A Recursive Session Token Protocol For Use in Computer Forensics and TCP Traceback

Brian Carrier
Center for Education and Research in
Information Assurance and Security (CERIAS)
Purdue University
West Lafayette, IN 47907
carrier@cerias.purdue.edu

Clay Shields
Department of Computer Science
Georgetown University
Washington, D.C., 20007
clay@cs.georgetown.edu

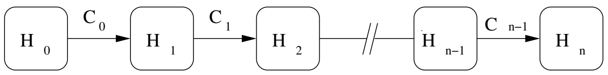

Fig. 1. Connection chain example between H_0 and H_n

***Abstract*—**
We introduce a new protocol designed to assist in the forensic investigation of malicious network-based activity, specifically addressing the stepping-stone scenario in which an attacker uses a chain of connections through many hosts to hide his or her identity. Our protocol, the *Session TOken Protocol* (STOP), enhances the Identification Protocol (`ident`) infrastructure by sending recursive requests to previous hosts on the connection chain. The protocol has been designed to protect user's privacy by returning a token that is a hash of connection information; a system administrator can later decide whether to release the information relating to the token depending on the circumstances of the request.

I. INTRODUCTION

To hide the network address of a host used for initiation of a network attack, an attacker will often log into a series of compromised hosts before attacking his or her intended target. This technique is commonly called stone stepping [23], and is used to allow the attacker to avoid responsibility for his or her actions. As shown in Figure 1, let H_i, $0 \leq i \leq n$, be a set of network 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 $0 \leq i < n$.

The current method of determining the source of an attack is to contact the previous host in the chain and to ask the administrator to investigate his or her system. If a prior hop from that host is discovered, then the administrator of that system must then be contacted, and so on. In practice this type of traceback is often hindered by the fact that many administrators lack the resources, knowledge, trust, or system data to continue the investigation. Particularly, in some instances, there is inadequate logging on a system to determine the previous host of a connection.

This work was conducted within and supported by the Center for Education and Research in Information Assurance and Security (CERIAS).

We present the *Session TOken Protocol* (STOP) [5], which is based on the `ident` protocol, and helps forensic investigation of stepping-stone chains while protecting the privacy of users. STOP saves application-level data about the process and user that opened the socket, and can also send requests to previous hosts to identify other hosts in the chain. At each stage, a hashed token is returned; at no point in the protocol does the requester ever directly learn user or process data. Instead, they must redeem the token to the system administrator who can determine the merit of releasing user information. Random session IDs in the requests allow cycle detection. Additionally, to allow for centralized control of security, STOP gives border gateways and firewalls the ability to send requests for inbound or outbound connections.

In the following section we describe previous work in the area of attack traceback, and outline the operation of the `ident` protocol. Section III describes the STOP protocol, and implementation results appear in Section IV. Finally, we present comparative performance results in Section V, and follow with our conclusions in Section VI.

II. PREVIOUS WORK

While there has been significant recent interest in determining the source of network attacks, the bulk of this work has been targeted at a separate problem — that of locating the source of spoofed IP packets [16], [4], [15], [17], [14], [6], [7], [2], [19]. While providing a solution to this problem is crucial in providing a response to distributed denial-of-service attacks [22], this work does not allow an attacker to be held accountable for their actions, as attackers rarely send spoofed packets directly from their host. Instead, they connect through a series of compromised hosts, and launch their attack from the tail of that chain.

A. Connection Chain Traceback

Previous work in determining the source of network traffic can be divided into two areas; network-based solutions, and host-based solutions.

The first work in connection chain traceback was a network-based solution proposed by Staniford-Chen and Heberlein [18]. They suggested saving content-based $thumbprints$ of active connections at network gateways. When an attack is detected, the thumbprints can be analyzed and pairs of connections statistically correlated. This analysis is based solely on packet content and therefore does not work when encryption or compression is used. Additionally, in the United States, it is often illegal for ISPs or officers of the government to examine the contents of packets [12].

