Forensix: A Robust, High-Performance Reconstruction System*

Ashvin Goel†, Wu-chang Feng, David Maier, Wu-chi Feng, Jonathan Walpole

†University of Toronto
ashvin@eecg.toronto.edu

Portland State University
{*wuchang,maier,wuchi,walpole*}*@cs.pdx.edu*

Abstract

When computer intrusions occur, one of the most costly, time-consuming, and human-intensive tasks is the analysis and recovery of the compromised system. At a time when the cost of human resources dominates the cost of CPU, network, and storage resources, we argue that computing systems should, in fact, be built with automated analysis and recovery as a primary goal. Towards this end, we describe the design, implementation, and evaluation of Forensix: a robust, high-precision reconstruction and analysis system for supporting the computer equivalent of "TiVo".

Forensix uses three key mechanisms to improve the accuracy and reduce the human overhead of performing forensic analysis. First it performs comprehensive monitoring of the execution of a target system at the kernel event level, giving a high-resolution, application-independent view of all activity. Second, it streams the kernel event information, in real-time, to append-only storage on a separate, hardened, logging machine, making the system resilient to a wide variety of attacks. Third, it uses database technology to support high-level querying of the archived log, greatly reducing the human cost of performing forensic analysis.

1 Introduction

The goal of Forensix is to build a system that supports reliable reconstruction of all computer system activity for forensic purposes. Forensics is concerned with the capture, analysis and reconstruction of system activities in order to determine, post-facto, how or whether a machine was compromised. While one of the goals of forensics is to generate evidence that can be used in a court of law, forensic analysis also enables system administrators to troubleshoot problems, track down suspicious behavior, and determine the extent of damage to compromised machines. It can even be used to prove or disprove claims of penetration by blackmailers, although in this paper, we will focus only on the discovery and analysis of intrusions. To be effective, a forensics system must gather an accurate, high-resolution image of system activities, sufficient for identifying a wide range of intrusions and answering questions such as "where did the attack come from" and "what security hole was exploited?" The forensic evidence should be gathered in a tamper-resistant way, so that intruders cannot modify it or remove it to obscure their tracks, and the collection mechanism should not render the target system more vulnerable to non-intrusion based assaults such as denial of service attacks. Ideally, the system should also have a small effect on the performance of the target system, and should be affordable in terms of its resource requirements. Finally, it should facilitate efficient and effective post-facto analysis, a process that is currently ad-hoc, time-intensive, manual and error-prone. In order to support such properties, this paper describes the design and implementation of *Forensix*: a high-resolution, analysis and reconstruction tool.

*This material is supported by the National Science Foundation under grant NSF ANI-0230960.

2 Motivation

Currently, when a system is compromised, investigators must manually sift for clues based on the current state of the system and the remaining log files of the system after the attack. This operation method is inherently "lossy", in that vital information about where the hacker connected from, how the hacker entered and what the hacker did after he entered was not collected or may have been deleted by the hacker. Consider a compromise in which the hacker has modified sensitive files such as `.ssh/authorized_keys` or `.rhosts` to gain a backdoor into the system. Upon discovery, it would be ideal if the system administrator could issue simple queries to the forensic system such as:

Query 1: Generate a list of sessions and processes that have written to the compromised file.

Query 2: Generate a system activity log for each session that was generated from *Query 1*.

In another scenario, if system administrators want to discover whether intrusions are happening, they could issue a pre-canned list of queries that identify suspicious behavior such as:

Query 3: Generate a list of all sessions that opened `/var/log/wtmp` or `/var/log/lastlog` in read-write mode (versus append mode).

Query 4: Generate an activity log of each session listed in *Query 3*.

There are many approaches for logging and auditing system usage, including application and system log files, process accounting mechanisms, network traffic traces, and file system checkers. While each has its strengths, none of them provide enough information by themselves to accurately recreate what happened in the system. For example, application and system log files only track events based on what the applications and system administrators think are necessary to log. Process accounting mechanisms only provide information as to how commands are executed and can fail to track what programs are doing internally. If a hacker downloads a binary onto the system and executes it, process accounting alone will not be able to show what the binary has done. For example, in the well-documented Mitnick case, a program called `zap2` was downloaded and compiled on the compromised system. The program was then executed multiple times in order to delete login entries from the system [8]. Network traffic traces alone are also problematic in that sessions are typically encrypted. In addition, even when they are not encrypted, they are targets for insertion and evasion attacks, thus making what has happened ambiguous. It is also difficult to correlate network information directly to higher-level application behavior that elucidates the actual damage done to the target system. Finally, file system activity logs can only detect modifications to files and thus are unable to address attacks in which running processes are compromised directly.

