Isolating Intrusions by Automatic Experiments

Stephan Neuhaus
Department of Computer Science
Saarland University
Stephan.Neuhaus@acm.org

Andreas Zeller
Department of Computer Science
Saarland University
zeller@acm.org

Abstract

When dealing with malware infections, one of the first tasks is to find the processes that were involved in the attack. We introduce Malfor, a system that isolates those processes automatically. In contrast to other methods that help analyze attacks, Malfor works by experiments*: first, we record the interaction of the system under attack; after the intrusion has been detected, we replay the recorded events in slightly different configurations to see which processes were relevant for the intrusion. This approach has three advantages over deductive approaches: first, the processes that are thus found have been experimentally shown to be relevant for the attack; second, the amount of evidence that must then be analyzed to find the attack vector is greatly reduced; and third, Malfor itself cannot make wrong deductions. In a first experiment, Malfor was able to extract the three processes responsible for an attack from 32 candidates in about six minutes.*

1. Introduction

When analyzing an attack, one of the first tasks is finding out which processes participated in the attack and how they are related. If we don't have statistical information that can help us classify and isolate malicious traffic [24, 28], this must happen before we can look for the input that caused the intrusion, the *attack or infection vector*. For example, when we analyze an infection with the Linux Slapper worm, we could arrive at this analysis [21]: "Attackers having the IP address 10.120.130.140 sent a malformed client key in an HTTPS request to our Web server. This caused a buffer overrun in the Web server and invoked the shell. The running shell then saved a uuencoded copy of the worm's source-code, decoded and compiled it, and ran the generated executable under the name *.bugtraq*. Once running, the worm tried to contact other computers on the network." In this example, the relevant processes are the Web server, the shell, the *cat*, *uudecode*, and C compiler commands, and finally the *.bugtraq* process. To find these processes, we usually start from an observed failure of the security policy (such as an unknown process named *.bugtraq*) and use log files or tools like The Coroner's Toolkit [6, 7] to reason backwards to its root cause (the malformed HTTPS request).

However, dealing exclusively with evidence after the fact severely limits even an expert's chances of reconstructing the event chain:

Completeness. The evidence might not be enough for the event chain to be reliably established.

Minimality. The relevant evidence might be buried in a host of other evidence and may thus be hard to see.

Correctness. Our reasoning (by human or machine) might be faulty, leading to wrong conclusions.

To alleviate these problems, we are exploring a novel approach with our system, called Malfor (short for MALware FORensics). In a typical scenario, Malfor would be deployed on a honeypot, capturing all attackable processes. When the honeypot is attacked, a host IDS signals a compromise and triggers Malfor's replay component. Instead of *deducing after the fact* what must have happened, we *experiment*: in order to learn which processes are necessary for the break-in to occur, we repeatedly replay slightly modified versions of the captured attack (Section 2) using a capture and replay infrastructure that enables not only verbatim replay, but replay under altered circumstances (Section 3). Malfor then finds the processes that were relevant for the attack and notifies a system administrator, who can then complete the analysis with a much smaller body of relevant evidence. This works without knowing the attack vector beforehand. Malfor can also be deployed on production systems at the price of some performance overhead. We evaluate the technique using an example and find that the extraction of three relevant processes from a total of 32 processes took about six minutes (Section 4). We review related work in Section 5, list some assumptions and limitations of Malfor in Section 6 and detail some of our future plans in Section 7.

2. Finding relevant processes

How can we efficiently find those processes that are relevant for a break-in, possibly among thousands? If we could capture the interaction of the attacked system, we could view a blow-by-blow account of it in slow motion and analyze what must have happened. This is certainly useful in order to study the attack, but is neither complete, minimal, or correct, as was indicated above. If we want to find which processes were *actually* relevant for an intrusion, we need to be able to make *experiments*. For example, if we want to check whether the *inetd* process was relevant for the attack or not, we would like to replay the attack without the *inetd* process. If the attack still succeeds, we have experimental and incontrovertible evidence that *inetd* was indeed not relevant. If the attack now fails, it must have been necessary for the attack.

If we can capture and replay the processes in a system so that we can control which processes will be executed and which will not, we want to find a *minimal process set that is necessary for the intrusion*. If there are n processes, this would take on the order of 2^n replays in the worst case, so we are willing to settle for a process subset that is small but not necessarily minimal, if we can only compute it with less replays.

This problem is solved by *Delta Debugging*, a technique that originated in automated debugging and test support. Delta debugging repeatedly runs various process subsets and uses a test function that yields ✔ (successful termination, no break-in), ✘ (the break-in occurred) or **?** (something unexpected happened). These results drive a strategy that finds a small subset of processes that make the break-in happen, but where removing any single process from that subset causes the break-in not to happen any more. It is comprehensively described in the papers by Zeller [31] and Zeller and Hildebrandt [32].

