Backtracking Intrusions

SAMUEL T. KING and PETER M. CHEN
University of Michigan

Analyzing intrusions today is an arduous, largely manual task because system administrators lack the information and tools needed to understand easily the sequence of steps that occurred in an attack. The goal of BackTracker is to identify automatically potential sequences of steps that occurred in an intrusion. Starting with a single detection point (e.g., a suspicious file), BackTracker identifies files and processes that could have affected that detection point and displays chains of events in a dependency graph. We use BackTracker to analyze several real attacks against computers that we set up as honeypots. In each case, BackTracker is able to highlight effectively the entry point used to gain access to the system and the sequence of steps from that entry point to the point at which we noticed the intrusion. The logging required to support BackTracker added 9% overhead in running time and generated 1.2 GB per day of log data for an operating-system intensive workload.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls: invasive software (e.g., viruses, worms, Trojan horses)*; K.6.4 [**Management of Computing and Information Systems**]: System Management-management audit; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses); unauthorized access (e.g., hacking, phreaking)*

General Terms: Management, Security

Additional Key Words and Phrases: Computer forensics, intrusion analysis, information flow

1. INTRODUCTION

The frequency of computer intrusions has been increasing rapidly for several years [CERT 2002a]. It seems likely that, for the foreseeable future, even the most diligent system administrators will continue to cope routinely with computer breakins. After discovering an intrusion, a diligent system administrator should do several things to recover from the intrusion. First, the administrator

This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085, ARDA grant NBCHC030104, and Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Authors' address: Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science University of Michigan, Ann Arbor, MI 48109; email: {kingst, pmchen}@umich.edu.

should understand how the intruder gained access to the system. Second, the administrator should identify the damage inflicted on the system (e.g., modified files, leaked secrets, installed backdoors). Third, the administrator should fix the vulnerability that allowed the intrusion and try to undo the damage wrought by the intruder. This article addresses the methods and tools an administrator uses to understand how an intruder gained access to the system.

Before an administrator can start to understand an intrusion, she must first detect that an intrusion has occurred [CERT 2001]. There are numerous ways to detect a compromise. A tool such as TripWire [Kim and Spafford 1994] can detect a modified system file; a network or host firewall can notice a process conducting a port scan or launching a denial-of-service attack; a sandboxing tool can notice a program making disallowed or unusual patterns of system calls [Goldberg et al. 1996; Forrest et al. 1996] or executing foreign code [Kiriansky et al. 2002]. We use the term *detection point* to refer to the state on the local computer system that alerts the administrator to the intrusion. For example, a detection point could be a deleted, modified, or additional file, or it could be a process that is behaving in an unusual or suspicious manner.

Once an administrator is aware that a computer is compromised, the next step is to investigate how the compromise took place [CERT 2000]. Administrators typically use two main sources of information to find clues about an intrusion: system/network logs and disk state [Farmer and Venema 2000]. An administrator might find log entries that show unexpected output from vulnerable applications, deleted or forgotten attack toolkits on disk, or file modification dates which hint at the sequence of events during the intrusion. Many tools exist that make this job easier. For example, Snort can log network traffic; Ethereal can present application-level views of that network traffic; and The Coroner's Toolkit can recover deleted files [Farmer 2001] or summarize the times at which files were last modified, accessed, or created [Farmer 2000] (similar tools are Guidance Software's EnCase, Access Data's Forensic Toolkit, Internal Revenue Services' ILook, and ASR Data's SMART).

Unfortunately, current sources of information suffer from one or more limitations. Host logs typically show only partial, application-specific information about what happened, such as HTTP connections or login attempts, and they often show little about what occurred on the system after the initial compromise. Network logs may contain encrypted data, and the administrator may not be able to recover the decryption key. The attacker may also use an obfuscated custom command set to communicate with a backdoor, and the administrator may not be able to recover the backdoor program to help understand the commands. Disk images may contain useful information about the final state, but they do not provide a complete history of what transpired during the attack. A general limitation of most tools and sources of information is that they intermingle the actions of the intruder (or the state caused by those actions) with the actions/state of legitimate users. Even in cases where the logs and disk state contain enough information to understand an attack, identifying the sequence of events from the initial compromise to the point of detection point is still largely a manual process.

This article describes a tool called *BackTracker* that attempts to address the shortcomings in current tools and sources of information and thereby help an administrator more easily understand what took place during an attack. Working backward from a detection point, BackTracker identifies chains of events that could have led to the modification that was detected. An administrator can then focus her detective work on those chains of events, leading to a quicker and easier identification of the vulnerability. In order to identify these chains of events, BackTracker logs the system calls that induce most directly dependencies between operating system objects (e.g., creating a process, reading and writing files). BackTracker's goal is to provide helpful information for most attacks; it does not provide complete information for every possible attack.

We have implemented BackTracker for Linux in two components: an on-line component that logs events and an off-line component that graphs events related to the attack. BackTracker currently tracks many (but not all) relevant operating-system (OS) events. We found that these events can be logged and analyzed with moderate time and space overhead and that the output generated by BackTracker was helpful in understanding several real attacks against computers we set up as honeypots.

2. DESIGN OF BACKTRACKER

BackTracker's goal is to reconstruct a time-line of events that occur in an attack. Figure 1 illustrates this with BackTracker's results for an intrusion on our honeypot machine that occurred on March 12, 2003. The graph shows that the attacker caused the Apache Web server (httpd) to create a command shell (bash), downloaded and unpacked an executable (/tmp/xploit/ptrace), then ran the executable using a different group identity (we believe the executable was seeking to exploit a race condition in the Linux ptrace code to gain root access). We detected the intrusion by seeing the ptrace process in the process listing.

There are many levels at which events and objects can be observed. Application-level logs such as Apache's log of HTTP requests are semantically rich. However, they provide no information about the attacker's own programs, and they can be disabled by an attacker who gains privileged access. Network-level logs provide more information for remote attacks, but they can be rendered useless by encryption or obfuscation. Logging low-level events such as machine instructions can provide complete information about the computer's execution [Dunlap et al. 2002], but these can be difficult for administrators to understand quickly.

BackTracker works by observing OS-level objects (e.g., files, filenames, processes) and events (e.g., system calls). This level is a compromise between the application level (semantically rich but easily disabled) and the machine level (difficult to disable but semantically poor). Unlike application-level logging, OS-level logging cannot separate objects within an application (e.g., user-level threads), but rather considers the application as a whole. While OS-level semantics can be disrupted by attacking the kernel, gaining kernel-mode control can be made considerably more difficult than gaining privileged user-mode control [Huagang 2000]. Unlike network-level logging, OS-level events can be

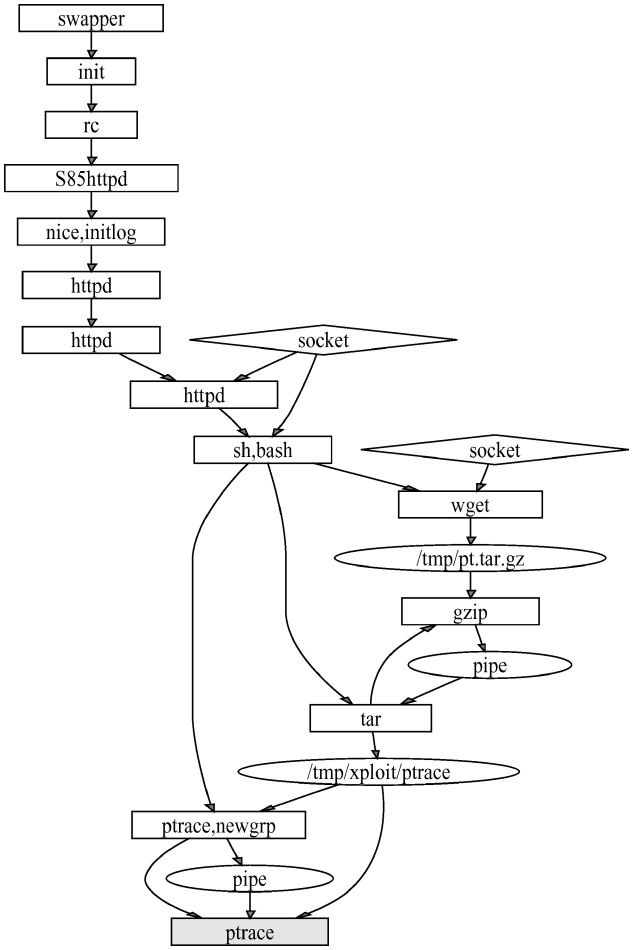


Fig. 1. Filtered dependency graph for *ptrace attack*. Processes are shown as boxes (labeled by program names called by execve during that process's lifetime); files are shown as ovals; sockets are shown as diamonds. BackTracker can also show process IDs, file inode numbers, and socket ports. The detection point is shaded.

interpreted even if the attacker encrypts or obfuscates his network communication.

