Hot-Hardening: Getting More Out of Your Security Settings

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

Applying optimized security settings to applications is a difficult and laborious task. Especially in cloud computing, where virtual servers with various pre-installed software packages are leased, selecting optimized security settings is very difficult. In particular, optimized security settings are not identical in every setup. They depend on characteristics of the setup, on the ways an application is used or on other applications running on the same system. Configuring optimized settings given these interdependencies is a complex and time-consuming task. In this work, we present an autonomous agent which improves security settings of applications which run in virtual servers. The agent retrieves custom-made security settings for a target application by investigating its specific setup, it tests and transparently changes settings via introspection techniques unbeknownst from the perspective of the virtual server. During setting selection, the application's operation is not disturbed nor any user interaction is needed. Since optimal settings can change over time or they can change depending on different tasks the application handles, the agent can continuously adapt settings as well as improve them periodically. We call this approach hot-hardening and present results of an implementation that can hot-harden popular networking applications such as Apache2 and OpenSSH.

1. INTRODUCTION

Configuration settings which are not well-chosen, not correctly applied or too weak can lead to security breaches and loss of sensitive information. For example, allowing users to set short passwords while also not limiting the amount of login attempts can make systems vulnerable to brute force password guessing attacks. Often, user selected security settings are lax and do not fit well into the appropriate scenario. On the other hand, strong and static security settings can be perceived as being cumbersome since they can drastically decrease usability. In the worst case, they can even cause users to become annoyed and motivated to disable or circumvent the settings. For example, recall the famous anecdote about users writing down complex passwords on a memo mounted on the computer display.

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA

http://dx.doi.org/10.1145/2664243.2664246.

In this work, we analyzed configuration settings of 28 popular Linux networking applications. The average application has 43 different settings while 17 of these settings are related to security characteristics (40%). This means if a server runs three different networking applications, on average more than 50 security settings can be adapted to the setup and its specific properties.

In this context, we introduce the term "custom-made" settings. For example, the optimal limit for the maximal allowed length of requests to a Web server varies depending on the type of content a Web server provides and which kind of tasks it usually processes. Selecting the same limit for different Web servers will not be optimal even if these Web servers are the same application – the properties of the target specific setup in which a Web server runs need to be included in the selection process. By analyzing properties and characteristics of a Web server's setup an optimal setting can be derived which we call custom-made setting.

Moreover, depending on how a setup is changing over time, if software is installed or removed, if the number of users or the ways in which applications are used varies, custom-made security settings can change over time as well. Maintaining an overview in this medley of settings is a complex and time-consuming task in which failures can occur. For example, SSL/TLS misconfiguration attacks have been presented where if default setting values are not correctly set they will result in potential man-in-the-middle attacks due to peer hosts not being verified [1]. Obviously, an automated approach for continuous hardening of applications seems to be a promising solution.

In this work, we present an agent that can autonomously and transparently improve the security settings of applications that run in virtual servers. The agent generates custom-made settings for each application from its setup by analyzing the application's characteristics and properties, for example by investigating an application's network traffic or certain log files. The agent autonomously deploys these settings to the running application unbeknownst from the perspective of the virtual server and without the need to restart the application. The agent can periodically adapt the settings and dynamically change them depending on occurred events or depending on changes in the characteristics of the setup over time. Furthermore, the agent can also change the settings promptly depending on different tasks an application handles. This way, the agent can continu-

ously provide optimized settings for an application in its unique scenario. We call this approach "hot-hardening". In particular, we define hot-hardening as the *continuous and transparent adaptation of security-related configuration settings of an application depending on properties and characteristics of its setup*. Hot-hardening can be provided as a scalable cloud service in a cloud computing scenario.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 shows the architecture of the proposed agent and the different techniques it uses. Section 4 explains the dynamic features of the agent in detail. Section 5 evaluates the agent by investigating different hot-hardening procedures, Section 6 discusses limitations and Section 7 concludes.

2. RELATED WORK

In this section we list related work in the fields of hot-patching and virtual machine introspection.

2.1 Hot-Patching of Applications

Hot-patching is an existing technique for transparently updating a target application's binary code, while our new technique hot-hardening focuses on security settings of applications and not the executable binary code. Hot-patching was developed in order to minimize application and server down-times, as it applies patches to an application during its run-time. In particular, this technique is of interest when administrators want to deploy regularly released security patches. Huang et al. [12] developed an autonomous hot patching framework which can automatically identify the cause of a failure and patch the binary code of Web-based applications. However, hot-patching can also be risky since the system can run into an inconsistent state. To solve the consistency issues, Ramaswamy et al. [16] proposed a method for hot-patching ELF binaries which supports synchronized global data. In their work, they use Patch Objects which are patches encoded as ELF relocatable object files and which can be automatically created and applied to a running process. Payer et al. [13] developed an update system that provides hot-patching by integrating dynamic patches with a sandbox based on dynamic binary translation. In a prototype implementation, they could patch 45 of 49 bugs on the Apache Web server at run-time.

2.2 Virtual Machine Introspection

Virtual machine introspection (VMI) allows to read and write the memory of a target running virtual machine (VM) from another privileged VM which is located on the same physical server. This way, transparent security tools can be developed [10, 15]. These tools are isolated from the operating system of the target VM which they monitor. Accordingly, these tools are tamper-resistant since they can not be manipulated or detected from the perspective of the target VM. VMI is the technique of timely and transparently applying forensic methods on a target operating system's memory with the difference that the system is still running and not terminated. Based on this technique, Baiardi et al. [2] proposed an intrusion detection tool which merges traditional host-based and network-based intrusion detection techniques with VMI. Fraser et al. [8] developed a security tool which automatically identifies and repairs infected kernels using various methods which include VMI. Payne et al. [14] proposed security tools which actively monitor hooks of VMs while these tools are isolated and tamper-resistant in another trusted VM. Benniger et al. [4] used light-weight introspection techniques based on para-virtualization in order to efficiently and transparently detect malware in commercial cloud computing environments. VMI can be used to build security tools that passively monitor target VMs, however, VMI can also be used to inject content into the operating system's memory of a VM and this way, for example, enforce the execution of code [11].