Delta debugging is like binary search: it halves the process set and tries each half separately. However, complications arise because the relevant processes need not all be in one half, which makes delta debugging somewhat more complicated than straightforward binary search. In contrast to other methods in the same general area such as slicing [27], which use deduction, delta debugging uses *experiments* to arrive at its conclusions: it *actually* tries various subsets of processes and lets the outcome of the test function drive its strategy. It does not necessarily find the smallest process subset that causes the failure, but in practice we find that results from delta debugging are close to optimal. In addition, delta debugging is not restricted to source code analysis.

Delta debugging is a practical method. It has already been used successfully to automatically find defects in programs as large as the GNU C Compiler [3]. Its worst case running time is $O(n^2)$ if there are n processes and executing a process takes unit time, but it usually finishes in $O(\log n)$ time.

Let us apply delta debugging to an example network server process to see how delta debugging can find the processes involved in the intrusion. This network server, named *Spud*[1], reads and parses a HTTP-like command set from a network socket. One of these commands will cause the file */tmp/pwned* to be created. In our evaluation, we use the existence of this file as a break-in indicator: as soon as this file has been successfully created, we say that *a break-in has happened*. Spud has the following structure which is typical of many network server programs (see also Figure 1):

1. it detaches itself from the controlling terminal, becoming a session leader;
2. the session leader opens a socket and binds it to a well-known port number;
3. it accepts a connection on that socket;
4. it forks a worker process; and
5. while the session leader continues listening, the worker reads a request from the newly opened socket, performs the requested action (possibly using subprocesses that run other programs), and exits.[2]

In this example, the intrusion is caused by a single system call, the one that creates the file */tmp/pwned*. The *set of relevant processes* then contains the process making that system call, and its ancestors.

Assume that we have 32 processes: the command-line program C, the session leader S and thirty workers W_1, …, W_{30}, where W_{20} executes the intrusion-causing system call. Delta debugging will try different subsets of the set of all processes $\{C, S, W_1, \ldots, W_{30}\}$ and test whether the intrusion still happens. In our example, the set of relevant processes would be $\{C, S, W_{20}\}$.

The actual sequence of process subsets tried by delta debugging is shown in Table 1. The column marked “R” contains the result of the test: ✘ if the intrusion occurs, and ✔ if it does not occur. The other columns contain a bullet if the corresponding process is included in the test. For example, in row 1, processes C, S, and W_1 through W_{14} are included, but W_{15} through W_{30} are not. Since W_{20} is the culprit, and since it is not executed, the intrusion does not happen and the output is ✔.

1 After a character from *Trainspotting* [29].

2 Spud does not use any subprocesses.

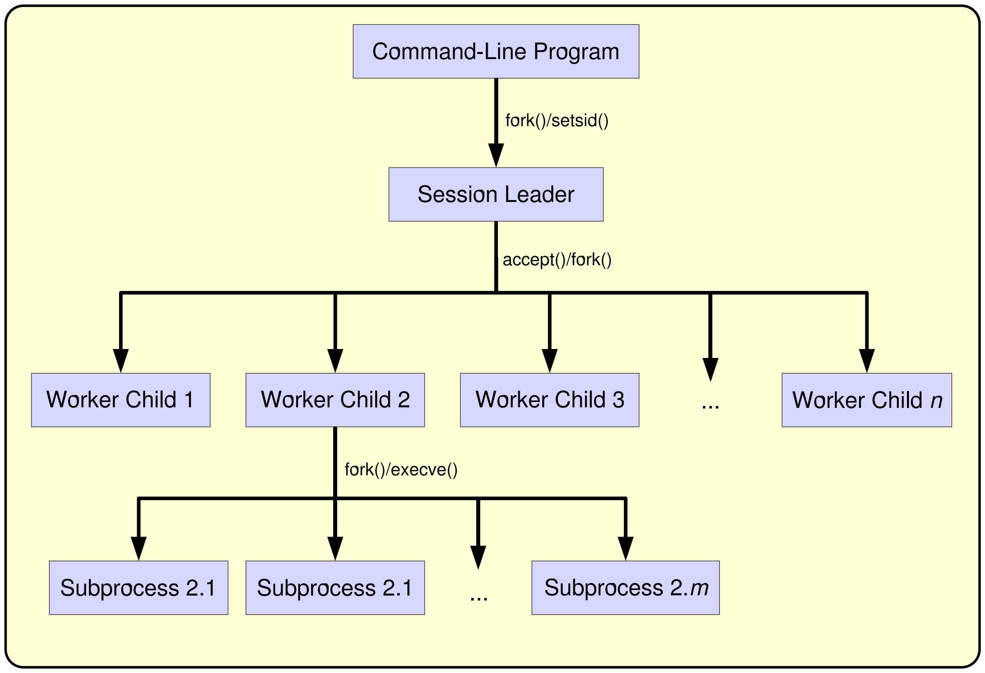

Figure 1. Typical process structure of a network server.

