interest. A stream is a linked list of `queue` structures, which are linked together using the `q_next` field. The `queue` structure itself is generic, but the `q_ptr` field points to a layer specific protocol control block structure. For example, in the IP layer, the `q_ptr` field points to an `ipc_s` structure. The `q_stream` field is a backward pointer to the stream head. The `q_qinfo` field is a pointer to a `qinit` structure that contains information about the layer. The `qi_minfo` field in `qinit` is a pointer to a `module_info` structure that contains information about the size, identification number, and name of this layer. The name field, `mi_idname`, is the protocol name in ASCII, `ip` for example.

Figure 4.7 shows the stream associated with an Internet domain socket starting at the `stdata` stream head. The first `queue` structure holds the buffers and data associated with the stdata structure above it. The mi_idname string for this layer is strwhead. The second layer is found by following the q_next field and it is for TCP, UDP, or ICMP protocols. If the module_info structure is read, then the mi_idname string is either: tcp, udp, or icmp. icmp is used when a RAW socket is opened.

The third layer is the IP layer and the mi_idname string is set to ip. At this layer, the q_ptr field points to an ipc_s structure and the local and remote addresses and ports can be determined. The ipc_s structure also contains a backward pointer to the read and write queue structures for the layer.

The ipc_s structures are organized in a hash table and we can use the table to lookup a specific connection. The hashing function is the exclusive-OR of the local and remote ports and addresses. Therefore, if we are given the ports and addresses of a socket, we can easily identify the ipc_s structure in the hash table. The process associated with the ipc_s structure is identified by using one of the backward queue pointers to read the q_stream field, which points to the stream head. The stream head contains a pointer to the vnode structure. All processes are searched for a vnode value equal to the one associated with the desired ipc_s structure.

Figure 4.8 shows the stream associated with a local domain socket. As shown in Figure 4.9, a local domain socket stream has four layers. When the write stream is followed, the fourth layer is the read input to the remote end of the local socket. Exactly as with Internet domain sockets, the first queue structure is for the writing stream head. The second layer is a transport layer for the write stream of the local end of the socket. The third layer is the transport layer for the read stream of the remote

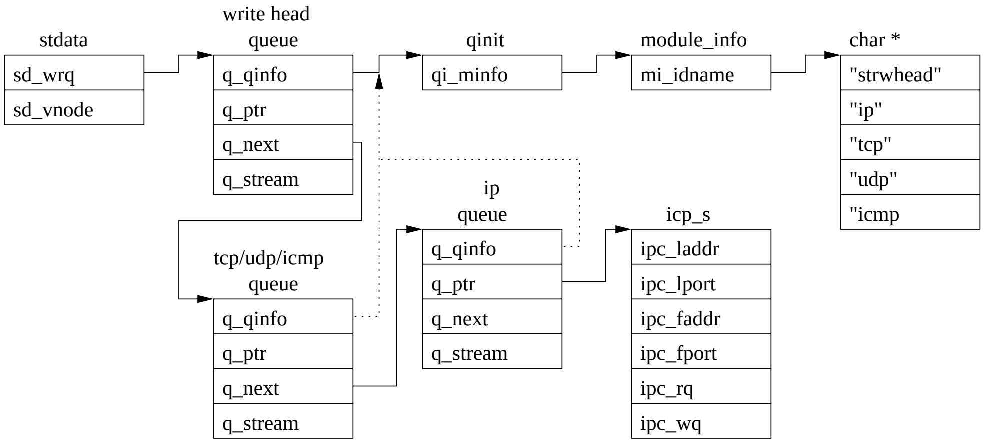

Figure 4.7. Solaris Internet domain stream structures

end of the socket and the fourth layer is the read stream head for the remote end of the socket. The `q_stream` field in the fourth layer, points to the `stdata` structure for the remote end of the socket and because the `stdata` structure contains a pointer to the `vnode` associated with it, we can identify processes with the other end of the socket open by comparing the `vnode` value in their `file` structures.

4.6.2 Data Collection

As mentioned in Section 4.6.1, process data is stored in kernel memory. We therefore do not have direct access to the data from our user space daemon. Data in kernel memory is copied to user memory using the `KVM` library functions. The memory handle is opened using `kvm_open()` and data is transfered using `kvm_read()`. A specific process's structure can be read using `kvm_getproc()` or all processes can by cycled through using `kvm_setproc()` and `kvm_nextproc()`. Of course, any pointer in the structures that are copied from kernel memory are no longer valid and `kvm_read()` must be used to follow them.