In order to successfully use VMI, information about the operation systems internals of the target VM is required. This is the so called "semantic gap" – certain static memory addresses depending on the used operating system need to be known to be able to extract useful information from a running VM. This introspection information is usually retrieved manually or assumed as being known. However, there are several approaches for automatically bridging this semantic gap. Dolan-Gavitt et al. [7] presented an approach to retrieve the introspection information by automatically analyzing dynamic traces of small programs running in the target VMs. Yangchun Fu et al. [9] identified the required introspection information using system wide instruction monitoring.

In our work, we use fine-grained VMI to transparently locate and modify security settings of applications which run in VMs. In particular, we focus on networking applications.

3. HOT-HARDENING ARCHITECTURE

In this section we describe our techniques used to transparently change settings of running applications, we introduce application-dependent hot-hardening templates used by our agent, and the ways in which settings are improved.

3.1 Hot-Hardening Procedure Overview

The agent runs in a separate agent-VM, which is a small VM equipped with our software. The agent is privileged and can access the hypervisor layer as well as all the memory addresses on the hardware component on which it runs. The agent runs co-located with multiple user-VMs and can periodically hot-harden the user-VMs and their applications with our proposed techniques. An simplified overview of a hot-hardening procedure can be seen in Figure 1.

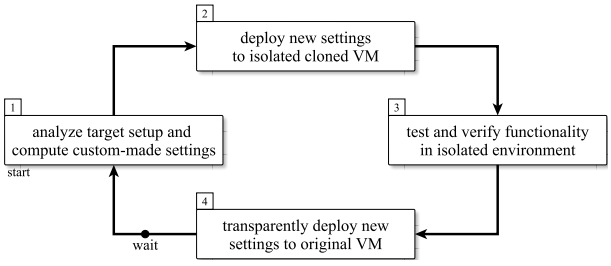


Figure 1: Simplified overview of the steps in a hot-hardening procedure performed by the agent.

In the first step, the agent discovers the operating system of the target VM and its running applications. For each operating system, there is a set of applications that the hot-hardening agent can recognize and hot-harden. For supported applications, the agent analyzes the properties and the status of the setup and computes custom-made settings

for that application. In the second step, the agent locates and deploys the new settings to a clone of the target VM in an isolated environment. This way, the settings can be tested and the execution of the original target VM is not disrupted. In the third step, the agent verifies the correct execution of the application as it runs in the cloned VM; and finally, in step four, it transparently deploys the new settings to the original target VM. Because custom-made settings can change over time, the agent repeats the hot-hardening procedure in certain intervals.

Since retrieving custom-made settings and testing the settings is a complex task requiring different strategies as well as semantic knowledge about the settings and their characteristics, we equip our agent with application-dependent hot-hardening templates – one template for every supported application. The agent can be equipped with multiple templates to support various applications. A template needs to be manually created and defines properties of an application and characteristics of its settings which are supported for hot-hardening. For each supported setting, the template can define two modules that are used by the agent: 1) to retrieve a custom-made setting from a setup (get-modules) and 2) to test a setting on a setup following a well-defined strategy (test-modules). The modules retrieve and analyze specific setting-dependent information from a setup.

3.2 Discovery and Hot-Hardening Templates

In first steps, the agent investigates which applications run on a target user-VM by transparently analyzing the VM's file system on the storage and open ports on the network layer. For discovery, the agent can use third party tools like Nmap[1] which offers a database of fingerprints of more than 2000 well-known services. The agent can also transparently scan the volatile memory of the VM and use a database of hashes of know applications to discover running applications. Once an application was discovered and the agent owns a hot-hardening template for this application, the agent initiates a hot-hardening procedure. Figure 2 shows an example template for an fictitious application called *mysrv*. It contains the name and the version of *mysrv* used to match a template to a discovered application. It defines the path to *mysrv's* configuration file located on the user-VM's storage which is required to retrieve *mysrv's* currently deployed settings. Not all settings of an application are suitable for hot-hardening. Supported settings are manually selected and defined in the template. Some applications may not be suitable for hot-hardening because they do not have any settings or they only use settings in their start-up phase and do not read them again during run-time or they do not store settings in the VM's volatile memory. For the example application *mysrv*, three different settings are supported:

- Setting s01 can be set in a range of $0 - 100$, stepwise changed by 10. The direction "low" is defined because the security characteristics of *mysrv* increase the lower s01 is set. There is a module *get01.py* available that derives a custom-made setting for s01 from a target setup. Furthermore, there is a module *test01.py* that can test the effect of s01 in a setup. In a real application, this setting could be a limit on the number of allowed requests.

[1]http://nmap.org

```
<template>
<name>mysrv</name>
<version>1.1b</version>
<config>/etc/example.cfg</config>
<setting name="s01">
        <range>0-100</range>
        <steps>10</steps>
        <direction>low</direction>
        <getModule>get01.py</getModule>
        <testModule>test01.py</testModule>
</setting>
<setting name="s02">
        <range>0-2</range>
        <steps>1</steps>
        <direction>low</direction>
        <testModule>test02.py</testModule>
</setting>
<setting name="s03">
        <range>1024-4096</range>
        <steps>128</steps>
        <direction>high</direction>
        <getModule>get03.py</getModule>
        <testModule>test03.py</testModule>
</setting>
<instruction>s01|0|s03|*|s02</instruction>
<hothard-steps>1,2,3</hothard-steps>
<location>heap</location>
</template>
```

Figure 2: Hot-hardening template of an application.