Lines 1–3 try to find a subset of the original processes that produce ✘, first by splitting the original set in roughly equal parts and then subdividing it further and trying complements when that does not work. Line 14 contains the minimal subset needed for the intrusion. Further subsets need not be considered, because they have already been tested (lines 9 and 13). Delta debugging finds the culprits (processes C, S, and W_{20}) with only fourteen tries.

From this, we can see that in this example, Malfor's result is complete, minimal and correct: the processes that Malfor found, and *only* those, really *were* relevant for the break-in.

We emphasize that Malfor's result does not only contain the root cause of the attack, but all intermediate attack-relevant processes too. So if an attack involves a long chain of events, Malfor will produce all the intermediate steps that are needed to reproduce the attack.

One concern is that a process could not exhibit its original behavior during replay because it took a different control path. For example, what if a process launches an attack only upon the existence of certain files, or a successful challenge-response authentication with a remote server? In these cases, the process must have made system calls that caused these actions to be performed. Malfor then captures these system calls and replays them. For example, if a process creates a random challenge as part of the challenge-response protocol, it will have to issue system calls to do so (for example, in order to read */dev/random*). When we replay the process, we also replay these system calls, so we will have recreated the state of the process as it was when it made the original challenge-response authentication and the computed challenge will be the same in both cases. In the case of files on the local file system, Malfor actually executes the system calls; in the case of a remote challenge-response authentication, it replays a previously recorded conversation.

3. Capture and replay

In this section, we give some technical details on our capture and replay infrastructure. In particular, we will focus on how we can alter captured events on replay and how we can test specific process subsets.

3.1. Overview

Security incidents happen because intruders send malicious inputs (attack vectors) to *processes*, which then issue *system calls* that cause some security policy to be violated [22]. For example, if we assume that there are no covert channels and that confidential data was disclosed, some process must have issued a *write*() system call that wrote this data to its forbidden destination. These malicious inputs are delivered to the processes also by system calls. Therefore, if we want to use delta debugging to find out

#	Processes																																R
	C	S	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●																	✓
2	●	●	●	●	●	●	●	●																									✓
3	●	●	●	●	●	●	●	●									●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	✗
4	●	●	●	●	●	●	●	●																	●	●	●	●	●	●	●	●	✓
5	●	●	●	●	●	●	●	●									●	●	●	●	●	●	●	●									✗
6	●	●	●	●																													✓
7	●	●	●	●													●	●	●	●	●	●	●	●									✗
8	●	●	●	●																	●	●	●	●									✗
9	●	●																															✓
10	●	●																			●	●	●	●									✗
11	●	●																					●	●									✓
12	●	●																			●	●											✗
13	●																																✓
14	●	●																				●											✗

Table 1. Process subsets actually tried by delta debugging Spud with Malfor.

how a policy violation happened, we must be able to capture and replay the system calls that those processes make.

In order to capture and replay system calls, Malfor uses a subsystem that sits between a process under observation and the operating system (see Figure 2). When capturing, all interesting system calls are intercepted by that subsystem and captured in a database (steps 1, 2, 2a, 3, and 4a). When replaying, the requested system call is matched in the database, modified, and returned; the original system call is never executed (steps 1, 2, 3, and 4b).[3]

Since a process has no reliable method of finding out whether a system call actually executed or whether it was replayed from a database *except by making a system call*, we play with the process's notion of the outside world. This part of Malfor is therefore named Solipsy[4]. Since capturing takes place on a real system, and since system calls are faithfully replayed, an attacking program cannot easily find out whether it is being fed replayed information or results from actual system calls.

Our implementation of Solipsy runs on Linux. In order to speed up replay, we use User Mode Linux (UML) [4] with copy-on-write (COW) disk images. We initially create two disk images, one for capture and one for replay. One contains the capture daemon and the other the replay daemon, but both are otherwise identical. To start replaying, we boot UML from the replay disk image. During operation, the COW file contains only those blocks that have changed with respect to the original disk image, so in order to reset the replay system to its initial state, we can simply delete the COW file and reboot UML.

3 Some system calls are problematic or impossible to replay, among them *fork*() or *execve*() (these change the control flow in a way that is difficult if not impossible to replay without actually creating a new process or executing a new program, respectively); *brk*() (on success, memory must actually be allocated); and *mmap*(), *shmat*() and related ones (they allow a process to do I/O without extra system calls). All these system calls are not replayed, but actually executed.

4 From solipsism, the idea that the outside world comes to us only through our senses and is therefore not necessarily real.

