Providing Process Origin Information to Aid in Network Traceback

Florian P. Buchholz
CERIAS
Purdue University
florian@cerias.purdue.edu

Clay Shields
Department of Computer Science
Georgetown University
clay@cs.georgetown.edu

Abstract

It is desirable to hold network attackers accountable for their actions in both criminal investigations and information warfare situations. Currently, attackers are able to hide their location effectively by creating a chain of connections through a series of hosts. This method is effective because current host audit systems do not maintain enough information to allow association of incoming and outgoing network connections. In this paper, we introduce an inexpensive method that allows both on-line and forensic matching of incoming and outgoing network traffic. Our method associates origin information with each process in the system process table, and enhances the audit information by logging the origin and destination of network sockets. We present implementation results and show that our method can effectively record origin information about the common cases of stepping stone connections and denial of service zombies, and describe the limitations of our approach.

1 Introduction

As the Internet has become a widely accepted part of the communications infrastructure there has been an increase in the number of network attacks [18]. One factor in the growth of attacks is that network attackers are only rarely caught and held accountable for their actions, giving them relative impunity in action. This situation has arisen, in part, because of the relative ease that attackers have in hiding their location, making it difficult and expensive for investigators to determine the origin of an attack.

In general, attackers use two different methods to hide their location [16]. One method, common in denial-of-service attacks, is to spoof the source address in IP packet headers so that recipients cannot easily determine the true source. As discussed further below, this has been an area of significant research in recent years. The other method, which has received significantly less attention from the research community, is for attackers to sequentially log into a number of (typically compromised) hosts. These forwarding hosts, often called *stepping-stone* hosts [40], effectively disguise the origin of the connection, as each host on the path sees only the previous host on the *connection chain*. A victim of an attack would not be able to determine the source of an attack without tracing the path back through all intermediate stepping-stone hosts. The audit data currently maintained at hosts is generally insufficient to correlate incoming and outgoing network traffic, so research about this problem has concentrated only on what can be deduced from network-level data. However, streams can be modified or delayed in a host so that a correlation is no longer possible from a network-level point of view, necessitating a host-based solution.

In this paper we will discuss a simple and inexpensive method for maintaining the necessary information to correlate data entering a host with data leaving a host. The goal of this work is to provide additional audit data that can help determine the source of network attacks. We include results from an implementation for the FreeBSD 4.1 kernel that show the technique is effective in providing information useful in tracing common attack situations, particularly for tracing stepping stones and denial-of-service attack zombies.

The next section provides a complete background of related work in the area and provides our view of the problem and design criteria to be addressed in providing a solution. Section 3 describes the technique we use to obtain and maintain location information for each process, and the logging mechanisms that can be used to provide forensic access to the data. Section 4 describes the specific application of the technique to the FreeBSD 4.1 kernel, and is followed in Section 5 by examples of the implementation in action. Section 6 outlines the limitations of our approach, and how these limitations can be addressed in future work. Finally, Section 7 provides a summary of our work.

2 Background

The goal of *network traceback* research is to allow determination of the source of attack traffic, so that a particular host used by a human to initiate an attack can be identified, and real-world investigative techniques used to locate the person responsible.

In order to accomplish this, the two problems described above — locating the source of IP packets and determining the first node of a connection chain – need to be solved. As described below, there has been significant research in locating the source of IP packets, and there have been efforts made to identify connection chains sources by examining network traffic. What is lacking is a reliable method of correlating incoming network traffic to a host with outgoing network traffic emanating from the host. This paper presents a mechanism for doing this. While our method is not always reliable, as discussed in Section 6, we believe that with further research and community involvement, this work can help address what is a serious problem.

2.1 Packet Source Determination

In normal operation, a host receiving packets can determine their source by direct examination of the source address field in the IP packet header. Unfortunately, this address is easy to falsify, making it simple for attackers to send packets that have their source effectively hidden. This is more common for one-way communication, such as the UDP and ICMP packets used in denial-of-service attacks, but has been of use in attacks using TCP streams [21, 2]. There has been significant recent research in how to locate the source of such packets, primarily motivated by distributed denial-of-service (DDoS) attacks in early February of 2000. While it is generally recommended that routers be configured to perform ingress or egress routing [11], it is clear from continuing denial-of-service attacks [20] that this is not widely done. There have been other methods proposed to perform filtering to limit the effect of such attacks [24, 14].

As it is currently not possible to prevent such attacks, recent work has focused on how to locate the source of attacks. Some methods add or collect information at routers to allow traceback of DoS traffic [6, 27, 35, 7, 30]. Other methods add markings to the packets to probabilistically allow determination of the source given sufficient packets [28, 31, 23, 8, 9], or forward copies of packets, encapsulated in ICMP control messages, directly to the destination [1, 37]. A more innovative method uses counter-DoS attacks to locate the source of on-going attacks [4]. While we do not require that these schemes be available, we can make effective use of the traceback information they provide.