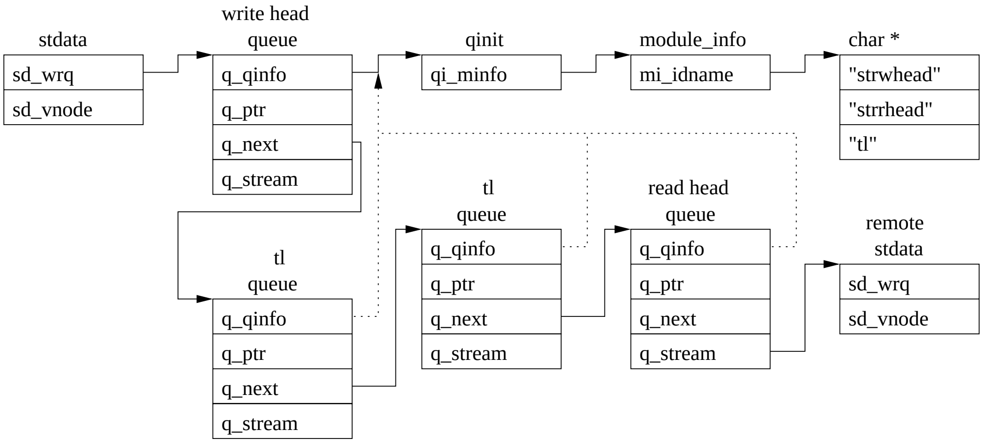

Figure 4.8. Solaris local domain stream structures

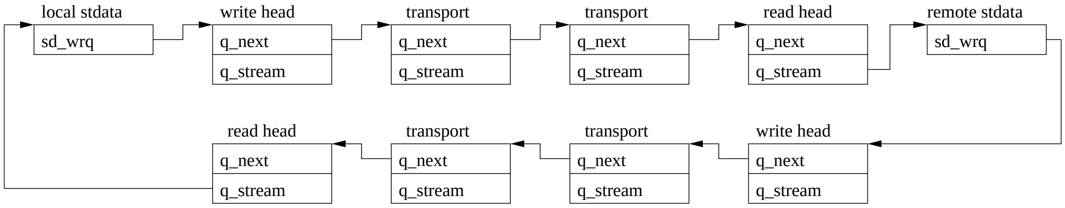

Figure 4.9. Solaris local domain stream layers

Table 4.9 contains the additional functions that were used in the Solaris implementation.

process_req

The `process_req()` implementation for Solaris calls `find_proc()` to identify the UID and PID of a socket, for all request types. Unlike Linux and OpenBSD, it takes the same amount of effort to find the PID as the UID. The details of this process will be given later.

Table 4.9
Solaris data collection functions

Function	Description
find_tcp()	Identifies the vnode address of a requested socket. This address can be used by find_proc() to determine the UID and PID of it.
save_pipe()	Saves pipe structure addresses in fddata structure
save_socket()	Saves appropriate address data for local and Internet domain sockets in a fddata structure.

If the request type is ID, then the process_req() function returns after calling find_proc(). Otherwise, walk_ptree() is called to identify the processes tree and resolve_lcl() is called to resolve forms of IPC.

walk_ptree

The walk_ptree() function saves data associated with the process tree starting at a specified process. The Solaris implementation uses kvm_getproc() to copy the proc structure into user memory. The cred credentials structure is read from kernel memory to determine the UID and GID values. The user structure is read to determine the process name and start time. The list of file structures is read from the user structure and each is processed to identify sockets and pipes. The vnode structure is read from the f_vnode field in file to determine the file type. The contents of the fddata address fields in Solaris are given in Table 4.10.

If the vnode is for a FIFO or pipe, then the fifonode structure is read from the v_data field in the save_pipe() function. The local vnode address is stored as the

Table 4.10
Solaris fddata address values

	fddata	
	lcladdr	**remaddr**
Internet domain socket	IP address	IP address
Local domain socket	vnode address	vnode address
pipe	vnode address	vnode address

local address in the fddata structure. The address of the remote vnode structure, located in the fn_dest field, is saved in the remote address field of fddata.

If the vnode is for a socket, then the sonode structure is read from within the save_socket() function to determine the domain and protocol type. As previously mentioned, the socket address information cannot be trusted so it is not gathered from here. If the socket is in the local domain, the sockaddr_un structures are read to determine the bound file system path. These paths were found to be reliable.

If the socket is in the Internet domain, then the stream head is read from v_stream. We follow the stream for three levels until we read the queue structure for the IP level. The icp_s structure is read and the IP address and ports are saved in the fddata structure.

