CAIN: Silently Breaking ASLR in the Cloud

Antonio Barresi
ETH Zurich

Kaveh Razavi
VU University Amsterdam

Mathias Payer
Purdue University

Thomas R. Gross
ETH Zurich

Abstract

Modern systems rely on Address-Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) to protect software against memory corruption vulnerabilities. The security of ASLR depends on randomizing regions in memory which can be broken by leaking addresses. While information leaks are common for client applications, server software has been hardened to reduce such information leaks.

Memory deduplication is a common feature of Virtual Machine Monitors (VMMs) that reduces the memory footprint and increases the cost-effectiveness of virtual machines (VMs) running on the same host. Memory pages with the same content are merged into one read-only memory page. Writing to these pages is expensive due to page faults caused by the memory protection, and this cost can be used by an attacker as a side-channel to detect whether a page has been shared. Leveraging this memory side-channel, we craft an attack that leaks the address-space layouts of the neighboring VMs, and hence, defeats ASLR. Our proof-of-concept exploit, CAIN (Cross-VM ASL INtrospection) defeats ASLR of a 64-bit Windows Server 2012 victim VM in less than 5 hours (for 64-bit Linux victims the attack takes several days). Further, we show that CAIN reliably defeats ASLR, regardless of the number of victim VMs or the system load.

1 Introduction

Virtualizing at machine boundaries provides strong isolation guarantees. Public Infrastructure-as-a-Service (IaaS) clouds leverage this isolation to securely multiplex physical resources between different tenants, each running their own customized operating system (OS) and applications for their unique needs. Running multiple instances of the same type of OS, often with similar software stacks, leads to multiple copies of the same data on memory and disk. Techniques have been developed to reduce the duplication of data (i.e., deduplication). Duplicate pages (for memory) or duplicate file system blocks (for disks) are shared among VMs. Deduplication directly reduces the required resources for VMs, allowing to run more VMs on the same hardware configuration.

While deduplication reduces resource requirements of VMs, it also introduces a side-channel that allows other VMs to probe for the existence of specific memory pages. This attack vector operates by crafting a memory page with equal contents to a memory page that exists on neighboring VMs (victims). If this page indeed exists on a victim VM, then the deduplication process *merges* this crafted page with the page residing on the victim's memory. This page is then marked as read-only and gets shared by the attacker and the victim. Measuring write times allows the detection of merged pages. If the merge has happened, writing to the page takes much longer because of the page-fault on the read-only shared page, and the resulting CoW (Copy-on-Write) for creating a private copy of the modified page. This attack has previously been used to leak binary versions of software [20]. Further, this attack has also been used to detect the location of sensitive data structures in a victim's memory [9], leading to successful AES [5] key retrieval.

We show that this side-channel attack allows attackers to break the core defense mechanism, Address-Space Layout Randomization (ASLR), of popular operating systems against memory-related security vulnerabilities. Common systems like Windows or Linux use Data Execution Prevention (DEP) to prohibit execution of data on the stack or heap regions of a process. The only regions that remain executable are program code and its (dynamically linked) libraries. ASLR randomizes the locations of these regions, making it probabilistically unlikely for an attacker to gain code execution or perform a code-reuse attack using a memory error, such as a stack or heap overflow. ASLR and DEP heavily rely on each other: Without DEP, an attacker can inject new executable code into the address space of a process and without ASLR, the attacker can mount code-reuse attacks through return-oriented programming (ROP) [15, 19] or other related

techniques [2, 4, 3, 21]. Hence, breaking only one of the two defenses is enough to compromise the main line of defense against memory-error attacks. While information leaks are abundant on clients, e.g., through a memory corruption vulnerability and client-side scripting (like JavaScript), they are much harder to find on servers.

We show that it is possible to "leak" (or recover) the randomized base addresses (RBAs) of libraries and executables loaded within user space processes in neighboring VMs by leveraging the memory page deduplication side-channel.

Using KVM [13], an off-the-shelf VM monitor (VMM) that employs memory deduplication, we show the effectiveness of our attack in a realistic setting. Our proof of concept exploit, **CAIN**, breaks ASLR of a 64 bit Windows Server 2012 victim VM in less than 5 hours using default configurations. We estimate a successful attack to take several days on a recent 64 bit Linux system, depending on how much memory is available (e.g., with 4GB of memory we estimate an attack time of around 18 days). Using a webserver scenario, we show that CAIN can reliably break ASLR even when the VM is under various degrees of system load. Further, we show that breaking ASLR of multiple neighboring VMs is also feasible and increasing the number of VMs running on the same physical hardware does mainly affect the required attack time. These results show that memory deduplication in a cloud setting compromises the current defense against memory-error attacks on popular systems. We will discuss mitigations against this attack, but we believe that additional research in this area is necessary due to the increasing popularity of IaaS clouds [1].