2.2 Correlating Streams

Research addressing determination of the source of a connection chain has mainly focused on correlating streams of TCP connections observed at different points in the network. Figure 1 shows an example of a connection chain.

The initial work in matching streams constructed *thumbprints* of each stream based on content [32]. While this technique could effectively match streams, it would be ineffective in compressed or encrypted streams as are common today. Other work compared the rate of sequence number increase in TCP streams as a matching mechanism, which can work as long as the data is not compressed at different hops and does not see excessive network delay [38]. Another technique, which relies solely on the timing of packets in a stream, is effective against encrypted or compressed streams of interactive user data [40]. This work was originally intended for intru-

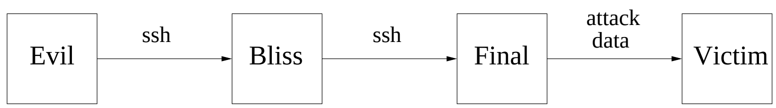


Figure 1: A sample connection chain

sion detection purposes but was also proposed as an effective method for finding the source of connection chains. While performing stream matching might be effective in some cases, such methods rely on examining network information, and might be vulnerable to the same methods that can be used to defeat network intrusion detection systems [26].

2.3 Forensic System Analysis

One of the objectives of computer forensics is the reconstruction of events that occurred on a system. Tools like the *Coroner's Toolkit* [10] attempt to discover hidden and deleted files and use access times to deduce system activities. A more formal model for file and event reconstruction is given by Andrew Gross [13].

In order to solve the host causality problem using forensic tools, it is necessary that network traffic is logged in some fashion on the host. Usually the essential information needed to associate incoming and outgoing traffic is not provided as the default on a system. While tools like `TCP wrappers` do a fine job logging incoming network traffic for the essential services, usually outgoing traffic is not logged at all. Furthermore, at best network activity can be tied to a particular user on the system. Exactly which processes and programs are involved may be obscured if there is a large amount of activity by that user.

2.4 Host causality

Though certain aspects of the network traceback problem have been addressed by the approaches described above, a new area of research that is concerned with data transformations or data flow tracking through a host is needed for a complete picture for attack origin traceback. We call this new area *host causality*, because we are attempting to determine what network input causes other network output.

Common operating systems do not currently provide information that can match incoming and outgoing network traffic. While there has been some work that attempts to use existing system information to match active incoming and outgoing streams [15, 5], this work has been either shown to be impractical to securely implement [3], or requires an external trigger to store forensic information. Ideally, it should be possible to determine whether network traffic was originated directly from a particular host, or occurred as a result of a connection from some other remote machine, and, if possible, which remote machine is involved. This would not only help in tracing back to the source of a network attack, but could be useful in showing due diligence, so that the owner of a machine used in attack could demonstrate that the attack originated elsewhere.

A solution that addresses the problem of tracing connections through a host is necessary because a host on the network can transform data passing through it in such a way that, from the network's point of view, it can no longer be easily related to traffic leaving it. This might be the case in a stepping stone scenario, if the traffic is delayed, or differently compressed or encrypted. Also, in attacks like a distributed denial-of-service (DDoS) attack [39], control traffic cannot be linked to the resulting attack traffic. In such an attack, packet source-location techniques might identify the source of a particular attack stream, but will not allow identification of the master or the controlling host. This is due to the fact that the datagrams that are used to perform the attack are seemingly unrelated to those that control the client. What is missing is information within the host that can be used to associate an incoming control packet with outgoing attack data.

2.4.1 Desired Properties

The following properties either need to be fulfilled or seem desirable in order to achieve a practical solution to the host causality problem:

1. It must be possible to determine whether a given process on the host was started by a local user or remotely.
2. If a process was started by a user at a remote location, information about that source must be maintained and associated with the process.
3. An audit facility must exist that allows the logging of incoming network traffic and processes that receive it. This will allow correlation between the source of a process and the source of incoming network packets.
4. An audit facility must exist that allows the logging of individual outgoing network traffic and processes that send it. Combined with the facility above, one could then relate incoming and outgoing traffic processed by the same process.
5. The logs maintained about origin information should be resistant to modifications by attackers.
6. Processes that spawn other processes need to pass on their source information to their children, or, if they provide a remote login service, pass on the remote location as the child's new source.
7. The modifications to a system should be minimal so that they do not interfere with existing software.
8. Due to restricted logging space, it should be possible to use rules to control what data the audit system collects.
9. It should be possible to quickly identify processes that were not started locally together with their remote location.