- Setting s02 can only be set in a range of $0 - 2$, stepwise changed by 1 where lower settings are beneficial for security of *mysrv*. There is no module available to derive a custom-made setting for s02 which means the setting is independent from a setup. However its effect can be tested with the module *test02.py*. In a real application, this setting could be a string, for example defining a log-level (high, medium, low), but stored in volatile memory as an integer.

- Setting s03 has a possible range of $1024 - 4096$ while it can be stepwise changed by 128 and higher settings are beneficial for security of *mysrv*. There is a module *get03.py* that computes a custom-made setting for s03 and a module *test03.py* that can test its effect. In a real application, this setting could be the length of a cryptographic key.

In a hot-hardening procedure, the agent does not just set the best possible (highest or lowest) setting. Instead, the agent uses the get-modules to find the best setting for the specific target setup in the allowed ranges defined by the template. Additionally, the template defines an instruction indicating how to generate a pattern which is used to locate the data structure that stores the currently deployed settings in *mysrv's* volatile memory. In this context, the order of the settings in memory as well as the allowed content in-between them needs to be defined. In order to define this instruction, manual investigations of the application's source code is required. Furthermore, the template defines a memory address region in *mysrv's* memory in which the data structure of the settings is stored (heap) to accelerate the localization procedure and the different steps required in *mysrv's* hot-hardening procedure. These steps are explained in Section 3.4.

3.3 Isolated Setting Testing

Changing settings via VMI during an application's runtime can lead to interruptions in the application's operation, unexpected results and to misleading entries in log files. In order to avoid these problems on the user-VM, the agent clones the user-VM, deploys and tests settings on the clone. Once better working settings could be found, the agent deploys them on the original user-VM. Live cloning [17, 5] is a modified live migration [6] process which creates a synchronous replica of a target VM during its run-time. A new VM-container is created and the VM's memory is copied while memory pages which were modified during the copy process are again copied in multiple iterations until a certain synchronization is reached. The VM's storage is not copied, instead a copy-on-write snapshot is used (Figure 3).

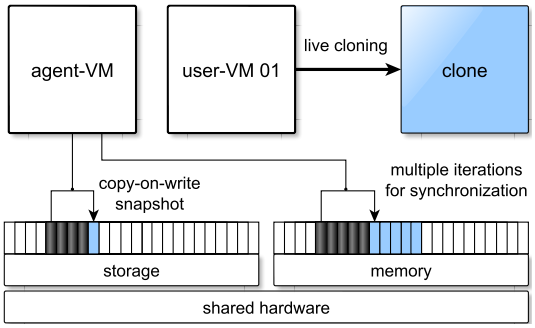


Figure 3: The live cloning process triggered by the agent to test new settings on the clone.

If the agent changes multiple different settings or settings of different applications on the same user-VM, dependencies of settings can play a role. For example, application A can operate together with application B and changing a setting of application A can increase their mutual workload while changing a setting of application B decrease the amount of workload application B can handle. In order to test new settings and to detect potential unknown dependencies or any other problems, the agent evaluates the effect of each new setting by executing a setting dependent test-module. Each test-module which tests the characteristics of a setting i returns a numerical result R_i that can deviate from the known theoretically expected result R_i^e depending on the currently deployed setting. If multiple settings are changed, the agent first deploys the best settings as defined in the template and tests the effects on the setup. If the results of the test of a new setting deviates from its expected effect ($R_i^e - R_i > 0$), the agent stepwise reduces the setting according to the defined steps in the template. In multiple rounds, the agent evaluates the tests of all changed settings and tries to find the best group of settings. This way, the agent can reveal if changing a setting influences the effect of another setting.

It may happen that an application terminates during the hot-hardening procedure. In this case, the agent creates a new clone. Once the best working group of settings could be found, the agent terminates the clone and finally deploys the new tested settings on the original user-VM.

3.4 Transparent Setting Deployment Steps

Transparently changing settings on a running application is facilitated by the underlying virtualization technologies that allow the agent to use different transparent techniques on a user-VM. First, the agent can use VMI to read and modify a running application's volatile memory. Second, the agent can use forensic methods on the running VM's raw storage and this way retrieve information about stored non-volatile data like files. Settings are automatically replaced by the agent unbeknownst from the perspective of the user-VM and without the need to interrupt an application's operation. In particular, the agent deploys new settings by executing up to three different steps which are illustrated in Figure 4:

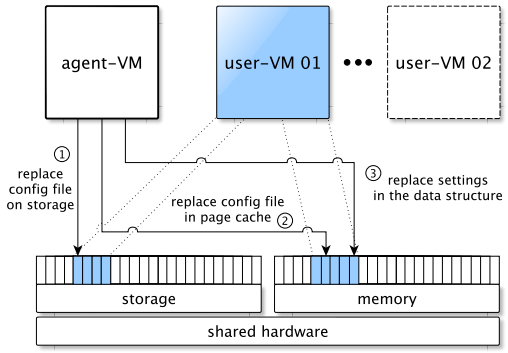


Figure 4: The agent can use three different steps to transparently deploy new settings to a running application on a user-VM.

1. The agent locates the configuration file of the application by investigating the file system's meta data on the user-VM's raw storage. Once the corresponding sectors are located, the sectors are overwritten and the content is replaced with new content in the same structure containing the new settings. Replacing the configuration file on the storage prevents reuse of old settings once the application is restarted or if it reloads settings from its configuration file during run-time.

