Lab 2: Spectre Attacks

Due date: Tuesday April 30th 11:59:59 PM ET

Collaboration policy: You may discuss the lab at a high level with other classmates, but you may only work and code together with a single partner.

Getting started: Log in to our lab machine at the ip address: 152.2.135.251. To connect via ssh, run:

ssh YourOnyen@152.2.135.251

You will need to access the machine from the campus network, or use the UNC VPN if you are working remotely.

Introduction

In this lab we will be putting the Spectre attack into practice against the Linux kernel. You will complete three CTF style problems of increasing difficulty to leak a secret from kernel memory using the cache as a side channel. Specifically, we will be attacking a custom kernel module which listens to commands on `/proc/lab2-victim`. We have provided starter code that communicates with the kernel module for you.

Each secret is a string of the form "`UNC{some_secret_value}`". The string can be up to 64 bytes in length, including the `NULL` terminator. You can consider the secret complete once you leak the `NULL` terminator. In the first part, we will begin by implementing a Flush+Reload attack against a shared memory region with the kernel. In this part you will get familiar with interacting with the kernel module and the general project codebase. In the second part you will implement a standard Spectre attack. In the third part you will implement a more advanced Spectre attack which requires carefully manipulating the speculative execution behavior of the victim program.

Note: Do not make any assumptions about the secret other than it is a `NULL` terminated string of length up to 64 bytes (including the `NULL` terminator). When doing the lab, you will notice that the secrets do not change from run to run (they are constant for the lifetime of the kernel module). During grading, we may use different secret values. They will never exceed 64 bytes and will always be `NULL` terminated. Our starter code handles this for you.

Get familiar with the lab codebase

- `inc/labspectre.h` and `src-common/spectre_lab_helper.c` provide a set of utility functions for you to use, similar to those provided in Lab 1.
- `src-common/main.c` is used in all three parts. The `main` function sets up a shared memory region named `shared_memory` of `LAB2_SHARED_MEMORY_SIZE` bytes, and a file descriptor for communicating with the kernel module. The shared memory region is shared between userspace and the kernel.
- `part1-src/attacker-part1.c` is the file you will modify in Part 1 to implement your Flush+Reload attack. The method `call_kernel_part1` can be used for calling into the kernel module.
- `part2-src/attacker-part2.c` is the file you will modify in Part 2 to implement your Spectre attack. The method `call_kernel_part2` can be used for calling into the kernel module.
- `part3-src/attacker-part3.c` is the file you will modify in Part 3 to implement your advanced Spectre attack. The method `call_kernel_part3` can be used for calling into the kernel module.
- `inc/lab2ipc.h` contains bindings for the interface to the kernel module from userspace. You can ignore this as our provided code handles the implementation details of communicating with the kernel.

Compiling and Testing the Project From the root directory (the `lab2` folder), you can use `make` to compile

the project. `part1`, `part2`, and `part3` will be produced in the same directory. Run them with `./part1`, `./part2`, and `./part3` respectively. The results of your attack will be printed to the console. On success, you should see the secret leaked from kernel memory printed to the console.

We show an example of the expected output below:

```
1: $ ./part1
2: UNC{part1_secret_value}
```

We have also provided a checker script that will run your attack many times and report its behavior over many runs. You can invoke this script from the main directory with `./check.py X`, where X is the part number to check. We have provided an example output of this tool below.

```
1: % ./check.py 1
2: Checking part 1 ...
   You passed 950 of 1000 runs (95.0%)
   Success! Good job
```

Background- The Linux Kernel

In this lab we will be attacking a custom Linux kernel module. A kernel module is a piece of software that runs in kernel space instead of user space. The kernel has the most privilege on the system (even greater privileges than programs run as root), and
has complete control over the system hardware. The kernel lives in its own separate address space that user programs cannot access. Figure 1 provides an overview of the address space separation used in this lab.

The user program (your code) cannot access kernel memory directly, as the kernel pages are marked private to the kernel in the page table. The kernel can read and write all memory on the system. In this lab, the user program will create a region of shared memory that the kernel module will map in and interact with. This shared memory region is a userspace memory region. From userspace we can use the cache as a side channel to reveal kernel data. Specifically, by making the kernel read secret-dependent parts of the shared memory region, we can inspect which parts of the shared memory region were cached to determine the secret.