2 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 (e.g., anyone with a credit card). Weak and vulnerable systems running on the same physical hardware pose a security risk for all their co-located neighbors. In this paper, we describe an attack that can be used to break ASLR of neighboring VMs. This attack abuses the sharing of physical memory resources between multiple VMs. Before delving into the low-level details of the attack, we give a brief overview of ASLR in modern systems and we discuss how memory sharing is done in KVM, one of the most popular VM monitors. While we focus on KVM in this paper, the same attack vector exists on other VM monitors as well [14, 22].

2.1 Address Space Layout Randomization

Windows Microsoft recently introduced improvements to ASLR with Windows 8 [12]. Microsoft Windows x86_64 OSes have High Entropy ASLR providing up to 33 bits of entropy for stacks and 24 bits for heaps [12]. Despite these improvements, base addresses of executables (17 bits) or DLLs (19 bits) still have limited amount of entropy. Another particularity of Windows ASLR is its per-system randomization of DLL base addresses. As PE files usually contain many relocations, ASLR is only done once per system boot. Thus, most DLLs (e.g., `ntdll.dll`) share the same virtual address between all processes on the same system. This setup allows Windows to keep only one copy of the DLL in memory, reducing memory consumption.

Linux In Linux based systems, ASLR for shared libraries is provided by `mmap()`. The user-space dynamic linker and loader, `ld-linux`, loads ELF dynamic shared objects by mapping the binary images from disk over `mmap()` into the process' virtual address space. `mmap()` randomizes the addresses of mappings over `mmap_rnd()` [8]. For 32 bit x86 Linux systems, `mmap_rnd()` provides 8 bits of entropy and for 64 bit x86 28 bits. Linux based systems deploy PIC (Position-Independent Code) for shared libraries. With PIC, base addresses can still be randomized individually for each process without having to duplicate the executable pages. However, the main executables are often prelinked for speed (e.g., on 32bit x86). These executable images are loaded at well-known addresses. Recent x86_64 Linux Distributions have started to ship executables as PIC, which allows full ASLR (randomizing every region of a process). On 32 bit x86 systems, PIC still incurs major performance overhead [17], making full ASLR costly.

2.2 Kernel Same-page Merging (KSM)

KSM is the memory deduplication implementation of the Linux kernel which is used by KVM. KSM's behavior can be configured over `sysfs` (`/sys/kernel/mm/ksm/`). Over `run` KSM can be enabled `'1'` or disabled `'0'`. The values in `sleep_millisecs` and `pages_to_scan` define how often KSM will be invoked (e.g., `sleep_millisecs` = 200, 5 times per second) and how many pages KSM scans per invocation (e.g., `pages_to_scan` = 100, 100 pages) which would result in 500 pages per second. Depending on the workload, different values are optimal [18].

When KSM is active, a kernel thread regularly inspects memory pages that are subject to memory deduplication. When two equal pages are found, they are merged and both page table entries will from now on point to the same physical page. The page is marked as read only and a write to it triggers a CoW operation.

Internally KSM uses two red-black trees with page hashes, a stable tree that holds all shared pages, and an unstable tree that keeps track of all potential candidates. When scanning, KSM looks at one page at a time. If there

is a match in the stable tree, the page is merged. If not, and the page was not modified since the last scan, the page is put into the unstable tree. If a page with the same content already exists in the unstable tree, the pages are merged as well. The unstable tree is then purged when all pages were scanned, and the next iteration starts.

3 Breaking ASLR with CAIN

Our proof-of-concept implementation of the attack called CAIN (Cross-VM ASL INtrospection) is implemented as a C++ program that runs as an unprivileged user space process on Linux. We make the following assumptions in the design of CAIN:

- There is a measurable and noticeable difference in the time a memory write operation takes depending on whether the page at the write address is shared or not.
- Noise will not eliminate these differences, i.e., when write operations take generally longer, writing to a CoW page will still be noticeable.
- There is a duration M which is smaller than any CoW operation. Depending on the architecture and the OS, we can assume that a CoW will take at least M cycles.
- The probability that two or more randomized base addresses (RBAs) over all running VMs are exactly the same, or with adjacent addresses is negligible.
- The probability that the same false positive is subsequently affected by noise multiple times such that it will always be identified as a correct guess is negligible.

Only the first assumption must hold for the attack to succeed. The others are important to merely increase the reliability and speed of CAIN.