This section's description of BackTracker is divided into three parts (increasing in degree of aggregation): objects, events that cause dependencies between objects, and dependency graphs. The description and implementation of BackTracker is given for Unix-like operating systems.

2.1 Objects

Three types of OS-level objects are relevant to BackTracker's analysis: processes, files, and filenames.

A process is identified uniquely by a process ID and a version number. BackTracker keeps track of a process from the time it is created by a fork

or clone system call to the point where it exits. The one process that is not created by fork or clone is the first process (swapper); BackTracker starts keeping track of swapper when it makes its first system call.

A file object includes any data or metadata that is specific to that file, such as its contents, owner, or modification time. A file is identified uniquely by a device, an inode number, and a version number. Because files are identified by inode number rather than by name, BackTracker tracks a file across rename operations and through symbolic links. BackTracker treats pipes and named pipes as normal files. Objects associated with System V IPC (messages, shared memory, semaphores) can also be treated as files, though the current BackTracker implementation does not yet handle these.

A filename object refers to the directory data that maps a name to a file object. A filename object is identified uniquely by a canonical name, which is an absolute pathname with all ./ and ../ links resolved. Note the difference between file and filename objects. In Unix, a single file can appear in multiple places in the filesystem directory structure, so writing a file via one name will affect the data returned when reading the file via the different name. File objects are affected by system calls such as write, whereas filename objects are affected by system calls such as rename, create, and unlink.

It is possible to keep track of objects at a different granularity than processes, files, and filenames. One could keep track of finer-grained objects, such as file blocks, or coarser-grained objects, such as all files within a directory. Keeping track of objects on a finer granularity reduces false dependencies (similar to false sharing in distributed shared memory systems), but is harder and may induce higher overhead.

2.2 Potential Dependency-Causing Events

BackTracker logs events at runtime that induce dependency relationships between objects, that is, events in which one object affects the state of another object. These events are the links that allow BackTracker to deduce timelines of events leading to a detection point. A dependency relationship is specified by three parts: a source object, a sink object, and a time interval. For example, the reading of a file by a process causes that process (the sink object) to depend on that file (the source object). We denote a dependency from a source object to a sink object as $source \Rightarrow sink$.

We use time intervals to reduce false dependencies. For example, a process that reads a file at time 10 does not depend on writes to the file that occur after time 10. Time is measured in terms of an increasing event counter. Unless otherwise stated, the interval for an event starts when the system call is invoked and ends when the system call returns. A few types of events (such as shared memory accesses) are aggregated into a single event over a longer interval because it is difficult to identify the times of individual events.

There are numerous events which cause objects to affect each other. This section describes potential events that BackTracker could track. Section 2.3 describes how BackTracker uses dependency-causing events. Section 2.4 then describes why some events are more important to track than others and

identifies the subset of these dependencies logged by the current BackTracker prototype. We classify dependency-causing events based on the source and sink objects for the dependency they induce: process/process, process/file, and process/filename.

2.2.1 *Process/Process Dependencies.* The first category of events are those for which one process directly affects the execution of another process. One process can affect another directly by creating it, sharing memory with it, or signaling it. For example, an intruder may login to the system through sshd, then fork a shell process, then fork a process that performs a denial-of-service attack. Processes can also affect each other indirectly (e.g., by writing and reading files), and we describe these types of dependencies in the next two sections.

If a process creates another process, there is a parent⇒child dependency because the parent initiated the existence of the child and because the child's address space is initialized with data from the parent's address space.

Besides the traditional fork system call, Linux supports the clone system call, which creates a child process that shares the parent's address space (these are essentially kernel threads). Children that are created via clone have an additional bidirectional parent⇔child dependency with their parent due to their shared address space. In addition, clone creates a bidirectional dependency between the child and other processes that are currently sharing the parent's address space. Because it is difficult to track individual loads and stores to shared memory locations, we group all loads and stores to shared memory into a single event that causes the two processes to depend on each other over a longer time interval. We do this grouping by assuming conservatively that the time interval of the shared-memory dependency lasts from the time the child is created to the time either process exits or replaces its address space through the execve system call.

2.2.2 *Process/File Dependencies.* The second category of events are those for which a process affects or is affected by data or attributes associated with a file. For example, an intruder can edit the password file (process⇒file dependency), then log in using the new password file (file⇒process dependency). Receiving data from a network socket can also be treated as reading a file, although the sending and receiving computers would need to cooperate to link the receive event with the corresponding send event.

System calls like write and writev cause a process⇒file dependency. System calls like read, readv, and execve cause a file⇒process dependency.

Files can also be mapped into a process's address space through mmap, then accessed via load/store instructions. As with shared memory between processes, we aggregate mapped-file accesses into a single event, lasting from the time the file is mmap'ed to the time the process exits. This conservative time interval allows BackTracker to not track individual memory operations or the unmapping or remapping of files. The direction of the dependency for mapped files depends on the access permissions used when opening the file: mapping a file read-only causes a file⇒process dependency; mapping a file write-only causes a process⇒file dependency; mapping a file read/write causes a bidirectional

process⇔file dependency. When a process is created, it inherits a dependency with each file mapped into its parent's address space.

A process can also affect or be affected by a file's attributes, such as the file's owner, permissions, and modification time. System calls that modify a file's attributes (e.g., chown, chmod, utime) cause a process⇒file dependency. System calls that read file attributes (e.g., fstat) cause a file⇒process dependency. In fact, any system call that specifies a file (e.g., open, chdir, unlink, execve) causes a file⇒process dependency if the filename specified in the call exists, because the return value of that system call depends on the file's owner and permissions.

2.2.3 *Process/Filename Dependencies.* The third category of events are those that cause a process to affect or be affected by a filename object. For example, an intruder can delete a configuration file and cause an application to use an insecure default configuration. Or an intruder can swap the names of current and backup password files to cause the system to use out-of-date passwords.

Any system call that includes a filename argument (e.g., open, creat, link, unlink, mkdir, rename, rmdir, stat, chmod) causes a filename⇒process dependency, because the return value of the system call depends on the existence of that filename in the file system directory tree. In addition, the process is affected by all parent directories of the filename (e.g., opening the file /a/b/c depends on the existence of /a and /a/b). A system call that reads a directory causes a filename⇒process dependency for all filenames in that directory.

System calls that modify a filename argument cause a process⇒filename dependency if they succeed. Examples are creat, link, unlink, rename, mkdir, rmdir, and mount.

2.3 Dependency Graphs

By logging objects and dependency-causing events during runtime, BackTracker saves enough information to build a graph that depicts the dependency relationships between all objects seen over that execution. Rather than presenting the complete dependency graph, however, we would like to make understanding an attack as easy as possible by presenting only the relevant portion of the graph. This section describes how to select the objects and events in the graph that relate to the attack.

We assume that the administrator has noticed the compromised system and can identify at least one *detection point*, such as a modified, extra, or deleted file, or a suspicious or missing process. Starting from that detection point, our goal is to build a dependency graph of all objects and events that causally affect the state of the detection point [Lamport 1978]. The part of the BackTracker system that builds this dependency graph is called *GraphGen*. GraphGen is run offline, that is, after the attack.

To construct the dependency graph, GraphGen reads the log of events, starting from the last event and reading toward the beginning of the log (Figure 2). For each event, GraphGen evaluates whether that event can affect any object that is currently in the dependency graph. Each object in the evolving graph has a time threshold associated with it, which is the maximum

```
foreach event E in log { /* read events from latest to earliest */
    foreach object O in graph {
        if (E affects O by the time threshold for object O) {
            if (E's source object not already in graph) {
                add E's source object to graph
                set time threshold for E's source object to time of E
            }
            add edge from E's source object to E's sink object
        }
    }
}
```

Fig. 2. Constructing a dependency graph. This code shows the basic algorithm used to construct a dependency graph from a log of dependency-causing events with discrete times.

