Leveraging Virtual Machine Introspection for Hot-Hardening of Arbitrary Cloud-User Applications

Sebastian Biedermann Stefan Katzenbeisser
Security Engineering Group, Technische Universität Darmstadt
{biedermann, katzenbeisser}@seceng.informatik.tu-darmstadt.de

Jakub Szefer
Computer Architecture and Security Laboratory, Yale University
jakub.szefer@yale.edu

Abstract

Correctly applying security settings of various different applications is a time-consuming and in some cases a very difficult task. Moreover, with explosion in cloud computing popularity, cloud users are able to download and run pre-packaged virtual appliances. Many users may assume that these come with correct security settings and never bother to check or update these settings. In this paper we propose an architecture that can automatically and transparently improve security settings of arbitrary network applications in a cloud computing setup. Users can deploy virtual machines with different applications, and our system will attempt to find and test better security settings tailored towards their specific setup. We call this approach "hot-hardening" since our techniques are applied to running applications.

1 Introduction

The proposed architecture improves cloud computing security and aids cloud computing users. In our scenario, we assume a cloud computing provider that offers Infrastructure-as-a-Service (IaaS) to its customers on which virtual machines (VMs) can be deployed by each cloud customer who has full administrative privileges over this VM and who can install or modify arbitrary applications. An example for such a cloud provider is the Amazon Elastic Compute Cloud (EC2)[1] that provides different pre-packaged virtual instances for each user who is willing to pay. Our work is inspired by the Netflix Chaos Monkey project[2], where different VMs are automatically and randomly shut down by autonomous agents to test resiliency of a cloud environment. In our architecture, however, we never shut down VMs.

Instead, we propose a Security Monkey which is an autonomous agent running in a separate VM. A Security Monkey randomly identifies other user-VMs and automatically attempts to improve their security settings – and eventually the whole cloud environment would be improved as a Security Monkey can periodically visit and check all user-VMs. With the help of our architecture, the security of user-VMs will be improved as a Security Monkey will automatically work within the backend network of a cloud computing environment in order to find user-VMs running applications of which the security settings are set too "low" and improve these settings.

In particular, automatically improving security configuration settings of known applications can be straightforward, because the meaning of the available settings are known as well as their possible ranges and their effects on the application. However, in this paper, we present our ongoing work in building-up an architecture that can automatically improve the configuration settings of arbitrary unknown applications. This goal introduces several difficulties, first to identify if and which kind of configuration settings are actually available and what kind of effect their modification has on the running application's characteristics.

Our proposed methods and techniques are generic, which means the settings of unknown applications can be improved and no previous knowledge about target applications is required. In our architecture, we only use transparent techniques. Accordingly, the work-flow of a target user-VM is not disrupted nor there is need to install software components of our architecture on a user-VM. Since the work-flow of the applications is not interrupted and there is no need to terminate or shutdown any applications, we call our approach "hot-hardening". To the best of our knowledge, this work is the first approach which tries to automatically improve security settings of unknown running applications by leveraging virtualization technologies without the need for any user interaction.

The remainder of the paper is organized as follows: Section 2 presents related work and Section 3 explains our architecture. Section 4 evaluates the architecture, Section 5 points out future work and Section 6 concludes.

2 Related Work

In this chapter, we present related work in the field of "hot-patching", of analysis of applications during their run-time and of Virtual Machine Introspection (VMI).

2.1 Hot-Patching

Hot-patching is the process of upgrading a running program by modifying its binary code while it executes. This way, new functionality can be added to closed-source applications, software updates can be deployed or security vulnerabilities can be fixed without rebooting. Hot-patching is very useful in order to reduce the downtime in systems which require a high availability but which need to be patched (e.g. important security patches). Ramaswamy et al. [16] proposed to use Patch Objects, a special encoding form for patches as ELF relocatable objects which can be applied more reliable. They presented their techniques as an extension of the Application Binary Interface (ABI). Huang et al. [12] proposed a hot-patching framework that autonomously patches the binary code by learning the possible causes of failures and they demonstrated the feasibility on Web-based applications. Payer et al. [13] described a hot-patching architecture by integrating dynamic patches with the help of a virtualized sandbox based on dynamic binary translation. Their evaluation showed that their system can hot-patch 45 of 49 web server bugs while adding only a low execution overhead.

In our work, we propose an approach which we call "hot-hardening". Instead of fixing binary code, we fix configuration settings.

