Co-funded by the Erasmus+ Programme of the European Union

ISSES – Information Security Services Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

Operating System Kernel Adaptations

Computer Security

Lecture 6

ISSES 2017-2020
Novi Sad, Serbia

Information Security Services Education in Serbia (ISSES)

6.1 Kernel Adaptations — an Introduction

Adaptations vs functionality

- Security functionality is built into the OS as a given
- Just to function, the OS must provide services which in themselves guarantee a certain amount of security.
- A good example of this is MMU and memory space functionality: this isn’t just for security’s sake.
- Just for virtual memory to work properly the OS must take care to separate address spaces.
- Adaptations, however, are modifications to the way the kernel arranges things which are *not* needed by core functionality and are not in and of themselves something which provides security.

Adaptations vs functionality

- Adaptations—in the terminology we'll use here, purely for educational reasons—are mechanisms which make certain types of attack harder.
- The attack doesn't target the kernel so much as something else, typically in userland.
- Adaptations, therefore, harden the system to certain types of attack in a, it is hoped, general way.
- A particularly important subtype of adaptations are adaptations created as a response to hardware security faults.

Adaptations vs functionality

- As has been covered extensively during talk about hardware/architecture providers of security and secure hardware systems, a lot of security depends on hardware subsystems designed to provide security.
- These systems may have errors and these errors, in turn, are very difficult to patch as the nature of the hardware fault may preclude it.
- In the case such an attack is detected, the correction must therefore come from the code making use of the hardware in the most direct way and this is the OS.
- Thus, the need for adaptations to *hardware* faults.

Layout of lecture

- This lecture, rather than iterate over all possible adaptations: an ever-growing and ever-mutating list instead focuses on one particular case.
- Therefore it will first introduce a simplified model (which later lectures will extend and expand upon) of the way operating systems manage memory.
- Then it will explain a class of attack known as CVE-2017-5754 or, more popularly, ‘Meltdown.’
- Lastly, it will explain how Meltdown is mitigated in the Linux kernel.
- Details about ‘Meltdown’ are in large part sourced from the excellent original paper from the 27th USENIX Security Symposium.

Connection to future lectures

- Other adaptations will be covered later in lectures dealing with memory-based attacks specifically (lectures 8 and 9) and, in particular, lecture 10.
- Of particular interest to us is address space layout randomization which is a classic example of using an adaptation to harden against attacks.
- The material of this lecture ought to serve to underline the importance of the section of lecture 7 where hardening an OS is considered with regard to keeping current with kernel updates.

Information Security Services Education in Serbia (ISSES)

6.2 A SIMPLIFIED MODEL OF MEMORY MANAGEMENT

Address translation

- The natural way for a CPU to work is to work with physical addresses, in other words, to work with addresses *in the actual memory chips* in which the data are stored.
- This presents a number of problems for writing reliable code as well as for extending memory through the effective use of swapping, and for allocating working code to an arbitrary section of memory.
- It is also a security nightmare as it gives anyone executing the most basic instructions which are necessary for any useful code complete control over the entire memory.
- To this end the process of address translation is the part of most operating systems and their corresponding hardware.

Address translation

- Put simply and very briefly, the actual code being executed works with notional addresses (‘linear’ addresses) which have significance only in one of many virtual address spaces.
- In this space memory is perfectly flat: any address is equally accessible, equally good, and addressed using the same method.
- However, at point of access this address is turned from this form into an actual physical address and used in that form.
- The translation is programmed by the operating system as part of the module of the kernel which manages memory, but it is performed by hardware.
- This provides speed and security.

Address translation

- Memory belonging to different processes (and the kernel) is thus kept rigidly segregated (memory not our own can't be addressed because none of our virtual space is mapped to it).
- Further, memory belonging to the kernel and memory belonging to user processes are kept not only separate but the difference is annotated.
- This was covered during the talk of privilege levels.
- The x86 architecture (which we generally focus on) provides a number of complex ways to map memory but the one we really care about is *paging.*
- In it, memory is mapped through a system of pages: sections of memory of fixed size.