3 Description of Model

A process on a computing host is an executing instance of a program [34]. Processes are therefore, among many other things, responsible for receiving and generating network data on a host that is connected to a network.

Processes can be started:

- explicitly by a human being
- by the system

A human being can start processes:

- while physically present at the host
- from a remote location
- indirectly through some other process he or she started

The system can start processes:

- through startup scripts (including **init**)
- through scheduling services like **cron** and **at**
- through system services like **inetd**

The *origin of a process* is the information about how any process running on the system was started in regard to the above possibilities. For the purpose of this paper only a distinction between a process that was started by a human being from a remote location (*remote origin*) and the other ways (*local origin*) is of importance, with the exception of the special case of indirectly started processes.

In case of a remote origin for a process, the origin information should include that remote location. If the system tracks the origin of a process and a process sends out network traffic and is of remote origin, then the system can make a connection between the traffic that was sent out and the traffic that was received from the origin of the process over the network. The traffic could be individual datagrams, or they can be part of an established connection.

In order to gain access to a system from a remote location and start new processes there, a user has to make use of a service offered on that particular system. Usually most systems provide well-known services such as `telnet`, `rsh`, or `ssh` that will give a remote user a shell on the system. However, there are other possibilities to create new processes that do not involve an interactive shell. In fact, any process listening on an open port on the system may be used or misused for such purposes. Our solution does not address these problems, and they are a topic for future investigation.

As the only legitimate remote access to a system is through its well-known services, it is feasible to store information about the existing connection with the newly created child process. After a successful login procedure, the source of the new process should reflect the information stored about the connection. Note that the origin of a process and its subsequent children is set at the time a user gains access to the system. All programs that will be started during that remote session will inherit that origin. At this point time delays become irrelevant, as origin information is stored with the processes no matter whether or not processes become dormant for any amount of time.

3.1 Information Storage

From the viewpoint of a host, all that can be deduced about the origin of an arriving network packet is the interface that it arrived on and the information that is contained in the packet itself. A host on its own cannot determine whether a network datagram was spoofed or not. Therefore, for IP packets, the five-tuple consisting of source and destination IP addresses and source and destination ports and the protocol number must suffice to distinguish source information maintained about processes on a host. If packet traceback schemes are deployed and can provide additional information, it is possible to maintain that information as well.

While storing information about active processes can be useful, for complete analysis of attacks, some additional information needs to be logged as well. The logging mechanism can maintain more explicit information than simply storing the IP five-tuples. Along with the five-tuple and timestamp, the system can also store the interface on which the packet arrives and the process id. If the system logs individual packets, it can also store a checksum of the non-changing parts of each packet header that is logged in case the need for a more detailed post-analysis matching arises.

Furthermore, it would be expensive and impractical to log an entire stream of packets that make up the entirety of a TCP stream. Since TCP is a connection oriented transport layer protocol, it is sufficient to only regard incoming and outgoing `SYN` requests for the purposes of logging. Unfortunately, UDP is a connection-less protocol. Thus for UDP, all packets need to be logged. Log-reducing mechanisms that group the same kind of UDP packets together can certainly be applied here, but this is out of the scope of this paper.

3.2 Limitations on Information Availability

For well-known services we can assume that there will only be one open network connection for each child process spawned as they adhere to the common style of running Unix servers that fork for each new request [33]. Non-standard server programs might behave differently, however, and there might be multiple open connections when we try to determine the origin information. In this case, it is impossible to be sure which connection should be considered as the origin of the process. Because of this, there can be a problem with using the latest data from the `accept` system call as the origin information. If a server program allows multiple open sockets before the call to *login*, then there is a possibility that the wrong origin information is stored with the process. It is possible to design a program that after accepting a connection opens another listening socket to receive a decoy connection from a completely different remote site or, even worse, from the local host itself. This would set the information obtained from the `accept` call to the new socket's source data, before *login* was invoked. After a successful login procedure, the origin information would be incorrect. If a local user installs such a program, then any attacks originating from it can be viewed as originating from the host, which is consistent with our definition of local origin.

Another problem is that a remote user may still hide his real origin by creating a connection from the system to itself. In this case the origin information of the process gets changed to the source information of the local host. While the process is still being considered of remote origin, it is of no value from a traceback perspective. If many remote processes "change" their origin in such a fashion, one cannot determine anymore what the "real" origin of any of those was. In order to prevent this obscuring of the origin of a process, one needs to keep track of an *inheritance line* for remote processes. That is, for any given process of remote origin, one must be able to determine its parent process if that parent process also was of remote origin.

4 Implementation

The model described above was implemented in the FreeBSD 4.1 operating system on an i386 based PC. While the implementation is therefore specific to the UNIX operating system, the general principles of the model should be applicable to other systems as well.

