Towards an Approach for Automatically Repairing Compromised Network Systems

Julian B. Grizzard, Sven Krasser, Henry L. Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250, USA
{*grizzard, sven, owen*}@*ece.gatech.edu*

Gregory J. Conti, Eric R. Dodson
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, USA
{*conti, edodson*}@*cc.gatech.edu*

Abstract

The widely accepted method to repair a compromised system is to wipe the system clean and reinstall. We think that there may be alternative methods. Specifically, we envision systems that are capable of automatically recovering from system compromises. Our proposed approach is a repair agent that resides in an isolated area on the system. We use a virtual machine approach to isolate the repair agent. The repair agent should roll back any undesirable changes, determine the point of entry, and prevent further compromise.

1. Introduction

Conventional wisdom states that once a system has been compromised, it can no longer be trusted. An attacker that has compromised a system can modify any state within the system, and so it is difficult to determine the extent to which the attacker has modified the system. Further conventional wisdom states that the only option to re-establish trust in a compromised system is to completely reinstall the operating system from known good media.

We propose that systems can be built in such a way that they automatically repair themselves after a system compromise has occurred. We call systems that are capable of automatically repairing themselves *self-healing* systems. The entire mechanism that is responsible for repairing the system is called the *repair agent*. In addition to repairing the system, the repair agent must re-establish trust in the compromised system such that the user can again *trust* the system.

Re-establishing trust in a compromised system is a difficult problem. One important reason for this is because traditional systems lack a true trusted computing base. In order to incontestably re-establish trust in a compromised system, an indisputable foundation of trust is needed. Our solution to this problem is a trusted immutable kernel extension (TIKE). TIKE can be used as a safe haven for the self-repair agent.

In this work, we use a virtual machine approach to establish a core foundation of trust as a proof of concept. The *host* operating system runs directly on the physical machine and is considered to be TIKE. The *guest* operating system runs on virtual hardware and is considered the *untrusted system*. The guest operating system accesses the physical hardware via proxy calls through the host operating system.

We discuss how self-healing systems can be built into the TIKE framework so that systems can automatically recover from compromises. Furthermore, we provide an overview of the details for such systems. There are many challenges to an approach for self-healing systems, and in this work we begin to address these challenges.

2. Motivation

With the proliferation of exploits targeted to today's computer systems, an attacker has the ability to compromise a vast number of systems. Once an attacker has compromised a system, he or she will want to retain access to that system even if the original security hole is patched. In order to retain access to a compromised system, the attacker will often install a *rootkit* onto the target system [1]. A rootkit will add a backdoor that can be used to reenter the system at a later time. As an example of the real-world threat, we set up a Red Hat 6.2 system on the Georgia Tech honeynet [2], and within a matter of days an attacker had compromised the box and installed a rootkit on the system.

We propose an approach for automatically repairing compromised systems even if a rootkit has been installed on the system. There are three important questions that come up when assessing the validity of our approach:

- Why repair rather than prevent?
  The complexity of computer systems continues to increase. As the complexity increases, it becomes more

difficult to prevent software or human errors that lead to system compromises. It may not ever be possible to completely prevent system compromises, so we need to explore methods for dealing with system compromises.

- Why repair rather than reinstall?
  In some instances it may be economically more efficient to repair part of a system than to reinstall the entire system. If the repair can be automated, then advantages may include minimizing down time, minimizing compromise damage, and minimizing administrative overhead.

- Why should methods other than reinstallation be explored?
  It is commonly accepted that once a system has been compromised there is only one solution: wipe the system clean and perform a fresh install. We think that alternative methods should be explored in order to determine if other methods can offer advantages over complete reinstallation.

3. Design Principles and Architecture

We propose a repair agent that exists as part of a production system. We limit the design of the agent to two types of *production* computer systems: *servers* and *clusters of workstations*. Given these two types of production systems, we present five design principles for designing the repair agent. These principles are *simplicity*, *isolation*, *trust*, *visibility*, and *adaptation*.

1. Simplicity – The repair agent must be designed to be as simple as possible. As the repair agent grows in complexity, it is difficult to attest that the agent itself is correct.

2. Isolation – The repair agent must be isolated from the production operating system. It must not be possible to alter, disable, or otherwise bypass the self-repairing system.

3. Trust – The repair agent must be trusted. There should be assurance that the agent is correct and operates as expected.

4. Visibility – The system must have complete visibility of the production system. A system compromise may alter any state within the system, so the repair agent must be capable of restoring any state to a trusted state.

5. Adaptation – The repair agent must consume minimal resources when the production system is under normal operation. When an attack occurs, the repair agent can consume as much of the computer system's resources as necessary in order to repair any damage caused by the attack.