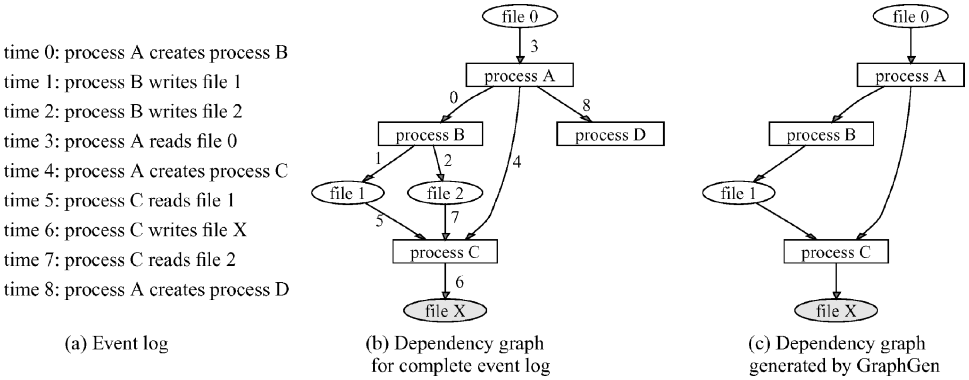


Fig. 3. Dependency graph for an example set of events with discrete times. The label on each edge shows the time of the event. The detection point is file X at time 10. By processing the event log, GraphGen prunes away events and objects that do not affect file X by time 10.

time that an event can occur and be considered relevant for that object. GraphGen is initialized with the object associated with the detection point, and the time threshold associated with this object is the earliest time at which the administrator knows the object's state is compromised. Because the log is processed in reverse time order, all events encountered in the log after the detection point will occur before the time threshold of all objects currently in the graph.

Consider how this algorithm works for the set of events shown in Figure 3a (Figure 3(b) pictures the log of events as a complete dependency graph):

(1) GraphGen is initialized with the detection point, which is file X at time 10. That is, the administrator knows that file X has the wrong contents by time 10.
(2) GraphGen considers the event at time 8. This event does not affect any object in the current graph (i.e., file X), so we ignore it.
(3) GraphGen considers the event at time 7. This event also does not affect any object in the current graph.
(4) GraphGen considers the event at time 6. This event affects file X in time to affect its contents at the detection point, so GraphGen adds process C

to the dependency graph with an edge from process C to file X. GraphGen sets process C's time threshold to be 6, because only events that occur before time 6 can affect C in time to affect the detection point.

(5) GraphGen considers the event at time 5. This event affects an object in the dependency graph (process C) in time, so GraphGen adds file 1 to the graph with an edge to process C (at time 5).

(6) GraphGen considers the event at time 4. This event affects an object in the dependency graph (process C) in time, so GraphGen adds process A to the dependency graph with an edge to process C (at time 4).

(7) GraphGen considers the event at time 3. This event affects process A in time, so we add file 0 to the graph with an edge to process A (at time 3).

(8) GraphGen considers the event at time 2. This event does not affect any object in the current graph.

(9) GraphGen considers the event at time 1. This event affects file 1 in time, so we add process B to the graph with an edge to file 1 (at time 1).

(10) GraphGen considers the event at time 0. This event affects process B in time, so we add an edge from process A to process B (process A is already in the graph).

The resulting dependency graph (Figure 3(c)) is a subset of the graph in Figure 3(b). We believe this type of graph to be a useful picture of the events that lead to the detection point, especially if it can reduce dramatically the number of objects and events an administrator must examine to understand an attack.

The full algorithm is a bit more complicated because it must handle events that span an interval of time, rather than events with discrete times. Consider a scenario where the dependency graph currently has an object O with time threshold t. If an event P⇒O occurs during time interval [x-y], then we should add P to the dependency graph iff $x < t$, that is, this event started to affect O by O's time threshold. If P is added to the dependency graph, the time threshold associated with P would be minimum(t, y), because the event would have no relevant effect on O after time t, and the event itself stopped after time y.

Events with intervals are added to the log in order of the *later* time in their interval. This order guarantees that GraphGen sees the event and can add the source object for that event as soon as possible (so that the added source object can in turn be affected by events processed subsequently by GraphGen).

For example, consider how GraphGen would handle an event process B⇒file 1 in Figure 3(b) with a time interval of 1–7. GraphGen would encounter this event at a log time 7 because events are ordered by the later time in their interval. At this time, file 1 is not yet in the dependency graph. GraphGen remembers this event and continually reevaluates whether it affects new objects as they are added to the dependency graph. When file 1 is added to the graph (log time 5), GraphGen sees that the event process B⇒file 1 affects file 1 and adds process B to the graph. The time threshold for process B would be time 5 (the lesser of time 5 and time 7).

GraphGen maintains several data structures to accelerate its processing of events. Its main data structure is a hash table of all objects currently in the dependency graph, called *GraphObjects*. GraphGen uses GraphObjects to determine quickly if the event under consideration affects an object that is already in the graph. GraphGen also remembers those events with time intervals that include the current time being processed in the log. GraphGen stores these events in an ObjectsIntervals hash table, hashed on the sink object for that event. When GraphGen adds an object to GraphObjects, it checks if any events in the ObjectsIntervals hash table affect the new object before the time threshold for the new object. Finally, GraphGen maintains a priority queue of events with intervals that include the current time (prioritized by the starting time of the event). The priority queue allows GraphGen to find and discard events quickly whose intervals no longer include the current time.

2.4 Dependencies Tracked By Current Prototype

Section 2.2 lists numerous ways in which one object can potentially affect another. It is important to note, however, that *affecting* an object is not the same as *controlling* an object. Dependency-causing events vary widely in terms of how much the source object can control the sink object. Our current implementation of BackTracker focuses on tracking the events we consider easiest for an attacker to use to accomplish a task; we call these events *high-control events*.

Some examples of high-control events are changing the contents of a file or creating a child process. It is relatively easy for an intruder to perform a task by using high-control events. For example, an intruder can install a backdoor easily by modifying an executable file, then creating a process that executes it.

Some examples of low-control events are changing a file's access time or creating a filename in a directory. Although these events can affect the execution of other processes, they tend to generate a high degree of noise in the dependency graph. For example, if BackTracker tracks the dependency caused by reading a directory, then a process that lists the files in /tmp would depend on all processes that have ever created, renamed, or deleted filenames in /tmp. Timing channels [Lampson 1973] are an example of an extremely low-control event; for example, an attacker may be able to trigger a race condition by executing a CPU-intensive program.

Fortunately, BackTracker is able to provide useful analysis without tracking low-control events, even if low-control events are used in the attack. This is because it is difficult for an intruder to perform a task solely by using low-control events. Consider an intruder who wants to use low-control events to accomplish an arbitrary task; for example, he may try to cause a program to install a backdoor when it sees a new filename appear in /tmp.

Using an existing program to carry out this task is difficult because existing programs do not generally perform arbitrary tasks when they see incidental changes such as a new filename in /tmp. If an attacker can cause an existing program to perform an arbitrary task by making such an incidental change, it generally means that the program has a bug (e.g., buffer overflow or race condition). Even if BackTracker does not track this event, it will still be able to

highlight the buggy existing program by tracking the chain of events from the detection point back to that program.

Using a new, custom program to carry out an arbitrary task is easy. However, it will not evade BackTracker's analysis because the events of writing and executing such a custom program are high-control events and BackTracker will link the backdoor to the intruder's earlier actions through those high-control events. To illustrate this, consider in Figure 3(b) if the event "file 1⇒process C" was a low-control event, and process C was created by process B (rather than by process A as shown). Even if BackTracker did not track the event "file 1⇒process C," it would still link process B to the detection point via the event "process B⇒process C."

BackTracker currently logs and analyzes the following high-control events: process creation through fork or clone; load and store to shared memory; read and write of files and pipes; receiving data from a socket; execve of files; load and store to mmap'ed files; and opening a file. We have implemented partially the logging and tracking of file attributes and filename create, delete, and rename (these events are not reflected in Section 5's results). We plan to implement logging and tracking for System V IPC (messages, shared memory, semaphores) and signals.

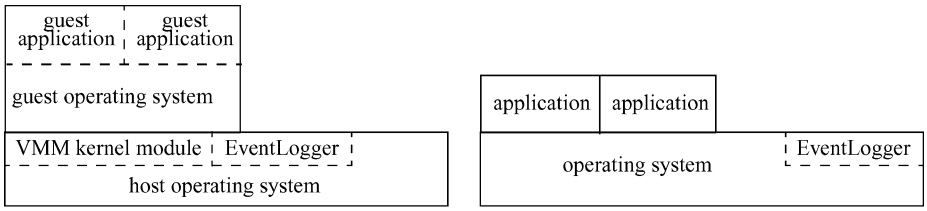
3. IMPLEMENTATION STRUCTURE FOR LOGGING EVENTS AND OBJECTS

While the computer is executing, BackTracker must log information about objects and dependency-causing events to enable the dependency-graph analysis described in Section 2. The part of BackTracker that logs this information is called *EventLogger*. After the intrusion, an administrator can run GraphGen offline on a log (or concatenation of logs spanning several reboots) generated by EventLogger. GraphGen produces a graph in a format suitable for input to the dot program (part of AT&T's Graph Visualization Project), which generates the human-readable graphs used in this article.