All processes that accept network connections do need to make use of the socket system calls provided by the system. Stevens [33] describes the necessary steps to set up a TCP or UDP server. They involve system calls to `bind`, `listen`, and `accept`, in that order. Thus any connection between two systems must have successfully undergone a call to `accept` on the server side. In the case of TCP, `accept` returns after a successful three-way handshake. In the case of UDP, `accept` returns upon reception of a packet that matches the socket characteristics.

As a successful connection implies a successful return from the `accept` system call, it seems reasonable to make modifications there in order to obtain location information. Specifically, with the assumption of only one open network connection, it is sufficient to record the data from the last call to `accept`. This information will then be accessible to the child process created by the `fork` system call. Finally, after a successful login procedure, the source of the new process should reflect the information stored about the connection. As the `login` program lies in user space and not all well-known servers utilize it, it will be necessary to perform this step through one of the system calls such as `setlogin`.

All the necessary information described in Section 3 is available within data structures used by the `accept` system call. Once the connection has been established, the socket descriptor contains the source IP address and the source port of the purported source of the traffic. To determine the destination IP address and port that was used to establish the connection, the system also has to access the protocol control block (PCB) that is associated with the socket and that is pointed to from the socket data structure[1]. The information can be obtained through simple pointer lookups.

4.1 Where to store source information

We decided to maintain the information directly in the process table itself, because it is simple to add another field that contains the necessary information, and creation and termination of processes is handled automatically. The inheritance problem is taken care of as well, as the `fork` system call causes certain fields of the process table to be copied to the child. The only time we therefore need to access the field in the process table is when origin information changes. The disadvantage of this approach is that some auxiliary programs such as `top` and `ps` might have to be adjusted to accommodate the changes.

It is possible to utilize existing logging facilities, such as `syslog` to record the data, or a logging program can develop its own format and location to store the information [25]. Ideally, there would be some mechanism to ensure the integrity of the logs. Write-once, read-many media, or a secure logging facility could be used [29].

4.2 Data structures and kernel modifications

For the source information, a new data type, `struct porigin`, was created as shown in Figure 2.

The `type` field denotes whether the source is local (0) or remote (1). If the type is 0, all other fields are undefined and can be ignored. The next five fields are the typical four-tuple for a TCP or UDP connection, consisting of source and destination IP addresses as well as source and destination ports, plus the protocol number. The last parameter is a timestamp, which denotes the time the connection was established in network time format [19]. Note that the network interface is

[1] See McKusick et al. [17] or Wright and Stevens [36] for further details.

```
struct  porigin {
        char            type;
        struct in_addr  source_ip;
        struct in_addr  dst_ip;
        u_short         source_port;
        u_short         dst_port;
        u_short         proto;
        time_t          tstamp;
};
```

Figure 2: The process origin data structure

not included here but can be obtained with the information stored if necessary.

In order to keep track of the corresponding source information for each process, the process table data structure (`struct proc`) was modified in two locations. It is necessary to retain the actual source information as well as information about the last accepted connection of a process. The latter is needed because all common TCP/IP based network services that provide a remote login facility first accept the connection and then fork off a child process where *login* is called.

Hence, two fields, `origin` and `lastaccept` were added to the process table structure, both of type `struct porigin`. The fields are located in the area that gets copied in the `fork` system call.

The copying of the `origin` field provides a simple and elegant solution for the inheritance mechanism. All it takes is a few more bytes to be copied in the `fork` system call, as the process structure is copied anyway. Thus, a child process always inherits the source information from its parent.

This leaves the question of where the two fields, `lastaccept` and `origin` are to be set. As the name already suggests, `lastaccept` is set in the `accept` system call, after a successful accept of an incoming connection. The modified `accept` system call was implemented as shown in Figure 3, which shows how to retrieve information from the PCB.

Note that `accept1` is called from the actual `accept` system call. The connection will be accepted in the procedure `soaccept`. If the call is successful, the type is set to 1, and the four-tuple is obtained from the PCB associated with the socket via the pointer `inp`. Note that this will only work for a TCP connection, which is used by services which provide a shell. For future work, other protocol types need to be considered. For instance, in the case of UDP, the `recvfrom` system call may be modified in a similar fashion.

The `origin` field will be copied from a parent process to its child. However, as discussed above, each time a *login* is performed within a process, the source information of the last accepted socket should become the new origin information for that process. Thus, at an invocation of *login*, the `lastaccept` field should be copied into the `origin` field. However, as discussed above, *login* is only a program in user space that simply utilizes several system calls to perform the actual user login. One could supply a separate system call to have the `lastaccept` field copied to the `origin` field, but that would imply that every program that supplies a login service to be changed and use it. Therefore, one of the system calls used by every login service, `setlogin` was modified so that the field is copied after a successful call.