2. The agent tries to locate and replace a cached copy of the application's configuration file in the operating system's page cache with the new content. The page cache helps applications to retrieve stored data faster by keeping recently requested data in memory. Replacing potentially cached content of the configuration file prevents reuse of old settings.

3. The agent locates and replaces the currently used configuration settings in the application's volatile memory. To reach this goal, data structure alignment and endianness of the settings need to be considered. There are different ways in which an application can process a configuration file and store its settings in memory, depending on the programmed data structure and on the used data types. Often, settings are summarized and stored in a struct data type, sometimes settings are stored independently. However, sometimes settings are not stored in memory and read from the file each time they are required. In order to locate the settings in memory, the agent uses the $< instruction >$ block from the template to generate a pattern and the target memory region defined in the template.

Sometimes, depending on the target application, only steps 1 and 2 are required, because the application does not store settings in memory. Sometimes only step 1 is required, because there is no page cache as well as there are no settings in memory. Sometimes only step 3 is required, because there is no configuration file but there are settings in memory fetched from other sources such as system variables or command line arguments. The template defines which steps are required for the application.

4. DYNAMIC CHILD ADAPTATION (DCA)

Up to this stage, the agent can transparently change settings, test the new settings in an isolated environment (cloned user-VM) and finally deploy them on a user-VM. In order to define the semantics of settings and allow the agent to work autonomously, we introduced application-dependent templates. Furthermore, the proposed agent uses setting-depending modules to retrieve a custom-made setting by analyzing a target setup and to test the effects of a new setting. According to Figure 1 the agent can periodically repeat this hot-hardening procedure on the same user-VM and this way adapt settings which may become suboptimal over time since the setup or the scenario can change.

However, sometimes it is beneficial to allow the agent to dynamically change settings depending on events. In this context, an event can be a new connection or a new task the application handles. Often, an application creates temporary child processes to handle different tasks separately, for example it creates a new child process per new network connection. Sometimes, an application also assigns the data structure of its settings to the child processes or the workflow of the child processes can be adapted by temporary changing the stored settings of the main process for a very short-time during child process creation. This allows the agent to modify the settings of new child processes separately in the volatile memory. This way, the agent can deploy different settings for each new child process depending on certain events or properties.

We call this fine-grained and dynamic hot-hardening procedure Dynamic Child Adaptation (DCA). In particular, a property of a child process, for example of the connection it processes, can lead to an adaptation of the child's settings. If an application has been hot-hardened once, the agent can continuously monitor the user-VM and adapt certain settings of the corresponding child processes. As an illustrative example, we extend the hot-hardening template of *mysrv* which now additionally defines for which settings DCA is supported and which properties lead to which adaptations (c.f. `<dca>` tags in Figure 5). The template allows the agent to use DCA for the two settings s01 and s03. Once a child process handles a connection from the own system, setting s01 is set to 100 for the child. Once a child process handles a connection from the same sub net, setting s03 is set to 1024 for the child. This way, security settings can be relaxed under certain circumstances in order to increase usability or security characteristics can be dynamically amplified. Furthermore, in this case, the settings are directly deployed on the user-VM without creating and testing them on a clone first. However, DCA is only supported after the application has already been hot-hardened, which means changing settings was already tested. The DCA procedure which the agent performs to adapt child processes is illustrated in Figure 6 in more detail. First, a new incoming

```
<template>
<name>mysrv</name>
<version>1.1b</version>
<config>/etc/example.cfg</config>
<setting name="s01">
        <range>0-100</range>
        <steps>10</steps>
        <direction>low</direction>
        <getModule>get01.py</getModule>
        <testModule>test01.py</testModule>
        <dca>
                <event><src>127.0.0.1</src></event>
                <set>100</set>
        </dca>
</setting>
...
<setting name="s03">
        <range>1024-4096</range>
        <steps>128</steps>
        <direction>high</direction>
        <getModule>get03.py</getModule>
        <testModule>test03.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>1024</set>
        </dca>
</setting>
...
```

Figure 5: Extensions to *mysrv's* hot-hardening template in order to support DCA for s01 and s03.

connection is detected and analyzed by the agent before it reaches the user-VM. The agent retrieves the source IP address and other information about the new connection. This is possible since communication is tunneled through the virtualized bridge of the hypervisor. If the properties of the new connection match a DCA event definition in the extended template, the connection is delayed until the settings of the application are adapted. After adaptation, the connection is allowed to proceed and the application now creates a specially adapted child process. Once the child was created, the agent immediately reverts the settings in the parent application back to the custom-made settings which are effective for other clients.

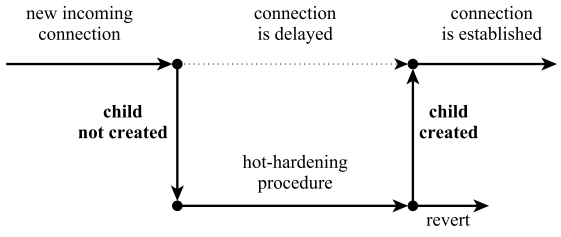


Figure 6: Steps in the Dynamic Child Adaptation (DCA) hot-hardening procedure in detail.

5. EVALUATION

In the evaluation, we investigate hot-hardening applied to popular applications and analyze the results in detail. We use a Xen [3] server with a Quad CPU (2.67 GHz) and 8 GB memory and a user-VM with 4 GB memory. Both run Ubuntu Linux 12.04 64-Bit.