2.2 Run-time Security Analysis

Analyzing the security characteristics of applications during run-time can become a costly task and proposed techniques are usually semi-automated. Often, the proposed approaches intervene once an errors occurred. Whitaker et al [19] proposed the Chronus tool, an architecture that automatically diagnoses configuration errors and helps to find the point of failure. This is achieved by searching across time for the point the system went into a failed state with the help of a virtual machine. Rabkin et al [15] proposed an architecture that uses static analysis in order to extract a list of configuration options for a target program. Their analysis could find 95% of configuration options and infer a type for the most options, however, their techniques only work on Java source code. Xu et al. [20] built a tool called Spex which automatically infers configuration requirements from software source code in order to detect misconfiguration. Especially, automated adaptions of the settings of firewall applications are desirable. Al-Shaer et al. [1] presented a framework for automatic testing of firewall configurations using policy and traffic generation to test successful deployment of new rules.

In our architecture, we automatically identify potential configuration settings of running applications.

2.3 Virtual Machine Introspection

Virtual Machine Introspection (VMI) is the technique of locating and accessing the memory of a running VM (usually a user-VM) from another isolated running VM (usually an admin-VM) which is co-located on the same hardware and which has required privileges to access the hypervisor layer. VMI is transparent and does not interrupt the work-flow of the target user-VM nor can it be detected from there. VMI is especially of interest for security-related techniques, e.g. intrusion detection, since it offers the opportunity to built security tools which are tamper-resistant ([10],[14],[4]). Fraser et al. [9] developed self-repairing immunity architecture running on an isolated VM that can rapidly restore processes in a user-VM which have been infected and modified by malware. Srivastava et al. [17] proposed an tamper-resistant application layer firewall based on VMI techniques.

Besides the advantages of VMI, there are also some difficulties, like the need for precise knowledge about the structure of the target software setup of the user-VM in order to close the so called "semantic-gap" ([8],[7]). This is required to know locations of useful information on the target user-VM's memory. Recently, there have been architectures proposed which do not only read the memory of a co-resident user-VM but also write to specific locations. This way, for example processes can be implanted into a user VM during run-time [11].

In our proposed architecture, we use fine-grained VMI techniques to locate, read and write to potential configuration settings of running applications.

3 Architecture of the Security Monkeys

As cloud computing deployments grow larger and lager, we assume it will be unlikely that all VMs can be inspected and improved at once. We propose a Security Monkey as being a movable VM having administrative privileges and working on one user-VM after another. The Security Monkey uses different techniques which are introduced in this section and illustrated in Figure 1.

The Security Monkey can be live migrated [5] to a new hardware node and this way relocated to a new pool of user-VMs. Depending on the size of the cloud, a troop of multiple Monkeys can run and improve the security settings of a continuously changing pool of arbitrary user-

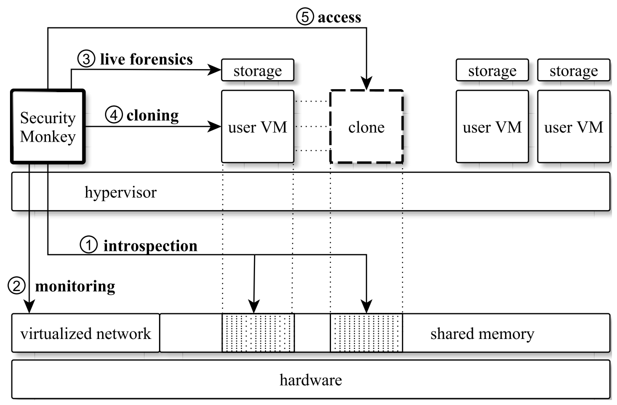

Figure 1: A Security Monkey working on a user-VM and its clone using different transparent techniques.

VMs. Once a Security Monkey is migrated to a new node it uses fine-grained transparent VMI techniques [6] on a selected user-VM's memory to discover the IP address of the operating system, to detect running applications and their virtual address ranges ① and it scans for corresponding open ports ②. Since we do not have any previous knowledge about any of the applications, the used approaches are fully generic. This enables a Security Monkey to improve settings even of unknown applications.

A Security Monkey uses a *Setting Discovery* strategy (Section 3.2) to identify available potential configuration settings for each found application and tries to locate these settings in the running application's memory. In this strategy, VMI and live forensic techniques ③ are used in order to transparently investigate specific memory regions of the target application and transparently examine disk sectors of corresponding target files on the raw storage of the user-VM.