3 Design Goals

To adequately perform forensic analysis, the following goals must be met:

1. **Completeness:** The system should collect and log enough information to completely capture user activity in order to reconstruct all currently known attacks and all future attacks. The system should also be able to glue the *who* (the user) and the *what* (all of the user's activities) together. To support deniability claims and to limit liability, such a system needs to ensure that all activity is logged independent of system load to show certain actions did not happen. In addition, the system should be able to support *fail-closed* operation when logging is compromised or disabled in order to prevent loss of any necessary logging information.

2. **Authenticity:** Since no one should be able to spoof logging messages or tamper with the logging facility, a strong authenticated relationship must be built between the logging facility and the storage system for the log data. In addition, the system should support logging immutability that prevents history from being rewritten. As seen in many cases, log files can be altered, which allows a hacker to change logging history.

3. **Reproducibility:** The forensic system should allow users to accurately determine *who* and *what* for a wide variety of system activities such as incoming and outgoing network connections, and files read or written by processes. It should allow correlating data based on time as well as system abstractions such as processes or sessions. The reconstruction process should be fast and should be independent of the length of time the system has been running.

4. **Efficiency:** The amount of data collected and its encoding size should be minimized. Although one method for achieving the previous goal of completeness is a simple brute-force log of everything, this approach can hinder the ability to perform accurate, high-performance replay, even when the power and capacity of current hardware and software systems is fully leveraged.

4 The Forensix Approach

Figure 1 shows the architecture of Forensix, a system that attempts to meet the design goals listed above. With Forensix, the *target system's* kernel is instrumented with a logging facility. In its current implementation, the logging facility streams system-call traces over a private network interface to a highly-secure *backend storage system*. While system call logging is prone to problems such as race conditions, we are currently adapting our system architecture and approach to incorporate other, more accurate forms of logging such as logging within well-placed locations within the kernel and virtual-machine based logging [3]. Our design is driven by the observation that a successful attack can only be caused by system-calls issued by processes running on the attacked system (provided the system is built correctly). Hence, if *all* system-call activity is captured and can be attributed to users, processes or connections, then it should be possible to accurately reconstruct *all* security incidents, immaterial of the type of attack. As a result, this approach helps satisfy our goal of completeness. In addition to completeness, system call logging provides compactness since

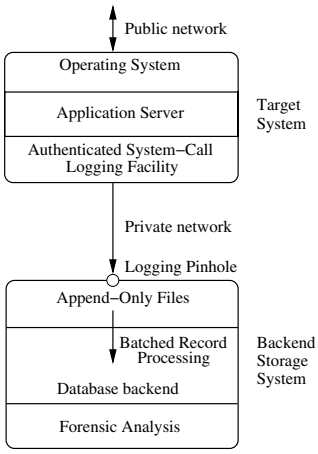

Figure 1. Forensix system architecture.

Forensix does not record other, application-specific, events that do not impact system state. Other methods for improving compactness include data compression and suppressing system-call logging under certain conditions, such as reads to load common shared libraries.

For tamper-proof and immutable operation, Forensix logs system-call activity over a private network interface to a separate, append-only backend storage system. Immutability is achieved via the file system or via CD-R or DVD-R burning while tamper-proof operation is achieved by authenticating each target system at startup and by exporting only a minimum set of network services needed for securely logging system-call data. Specifically, the backend authenticates the logging facility before storing any trace data. It also supports a heartbeat mechanism to detect physical tampering of the logging facility. To further ensure the integrity of the system and to improve the possibility of detecting attacks, both the target and backend systems could be locked down via LIDS in order to ensure safety.

To support efficient and flexible querying, the backend periodically loads log data to a relational database. This forms the basis for accurate and high-performance replay. Queries are efficient because the database allows indexing frequently queried fields such as the user ID and the command executing the system call and the starting time of the system call. In essence, the database holds a data warehouse for forensic analysis and query. While the amount of data being collected can be large, we argue that the system is feasible given the capacity of networking, CPU, and storage capacity available today. As a result, sacrificing some host and networking resources in order to add forensic capability will be a fairly attractive proposition. The following subsections describe the logging facility and the backend storage system in more detail.