5.1 Example: Apache2

First, we investigate a hot-hardening procedure for the Apache2 Web server 2.2. Our Apache2 Web server is a production system hosting our Web page and giving access to a regularly used SVN repository. Without considering any Apache2 extensions, we identified 7 settings of the application which are related to security and which are suitable for hot-hardening. Apache2's configuration file is located in */etc/apache2/apache2.conf*. The settings are stored in Apache2's volatile memory as a *struct* data type called *server_rec* defined in the source code in *include/httpd.h*:

```
struct server_rec {
    ...
    /** Timeout, before we give up */
    apr_interval_time_t timeout;
    /** Interval we will wait for another request */
    apr_interval_time_t keep_alive_timeout;
    /** Maximum requests per connection */
    int keep_alive_max;
    ...
};
```

By default, Apache2 handles new connections in child processes which maintain an own struct of the settings. This means that DCA is applicable.

5.1.1 Apache2's Hot-Hardening Template

A fragment of the hot-hardening template for Apache2 can be seen in Figure 7. Seven different settings are supported. Their effects mitigate Brute-Force attacks, prevent malicious tools scanning for vulnerabilities and misconfiguration and avoid Denial-of-Service (DoS) attacks and the execution of exploits (for example based on large or unusual input). The supported settings are briefly explained in the following:

- s01 defines the log level ($0-7$) and is originally a string (emerg, alert, crit, ..., info, debug). The template limits the setting to seven levels $0-6$ (no debug mode) which can be changed stepwise by 1 while 6 logs the most (info). Logging more details increases Apache2's security characteristics.
- s02 defines the time Apache2 waits to receive a request from a new connection before terminating it. The template defines a range of $10-300$ seconds for s02 which can be stepwise changed by 10. A low setting increases the security characteristics of Apache2.
- s03 defines the maximum number of requests allowed during a persistent connection before Apache2 terminates it. The template defines a range of $10-100$ for s03 that can be stepwise changed by 2, the lower it is set the better the security characteristics are.
- s04 defines the time for which a connection is maintained for a next request. The template defines a range of $2-30$ seconds, a lower value is beneficial for security.
- s05 defines a limit for the size of HTTP request lines in bytes. The template defines a range of $1024-8192$ while low is assumed as being better.
- s06 limits the size of request header fields in bytes and ranges between $1024-8192$.
- s07 limits the number of header fields in requests. It can be set to a range of $10-100$.

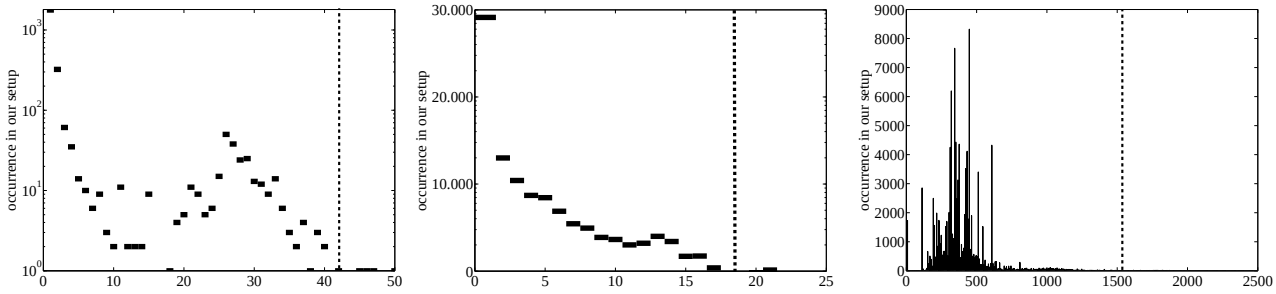
```
<template>
...
<setting name="s01">
        <range>0-6</range>
        <steps>1</steps>
        <direction>high</direction>
        <testModule>test01.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>1</set>
        </dca>
</setting>
...
<setting name="s06">
        <range>1024-8192</range>
        <steps>512</steps>
        <direction>low</direction>
        <getModule>get06.py</getModule>
        <testModule>test06.py</testModule>
        <dca>
                <event><src>192.168.0.0/24</src></event>
                <set>8192</set>
        </dca>
</setting>
...
<instruction>
s01|*|s02|0|s04|0|s03|.|s05|0|s06|0|s07</instruction>
<hothard-steps>1,2,3</hothard-steps>
<location>anonymous</location>
</template>
```

Figure 7: The hot-hardening template of Apache2.

For s01 no get-module is available since this setting is assumed as being independent from a setup. For the settings s02, ..., s07, get-modules are available which can retrieve a custom-made setting from a setup (see Section 5.1.2). The effect of each setting can be tested with a setting-dependent test. The template also defines an instruction how to generate a pattern in order to locate the currently deployed settings (derived from the configuration file) in Apache2's volatile memory. In this context, the order of the stored settings and the content in-between them (for example other settings) is given. This information was derived from the source code. The template defines which hot-hardening steps are required (Section 3.4) and the memory region in which the data structure of the settings is stored. In this case, it is stored in anonymous mappings created by mmap() with the *MAP_ANONYMOUS* flag. Analysis of the source code showed that a child process maintains the data structure. Accordingly, the template allows dynamic child adaptation (DCA), but restricted to the settings s01 and s06. Setting s01 is dynamically set to 1 which means less events need to be logged if a client connects from our sub network. Setting s06 is set to the maximum of 8192 for clients connecting from our private sub network which allows them to perform longer requests with more headers.

5.1.2 Apache2's Hot-Hardening Modules

The template specifies six get-modules to retrieve custom-made settings for s2, ..., s7 from our setup. The computed settings depend on various properties, for example the workload of our system, other installed applications or the kind of data our Web server provides. Our Apache2 server hosts



(a) Requests per connection. (b) Time between requests [s]. (c) Length of requests [bytes].

