Presentation script:

Introduction to Spectre

Hello, I am Alex Young, a student who has been taking CS 373, Defense against the Dark Arts over the summer of 2021 for the Oregon State ecampus. For my final write up presentation, I did some research on the topic of Spectre and I plan to share what I've learned over the past few days in this video. I found Spectre to be an intensely interesting topic and hope this presentation will show you why I think so.

(0:15)

- What is Spectre?

So, I am going to start out by explaining what Spectre actually is. A broad way of describing it is that Spectre is a current security vulnerability in computer systems, or specifically in CPUs. In order to understand the full effect of what this means, I want to emphasize some points regarding the background of computer security and Spectre.

Security vulnerabilities themselves are essentially what they sound like, patches or holes in the system that can be exploited by attackers with malicious intentions. These are weak spots that could potentially be hacked into and used to steal sensitive and protected information that normally should not be accessible. This is different from the virus itself, which usually uses these vulnerabilities but in general is a separate entity from the vulnerability. This is important because it should be clear that Spectre is only a vulnerability and not a virus.

However, despite being "just a vulnerability", Spectre has gained enough infamy and importance to be a large topic of discussion and interest in itself. For me this was quite amazing because when I usually think about security related topics, I think of different viruses or malicious groups. However this vulnerability was scary enough that it deserves a name for itself.

Continuing with explaining what Spectre is and why it's so scary, we need to look at how it works. I will go over this in more detail later, for now what I want to emphasize is that Spectre is a vulnerability of modern processors and specifically speculative execution. In simple terms, modern processors have increased in complexity over the years and use many tricks to increase their speed. Over the past 20 years, some of these tricks have been vulnerable, and yet they haven't actually been discovered until Spectre, and a similar vulnerability called Meltdown.

(2:15)

- Who is impacted by Spectre?

So now that we have a basic understanding of what Spectre actually is, it is important to understand why it is so scary. The core of the issue is something I already mentioned. Here it is: Almost all modern CPUs have this vulnerability. In fact, all Intel x86 microprocessors, AMD, ARM and IBM processors are vulnerable. This means that billions of devices and computers are all vulnerable to Spectre and could potentially be exploited and used to retrieve private data from.

The one caveat to this is that there are little to no known usages of this vulnerability in malware in the wild and a system has to already be infected for this vulnerability to be taken advantage of. Despite this caveat, there are two main points to take into consideration. One is that this vulnerability can be exploited using javascript, which means that most browsers are very vulnerable to Spectre. Secondly, because of the way Spectre works by directly accessing

memory using the CPU, it is potentially able to bypass fences that block off sections of virtual machines, meaning that cloud based systems need to be especially cautious of being exploited. (3:30)

- Why is Spectre important?

There are three reasons why Spectre is so important. First of all is because it can impact so many people and so many devices. After the release of information regarding Spectre, processor developers had to work as quickly as possible to try to update their systems so that bad actors couldn't take advantage of Spectre.

However the second reason is largely related to this. It is very difficult to defend or protect against Spectre. Despite trying to update systems using software, the core of Spectre is that it is a hardware issue that is built into the processor itself. This means that the vulnerability actually works by using the processor in a way that is intended. Essentially, this vulnerability is technically not seen as an issue by the processor, which in turn makes it not only hard to defend, but also hard to find using tools like anti-virus in the case of attacks.

Because Spectre takes advantage of the design of the system architecture itself, and is essentially a product of the system that is "working as intended", it means that in order to ever completely get rid of Spectre and have it disappear would take quite a long time. In fact, the researchers who found Spectre named it as such because, quote, it will haunt us for quite some time.

Finally, the last reason that Spectre is important is that it can be used to read from a computer system's memory. This means that sensitive information like passwords or emails could be quickly accessed by malware employing Spectre.

(5:10)

- When/Where was Spectre discovered?

Before I get into describing how Spectre actually works, I want to quickly explain some of the background of Spectre's discovery while I am on the topic. The process of reporting Spectre to the public was quite interesting because it was actually independently discovered by two different research teams, Google Project Zero, and a group of university researchers at around the same time. In addition, at again around the same time, the similar Meltdown vulnerability was independently discovered by three different researcher teams as well. For this reason, as well as because of the similarities in how they work, the Spectre and Meltdown vulnerabilities are often grouped together in discussion. The documentation regarding these two vulnerabilities was released in January 2018. While the focus of this presentation is on Spectre, I just wanted to mention this because the Meltdown vulnerability is also very interesting and is a good topic to look into on your own time outside of Spectre.

(6:15)

Understanding how Spectre works:

Okay, so now I am going to start going more in depth and begin explaining how the Spectre vulnerability actually works. For me, this was the most interesting part of Spectre, but I will have to start by explaining some of the background so that everything makes sense.