To keep track of the inheritance line for a remote process, it is necessary to modify the `fork` system call, as well. It is sufficient to record the process IDs of the parent and child processes in case the parent is of remote origin. From this information, it is possible to reconstruct the entire inheritance line for a remote process up to the first parent that was of remote origin. The `syslog` facility provides an easy way to log kernel messages, and was chosen to record the information out of reasons of simplicity. Figure 4 shows the modifications made to `fork`. In future work, this recording mechanism needs to be refined and optimized.

```
accept1(p, uap, compat)
...
{
        (void) soaccept(so, &sa);
        inp = sotoinpcb(so); // pointer to protocol control block
        populate fields in p->lastaccept from information
          pointed to by inp;
        p->lastaccept.type = 1;
...
}
```

Figure 3: The modified accept system call (pseudo-code)

4.3 System calls

In order to access the source information for a given process, a new system call, `getorigin` was added to the system. It takes as parameters a process identifier and a buffer, into which the source information is copied.

Note that there is no system call to set or reset the origin field. With the `getorigin` system call, it is now possible to design logging facilities and administrative programs within user space that make use of the source information of a process. For reasons of simplicity, the call was implemented to be unrestricted.

Another system call, `portpid`, was added to give support for the logging facility described below. If one wants to associate incoming TCP or UDP packets with the receiving process, one needs to find the process id of the socket that will handle an IP packet. The same is true for sockets that are responsible for outgoing packets. Those sockets are identified in the network layer by the four-tuple of source and destination addresses and ports, but, unfortunately, there is no mechanism in FreeBSD to obtain that information within user space. Thus the system call `portpid` will take such a four-tuple as well as a protocol identifier (TCP or UDP) and will return the process id of the process that belongs to the listening socket that will accept packets matching the four-tuple, or belonging to the socket that sent the packet. If there is no such socket, an error will be returned. A weakness of this design is that a process may exit and be removed from the process table before the `portpid` call occurs. More verbose logging could offset this problem.

The FreeBSD operating system uses *protocol control blocks* (PCBs) to demultiplex incoming IP packets. The PCBs are chained together in a linked list and contain IP source and destination addresses and TCP or UDP ports or wildcard entries for incoming packets to match against. Each PCB also contains a pointer to the socket that is destined to receive a packet, should it match the four-tuple specified in the PCB. From the socket, one can then look in the receive or write buffer to obtain the actual process id of the receiving or sending process, respectively. In order to determine which process will receive a packet or which process sent a packet, one needs to traverse the list of PCBs until the best match is found, and then obtain the process id of the socket associated with the PCB.

4.4 Logging facility

The logging facility that was implemented is merely a proof of concept, and there are many feasible ways to design and implement one. Our implementation of the logging facility uses the `libpcap` library, which is part of the Berkeley Packet Filter (BPF). The BPF will make a copy of each incoming and outgoing network packet that matches given filter criteria and supply that copy to the process utilizing the filter.

This prototype logging facility can therefore be considered as a network sniffer, but a more robust and efficient implementation would be one that is part of the kernel itself. For each TCP SYN or UDP packet seen by the sniffer, the `portpid` system call is invoked to obtain the process id of the process responsible for the packet. Once the process id is obtained, `getorigin` is called for that process id to determine whether the process is of remote origin or not. If it is of remote origin, then the packet as well as the origin

```
int
fork(p, uap) {
        ...
        error = fork1(p, RFFDG | RFPROC, &p2);
        if (error == 0) {
                p->p_retval[0] = p2->p_pid;
                p->p_retval[1] = 0;
                if (p->origin.type)
                        log(LOG_INFO,
                            "remote process %d spawned child %d\n",
                            p->p_pid, p2->p_pid);
        }
        ...
}
```

Figure 4: The modified fork system call

information is printed out. Figure 5(a) shows the interaction of the different parts of the system with the logging facility, and Figure 6 shows the important parts of the routine that processes the packets passed on by the BPF.

There is a problem with logging outgoing UDP packets. The `portpid` system call relies on the socket that sent the packet to be still open so that it can find it in the PCB list. If an application opened a socket, wrote one UDP packet, and immediately closed the socket again, there is a chance that the socket no longer exists when the packet is examined by the logging facility. DDoS clients usually keep the socket they send packets from open so that packets can be sent at a faster rate, but for outgoing control packets, this is a problem. For TCP, this is not a serious issue, as there is either a three-way handshake or a time-wait period at the end of each connection.