There are several ways to implement EventLogger, and the results of BackTracker's analysis are independent of where EventLogger is implemented.

The strategy for our main BackTracker prototype is to run the target operating system (Linux 2.4.18) and applications inside a virtual machine and to have the virtual-machine monitor call a kernel procedure (EventLogger) at appropriate times (Figure 4). The operating system running inside the virtual machine is called the *guest operating system* to distinguish it from the operating system that the virtual machine is running on, which is called the *host operating system*. Guest processes run on the guest operating system inside the virtual machines; host processes run on the host operating system. The entire virtual machine is encapsulated in a host process. The log written by EventLogger is stored as a host file (compressed with gzip). The virtual-machine monitor prevents intruders in the guest from interfering with EventLogger or its log file.

EventLogger gleans information about events and objects inside the target system by examining the state of the virtual machine. The virtual-machine monitor notifies EventLogger whenever a guest application invokes or returns from a system call or when a guest application process exits. EventLogger learns



(a) Virtual-machine implementation (b) Standalone implementation

Fig. 4. System structures for logging events. We have implemented the EventLogger portion of BackTracker in two ways. In the virtual-machine implementation (Figure 4(a)), we run the target operating system and applications in a virtual machine and log events in the virtual-machine monitor running below the target operating system. The virtual-machine monitor (VMM) kernel module calls a kernel procedure (EventLogger), then EventLogger reads information about the event from the virtual machine's physical memory. In the standalone implementation (Figure 4(b)), we run applications directly on the host operating system and log events from within that operating system.

about the event from data passed by the virtual-machine monitor and from the virtual machine's physical memory (which is a host file). EventLogger is compiled with headers from the guest kernel and reads guest kernel data structures from the guest's physical memory to determine event information (e.g., system call parameters), object identities (e.g., file inode numbers, filenames, process identifiers), and dependency information (e.g., it reads the address map of a guest process to learn what mmap'ed files it inherited from its parent). The code for EventLogger is approximately 1300 lines, and we added 40 lines of code to the virtual-machine monitor to support EventLogger. We made no changes to the guest operating system.

Another strategy is to add EventLogger to the target operating system and not use a virtual machine. To protect EventLogger's log from the intruder, one could store the log on a remote computer or in a protected file on the local computer. We have ported EventLogger to a standalone operating system (Linux 2.4.18) to give our local system administrators the option of using BackTracker without using a virtual machine. To port EventLogger to the target operating system, we modified the code that gleans information about events and objects; this porting took one day.

The main advantage of the virtual-machine-based system is its compatibility with ReVirt, which enables one to replay the complete, instruction-by-instruction execution of a virtual machine [Dunlap et al. 2002]. This ability to replay executions at arbitrarily fine detail allows us to capture complete information about workloads (e.g., real intrusions) while still making changes to EventLogger. Without the ability to replay a workload repeatably, we would only be able to analyze information captured by the version of EventLogger that was running at the time of that workload. This ability is especially important for analyzing real attacks, since real attackers do not reissue their workloads upon request. EventLogger can log events and objects during the original run or during a replaying run. All results in this article are collected using the virtual-machine implementation of EventLogger.

One of the standard reasons for using a virtual machine—correctness in the presence of a compromised target operating system—does not hold for

BackTracker. If an attacker gains control of the guest operating system, she can carry out arbitrary tasks inside the guest without being tracked by BackTracker (in contrast, ReVirt works even if the attacker gains control of the guest operating system).

We use a version of the UMLinux virtual machine [Buchacker and Sieh 2001] that uses a host kernel (based on Linux 2.4.18) that is optimized to support virtual machines [King et al. 2003]. The virtualization overhead of the optimized UMLinux is comparable to that of VMWare Workstation 3.1. CPU-intensive applications experience almost no overhead, and kernel-intensive applications such as SPECweb99 and compiling the Linux kernel experience 14–35% overhead [King et al. 2003].

4. PRIORITIZING PARTS OF A DEPENDENCY GRAPH

Dependency graphs for a busy system may be too large to scrutinize each object and event. Fortunately, not all objects and events warrant the same amount of scrutiny when a system administrator analyzes an intrusion. This section describes several ways to prioritize or filter a dependency graph in order to highlight those parts that are mostly likely to be helpful in understanding an intrusion. Of course, there is a tradeoff inherent to any filtering. Even objects or events that are unlikely to be important in understanding an intrusion may nevertheless be relevant, and filtering these out may accidentally hide important sequences of events.

One way to prioritize important parts of a graph is to ignore certain objects. For example, the login program reads and writes the file /var/run/utmp. These events cause a new login session to depend on all prior login sessions. Another example is the file /etc/mtab. This file is written by mount and umount and is read by bash at startup, causing all events to depend on mount and umount. A final example is that the bash shell commonly writes to a file named .bash_history when it exits. Shell invocations start by reading .bash_history, so all actions by all shells depend on all prior executions of bash. While these are true dependencies, it is easier to start analyzing the intrusion without these objects cluttering the graph, then to add these objects if needed.

A second way to prioritize important parts of a graph is to filter out certain types of events. For example, one could filter out some low-control events.

These first two types of filtering (objects and events) may filter out a vital link in the intrusion and thereby disconnect the detection point from the source of the intrusion. Hence they should be used only for cases where they reduce noise drastically with only a small risk of filtering out vital links. The remainder of the filtering rules do not run the risk of breaking a vital link in the middle of an attack sequence.

A third way to simplify the graph is to hide files that have been read but not written in the time period being analyzed (read-only files). For example, in Figure 3(c), file 0 is read by process A but is not written during the period being analyzed. These files are often default configuration or header files. Not showing these files in the graph does not generally hinder one's ability to understand an attack because the attacker did not modify these files in the time period being

considered and because the processes that read the files are still included in the dependency graph. If the initial analysis does not reveal enough about the attack, an administrator may need to extend the analysis further back in the log to include events that modified files which were previously considered read-only. Filtering out read-only files cannot break a link in any attack sequence contained in the log being analyzed, because there are no events in that log that affect these files.

A fourth way to prioritize important parts of a graph is to filter out helper processes that take input from one process, perform a simple function on that input, then return data to the main process. For example, the system-wide bash startup script (/etc/bashrc) causes bash to invoke the id program to learn the name and group of the user, and the system startup scripts on Linux invoke the program consoletype to learn the type of the console that is being used. These usage patterns are recognized easily in a graph: they form a cycle in the graph (usually connected by a pipe) and take input only from the parent process and from read-only files. As with the prior filtering rule, this rule cannot disconnect a detection point from an intrusion source that precedes the cycle, because these cycles take input only from the main process, and the main process is left in the dependency graph.

A fifth way to prioritize important parts of a graph is to choose *several* detection points, then take the intersection of the dependency graphs formed from those dependency points. The intersection of the graphs is likely to highlight the earlier portion of an attack (which affect all detection points), and these portions are important to understanding how the attacker initially gained control in the system.

We implement these filtering rules as options in GraphGen. GraphGen includes a set of default rules which work well for all attacks we have experienced. A user can add to a configuration file regular expressions that specify additional objects and events to filter. We considered filtering the graph after GraphGen produced it, but this would leave in objects that should have been pruned (such as an object that was connected only via an object that was filtered out).

Other graph visualization techniques can help an administrator understand large dependency graphs. For example, a postprocessing tool can aggregate related objects in the graph, such as all files in a directory, or show how the graph grows as the run progresses.

We expect an administrator to run GraphGen several times with different filtering rules and log periods. She might first analyze a short log that she hopes includes the entire attack. She might also filter out many objects and events to try to highlight the most important parts of an intrusion without much noise from irrelevant events. If this initial analysis does not reveal enough about the attack, she can extend the analysis period further back in the log and use fewer filtering rules.

5. EVALUATION

This section evaluates how well BackTracker works on three real attacks and one simulated attack (Table I).

