xorlab

Silently Breaking ASLR In The Cloud

In collaboration with:

Title: Silently Breaking ASLR In The Cloud
Authors: Antonio Barresi (xorlab), Kaveh Razavi (VU Amsterdam),
Mathias Payer (Purdue University), Thomas R. Gross (ETH Zurich)

Date: 11.11.2015
Version: 1.0
Classification: Public

Contents

1 Introduction

This white paper describes Cross-VM Address Space Layout INtrospection (CAIN), an attack vector against page based same content memory deduplication in Virtual Machine Monitors (VMM).

Overall CAIN demonstrates that if a VMM uses page based same content memory deduplication (like e.g. found in KVM, VirtualBox, VMware) then it is feasible for an attacker to silently infer the base addresses of code locations (mapped executables and libraries) of co-located VMs.

The issue is tracked under CVE-2015-2877 and there is a vulnerability note from cert.org under VU#935424 (https://www.kb.cert.org/vuls/id/935424).

Please refer also to the WOOT'15 paper (CAIN: Silently Breaking ASLR in the Cloud 2015) describing the issue in more detail. This white paper focuses on the practical aspects and implications of the attack.

2 Background

2.1 Threat Landscape in Public Clouds

In public clouds, systems are not only subject to attacks from the outside but also from malicious neighbors running on the same physical hardware. VMs of different organizations and companies are deployed right next to VMs of potentially untrusted parties. Weak and vulnerable systems running on the same physical hardware pose a security risk for all their co-located neighbors, since they can be a starting point for attackers.

The attack described in this white paper assumes the threat model depicted in Figure 1. We assume an attacker with user-privileges on an attacker VM to attack a victim VM running on the same physical hardware.

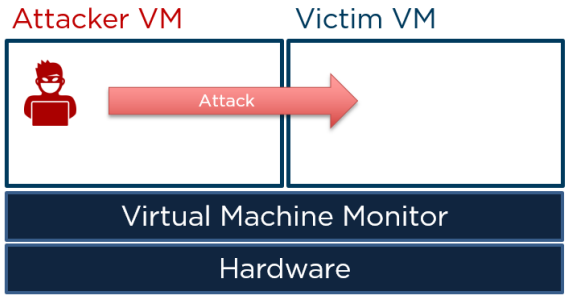

Figure 1: VMs can be threatened by co-located VMs.

2.2 Memory Deduplication

Virtualizing at machine boundary in a public cloud provides isolation between different tenants, and gives them the ability to run an entirely customized operating system for their unique needs.

While VMs are highly desired, they are also quite wasteful. Running multiple instances of the same OS, often with the same software stack, leads to multiple copies of the same data on memory and disk. To avoid this wasteful duplication, techniques have been deployed to "deduplicate'" the VMs data, in memory by sharing physical memory pages, and on disk by sharing file system blocks. Deduplication directly results in a better profit margin for the providers, since now they can pack even more VMs on the same physical host.

While deduplication sounds like a good idea, there is a security pitfall: As previously shown (Memory Deduplication As a Threat to the Guest OS 2011) memory deduplication introduces a side-channel that can be exploited by co-located VMs. The side-channel makes it possible to detect pages that exist in co-located VMs which allows attackers to infer certain information based on the existence of pages in co-located VMs.

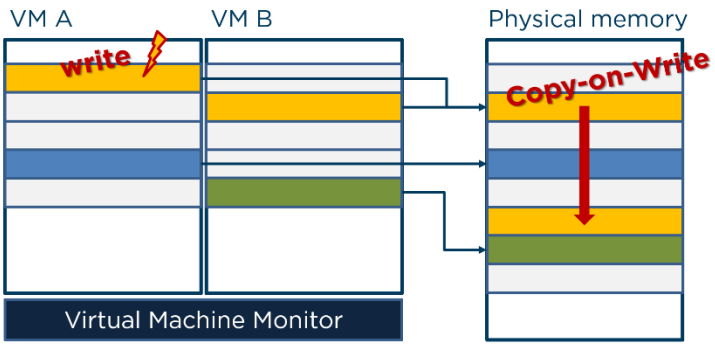

Figure 2: copy-on-write behavior when memory deduplication is used.