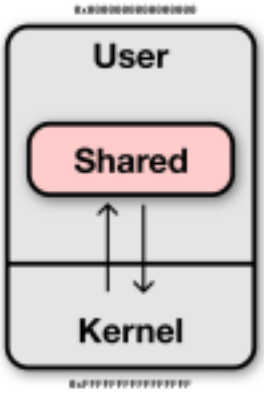

Figure 1

The kernel is configured to boot with all standard Spectre/ Meltdown mitigations enabled. SMAP (supervisor mode access prevention) and SMEP (supervisor mode execution prevention) are both on, which means that the kernel cannot directly read or execute userspace memory. Instead, the kernel module temporarily *remaps* an alias of the shared memory region into kernel space so it can interact with it, and then unmaps it before returning to the user. This is common practice in the kernel.

As a consequence of this remapping operation performed during context switch between user and kernel space, a TLB miss on the shared memory region will happen each time the kernel attempts to read from the shared memory region. For Parts 1 and 2, we prevent the TLB misses to make your attack easier by forcing page walks before the secret-dependent memory access so that the TLB will hit on a read. We do not do this for Part 3. As a result, you will need to craft a Spectre attack that can succeed despite the added latency of a TLB miss.

The kernel module is defined in `module-src/lab2km.c`. For those who are curious, we use the procfs write handler as the API through which user code can communicate with the kernel module. The bulk of message processing and vulnerable code are all handled in the procfs write handler. You do not need to read the kernel module code for this lab, the pseudocode provided here is sufficient to understand the vulnerabilities you need to exploit.

1: Leaking Kernel Memory via Flush+Reload (35%)

In this part we will set up a side channel to leak information from the kernel using the cache and shared

memory. We will use this side channel later in our Spectre attacks as the channel through which the speculatively loaded information is leaked.

In Lab 1 we implemented eviction by accessing many lines in our own address space, and then scanning for changes made by the other process. Since we have shared memory in Lab 2, we can use the more reliable Flush+Reload technique to leak information.

The pseudocode for the kernel victim code of part 1 is provided below:

```
1 def victim_part1(shared_mem, offset):
2     secret_data = part1_secret[offset]
3     load shared_mem[4096 * secret_data]
```

Listing 1: Part 1 Victim Pseudocode

The victim takes a pointer `shared_mem` which points to the starting of a shared memory region and an integer `offset` as input. First, on line 2, the secret byte to leak is loaded from the secret array (`part1_secret`). The byte to leak is chosen by `offset`. When offset is 0, the first byte will be leaked; when offset is 1, the second byte will be leaked, and so on. Next, the victim converts the secret data into an address by using the *value* of the byte to index the shared memory array. If the secret data was the character `'A'` (`0x41`), then the `0x41`'th page in the shared memory region will be loaded into the cache.

Allowed Code: You can define your own helper methods as you desire. You can use any method in `inc/lab2.h`, as well as the provided methods in `attacker-part1.c`. You should only use the provided `call_kernel_part1` method to interact with the kernel module.

Exercise 1: Run your Lab 1 Part 1 latency measurement code and derive the threshold to distinguish L1/L2 and DRAM latency, as you will need this value later.

Attack Outline: Recall that the secret is a string up to 64 characters long (including the `NULL` terminator). The attacker should leak the secret one byte at a time with the Flush+Reload technique:

1. Flush: Flush the memory region from the cache using `clflush`.
2. Victim execution: Call the victim method to leak a given secret byte by properly configuring `offset`.
3. Reload: Reload the memory region, measure the latency of accessing each address, and use the latency to determine the value of the secret. When the value is `0x00` (`NULL` terminator) the attack is complete. We strongly suggest you build your attack step by step: that is, start by successfully leaking one character first, then try to leak the whole string.

Discussion Question 1: How many addresses need to be flushed at the first step?

Exercise 2: Implement the Flush+Reload attack in `part1-src/attacker-part1.c` to leak the secret string. Build the project with `make` and run `./check.py 1` from the lab 2 main directory to check whether you have passed part 1 or not.

Discussion Question 2: Now assume the attacker and victim no longer share a memory region. Would your attack still work? If not, changes could you need to make to make it work?

Interacting with the Kernel Module. We have provided a helper method called `call_kernel_part1` that you can use to interact with the kernel module via the procfs system. It takes three arguments– a file descriptor to the kernel module, a pointer to the shared memory region, and the desired offset. `kernel_fd` and `shared_memory` can be directly passed to this method without modification. The `offset` to use for a given invocation is up to you.

Submission, Grading, and Checkoff: There is no checkoff for Part 1. This part is graded based only on your changes to `part1-src/attacker-part1.c` and your answer to Discussion Question 1.