One method to solve the outgoing UDP packet problem could entail further modification of the kernel, keeping the process ids of sending processes in a cache and making that information available to the `portpid` system call. A similar approach could also improve lookup performance for incoming packets. Instead of duplicating the de-multiplexing effort made in the networking stack, modifications to the stack could result in a new data structure that returns the correct process id for a given five-tuple.

5 Implementation Results

The modified kernel was installed on an Intel Pentium III 866 MHz Celeron PC. The machine used is part of a small networking lab. We will discuss the effects of the changes on the normal system behavior as well as give two examples of processes of remote origin handling traffic.

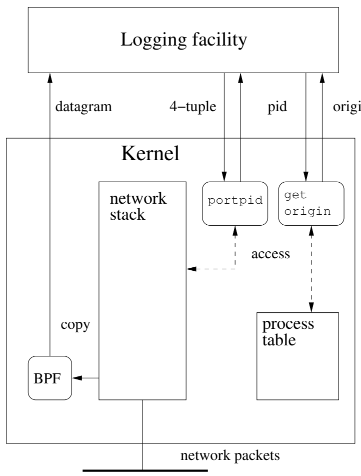
5.1 Effects on normal system behavior

As the changes to the system were only few and cheap, the impact on the system is minimal. The `getorigin` copies a few bytes from the process table and is only executed for TCP SYN and UDP packets. For those packets, the call to `portpid` causes a linked-list traversal of the protocol control blocks in the same manner the networking stack does its de-multiplexing. In every call to `accept`, the `lastaccept` field is set from the socket information. These operations are very few and inexpensive compared to the entire set of operations within `accept`. In every call to `fork`, an extra few bytes need to be copied to pass on the origin information to a process's child. The way the `syslog` facility was used to keep record of an inheritance line is very inefficient. On a system where processes spawn many children, the logs may quickly wrap around. That and the fact that the inheritance line needs to be reconstructed manually from the logs suggests the need for a redesign of the inheritance line for future work.

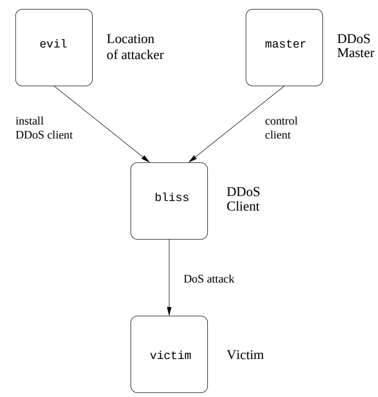
5.2 Examples

5.2.1 Stepping Stone

In this example, `bliss` was used as a stepping stone. A user from `evil` (`10.0.0.1`) logged into `bliss` (`192.168.0.1`) via `ssh`. From there, he used `ssh` again, to log into `final` (`172.16.0.1`). The actual host names and IP addresses have been replaced by fictitious



(a) The logging facility



(b) Setup for the DDoS attack

Figure 5: The logging facility and DDoS attack experimental setup

ones. This setup is equal to the example given in Figure 1 with the exception of the very last host.

The logging facility recorded the following entry from this:

```
192.168.0.1:1022->172.16.0.1:22 sent by pid 285
Origin: 10.0.0.1:1022-192.168.0.1:22
```

One can observe, that the origin information indicates the connection from `evil`, port 1022, to `bliss`, port 22 (`ssh`). The logging mechanism didn't log the connection from `evil` to `bliss`, as `sshd` is a local process. However, `evil` is clearly shown as the origin for the process that connected to `final`. Therefore one can now associate the stream from `bliss` to `final` to the one from `evil` to `bliss` for traceback purposes.

5.2.2 DDoS Client

In this example, a DDoS `trinoo` client, obtained from the Packet Storm archive [22], was installed on `bliss` from `evil`. The corresponding master was installed on another machine, `master` (`192.168.0.2`). `Bliss` was then used via `master` to perform a denial of service attack against `victim` (`172.16.0.2`), a third machine in the test network. Figure 5(b) shows the setup for the attack. Again, host names and IP addresses have been changed.

A sample of the logging output is presented in Figure 7.

The first logged event is a UDP packet from `bliss` to `master`, notifying the `trinoo` master that a client is active. The next event is then a UDP packet from `master` to `bliss`, triggering the DoS attack. The rest of the log shows UDP packets sent from `bliss` to `victim` as part of the attack.