4.1 Kernel Logging Facility

To address the problems associated with the piecemeal logging approaches discussed in Section 2, Forensix logs within the kernel. In its current implementation, all activity across the system-call interface is captured and logged. By collecting all system-call activity and attributing this activity to individual connections and sessions, the forensic backend will be able to recreate security incidents in an accurate, application and attack-independent manner. As attacks and attack signatures change, capturing activity at this point thus addresses the problem at a more fundamental, unified level. If the system is built correctly, the hacker will need to figure out a way to compromise a system without using a process, file, or connection in order to go undetected. For accurately attributing system activity to users, processes or connections, the key issue for the logging facility is the *type* and the *amount* of information needed.

4.1.1 Attributing System Activity

The overall design of our logging system is founded on the notion that all intrusions start with a network connection or a console login, are processed by a daemon (`httpd`, `in.telnetd`, `in.ftpd`, `sshd`, `login`, etc.), and cascade into multiple system activities including other processes, file accesses, and outgoing connections. Our high-level goal is to assign these system activities to the initiating session, which helps to simplify and enhance the intrusion-analysis process.

Figure 2 shows a diagram of various system activities and their relationships. The basic idea for capturing these relationships is to assign the identifier or the PID of the process that executes the activity as ownership information to each link of the graph. For example, incoming sessions, file accesses and outgoing connections are all associated with a process, while process creation via `exec` or `fork` is associated with the parent process. Based on the observations above, at a minimum, each system-call trace record has an associated PID and a time-stamp that helps to construct the activity relationship. Section 5 shows that this information allows constructing powerful forensic queries. In the future, we plan on examining other low-overhead mechanisms for unambiguously assigning ownership of activities to individual sessions.

4.1.2 Trace Size

Unlike previous approaches, which only use short sequences of *names* of system calls for intrusion detection [2, 4, 6, 12], Forensix captures each system call and, among other information, its timing, parameters, return values, the process issuing the call, and the owner of that process, throughout the lifetime of the server. This type of in-

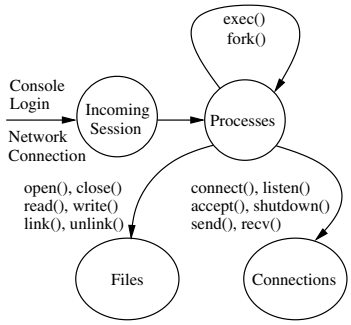

Figure 2. Relationships of system activities.

formation is difficult to collect due to its size and semantic content. However, it is absolutely necessary, as described in the previous section, to recreate system activity.

It is clear that a limiting factor of our approach is the storage space for information being collected. As the capacity for processing and storing auditing information grows, the capacity of the server being traced and its network connection will as well. Fortunately, given the massive amount of unused local networking and storage resources and the ability to acquire such resources at relatively modest costs, we believe that the amount of data being collected is manageable. In our current implementation, a $40 disk can store a complete 6-month activity log of a relatively popular game web site [9].

4.2 Backend Storage System

The main job of the backend is to receive trace data from the network and store it in a form that allows issuing forensic queries. A simple form of storage is log files. While such log files will contain all information needed for performing intrusion analysis, they will not necessarily be in a form amenable for efficient searching and manipulation. For example, we anticipate the need to make forensic queries such as

- Show all user sessions that executed `/bin/sh` from daemon processes other than `sshd`, `telnetd`, or `login` and group sessions by user.
- Show all activity for a particular user session *S*, specified by a source IP address and port, a user ID, and a connection timestamp.

To get some degree of efficiency, it is desirable to index data and ideally provide complete DBMS query processing capabilities to run the types of queries described above. To do so, Forensix stores the trace data in a relational database. While the keys chosen for building indexes depend on the types of queries that are likely to be executed, we have identified three candidate keys, *process ID*, *time* and *incoming connection identifier*, based on our model of attributing ownership to activities, as described earlier in Section 4.1.1.