Figure 8: Distributions of different events extracted from our setup by the get-modules which use this data to derive a custom-made setting for s03, s04 and s05 (marked as dotted line).

a Web page and a repository (SVN) that can be remotely used by employees who mostly connect from within our private network. This is a special property of our setup which actually leads to a frequent occurrence of multiple GET or PUT requests in a row caused by users rapidly synchronizing SVN projects consisting of multiple files. The get-modules compute an average value of events occurring on the setup which are affected by a setting or analyze the distribution of events based on analysis of log files or network traffic of the application. The get-modules select the next best possible value as custom-made setting according to the definitions in the template.

- get02.py analyzes network traffic and the timings of new connections and their first request. In our setup, the mean is 2.89±4.01 seconds which leads to a custom-made setting of 10 seconds (default is 300).
- get03.py analyzes network traffic and investigates the amount of requests performed per connection (Figure 8a). In our setup, there is a second peak in the figure caused by users synchronizing multiple files with our SVN repository. This is a special property of our setup and leads to a custom-made setting for the limit of requests per connection of 42 (default is 100).
- get04.py analyzes timings between multiple request on a persistent connection (Figure 8b). It calculates a custom-made setting of 18 seconds for our setup (default is 30).
- get05.py analyzes the length of requests retrieved from the Apache2 access log file (Figure 8c). It calculates a custom-made setting of 1536 bytes for our setup (default is 8190).
- get06.py analyzes the length of the headers in requests from the Apache2 log file. It calculates a custom-made setting of 1024 bytes for our setup (default is 8190).
- get07.py analyzes the numbers of fields used in headers in requests. It calculates a custom-made setting of 30 (default is 100).

The get-modules consider the special properties of our setup and derive custom-made settings for Apache2 which increase security while minimally affecting the work-flow. The test-modules actively perform tests on the setup. Each test-module analyzes the effect of a currently deployed setting. For example, they establish new connections to Apache2 to verify expected timeouts, limits or other characteristics caused by the settings. In this context, the test-module for s01 analyzes the information that is logged while it actively performs requests, other test-modules analyze the timeouts for s02 and s04 while maintaining a connection and the module for s03 tests how many requests are possible on a persistent connection. The modules which test s05, s06 and s07 perform different kinds of requests (length and header fields) to evaluate the corresponding setting.

5.1.3 Apache2's Hot-Hardening Procedure

In this section, we investigate the hot-hardening procedure in detail after the custom-made settings have been computed. First, the agent clones the user-VM before applying and testing any new settings. In our setup, live cloning of a running user-VM with 4 GB of memory into a synchronous replica can be performed in 17 ± 2 seconds on average. This timing strongly depends on the current workload of the VM. In order to reduce the time required for cloning, prepared clones can be maintained or they can be created using copy-on-write (CoW) methods for both memory and storage which enables the creation of a clone in the range of milliseconds [17]. In the first step, the agent identifies the currently deployed settings by locating and reading the corresponding sectors of Apache2's configuration file on the clone's raw storage. In general, struct members are stored in the order they are declared (C99 standard) and if necessary, padding is added before each struct member to ensure correct alignment. Setting s02 and s04 are stored as long data types representing microseconds. The other settings are stored as integer data types. Furthermore, the order of the byte representation of s02, s04, s05 and s06 has to be adapted according to Little-Endianness. With the help of the current settings and the instruction defined in the template, the agent generates a pattern to locate the deployed settings in Apache2's memory. The generated characteristic pattern P in hexadecimal representation is

```
P = 04(.)*?00a3e111(0)*?80c3c901 ... fe1f(0)*?64
```

The pattern allows padding with zeros between successive settings and arbitrary content between settings which have other settings in-between.

Retrieving custom-made settings from our setup using the seven different get-modules takes 121 ± 4 seconds based on

an anonymized network capture of 62 MB and a log file of 24 MB. Finding the corresponding sectors of the configuration file on the storage and replacing the content with new content takes only 0.3 ± 0.1 seconds. Locating and replacing the content of the cached configuration file in the Linux page cache takes 5.2 ± 0.9 seconds. Generating the pattern according to the instructions, locating the data structure of the settings in Apache2's memory and replacing the current settings with new settings takes 2.4 ± 0.1 seconds. Once new settings are deployed, the agent executes the setting dependent tests and analyzed the results in order to reveal unexpected deviations. Evaluating the seven tests takes 46 ± 8 seconds. Finally, a full hot-hardening procedure for Apache2 takes around 3 minutes in our setup without any user interaction. This is a good result considering that the agent performs deep analysis on network traffic and log files and that the agent needs to wait during executing test-modules in order to realistically verify timeouts.

For DCA, no configuration files need to be replaced. The settings s01 and s03 are dynamically hot-hardened. Hot-hardening of a child once the triggering properties of a new client could be detected takes only 2.1 ± 0.1 seconds.

5.1.4 Extension: Apache2 and PHP5

Apache2 usually runs in combination with several extensions. In this section, we introduce a template for PHP5 hot-hardening and let the agent hot-harden Apache2 and PHP5 on the same setup. Interesting security-related settings of PHP5 are stored in a struct data type called *php_core_globals* defined in the source code in the file *php_globals.h*:

```
struct _php_core_globals {
    ...
    long memory_limit;
    long max_input_time;
    ...
};
```

The template for PHP5 (no Figure shown) provides five different settings which are related to security and supported by the agent:

- s01 defines the maximum amount of memory (MB) a PHP script can consume (default 128MB).
- s02 defines the maximum time a PHP script can spend parsing a request (default 60 seconds).
- s03 defines the maximum size (MB) for uploads (default 2MB).
- s04 defines the maximum level of nested requests, for example of an array[][][] (default 64).
- s05 defines the maximum number of GET and POST input variables (default 1000).