- The Evolution of the CPU

Despite having personally very little knowledge about CPUs and computer hardware, I have heard of Moore's law. This basic principle says that the speed and power of computer

processors will increase over time. In order for this principle to have remained true over the past many years, processor developers have to use many different tricks and techniques.
One of these tricks that the modern CPU uses to speed up the amount of instructions it can execute is called pipelining. In the image shown here, instruction pipelining occurs during parallel processing. The idea behind pipelining is breaking up sections of execution into stages. The instructions in these stages of the pipeline can be run much faster by reducing the amount of clock-cycles wasted by using out-of-order execution and speculative execution techniques. (7:30)

- Out-of-order Execution

The out-of-order execution technique that CPUs use is essentially based around the idea that many instructions that are run in a period of time are essentially independent of each other, meaning that the order they are processed in doesn't matter. As such, they can be run in "parallel". This is especially useful because some instructions take quite a bit longer than others. Specifically instructions to read from memory can take a long time, which is quite important later in this exploit. The out-of-order execution is not the focus of the Spectre vulnerability but does play a larger role in the Meltdown vulnerability. To get a more clear idea of how this works, we can look at this next slide where you can see that while normally instructions b c and d would have to wait for a to finish executing to start, they are able to execute themselves out of order. (8:25)

- Speculative Execution (branch prediction)

After out-of-order execution in the pipeline comes speculative execution. This is the real most important thing to understand when understanding how Spectre works. The idea behind speculative execution is that when there are possible branches in the instructions the processor predicts what the following instructions will be. This prediction is done with a special unit called a branch predictor.
For example, this could occur at an if statement. The idea is that by processing these instructions ahead of time, the processor will save clock cycles later. In the case that the prediction is incorrect, the processor will get rid of the previously speculated executions.
In theory and in practice this works very well, however there is an issue with speculative execution in that the branch predictor can be tricked. This can be done via poisoning or misdirecting the branch predictor to speculate on data of the user's choosing.
While the speculative execution will end up failing and end up being erased and thrown away, it will leave behind invisible traces that can be tracked down by attackers to reveal data that the speculative execution ended up accessing. This data could include private and protected data that speculative execution has the right to access.
(9:50)

- Cache

Now the big question is, how is it possible to actually get any valid information from the failed and misdirected speculative execution that occurred? After all, was all the information just thrown away before anything could access it? The answer lies in shared resources, and specifically in the CPU cache.
One of the things that happens during speculative execution is that the CPU cache gets updated with fast access memory. This is important because of the pipelining technique that I discussed before. In order to make processors as fast as possible, developers rely on a CPU

cache for quick access to memory, and it is far faster than RAM because it is built directly into the CPU.
The reason this is important is because there have already been well established timing attacks that target the cache that can be used to access the memory addresses stored in the cache. These timing attacks use the fact that it takes a very short amount of time to access the memory stored in the cache to read memory addresses from the cache. The idea is simple: information about data in the cache can be found by seeing how long it takes to access certain data. This can be further used to read the information that was hidden in memory, including sensitive data like passwords or encryption keys.
(11:15)

- Putting it all together

Overall, the Spectre vulnerability is based around the fact that it is possible to redirect the Speculative Execution towards memory locations of a user's choosing. Using this, a user can use a cache timing attack to read the information that was speculatively executed.
(11:30)

Spectre Attacks
Now I am going to go further in-depth to discuss how theoretical Spectre attacks would actually work. The official documentation for Spectre includes many proofs of concepts of different variations to show that it worked. I will be using examples from this documentation to explain how a simple attack could happen.

- Branch Redirection

To start, given what we know about Spectre, the first step is to see how the branch predictor can be mistrained. In the documentation there are two main variations, among others, that they have. This includes using conditional branches like if-statements repeatedly to mistrain the predictor to head towards a hidden address in memory and in the other case training the predictor to head towards a planted gadget that can leak information. In the first case an example of the code used is quite simple. By repeatedly having the instructions run the code with valid inputs, the predictor will execute the code inside even if there is an invalid input and save the information in the cache. We can see that this is quite devious because there is nothing here that is actually incorrect coding.
An example of this is shown here on the slide. In this example code, you can see that there is an if statement that compares the x value to the size of array 1. Here x is initially run in this code using a valid input to train the branch predictor to assume that the if statement is true. Later, x is run using an out-of-bounds location, and if the branch predictor has been successfully trained, it will execute on the code inside before the real instructions catch up. This concept is shown in the image here. You can see that when the if statement is false but had been predicted as true, the attacker can access hidden information. Continuing on with the code, when x is chosen out-of-bounds, the idea is that array1 at x is the location of sensitive memory. If this location is cached but array1_size and array2 are uncached then the instructions will take a long time to reach the conclusion that x is out of bounds and the speculative execution will cache the information in array1 at x and array2 at this value times 4096. This cached information can be read using a timing attack which I will explain.
(13:55)

- Flush and Reload

Going on to reading the information from the cache, there are quite a few possible shared resource timing attacks that could be used. However, the main ones to take note of are Flush and Reload and Evict and Reload. The first steps of these attacks accomplish the same goal of flushing the cache so I will focus on the Flush and Reload attack. In this attack, the attacker will use x86 instruction clflush to flush out a certain line from the cache. Then the attacker can then see if the victim has accessed the flushed line by checking how long it takes to access the line. This sort of attack can be greatly advanced with complicated algorithms to extract crypto keys, look at instructions, look at branch history, keystrokes, websites, and even create cloud-based and educational-based attacks.
(14:45)