3.1 Attack Overview

We designed the attack to be automated and independent of the VMM, the VMM's underlying memory deduplication implementation, and the memory architecture. Conceptually, we brute force all possible randomized base addresses (RBAs) by taking advantage of time differences in memory write operations to deduplicated memory pages that are marked as read-only. These time differences provide us with a side-channel that reveals if a page is shared between VMs or not. To get correct results, the attacker must wait a certain amount of time for the crafted page to get merged. The right amount of time is not known upfront and the attack becomes unsuccessful if the attacker does not wait until after the merge. Furthermore, the side-channel contains noise, resulting in false positives and false negatives.

To brute force the RBA, we construct guesses that consist of entire pages that we know must exist in the victim VMs. The entropy of these pages should depend entirely on the RBA. We can therefore create all possible pages with one base address guess per page and then derive its existence over the memory deduplication side-channel, effectively derandomizing ASLR.

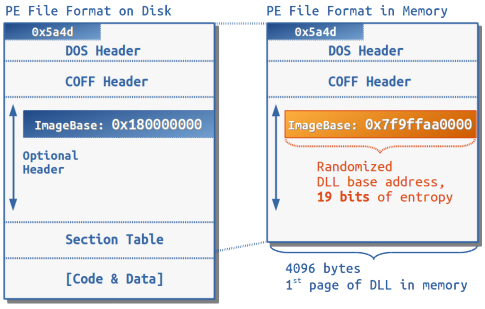

Figure 1: PE file format on disk and in memory.

There are two main challenges. First, we need to construct pages that are suitable for the attack. We discuss how to construct these pages for Windows in Section 3.2, and for Linux in Section 3.3. Second, the attack must be automated and reliable, requiring handling noise that is present within the memory deduplication side-channel. To address these challenges, CAIN runs in three distinct phases that we discuss in Section 3.4.

3.2 Attacking Windows

To attack ASLR on Windows, we need a page that contains the ASLR base address of a specific library, i.e., the page's entropy has to be exactly the entropy of ASLR. In addition, the page should be mostly static or, ideally, read-only on the victim VM such that it is reliably merged with our guess (given enough time). Data pages on the stack or heap are therefore not suitable as they are (i) highly dynamic and (ii) contain high entropy. When examining a Windows process in memory, we see that similar to other systems, executable file images (PE executables and DLLs) are memory mapped into the process' address space. These mappings usually contain code pages, data pages, and information required by the loader to setup the memory layout of the process and to perform all the required relocations. Examining the very first page of a PE file (e.g., `ntdll.dll`) reveals an interesting property: it is entirely static except for the ImageBase field in the optional header.

As illustrated in Figure 1 when a PE file is mapped into memory during process creation, the first page contains the DOS, COFF, and Optional Header. Within the Optional Header, there is a field `ImageBase` that contains `0x180000000` on the disk image of the file. But once loaded in memory, this field will be updated with the load address of the PE file in memory, i.e., the RBA. We can easily extract this first page given any DLL or executable image. Given this page, we can now brute force the RBA with the technique described in Section 3.1.

Note that when running multiple Windows VMs with the exact same OS version, the first page of, e.g., `ntdll.dll` will be exactly the same (except for the different system-wide RBA). By detecting all RBAs, the

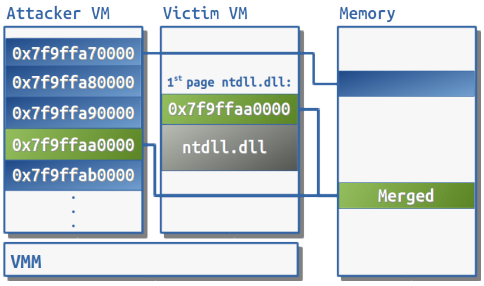

Figure 2: Attacker memory when attacking Windows.

attacker can reveal the exact number of Windows VMs running on the same physical machine.

To brute force the RBA of a specific DLL, we reconstruct the exact first page of that DLL in the attacker's memory and adapt the RBA within the `ImageBase` field. Figure 2 shows how attacker memory looks like. The highlighted pages contain the same contents (the attacker's guess and the first page of the target DLL in the victim's memory), which is then merged at some point in time. This is the attacker's goal, and writing to this page triggers a CoW, which allows the attacker to detect the existence of the same page and the exact RBA of the victim VM.

3.3 Attacking Linux

Attacking Linux based VMs requires more effort and is less effective due to the different ASLR design. The 64 bit Linux ASLR implementation uses 28 bits of entropy for `mmap()` [8].

Mapped code pages of ELF files under Linux are mostly identical due to PIC (Position-Independent Code) support (at least for ELF shared objects). Other pages like data pages are not suitable under Linux because of their high entropy characteristics. Looking closer at the ELF file mappings of Linux processes, we can see mappings with different protection flags. The `GNU_RELRO` segment provides some pages that are mapped into a process as `r--`. Before the pages are actually marked read-only relocations are performed. Many `R_X86_64_RELATIVE` relocations point exactly to sections in the `GNU_RELRO` segment. These relocation entries instruct the dynamic linker to add the base address of the ELF binary image to the value found at these locations (originally loaded from the ELF disk image). This is exactly what the attacker needs as it is now possible to reconstruct these pages, given the binary image and a guess for the RBA of that specific ELF shared object (e.g., the `libc.so`).