Recent work by Zhang and Paxson avoided the problems of using packet content by relying instead on timing analysis of user idle times in interactive connections [23]. Their design places a network analyzer the border of network domains that performs real-time analysis to determine if connections entering and leaving the domain are part of the same stepping-stone chain. The times when TCP connections go from an idle state to a non-idle state between inbound and outbound connections are correlated to identify inbound and outbound pairs. As this does not rely on content, it is not affected by use of cryptography or compression. This work was intended to perform an intrusion detection function, so that compromised hosts being used as stepping stones could be easily identified, but the authors have suggested that this technique could be used for traceback if the appropriate analysis were done at each domain on the connection chain.

Yoda and Etoh also avoided using content for analysis by using TCP sequence numbers [21]. TCP sequence numbers for a given connection always increase at a rate proportional to the amount of data transmitted. Therefore, if two connections are in the same chain, then their sequence numbers should increase at the same rate. Their work identifies connection chains by graphing the sequence numbers versus time and calculating the differences between the graphs. If two graphs have little difference, then they are likely to be from the same chain. This design will not work if compression of the data stream occurs differently on each part of the chain, as may be the case in such protocols as SSH [20], because the sequence numbering will be different, nor will it work if link encryption is used as TCP headers will be encrypted.

The sole prior host-based solution that has been proposed was the Caller Identification System (Caller ID) [11]. The Caller ID system is a set of protocols designed to authenticate a user and the path of hosts he or she is logged into. Its primary purpose is for authentication, but the data it gathers can also be used to trace an attacker. When a user logs into host H_i from host H_{i-1}, H_i sends a request to H_{i-1} with the TCP port pair. H_{i-1} identifies the user that has the TCP connection and returns a cached list of previous host and user name pairs to H_i. H_i verifies the list by sending a request to each host with the corresponding user name. The host returns `yes` if it has a process running as that user and returns `no` otherwise. If any host returns `no` to H_i, then the user is not authorized to log in. Host H_i saves the list in case it receives a request from H_{i+1}.

This work does not address the problem of mapping an outgoing TCP connection with the previous incoming connections. Instead, it simply verifies that a process with the appropriate user name is running on the system. Additionally, as the host saves the list of previous hosts for a user, it will not know which host list to return when the user has several active login sessions [3]. This protocol may also add considerable delay while the list of previous hosts is verified.

B. The Identification Protocol

The Identification Protocol (`ident`) [10] is a simple protocol that is designed to allow a server to identify the client-side user name. While the full grammar for the protocol is available in RFC 1413 [10], the protocol works as follows:

1. A client and server establish a TCP connection between TCP port `<CL_PORT>` on the client and TCP port `<SV_PORT>` on the server.

2. The server establishes a connection to port 113/TCP on the client and sends the following message:

`<CL_PORT>,<SV_PORT>`

3. The client 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 and corresponding UID is found, a message such as is returned:

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

or in case of error:

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

As `ident` returns user names, it can be invasive on user privacy and can also be used for other purposes besides security. Goldsmith showed that the `ident` RFC did not specify that the daemon need only return the identity of connections that originated on that host [9]. Therefore, an attacker could establish a connection with any service, send an `ident` request, and receive a reply with the user name of the service running. This allows an attacker to determine which services are running as $root$. Additionally, web servers can collect email addresses of users that visit their sites and add them to bulk email lists for SPAM.

Several implementations take additional steps to protect user privacy. The OpenBSD daemon [1] returns a string of 80 random bits in hex instead of the user id. The random token can be redeemed for the user name after proper identification and need have been presented to an administrator. Similarly, the $pidentd$ daemon [8] can return the user name encrypted using DES, which can later be decrypted by an administrator. Other measures include always returning `"OTHER"` as the operating system type and returning `"UNKNOWN-ERROR"` for all errors.

III. A Recursive Session Token Protocol

STOP provides additional functionality to what is offered from `ident`. It can be run on any host with no modification of protocols, network topology, or kernel. It can also be run in parallel with any of the network-based connection chain analysis tools previously proposed.

A. Protocol Design Goals

The design of STOP followed a number of design goals, enumerated here:

1) Must be backward compatible with the Identification Protocol as specified in RFC1413 [10] because of its widespread usage and implementation.
2) Must not release any user or system data until proper credentials have been provided.