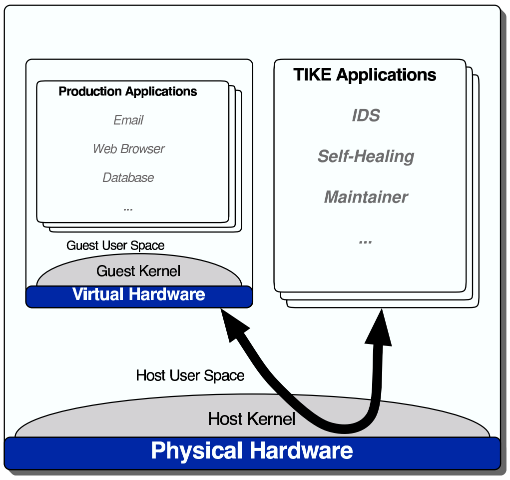

Figure 1. Overview of architecture

Building on these five principles we describe a system architecture that is capable of autonomous self-repair. Figure 1 shows an overview of the self-repairing architecture. The core mechanism in the architecture is TIKE. There are many architectures for TIKE, and Figure 1 shows one approach, which is a virtual machine approach. The virtual machine approach attempts to address the five design principles. However, the virtual machine approach may not be the best approach in terms of performance. We will look at other architectures in future work such as a microkernel approach or a hardware approach.

Building on this core mechanism, the architecture consists of a scheduler that couples an intrusion detection system with a self-repairing mechanism. A discussion of the operation is described below, followed by the details of the different components of the architecture.

3.1. System Operation

The design of our proposed self-healing system requires initial setup prior to bringing the system online. First, the repair agent is installed on the computer system. Then, the production operating system is installed on the system. Next, the repair agent establishes a known good baseline for the production system and initializes its various components including the scheduler, the intrusion detection system (IDS), the self-repair mechanism, and the maintainer. Now the system can be brought online, and any legitimate updates to the system will be corroborated with the maintainer.

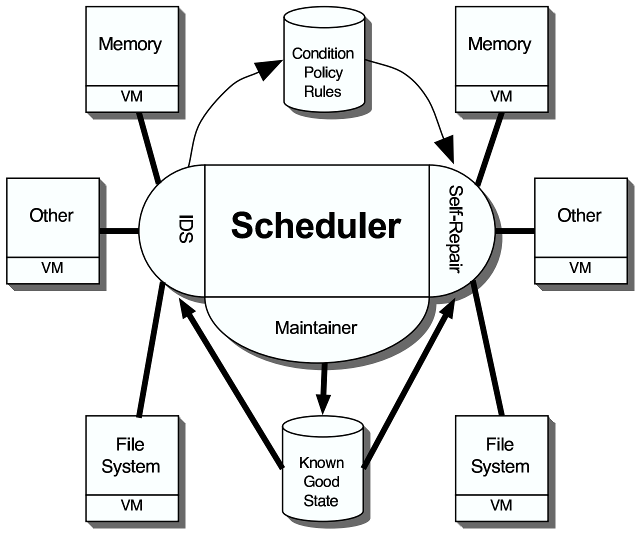

Figure 2. Architecture of repair agent

When the power is turned on, the repair agent takes control of the computer and initializes itself. After the repair agent has been initialized, it verifies the integrity of the production system and then begins booting it. After the production system boots, the repair agent enters into its cyclic algorithm to ensure stability and integrity of the system. First the system is put into mode one; the scheduler grants a given number of CPU cycles to the production system. After those CPU cycles are exhausted, the scheduler puts the system into mode two. In mode two, the IDS checks the validity of the system and the maintainer checks for legitimate updates to the system. If a compromise is detected, the scheduler puts the system into mode three. The self-healing mechanism repairs the system and then hands control back to the scheduler, which puts the system back into mode one. If no compromise is detected, the scheduler puts the system back into mode one.

3.2. Trusted Immutable Kernel Extension

The core mechanism for the repair agent is TIKE. The repair agent resides within TIKE. TIKE is an enabling architecture that serves as a safe-haven for intrusion detection and self-repair [3].

3.3. Repair Agent

The mechanism that automatically repairs compromised systems is the repair agent. The repair agent consists of a scheduler, an IDS, a self-repairing mechanism, and a maintainer as seen in Figure 2.

3.3.1. Scheduler

The scheduler sits at the center of the repair agent. It controls when and which components run on the CPU. To enable an adaptive repair agent, the scheduler exports an application programming interface (API) to increase or decrease priority to the various components. The IDS is responsible for maintaining the adaptive nature of the repair agent. If intrusions are detected, the system alert level is increased. If intrusions are not detected for a period of time, the system alert level is decreased. The production system has no visibility of the scheduler.