If the socket is in the local domain, then the stream head is read from v_stream and the local address field in fddata is set to the local vnode address. The remote address field is set to the remote vnode address, which is found by following the stream for four levels. The stream head is read from the fourth level using the q_stream field and the remote vnode is read from the sd_vnode field.

Table 4.11
Solaris find_proc() argument type

	Data type	Contents
Internet domain socket	reqtype	Connection tuples
Local domain socket	uint32_t	vnode address
pipe	uint32_t	vnode address

The walk_ptree() function then repeats this procedure on the parent process until we reach a process we have already analyzed or we reach the top of the process tree.

find_proc

The find_proc() function determines the PID of the process that has a socket or pipe open. As shown in Table 4.10, when a local domain socket or a pipe is searched for, the find_proc() is passed the address of a vnode structure.

When an Internet domain socket is searched for, find_proc() is passed a reqtype structure. It is expensive to find a process based on connection tuples because there are many layers of structures to read, but we can easily identify the vnode address for TCP connections using the IP protocol control block hash table, as described in Section 4.6.1. The find_tcp() function uses the hash table to determine the vnode address of a TCP socket. Therefore, all TCP connections are first sent to find_tcp() to identify the vnode address. All other types of Internet domain sockets (UDP and RAW) must be searched for based on the address tuples.

The find_proc() function cycles through all processes using kvm_setproc() and kvm_nextproc(). The file descriptors of every process are examined and the f_vnode field is compared with the vnode address identified above, or similarly the tuples are compared for other sockets.

When a process has been found with the same vnode address, or connection tuple, the PID is added to the PID array that was passed as an argument. The function returns after finding one process for sockets, but searches all processes for pipes. The UID is copied to the ruid field of the reqtype structure when a TCP socket is found, because this function is also used to identify the UID for ID type requests.

4.6.3 Conclusion

Using Solaris, we were able to identify all needed process information and resolve the needed forms of IPC. It is inconvenient that the sonode structure does not contain the correct addresses. It would be much more efficient if it did. There also exists a lack of documentation about the stream structures and hash tables. The hash table was identified from the *oidentd* source [McC00] and the *lsof* source code was used as a guide for using the stream layers [Abe00].

The Solaris implementation would have been more efficient if there were a backward reference from file descriptors to processes so they would not have to be searched. The direction of sockets should be noted in the sonode structure.

4.7 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 128MB of RAM.

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 4.7.1. 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 4.7.2.

4.7.1 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 minutes to complete.

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

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 process tree shown in Figure 3.4. It is assumed that a `STOP` daemon would process this type of structure the most frequently.

Table 4.12 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 3.1.2, an `ID` type request is equivalent to the traditional `ident` protocol 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 saves the data to a file, while the second does not. As described in Section 3.1.2, 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 `/proc/` files. 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.

Table 4.12
Average lookup time for six unique processes

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

Table 4.13
Average lookup time for 14 unique processes

Platform	SV	SV with 100 procs
Linux	63.354 mS	224.589 mS
OpenBSD	10.256 mS	32.059 mS

Complex Process Structure

The test program was then run on the 14-process structure described in Section 3.5.2 and shown in Figure 3.6. 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 4.13. These results show that the OpenBSD lookup time for the 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. The tests were repeated with the addition of 100 processes that had the three standard file descriptors, two open pipe descriptors and one open file descriptor. 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 non-typical example. As will be shown next, the system impact of processing requests is minimal.

4.7.2 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 minutes to run with no load. The code is given in Appendix C.1. 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 non-sequential memory accesses, operations are performed on random elements in the array. The initial values, such as 0.1029384756, were chosen at random and the random number generator was always seeded with 123456.

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.

Table 4.14
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%

Table 4.14 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 6-process basic process tree with the output printed to a file. Figure 4.10 shows these values in a graph.

This data shows 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 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 into another system after logging into `expert`. Few users do this on a regular basis.