• You need to submit your code (`part1-src/attacker-part1.c`) to the Canvas submission

• Include your discussion question answers in a text file named `solutions.txt`. This file should include the discussion questions from all parts of lab 2. Full credit will be awarded to solutions that report the correct secret more than 95% of the time. Partial credit will be awarded for solutions that behave worse than that. Each attempt should take no longer than 30 seconds.

2: Basic Spectre (40%)

Now that Flush+Reload is working, let's move onto actually implementing Spectre! Below is the pseudocode for part 2's victim code:

```
1 part2_limit = 4
2 def victim_part2 (shared_mem, offset):
3   secret_data = part2_secret[offset]
4   mem_index = 4096 * secret_data
5
6   # to delay the branch on line 9
7   flush(part2_limit)
8
9   if offset < part2_limit:
10          load shared_mem[mem_index]
```

Listing 2: Part 2 Victim Pseudocode

This victim code is a lot like Part 1, except it will only perform the load if `offset` is within a specific size (namely, if `offset` is no larger than 4 bytes into the secret). The first 4 secret bytes can be loaded non-speculatively as in Part 1. However, anything beyond that can only be loaded speculatively.

> Exercise 3: Copy your code in `run_attacker` from `attacker-part1.c` to `attacker-part2.c`. Test whether your previous implementation works with the new victim.

You should see your Flush+Reload attack does not work any more. The reason is that the secret data after the 4th character is never loaded if this code is executed in order. However, we can use Spectre to bypass this limit!

Vulnerability Explanation: When the processor encounters the `if`, it will begin computing the result of `offset < part2_limit`. While the result of this comparison is unavailable, it will begin to either speculatively execute the true or false condition. This speculation is based on the *branch predictor* 's prediction for what the branch will do. This prediction is based on past observations. So, if the `victim_part2` method is called many times, with `offset` small enough to fit into the limit, the predictor will be *trained* to always predict that the true condition will evaluate to true.

We can train the branch predictor to produce a favorable prediction so that the branch we want is speculatively executed while the branch condition is unavailable (that is, the CPU will guess our offset is within the limit, and speculatively perform the load). We deliberately introduce *`flush`*`(part2_limit)` on line 7 so that computing which direction the branch goes will take many cycles (due to the DRAM access). This makes our attack easier to implement, as we have many cycles before the CPU figures out that the branch should not have been taken.

Attack Outline: Below are the steps required to leak a single byte. You may need to alter your approach to account for system noise.

1. Train: Train the branch predictor to expect to perform the load (take the "if" branch) by calling the

method many times with a small offset.
2. Flush: Flush the memory region from the cache using `clflush`.
3. Victim execution: Call the victim method to leak a given secret byte past the limit during speculative execution.
4. Reload: Reload the memory region, measure the latency of accessing each address, and use the latency to determine the value of the secret.

Common pitfalls: From your experience in Lab 1, you should be used to the fact that side channel attacks generally do not work on the first attempt. We provide several tips below that may help you if you get stuck.

• Noise: It is common to observe noise in performing any types of side channel attacks. Calling the victim method just one time might not produce the most accurate results due to noise. You might benefit from repeating the attack multiple times and use statistical methods to decode the secret to achieve a higher precision.

• Branch predictor: Branch predictors involve significant complexity in modern processors. Since the branch predictor is shared between userspace and kernel space, if the speculation is not working as expected, you may need to reduce the number of branches in your code.

• Mixing measurements and output: In general with cache-based side channel attacks and Spectre-style attacks, it is a bad idea to print latencies while measuring them. It is best to perform all measurements in one loop, storing the results in an array, and then printing them in a second loop. The reason is that any operations that you perform on the array will affect the branch history and potentially pollute your cache state.

Exercise 4: Implement the Spectre attack in `attacker-part2.c` to leak the secret string. Build the project with `make` and run `./check.py 2` from the lab 2 main directory to check whether you have passed part 2 or not.

Discussion Question 3: In our example, the attacker tries to leak the values in the array `secret_part2`. In a real-world attack, people can use Spectre to leak arbitrary values in the victim's address space. Explain how the attacker can achieve this.

Discussion Question 4: Try to tune the training parameters and answer this question: What is the fewest number of times you need to train the branch on line 9 in Listing 3 to make the attack work?

Interacting with the Kernel Module. We have provided a helper method called `call_kernel_part2` that you can use to interact with the kernel module, just like part 1.

Submission, Grading, and Checkoff: There is no checkoff for Part 2. This part is graded in the same way as Part 1.