3) Must provide a mechanism for a server to request that the client save additional data, in addition to just the user name.
4) Must provide a mechanism for a server to request that the client trace the user's path of previous hosts.
5) Must be configurable such that it can comply with the system security and privacy polices.
6) Should be efficient and not add considerable load to the daemon host or delay to the requester.
7) Should allow a host that is not on the connection chain to make requests.

The standard `ident` protocol satisfies goals 1 and 6. Some implementations satisfy goals 2 and 5 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 7.

B. Protocol Design

The new protocol builds upon the `ident` protocol by modifying the request message to provide more options and the response message to protect privacy. The request message now contains a request type, which can be one of the following:

- **ID**: The daemon saves the user name in its log file and returns a random token. This is the same behavior as the original `ident` protocol.
- **ID_REC**: The daemon saves the user name and return a random token. The daemon then sends an `ID_REC` request to the host that the user logged in from. This option also requires a random session identifier to identify cycles in the recursion.
- **SV**: The daemon saves the user name and other data associated with the process that opened the port.
- **SV_REC**: The daemon saves the same data as with `SV` and also has the recursive property as described with `ID_REC`. This type also requires a session identifier.

The request may optionally contain an IP address in dotted decimal format that when given is used as the remote address of the TCP connection. This is intended for use by network gateways and Intrusion Detection Systems (IDS). When the IP address is specified, no error messages or user names must be returned, only normal responses with a random token. This prevents information gathering by attackers posing as legitimate requesters.

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 goal 2 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 a user to specify that his or her user name not be sent to other systems. Our protocol will only return random tokens and therefore does not need this error type, as the user's privacy is still preserved. The last change is that only printable ASCII is allowed in the random token. The original protocol allowed the return token to be any octet value except NULL, CR, and LF. As the protocol is returning random tokens that will be later redeemed for actual data, it will be easier if tokens are generated using only printable ASCII. The full grammar is shown in Figure 2.

Due to the second design goal, STOP by default will return a random token. In some implementations, the user may opt-in to having 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 random token for all established connections
- Random tokens need not be cryptographically secure, but must not contain any 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.
- Return a random 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 process data when `SV` or `SV_REC` requests are received (see Section III-C).
- 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 III-D).
- Save tokens from recursive requests with the original reply token. The recursive 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 at least 120 seconds. If a second request is received within 120 seconds of the first, return a random token and log the loop detection.

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

- Provide an option to return a `<user-reply>` message with a random token instead of error messages.
- Provide an user-based option to return the actual user name instead of a random token for an `ID` type request. All other request types must return a random token.
- Provide options for what process and system data to save on behalf of `SV` and `SV_REC` requests to satisfy policies or resources such as disk space.

C. Saving Process State

By sending a `SV` or `SV_REC` request to the client, the server can later gather additional user- and application-level data on the process that made the connection. Upon receiving this request, the daemon will save the additional data to a file in a directory, such as $/var/tokens$. Process data that we believe is important to save include:

- Process name and identifier (PID)
- Parent PID
- Real user id and effective user id
- Process start time and priority

```
<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> ::= CR LF
<reply> ::= <port-pair> ":" <reply-text> <EOL>
<reply-text> ::= <ident-reply> | <error-reply>
<ident-reply> ::= "USERID" ":" "OTHER" ["," <charset>]
        ":" <user-token>
<error-reply> ::= "ERROR" ":" <error-type>
<error-type> ::= "INVALID-PORT" | "UNKNOWN-ERROR" |
        "NO-USER" | <error-token>
<charset> ::= "US-ASCII" | as defined in RFC 1340
<user-token> ::= 1*512<token-characters>
<error-token> ::= "X"1*63<token-characters>
<byte> ::= integer values 0 to 2^8 in ASCII
<int> ::= integer values 0 to 2^32 in ASCII
<token-characters> ::= All printable ASCII except ":"
```

Fig. 2. STOP Protocol Grammar

- Terminal device
- List of open sockets, files, and pipes

In addition, the following data should also be saved to help an investigation should one occur.