Once such a page is found, the attacker mounts the same attack as for Windows by taking advantage of the memory deduplication side-channel. However, because of the additional entropy bits in Linux for `mmap` (28 bits compared to 19 bits on Windows), the attacker either needs more memory or more time to brute force this significantly larger space.

3.4 Attack Phases

CAIN operates in these three phases in order to find the RBAs: (i) sleep time detection, (ii) filtering, and (iii) verification.

To automate the attack wait time for the underlying memory deduplication implementation to merge pages has to be estimated. Section 3.5 describes this step in more detail. Next, we eliminate as many guesses with as little memory as possible. 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 $2^{19} \times \texttt{PAGESIZE} = 2\,\text{GB}$ to attack a 64 bit Windows or $2^{28} \times \texttt{PAGESIZE} = 1024\,\text{GB}$ to attack a 64 bit Linux. Depending on the available memory, filtering can be done in one round (at least against Windows) or multiple rounds by splitting up the search space. More on filtering can be found in Section 3.6. The last step consists of verifying the remaining guesses in a more reliable way and thus eliminating false positives, by using more memory per guess. This is described in Section 3.7.

3.5 Sleep Time Detection

To perform a fully automated attack, CAIN estimates the amount of time it waits (the attack's sleep time per round) for the guess (crafted page) to be merged with the target page. This is important as waiting too little would miss a correct guess and waiting too long would unnecessarily extend attack time. The required sleep time for the memory deduplication implementation to merge pages varies and depends on several factors. First, it depends on the algorithms and data structures used by the implementation. Second, run-time factors like total amount of memory subject to deduplication, CPU load, and other characteristics can influence the required sleep time. We consider the memory deduplication implementation to be a blackbox and propose a simple iterative approach to estimate the required sleep time.

We first allocate a buffer and within it we generate a large number of random pages. We then copy every second page of the first half of the buffer to the same location in the second half of the buffer. The buffer now consists of N pairs of pages (that contain random bytes) with the same content (`R-1, R-2,..., R-N`) and unique random pages between these pages. We now try a first estimate and wait, e.g., 10min. After that time, we start touching all the pages within the first half of the buffer by moving a pointer over the first byte of every page and write to it. During this sweep, we measure write times and when the write time of every second page is significantly longer than the ones of the adjacent random pages we mark the page as potentially merged. The page was either merged or accidentally detected as merged because of noise. Depending on the detection rate, we do the following:

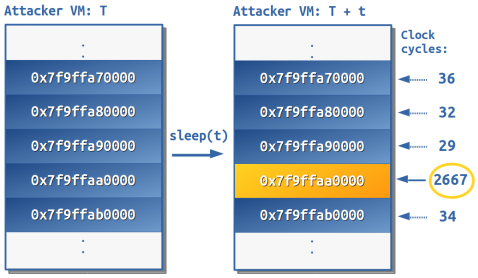

Figure 3: Attacker memory during filtering.

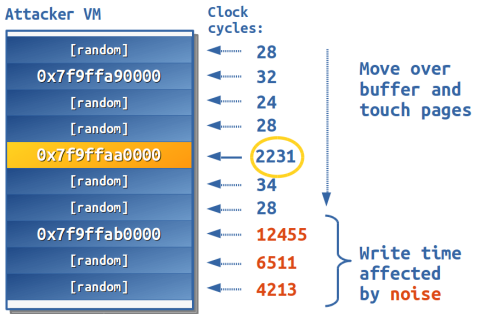

Figure 4: Attacker memory during verification.

- if detection rate is greater than 95%, we add a safety margin of 20% to the last sleep time estimate and return this as the result;
- if detection rate is less than 50%, we double the last time estimate, and rerun the algorithm with the new (twice as large) estimate;
- if detection rate is in between 50% and 95%, we multiply the last time estimate with the inverse of the detection rate, and rerun the algorithm with the new estimate.

This simple search will return a sleep time that has a high detection rate. Depending on the memory deduplication implementation this technique might not produce optimal results. In this case the attacker can either come up with a better way to estimate sleep time, or just use a conservatively long wait time (e.g., 10 hours).

3.6 Filtering

Depending on the number of entropy bits and the memory available for the attack, R filtering rounds with $R = 2^{(\texttt{Entropy} - \log_2(\texttt{Memory}/\texttt{PAGESIZE}))}$ are necessary. Assuming the attacker has 2 GB of available memory, attacking Windows with 19 bits of ASLR entropy requires only one filtering round because all the crafted guesses fit into one memory buffer of 2GB.