- You need to submit your code (`part2-src/attacker-part2.c`) to the Canvas submission.
- Include your discussion question answers in a text file named `solutions.txt`. This file should include the discussion questions from all parts of lab 2. Full credit will be awarded to solutions that report the correct secret more than 95% of the time. Partial credit will be awarded for solutions that behave worse than that. Each attempt should take no longer than 30 seconds.

3: Advanced Spectre (25%)

Now that we've gotten Spectre working, let's try a harder version of the same problem. Below is the pseudocode for Part 3:

```
part3_limit = 4
def victim_part3 (shared_mem, offset):
     if offset < part3_limit:
          false_dependency = long latency computation resulting in 0
             secret_data = part3_secret[offset]
             mem_index = 4096 * secret_data
             load shared_mem[mem_index + false_dependency]
```

Listing 3: Part 3 Victim Pseudocode

There are two differences in the victim code compared to the version in part 2. First, the victim no longer flushes the limit variable before the branch. Second, we have added a long latency false dependency to the memory access, which means the memory access starts later in the speculation window. If you are curious, you can check the code in `labspectrekm.c` for details.

> **Exercise 5**: Copy your code in `run_attacker` from `attacker-part2.c` to `attacker-part3.c`. Test whether your previous implementation works with the new victim.

You should see your attack does not work with the new victim any more. The reason is that in the modified victim code, the memory access instruction that we try to monitor may not be issued speculatively for three reasons.

First, the speculation window becomes shorter. The speculation window starts in the cycle the branch enters the processor, and ends in the cycle where the condition is resolved. If the `part3_limit` variable is cached, it will take a very short time to return to the core, and the branch will be resolved very quickly. As a result, the speculation window becomes shorter. Second, the long latency dependence on the memory access means that the long latency chain of instructions must be completed before the speculative load can begin. It is possible the branch condition is resolved before the speculative load even starts. Third, as discussed in the Background section, there is a TLB miss the first time the kernel tries to access a line in the shared memory region. For parts 1 and 2, we secretly resolved this by forcing page walks on the shared memory region for you. However, in part 3 we do not force page walks before running the vulnerable code. As a result, the memory load must first perform a page table walk, which increases the time it takes to perform the load (thereby increasing the chance the speculation window closes before your address makes it to the cache).

To make your attack work, you will need to find a way to increase the speculation window so that the speculative load has a higher chance of succeeding.

Hint: You cannot change the long latency memory address dependency, nor can you change the fact that a TLB miss occurs when performing the load.

> **Discussion Question 5**: Describe your strategy to extend the speculation window of the target branch in the victim.

> **Exercise 6:** Optimize the attack in `attacker-part3.c` to leak the secret string. Build the project with `make` and run `./check.py 3` from the lab 2 main directory to check whether you have passed part 3 or not.

Interacting with the Kernel Module. We have provided a helper method called `call_kernel_part3` that you can use to interact with the kernel module just like parts 1 and 2.

Submission, Grading, and Checkoff: You need to submit your code (`attacker-part3.c`) to Canvas. There is no check-off for Part 3. We understand the attack may not work 100% reliably due to noise,

non-determinism, etc. We will give full credit if the attack can completely leak the secret at least once out of 5 attempts. Each attempt should take no longer than 10 minutes. We will give partial credit if the attack can recover most of the characters.

> **Discussion Question 6**: Assume you are an attacker looking to exploit a new machine that has the same kernel module installed as the one we attacked in lab 2. What information would you need to know about this unknown machine to port your attack to the new system? Could it be possible to determine that information experimentally? Briefly describe in 5 sentences or less.

Partial credit will be awarded for solutions that are functional but do not meet the accuracy requirements. In general, if `check.py` reports "Success," then you will receive full credit for that part. You can check all parts at once with `make` and then `./check.py all`.

Tips

- Calling the victim method just one time might not produce the most accurate results due to noise. You might benefit from making helper methods- one to perform the attack once, and one to repeat it a few times, and report the most common response.
- Don't forget you can flush whatever address you want with the `clflush` helper method!

Acknowledgments

This was based off the MIT's course 6.5950/6.5951 at shd.mit.edu

References

[1] Wenliang Du. *Spectre Attack Lab*. https://seedsecuritylabs.org/Labs_16.04/System/Spectre_Attack/. 2019.

[2] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. doi: 10.1109/SP.2019.00002.

[3] Moritz Lipp et al. *Meltdown*. 2018. arXiv: 1801.01207 [cs.CR].