4.7.3 Conclusion

Clearly, resolving processes is an expensive operation, but the complex structure as shown in Section 4.7.1 is not typical. The lookup times with the additional 100

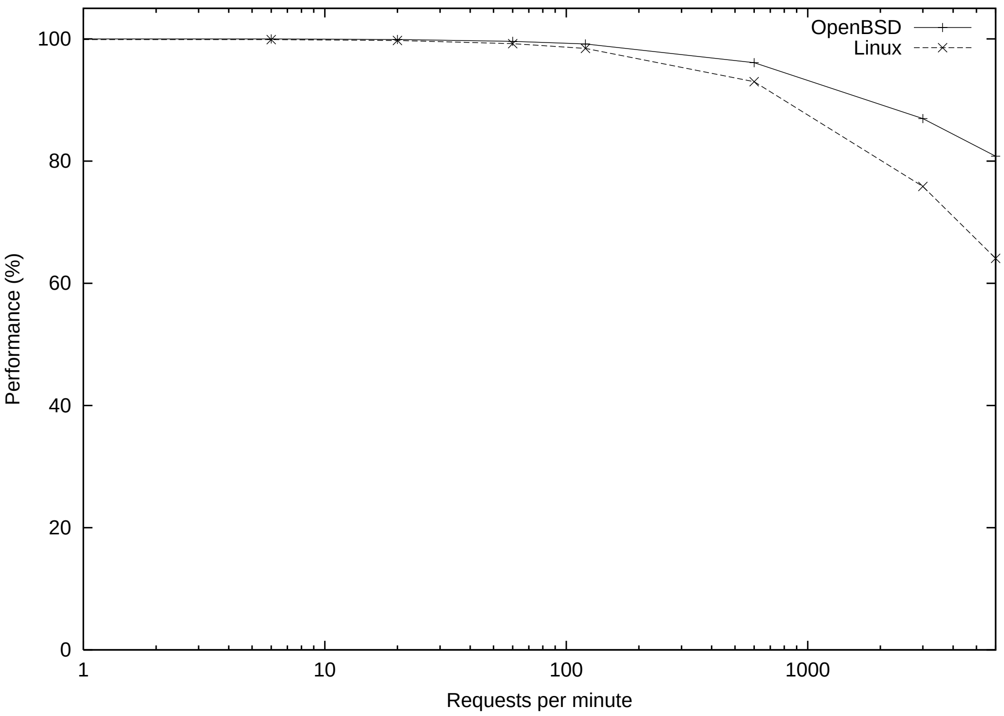

Figure 4.10. Performance impact graph

processes shows that on a multiple user system, significant time could be spent on these lookups and possibly result in a denial of service scenario if enough requests are being serviced. As shown in Section 4.7.2, a system can process 100 requests per second for basic process trees and only see an 80% or 64% 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 shows that by only using a process pseudo file system, the daemon does not scale as well.

5. CONCLUSION

This thesis has provided the framework and details for a protocol to provide data that is commonly missing during forensic investigations. The Session Token Protocol (STOP) can provide a record of socket activity and allows an attacker who is using a series of hosts to be traced. By returning only random tokens, the protocol protects a user's privacy and makes it difficult for other systems to rely on it as a method of authentication.

This protocol is most effective when many hosts are running it and could be used across the Internet, but is best suited in universities and other large networks. Research has found that existing `ident` daemons will correctly parse the new request format and not return an error.

The ability to make requests on behalf of other machines provides border gateways and intrusion detection systems with a method to request data on suspicious inbound and outbound traffic.

This thesis has shown that this protocol can be implemented and is effective in saving data about a network session and tracing connection chains. It can be used in parallel with other traceback techniques such as network traffic analysis to provide application-level data to investigators.

5.1 Recommended Features

This design collects data in a manner that operating systems were not designed to do. This section contains some features that would make the data collection easier.

The most obvious feature is to add process data to the file descriptor structure. This will not add considerable overhead because the kernel already increments the file descriptor's reference count whenever a new process opens the file and decrements it whenever a process closes it. The file descriptor could either save a list of `proc` pointers or a list of PID values. If this list existed, then sockets can be easily identified using the protocol control block hash table and no additional work would be required to identify the processes involved. This would save considerable amount of the searching that this daemon does.

Another feature is to save the socket direction, which can already be found in OpenBSD. The socket direction is used when responding to a request for an inbound socket. Even though this protocol is not vulnerable to the attack described by Goldsmith [Gol96] because it only returns random tokens, processing inbound connections could make the system more susceptible to a denial of service attack. This data is also useful when determining to which hosts to send traceback requests. This feature requires little additional time and space. It can be implemented as a 1-bit flag and set when the `listen()` or `connect()` system calls are called.

The Linux process file system should provide some method of identifying local domain sockets that are connected. Otherwise, we have no way of resolving local domain sockets.

Process entries should be able to be retrieved by UID. OpenBSD already provides this option, but Linux and Solaris do not. In some cases, we knew the UID of the process we were looking for, but had to examine the process structure data to identify it. It is much faster if the UID of a process was examined in kernel memory before it is copied out. This could be achieved in Solaris by adding an option to `kvm_getproc()` and in Linux by using a subdirectory for each user, which contains symbolic links to the processes they are running.