Paging

- Briefly, virtual memory is defined as consisting of a series of fixed-size pages and these are mapped into equally-sized physical pages (in memory) using a table.
- Since pages are typically 4096 bytes to keep the table from being impractically vast, it’s generally kept in a four-level hierarchical structure.
- This mapping also contains bits indicating what may be done to and by a certain page. We’ll talk more of these soon, but an important detail is that everyone’s virtual address space contains not just the space for the user application but also space for the kernel.

Kernel ubiquity

- This may seem odd but it is both safe and very useful.
- It is essentially impossible to write a piece of userland code which is in any way useful and which does not interact with the kernel.
- This interaction, done through system calls, requires that code access both user data (to read what must be done and return results) and kernel data (since that's where the code which, e.g., reads a file actually is).
- This is done by having the kernel be everywhere accessible, even though the mapping is to the exact same physical memory.

Kernel security

- Kernel memory is only accessible if the privileged bit of the CPU is set which means that it is executing kernel code.
- This limitation is enforced by the operating system by setting the user accessible bit to false for the data and the needed translation tables.
- This means that while the local process technically ‘sees’ the mapping to kernel space it can never access it except through mechanisms which allow for the orderly and controlled transfer of power—system calls.
- The Kernel meanwhile not only has access to its own space but *everyone’s* space. In Linux this is an explicit part of system startup: the entirety of the physical memory space is mapped to a predefined virtual address.

Brief note on size

- This cavalier attitude to having various bits of virtual space mapped to physical space may seem wasteful but it should be noted that *theoretically* a 64-bit system can map 2 to the 64^{th} power bytes which is works out to 18 million TB of memory.
- In practice, it should be noted, that most modern processors, while storing addresses as 64-bit values internally really use 48 address lines (in rare cases 56-bit) which means that the actual limits are typically 256TB of memory.
- This is an order of magnitude more than what's currently used.

Information Security Services Education in Serbia (ISSES)

6.3 How meltdown works

Background—OOOE

- Out-of-Order execution is an optimization trick used by modern CPUs.
- The idea is to use as much of the computing resources of a given CPU at any one time under the notion that anything computed now won't have to be computed later.
- This is a vitally important concern due to the problems of the performance pyramid.
- The performance pyramid is the result of radically different speeds various subcomponents of a modern computer system operate on, and especially incredibly fast CPU execution times.
- Therefore the OOOE technique is designed to execute instructions in order other than the one defined by the strict sequence specified in code.

Background—OOOE

- If we imagine that we are summing a sequence of 3D points in order to compute a centroid in each iteration of some loop we perform three additions.
- The order of the additions is *entirely* immaterial, but because of the way programming generally works by coding we've *specified* an order of execution anyway.
- This is a prime target for OOOE to optimize, allowing these instructions to be executed in parallel.
- This is a trivial case, but for best possible performance OOOE works even in the case where instructions may have some sort of interaction.

OOOE dependencies

- The ‘scheduler’ component of the execution engine of a CPU core is responsible for performing the function o an Unified Reservation Station.
- This distributed the micro-operations that are a part of an instruction to various execution units (a core has multiple, each specified for a given subfunction of a processor) and maps the registers of the processor. This lets the parallel executions of related microperations use the same physical registers as mapped onto *logical* registers.
- The URS/Scheduler also performs the function of resolving conflicts between multiple executions so that one execution does not clobber the execution of another.

OOOE Speculative Execution

- Instructions are not only executed out of order in order to parallelize operations and increase performance but also in order to maximize utilization a practice known as *speculative execution* is employed.
- When a CPU reaches a branch *strictly speaking* it should stop any further execution until it has evaluated the nature of the branch: i.e. if it is jumping or not.
- This is because the CPU can't execute the next instruction *unless it knows which one it is.*
- This causes a massive loss of performance as the bulk of the core's resources are forced to stand idle, buffers empty and parallelism fails.

OOOE Speculative Execution

- Therefore the likely outcome of a branch is *predicted* by a component of the ‘frontend’ portion of the core known as a ‘Branch Predictor.’
- The Branch Predictor tries to guess how a given conditional will evaluate before it actually evaluates and, based on the guess executes the instructions which *would* have gone next if the guess is correct.
- Not all instructions might be executed: just those without preferences, but still it allows the results to be cached ahead of time.
- The reorder buffer subcomponent allows the CPU to, in case of a bad guess, restore the state before the speculative execution and to try again. Of course this causes a performance hit.