3.3.2. IDS

The IDS is responsible for scanning the production system for compromises. If a compromise has occurred, the IDS builds a report and sends it to the self-repair mechanism through the condition policy rules (CPR) database. The CPR database contains the rules that determine what action the self-repair mechanism will take based on the given condition. The IDS is responsible for monitoring three areas of state: memory, file system, and other. Memory monitoring includes processes, open ports, kernel execution code, and so forth. The file system includes the files on the hard disk. Other state includes such things as external hardware, CPU registers, and so forth.

3.3.3. Self-Repairing Mechanism

The self-repairing mechanism is a simple component in the architecture. It operates on compromise events and performs any action necessary to repair the compromise event based on the CPR. The self-repairing mechanism is not scheduled unless there are outstanding compromise events that need to be serviced.

3.3.4. Maintainer

The maintainer is responsible for keeping a copy of the known good system state up to date. More specifically, the maintainer maintains the trusted hash table. If the system state changes due to an upgrade, then the maintainer will update the corresponding hash entry in the trusted hash table.

4. Further Approach Details

There are a number of details that need to be further explored. These include:

- System Validity (Attestation/Hashing/Reattestation)
  Our approach to determining system validity is based on hashing and attestation. When a system is com-

promised, the repair agent repairs the system and then can attest for the validity of the system. We call this *reattestation*.

- State (Memory/File System/Other)
  The three types of state that the repair agent is responsible for monitoring are the memory, the file system(s), and any other type of state.

- Root Access
  Even if the attacker gains root access to the production operating system, the repair agent should remain inaccessible. We use a virtual machine approach to meet this need.

- Originating Entry Point and Patch
  King and Chen introduced a framework for backtracking intrusions called BackTracker in [4]. BackTracker is capable of finding the entry point of an attack. Their work may be applicable in our system in order to find the originating entry point and prevent further attacks.

- Adaptation and Performance
  The system should adapt to increasing threats from the attacker, but when the threat level is low, the system should have high performance.

- Denial of Service
  The repair agent should minimize the denial of service experienced by the user after a system compromise has occurred.

5. Approach Limitations

There are many limitations and challenges to our approach that must be addressed. One of the biggest challenges is how to recognize that a system has been compromised. Intrusion detection is known to suffer from false positives and false negatives. Our approach moves toward meeting this challenge by having signatures for known good state with attestation and hashing as opposed to relying on signatures for known bad state.

Two limitations of our approach that need to be addresses are the physical access threat and data compromise. Our assumptions assume the attacker will be attacking from a remote location. However, if the attacker has physical access to the box, he or she will be able to bypass our system. Our approach could be enhanced and moved towards protection against physical access by implementing the TIKE architecture in hardware. The problem of data compromise is that once an attacker has compromised a system, he or she may be able to gain access to sensitive data. Our approach does not solve this problem completely, but we think that the system will be able to minimize the amount of damage done by automatically discovering the compromise and taking action immediately rather than relying on slower manual methods.

There are a few other limitations to our approach. We have described a system in which everything should be attested by a trusted party. In theory the mechanism is sound, but in reality it may be challenging. Another point to address is that some performance will lost; however, the trade-off in performance may be acceptable given the increased security. Finally, one assumption that our approach makes is that the repair agent itself will not be compromised. This is an important assumption as the entire system relies upon it. We base this assumption on the premise that the repair agent will be much less complex than the operating system it is monitoring, and so it should be easier to verify the correctness of a small repair agent than an entire operating system.

6. Conclusions

We have discussed an approach to building systems that are capable of automatically recovering from a system compromise. The feasibility of building such systems is difficult to determine. We do not think we have yet covered all aspects for the requirements of such a system, but our approach is a step in that direction. Since our computer systems continue to increase in complexity and security is becoming more important, we think that approaches to recoverable systems should continue to be explored.

We have built part of the system described in our work. Future work will include adding more to our prototype to better understand the feasibility of self-healing systems. We will also test our prototype on the Georgia Tech honeynet in the form of self-healing honeypots. Based on our experience gained, we will then refine, add to, and draw more conclusions from our approach.

References

[1] J. G. Levine, J. B. Grizzard, and H. L. Owen, "A methodology to characterize kernel level rootkit exploits that overwrite the system call table," in *Proceedings of IEEE SoutheastCon*. IEEE, March 2004, pp. 25–31.

[2] (2004, June) Georgia Tech honeynet research project. http://users.ece.gatech.edu/~owen/Research/HoneyNet/HoneyNet_home.htm.

[3] J. B. Grizzard, E. R. Dodson, G. J. Conti, J. G. Levine, and H. L. Owen, "Towards a trusted immutable kernel extension (TIKE) for self-healing systems: a virtual machine approach," in *Proceedings of Fifth IEEE Information Assurance Workshop*. IEEE, June 2004, pp. 444–446.

[4] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, 2003, pp. 223–236.