When pages in two different VMs are deduplicated the physical page gets marked as read-only. As soon as one VM writes to it a copy-on-write will be performed before the writing VM can effectively modify the page. Figure 2 illustrates the copy-on-write behavior of two shared pages (yellow pages) between VM A and VM B.

This copy-on-write behavior introduces a side-channel that can be exploited by a malicious VM. As the copy-on-write takes significantly more time than a write to a non-shared page an attacker can measure the write time and thus deduce if the page is shared between his VM and a co-located VM.

Figure 3 illustrates the side-channel: an attacker can construct a page (guess) and waits. As soon as the page is deduplicated a write will reveal if the page exists in a co-located VM or not.

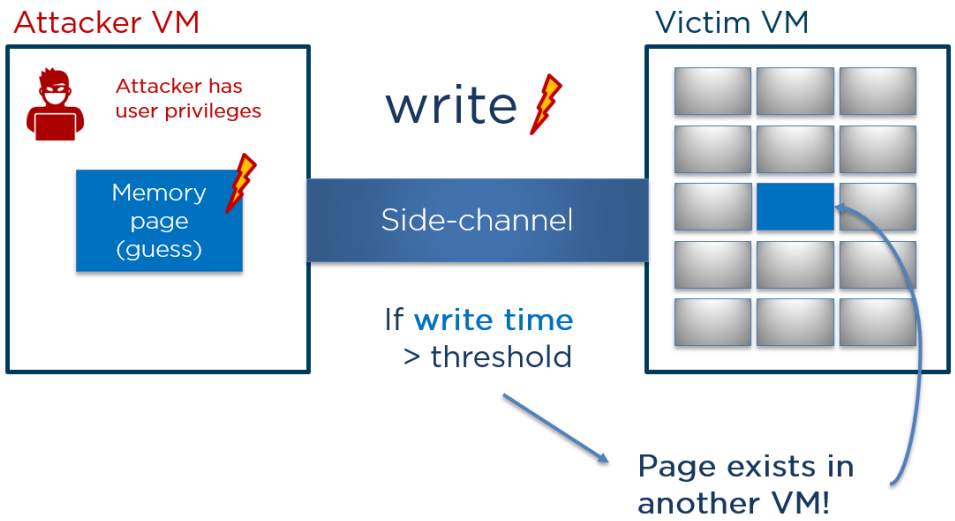

Figure 3: the attacker can observe write time and deduce that the page exists in the Victim VM.

There are different types of deduplication. The issue described in this white paper assumes page based same content memory deduplication like implemented in Kernel Same-page merging (KSM) used in KVM.

2.3 Address-Space Layout Randomization

Together with DEP (Data Execution Prevention), ASLR (Address Space Layout Randomization) is considered one of the main software and system hardening techniques that has found widespread adoption in modern systems over the last decade.

When exploiting memory corruption vulnerabilities, attackers ideally try to construct a write primitive operation. This operation allows the attacker to write arbitrary values to arbitrary memory locations within the victim process virtual address space. This is then leveraged to a control-flow hijack attack by using the write primitive to overwrite control-flow sensitive data (like function pointers, virtual table entries, return addresses on the stack, etc.).

To do so, attackers usually have to know the exact memory location of the values they want to overwrite. The purpose of ASLR is to make it probabilistically infeasible for attackers to guess the exact memory location of the target memory. ASLR introduces randomness for memory addresses during process creation and memory mapping operations that ensures different memory layouts for different processes and systems. The effectiveness of ASLR depends on the number of its entropy bits that are hidden from an attacker and on the ability to not disclose any information about the current layout during run-time.

The ASLR implementation for code, stacks and heaps usually differs on the same system. While stacks and heaps can usually be randomized in a more fine-grained way (e.g., with random padding), code images (executables and libraries) are usually required to be mapped from the file-system to some page aligned base address. Although the base-address can be randomized, the content is much more static and predictable than stack or heap pages.

The main goal of CAIN is leaking randomized base addresses of code images; under certain circumstances it might also be possible to infer randomized addresses of stack or heap regions, but we did not investigate this further.

3 Breaking ASLR with CAIN

3.1 Attack Overview

The goal of the attacker is illustrated in Figure 4. The attacker wants to infer the base address of a code image (e.g. a DLL) mapped into the memory of a process running on a co-located VM.