Table I. Statistics for BackTracker's Analysis of Attacks
(This table shows results for three real attacks and one simulated attack. Event counts include only the first event from a source object to a sink object. GraphGen and the filtering rules drastically reduce the amount of information that an administrator must peruse to understand an attack. Results related to EventLogger's log are combined for the bind and ptrace attacks because these attacks are intermingled in one log. Object and events counts for the *self attack* are given for two different levels of filtering.)

	bind (Figures 5–6)	ptrace (Figure 1)	openssl-too (Figure 7)	self (Figure 8)
Time period being analyzed	24 h		61 h	24 h
# of objects and events in log	155,344 objects 1,204,166 events		77,334 objects 382,955 events	2,187,963 objects 55,894,869 events
# of objects and events in unfiltered dependency graph	5,281 objects 9,825 events	552 objects 2,635 events	495 objects 2,414 events	717 objects 3,387 events
# of objects and events in filtered dependency graph	24 objects 28 events	20 objects 25 events	28 objects 41 events	56 (36) objects 81 (49) events
Growth rate of EventLogger's log	0.017 GB/day		0.002 GB/day	1.2 GB/day
Time overhead of EventLogger	0%		0%	9%

To experience and analyze real attacks, we set up a honeypot machine [Cheswick 1992; The Honeynet Project 2001] and installed the default configuration of RedHat 7.0. This configuration is vulnerable to several remote and local attacks, although the virtual machine disrupts some attacks by shrinking the virtual address space of guest applications. Our honeypot configuration is vulnerable to (at least) two attacks. A remote user can exploit the OpenSSL library used in the Apache Web server (httpd) to attain a nonroot shell [CERT 2002b], and a local user can exploit sendmail to attain a root shell [CIAC 2001]. After an attacker compromises the system, they have more-or-less free reign on the honeypot—they can read files, download, compile, and execute programs, scan other machines, etc.

We ran a variety of tools to detect intruders. We used a home-grown imitation of TripWire [Kim and Spafford 1994] to detect changes to important system files. We used Ethereal and Snort to detect suspicious amounts of incoming or outgoing network traffic. We also perused the system manually to look for any unexpected files or processes.

We first evaluate how necessary it is to use the filtering rules described in Section 4. Consider an attack we experienced on March 12, 2003, that we named the *bind attack*. The machine on this day was quite busy: we were the target of two separate attacks (the *bind attack* and the *ptrace attack*), and one of the authors logged in several times to use the machine (mostly to look for signs of intruders, e.g., by running netstat, ps, ls, pstree). We detected the attack by noticing a modified system binary (/bin/login). EventLogger's log for this analysis period covered 24 h and contained 155,344 objects and 1,204,166 events (all event counts in this article count only the first event from a specific source object to a specific sink object).

Without any filtering, the dependency graph generated by GraphGen for this attack contained 5281 objects and 9825 events. While this was two orders of magnitude smaller than the complete log, it was still far too many events and

objects for an administrator to analyze easily. We therefore considered what filtering rules we could use to reduce the amount of information presented to the administrator, while minimizing the risk of hiding important steps in the attack.

Figure 5 shows the dependency graph generated by GraphGen for this attack after filtering out files that were read but not written. The resulting graph contained 575 objects and 1014 events. Important parts of the graph are circled or labeled to point out the filtering rules we discuss next.

Significant noise came from several root login sessions by one of the authors during the attack. The author's actions are linked to the attacker's actions through /root/.bash_history,/var/log/lastlog, and /var/run/utmp. /etc/mtab also generates a lot of noise, as it is written after most system startup scripts and read by each bash shell. Finally, a lot of noise was generated by helper processes that take input only from their parent process, perform a simple function on that input, then return data to the parent (usually through a pipe). Most processes associated with S85httpd on the graph are helper processes spawned by find when S85httpd starts.

Figure 6 shows the dependency graph for the *bind attack* after GraphGen applied the following filtering rules: ignore files that were read but not written; ignore files /root/.bash_history, /var/run/lastlog, /var/run/utmp, /etc/mtab; ignore helper processes that take input only from their parent process and return a result through a pipe. We used these same filtering rules to generate dependency graphs for all attacks.

These filtering rules reduced the size of the graph to 24 objects and 28 events, and made the *bind attack* fairly easy to analyze. The attacker gained access through httpd, downloaded a rootkit using wget, then wrote the rootkit to the file "/tmp/ /bind." Sometime later, one of the authors logged in to the machine noticed the suspicious file and decided to execute it out of curiosity (don't try this at home!). The resulting process installed a number of modified system binaries, including /bin/login. This graph shows that BackTracker can track across several login sessions. If the attacker had installed /bin/login without being noticed, then logged in later, we would have been able to backtrack from a detection point in her second session to the first session by her use of the modified /bin/login.

Figure 1 shows the filtered dependency graph for a second attack that occurred in the same March 12, 2003, log, which we named the *ptrace attack*. The intruder gained access through httpd, downloaded a tar archive using wget, then unpacked the archive via tar and gzip. The intruder then executed the ptrace program using a different group identity. We later detected the intrusion by seeing the ptrace process in the process listing. We believe the ptrace process was seeking to exploit a race condition in the Linux ptrace code to gain root access. Figures 1 and 6 demonstrate BackTracker's ability to separate two intermingled attacks from a single log. Changing detection points from /bin/login to ptrace is sufficient to generate distinct dependency graphs for each attack.

Figure 7 shows the filtered dependency graph for an attack on March 2, 2003, which we named the *openssl-too attack*. The machine was used lightly by one

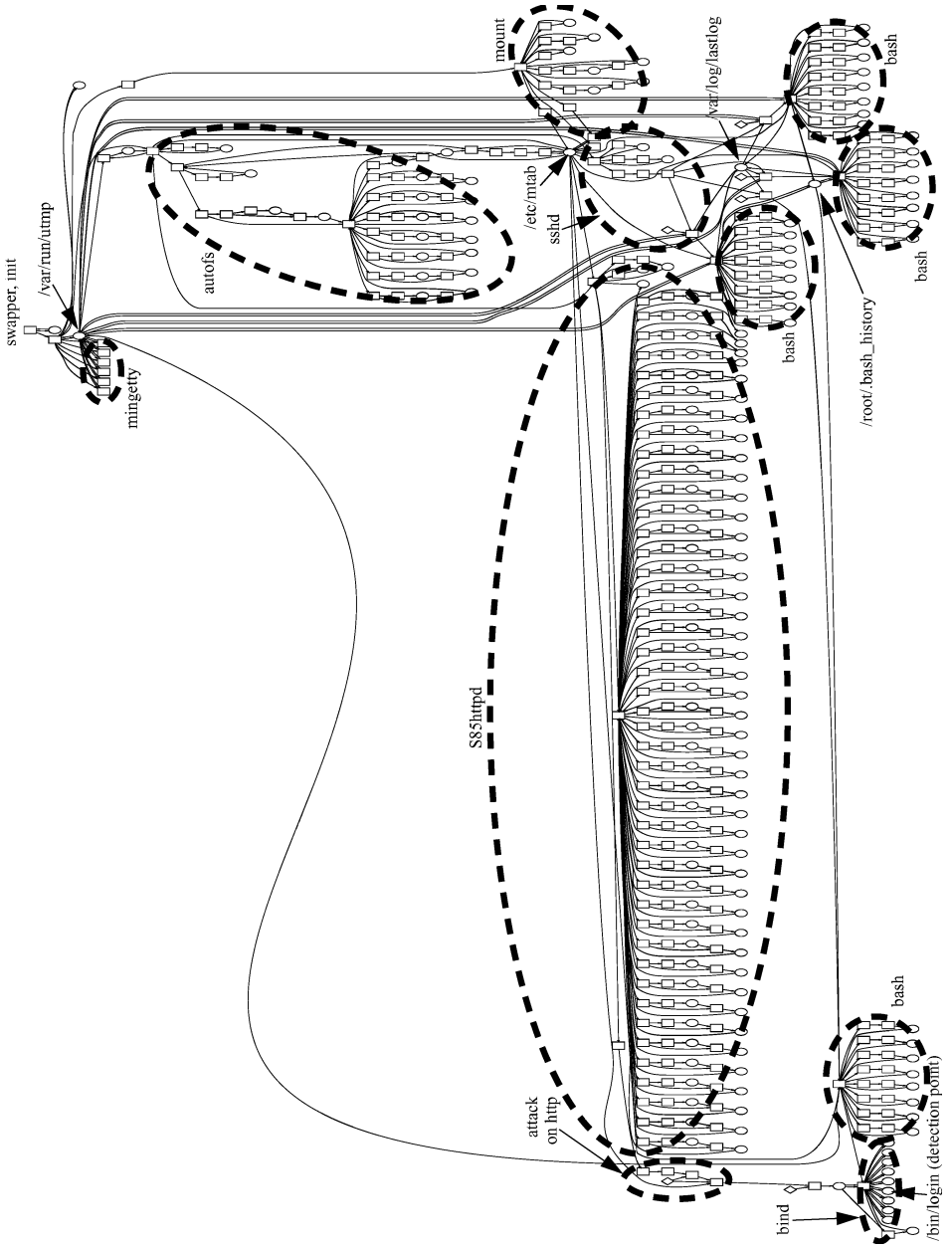


Fig. 5. Mostly unfiltered dependency graph generated by GraphGen for *bind attack*. The only filtering used was to not show files that were read but not written. The circled areas and labels identify the major portions of the graph. Of particular interest are the files we filter out in later dependency graphs: /var/run/utmp, /etc/mtab, /var/log/lastlog, /root/.bash_history. We will also filter out helper processes that take input from one process (usually via a pipe), perform a simple function on that input, then return data to the main process. Most objects associated with S85httpd are helper processes spawned by find when S85httpd starts.