Solaris should update the `sonode` structure with the correct IP addresses at all times. The streams structure should not have to be traversed to get accurate data when the field already exists.

APPENDICES

A. Protocol Interface Specification

A.1 External Interface Data Structures

reqtype		
type	uint8_t	Type of request (`ID`, `ID_REC`, `SV`, `SV_REC`)
lcladdr	uint32_t	IP address of local interface
remaddr	uint32_t	IP address of remote host
lclport	uint16_t	TCP port on local interface
remport	uint16_t	TCP port on remote host
reqaddr	uint32_t	IP address of requesting host
sessid	uint32_t	Random id for recursive requests
ruid	uid_t	Real user id of requested process
pid	pid_t	Process id of requested process

procdata		
*parent	procdata	pointer to parent process
*name	char	process name
pid	pid_t	identifier of process
ppid	pid_t	identifier of parent process
ptype	uint8_t	type of process entry (primary or secondary)
ruid	uid_t	real user id of process
euid	uid_t	effective user id of process
rgid	gid_t	real group id of process
egid	gid_t	effective group id of process
dev	dev_t	terminal device identifier
start	time_t	time that process started
prio	int32_t	priority of process
*fd	fddata	pointer to `fddata` structures for pipes and sockets

fddata		
*next	fddata	Pointer to next `fddata` structure
type	uint8_t	type of file descriptor (ISOCK, USOCK, PIPE)
protocol	uint8_t	socket protocol (STR, GRAM, RAW)
dir	uint_t	socket direction (IN, OUT, UNKNOWN)
lcladdr	uint32_t	local Internet domain IP address or address of local end of pipe structure
remaddr	uint32_t	remote Internet domain IP address or address of remote end of pipe structure
lclport	uint16_t	local Internet domain socket port
remport	uint16_t	remote Internet domain socket port
*lclunix	char	local local domain socket path
*remunix	char	remote local domain socket path

A.2 External Interface Functions

parse_req	
in	char *str reqtype *req
out	int
desc	`parse_req()` takes a NULL-terminated string (`str`) as an argument and parses it according to the STOP protocol grammar. The fields of `req` are filled based on the values in `str`. Returns 0 on success and 1 on error.

process_req	
in	reqtype *req
out	procdata *
desc	`process_req()` takes a `reqtype` structure and processes the request based on the contents. If the request type is `ID`, then it returns NULL and the `ruid` field of `req` is filled in or the `ruid` field is `-1` on error. If the request is non-`ID`, then it returns a linked list of `procdata` structures for the socket in the request. All IPC forms will have been resolved. NULL is returned on error.

print_procdata	
in	procdata *pdptr FILE *hLog reqtype *req
out	int
desc	`print_procdata()` prints the contents of the `pdptr` linked list to `hLog`. The contents of `req` are also printed to `hLog`. Returns 0 on success and 1 on error.

send_reqs	
in	procdata *pdptr FILE *hLog reqtype *req
out	int
desc	`send_reqs()` searches the `pdptr` list for inbound Internet domain socket connections from a remote host. It sends a request of the same type as specified in `req` to the host, using the session identifier specified in `req`. The tokens that are returned from the requests are printed to `hLog`. Returns 0 on success and 1 on error.

A.3 Internal Interface Data Structures

lclcomm		
*next	lclcomm	Pointer to next structure in linked list
type	uint8_t	Type of connection (same values as `fddata`)
p1	uint16_t	Primary field to sort on (port number or memory address)
p2	uint16_t	Secondary field to sort on (port number or memory address)
rev	uint8_t	Set to 1 if `p1` and `p2` were reversed because `p2` > `p1`
addr	uint32_t	IP address of Internet domain socket

A.4 Internal Interface Functions

walk_ptree	
in	pid_t pid struct procdata *pdbase int *stat
out	struct procdata *
desc	`walk_ptree()` takes a PID (`pid`) as an argument and saves process state data about it. It saves data into a `procdata` linked list for `pid` and its parent processes until it reaches the top of the process tree or until it reaches a process that already exists in `pdbase`. It returns the linked list of `procdata` structures of processes that were analyzed. Upon error, it sets `stat` to 1 and returns NULL. When it has seen all of the processes it returns NULL and sets `stat` to 0.

find_proc	
in	void *goal char type pid_t *pids int size
out	int
desc	The `find_proc()` function finds processes that have a certain file descriptor trait. The type of file descriptor to find is specified by `type` and the contents of `goal` are platform and `type` specific. Processes that are found to have this trait are placed in `pids`, which has size `size`. The number of processes found is returned, or -1 on error.

resolve_lcl	
in	struct procdata *pdhead
out	struct procdata *
desc	`resolve_lcl()` takes a list of `procdata` structures (`pdhead`) and searches them for methods of IPC that need to be resolved. It resolves them using `find_proc()` and adds the new processes to the tail of `pdhead`. The full `pdhead` list is returned or NULL on error.