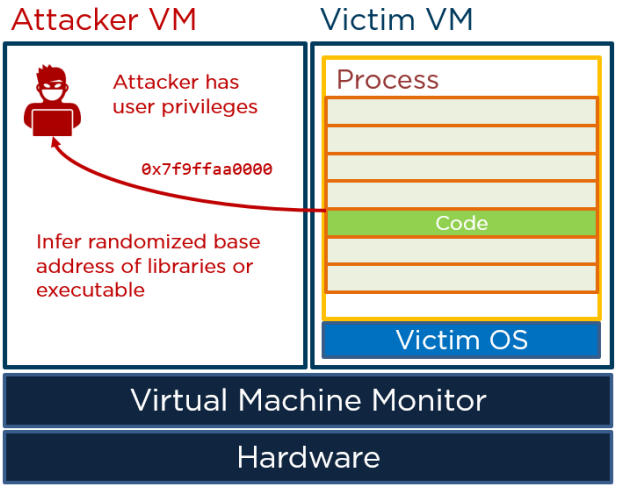

Figure 4: the attacker's goal is to infer the base address of a code image of a co-located VM.

The concept of the attack is to take advantage of specific memory pages, these pages are known to exist in the Victim VM, mostly static so they can be reconstructed, page aligned and they have to contain the randomized base address of the code image of interest (e.g. the base address of ntdll.dll). We will then use the memory deduplication side-channel to detect if such a page exists which will allow us to infer the randomized base address. We use these suitable pages to guess a randomized base address one guess per page. Thus, to infer the unknown base address we simply brute-force the randomized base address by creating many of these guesses concurrently.

There are two main challenges. First, we need to find and construct pages that are suitable for the attack. Second, the attack must be automated and reliable, which requires handling the noise that is present within the memory deduplication side-channel (i.e., higher write times not caused by the Copy-on-Write.).

3.2 Finding suitable pages

Depending on the targeted system, a page with specific properties has to exist for the attack to work. The page must fulfill at least the following properties:

- The data is page aligned (no random padding)
- The content is mostly static and known to the attacker
- The page contains the value we want to infer (i.e., the randomized base address). Or values derived from it, basically we should be able to exactly reconstruct the content of the page given the value to be inferred (i.e., the entropy of the page is the entropy of the value to be inferred)
- The page is more or less always in main memory (in the victim's VM)

At first, it might seem that finding such a page is difficult. But these pages exist. Despite application specific pages that might adhere to the properties above, code pages with base relocations e.g., fulfill the criteria.

3.2.1 Suitable pages in Windows

On Windows systems for instance the very first page of every Portable Executable (PE) image fulfills the properties described above.

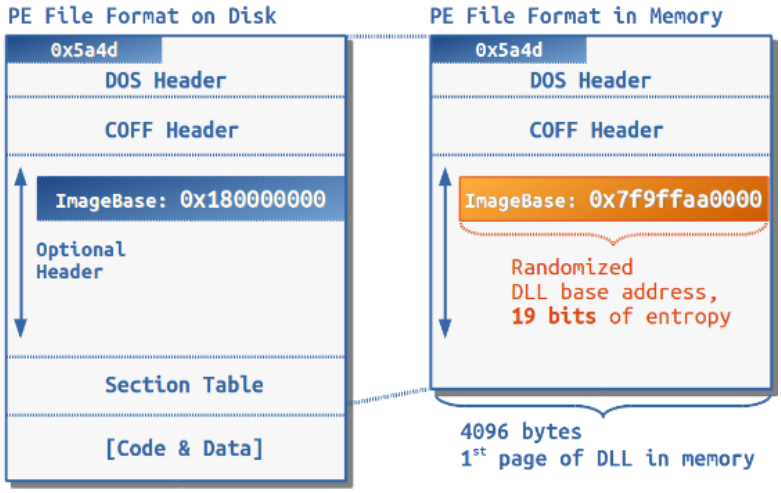

Figure 5: the very first page of a PE image on disk and in memory.

When looking at the first page of memory mapped from a PE image on disk we notice that the page is easy to predict (assuming a certain .exe or .dll image) except for the `ImageBase` field in the Optional Header. This field contains a static value on disk but in memory the value is modified to contain the run-time randomized base address of that specific PE image.

