Isolating Intrusions by Automatic Experiments

Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes
Stephan.Neuhaus@acm.org

April 13, 2006

1 Introduction

The analysis of security incidents remains one of the most taxing things a computer scientist can do. Why is there no automated support for this task? We think this is so because existing tools use an inadequate methodology.

Intrusion analysis aims at reconstructing the break-in based on the current state of the system. To this end, we analyze traces and then deduce what must have happened inside the system so that these traces appear the way they do. For example, an analysis of the Linux Slapper worm could look like this: "Attackers with the IP address 10.120.130.140 sent a specially crafted HTTP request to our web server, which contained a malformed client key. This caused a buffer overflow and called a shell. This shell then saved a uuencode-encoded copy of the work source code, decoded and compiled it, and started the resulting program under the name *.bugtraq*. As soon as the program ran, it tried to contact other hosts i the network." (Example taken from [5].)

An investigator analyzing this intrusion will probably first see the rogue *.bugtraq* process and will then try to isolate those processes that were responsible for the attack. This holds for processes that are still running (such as the web server) and processes that have already terminated (such as the *uudecode* process).

The usual method is to begin with the violation of the security policy (the *.bugtraq* process) and then work backwards using tools like The Coroner's Toolkit [2] to the root cause (the malformed HTTPS request). This deductive approach has a number of serious drawbacks:

Completeness. The traces may not be sufficient in order to deduce the cause-effect chain reliably.

Minimality. Important traces are often buried in a large number of irrelevant traces and need to be laboriously extraced.

Correctness. Our proofs could base on wrong assumptions which may invalidate our deductions.

We have developed a tool called Malfor (short for MALware FORensics) which avoids these drawbacks by using *experimental* methods. Instead of interpreting traces and deducing a cause-effect chain backwards, Malfor works experimentally: in a first phase, Malfor *captures* events (processes in pur case) as the system is running. As soon as a break-in is detected, Malfor uses these events to *partially replay* the system. By cleverly choosing which events to repeat, we isolate those events that are reevant for the break-in: if we repeat the system without process X and if the break-in still occurs, process X cannot have been relevant for the attack.

2 Capture and Replay

Malfor's subsystem for capture and replay works by System Call Interposition. In this method, system calls like *fork*, *execve*, *read*, *getpid* and so on are diverted to Malfor's own routines. These execute the original routines and upload the system calls' parameters and results to a database. In security research, this method has been used in Systrace [6] in order to create on-the-fly security policies for system calls.

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/696

Malfor must take care of many details when replaying system calls, because otherwise replay will not work. For example, processes may have a different process ID during replay than it had during capturing. Still, the process must see its original PID so that library calls that use the PID (such as *gethostbyname*) still work as expected when replayed.

Our method works by replaying captured processes in ever different configurations. For this it is necessary that Malfor be able to suppress a process's execution. But on one hand, you can't force a parent process *not* to call *fork*. On the other hand, process creation must not simply fail because this would be too strong a difference with respect to the original run. Our solution is to create the child process, but to terminate it again at the next syscall.

These measures are typical when one wants to repeat only parts of a system.

3 Minimization

In order to find the responsible processes among all captured processes, we use Delta Debugging [3]. Delta Debugging is a technique that uses repeated experiments to minimize *any* set of failure-inducing circumstances.

Delta debugging works like binary search: first, we try with one half of all circumstances removed. If that reproduces the failure, we continue with this reduced set of circumstances. If not, however, we try by removing the other half. If that doesn't work either, we try the complements ofour subsets. If that doesn't work either, we split the original set into more than two parts and try again.

Zeller and others have shown that the final result contains only circumstances that are relevant for the failure. If there are initially n circumstances, delta debugging will need at most $O(n^2)$ tests to minimize them.

4 First Experiences

In order to test our prototype, we have witten a network server that contains a security hole: once it receives a specially prepared request, it creates a file */tmp/pwned* with administrator privileges. In a simulated attack, we have hidden one malicious request among twenty-nine others.

This run caused about 1,500 system calls, which were executed and captured by the original system in about 6 seconds. This is a performance overhead of about 8% with respect to the throughput without capturing. Capturing takes place in a virtual machine in order to simplify replay. Takting that into account as well, the overhead rises to 13% with respect to a dedicated machine. These penalties compare favourably with other research [1] and make Malfor suitable for production environments.

Malfor used about three minutes and 14 tests to isolate all relevant processes (three of 32) [5]. Replay was slower than capturing by a factor of about two. These numbers emphasize Malfor's suitability for production use.

5 Further Work

We first want to extend Malfor to a realistic example. We have already prepared an attack on Apache which adds another root account to the password file without opening the password file for reading. This attack is constructed especially to fool tools like BackTracker which analyze attacks by constructing relationships between system calls [1, 4]. This attack never opens the password file; yet it is modified afterwards.

The next task is to extend Malfor to distributed systems. Malfor is already designed to be used in such environments, but replaying needs to observe certain constraints so that the consistency of the entire system is preserved.

6 Conclusion

We have introduced Malfor, a system that uses experimental methods to analyse intrusions automatically. It can be used on production systems and is especially suitable for the analysis of targeted attacks.

Literatur

[1] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementati-*

on, pages 211–224, New York, NY, USA, December 2002. ACM Press.

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

[3] Ralf Hildebrandt and Andreas Zeller. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 26(2):183–200, February 2002.

[4] Samuel T. King and Peter 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.

[5] Stephan Neuhaus and Andreas Zeller. Isolating intrusions by automatic experiments. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, pages 71–80, Reston, VA, USA, February 2006. Internet Society, Internet Society.

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