- Results

Using the previously described methods, the official Spectre documentation paper describes that using their proof-of-concept unoptimized code, they successfully ran their code on a variety of i5 to i7 CPUs and can read about 10 kilobytes per second of memory locations on a x86 processor with a less than 0.1% error rate. In addition to this, they were able to run the code far ahead of the instruction pointer with speculation, with up to 188 instructions in between them.
(15:15)

- Variations

Not only does this proof of concept show the danger of Spectre, the documentation showed how the same exploit can be run using JavaScript and by exploiting eBPF. These different methods of exploiting Spectre all increase its danger and possible use in malware.
In addition to using different scripting languages, the documentation showed and analyzed three other variations of Spectre. One of these variations was poisoning indirect branches, which showed how Spectre could be used to read memory from other processes than the victim was using through indirect branches. The results of this test found that they were able to jump in memory successfully 99.7% of the time and correctly poison the branches 98.6% of the time, with the capability of leaking 1809 B/s. While this is quite a bit less than the 10 KB per second using the first method, it is quite scary that Spectre can read sensitive information from other processes than one that the victim is running. In addition, while these are only 2 of the 4 variations that were discovered when Spectre was found in January 2018, in just around a year and a half, nine more variants of Spectre had been discovered. Even more scarily, in this last April, researchers discovered a new variant using an exploit of the micro-op cache which allows attacks to steal data when the processor accesses this cache.
(16:45)

Spectre Defense/Mitigation

Now that we have a good understanding of how Spectre works and how it can be used in attacks and malware, it is important to consider how we can defend or mitigate against Spectre. The first thing to understand is that there is no complete defense against Spectre until new hardware is designed and widely used that is not vulnerable. Despite many software patches, many new variations of Spectre have been researched.
(17:15)

- Preventing Speculative Execution on Protected Data

Preventing speculative execution in general is one way to be sure that a computer is not vulnerable to Spectre. However, this will greatly decrease the processing speed because it would be essentially turning off branch predictions and limiting pipelining.
Instead a preferred method would be to limit speculative execution from accessing hidden data. The official documentation on Spectre provided some ideas for this, including using Google Chrome's strategy of executing separate web pages in different processes. Another strategy uses a mask to limit speculative execution from going out of bounds.
(17:55)

- Further Defense

Other defense strategies include preventing branches from being poisoned. There are several methods to achieve this, and Google has one called retpolines, which replaces any indirect branches with rets.
Another defense to consider is that against the timing attacks that allow Spectre to become a vulnerability. However these sorts of attacks have been based on the inherent properties of the processor and would likely require some fundamental change in architecture to completely remove.
(18:25)

- Mitigation in Use

The google chrome strategy of preventing speculative execution on protected data has been further developed by MIT researchers to create the DAWG - Dynamically allocated way guard - technology which separates the cache to avoid sharing resources.
These mitigation strategies are similar to the strategies used by Linux, Intel and AMD. Linux operating systems use a kernel page-table isolation to unmap protected data from the kernel page table to prevent speculative execution from accessing them. Meanwhile, the Intel and AMD patches to Spectre include changes to their microcode and assembly instructions in order to limit speculative execution.
The key thing to notice from these patches is that in order to get some protection from Spectre, software updates are required. In addition, these patches often are not able to counter every variant of Spectre, and often result in a loss of processing speed. For the CPU manufacturers to completely resolve this problem, it would mean changing the design architecture of their CPUs.
(19:35)

Conclusion

In the end, Spectre is quite a beast of a modern vulnerability with new and upcoming variants. I have barely been able to dig into a few of the variants here myself and it is clear that Spectre is quite an issue for processors and their developers. After all, it is a vulnerability on practically every computer and device that allows an attacker to read sensitive information from memory, and is practically undetectable.
(20:05)

- Key Takeaways

For me, the key takeaway from Spectre is to make sure your software is up to date. With Spectre being so hidden and a result of a design flaw in microprocessors, there really is no foolproof defense. Hopefully by researching vulnerabilities like Spectre, future processors can

be improved in even more robust ways with these sorts of flaws in mind. Finally, it is important to keep in mind that the processor is not infallible and completely optimizing security in hardware to software development is difficult.
(20:35)

- Sources

https://spectreattack.com/spectre.pdf
https://spectrum.ieee.org/how-the-spectre-and-meltdown-hacks-really-worked
https://www.sciencedaily.com/releases/2021/04/210430165903.htm
https://www.techrepublic.com/article/spectre-and-meltdown-explained-a-comprehensive-guide-for-professionals/

The sources that I used in researching and making this presentation are listed here. I mainly used the official spectre attack documentation as well as the ieee blogpost on how spectre works, which are the first two links. These sources are very well written and I suggest looking into them if you have any further questions or are looking for the source code of the proof of concept.

Thank you for watching my presentation.
(21:05)