During filtering we construct the memory layout shown in Figure 3. We fill all the available memory with guess pages. For Windows this is the first page of the DLL that we want to derandomize. For all the base address guesses, we create a page and adapt the `ImageBase` field of these pages, with one guess for each base address. We then wait for the estimated sleep time.

Next, we touch all pages by writing to each page's first byte, measuring how long each write takes. If we encounter a page with a significantly higher write time than its neighbors, we mark it as a potential candidate, admitting it to the next phase. We define significantly higher write time as at least two times greater than the average write time of the adjacent pages. In Figure 3, the highlighted page has a measured write access time of 2,667 cycles, whereas the neighbor pages have an average access time of $(29+34)/2 = 31.5$ cycles. We found a threshold factor of two to be very conservative. Write times that trigger a CoW often take much longer than non-CoW write times. However, to avoid false negatives the threshold factor should not be too large. Furthermore, we also check the write time to be above a certain minimal write time (M) for CoW operations to avoid false positives. In our implementation we use 1,000 cycles, but this value can be adapted according to the underlying architecture and system. The exact same parameters worked well in our tests on a Workstation setup (Intel i7-4770 CPU, 8 cores, 8 GB memory) and a Server setup (AMD Opteron 6272 Dual CPU, 32 cores, 32 GB memory). An attacker can select alternative values if our heuristics do not work.

Last, we check if the adjacent write times do not diverge significantly from each other. We verify that the differences between write times is not greater than a third of the larger write time of both. We noticed that when write access times are affected by noise, they start to diverge. This additional criteria helps us filter out such cases. These detection heuristics may cause false negatives and false positives. False negatives happen when noise results in a write of an adjacent page to a page with the correct guess. The probability of this is low, but if it happens, CAIN terminates without finding a result. The attacker can then run CAIN again. In our experiments, false negatives occurred rarely (usually due to short wait times). False positives are, however, more common due to noise. We deal with them in the next phase of the attack.

At the end of all filtering rounds, we have a set of potential guesses that contain all the RBAs. Note that every guess is checked exactly once during filtering.

3.7 Verification

We now have to verify each of the remaining guesses to remove false positives. Because most candidates are already eliminated, we now have more memory at our disposal. We again craft pages with the corresponding guesses. This time, we allocate random pages in between these pages. Figure 4 shows the memory layout during verification rounds.

After waiting the estimated sleep time, we try again to detect all merged pages. Candidates that are not detected as being shared are removed. The number of rounds depends on the results. We continue verifying candidates as long as the set of candidates between verification rounds differ. As soon as the set between verification rounds does not change, or there is only one candidate left, we stop the

attack. This convergence usually happens after 2 to 3 verification rounds. The main purpose of verification rounds is to reliably filter out false positives caused by noise. We can assume that if a page is falsely detected as a candidate because of noise, the page will with high probability not be affected by noise again in the next verification round. The attack completes after the last verification round, and all the remaining guesses are reported as hits.

3.8 Post-CAIN Attack

Depending on the victim OS and the number of victims, at the end of the CAIN phase the attacker will have one or a set of derandomized base addresses for a specific library module like e.g., `ntdll.dll` or `libc.so`. In the case of multiple Windows victims or one or more Linux victims the base addresses are not linked to specific VMs (Windows and Linux) or processes (Linux). First, note that code-reuse attacks do not require more code or gadgets than usually found in one large library [19]. It is therefore sufficient to derandomize the base address of `ntdll.dll` (Windows) or `libc.so` (Linux) to perform a code-reuse attack and thus fully bypass ASLR. Second, in most cases attacked processes or threads will restart after a failed attack attempt. Thus trying out all derandomized base addresses will work in case the memory layout is not re-randomized. Even in the presence of ASLR brute-force protections that limit the number of process restarts, since the number of required trials is very low, these protections are likely not effective. Third, attackers can also just derandomize a library or the executable itself that is known to be used only in the targeted VM or process thereby eliminating the need to guess at all. An extension of CAIN further uses the derandomized base addresses to link the base addresses to a specific VM or process by constructing pages that contain values derived from the derandomized base address of e.g., `ntdll.dll` or `libc.so` and that are known to only exist in a specific VM or process. With the knowledge of a library's base address and the software running on the host, CAIN constructs targeted pages that are only used by that software to identify the correct base address. Possible candidates are pages containing the Import Address Table (IAT) in Windows or the PLT (Process Linkage Table) entries in the GOT (Global Offset Table) `.got.plt` in Linux. These pages contain pointers to libraries, allowing CAIN to further link a base address to a process or even a VM in case the process is known to only exist in one VM.