All the traffic can be unambiguously associated with the process 3760, the DDoS client. From the origin, one can see that the process was started from `evil`. In this example, it is clear that the attack was controlled from `master`. This might not always be possible, as multiple packets from different locations could be received by the process just before an attack. However, by examining the logs a good estimate might be derived. At the very least it will give a list of possible hosts from where the attack was launched. Network traceback mechanisms can now be used to determine the location from where the software was set up and `master` could now be investigated in the same manner as `bliss` to determine more information about the attack

```
if (protocol is TCP) {
  set pointer to TCP header within the packet;
  remember source and destination ports;
  if (this is the start of a new connection)
    set log flag;
}
else if (protocol is UDP) {
  set pointer to UDP header within the packet;
  remember source and destination ports;
  set log flag;
}
if (log flag is set) {
  if (packet is coming in)
        invoke portpid with parameters for incoming packets;
  else if (packet is going out)
        invoke portpid with parameters for outgoing packets;
  else
      set error;

  if (portpid returned successfully) {
    call getorigin with pid returned by portpid;
    if (origin is remote) {
      if (packet is coming in)
        print log for incoming packets;
      else if (packet is going out)
        print log for outgoing packets;
    }
  }
```

Figure 6: The packet processing routine of the logging facility (pseudo-code)

```
192.168.0.1:1117->192.168.0.2:31335 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419

192.168.0.2:39805->192.168.0.1:27444 (17) received by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419

192.168.0.1:1135->172.16.0.2:12865 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
192.168.0.1:1135->172.16.0.2:59850 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
192.168.0.1:1135->172.16.0.2:10435 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
192.168.0.1:1135->172.16.0.2:4577 (17) sent by pid 3760
Origin: 10.0.0.1:32155-192.168.0.1:13419
```

Figure 7: Output of the logging facility

and the location of the attacker.

6 Limitations and Future Work

This paper presents a first attempt at a mechanism designed to address the problem of determining host causality. While it is progress in a forward direction, it is not a complete solution to a problem, though its use could prove beneficial in many cases. We hope that discussion of the limitations will foster other research on the problem.

While available origin information is maintained for processes that utilize `setlogin`, there are other mechanisms that attackers can use to start processes on a system. Remotely, attackers might gain access to a system using processes that service network requests, such as mail, web, or ftp servers. Exploits such as buffer overflows against these processes can produce user shells for the attacker, bypassing the system call. In these cases, origin information will not be properly recorded. For these cases the question arises when exactly to set the origin information so that it is meaningful. Furthermore, an attacker who gains access to a system might use a `cron` or `at` job to create a process after the attacker has logged off; this would also result in processes that lack the correct origin information. A solution to this problem might be to include origin information in the file system so that when the new process was started the appropriate location information was available.

Sometimes login servers can open a second connection to the client for out-of-band data. Currently this scenario is not handled in the design. However, its seems that in the worst case the wrong port is recorded for the origin within the modified `accept` system call.

An attacker also might use a covert channel between processes to obscure the proper location information. In this scenario, an attacker, who perhaps enters the system though a mechanism that invokes `setlogin` and whose processes therefore have correct origin information, uses some form of IPC to cause a process that has other origin information to send data into the network. This is a difficult problem to deal with, as it has always been [12], and we do not have an immediate solution for it. Any process that listens on a covert channel needs to have been started either locally or remotely, however, and in the case of an external attacker, most likely remotely. Thus, any outgoing traffic from that process will still be logged.

While our implementation only operates on TCP and UDP packets, any protocol could be used by an attacker. For example, some DDoS tools use ICMP messages to send control messages over the network. In this case, an attacker would either have to modify the routines for ICMP processing in the kernel or may have to sniff the incoming traffic using a library like `libpcap`. If the attacker has modified the kernel to listen to and process these messages, there seems to be little that can be done to establish the origin information for a process, because if the kernel can been modified by the attacker, the origin information can be tampered with as well. In the latter case one can check for open BPF filters and also be aware of processes that utilize other protocols or do not receive network packets from the networking stack but rather through the packet filter.

The mechanism for keeping track of the inheritance line for a process needs to be improved. The current mechanism, while very simple to implement, is the only part of the modifications we made that affects the system in a noticeable fashion. One problem is that with each new child process, more information needs to be stored, even though it is small. Once a separate data structure for keeping inheritance lines is used, a simple improvement would be to delete inheritance lines or parts of it where all the processes involved have terminated. However, overall management of the inheritance lines remains as future work.

In the event of a system compromise, in which an attacker gains root capabilities, the origin information in the kernel and recorded information in the file system is just as vulnerable to modification or deletion as any other kernel or file system information. We consider this outside of the scope of our work, but point to other work that attempts to make audit information survive such attacks [29], and suggest that current forensics tools could be modified to recover the altered origin information in some cases.

Finally, as mentioned above, the packet logging system is a prototype only; a more effective design would be to include the logging mechanism in the kernel itself. Instead of sniffing for outgoing packets, writes to a network socket would cause the outgoing packet to be logged before the socket could be closed, alleviating the problem with trying to find the source of UDP packets mentioned above. Additionally, the current mechanism logs all TCP SYN and UDP packets, creating a denial-of-service opportunity for attackers to fill up disk space, so a more selective approach to recording packets is clearly in order, where possible.