In addition to fast queries, the backend must provide high throughput storage or else it can become a bottleneck for the target system. The reason is that Forensix ensures that logging information is not lost, i.e. *fail-closed* behavior, by matching target system performance to the ability to log data and blocking the target system when the backend is unable to keep up. A naive approach for building the backend is to continually insert records from the log files into the database. However, this approach places restrictive limits on rates that log file data can be absorbed, particularly because of the indexing overhead typically seen on multiple, small updates. To address this problem, we use bulk loading facilities available in most DBMSs for inserting large batches of traces with deferred indexing. Our experiments show that this approach removes much of the cost of fine-grain index updates. We are currently examining ways to avoid copying data from the log file to the database during loading by constructing log files so that they can be directly mapped into the data space of specific DBMS. To keep the database current for near-time intrusion analysis, we can use time or space-based thresholds to start bulk loading. For example, currently we load the database every 24 hours. If the target system can generate more trace data than the backend can handle in the long term, the backend has the ability to exert back-pressure on the target system, throttling activity to ensure completeness and correctness.

5 Implementation

Forensix has been implemented on Linux and is freely available [1]. The implementation consists of 1) an auditing module on the target system, 2) a receiver daemon and a database injector running on the backend, and 3) database queries and scripts that allow replay of system activities for forensic purposes. Each is described below.

5.1 Target System

The auditing module of the target system consists of a Linux kernel module that traps system calls and logs data in a kernel buffer. The module code, which is derived from the SNARE project [11], hijacks the system call table and adds stub code around several system calls to capture the system call, its timing, its parameters, its return value and the PID of the process making the call. Based on our model of attributing system activity (see Figure 2), the system calls traced fall broadly in the three categories: networking, process management and file system. Network calls include such calls as `connect`, `accept`, `listen` and `shutdown`. Process management calls include `fork`, `exec`,

`kill`, `exit` and `setuid`. Important file system calls include `open`, `read`, `write`, `close`, `symlink`, `link`, `mount`, `unmount`, `dup` and `chown`. Upon collecting the data in a kernel buffer, the audit module periodically transmits the records directly over a private network to the backend, thus avoiding unnecessary data copies and context switches. For fail-closed behavior, the auditing module stops system activity when the kernel buffer becomes full.

While system call auditing allows the audit code to be implemented as a separate module, it can introduce race conditions that prevent system activity from being completely reconstructed [5]. For example, a race condition exists between writing to a file *A* via a symbolic link and modifying the symbolic link to point to a different file *B*. Our stub approach may not accurately capture whether the file *A* or the file *B* was written because the precise timing of the two operations (writing the file and modifying the symbolic link) is not known to the tracing system. A solution to this problem is to capture the output of pathname resolution while reading the symbolic link during the write operation. While this solution is simple and similar techniques can in principle be applied to resolve most timing-related race conditions, the code will be more intrusive than our auditing code.

5.2 Backend System

The receiver daemon on the backend is a small process that reads data from the network and stores it to human-readable, tab-separated, log files. Once every 24 hours, it loads the log files into a database (currently MySQL). In Forensix, the database is optimized for bulk loading (with index generation) and for queries. In particular, data is read-only after it has been loaded and thus transactional guarantees are not essential. The database stores several tables for the system call traces. The main table is called `events`, which stores common attributes, such as id, time, PID and return value, of every system call event. Data from system calls that is unique to specific calls is stored in separate tables to reduce redundancy and to minimize the chances of inconsistency. Examples of such tables include `io`, `dup` and `connections`. The `io` table stores all reads and writes, while the `dup` table stores file opens, closes and file descriptor duplications. The `connections` table stores all network-related system calls.