OOOE Speculative Execution

- Code without branches is rare. Therefore, the performance of a chip may be in large part determined by the sophistication of the branch predictors.
- Dynamic branch prediction is typically used where runtime statistics are kept allowing the CPU to make educated guesses. Modern CPUs often use adaptive predictors with two levels, keeping a history of outcomes of a given branch.
- This approach even allows for simple patterns to be detected.
- Already, in the search for more performance, neural branch prediction is the new state of the art. AMD Ryzen already implements perceptron-based branch predictor.

Background—Cache attacks

- Cache attacks will be covered in more detail in later lectures.
- Cache represents small memory buffers of various speeds that are integrated into or near a processor.
- Modern processors have a very sophisticated web of multiple levels of cache that may or may not be shared between multiple cores.
- The cache is there to hide memory latency issues that result from the pyramid of performance—the CPU outpaces memory to a great degree, and waiting for it all the time would make high CPU performance pointless.

Background—Cache attacks

- The shared nature of cache makes it a prime attack surface for side-channel attacks.
- A trivial example is a family of cache timing side-channel attacks in which a memory location is repeatedly targeted with the clflush instruction.
- Clflush causes the cache of a given instruction to be invalidated and any changes written to main memory.
- Then, the time needed to fetch data back into cache is measured carefully and used to discover if any other process has loaded data into cache in the meantime.

How does that help?

- Using Gruss' 2015 USENIX paper on the topic as a reference, it is possible to exploit this by, first, targeting L3 cache (since all processes share it) and then using the timing attack to periodically determine what addresses a process has accessed in a given period.
- Since programs act differently for, e.g. different input events it is possible to match the Boolean vector containing info whether a given address has been touched or not to a specific event.
- And just like that you can have a keylogger that requires *no specific privileges.*

Cache-attacks as exfiltration

- Since all processes share L3 cache and since it is possible to extract useful data from timing variations for a given cache line, this means that two processes in separate execution and security domains which should not be able to communicate in any way can communicate by having one distort particular cache-lines and the other read out those lines.
- This is significant if we have, e.g. managed to obtain some data in one memory space and wish to convey them to another part of memory space.
- This will become relevant shortly.

Core idea of Meltdown

- This information can be used to construct the core idea of meltdown.
- Imagine we have code which consists of only two lines: one which inevitably raises an exception (say it tries to divide by zero) and one which reads an array location indexed by a counter value multiplied by 4096.
- This second line, accessing the array with some function ‘f’ *cannot execute.* The moment the function which raises the exception is executed it will be off to the exception handler in the kernel (if we’ve divided by zero, say).
- Despite never executing *officially* the array access still has effects within the processor because of speculative execution. The CPU simply guessed there won’t be an exception because most of the time there isn’t.

Core idea of Meltdown

```
1  cause_exception();
2  f("%d",probe[c * 4096]);
```

Guarantees access is spread across different 4K pages.

This line is unreachable, and yet causes an *effect.*

Core idea of meltdown

- Therefore if we check timings for various cache lines in the exception handler we can identify what the value for ‘data’ was, as that will be the address with a shorter fetching time. (See cited paper, figure 4)
- This allows unexecuted instructions to leak information.
- However, this core idea shows *odd* behavior, but doesn’t harm the security of the system in any way.
- Let’s assume that the problem is fetching some secret value hidden somewhere in physical address space.
- We need to somehow read this value using instructions that don’t actually get executed (the term is ‘transient instructions’) and based on the secret modify the microarchitectural state of the CPU. Then getting the secret is only a matter of reading the state.