4 Evaluation

We evaluate CAIN attacks in different scenarios. All experiments run on a dual CPU blade server with two AMD Opteron 6272 CPUs (16 cores each) and 32 GB of RAM. The system runs on Ubuntu Server 14.04.2 LTS x86_64 (Linux Kernel 3.16.0), using the standard KVM software in its default configuration as VMM. Except for the `sleep_millisecs` (`/sys/kernel/mm/ksm/sleep_millisecs`) parameter that instructs KSM how many milliseconds it must sleep in-between scan cycles, we keep all the parameters at their default values. All the VMs are configured with 4 virtual CPUs and 4 GB 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).

4.1 Attacking a Single Windows VM

We first evaluate the attack against a single victim. For the default configuration of `sleep_millisecs` (200 milliseconds), a total successful attack takes 288 minutes, i.e., less than 5 hours: Our sleep time estimate is 96 minutes per round, and the attack requires one filtering round and two verification rounds. From the initial 524'288 (19 bits of entropy) possible base addresses 6179 (12.6 bits of entropy) advance to the verification phase. The first verification round eliminates 6172, and the last round eliminates the remaining 6 false positives. We verified that the remaining candidate is the correct RBA of `ntdll.dll` in the victim VM.

Figure 5 shows the attack duration with different values for `sleep_millisecs`. Each dot represents one completed round. After each round the set of base addresses and therefore the ASLR entropy is reduced till the correct base address is found (i.e., ASLR entropy = 0). As expected, the shorter the KSM cycles (more pages are potentially merged within the same amount of time), the faster the attack. The ASLR entropy is effectively reduced from 19 bits to 0 bits, i.e., the exact value. In our experiments, the attack always takes 3 rounds despite the `sleep_millisecs` value. Depending on noise the number of rounds can vary. For `sleep_millisecs` 20 milliseconds, the attack takes only 36 min (12 min per round). Note that our sleep time estimation is not optimal, and an attacker could invest more time and effort to optimize it, resulting in even shorter attack times. This becomes clear when looking at the attack time for 120, 160, and 200 milliseconds. All these configurations result in the same sleep time estimation and thus the exact same overall attack time. A more fine-grained sleep time estimation would probably reduce the attack time for these `sleep_millisecs` configurations. The continuous lines in Figure 5 denote fully performed and verified attacks whereas the dotted lines denote only estimations.

Next, we look at how a system under load reacts to the attack. We use a victim VM running a Internet Information Services (IIS) webserver and a different system to generate web requests with `ab` (the Apache Benchmark tool). The IIS webserver hosts two files. One static html file of size 56 KB and one jpg file of size 1054 KB. The files correspond to average html and jpg file sizes hosted

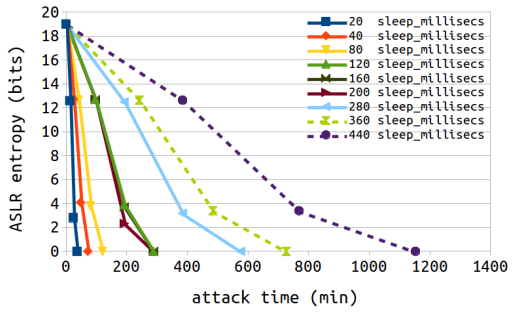

Figure 5: Attacking a single Windows VM.

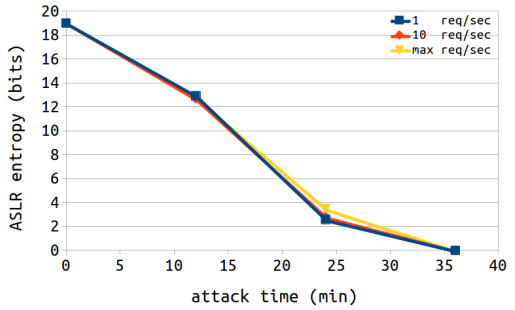

Figure 6: Attacking a Windows VM under load. sleep_ms = 20

by webservers [6]. We use the following configurations:

1. 1 request per second per file ("1 req/sec");
2. 10 requests per second per file ("10 req/sec");
3. 100,000,000 requests with 1,000 concurrent connections kept alive per second per file ("max").

Figure 6 shows that the attack is not influenced by the generated load. We successfully leaked the correct RBA for all load scenarios in 36 minutes per attack. Note that we use `sleep_millisecs` 20 to reduce the required time for testing.