We can therefore use this page to guess the randomized base address in co-located Windows based VMs. Let's assume we know the base address, constructing this page and detecting its existence will confirm that the specific executable PE image mapped in memory has the specific base address we put into `ImageBase` in one of the co-located VMs. Of course we don't know the base address but detecting the existence allows us to infer the base address. At this point the only thing we know is that the base address is within a certain range of values (e.g., 2^{19} for Windows x86_64 systems).

Other pages might be used as well, e.g., code pages that just contain base relocations. In this case constructing a guess requires a bit more effort but leads to the same effect.

3.2.2 Suitable pages in Linux

Linux seems to have pages that fulfill the described properties. Although almost all ELF images contain Position-Independent Code we found, e.g., read-only pages mapped from the `libc` binary image to the process' memory that contained base relocations. We verified this on an Ubuntu 14.04 x86_64 system. Although only some of these pages existed, the attack actually just requires one of these pages to work.

Because of the higher ASLR entropy for ELF shared libraries and the fact that every process has its own layout, attacking a Linux system requires generally more time and yields less reliable results than targeting a Windows system.

3.3 Automated exploitation and handling of noise

Given the existence of suitable pages to attack a certain VM we now need to automatically and reliably brute-force the Random Base Address (RBA) through the memory deduplication side-channel.

CAIN operates in these subsequent three phases in order to find the RBAs:

1. Adaptive sleep time detection
2. Filtering (1 to many rounds)
3. Verification (1 to many rounds)

To automate the attack, the wait time for the underlying memory deduplication implementation to merge pages has to be estimated. This wait time depends on the memory deduplication implementation and its configuration. We implemented a way to test the required "time to wait" by creating a lot of pages that are pairwise equal. This generates a lot of sharing opportunities. Then, after waiting a certain amount of time, writing to these pages and detecting if they were effectively merged will indicate if the tested wait time was sufficient or not. If the tested time was to short, repeating this while increasing the wait time adaptively until the detection rate is high enough, will finally reveal a suitable time window for the attacker to wait. This is then used as the sleep time in between attack rounds.

Next, the actual brute-force attack can be initiated. We use the suitable page to create a large buffer of pages each containing the same content except for the base address value to be brute-forced. At the beginning we eliminate as many guesses with as little memory as possible by trying to detect which page is actually shared after waiting for the previously probed sleep time. Bruteforcing the entire ASLR entropy space requires a lot of memory. As we need one page per guess, this step would at least require the creation of 2^{19} * 4096 KB (pagesize) = 2 GB to attack a 64bit Windows or 2^{28} * 4096 KB (pagesize) = 1024 GB to attack a 64bit Linux system. Depending on the available memory, filtering can be done in one round or multiple rounds by splitting up the search space.

After having performed one or many rounds of filtering the potential number of candidates will be significantly reduced. There are still false positives that need to be filtered out by performing subsequent verification rounds. A verification round is the same as a filtering round with the difference that now in-between guesses we place random content pages. As we know that the random content pages are not shared we can thus better detect potentially shared pages. The reason why we use filtering rounds (no random pages in-between) at the beginning is simply a matter of available memory. 2^{19} (Windows 64bit) or 2^{28} (Linux 64bit) guesses just require too much memory in the beginning.

To actually detect if a page is shared or not we write to the first byte of that page (after having waited for some time) and we compare its write time in cycles (measured with the read time stamp counter instruction, `rdtsc`) with the write time of the adjacent pages.

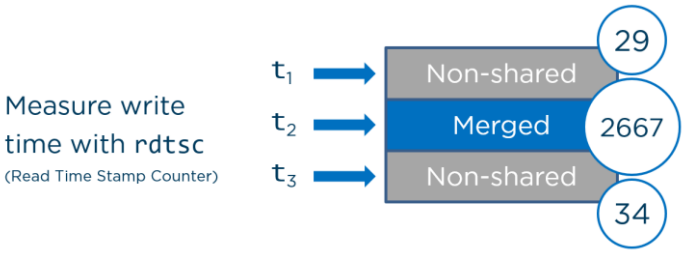

Figure 6: detection of merged pages is done using heuristics to compare write times of adjacent pages with the potentially merged page in-between.