3.2. Matching system calls

When we want to replay a system call, we have to find its matching counterpart in the database. If replay were perfect, then everything during replay would be the same as it was during capture: system call parameters, file descriptor numbers, and process IDs would all be unchanged. In practice, this is not the case. For example, process IDs during capture will generally differ from those during replay because of un-replayed processes, and file descriptor numbers will differ because some file operations are replayed from the database (bypassing the operating system) and some are executed (going through it). These parameters are mapped between the user-mode process, the operating system, and the replay daemon.

Some system call parameters may be judged irrelevant for matching because they can change from run to run without affecting the mapping of captured to replayed calls. An example is the exact value of a buffer address in a *read*() system call. While irrelevant parameters are ignored, all others are used as a key into the database.

When a system call is thus matched and retrieved, it is marked as "spent" and cannot be replayed again. This is needed to distinguish between otherwise identical system calls: a process can call *read*() repeatedly with the exact same parameters. In this case, we take the earliest unspent match.

3.3. Signals

Signals are events that are asynchronously delivered to a process. In other words, a signal is delivered to a process without the process having to ask for it. This is in contrast to system calls, where processes must explicitly request their services. When a signal arrives, process execution is suspended and a special routine in the process, called a signal handler, is executed. Signals may occur at any time during process execution, even while executing a signal handler. Typical examples are SIGSEGV, which occurs when

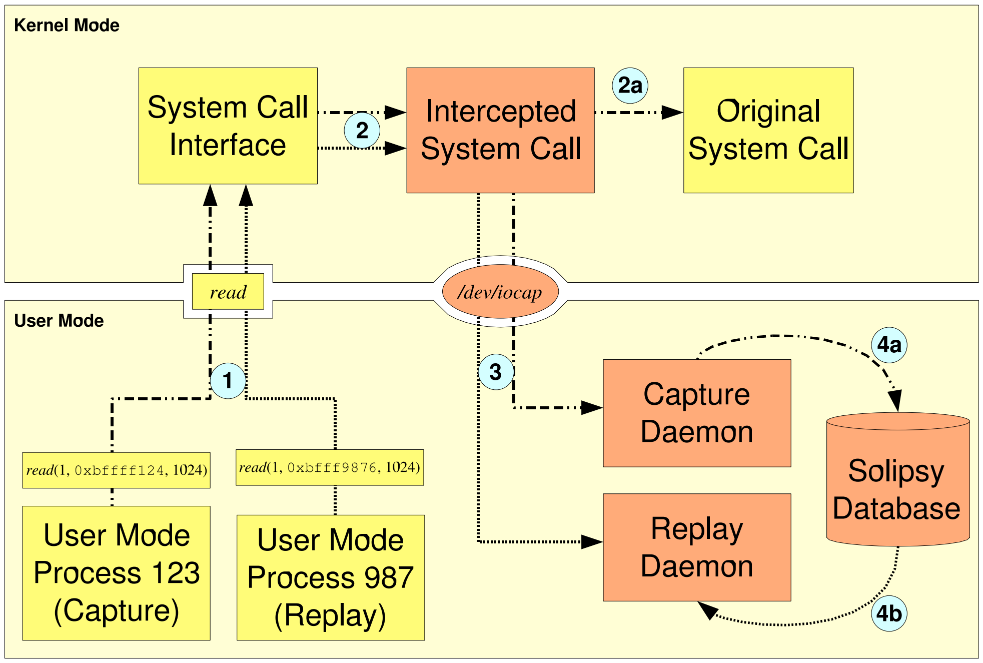

Figure 2. Architecture of Solipsy. Parts of Solipsy are shown darker.

a process references memory that it cannot access, such as a null pointer; SIGCHLD, which occurs when a child process dies; SIGALRM, which occurs when a timer expires; or SIGINT, which occurs when a user interrupts a process from the keyboard.

For externally-generated signals such as SIGINT, it is certainly possible to capture and to replay them at the exact time they occurred during capturing [5]. However, our method works by altering the process's control flow in order to find out which circumstances are relevant. Therefore, replay with Malfor is not verbatim, and it becomes impossible to say when or if a signal should be delivered. For this reason, Solipsy currently does not capture or replay signals.

Fortunately, many important signals will be delivered correctly, even if they are not captured or replayed. For example, SIGSEGV, SIGCHLD and SIGALRM will be delivered at the correct time anyway, because they have their origin directly in a process's former actions: accessing invalid memory, creating a child process which then exited, or installing a timer, respectively.

3.4. Changing a process's execution path

Malfor works by testing various subsets of processes to see if the intrusion happens when only these processes are present. However, Malfor cannot prevent a process from calling *fork*(), so if we want to test a process subset that does not contain a process P, but P's parent forks, we let it fork, but as soon as P makes a system call, we make the process exit instead of replaying the call.