The get-module of s02 analyses the timings between requests for PHP scripts and the resulting replies sent from the server. The get-module of s03 analyzes the Apache2 log for uploads and their sizes. The get-modules of s04 and s05 analyze the PHP code found in Apache2's public directory (default */var/www/*) for definitions of nested variables and their depth and the amount of variables accepted by scripts.

The agent retrieves the currently deployed settings from the configuration file */etc/php5/apache2/php.ini* and generates a pattern according to the instruction given in the PHP5 template while taking into account data alignment and endianness. The settings are stored in the anonymously mapped memory region of libphp5.so used by Apache2. The agent uses the get-modules to derive new custom-made settings from our setup. The agent uses the test-modules to evaluate the new settings, in this case also by creating and inserting PHP code on the clone. To reach this goal, the agent creates specific PHP files in the public directory of Apache2 and triggers the execution in order to evaluate the PHP characteristics. Execution of the modules could be performed in 104 ± 3 seconds and insertion of new custom-made settings to PHP5 running in the clone while also replacing the stored configuration file and the file in the page cache could be performed in 8 ± 1 seconds.

In the first step, Apache2 is hot-hardened and the setting dependent tests are evaluated for both Apache2 and PHP5 according to the agent's testing strategy defined in Section 3.3. In the second step, the agent hot-hardens PHP5 and again evaluates the tests for both applications. Since the agent evaluates two templates, dependencies can come into play. For example, during Apache2's hot-hardening, the evaluation of PHP5 test03.py detected a high deviation between the expected characteristics and the actually measured characteristics on our setup. PHP5 test03.py evaluates the maximum allowed size for uploads according to the PHP5 setting s03 which defines 2 MB in our setup before hot-hardening. After the agent changed Apache2 setting s02 which defines the timeout for requests to the new custom-made setting of 10 seconds, the PHP5 test03.py could not upload a file of size 2 MB since the new timeout was too low for the upload. According to the proposed strategy, the agent automatically revoked Apache2 setting *s*02 and deployed the next possible step defined in Apache2's template, which is 20 seconds. However, identifying a dependency between Apache2 setting *s*02 and PHP5 setting s03 caused two additional testing rounds and which led to a hot-hardening time for Apache2 of 8 minutes instead of 3 minutes. Dependencies like this can occur when applications share resources or somehow work together like in this scenario. However, with the help of the agent's strategy and well-defined templates, dependencies can be revealed and settings can be successfully adapted. Finally, hot-hardening of Apache2 and PHP5 takes around 11 minutes in our setup.

5.2 Example: OpenSSH2

Finally, we investigate hot-hardening of OpenSSH2 in detail. Our OpenSSH2 server is a production system having several users who log in regularly, also using SCP and sFTP.

5.2.1 OpenSSH2's Hot-Hardening Template

The settings of OpenSSH2 are stored in a struct data type called *ServerOptions* defined in *servconf.h*. The template for OpenSSH2 can be partially seen in Figure 9 and supports 8 different security settings. The supported settings are briefly explained in the following:

- s01: Size of the key (protocol 1, default 768 bits).
- s02: Time after which the key is regenerated (protocol 1) in seconds (default 3600 seconds).
- s03: Different ciphers which are allowed (protocol 2) (e.g. aes128-cbc, blowfish-cbc, ...). The template defines three groups classified by strength.

```
<template>
...
<setting name="s04">
        <range>5-120</range>
        <steps>5</steps>
        <direction>low</direction>
        <getModule>get04.py</getModule>
        <testModule>test04.py</testModule>
</setting>
<setting name="s05">
        <range>1-6</range>
        <steps>1</steps>
        <direction>low</direction>
        <getModule>get05.py</getModule>
        <testModule>test05.py</testModule>
</setting>
<setting name="s06">
        <range>0-1</range>
        <steps>1</steps>
        <direction>low</direction>
        <getModule>get06.py</getModule>
        <dca>
                <event><src>192.168.1.7</src></event>
                <set>1</set>
        </dca>
</setting>
...
</template>
```

Figure 9: The template of OpenSSH2.

- s04: The server disconnects after this defined time if the client has not logged in (default 120 seconds).
- s05: Maximum number of authentication attempts allowed before the connection is terminated (default 6).
- s06: Specifies if host-based authentication (together with public key) is allowed (default *no*).
- s07: Specifies if forwarding of X11 is allowed (default *no*).
- s08: Specifies if root login is allowed (default *no*).

The template defines DCA for the setting s06, s07 and s08. Host-based authentication is only enabled for the back-up server (192.168.1.7) that automatically transfers and stores new data daily. This is done by DCA. Accordingly, no other client is allowed to use host-based authentication. Forwarding of X11 is allowed only for clients connecting from our private network and remote root login only allowed for one host which is used by the administrator. The OpenSSH2 server is dynamically hot-hardened which allows different settings under different circumstances which normally could only be statically deployed for all clients.

5.2.2 OpenSSH2's Hot-Hardening Modules

The get-modules and their results in our specific setup are briefly explain in this section:

- get01.py investigates if protocol 1 is supported and which key size is used by default. It then suggests to use highest setting (2048).
- get02.py and get03.py investigate the status of the setup and propose 300 for s02 and 3 for s03, which means only the strongest cipher algorithms are supported.
- get04.py investigates a network capture and the authentication log file and analyzes the timings between the establishment of a new connection and a successful login. In our setup 7.31 ± 2.76 which leads to a suggested custom-made setting of 10.
- get05.py analyzes the login tries users need to successfully log-in. In our setup 1.11 ± 0.39 which leads to a custom-made value of 2.