We can assume that the adjacent pages are not shared as the probability that two adjacent guesses correspond to the randomized base addresses of the very same image in two different co-located victim VMs is rather unlikely (during filtering). During verification rounds the adjacent pages contain random content as we placed these pages in-between.

We use different heuristics to detect if a page is merged. For instance, we check if t_2 is greater than two times the average of t_1 and t_3 (which is $t_1 + t_3$). Based on observations we made we implemented some additional heuristics in our Proof-of-Concept that helped us handle noise. These heuristics might depend on the hardware configuration and the underlying VMM but we were able to use the same heuristics on different hardware configurations. A more determined attacker could also optimize detection of a merged page by using better heuristics.

Please refer to the WOOT'15 paper (CAIN: Silently Breaking ASLR in the Cloud 2015) for more details on the heuristics used and the different attack phases.

3.4 Evaluation

We have implemented a Proof-of-Concept exploit and evaluated it on the following system:

- A dual CPU blade server with two AMD Opteron 6272 CPUs (16 cores each) and 32GB of RAM.
- The system runs on Ubuntu Server 14.04.2 LTS x86_64 (Linux Kernel 3.16.0), using the standard KVM in its default configuration as VMM.
- Except for the `sleep_millisecs` (`/sys/kernel/mm/ksm/sleep_millisecs`) parameter that tells KSM how many milliseconds it must sleep in-between scan cycles, we kept all the parameters at their default values.
- All the VMs are configured with 4 virtual CPUs and 4GB of memory. We run one attacker VM with Ubuntu Linux 14.04, and up to 7 VMs with Windows Server 2012 Datacenter (6.2.9200).

Figure 7 shows the attack times for different `sleep_millisecs` configurations. We used the attacker VM to attack one victim VM. The `sleep_millisecs` configuration tells KSM how long it has to wait in-between scan cycles. The lower the value the faster KSM is. The default value of `sleep_millisecs` on our system (Ubuntu Linux 14.04) was `200`.

The dotted lines in Figure 7 denote estimates the straight lines are attacks we effectively performed. As expected the memory deduplication speed has a direct influence on the attack time required. The individual points denote the end of a filtering (the first one) or verification round. Every attack starts at time 0 and with the initial entropy of 19 (as we attack a 64bit Windows system). After the first round entropy is already reduced by almost 7 bits (in average) i.e. from the initial 2^{19} (524'288) potential base addresses to several thousand remaining candidates (2^{12} = 4096). This is independent of the `sleep_millisecs` configuration. The only influence `sleep_millisecs` has is on the required sleep time. As the sleep time between rounds is longer the overall attack time will increase as well. The attacker has to wait longer till he can safely assume that all the potential sharing opportunities were deduplicated.

For the default configuration the attack took less than 5 hours.

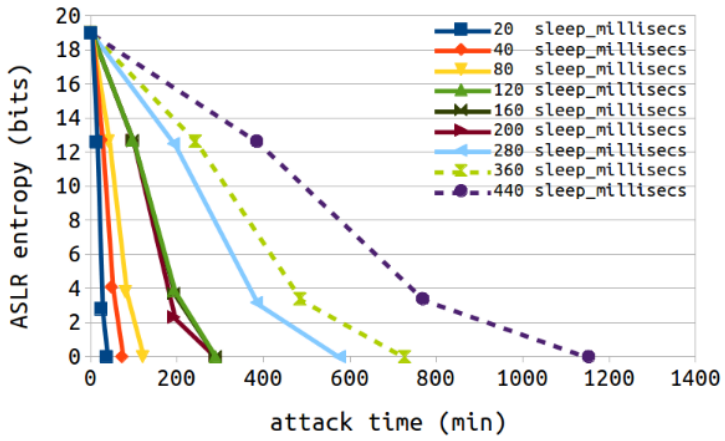

Figure 7: attacking one single Windows victim VM, overall attack time to infer the ntdll.dll base address. (sleep_millisecs = 200 is default)

Next we wanted to see how the overall attack time behaves when attacking multiple victim VMs. All victims run the same Windows operating system. We used `sleep_millisecs` = 20 to reduce overall attack time for our experiments. As Figure 8 shows the attack takes longer the more victims run on the same physical hardware. As the total amount of memory used by all the VMs increases the memory deduplication scheme needs more time to find all the sharing opportunities. This is why the sleep time estimates increase as well which directly influences the overall attack time. For more than 1 victim there is also an additional verification round at the end. This is required because we need to verify if the final set does not change. In case of a single victim we can optimize the attack as we can assume that the last single base address is the right one. This of course assumes that the attacker knows that there is one victim VM running. Interestingly the attack against 6 victims took less time than attacking 5 victims. This is due to the sleep time estimate which happened to be higher. As many factors might influence the estimated sleep time required this might look different in subsequent rounds. We think there is also a lot of improvement potential in how we estimate the required sleep time such that the overall attack time can be further reduced by a more determined attacker.