To support this behavior, the replay daemon registers event handlers and will forward system calls to these handlers which then decide what to do with the system call and with the calling process:

- the system call can be matched in the database (the default action);
- the system call can be executed by the operating system (the default action if the call cannot be found in the database);
- the process can be made to exit instead of executing the system call;
- if the system call is *fork*(), the newly created *child process* can be made to exit at the next call; or
- the result of the system call can be computed on the fly.

This framework is more general than is strictly needed to control process creation, but it allows us to extend Malfor to find not only the relevant processes, but also the relevant *inputs* to those processes (see Section 7 on further work).

4. Initial experience

If Malfor is to be a practical system, it needs to be accurate, fast and easily deployed. Malfor's accuracy is cur-

What	Total	Median	μ	σ
UML + Replay	$364\,s$	$26\,s$	$26.0\,s$	$3.7\,s$
Replay only	$174\,s$	$13\,s$	$12.4\,s$	$3.3\,s$

Table 2. Performance when analyzing the sample attack on Spud (14 tests). All times are in seconds. The column labeled μ holds the mean and the column labeled σ holds the standard deviation.

rently being evaluated. All we can say at this point is that it has so far found the relevant processes in all our tests. The next two sections contain a preliminary performance evaluation, which has results only for capturing and for delta debugging the example from the previous section. We can show only a few results here; we are currently working on a more complete set of statistics. The last section looks at Malfor's deployability.

For these experiments, the (un-tuned) MySQL database, UML with Solipsy and the outside "attacker" were all on the same host, a 3 GHz Pentium 4 PC running Linux 2.4, both as the host kernel and the UML kernel. All kernels were otherwise unoccupied.

4.1. Performance of delta debugging

When we actually analyzed the example attack from Section 2 with Malfor, we got the results summarized in Table 2. We can see that the time spent replaying the processes is on average only about half of the UML running time. In other words, about half the time is spent booting and shutting down Linux kernels. Linux startup time is hard to speed up; in our case, we have already disabled all unneeded services. Shutting down a UML, however, takes about ten to eleven seconds in our setup, so if we just killed the UML instead of shutting it down cleanly, we would save between $14 \cdot 10\,s = 140\,s$ and $14 \cdot 11\,s = 154\,s$ of run time. If we did that, the proportion of replay time to total running time would rise to about 75 percent and the total running time itself would decrease by about 40 percent, to about $217\,s$.

4.2. Performance of capturing

In order to measure the performance of capturing, we ran Spud in successively more complete Solipsy environments, as explained below. In each environment, we called Spud 257 times in rapid succession. One of these times, the service was made to exhibit its vulnerability. Overall, we therefore have one command-line process, one session leader, one intrusion-causing interaction and 256 harmless interactions. Table 3 has our results.

Environment	Time	OH Ded.	OH UML
Dedicated machine	$21.5\,s$	0%	
UML w/o Solipsy	$22.4\,s$	4%	0%
UML w/Solipsy, disabled	$24.0\,s$	12%	7%
UML w/full Solipsy	$24.3\,s$	13%	8%

Table 3. Performance of Spud in various environments. The column labeled "OH Ded." has the overhead of running Spud in the given environment relative to running it on a dedicated machine; the column labeled "OH UML" has the same overhead relative to running Spud on a UML without Solipsy.

On a dedicated system that did not run inside UML or use Solipsy (that is, Spud performed only steps 1 and 2a in Figure 2), this took $21.5\,s$. On an UML system that did not have Solipsy (steps 1 and 2a are performed, but the network I/O has to cross a machine boundary), it took $22.4\,s$. Once Solipsy was loaded and enabled, but the service not traced (steps 1, 2, and 2a), execution time rose to $24.0\,s$. When the vulnerable service was also traced and the results put in a database (steps 1, 2, 2a, 3, and 4a), execution time was $24.3\,s$.

While these preliminary results cannot be definitive, we feel that the system calls captured in this experiment (see Table 4) are typical of larger systems and that therefore the numbers obtained in this experiment are representative. If that is indeed the case, the overhead of capturing would be about 8% when compared to an un-traced process running inside UML (the "Overhead UML" column Table 3), or about 13% when compared to a dedicated machine without either UML or Solipsy (the "Overhead Dedicated" column).

On the one hand, both results are excellent. They also compare well with those by Dunlap and others [5]. On the other hand, it seems as if these numbers are so good only because the program takes so long, even on a dedicated machine (the first row in Table 3). Its performance is about six requests per second, which seems rather slow. It remains to be seen whether the performance figures are indeed representative of larger network services, such as Apache.