B. Daemon Sample Outputs

B.1 Simple Process Structure Report

```
- BEGIN -
HOST INFORMATION
Name: host-2.cerias.purdue.edu
Boot: Fri Feb 02 00:56:23 2001
OS: OpenBSD  2.8  rev 200012
OpenBSD 2.8 (HOST-2) #1: Thu Jan 11 22:52:49 EST 2001

REQUEST INFORMATION
Date: Sun Apr  1 12:01:00 2001
Remote: 3.3.3.3:23
Local: 2.2.2.2:968
Requester: 3.3.3.3
Type: SV

PROCESS INFORMATION
Primary Processes
1: telnet [8339]
   priority: 0   parent: 8338
   ruid: user1 (1000) euid: user1 (1000)
   device: ttyp1
   Sockets:
     INET_TCP: 2.2.2.2:968 -> 3.3.3.3:23

2: csh [8339]
   priority: 0   parent: 8337
   ruid: user1 (1000) euid: user1 (1000)

3: sshd [8337]
   priority: 0   parent: 8000
   ruid: root (0) euid: root (0)
   Sockets:
     INET_TCP: 2.2.2.2:22 <- 1.1.1.1:616
     INET_TCP: localhost:6012 <- any

4: sshd [8000]
   priority: 0   parent: 1
   ruid: root (0) euid: root (0)
   Sockets:
     INET_TCP: localhost:22 <- any
```

```
5: init [1]
   priority: 0   parent: 0
   ruid: root (0) euid: root (0)

6: swapper [0]
   priority: 0   parent: 0
   ruid: root (0) euid: root (0)

ADDITIONAL USER INFORMATION
user1 (1000)
  ttyp1  host-1.cerias.pu  Sun Apr  1 12:00:00 2001
- END -
```

B.2 Complex Process Structure Report

```
- BEGIN -
HOST INFORMATION
Name: host-2.cerias.purdue.edu
Boot: Fri Feb 02 00:56:23 2001
OS: OpenBSD  2.8  rev 200012
OpenBSD 2.8 (HOST-2) #1: Thu Jan 11 22:52:49 EST 2001

REQUEST INFORMATION
Date: Sun Apr 1 01:01:00 2001
Remote: 3.3.3.3:9010
Local: 2.2.2.2:8526
Requester: 3.3.3.3
Type: SV

PROCESS INFORMATION
Primary Processes
1: resolve [26776]
   priority: 0   parent: 22614
   ruid: root (0) euid: root (0)
   Sockets:
     INET_TCP: 2.2.2.2:8526 -> 3.3.3.3:9010
   Pipes:
     E07F2600 -> E08CA780

2: resolve [22614]
   priority: 0   parent: 1
   ruid: root (0) euid: root (0)

3: init [1]
   priority: 0   parent: 0
   ruid: root (0) euid: root (0)
```

```
4: swapper [0]
   priority: 0   parent: 0
   ruid: root (0) euid: root (0)

Resolved Processes
5: resolve [25697]
   priority: 0   parent: 7226
   ruid: root (0) euid: root (0)
   Pipes:
     E08CA780 -> E07F2600
     E08CAE80 -> E0801880

6: resolve [7226]
   priority: 0   parent: 23690
   ruid: root (0) euid: root (0)
   Pipes:
     E0801C00 -> E0801400

7: resolve [23690]
   priority: 0   parent: 26776
   ruid: root (0) euid: root (0)

8: resolve [10236]
   priority: 0   parent: 7226
   ruid: root (0) euid: root (0)
   Pipes:
     E0801880 -> E08CAE80

9: resolve [324]
   priority: 0   parent: 10861
   ruid: root (0) euid: root (0)
   Pipes:
     E0801400 -> E0801C00

10: resolve [10861]
   priority: 0   parent: 26776
   ruid: root (0) euid: root (0)

11: resolve [12596]
   priority: 0   parent: 18587
   ruid: root (0) euid: root (0)
   Pipes:
     E08CA780 -> E07F2600
     E08CA380 -> E07E8080
```

```
12: resolve [18587]
   priority: 0   parent: 22614
   ruid: root (0) euid: root (0)
   Sockets:
     INET_TCP: localhost:8012 <- any
     INET_TCP: 127.0.0.1:8012 <- 127.0.0.1:32145
   Pipes:
     E07E8080 -> E08CA380

13: resolve [11995]
   priority: 0   parent: 1
   ruid: root (0) euid: root (0)
   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 [26780]
   priority: 0   parent: 1
   ruid: root (0) euid: root (0)
   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

ADDITIONAL USER INFORMATION
root (0)
- END -
```