6.1 Future work in Host Causality

Even though the origin information was designed with network traceback in mind, there are other applications or foundations for new modifications of the system:

- A system administrator can use the origin information to determine the origins of all running processes and identify ones that have a very unusual source. This can lead to the discovery of running DDoS clients on a machine, for example.
- The origin information can be incorporated into the file system. By storing a process's origin information with a file whenever the process writes to the file system. Not only can this help in solving the problem with `cron` and startup scripts, but it can also aid in locating suspicious programs in users' home directories. This would be especially ef-

fective with logging file systems, so that the changes in files could be tracked by location as well.

- Origin information adds another dimension to access control. Access control mechanisms can be altered so that they take origin information into account and grant certain privileges only when certain origin conditions are met.
- Statistics based on origin of processes can be gathered, which can be used to profile normal system behavior or to locate trends that may help in better system administration.

Origin information may well benefit in other security related fields. The prospect of access control in combination with origin information seems to be an especially interesting area. Research in that direction may well improve overall robustness of the origin mechanism itself.

7 Conclusion

In this paper, we have introduced the notion of host causality as a mechanism to complement current research in network traceback. With the addition of origin information to a process, we have developed a mechanism that, with only minor changes to the given system, works well under the simple circumstances. The two examples show that important information for network traceback can be obtained with origin information and the new logging possibilities that result from that.

The work we presented here is only the start of work in the overall area. We have identified many limitations of our mechanism, and outlined what future work needs to be done to better address the problem. Host causality is not a complete solution to all the problems that faced in tracing connections through a network, but providing solutions could prove a valuable tool to help improve security in a future networking environment.

References

[1] S. Bellovin. ICMP Traceback Messages. Technical report, IETF Internet draft, March 2000. Work in progress.

[2] S. M. Bellovin. Security Problems in the TCP-IP Protocol Suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[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 and C. Shields. A Recursive Session Token Protocol for use in Computer Forensics and TCP Traceback. In *Proceedings of the IEEE Infocomm 2002*, 2002. To appear.

[6] H. Chang and D.Drew. DoSTracker. This was a publicly available PERL script that attempted to trace a denial-of-service attack through a series of Cisco routers. It was released into the public domain, but later withdrawn., June 1997.

[7] Characterizing and Tracing Packet Floods Using Cisco Routers. `http://www.cisco.com/warp/public/707/22.html`.

[8] 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.

[9] 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.

[10] D. Farmer and W. Venema. The Coroner's Toolkit (TCT). `http://www.fish.com/tct/`.

[11] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical Report RFC 2827, Internet Society, May 2000.

[12] Virgil Gligor. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, Ft. George G. Meade, Maryland, U.S.A., November 1993. Approved for public release: distribution unlimited.

[13] A. H. Gross. *Analyzing Computer Intrusions*. PhD thesis, University of California, San Diego, 1997.

[14] J. Ioannidis and S. M. Bellovin. Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of the 2002 Network and Distributed System Security Symposium*, San Diego, CA, February 2002.

[15] 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.

[16] 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.

[17] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, Boston, MA, 1996.

[18] B. McWilliams. CERT: Cyber Attacks Set To Double In 2001. `http://www.sercurityfocus.com/news/266`.

[19] D.L. Mills. Network Time Protocol. RFC 1059, July 1988.

[20] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Proceedings of the 2001 USENIX Security Symposium*, Washington D.C., August 2001.

[21] R.T. Morris. A Weakness in the 4.2BSD Unix TCP-IP Software. Technical Report 17, AT&T Bell Laboratories, 1985. Computing Science Technical Report.

[22] Distributed Attack Tools Section. `http://packetstorm.securify.com/distributed/`.

[23] 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.

[24] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *Proceedings of the 2001 ACM SIGCOMM*, San Diego, CA, August 2001. To Appear.

[25] J. Picciotto. The Design of An Effective Auditing Subsystem. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 13–22, 1987.

[26] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[27] J. Rowe. Intrusion Detection and Isolation Protocol: Automated Response to Attacks. Presentation at Recent Advances in Intrusion Detection (RAID), 1999.

[28] 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.

[29] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.

[30] 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.

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

[32] 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.

[33] W. R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall PTR, second edition, 1998.

[34] W.R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, 1993.

[35] R. Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[36] G.R. Wright and W.R. Stevens. *TCP/IP Illustrated Volume 2, The Implementation*. Addison Wesley, Boston, MA, 1995.

[37] S. F. Wu, L. Zhang, D. Massey, and A. Mankin. Intention-Driven ICMP Trace-Back. IETF Internet draft, February 2001. Work in progress.

[38] 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.

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

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