- Host name
- Boot time
- Operating System, version, kernel date and build
- Address of machine that sent request
- Address and port of remote end of socket
- Address and port of local end of socket
- Type (i.e. SV_REC) and time of request
- Entries from *utmp* for all users mentioned in report

D. Recursion

The ID_REC and SV_REC request types allow tokens to be generated along an entire path of hosts. The original response 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 random 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.

The recursive 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 the past 120 seconds. If it receives a duplicate request for a port pair with the same random identifier and from the same host within 120 seconds, it must not process the request and return a <user-reply> type message to stop the cycle. This method will prevent a denial of service situation caused by processing the same request in a cycle, but it will not prevent one from a flood of requests using different session identifiers. To prevent this situation, the number of requests that are processed at a time must be limited by the daemon.

E. Security Analysis

STOP does not solve all tracing connection chain tracing problems, as the daemon can be killed on any system 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 does not save any data. This is the same as if it were not running and the path will be known to H_i.
- The daemon does not send recursive requests, which will cause the path to also end at H_i if it is not saving the previous host data or at H_{i-1} if it is saving the previous host data.
- The daemon saves false user and recursive data. For example, the daemon picks another login session at random, and claims that it is the attacker's session. This scenario can lead an investigator far from the true path, but the compromised host would be looked at by following the path of tokens and investigated for malicious activity.

Each of 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 stone scenario and must be used only as a tool whose data must be verified.

IV. IMPLEMENTATION

A prototype of the STOP protocol was implemented by modifying an open source ident daemon, oidentd [13]. The STOP daemon allowed 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 processing the daemon does.

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

The prototype was built on Solaris 2.7, OpenBSD 2.8 and Debian Linux 2.2. The process state data was determined in OpenBSD and Solaris using the Kernel VM library functions and in Linux used the *procfs* pseudo-file system. As will be shown in Section V, the OpenBSD and Solaris implementations have better performance because they read directly from kernel memory.

When an ID type request is received, the daemon acts like a basic ident daemon and determines the UID from kernel memory, or from the /proc/net/tcp file in Linux. When an ID_REC, SV or SV_REC type request is received, the daemon identifies the process that has the socket and saves state data about it. It then saves the same data about the parent process and 'walks' up the process tree by repeating this procedure until the process with PID 0 is reached. Recursive request messages are sent to the remote end of each incoming socket identified during the 'walk'.

Performing a simple 'walk' up the process tree may not be adequate when tracing malicious users. For example, let an attacker run the following command to 'pass through' host H_i:

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

This command uses *netcat* 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 process that connects to H_{i+1} is analyzed no other sockets are encountered. Therefore, if H_{i+1} sent a request of type SV_REC then no recursive requests will be sent. By resolving the pipe and identifying the process was at the other end, H_{i-1} can be determined.

In our implementation, all pipes, local domain sockets (also called UNIX domain sockets), and Internet domain sockets connected to the local host are resolved. This is done by searching the file descriptors of all processes. The identified processes are then 'walked' up and their sockets and pipes are resolved. This continues until all sockets and pipes are resolved.

If the request type is SV or SV_REC, then the state data is stored in a file and the SHA-1 hash of the data is computed and sent to the requester as the random token. The SHA-1 hash is sent to detect any tampering the attacker may do to the data file. For a typical process tree with 6 processes, the output file is roughly 1600 bytes when all variables mentioned in Section III-C are saved. If the tokens are saved to a small disk, an attacker could cause the drive to fill with token files to prevent the system from saving tokens involved with an attack. The data files can also be compressed to roughly 700 bytes.

If a request has type ID_REC or SV_REC, then the saved process data is analyzed for open Internet domain stream sockets. We tried to limit requests for inbound sockets only, but this was unsuccessful because only OpenBSD socket structures save data about direction. When the direction is unknown, 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.

Experiments were performed to verify that the protocol would trace a connection chain, save appropriate data, and detect cycles. The implementations passed all experiments by identifying all hosts in a connection chain of four hosts, saving the correct process and IPC data, and not sending requests when a cycle was identified.

V. PERFORMANCE

The Linux and OpenBSD systems that were used to implement this program have identical hardware and were tested for performance results. The systems had a 600 MHz Intel Pentium III processor and 128MB of RAM. The results of two performance tests are given here, request processing times and system performance.