4.2 Attacking Multiple Windows VMs

We run the attack against multiple victim VMs with `sleep_millisecs` 20. The scenarios consist of up to six concurrently running victims. Figure 7 shows that increasing the number of concurrent VMs also increases the required attack time (due to the additional memory which requires longer sleep times). The attack against six running victim VMs takes 384 minutes in total, with one filtering and three verification rounds. Compared to the single victim attack (two verification rounds), the attack requires one additional verification round to make sure the set of candidates converges (last horizontal step where entropy stabilizes). The remaining 6 candidates are the correct `ntdll.dll` RBAs of all running victim VMs. Note that due to the nature of our attack, all RBAs of all co-located VMs are leaked during a single run of our attack. In Figure 7, we assume the attacker targets any of the co-located Windows VMs (i.e., in case of 6 victims, of the 2^{19} = 524288 base addresses 524288 - 5 = 524283 have to be eliminated). In all reported attack

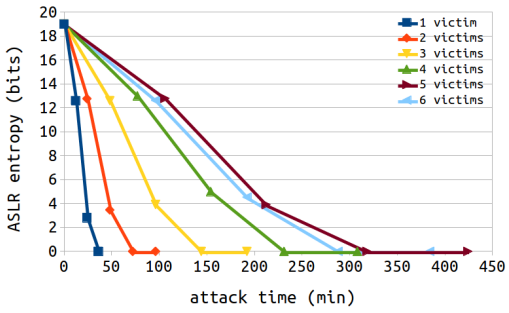

Figure 7: Attacking multiple Windows VMs. sleep_ms = 20

times our attack recovered the RBAs of all co-located Windows VMs (no false negatives and no false positives).

4.3 Attacking a Single Linux VM

We propose using the page described in Section 3.3 to mount attacks against Linux victims. Linux RBAs of shared libraries have higher entropy, resulting in increased memory requirements on the attacker side. We can not expect to be able to allocate $2^{28} * \texttt{PAGESIZE}$ (=1 TB) of memory, but we can divide the attack into multiple filtering rounds. Assuming 4 GB of memory we estimate an attack against a 64 bit Linux VM to require $2^{(28-\log_2(4\text{GB}/4096))} = 256$ filtering rounds. Assuming the average elimination rate for filtering rounds in our experiments, additionally, four verification rounds are required (last one for convergence). The 261 rounds in total with the default `sleep_millisecs` configuration (200 milliseconds i.e., 96 minutes per round) will result in a total estimated attack time of 261×96 minutes $=$ 25056 minutes $=$ 17.4 days. Doubling the available attacker memory will reduce the required attack time by half. With 8 GB of memory, we therefore estimate an attack time of 8.7 days and with 16 GB, 4.35 days. Assuming a Linux victim VM with a reasonable uptime this attack becomes feasible.

4.4 Attacking Real-World Environments

Based on our experimental setup, we believe that CAIN-like attacks are highly feasible in real cloud environments. Our setup uses standard software and server-grade hardware. Still, it is crucial that memory deduplication is enabled within the VMM for our attack. We contacted a number of commercial cloud providers with a questionnaire to investigate the popularity of memory deduplication in real-world settings. Based on the responses we received, it is clear that memory deduplication is indeed used in public cloud settings. Some providers use it as their default configuration and some providers do not use it at all: i.e., they explicitly disable it or their underlying VMM does not support it.

Further, the resources available to an attacker and the number of VMs running have an impact on attack time. As most commercial IaaS offerings have more than 4 GBs of RAM, memory availability is not an issue. As we

performed our experiments with up to 6 victim VMs, concurrently running VMs should not be an issue either. One point to consider is operational noise, e.g., due to the migration of VMs to different hosts (e.g., for a host kernel update or to rebalance load). These events can not be excluded and might prevent an ongoing attack, especially when attack time is in the order of weeks. As these maintenance tasks are mostly performed in larger intervals, we do not consider this a serious problem for a determined attacker.

5 Mitigations

The CAIN issue cannot be accounted to one specific vulnerability in the VMM or system that can be punctually fixed. We describe multiple potential mitigations for the attack and discuss their effectiveness and implications.

VMM layer: Deactivation of memory deduplication Deactivating memory deduplication will effectively mitigate all attack vectors. This measure unfortunately eliminates all the highly appreciated benefits of memory deduplication, namely the increase of operational cost-effectiveness through inter-VM memory sharing.

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 entropy in RBAs. 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 on 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.

6 Related Work

There are a number of studies that use a similar side-channel attack on memory deduplication for different purposes as ours. Suzaki et al. [20] showed that it is possible to identify running applications in neighboring VMs. Owens and Wang [16] fingerprinted neighboring OSes with a similar attack, and Irazoqui et al. [10] managed to detect cryptographic libraries along with the IP addresses of the neighboring VMs. Xiao et al. [23] created a covert channel in virtualized environments using this side-channel to create a stealthy backdoor.