The get-modules get06.py, get07.py and get08.py investigate the status and suggest the best setting if feasible.

5.2.3 OpenSSH2's Hot-Hardening Procedure

In our test-runs, the agent clones the user-VM in 16 ± 4 seconds and evaluates the get-modules to retrieves custom-made settings in 47 ± 5 seconds Finding the sectors of the OpenSSH2 configuration file and replacing the content with new content takes 0.2 ± 0.3 seconds. Locating and replacing the file's content in the Linux page cache takes 5.4 ± 0.8 seconds. Generating the pattern and locating the deployed settings in OpenSSH2's volatile memory and replacing the current settings with new settings takes 1.9 ± 0.2 seconds. The testing phase takes only 31.2 ± 2.2 seconds. Finally, the hot-hardening procedure of OpenSSH2 could be automatically performed by the agent in around 1 minute and 42 seconds.

After the main hot-hardening procedure is finished, DCA is enabled, which means the agent continuously monitors the user-VM and dynamically adapts the settings s06, s07 and s08 for new child processes depending on properties of the IP address of a new client. DCA of OpenSSH2 can be performed by the agent in 2.1 ± 0.2 seconds which means a new TCP connection to the SSH port for which DCA is allowed needs only to be delayed for that period of time until it can proceed. Other connections for which DCA is not enabled are only delayed in the range of milliseconds. In our scenario, DCA enables specific settings for our backup server which now can log-in via host-based authentication while this option is disabled for all other clients. In the same manner, X11 forwarding and remote root login is only allowed for specific clients while maintaining stronger security settings for the others. DCA allows to define the utilization of standard settings on a more fine-grained way and can greatly increase security as well as usability.

6. LIMITATIONS

The agent can not hot-harden every application. Some applications do not store configuration settings in memory or they do not provide any configuration settings. Only applications which store settings in memory and which also regularly read and use these settings during their operation are suitable. Some applications only adapt their operation during the start and do not read the settings again, they use external resources or there is a risk of creating race conditions. Additionally, not all security settings of an application are suitable for hot-hardening – they need to be well-chosen and defined in the template. Furthermore, there is no guarantee that unknown dependencies of different applications

can be detected in every scenario. The templates need to be well-defined and the agent should only hot-harden settings which effects can be reliably estimated. For DCA, it is only applicable if the application assigns separate settings to the child processes or adapts their operation based on settings.

7. CONCLUSION

Applying optimized security settings for different applications is a difficult and time-consuming task. In this work, we presented an autonomous agent that can periodically improve the security settings of different applications running on virtual servers which we called hot-hardening. The proposed agent only uses transparent techniques based on virtualization technologies in order to update configuration files and data structures that store settings in an application's memory. During setting selection and improvement, the operation of an application is not disturbed as well as no user interaction is required. The agent can support applications for which a template is defined and for which setting-dependent modules are available. Modifying the source code of an application is not required. The agent uses setting-dependent modules to retrieve custom-made settings from a specific setup and setting-dependent tests in a testing strategy to verify correct functionality. Since optimal settings can change over time the agent can periodically analyze the characteristics of a setup and periodically adapt the security settings. In order to support different setting for different tasks, we introduced Dynamic Child Adaptation (DCA) which enables the agent to rapidly deploy specific settings for new child processes depending on properties of the task the child handles. In an evaluation, we showed that the agent can hot-harden three different popular applications (Apache2, PHP5 and OpenSSH2) in feasible time and automatically improve their security characteristics. Code related to this publication will be made available online at `http://caslab.eng.yale.edu/code`.

Acknowledgments

The work presented in this paper was performed in the context of the Software-Cluster project SINNODIUM (`www.software-cluster.org`). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. "01C12S01V".

8. REFERENCES

[1] #PeerJacking - SSL Ecosystem Attacks Against Online Commerce. `http://www.unrest.ca/peerjacking`.

[2] F. Baiardi and D. Sgandurra. Building Trustworthy Intrusion Detection through VM Introspection. In *Proceedings of the Third International Symposium on Information Assurance and Security (IAS)*, pages 209–214, 2007.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[4] C. Benninger, S. Neville, Y. Yazir, C. Matthews, and Y. Coady. Maitland: Lighter-weight vm introspection to support cyber-security in the cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, pages 471–478, 2012.

[5] S. Biedermann and E. Tews. How to enable Live Cloning of Virtual Machines using the Xen Hypervisor. Technical report, 2013.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the Symposium on Networked Systems Design & Implementation*, pages 273–286, 2005.

[7] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 297–312, 2011.

[8] T. Fraser, M. Evenson, and W. Arbaugh. Vici: Virtual machine introspection for cognitive immunity. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 87–96, 2008.

[9] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 586–600, 2012.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pages 191–206, 2003.

[11] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 147–156, 2011.

[12] H. Huang, W.-T. Tsai, and Y. Chen. Autonomous hot patching for web-based applications. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 51–56, 2005.

[13] M. Payer and T. Gross. Hot-patching a web server: A case study of asap code repair. In *Proceedings of the 11th Annual International Conference on Privacy, Security and Trust (PST)*, pages 143–150, 2013.

[14] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 233–247, 2008.

[15] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 385–397, 2007.

[16] A. Ramaswamy, S. Bratus, S. Smith, and M. Locasto. Katana: A Hot Patching Framework for ELF Executables. In *Proceedings of the International Conference on Availability, Reliability, and Security (ARES)*, pages 507–512, 2010.

[17] Y. Sun, Y. Luo, X. Wang, Z. Wang, B. Zhang, H. Chen, and X. Li. Fast Live Cloning of Virtual Machine Based on Xen. In *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 392–399, 2009.