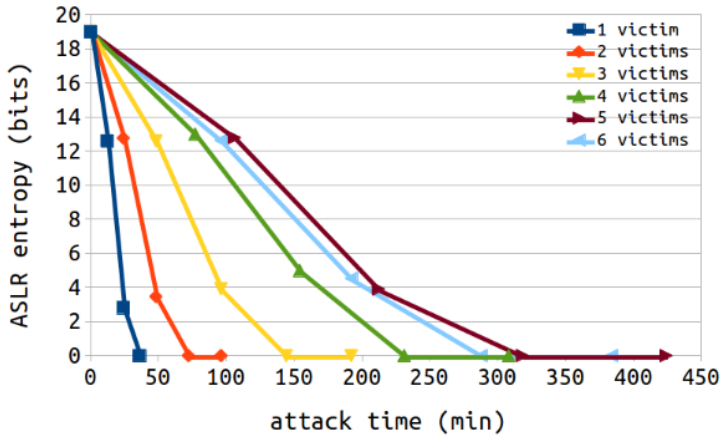

Figure 8: attacking several Windows victim VMs, overall attack time to infer the ntdll.dll base addresses. sleep_millisecs = 20

3.5 Attacking Weak ASLR Implementations

Our evaluation focused on modern strong ASLR implementations and assumes no weaknesses. Implementations in older systems, and especially on the 32 bit x86 architecture, usually offer significantly less entropy. Windows 7 (32 and 64 bit) and 32 bit Windows 8 have only eight bits of entropy for executables and DLLs. `mmap()` on a 32 bit Linux offers only eight bits of entropy as well. Naturally, 32 bit architectures with smaller virtual address spaces cannot offer high entropy as the virtual address space is limited. Even conceptually higher entropy ASLR is prone to implementation weaknesses as recently shown (AMD bulldozer linux ASLR weakness: Reducing entropy by 87.5% 2015). In cases where entropy is reduced, our attack's feasibility and time requirements improve from an attacker's perspective. Our evaluation shows that the memory deduplication configuration and the number of victim VMs have a direct effect on the required attack time. The number of

entropy bits, however, has a more significant role on the total attack time as they increase the number of filtering rounds. We therefore want to emphasize that weak ASLR implementations amplify the issue and the severity of our attack.

3.6 Post-CAIN exploitation

After having performed the CAIN attack the attacker will know the base addresses of the DLLs of interest (e.g., ntdll.dll). From here, the attacker can construct a code-reuse attack to target the specific vulnerable application running on the co-located VM. Depending on the victim VM system and the number of same systems running on the other VMs the attacker either gets one exact base address (one single Windows victim) or several base addresses (many Windows victims or a Linux victim where every process has its own address-space layout).

If the exact base address was inferred the attacker just uses that base address. If the attack results in many base addresses the attacker can either try out all of them in the hope that the address-space layout is not re-randomized (Windows targets or Linux where an attack attempt crashes only a thread and not the entire process) or he will have to further map the base addresses to the specific vulnerable process. This can potentially be done by finding pages that are specific to the vulnerable process and which have references to the now de-randomized executable image. These pages can be reconstructed with the different base addresses obtained and the same attack will then reveal the right base address for a specific vulnerable process. This of course assumes the existence of such pages that we did not further investigate.

Another possibility would be to just attack executable images (e.g., DLLs) that are known to only exist in the specific vulnerable process. If the gadgets found in that image are enough to perform a code-reuse attack the mapping becomes obsolete.