5.3 Queries

In order to be useful, a powerful set of queries must be supported for post-facto analysis. One problem we faced while implementing queries was uniquely identifying processes based on their PID, which is a 16-bit quantity in Linux that frequently rolls over. To resolve this problem, we augmented the PID with the process creation time. Table 1 lists some examples of queries we have implemented. Simple queries are implemented using SQL directly. For more complex queries, which require conditionals or looping, we use shell programming or code in C for performance. Figure 3 shows samples of SQL and pseudo-code that implement some of these queries. Much of the code should be self-explanatory, which suggests that these queries can be written relatively easily. Note that, the tables in our system are not fully normalized. Doing so would reduce space usage at the expense of some queries taking longer. In terms of performance, some of the queries presented require repeatedly issuing complex SQL sub-queries, which can be expensive. It is possible to optimize the implementation by changing the order of the queries and simplifying the sub-queries. We have done so for the `IO` query, which we evaluate in the next section.

```
Active_Processes(start_time, end_time) {
  SELECT DISTINCT pid FROM event
  WHERE date ¿= %1 AND date ¡= %2;
}

Immediate_Children(PID) {
  SELECT rc FROM event /* return code */
  WHERE (syscall = 2 OR syscall = 120 OR syscall = 190 ) AND pid = %1;
        /* fork() = 2 , clone() = 120, vfork() = 190 */
}

Immediate_Parent(PID) {
  SELECT pid FROM event
  WHERE (syscall = 2 OR syscall = 120 OR syscall = 190) AND rc = %1;
}

FDs_Written(PID, start_time, end_time) {
  SELECT fd, date FROM io, event
  WHERE io.parent = event.id
      AND event.pid = %1 AND event.syscall = 4
      AND event.date ¿ %2 AND event.date ¡ %3;
        /* write() = 4 */
}
```

Figure 3. Pseudo-code for sample queries

6 Evaluation

A viable auditing and replay system should have low auditing overhead, reasonable space requirements and should be able to replay system activity in near-time. Hence, to evaluate Forensix, we performed two types of experiments that measure the performance and space overhead induced by auditing and the time taken to run queries. To measure system overhead, we ran two benchmarks on the target system: 1) Linux kernel build and 2) Webstone. The kernel build benchmark is mainly CPU bound and does not stress the system much. However, it determines the viability of Forensix when running similar applications in a regular desktop environment. The second benchmark, Webstone, stresses a web server and is representative of a loaded server environment.

Query Name	Arguments	Output
Active_Processes	start_time, end_time	List all active processes within a given time interval.
Immediate_Children	PID	List all immediate children of a process.
Children	PID	List all children of a process.
Immediate_Parent	PID	List immediate parent of a process.
Parents	PID	List all parents of a process.
FDs_written	PID, start_time, end_time	List all file descriptors written by a process within a given time interval and the time they were written.
All_FDs	PID, filename, fd_list, time	List all file descriptors that refer to a filename or to other file descriptors in fd_list at a given time.
Did_Process_Write	PID, filename, start_time, end_time	Did process write to filename within a given time interval?
Writers	filename, start_time, end_time	List all processes that wrote to filename within a given time interval.
IO	PID, fd_list	List the timing and the data for I/O performed on file descriptors in fd_list by a process.
Replay_Shell	PID	Run IO query on file descriptors 0, 1 and 2 for a shell process.

Table 1. Examples of Forensix queries.

Our experiments were run on 1.8 GHz Intel Pentium-4 processors with 1 GB of memory. Both the target and the backend machines had the same configuration. In addition, for Webstone, the client process was run on a third similar machine. All the machines are connected with a Gigabit network using a CISCO WS-C4006 switch. The connection between the target machine and the backend machine was on a separate VLAN so it was not affected by other traffic, such as the client to target machine traffic during the Webstone benchmark. All machines run Redhat Linux 2.4.20 with the ext3 file system and the target machine runs the Forensix auditing module. The backend machine uses the MySQL version 3.23 database.

	Total time	System time
No auditing	$233.2s$	$14.0s$
Auditing, No network	$247.1s$	$26.3s$
Auditing with network	$252.0s$	$30.7s$

Table 2. Kernel build times.

	Throughput (Mb/s)
No auditing	296.8
Auditing, No network	276.2
Auditing with network	186.9

Table 3. Webstone throughput.

6.1 Target System

Table 2 shows the results of the kernel build benchmark. The base result for building a kernel under Linux without Forensix auditing corresponds to "No auditing". "Auditing, No network" shows the results when auditing is turned on in the kernel, but the records are not streamed to the backend. Finally "Auditing with network" shows the results when the auditing is turned on and the audit data is streamed to the backend and stored in log files. The numbers in the table are generated by running the time command on the kernel build process.

The table shows that the benchmark completion time in our unoptimized implementation increases by 6% when auditing and by 8% when auditing and transmitting data. We believe that this overhead is a small price to pay for the ability to accurately and systematically reconstruct system state to capture the increasing number of system compromises we see today. Note that, as expected, almost all the additional time is spent in system activity.