Meltdown—Attack setting

- For a more concrete example of Meltdown we assume that we have access to a system allowing arbitrary unprivileged code execution.
- We do not assume any privileged access.
- We do not assume physical access to the machine.
- We do not assume any vulnerability in the OS itself.
- We are targeting secret user data: keys, passwords, banking details…

Meltdown—spy code

- The spying is done by accessing and exfiltrating data in kernel space.
- Let us assume for simplicity's sake that we focus on reading a *single* byte from kernel space and that we have the address needed in register rcx.
- Let us assume further we have a probe array like before with an address in rbx. We flush rbx so we can be sure it is not cached. The probe array is otherwise normal and is 1MB long i.e. 256 4KB units.
- Then what we can do is try to move the data from the kernel address into our own register. This will cause an exception that will crash our program *but not before the CPU executes transient instructions.*

Meltdown—spy code

- Transient instructions use the secret value (which they have access to since they don't really *exist* yet and the OOOP mechanism provides early access before the exception)
- The transient instructions therefore use the secret value as the 'counter' value for accessing into the probe array.
- Therefore, by determining which cache line is unusually quick we can tell what the counter value must have been and therefore what the secret kernel value was.
- Thus information has leaked from the spy code into monitoring code.

Meltdown—spy code

```
1 xor rax, rax ;anull receiving register
2 retry:
3 mov al, byte [rcx] ;load secret byte into low-rax
4 ; Transient code starts here
5 shl rax, 0xc ;multiplies by 4096
6 jz retry ;denoises
7 mov rbx, qword [rbx + rax] ; this leaks information
```

How can this read data?

- The secret is in not just speculative but out of order execution. This means that it is highly likely that we will have our data-exfiltrating functions decoded and waiting on data at the scheduler before our MOV instruction is decoded.
- This means that there will be an interval where valid data is present and fetched and our code is executing (since kernel data always maps to all of physical space) and the exception hasn’t happened yet.
- This race condition is all it takes to get data out.

Denoising?

- The system has a noise-bias towards zero.
- This happens when CPUs don’t wait for a missing value, but assume it. Often the assumed value is ‘0.’ This means that sometimes when the attack loses the race condition instead of stalling out it will read a zero.
- This means that in case of a zero being read a retry attempt is made.
- Hence the jz operation which in case the previous operation executes with a result of zero will jump to the retry step.
- This retry operation is tried by the spy-sending code a number of times (depending on optimization parameters) so as to filter out spurious zeros.

Doesn’t this crash the system?

- Inevitably.
- This is why typically one process reads and a another, sacrificial one is let to crash repeatedly.
- This is not the only way to do it, techniques known as exception suppression can increase performance, but this introduces a complication we can do without.
- The simplest way to do this is to have the process fork with one fork performing the spy code and the other executes the Flush+Reload side-channel attack.
- This means that the forked copy will crash but another will take its place shortly.
- This technique allows us to read out, byte by byte, the entire system memory, including any secrets the system may have.

We are doomed?

- This means that any code we run on our system can dump our entire system memory and send it somewhere for analysis.
- There’s no defense here because it assumes that everything that isn’t the CPU is operating exactly as specified.
- There’s no bug to patch, and no countermeasure to take.
- When Meltdown was first reported researchers assumed that it was *a hoax* because it was *too catastrophic.*
- Is there anything to be done?

Information Security Services Education in Serbia (ISSES)

6.4 Defending against Meltdown

Adaptation #1: KASLR

- Meltdown isn’t the first attack to try to fiddle with Kernel memory space for nefarious purposes.
- In these instances, one technique which we will be seeing again, is to make the attacker’s life as hard as possible.
- So far, we assumed we can just start reading that part of Kernel space which contains mapped the entirety of physical memory.
- Okay, but where is that? In older versions of the Linux Kernel the layout was instantly known since it never changed and the precise offsets were clearly visible in the code. But since version 3.14 (and on by default since version 4.12) a technique was introduced to cause us headaches.

Adaptation #1: KASLR

- Where does the headache come from? But as we've discussed previously, the virtual kernel space is potentially 128TB large. Finding things in there might be… troublesome.
- Luckily, the way KASLR is implemented is a bit simpler: the map of all of physical space in kernel space is not where we expect it to be but moved by an offset chosen at boot which has 40bits of randomness.
- This allows an enterprising attacker to try to read memory from the initial position of the offset in steps equal to the size of the memory.
- The moment an address is read successfully the offset has been found and the attack proceeds as before.