B.3 Reverse Telnet Report

```
- BEGIN -
HOST INFORMATION
Name: host-3.cerias.purdue.edu
Boot: Fri Feb  9 23:16:22 2001
Linux version 2.2.17 (herbert@arnor) (gcc version 2.95.2 20000313
  (Debian GNU/Linux)) #1 Sun Jun 25 09:24:41 EST 2000

REQUEST INFORMATION
Date: Sun Apr 1 02:01:00 2001
Remote: 1.1.1.1:8000
Local: 2.2.2.2:1885
Requester: 3.3.3.3
Type: SV_REC

PROCESS INFORMATION
```

```
Primary Processes
1: /usr/bin/telnet [10212]
   priority: 0   parent: 9818
   start time: Sun Apr 1 02:00:00 2001
   ruid: nobody (65534) euid: nobody (65534)
   Sockets:
     INET_TCP: 2.2.2.2:1885 <> 1.1.1.1:8000
   Pipes:
     0 -> 38AB64

2: server [9818]
   priority: 0   parent: 1
   start time: Fri Feb 9 23:18:00 2001
   ruid: nobody (65534) euid: nobody (65534)
   Sockets:
     INET_TCP: localhost:80 <> any

3: /sbin/init [1]
   priority: 0   parent: 0
   start time: Fri Feb  9 23:16:22 2001
   ruid: root (0) euid: root (0)

4: sched [0]
   priority: 0   parent: N/A
   start time: Fri Feb  9 23:16:22 2001
   ruid: root (0) euid: root (0)

Resolved Processes
5: /bin/sh [10213]
   priority: 0   parent: 9818
   start time: Sun Apr 1 02:00:00 2001
   ruid: nobody (65534) euid: nobody (65534)
   Pipes:
     0 -> 38AB64
     0 -> 38AB65

6: /usr/bin/telnet [11715]
   priority: 0   parent: 6818
   start time: Sun Apr 1 02:00:00 2001
   ruid: nobody (65534) euid: nobody (65534)
   Sockets:
     INET_TCP: 1.1.1.1:1886 <> 2.2.2.2:8001
   Pipes:
     0 -> 38AB65

ADDITIONAL USER INFORMATION
```

```
nobody (65534)
- END -
```

C. Benchmark Code

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

#define SIEVEL 1000000

#define ROUNDS 10
#define LOOPS 1000 /* ~100 per minute */

int main() {
    float a, b, c;
    int i, j, l;
    float sieve[SIEVEL];
    struct timeval st, fin;
    time_t tim;
    unsigned long  sum = 0;
    signed long usum = 0;
    double tmp;

    /* make sure every run is the same */
    srandom (123456);

    a = 0.1029384756;
    b = 9753124680.1470258369;
    c = 1627.384950;

    printf("Rounds: %d     Loops per Round: %d\n", ROUNDS, LOOPS);
    tim = time(NULL);

    for (l=0; l<ROUNDS; l++) {
        gettimeofday(&st, NULL);

        for (i=0; i<LOOPS; i++) {

            a *= b;
            a /= c;
            b = a - b;
            a *= 22.123456789123456789;
            for (j=0; j<SIEVEL; j++) {
                sieve[j] =
                  b * (float)j / 710.9638642 + sieve[(j+2121) % SIEVEL];
            }
            for (j=0; j<20000; j++) {
                sieve[random() % SIEVEL] *=
                  (sieve[random() % SIEVEL] / 88442211.99);
            }
        }

        gettimeofday(&fin, NULL);

        sum += (fin.tv_sec - st.tv_sec) ;
        usum += (fin.tv_usec - st.tv_usec) ;

    }

    /* total time */
    tmp = ((double)sum  + (double) usum / (double) 1000000);
    printf("Total seconds: %f\n", tmp);

    /* time per round */
    tmp /= (double)ROUNDS;
    printf("Ave sec/round %f\n", tmp);