A. Request Processing Time

The first test program simulated a daemon that would implement the STOP protocol. It forked a new child process, waited for it to finish, and repeated for a specified number of times. The child process parsed a request string and processed it. The total time was divided by the number of lookups to calculate the average processing time.

The program was first run on a simple process tree, shown in Figure 3, that contained 6 unique processes and contained no forms of IPC. Table I shows the average number of seconds per request from our tests. The first data column shows the processing time for an ID type request. As described in Section III-B, 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 request. This shows 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 a sysctl() system call.

The second data column contains the times for performing a SV type request and not saving the process data to a file while the third column contains the times for a SV type request and saving the data to a file. As described in Section III-B, 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 much 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

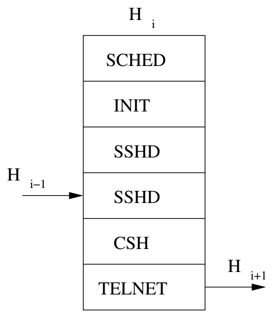

Fig. 3. Process tree with 6 unique processes

TABLE I
AVERAGE LOOKUP TIME FOR 6 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

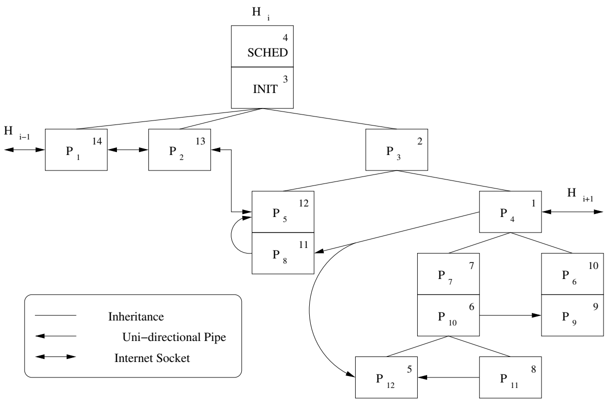

Fig. 4. Process tree with 14 unique processes

TABLE II
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

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, as the write output is the slowest operation.

To show that the daemon can handle non-simple process structures, a more complex test environment was used. This structure can be found in Figure 4 and resolves to 14 unique processes, 3 process groups, and contains 6 instances of IPC to resolve using pipes and Internet domain sockets. The structure contains a one-way communication path from P_4 to P_1. It was resolved starting with the outbound socket on process P_4. An example resolution ordering is shown in the upper right of each process box.

The testing program performed lookups on the socket from

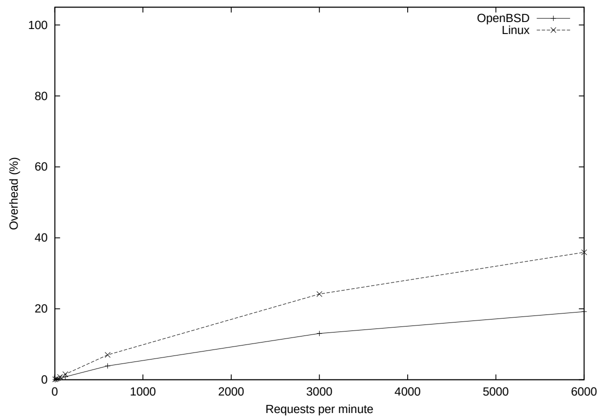

Fig. 5. STOP Overhead

P_4 on a system with no other users and the results can be found in data column one of Table II. These results show that the OpenBSD lookup time for the 14-processes structure is 324% longer than for the 6-process structure. Linux had a 1008% increase over the 6-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. As the daemon has to search through all file descriptors to resolve pipes, each lookup has to examine 600 additional file descriptors for each of the 6 resolutions. 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.

Clearly, resolving processes is an expensive operation, but the complex structure as shown is not typical. As will be shown in the next section, even the complex structure does not greatly impact the system.