Only if potential configuration settings could be identified and located, the Security Monkey triggers live-cloning ④ of the user-VM [18] with on-the-fly memory modifications. During the copy process of this procedure, the clone's memory is modified so that it allows a Security Monkey to log in ⑤ as an administrator [3]. A clone of the user-VM is required since actively changing settings of an application running on the original user-VM could lead to interruptions and crashes as well as it would cause misleading entries in the log files. The storage of the user-VM is not cloned, in this case we rely on a logical volume manager and only add a copy-on-write virtual snapshot of the original storage to the clone.

In further steps, a Security Monkey uses a *Setting Improvement* strategy (Section 3.3) in which it applies VMI techniques on the clone's memory to set new well chosen configuration settings for an application and to evaluate if these new settings improved the target application's security characteristics with the help of accessing the clone and executing test-runs. This strategy is continuously repeated until better settings could be found, VMI is finally used to deploy the new configuration settings on the target running application on the original user-VM.

3.1 Assumptions

The proposed architecture can improve the settings of arbitrary and unknown applications. However, a few assumptions about an application *App* need to be made which allow the functioning of our Security Monkeys:

- *App* is a running Linux process and its virtual memory space is located within a user-VM's memory.
- *App* maintains at least one open TCP or UDP port for network communication with which a Security Monkey can establish a connection.
- *App* has at least one configuration file located on the user-VM's storage.

These assumptions are very generic and usually met by arbitrary Linux applications which use networking.

3.2 The Setting Discovery Strategy

In this section we describe in detail how a Security Monkey can identify and locate configuration values of even an unknown application on the original user-VM.

3.2.1 Identification of Potential Configuration Files

According to our assumptions, a configuration file should exist on the user-VM's storage for each application. Since we do not know which and even if configuration values for the application exists, and we also do not know in which sectors the configuration file is stored or how it is named, a Security Monkey needs to use a generic strategy. To find files which are somehow related to the application, we can not use the application's open file descriptors, because configuration files are usually only read once during start-up. First, the Monkey investigates the application's memory regions in which the executable (ELF) is stored as well as its allocated heap and stack memory for short strings that appear to be file names having a path. Subsequently, the Monkey examines the user-VM's raw storage with live forensics techniques to verify which files do exist and to find other files located under these paths. This way, the Monkey creates a list of several existing files which are somehow related to the application's functionality. With the help of a blacklist, the Monkey removes certain known files (e.g. '/etc/passwd') from the list which finally results in a list of potential configuration files.

3.2.2 Pattern Generation and Value Localization

In the next steps, the Security Monkey reads the corresponding sectors of each identified potential configuration file on the raw storage and analyzes the text-content line-by-line. In multiple runs, it splits each line with different separators (e.g. tabs), removes blank content and tries to convert each resulting token into an integer value. If values could be found in the file, the Monkey combines them to a value union U_n of n values because we assume that an application's configuration setting is usually stored in memory combined in a struct or the locations of these values are at least close-by.

Finally, identified values in each potential configuration file can also lead to a value union U_n. In order to locate a value union U_n in the application's memory, a Monkey needs to generate patterns (regular expressions) for each U_n for which it can search. Based on experiments, we decided that a pattern needs to have at least three values to be characteristic and at most seven values to ensure a search process being feasible. If U_n has less than three values, U_n is discarded. If U_n contains more than seven values, the Monkey selects the seven most characteristic values.

Since the Monkey does not know the order of U_n in memory and it can not be sure that U_n occurs in the same order like it occurred in the file, the Monkey needs to generate $\sum_{i=3}^{n}(\frac{n!}{(n-i)!})$ combinations of patterns for each U_n. This can result in a minimum of 6 and a maximum of 13650 different patterns. Finally, the Monkey searches for these patterns in the heap and the stack memory regions of the application using VMI.

3.3 The Setting Improvement Strategy

In the most cases, an application has a configuration setting which is not necessarily the optimal setting for this application, especially regarding to its security characteristics. In this section, we describe our ongoing work in how a Security Monkey can automatically improve an unknown application's configuration setting once the potential values could be identified and located in the running application's memory.

Since the Security Monkey does not know the meaning of the located configuration settings and if these settings are somehow related to security characteristics in general, it executes multiple-test runs while it receives feedback about the application's characteristics in each test-run (Figure 2). The Monkey repetitively replaces a located value union with a new value union in the clone's memory via VMI①and actively triggers the running application②in order to receive and analyze the feedback ③. The generated feedback is analyzed regarding to defined security measures.