The Webstone benchmark stresses a standard Apache web server running on the target system by issuing back-to-back client requests. Figure 3 presents the key results for this benchmark, the throughput achieved by the web server. All the Webstone tests were run for approximately 36 minutes. The "Auditing off" row is the base throughput under Linux without Forensix auditing. The next row shows the throughput when auditing data and retrieving it from the kernel. The decrease in throughput in this case is 7%, which is similar to the overhead observed earlier for the kernel build benchmark.

The final row shows the result when data is also streamed to the backend and stored in log files. In this case, the

throughput decreases by approximately 36%. Currently, we are in the process of profiling the kernel to investigate the reasons for this decreased throughput. However, we believe that there are two obvious optimizations that will help improve our results. First, our implementation is unoptimized and uses a very simple memory allocation mechanism for storing trace data. We expect that improving the auditing module's memory allocator will significantly reduce performance overhead. Second, for simplicity, the auditing module copies code from the kernel to the user space which is then copied back to the kernel to be sent to the backend. To minimize copies, data can be sent to the backend directly from the kernel. This optimization will also help reduce pressure on the memory subsystem.

6.2 Backend System

To evaluate the throughput of the database, we measured the row insertion rate of the database, i.e. the actual number of rows that can be inserted per second in the database. For the Webstone log files, the MySQL database could be bulk loaded at approximately 7400 rows/second. We also measured the row generation rate or the number of rows that are generated per second as data is captured in log files in real-time. For the Webstone test, the row generation rate is 17900 rows/second. This result indicates that for near-time intrusion analysis, where database loading takes less time on average than data generation, the web server can be heavily loaded for no more than 40% time during the day. We expect that this limitation will not be a problem in practice because of typical diurnal server activity.

Next, we measured the space requirements of the compressed log files for the kernel build and the Webstone benchmarks. For the kernel build benchmark, the log files grow at 8.8GB/day, while for the Webstone benchmark they grow at 30GB/day. There are several reasons that these numbers are significantly larger than comparable data generated by ReVirt [3]. The first is that, unlike Forensix, ReVirt does not log filesystem I/O, relying instead on periodic checkpoints whose storage costs are not reported. Moreover, if checkpoints are infrequent, then replaying system activity for forensic analysis can take a long time, as much as the time period since the last checkpoint. The second reason is that we use a Gigabit network in our Webstone experiments and thus produce much more data than the 100 Mb/s network used in evaluating ReVirt. Normalizing for network speed, the Webstone log-file growth rate for Forensix is comparable to ReVirt.

6.3 Queries

In order to be useful, queries must be efficiently supported in near real-time. For evaluation, the Webstone benchmark was re-run and at the same time a user edited the `/etc/passwd` file on the target machine. We executed the `Replay_Shell` query (which uses the `IO` query, see Table 1) with the PID of the shell process in which the password file was modified. This complex query took 100 seconds to run under MySQL, which we believe is a reasonable time to replay this system activity.

7 Forensix in practice

In this section, we describe results from experiences in using Forensix on a production web site as well as using Forensix to analyze a variety of exploits.

7.1 Web Server Reconstruction

We installed Forensix on a web server of a popular, Counter-Strike game community [9] for a week. Besides serving approximately 1000 static pages a day, the web server delivers dynamic content using PHP and MySQL and runs a continuous player statistics aggregation service (HLstats) for a heavily loaded Counter-Strike server (`cs.mshmro.com`). System activity for the web server was stored and loaded into a separate database daily. On the backend, the database grew at a rate of 0.45 ± 0.13 GB/day, a rate that is reasonable for a medium-loaded server. Note that the large deviation occurs since we have no control over user accesses.

After collecting a one-week log of the web server, we ran two queries over the entire database and timed their performance. The first query determined whether there were any `ssh` logins to the system during the week. The second query replayed the complete system activity for one of the logins using `Replay_Shell`. Together, the queries took 708 seconds when no database indices were created. With a PID index on the `event` table, the queries took 488 seconds. While we expect to implement many more queries and optimize our database for them, we believe that our initial implementation is usable.