    /* time per loop */
    tmp /= (double)LOOPS;
    printf("Ave sec/lookup %f\n", tmp);

    return 0;
}
```

LIST OF REFERENCES

LIST OF REFERENCES

[Abe00] Vic Abell. lsof v4.52. available at: http://vic.cc.purdue.edu, November 8, 2000.

[BDKS00] Florian Buchholz, Thomas Daniels, Benjamin Kuperman, and Clay Shields. Packet tracker final report. Technical Report 2000-23, CERIAS, Purdue University, 2000.

[Bel00] Steven Bellovin. ICMP traceback messages. Technical Report draft-bellovin-itrace-00.txt, IETF Internet draft, March 2000.

[BS01] Florian Buchholz and Clay Shields. Providing process origin information to aid in network traceback. Technical Report 2001, CERIAS, Purdue University, April 2001.

[BY97] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[Eri00] Peter Eriksson. pidentd ident daemon v3.0.12. available at: http://www2.lysator.liu.se/p̃en/pidentd/, December 3, 2000.

[Gol96] Dave Goldsmith. ident-scan. Email post to bugtraq mailing list. Available at: http://lists.insecure.org/bugtraq/1996/Feb/0024.html, February 13, 1996.

[Hob96] Hobbit. netcat v1.10. available at: http://www.l0pht.com/ weld/netcat/, March 20, 1996.

[JKS$^+$93] Hyun Tae Jung, Hae Lyong Kim, Yang Min Seo, Ghun Choe, Sang Lyul Min, Chong Sang Kim, and Kern Koh. Caller identification system in the Internet environment. In *Proceedings USENIX UNIX Security Symposium IV*, 1993.

[Joh85] M. St. Johns. Authentication server. RFC 931, TPSC, January 1985.

[Joh93] M. St. Johns. Identification protocol. RFC 1413, US Department of Defense, February 1993.

[lin00] Debian Linux operating system v2.2. available at: www.debian.org, August 14, 2000.

[MBKQ96] Marshall McKusick, Keith Bostic, Michael Karels, and John Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996.

[McC00] Ryan McCabe. oidentd ident daemon v1.7.1. available at: http://ojnk.sourceforge.net/, October 22, 2000.

[MM01] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture.* Sun Microsystems Press, 2001.

[Mor98] R. Morgan. S/ident: Security extensions for the ident protocol. draft-morgan-ident-ext-04.txt, Stanford University, March 1998.

[Ope00] OpenBSD operating system v2.8. available at: www.openbsd.org, December 1, 2000.

[PL01] K. Park and W. Lee. On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack. In *Proceedings of the IEEE INFOCOM01*, Anchorage, Alaska, 2001.

[Pos81] J. Postel. Internet protocol. RFC 791, ISI, September 1981.

[RP92] J. Reynolds and J. Postel. Assigned numbers. RFC 1340, ISI, July 1992.

[SCH95] Stuart Staniford-Chen and L. Todd Heberlein. Holding intruders accountable on the Internet. In *Proceedings IEEE Symposium on Security and Privacy*, 1995.

[sen] Sendmail. available at: www.sendmail.org.

[SHA93] Secure hash standard. National Institute of Standards and Technology, FIPS PUB 180, May 1993.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.

[SMK01] Joel Scambray, Stuart McClure, and George Kurtz. *Hacking Exposed*, pages 319 – 321. Osborne: McGraw Hill, 2 edition, 2001.

[sol] Sun Solaris operating system v2.7. available at: www.sun.com.

[SP01] D. Song and A. Perrig. Advanced and authenticated marking schemes for IP traceback. In *Proceedings of the IEEE INFOCOM01*, Anchorage, Alaska, April 2001.

[Ste93] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1993.

[SWKA00] Stefan Savage, David Wetherall, Anna R. Karlin, and Tom Anderson. Practical network support for ip traceback. In *SIGCOMM*, pages 295–306, 2000.

[Vah96] Uresh Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, 1996.

[Ven92] Wietse Venema. TCP wrapper: Network monitoring, access control, and booby traps. In *Proceedings USENIX UNIX Security Symposium III*, 1992.

[YE00] Kunikazu Yoda and Hiroaki Etoh. Finding a connection chain for tracing intruders. In *Proceedings 6th ESORICS*, 2000.

[YKS$^+$93] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. Ssh protocol architecture. draft-ietf-secsh-architecture-04, Network Working Group, June 1993.

[ZP00] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proceedings 10th USENIX Security Symposium*, 2000.