4.3. Deployability

All of Malfor's components are easily installed: Solipsy is a loadable kernel module, the capture and replay daemons are ordinary processes, the delta debugger is also an ordinary process that can additionally reside on a remote machine, and the database is an ordinary MySQL database

accept	access	bind	brk	close
connect	execve	exit	fcntl64	fork
fstat64	listen	llseek	mmap	munmap
open	read	setsockopt	socket	stat64
unlink	wait4	write		

Table 4. List of captured system calls in the experiment. Solipsy captures many more system calls than given in this table; they just weren't used by Spud.

without any tuning. Neither the kernel image nor the captured processes need to be changed. The latter is particularly important if we want to analyze processes whose programs we cannot debug. We therefore believe that Malfor is easily deployed.

5. Related work

There are a number of tools that enable deterministic replay debugging. One of the first proposals for a deterministic replay debugging system was Flight Data Recorder [30]. Flight Data Recorder is geared at replaying an entire multiprocessor system faithfully. It works by checkpointing the system and recording race-relevant information that would be needed to allow faithful replay from the last checkpoint. To record this information, it needs some extra hardware. Flight Data Recorder itself does not replay; this is left to other programs like InstantReplay [13].

Flashback [26] was motivated by the need for a rollback facility to allow debugging large and very long-running programs that might display a bug only after days of execution, or only in specific circumstances. It works by providing facilities for *checkpointing* and *deterministic replay*. Checkpointing is done by using *shadow processes*: a copy of the currently running process is created at some specific time using *fork*(), and suspended immediately. This shadow process is then reanimated when the execution is rolled back to the creation point. Replay is done by hijacking the system call handler, substituting one's own, and capturing system calls and their side effects. Solipsy uses essentially the same technique. Signals are also handled as in Solipsy, that is, they are ignored.

Another system for deterministic replay debugging is BugNet [16]. BugNet's goal is to record enough information to replay the instructions immediately preceding a program crash. BugNet aims only to replay the user code and shared libraries; the user will not be able to see what goes on during interrupts or system calls. BugNet works by saving the processor registers at the beginning of a checkpoint and then capturing the value of load instructions to registers. On replay, the initial processor state is restored and load instructions are replayed. Like Flight Data Recorder, BugNet needs hardware support.

Dunlap and others developed ReVirt, an addition to User Mode Linux that captures and replays the sequence of machine instructions during an execution of Linux [5]. ReVirt works by virtualizing the processes to be logged—wrapping them in a virtual machine—and logging asynchronous events to guarantee instruction-level replay. The novel idea here is that it is not the host machine that is being logged, but the virtual machine. This obviates the need for hardware extensions that systems like Flight Data Recorder or BugNet need.

All these systems were designed to facilitate or enable deterministic replay, that is, replaying the exact sequence of instructions that were executed previously. Malfor makes no claim of determinism. In fact, the whole point of Malfor is to change the execution flow of the participating processes to see whether the intrusion still occurs. When the replayed process's control flow changes, the whole notion of "deterministic" replay becomes problematic because the changed run cannot be compared to any previous run.

System call capturing or interposition has been in use for some time to enable security analyses or policy enforcement [9, 10]. System call interposition in the face of hostile applications is difficult and most research in that area is aimed at overcoming these difficulties [8]. The system that is most similar to Solipsy is probably *Systrace* [22], a system that helps formulate and enforce system call policies, which also modifies system calls on the fly.

King and others used ReVirt to develop Backtracker, a system that builds a graph of dependencies between events such as process creation, I/O, or file deletion and uses these dependencies for break-in analysis [12]. For example, if Backtracker wants to find the cause for a modification of the file */etc/passwd*, it will look through its dependence graph, find the process P that last modified the file and will then recursively find all events and processes that could have influenced P.

An inherent limitation of Backtracker is that it cannot find effects whose cause does not appear in the dependency graph. In our above example, if Backtracker wants to find a modification to */etc/passwd*, it will look for system calls that open and write the password file. We have written a program that creates a new root account in the password file without leaving traces in system calls. This program works as follows:

1. It loads a purpose-built kernel module.[5]

5 This presupposes that the attacker has already gained root access. However, this program's purpose is not to attack, but to install a back door that cannot be analyzed by Backtracker.

2. Upon module initialization, the kernel module causes a new root account to be added to the password file. It does so by calling functions inside the kernel and not by making additional system calls.

3. Finally, the program unloads the kernel module.

Backtracker will not be able to answer the question "Which events caused the extra root account to appear in the password file?" because it looks at the system calls and deduces that there is no relationship between */etc/passwd* and the process that loads the kernel module. Malfor will *actually* execute the processes in question and will be able to produce a fairly small list of processes that have experimentally been found to be responsible for the alteration of the password file. This limitation exists for all purely deductive methods and is not specific to Backtracker.

We can also modify the attack so that the kernel module delays installing the new root account. This means that systems are fooled that freeze the system as soon as they detect a compromise in order to find the responsible processes.