7.2 Capturing Exploits

In order to demonstrate the utility of Forensix in analyzing exploits, we ran several local privilege escalation attacks on our target system. Local privilege escalation attacks allow a user with normal privileges to gain and retain superuser privileges illegally. The first attack, `ptrace/kmod` [7], exploits a race condition in `kernel/kmod.c` that creates a kernel thread in an insecure manner on behalf of a user process. The flaw allows an unprivileged user process to use `ptrace()` to take control over the privileged `modprobe` binary. The second attack, `mremap` [10], exploits incorrect bound checking of page

```
Unprivileged_seteuid_0(){
  SELECT u_id, g_id, pid, ppid, name source_path, pwd FROM event
  WHERE syscall = 11 AND eu_id = 0 AND u_id ¡¿ 0 AND rc = 0
        /* execve() = 11*/
}

Seteuid_0_to_setuid_0(pid, ppid, name) {
  SELECT name FROM event
  WHERE (syscall = 23 OR syscall = 70 OR syscall = 164)
      AND u_id = 0 AND eu_id = 0 AND pid = %1
      AND ppid = %2 AND name = %3 AND rc = 0
        /* setuid() = 23 ; setreuid = 70 ; setresgid = 164 */
 UNION
  SELECT name FROM event
  WHERE (syscall = 46 OR syscall = 71 OR syscall = 170)
      AND g_id = 0 AND eg_id = 0 AND pid = %1
      AND ppid = %2 AND name = %3 AND rc = 0
        /* setgid() = 23 ; setregid = 70 ; setresgid = 170 */
}
```

Figure 4. Local privilege escalation query

Time period analyzed	6 hours
# of system calls	5875698
Size of database	0.6 GB
Time taken	$< 1s$

Table 4. Performance of local privilege escalation query

counters in the `mremap()` system call. The vulnerability allows an attacking process to execute arbitrary code with kernel level access.

In order to construct a query that identifies such attacks regardless of the vulnerability they exploit, we use the fact that such classes of attacks all follow a common pattern. Specifically, through the use of an elevated effective user ID (`eu_id`) of 0 that they are temporarily granted via a setuid binary, they illegally change their real user ID (`u_id`) permanently to spawn a root shell. Note that in its current form, this query captures all successful occurrences of legitimate `su` invocations. This is necessary since a hacker can easily rename all of his/her local exploits to `su` before invocation if the system filtered escalations based on the name of the setuid binary. Given this pattern and with the understanding that a local root exploit takes a user process with normal privileges and converts it into a shell with superuser privileges, we wrote the query shown in Figure 4 that successfully returns only the PIDs of attacking processes that have successfully executed the `ptrace/kmod` and `mremap` exploits.

Because of its generality, the above query is executed daily to identify and extract all local root exploits that have occurred on the system. Table 4 shows the performance of the query on a sample 6-hour trace in which both exploits were successfully executed. As the table shows, the single query which uncovers both exploits takes under a second to execute.

8 Acknowledgments

The authors would like to thank Mike Shea, Kenneth Po, Kamran Farhadi, Jin Choi, Sourabh Ahuja, Ho-Jeong An, Gary Yeung, Miria Grunick, and Jennifer Johnson for their contributions to the Forensix project.

References

[1] 4N6 Developers. The Forensix Project. `http://forensix.sourceforge.net/`.

[2] M. Crosbie and B. Kuperman. A Building Block Approach to Intrusion Detection. In *Recent Advances in Intrusion Detection (RAID 2001)*, Davis, California, October 2001. Springer.

[3] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of OSDI*, December 2002.

[4] E. Eskin, W. Lee, and S. Stolfo. Modeling System Calls for Intrusion Detection with Dynamic Window Sizes. In *Proceedings of DARPA Information Survivability Converence and Exposition II*, June 2001.

[5] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[6] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[7] iSEC Security Research. Linux Kernel ptrace/kmod Local Root Exploit. `http://downloads.securityfocus.com/vulnerabilities/exploits/ptrace-kmod%.c`, 2003.

[8] K. Mitnick. Takedown Transcripts: 1995 Feb 5 11:48:08. `http://www.takedown.com/cgi-bin/transcript.pl?4002`, February 1995.

[9] mshmro.com. `http://www.mshmro.com/`.

[10] P. Starzetz. Linux kernel do_mremap VMA limit local privelege excalation vulnerability. `http://www.isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt`, 2004.

[11] L. Purdie and G. Cora. SNARE - System iNtrusion Analysis & Reporting Environment. `http://www.intersectalliance.com/projects/Snare/`. Viewed in Jan 2003.

[12] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the USENIX Security Symposium*, pages 185–198, August 2000.