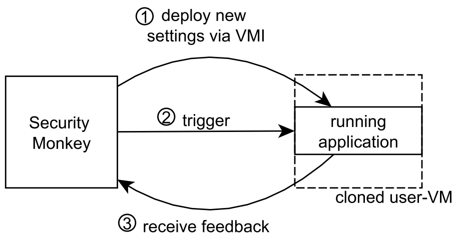

Figure 2: A Security Monkey deploying settings to an application running inside a cloned user-VM and analyzing the feedback after triggering.

A Security Monkey can retrieve measurements from different points on the clone and interpret the feedback in order to estimate the current security characteristics of the unknown application to which new settings were set. To take these measurements, the Monkey can actively trigger the application on the network, on the system level and it can also terminate and restart it. Depending on the received feedback, a new value union for the next test-run is selected. In particular, the Monkey selects new values depending on the properties of the initially located values (e.g. bigger differences for large initial values or only powers of two for values which initially were powers of two). Most of the new settings will not increase the measurement and some settings may cause the application to terminate or to crash. In these cases, the Security Monkey deploys and tests other settings and it also maintains a dump of the clone for rapid recovery.

4 Evaluation

For an implementation, we used a modified the Xen hypervisor [2] which offered live migration [5] and live cloning with account injection [3]. A Security Monkey is an admin VM equipped with our Python scripts using an introspection library[3] and a forensic framework[4]. In an evaluation, we used a Quad CPU with 2.67GHz and 4GB memory and a user-VM with Ubuntu Linux 12.04 and 1024MB memory. We installed 8 popular Linux network applications on the user-VM.

In a first step, the Security Monkey analyzed the user-VM and retrieves its IP address, a list of open ports and information about all running network applications (8) via VMI. On average (10 runs), this could be performed in 5.4 ± 0.1s. In the next steps, each detected application is analyzed in detail. Figure 3 shows the timings for the *Setting Discovery* strategy applied on each application.

On average (10 runs), the *Setting Discovery* strategy required 13 ± 5s for an application. Most of the time was needed to find the allocated virtual address ranges of the specific memory regions (ELF, heap, stack) and

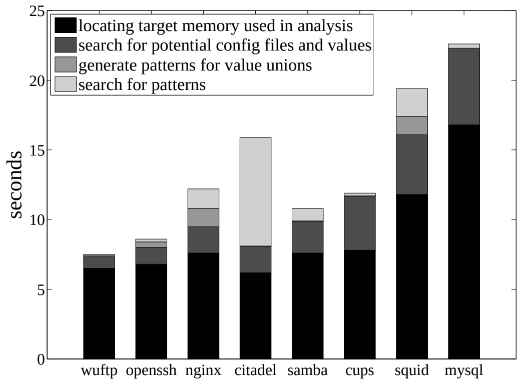

Figure 3: Timings required to analyze each application.

to retrieve the content. Finding potential configuration files by searching for paths in the memory and locating them on the raw storage required up to 5s. Extracting value unions and generating patterns could be performed in up to 1s, while searching for the patterns depends on the memory content of the application and could take up to 7s. Table 1 shows more results in detail.

For each application, multiple potential configuration files could be automatically identified and in each case the real configuration file was below them which finally lead to the automatic identification of real configuration values of the unknown applications and the generation of useful patterns.

	potential configs	**found values (union(s))**	**patterns**	**found at least one pattern**
wuftp	6	3 (1)	6	yes
openssh	10	6 (1)	1920	yes
nginx	12	7 (2)	13650	yes
citadel	11	5 (1)	300	yes
samba	6	4 (1)	48	no
cups	11	5 (1)	300	no
squid	5	7 (1)	13650	yes
mysql	7	5 (1)	300	yes

Table 1: *Setting Discovery* details for each application.

Finally, for 6 of the 8 applications, at least one of the patterns could be located in the target application's memory. Finding multiple patterns can be caused by false-positives or the values being stored multiple times. Finding no patterns can have different reasons: the values can be stored in other regions than the stack and the heap (e.g. anonymously mapped regions), not stored (e.g. swapped) or converted or combined to unknown values. After identifying applications and locating their potential configuration settings, live cloning of the user-VM could be performed in 6.5 ± 0.2s on average while on-the-fly memory modification (with account injection) caused an additional overhead of 2.1 ± 0.6s (10 runs).

In order to test the feasibility of the automatic *Setting Improvement* strategy, we first create a mechanism which triggers a running application on the network with a lot of new connections, sequenced and in parallel, which tries to log-in with invalid and valid credentials (injected account) and which sends different amounts of data. The Security Monkey uses this automatic mechanism to trigger an application on its port and to monitor the sent traffic of this application on the virtualized network. In our test-runs, the Monkey analyzed how many TCP RST and FIN flags are sent by the application and uses this information as feedback assuming a higher occurrence of these flags is more restrictive and can be caused by the application's improved security characteristics.