James Newsome and others have developed Taintcheck [20]. Taintcheck and produces attack signatures for certain common exploits such as buffer overflows or format string vulnerabilities by tainting all user-supplied input and following it through the computation with the help of Valgrind [18]. Taintcheck looks for potentially harmful uses of user-supplied input, whereas Malfor looks for the causes of specific events, regardless of whether they are based on harmful uses of input.

If there is much malicious traffic, such as in a worm attack, this traffic may be characterized and characteristic features exploited so that it may be possible to extract attack vectors without deduction or experimentation. Examples of such systems are Autograph [11], EarlyBird, [24] and PAYL [28]. If the attack is not only massive, but also polymorphic, Polygraph can be used [19]. However, since these systems are statistical in nature, they all need large amounts of attack traffic in order to work. In contrast to this, Malfor finds the relevant processes in a single targeted attack.

Another system to detect targeted attacks is shadow honeypots, introduced by Anagnostakis and others [1]. Traffic that is classified as anomalous is processed separately on a *shadow honeypot*, in order to see whether it leads to an attack. If it does, the traffic is discarded. If it does not, the traffic is transparently forwarded to the production system. This approach will work well for attacks that move the system from a secure to an insecure state in a short amount of time. Attacks where the system is gradually and gently nudged to an insecure state through multiple stages of attack are more difficult to handle because each stage except the last one might escape detection. These attacks are at least in principle analyzable by Malfor.

Sidiroglou and Keromytis introduce a technique that patches a vulnerable program's source code on the fly, once the infection vector is isolated [23]. They focus on stack-based buffer overflows and use heuristics that transform the program's source code so that the buffer overflow is contained. The patched program is tested in a clean-room environment, both against the original attack vector and against a local test suite to ensure that it is no longer vulnerable and that it still works as expected. Their technique presupposes that the attack vector has already been isolated, something that Malfor is designed to do.

6. Assumptions and limitations

Malfor works only under certain assumptions. If these assumptions don't hold, Malfor can be defeated. This section lists some of these assumptions.

Determinism. Malfor will have problems analyzing attacks that use race conditions or other forms of non-determinism to succeed. In principle, context switches could be added to the set of debuggable items, and delta debugging has even been used to find failure-inducing thread schedules [2], but it would be impractical to implement this in Malfor, since Malfor is very much geared towards replaying system calls without modifying the operating system or any applications.

Focus on Processes. If an attack succeeds because of a bug in a shared library, Malfor will only find the processes that were involved in this particular attack, not the library that is the real culprit. We hope, however, that Malfor's diagnosis enables administrators to analyze the attack further and ultimately to find the bug in the library.

Suitable Test. Malfor uses an automated test to check for an intrusion. If this test is fuzzy or produces false positives or negatives, Malfor's diagnosis can be faulty. This can happen for example if the attack has a delayed effect. This can be avoided by letting malfor replay all processes prior to testing subsets. If Malfor does not detect an attack immediately after replaying all processes, the test is not suitable for Malfor and needs to be adjusted.

State Equivalence. For our prototype, we cannot formally prove that system states are equal during capture and replay, because there exists no formal specification for Linux's behavior. In practice, it is also unrealistic to insist on bit-by-bit identity. We settle instead for a form of isomorphism between system states, plus mappings that make that isomorphism (hopefully) undetectable to user-mode processes. Finding out the system calls that need to be replayed and finding out the right mappings is a manual process that is imperfect by nature (because it relies on our understanding of the semantics of Linux system calls), but we believe that such isomorphisms can be created, given enough resources.

Undetectablilty. At the moment, Malfor can be circumvented once a process has gained sufficient privileges. This means that a process could deliberately perform differently during replay than during capturing. Integrating Malfor more tightly with the kernel and using mandatory access controls like SELinux [17] could mitigate this problem; these measures would leave Malfor visible, but the attacker could not easily find out whether it is in the capture or the replay phase.

7. Conclusion and further work

We have introduced Malfor, a system that uses a new experiment-based approach to analyze security incidents. Malfor produces results that are complete, minimal, and correct because the processes it finds—and *only* these processes—have been experimentally shown to be relevant. The amount of information that needs to be examined in order to find the attack vector is greatly reduced, so the relevant evidence has a much higher visibility. Attacking processes cannot easily distinguish between their attack and a replayed version.

In the future, we plan to extend Malfor in several ways. These extensions include:

Finding attack vectors. Most importantly, we are already working on applying the same simple technique to find the relevant *inputs*, that is, the infection vector. In fact, one of the original applications of delta debugging was to minimize inputs to failing test cases [32]. This could lead to the automatic generation of a signature for a NIDS like snort [25], or to a vector that can be used by Sidiroglou and Keromytis's patch generation system [23].