B. System Performance

A memory-intensive benchmark program was used to determine the impact that this daemon had on a system. The benchmark program was executed and timed without the daemon running to determine the base time. The benchmark program was then executed and timed again, but with the daemon processing a specified number of requests per minute. The difference from the base time was calculated as a percentage and can be found in Table III for seven different request rates. Each lookup was a `SV` type request on a 6-process basic process tree with the output printed to a file. Figure 5 shows a graph of these values.

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 accounts on it. Over a seven hour period, there were 2499 logins, or almost six per minute. If this value is used as an upper bound for the number of requests a host may receive a minute, the daemon impact is negligible. This upper bound is the extreme case that every user logs into

TABLE III

System performance data

Platform	requests per minute						
	6	20	60	120	600	3000	6000
Linux	0.12%	0.26%	0.79%	1.55%	7.01%	24.15%	35.92%
OpenBSD	0.01%	0.10%	0.39%	0.83%	3.89%	13.04%	19.20%

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

VI. Conclusion

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

This protocol is most effective when many hosts are running it. Because of that, though this protocol could be used for tracing TCP chains across the Internet, we do not expect it to be used there. Instead, it is more useful in a more tightly constrained environment in which there are enforceable policies that require the STOP daemon to be run. This could be a single network or an intranet, as the ability to make requests on behalf of other machines provides border gateways and intrusion detection systems with a method to request data on suspicious inbound and outbound traffic.

We have 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.

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

References

[1] OpenBSD operating system v2.8. available at: www.openbsd.org.

[2] S. Bellovin. ICMP Traceback Messages. Technical Report draft-bellovin-itrace-00.txt, IETF Internet draft, March 2000.

[3] F. Buchholz, T. Daniels, B. Kuperman, and C. Shields. Packet Tracker Final Report. Technical Report 2000-23, CERIAS, Purdue University, 2000.

[4] H. Burch and B. Cheswick. Tracing Anonymous Packets to their Approximate Source. In *Proceedings of the 14th Conference on Systems Administration (LISA-2000)*, New Orleans, LA, December 2000.

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

[6] D. Dean, M. Franklin, and A. Stubblefield. An Algebraic Approach to IP traceback. In *Proceedings of the 2001 Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[7] T. W. Doeppner, P. N. Klein, and A. Koyfman. Using Router Stamping to Identify the Source of IP Packets. In *7th ACM Conference on Computer and Communications Security*, pages 184–189, Athens, Greece, November 2000.

[8] P. Eriksson. pidentd ident daemon v3.0.12. available at: http://www2.lysator.liu.se/p̃en/pidentd/, Dec, 3 2000.

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

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

[11] H. T. Jung, H. L. Kim, Y. M. Seo, G. Choe, S. L. Min, C. S. Kim, and K. Koh. Caller Identification System in the Internet Environment. In *UNIX Security Symposium IV Proceedings*, pages 69–78, 1993.

[12] S. C. Lee and C. Shields. Tracing the Source of Network Attack: A Technical, Legal, and Societal Problem. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY, June 2001.

[13] R. McCabe. oidentd ident daemon v1.7.1. available at: http://ojnk.sourceforge.net/, Oct, 22 2000.

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

[15] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, August 2000.

[16] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, and W. T. Strayer S. T. Kent. Hash-Based IP Traceback. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA, August 2001. To Appear.

[17] D. X. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *Proceedings of the IEEE Infocomm 2001*, April 2001.

[18] S. Staniford-Chen and L.T. Heberlein. Holding Intruders Accountable on the Internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 39–49, Oakland, CA, May 1995.

[19] S. F. Wu, L. Zhang, D. Massey, and A. Mankin. Intention-Driven ICMP Trace-Back. IETF Internet draft, February 2001. draft-wu-itrace-intention-00.txt.

[20] T. Ylonen. SSH — Secure Login Connections Over the Internet. In *6th USENIX Security Symposium*, pages 37–42, San Jose, CA, USA, July 1996.

[21] K. Yoda and H. Etoh. Finding a Connection Chain for Tracing Intruders. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, October 2000.

[22] ZDNet Special Report: It's War! Web Under Attack. http://www.zdnet.com/zdnn/special/doswebattack.html, February 2000.

[23] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.