Experiments showed that a Security Monkey can automatically improve the security characteristics of three of the eight selected applications (37.5%) with the help of this triggering mechanism and the analysis of the application's feedback. These results are caused by the fact that each of the identified values of these three applications included a value which was directly related to network restrictions (maximum login retries or maximum connections in parallel). The effects of modifying these values could be successfully detected by the Security Monkey in the feedback.

5 Ongoing and Future Work

In future work, we want to improve our *Setting Discovery* strategy, first by performing a large-scaled analysis of configuration files in order to identify interesting classes of values and second, by including more different configuration setting types (e.g. also strings like "md5" or "sha1"). In the *Setting Improvement* strategy, we want to include more different triggering mechanisms and measure and analyze feedback from multiple sources (file and library usage, system calls). Additionally, we want to show the feasibility of a fully automated Security Monkey by using a combinatorial optimization algorithm based on our techniques to find the optimal settings.

6 Conclusion

In this paper, we showed our current work in exploring how security configuration settings of arbitrary and unknown running applications can be automatically improved in a cloud computing setup. We call this approach "hot-hardening" and propose a Security Monkey, which is an agent running in a separate VM that operates transparently on identified applications by leveraging virtualization technologies and that uses our proposed *Setting Discovery* and *Setting Improvement* strategy. In an evaluation, we showed the feasibility of our strategies.

References

[1] AL-SHAER, E., EL-ATAWY, A., AND SAMAK, T. Automated pseudo-live testing of firewall configuration enforcement. *Selected Areas in Communications, IEEE Journal on 27*, 3 (2009), 302–314.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), SOSP '03, ACM, pp. 164–177.

[3] BIEDERMANN, S., AND TEWS, E. How to enable live cloning of virtual machines using the xen hypervisor. In *Technical Report* (2013).

[4] CARBONE, M., CONOVER, M., MONTAGUE, B., AND LEE, W. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. Stolfo, and M. Cova, Eds., vol. 7462 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 22–41.

[5] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286.

[6] DOLAN-GAVITT, B., BRYAN, P. D., AND LEE, W. Leveraging forensic tools for virtual machine introspection. In *Technical Report* (2011).

[7] DOLAN-GAVITT, B., LEEK, T., HODOSH, J., AND LEE, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security* (2013), CCS '13, ACM, pp. 839–850.

[8] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (may 2011), pp. 297 –312.

[9] FRASER, T., EVENSON, M., AND ARBAUGH, W. Vici virtual machine introspection for cognitive immunity. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (dec. 2008), pp. 87 –96.

[10] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium* (2003).

[11] GU, Z., DENG, Z., XU, D., AND JIANG, X. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on* (oct. 2011), pp. 147 –156.

[12] HUANG, H., TSAI, W.-T., AND CHEN, Y. Autonomous hot patching for web-based applications. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International* (July 2005), vol. 2, pp. 51–56 Vol. 1.

[13] PAYER, M., AND GROSS, T. Hot-patching a web server: A case study of asap code repair. In *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on* (July 2013), pp. 143–150.

[14] PAYNE, B. D., CARBONE, M., SHARIF, M. I., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy* (2008), pp. 233–247.

[15] RABKIN, A., AND KATZ, R. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ICSE '11, ACM, pp. 131–140.

[16] RAMASWAMY, A., BRATUS, S., SMITH, S., AND LOCASTO, M. Katana: A hot patching framework for elf executables. In *Availability, Reliability, and Security, 2010. ARES '10 International Conference on* (Feb 2010), pp. 507–512.

[17] SRIVASTAVA, A., AND GIFFIN, J. Tamper-resistant, application-aware blocking of malicious network connections. In *Recent Advances in Intrusion Detection*, vol. 5230 of *Lecture Notes in Computer Science*. 2008, pp. 39–58.

[18] SUN, Y., LUO, Y., WANG, X., WANG, Z., ZHANG, B., CHEN, H., AND LI, X. Fast live cloning of virtual machine based on xen. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on* (2009), pp. 392–399.

[19] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: finding the needle in the haystack. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 6–6.

[20] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, ACM, pp. 244–259.

Notes

[1]http://aws.amazon.com/ec2/
[2]https://github.com/Netflix/SimianArmy/wiki/
[3]https://code.google.com/p/vmitools/
[4]https://code.google.com/p/volatility/