Using intrusion-causing and harmless runs together. Currently, we use delta debugging only one run—the run that causes the intrusion. We could also use it with two runs: one that causes the intrusion, and another similar one that does not. With these two runs, we can find a relevant *difference* between two similar runs. Delta debugging is much faster for this case than using only one run. Harmless runs are easy to come by for network services like Spud or Apache because the session leader forks many similar workers, most of which will not be causing intrusions.

Suggesting fixes in configurations. If we have two systems, one of which is vulnerable to an attack and another that is not, we can use delta debugging to find a relevant difference in the two systems' *configurations*. Which configuration files are read is apparent from the process's system calls. In fact, this information is already extracted from the system calls based on heuristics. This could lead to an automated "quick fix" feature that suggests to apply a minimal set of changes to the vulnerable system in order to make it immune to a specific attack.

Analyzing distributed attacks. At the moment, we analyze break-ins that happen on a single computer. Many important systems today are distributed, however, and incident analysis on distributed systems is a relevant problem. Apart from the problem of synchronizing event streams between machines [14], this work could also make use of results derived by Mattern and others [15] to find events that cannot be the cause of the break-in because they happen concurrently with it. It is easier to find concurrent events in a distributed system than on a single machine, so it could turn out that finding causes of break-ins is easier in distributed systems.

Our broad vision is that of a *self-diagnosing and self-healing system*: computers detect when they are under attack, use Malfor to find the attack vectors and possibly even fixes, apply the fixes and deploy the fixed components. In the race between attackers and administrators, this should give the administrators some breathing space in which they can fortify their systems and devise and deploy more general defenses.

Information about Malfor can be obtained from `http://www.st.cs.uni-sb.de/malfor/`.

8. Acknowledgements

Thanks to Silvia Breu, Holger Cleve, Valentin Dallmeier, Christian Lindig and Thomas Zimmermann for their valuable comments on earlier revisions of this paper. Valentin Dallmeier and Thomas Zimmermann also gave good advice on some of the database implementation details in Solipsy. The anonymous reviewers made many helpful comments. Thanks are also due to Michael Mirold for the name Solipsy and for valuable early discussions about Solipsy's architecture.

References

[1] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th Usenix Security Symposium*, Berkeley, CA, USA, Aug. 2005. Usenix Association.

[2] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and*

Analysis, pages 210–220, New York, NY, USA, 2002. ACM Press.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, New York, NY, USA, May 2005. ACM Press.

[4] J. Dike. The user-mode linux kernel homepage. `http://user-mode-linux.sourceforge.net/`, Jan. 2005.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, New York, NY, USA, Dec. 2002. ACM Press.

[6] D. Farmer. Frequently asked questions about the coroner's toolkit. `http://www.fish.com/tct/FAQ.html`, Jan. 2005.

[7] D. Farmer and V. Venema. *Forensic Discovery*. Addison-Wesley, Jan. 2005.

[8] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, pages 19–34, Reston, VA, USA, Feb. 2003. Internet Society.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th Usenix Security Symposium*, Berkeley, CA, USA, July 1996. Usenix Association.

[10] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, pages 19–34, Reston, VA, USA, Feb. 2000. Internet Society.

[11] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium*, Berkeley, CA, USA, Aug. 2004. Usenix Association.

[12] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 223–236, New York, NY, USA, 2003. ACM Press.

[13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[14] F. Mattern. Logical time. In J. U. P. Dasgupta, editor, *Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, 1999.

[15] F. Mattern and R. Schwarz. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[16] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the International Symposium on Computer Architecture 2005*, New York, NY, USA, June 2005. ACM Press.

[17] National Security Agency. Security-Enhanced Linux homepage. `http://www.nsa.gov/selinux/`.

[18] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, Nov. 2004.

[19] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Internet Society Symposium on Network and Distributed System Security*, Reston, VA, USA, Feb. 2005. Internet Society.

[21] F. Perriot and P. Szor. An analysis of the slapper worm exploit. `http://securityresponse.symantec.com/avcenter/reference/analysis.slappe%r.worm.pdf`, 2002.

[22] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*, pages 257–272, Berkeley, CA, USA, Aug. 2003. Usenix Association.

[23] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 2005. To appear.

[24] S. Singh, C. Estan, G. Varghese, and S. Savage. The Early-Bird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, Aug. 2003.

[25] Sourcefire, Inc. Snort homepage. `http://www.snort.org/`, June 2005.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2005 Usenix Annual Technical Conference*, Berkeley, CA, USA, 2005. Usenix Association.

[27] F. Tip. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

[28] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection, LNCS 3224*. Springer, Sept. 2004.

[29] I. Welsh. *Trainspotting*. W. W. Norton & Company, June 1996.

[30] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, New York, NY, USA, 2003. ACM Press.

[31] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 1–10, New York, NY, USA, Nov. 2002. ACM Press.

[32] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 26(2):183–200, Feb. 2002.