3.7 Affected VMMs

We verified the issue on KVM (Ubuntu Server 14.04.2 with Linux Kernel 3.16.1). But most probably other VMMs are also affected. Table 1 provides an overview of vendors with notes on the memory deduplication implementation along with the information if the vendor's product is potentially affected (cert.org assessment and our assessment). It's also important to note that some VMMs have same content page based memory deduplication but it is not enabled by default (VMware and Oracle VirtualBox).

Vendor	Memdedup	cert.org[1]	We think	Enabled by default
Linux KVM	KSM	Affected	Yes	n.a.
Ubuntu (KVM)	KSM	-	Yes	Yes
Red Hat (KVM)	KSM	Affected	Yes	Yes
Parallels	-	Affected	Not sure	-
Microsoft Hyper-V	None	Not Affected	Not Affected	n.a.
Xen	None (yet?)	Not Affected	Not Affected	n.a.
Oracle VirtualBox	PageFusion	Unknown	Probably	No
VMware	TPS	Unknown	Probably	No

Table 1: vendors affected by the issue.

3.8 Affected Operating Systems

We verified the attack targeting a Windows Sever 2012 Datacenter (6.2.9200 Build 9200) system. We also verified the existence of a suitable page to attack Ubuntu Linux 14.04 x86_64 (pages found within `libc`).

We assume that most Windows systems (32bit and 64bit) are attackable as the page we use in our PoC most probably also exists in other Windows versions. We also assume that Linux based systems are attackable although the attack requires much more time and the gain for the attacker is not as promising as when attacking a Windows system (every process has its own randomized address-space). Other operating systems using PE or ELF based executable images might be affected as well but this needs further investigation.

[1] https://www.kb.cert.org/vuls/id/935424

4 Mitigations

As the problem cannot be attributed to one specific vulnerability in the VMM or system the issue cannot just “be fixed”. The described issue is the result of a VMM feature (memory deduplication), the implementation of a system hardening technique (ASLR) and the particularities of process creation and loading. Therefore, mitigations can be implemented at all these layers. We see the following mitigations:

VMM layer: Deactivation of memory deduplication

Deactivating memory deduplication will effectively mitigate all attack vectors including attacks that just try to leak confidential data without breaking ASLR. This measure unfortunately eliminates all the highly appreciated benefits of memory deduplication.

VMM layer: Attack detection

Instead of preventing the attack directly, the VMM can observe the guest VM and detect an ongoing attack based on memory creation and page fault behavior. We do not propose any specific heuristic but we suggest the concept of detecting the attack instead of preventing it.

ASLR layer: Increase ASLR entropy

One of the factors making the attack feasible is the limited ASLR entropy of code regions. We suggest to further increase ASLR entropy to not only mitigate the attack described here, but to continue making ASLR more effective.

Process layer: More entropy in sensitive memory pages

The pages we use in the attack are good candidates because their entropy consists solely of the ASLR entropy i.e., we can reliably construct the page once a base address is known or guessed. Increasing entropy in these pages or making sure that no such pages exist can mitigate the issue (e.g., increase entropy by writing some random values into the first PE image page).

5 Conclusion

We described CAIN, Cross-VM Address-Space Layout INtrospection an attack vector against memory deduplication in Virtual Machine Monitors. The issue is made possible as a result of different circumstances, notably, the memory deduplication side-channel, the existence of certain pages to guess ASLR base addresses and the limited entropy found in modern ASLR implementations for code images. Although the attack focuses on leaking ASLR base addresses of code locations other sensitive information might be leaked and brute-forced as well.

There is currently no real mitigation to this issue except for disabling memory deduplication in deployments that are sensitive to the threat scenario described in this white paper. We also think that there is a lot of potential to further improve the attack and to cross-VM brute-force other sensitive information.

6 References

Antonio Barresi, Kaveh Razavi, Mathias Payer, Thomas R. Gross. 2015. "CAIN: Silently Breaking ASLR in the Cloud." *9th USENIX Workshop on Offensive Technologies.* Washington DC: USENIX.

Hector Marco, Ismael Ripoll. 2015. *AMD bulldozer linux ASLR weakness: Reducing entropy by 87.5%.* http://hmarco.org/bugs/AMD-Bulldozer-linux-ASLR-weakness-reducing-mmaped-files-by-eight.html.

Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. "Memory Deduplication As a Threat to the Guest OS." *Proceedings of the 4th European Workshop on System Security.* EUROSEC.