There has been a large body of work around side-channel attacks over shared CPU caches. Most notably `FLUSH+RELOAD` [25] showed that it is possible to leak data from a sensitive process such as one that does cryptographic primitives. Irazoqui et al. [9] improved the attack, retrieving AES cryptographic key in the cloud through a combination of `FLUSH+RELOAD` and a memory deduplication side-channel. They recently improved their attack to successfully retrieve private keys from other cryptographic libraries [11]. Using a similar attack, Zhang et al. [26] could leak sensitive application-level data to hijack user accounts, and to break SAML single sign-on.

Hund et al. [7] propose a technique based on shared CPU cache timing side channels to leak kernel space ASLR from user space.

Recent work showed that the VMM can force page-faults in the guest VMs in order to retrieve sensitive information from the VMs, such as RBAs [24]. In this paper, we showed that we can retrieve RBAs without the VMM privileges required for inducing page-faults in the guest VM. Instead, our exploit, CAIN, relies on the page-faults resulting from the memory deduplication process.

7 Conclusion

We described CAIN (Cross-VM ASL INtrospection), an attack against memory deduplication in virtualized environments that breaks ASLR of Windows and Linux targets running on the same physical hardware. Our attack prepares well-known memory pages that contain the randomized base address and uses the memory deduplication side-channel to detect if the page exists on the target VM.

Our proof-of-concept exploit against KSM, the memory deduplication implementation used by KVM, a popular VMM, breaks ASLR of a 64 bit Windows in less than 5 hours, and of a 64 bit recent Linux system in around 18 days. We have shown that the attack time depends on attacker available memory, the number of co-located VMs and the memory deduplication configuration in place.

We discussed possible mitigations and their implications, but given the great interest to support multiple tenants in cloud-based environments, we think that additional investigation of CAIN-like attacks and defenses is urgently required.

We have notified the affected vendors about CAIN; the issue is tracked under CVE-2015-2877.

Acknowledgments

We thank Andrew Honig from Google, Hanieh Bagheri, Nuno Tavares and Robert van der Meulen from LeaseWeb and all the other anonymous cloud providers that helped us better understand real productive cloud setups. We would also like to thank Lucas Davi and the anonymous reviewers for feedback on improving the paper.

References

[1] Gartner Forecast Overview: Public Cloud Services, Worldwide, 2011-2016, 4Q12 Update, published February 2013 and Alsbridge Analysis, G00247462.

[2] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ASIA CCS '11.

[3] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *SEC'15: 24th Usenix Security Symposium* (2015).

[4] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2010).

[5] DAEMEN, J., DAEMEN, J., DAEMEN, J., RIJMEN, V., AND RIJMEN, V. Aes proposal: Rijndael, 1998.

[6] HTTP ARCHIVE. Http archive - interesting stats - average sizes of web sites and objects, 2014. `http://httparchive.org/interesting.php?a=All&l=Mar%201%202014`.

[7] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 191–205.

[8] INGO MOLNAR, ANDI KLEEN. Linux cross reference: mmap.c, 2009. `http://lxr.free-electrons.com/source/arch/x86/mm/mmap.c#L68`.

[9] IRAZOQUI, G., INCI, M., EISENBARTH, T., AND SUNAR, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses* (2014), RAID '14.

[10] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies* (2015).

[11] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ASIA CCS '15.

[12] KEN JOHNSON, MATT MILLER. Exploit mitigation improvements in windows 8, 2012. `http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf`.

[13] QEMU's Kernel Virtual Machine. `http://wiki.qemu.org/KVM`. [Online; accessed 17-05-2015].

[14] MILOS, G. Memory CoW in Xen, 2007. `http://goo.gl/zxfMTD`.

[15] NERGAL. The advanced return-into-lib(c) exploits. *Phrack 11* (2007), `http://phrack.com/issues.html?issue=67&id=8`.

[16] OWENS, R., AND WANG, W. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *In the Proceedings of Performance Computing and Communications Conference* (2011), IPCCC '11.

[17] PAYER, M. Too much PIE is bad for performance. Technical report, ETH Zurich. `http://nebelwelt.net/publications/12TRpie`, 2012.

[18] RACHAMALLA, S., MISHRA, D., AND KULKARNI, P. Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems. In *High Performance Computing (HiPC), 2013 20th International Conference on* (2013).

[19] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security* (2007), CCS '07.

[20] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory Deduplication As a Threat to the Guest OS. In *Proceedings of the 4th European Workshop on System Security* (2011), EUROSEC '11.

[21] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE S&P '13.

[22] VMWARE. Understanding Memory Resource Management in VMware vSphere 5.0, 2011. `goo.gl/60aq17`.

[23] XIAO, J., XU, Z., HUANG, H., AND WANG, H. A Covert Channel Construction in a Virtualized Environment. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12.

[24] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side-channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), IEEE S&P '15.

[25] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014), SEC'14.

[26] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14.