Adaptation #1: KASLR

- Assuming a memory size of m bytes and a randomness of n-bits the number of tries, t you need to find the offset in the worst case scenario is

$$t = \frac{2^n}{m}$$

For a n = 40, and a computer with 32GB of memory you have

$$t = \frac{2^{40}}{2^{35}} = 2^5 = 32$$

KASLR failure

- A valiant attempt when attempting to hide a very small, specific thing.
- However, a 40-bit wide spice isn’t at all that big when we can search it in giant gulps corresponding to vast modern memory sizes.
- This means that KASLR slows down Meltdown implementations by a few seconds at best.
- The security implementations of that are negligible.

Adaptation #2: KAISER/KPTI

- Daniel Gruss (responsible for a lot of early work on cache attacks) proposed an early solution to the problem at a software level, adapting the kernel for hardware problems. The original proposal was published as a paper which swiftly lead to a patch being made for the Linux kernel and similar fixed implemented in other operating systems.
- KAISER stands for Kernel Address Isolation to have Side-channels Efficiently Removed.
- With less affection for clever acronyms and more for clarity of communication the same technique is generally called KPTI—kernel page table isolation and is a part of the Linux kernel since version 4.15.

What actually gets isolated?

- The kernel still needs to have access to everything. That's not amenable to change.
- But there's not an actual *rule* that says you can only really have one paging table for all.
- All the user *needs* to have access to is a few kilobytes of kernel space for interrupt handlers and the like.
- So what can be done is to have two different tables: one for the kernel with everything in it, and one for processes which have no access to the kernel data (aside from those few kilobytes).
- This means that our transient instructions can no longer find purchase: they get executed but they'll never win the race condition because the data will never get there, because there will be no map to a real address.

KPTI—The good

- KPTI is a complete mitigation for Meltdown. Systems implementing it cannot be targeted by it.
- The implementation is simple and doesn’t break anything as no user code is meant to rely on the presence of kernel space in its address space which it can’t even access.
- The implementation arrived before any exploit of Meltdown appeared in the wild.
- It was adaptable to other systems—we aren’t privy to the implementation details but Microsoft has its own version and it is called KVA Shadow, and MacOS X has Double Map which is the same general notion.
- KPTI also fixes KASLR bypass by making it *much* harder to search the 40-bit space with only a few kilobytes to work with.

KPTI—The Bad

- KPTI doesn’t protect against a related exploit known as Specter.
- Specter uses the exact same approach of tricking the CPU of executing transient instructions.
- The difference is that instead of going after the entire memory it attacks the memory of the same process.
- This isn’t normally that bad, but the mechanism employed by the authors to demonstrate their attack is tailored JavaScript code which completely violates the sandboxing policy and accesses the entirety of the process space mapped to the browser process.
- Browser keep an awful lot of secrets.

So how to fix Spectre?

- …nobody's *quite* sure.
- It's such a wide thing that it isn't *fixed* so much as mitigated by a panoply of solutions and techniques.
- We'll be discussing one or two when we get to compiler mitigation.
- But the technique revolves around fixing a bit of the CPU and tweaking the binary code to be resistant to external manipulation.

KPTI—The ugly

- There’s a performance hit.
- Nobody can quite agree on how *big* a performance hit
- Some measures report an inconsequential 0.28%, while others report an apocalyptic 30%.
- The confusion comes from the source of the performance hit: switching how memory is mapped between userspace and kernelspace requires flushing the transaction lookaside buffer each time the context switches between the kernel and the userland application.
- This happens *every time you read something or use a syscall.*
- This means that high-performance I/O-heavy operations are heavily impacted.

Information Security Services Education in Serbia (ISSES)

6.5 Conclusion

Adaptation and mitigation

- Operating systems manage hardware: sometimes this includes managing mistakes, too.
- Some adaptations are like KPTI: identifying a problem and arranging so that it cannot be exploited at all.
- Some adaptations are like KASLR: identifying a potential general thing an attacker can rely on and spoiling the attacker's day.
- Some adaptation are like those for Spectre: a mixture of patches for specific problems and added elaborations that attempt to make exploitation more difficult.

What needs to be adapted to?

- Potentially *anything.*
- Spectre and Meltdown are hardly the only problems that have been detected with modern hardware. Others exist.
- As matters stand the kernel depends on the promises made by the manufactures of hardware security systems.
- Any of those promises may be broken.