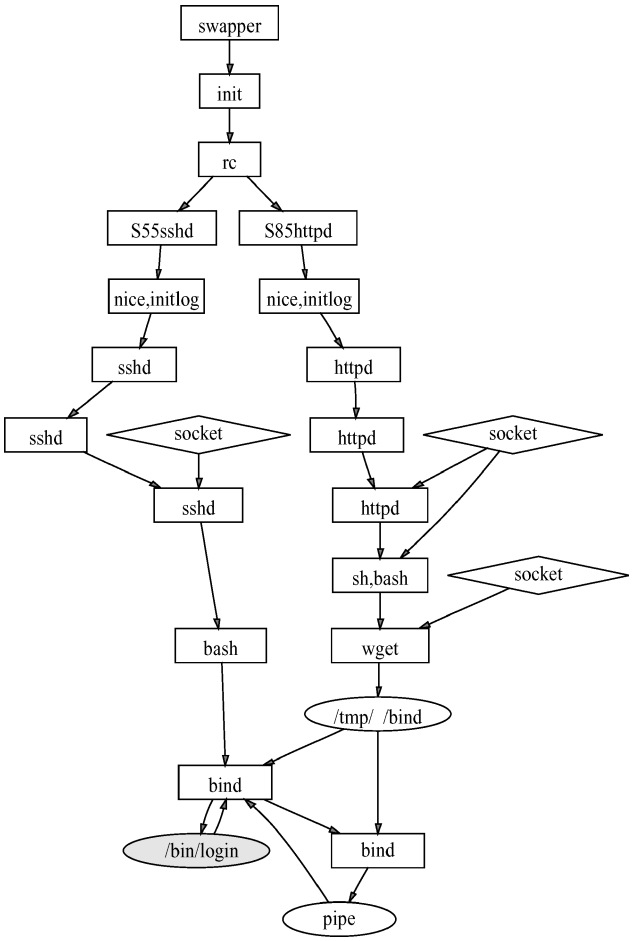


Fig. 6. Filtered dependency graph for *bind attack*.

of the authors (to check for intrusions) during the March 1–3 period covered by this log. The attacker gained access through httpd, downloaded a tar archive using wget, then installed a set of files using tar and gzip. The attacker then ran the program openssl-too, which read the configuration files that were unpacked. We detected the intrusion when the openssl-too process began scanning other machines on our network for vulnerable ports.

Another intrusion occurred on our machine on March 13, 2003. The filtered dependency graph for this attack is almost identical to the *ptrace attack*.

Figure 8(a) shows the default filtered dependency graph for an attack we conducted against our own system (*self attack*). *self attack* was more complicated than the real attacks we have been subjected to. We gained unprivileged access via httpd, then downloaded and compiled a program (sxp) that takes advantage of a local exploit against sendmail. When sxp runs, it uses objdump to find important addresses in the sendmail binary, then executes sendmail through execve to overflow an argument buffer and provide a root shell. We used this

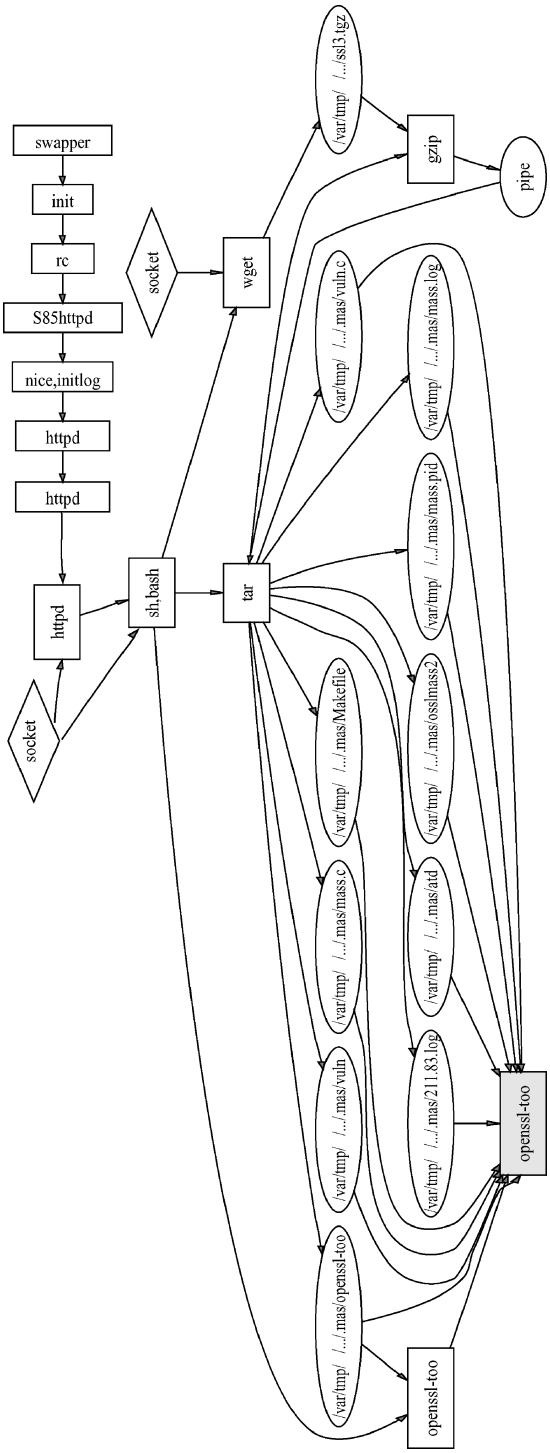


Fig. 7. Filtered dependency graph for *openssl-too attack*.

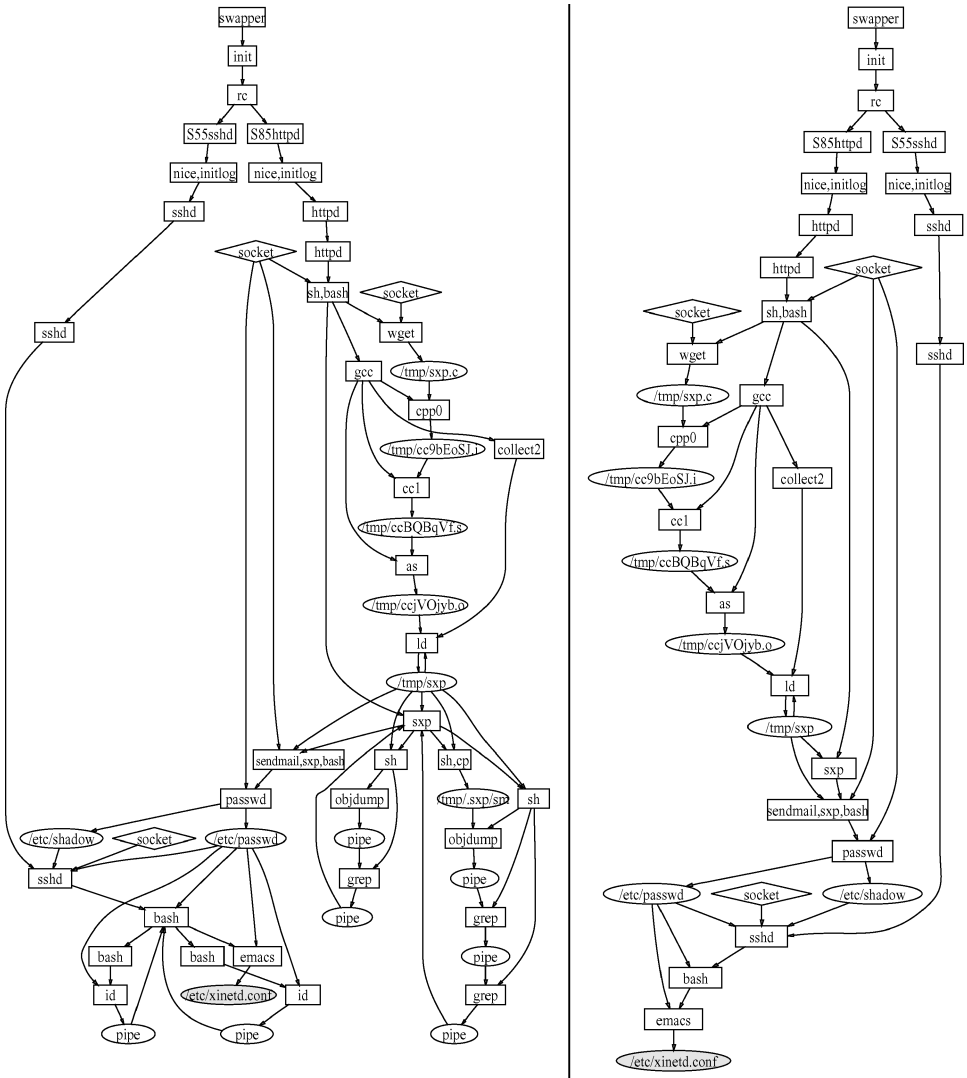


Fig. 8. Filtered dependency graph for *self attack*. Figure 8(a) shows the dependency produced by GraphGen with the same filtering rules used to generate Figures 1, 6, and 7. Figure 8(b) shows the dependency graph produced by GraphGen after adding a rule that filters out pipes. Figure 8(b) is a subgraph of Figure 8(a).

root shell to add a privileged user to the password files. Later, we logged into the machine using this new user and modify /etc/xinetd.conf. The detection point for this attack was the modified /etc/xinetd.conf.

One goal for this attack was to load the machine heavily to see if BackTracker could separate the attack events from normal events. Over the duration of the workload, we continually ran the SPECweb99 benchmark to model

the workload of a Web server. To further stress the machine, we downloaded, unpacked, and continually compiled the Linux kernel. We also logged in several times as root and read /etc/xinetd.conf. The dependency graph shows that BackTracker separated this legitimate activity from the attack.

We anticipate that administrators will run GraphGen multiple times with different filtering rules to analyze an attack. An administrator can filter out new objects and events easily by editing the configuration file from which GraphGen reads its filter rules. Figure 8(b) shows the dependency graph generated with an additional rule that filters out all pipes. While this rule may filter out some portions of the attack, it will not usually disconnect the detection point from the from an intrusion source, because pipes are inherited from a process's ancestor, and BackTracker will track back to the ancestor through process creation events. In Figure 8, filtering out pipes eliminates objdump, which is related to the attack but not critical to understanding the attack.

Next we measured the space and time overhead of EventLogger (Table I). It is nontrivial to compare running times with and without EventLogger, because real attackers do not reissue their workloads upon request. Instead we used ReVirt to replay the run with and without EventLogger and measure the difference in time. The replay system executes busy parts of the run at the same speed as the original run (within a few percent). The replay system eliminates idle periods, however, so the percentage overhead is given as a fraction of the wall-clock time of the original run (which was run without EventLogger).

For the real attacks, the system was idle for long periods of time. The average time and space overhead for EventLogger was very low for these runs because EventLogger only incurs overhead when applications are actively using the system.

The results for *self attack* represent what the time and space overheads would be like for a system that is extremely busy. In particular, serving Web pages and compiling the Linux kernel each invoke a huge number of relevant system calls. For this run, EventLogger slowed the system by 9%, and its compressed log grew at a rate of 1.2 GB/day. While this is a substantial amount of data, a modern hard disk is large enough to store this volume of log traffic for several months.

GraphGen is run after the attack (offline), so its performance is not as critical as that of EventLogger. On a 2.8-GHz Pentium 4 with 1 GB of memory, the version of GraphGen described in Section 2.3 took less than 20 s to process the logs for each of the real attacks and 3 h for the self attack. Most of this time was spent scanning through irrelevant events in the log. We implemented a version of GraphGen that stores event records in a MySQL database, which allowed GraphGen to query for events that affect specific objects and thereby skip over events that do not affect objects in the graph [Goel et al. 2003]. This technique reduced the time needed for GraphGen to process the self attack to 26 s.

6. ATTACKS AGAINST BACKTRACKER

In the prior section, we showed that BackTracker helped analyze several real attacks. In this section, we consider what an intruder can do to hide his actions

from BackTracker. An intruder may attack the layers upon which BackTracker is built, use events that BackTracker does not monitor, or hide his actions within large dependency graphs.

An intruder can try to foil BackTracker by attacking the layers upon which BackTracker's analysis or logging depend. One such layer is the guest operating system. BackTracker's analysis is accurate only if the events and data it sees have their conventional meaning. If an intruder can change the guest kernel (e.g., to cause a random system call to create processes or change files), then he can accomplish arbitrary tasks inside the guest machine without being tracked by BackTracker. Many operating systems provide interfaces that make it easy to compromise the kernel or to work around its abstractions. Loadable kernel modules and direct access to kernel memory (/dev/kmem) make it trivial to change the kernel. Direct access to physical memory (/dev/mem) and I/O devices make it easy to control applications and files without using the higher-level abstractions that BackTracker tracks. Our guest operating system disables these interfaces [Huagang 2000]. The guest operating system may also contain bugs that allow an intruder to compromise it without using standard interfaces [Ashcraft and Engler 2002]. Researchers are investigating ways to use virtual machines to make it more difficult for intruders to compromise the guest operating system, for example, by protecting the guest kernel's code and sensitive data structures [Garfinkel and Rosenblum 2003].

Another layer upon which the current implementation of BackTracker depends is the virtual-machine monitor and host operating system. Attacking these layers is considerably more difficult than attacking the guest kernel, since the virtual-machine monitor makes the trusted computing base for the host operating system much smaller than the guest kernel.

If an intruder cannot compromise a layer below BackTracker, he can still seek to stop BackTracker from analyzing the complete chain of events from the detection point to the source of the attack. The intruder can break the chain of events tracked if he can carry out one step in his sequence using *only* low-control events that BackTracker does not yet track. Section 2.4 explains why this is relatively difficult.

An intruder can also use a hidden channel to break the chain of events that BackTracker tracks. For example, an intruder can use the initial part of his attack to steal a password, send it to himself over the network, then log in later via that password. BackTracker can track from a detection point during the second login session up to the point where the intruder logged in, but it cannot link the use of the password automatically to the initial theft of the password. BackTracker depends on knowing and tracking the sequence of state changes on the system, and the intruder's memory of the stolen password is not subject to this tracking. However, BackTracker will track the attack back to the beginning of the second login session, and this will alert the administrator to a stolen password. If the administrator can identify a detection point in the first part of the attack, he can track from there to the source of the intrusion.

An intruder can also try to hide his actions by hiding them in a huge dependency graph. This is futile if the events in the dependency graph are the intruder's actions because the initial break-in phase of the attack is not

obfuscated by a huge graph after the initial phase. In addition, an intruder who executes a large number of events is more likely to be caught.

An intruder can also hide his actions by intermingling them with innocent events. GraphGen includes only those events that potentially affect the detection point, so an intruder would have to make it look as though innocent events have affected the detection point. For example, an intruder can implicate an innocent process by reading a file the innocent process has written. In the worst case, the attacker would read all recently written files before changing the detection point and thereby implicate all processes that wrote those files. As usual, security is a race between attackers and defenders. GraphGen could address this attack by filtering out file reads if they are too numerous and following the chain of events up from the process that read the files. The attacker could then implicate innocent processes in more subtle ways, etc.

Finally, an attacker can make the analysis of an intrusion more difficult by carrying out the desired sequence of steps over a long period of time. The longer the period of attack, the more log records that EventLogger and GraphGen have to store and analyze. In conclusion, there are several ways that an intruder can seek to hide his actions from BackTracker. Our goal is to analyze a substantial fraction of current attacks and to make it more difficult to launch attacks that cannot be tracked.

7. RELATED WORK

BackTracker tracks the flow of information [Denning 1976] across operating system objects and events. The most closely related work is the *Repairable File Service* [Zhu and Chiueh 2003], which also tracks the flow of information through processes and files by logging similar events. The Repairable File Service assumes an administrator has already identified the process that *started* the intrusion; it then uses the log to identify files that potentially have been contaminated by that process. In contrast, BackTracker begins with a process, file, or filename that has been affected by the intrusion, then uses the log to track back to the source of the intrusion. The two techniques are complementary: one could use backtracking to identify the source of the intrusion, then use the Repairable File Service's forward tracking to identify the files that potentially have been contaminated by the intrusion. However, we believe that an intruder can hide her actions much more easily from the forward tracking phase, for example, by simply touching all files in the system. Even without deliberately trying to hide, we believe an intruder's changes to system files will quickly cause all files and processes to be labeled as potentially contaminated. For example, if an intruder changes the password file, all users who subsequently log into the system will read this file, and all files they modify will be labeled as potentially contaminated.

In addition to the direction of tracking, BackTracker differs from the Repairable File Service in the following ways: (1) BackTracker tracks additional dependency-causing events (e.g., shared memory, mmap'ed files, pipes, and named pipes; (2) BackTracker labels and analyzes time intervals for events, which are needed to handle aggregated events such as loads/store to mmap'ed

files; and (3) BackTracker uses filter rules to highlight the most important dependencies. Perhaps most importantly, we use BackTracker to analyze real intrusions and evaluate the quality of the dependency graphs it produces for those attacks. The evaluation for the Repairable File Service has so far focused on time and space overhead—to our knowledge, the spread of contamination has been evaluated only in terms of number of processes, files, and blocks contaminated and has been performed only on a single benchmark (SPEC SDET) with a randomly chosen initial process.

Work by Ammann et al. [2002] has tracked the flow of contaminated transactions through a database and rolls data back if it has been affected directly or indirectly by contaminated transactions. The Perl programming language also tracks the flow of tainted information across perl program statements [Wall et al. 2000]. Like the Repairable File Service, both these tools track the forward flow of contaminated information rather than backtracking from a detection point to the source of the intrusion.

Program slicing is a programming language technique that identifies the statements in a program that potentially affect the values at a point of interest [Tip 1995]. Dynamic slicers compute the slice based on a specific set of inputs. BackTracker could be viewed as a dynamic program slicer on a self-modifying program, where variables are operating system objects, and program statements are dependency-causing operating system events.

Several other projects assist administrators in understanding intrusions. CERT's Incident Detection, Analysis, and Response Project (IDAR) seeks to develop a structured knowledge base of expert knowledge about attacks and to look through the post-intrusion system for signs that match an entry in the existing knowledge base [Christie 2002]. Similarly, SRI's DERBI project looks through system logs and file system state after the intrusion for clues about the intrusion [Tyson 2001]. These tools automate common investigations after an attack, such as looking for suspicious filenames, comparing file access times with login session times, and looking for suspicious entries in the password files. However, like investigations that are carried out manually, these tools are limited by the information logged by current systems. Without detailed event logs, they are unable to describe the sequence of an attack from the initial compromise to the detection point.

8. CONCLUSIONS AND FUTURE WORK

We have described a tool called *BackTracker* that helps system administrators analyze intrusions on their system. Starting from a detection point, such as a suspicious file or process, BackTracker identifies the events and objects that could have affected that detection point. The dependency graphs generated by BackTracker help an administrator find and focus on a few important objects and events to understand the intrusion. BackTracker can use several types of rules to filter out parts of the dependency graph that are unlikely to be related to the intrusion.

We used BackTracker to analyze several real attacks against computers we set up as honeypots. In each case, BackTracker was able to highlight effectively

the entry point used to gain access to the system and the sequence of steps from the entry point to the point at which we noticed the intrusion.

In the future, we plan to track more dependency-causing events, such as System V IPC, signals, and dependencies caused by file attributes. We have also implemented a tool to track dependencies forward. The combination of this tool and BackTracker will allow us to start from a single detection point, backtrack to allow an administrator to identify the source of the intrusion, then forward track to identify other objects that have been affected by the intrusion. Significant research will be needed to filter out false dependencies when tracking forward because, unlike for backward tracking, an intruder can easily cause an explosion of the dependency graph to include all files and processes.

ACKNOWLEDGMENTS

The ideas in this article were refined during discussions with George Dunlap, Murtaza Basrai, and Brian Noble. Our SOSP shepherd Frans Kaashoek and the anonymous reviewers provided valuable feedback that helped improve the quality of this article.

REFERENCES

AMMANN, P., JAJODIA, S., AND LIU, P. 2002. Recovery from malicious transactions. *IEEE Trans. Knowl. Data Eng. 14*, 5 (Sept.), 1167–1185.

ASHCRAFT, K. AND ENGLER, D. 2002. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. 131–147.

BUCHACKER, K. AND SIEH, V. 2001. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering* (HASE). 95–105.

CERT. 2000. Steps for recovering from a UNIX or NT system compromise. Tech. rep. CERT Coordination Center. Available online at `http://www.cert.org/tech_tips/win-UNIX-system_compromise.html`.

CERT. 2001. Detecting signs of intrusion. Tech. rep. CMU/SEI-SIM-009. CERT Coordination Center. Available online at `http://www.cert.org/security-improvement/modules/m09.html`.

CERT. 2002a. CERT/CC overview incident and vulnerability trends. Tech. rep. CERT Coordination Center. Available online at `http://www.cert.org/present/cert-overview-trends/`.

CERT. 2002b. Multiple vulnerabilities In OpenSSL. Tech. rep. CERT Advisory CA-2002-23. CERT Coordination Center. Available online at `http://www.cert.org/advisories/CA-2002-23.html`.

CHESWICK, B. 1992. An evening with Berferd in which a cracker is lured, endured, and studied. In *Proceedings of the Winter 1992 USENIX Technical Conference*. 163–174.

CHRISTIE, A. M. 2002. The Incident Detection, Analysis, and Response (IDAR) Project. Tech. rep. CERT Coordination Center. Available online at `http://www.cert.org/idar`.

CIAC. 2001. L-133: Sendmail debugger arbitrary code execution vulnerability. Tech. rep. Computer Incident Advisory Capability. Available online at `http://www.ciac.org/ciac/bulletins/l-133.shtml`.

DENNING, D. E. 1976. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May), 236–243.

DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M., AND CHEN, P. M. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*. 211–224.

FARMER, D. 2000. What are MACtimes? *Dr. Dobb's J. 25*, 10 (Oct.), 68, 70–74.

FARMER, D. 2001. Bring out your dead. *Dr. Dobb's J. 26*, 1 (Jan.), 104–105, 107–108.

FARMER, D. AND VENEMA, W. 2000. Forensic computer analysis: an introduction. *Dr. Dobb's J. 25*, 9 (Sept.), 70, 72–75.

FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. 1996. A sense of self for Unix processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*. 120–128.

GARFINKEL, T. AND ROSENBLUM, M. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Network and Distributed System Security Symposium* (NDSS).

GOEL, A., SHEA, M., AHUJA, S., AND CHANG FENG, W. 2003. Forensix: A robust, high-performance reconstruction system. In *Proceedings of the 2003 Symposium on Operating Systems Principles* (poster session).

GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. 1996. A secure environment for untrusted helper applications. In *Proceedings of the 1996 USENIX Security Symposium*. 1–13.

HUAGANG, X. 2000. Build a secure system with LIDS. Available online at `http://www.lids.org/document/build_lids-0.2.html`.

KIM, G. H. AND SPAFFORD, E. H. 1994. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of 1994 ACM Conference on Computer and Communications Security* (CCS). 18–29.

KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 2003. Operating system support for virtual machines. In *Proceedings of the 2003 USENIX Technical Conference*. 71–84.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the 2002 USENIX Security Symposium*.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LAMPSON, B. W. 1973. A note on the confinement problem. *Commun. ACM 16*, 10 (Oct.), 613–615.

THE HONEYNET PROJECT. 2001. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison Wesley, Reading, MA.

TIP, F. 1995. A survey of program slicing techniques. *J. Programm. Lang. 3*, 3.

TYSON, W. M. 2001. DERBI: Diagnosis, explanation and recovery from computer break-ins. Tech. rep. DARPA Project F30602-96-C-0295 Final Report. SRI International, Menlo Task, CA. Artificial Intelligence Center. Available online at `http://www.dougmoran.com/dmoran/publications.html`.

WALL, L., CHRISTIANSEN, T., AND ORWANT, J. 2000. *Programming Perl, 3rd ed*. O'Reilly & Associates, Sebastopol; CA.

ZHU, N. AND CHIUEH, T. 2003. Design, implementation, and evaluation of repairable file service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks* (DSN). 217–226.

Received October 2003; revised July 2004; accepted May 2004