Daniel Oliveros

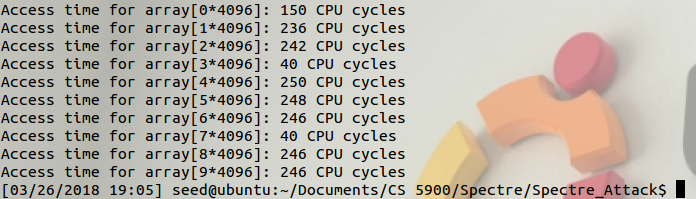
Garrett Bogart

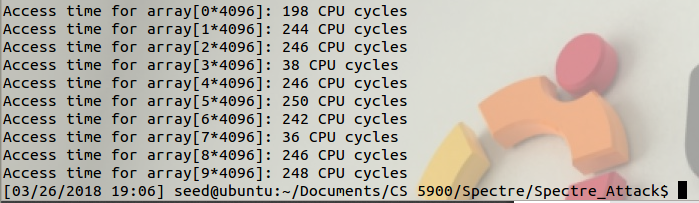
Spring 2018 – Independent Study

Spectre Vulnerability Lab

**Task 1:**

This program is used to test the differences between memory in the cache and memory outside the cache. Memory in the cache is accessed significantly faster. This can be seen by [3\*4096] and [7\*4096] where the memory is stored in cache and is accessed in less than 100 cpu cycles.

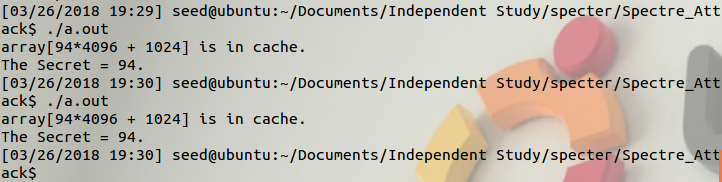




The threshold for whether or not the memory accessed was located in the cache or not seems to be around the 100 CPU cycles mark. Compared to when it’s in cache, it takes typically around 5-8 more times the amount of CPU cycles to access memory that’s not in it.

**Task 2:**

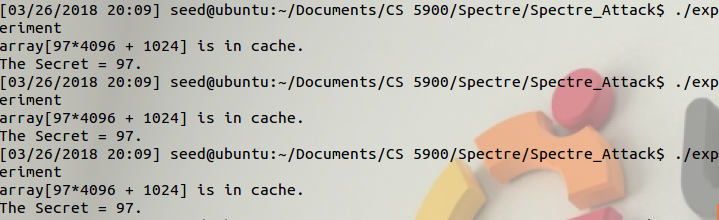
This program uses the FLUSH+RELOAD technique to flush out all memory in the cache, run a vulnerable program, and then proceed to try to access various locations in memory and determine, based on how long it took to access this memory, whether or not it was placed in the cache by the vulnerable program, it then prints this array location out.



**Task 3:**

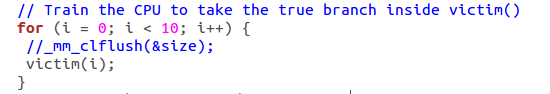
This task takes advantage of out of order execution. When a program hits an if statement and the parameters needed for it are not in the cache then the computer runs predictive branching. Predictive branching looks at previous runs of the program to determine the most probable path the branch will take. The program then runs this branch and stores the memory. Once the parameter is retrieved from memory a check to see if the computer took the right branch. If the cpu took the right branch the memory is kept, if not, the idea was to throw the memory away. The problem is that the memory is kept in the cache. If a program was supposed to keep some data secret in an if statement then by purposely allowing predictive branching to occur you can steal the information from the cache without actually having permission to see the data itself.

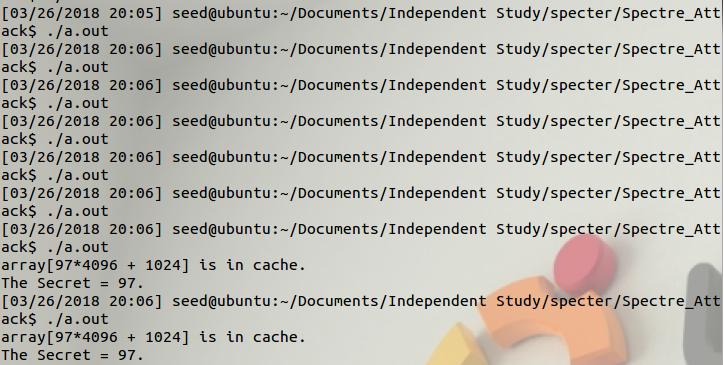
**Part 1**

****

**Part 2**

Once these lines are commented out, the program is much less reliable at finding the correct data. This is because it is no longer flushing the size variable out of the cache, making it so speculative programming does not occur. This means that the program is now unable to be trained to go down a wrong path, and highly reduces the likelihood of size being out of the cache when the program is run in a way that tries to exploit predictive branching

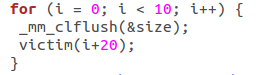
****

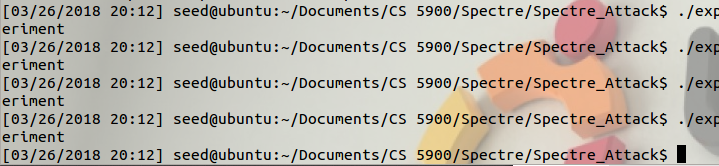
****

Sometimes the program does execute correctly, this occurs when the program accidentally loads memory over the location of size in cache, making it so the program has to fetch it when needed

**Part 3:**

This exploit works because when a program predicts an outcome and it fails the data it used for this prediction is not removed from the cache. By giving it the value i+20, the program is trained to never expect the if statement in the victim program to be executed. So, when we try to exploit the vulnerability, the issue that comes up is that the program does not go down a wrong path where it accesses the data like we would like, but instead it just waits until the data is fetched since it didn’t even expect the data retrieval process to be executed.

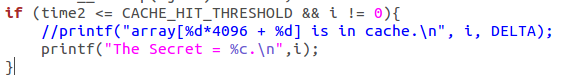
****

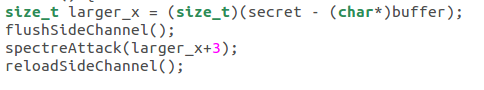
****

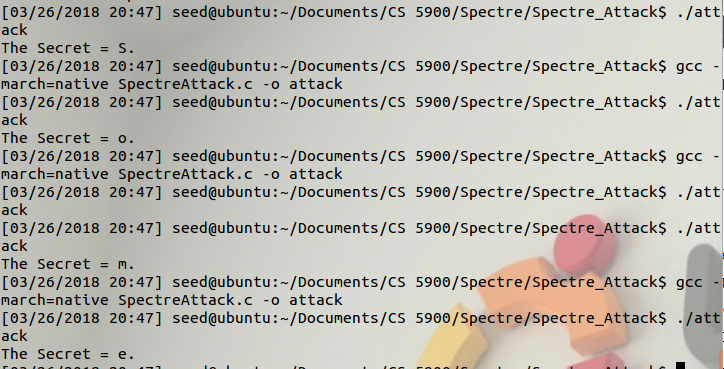
**Task 4:**

The specter attack takes advantage of out of order execution and looking at cache memory. After getting the predictive branching setup so that the restricted path is taken the specter attack can begin. The specter attack flushes the cache and then lets the out of order execution take place. That memory is stored in the cache and we can use a side channel attack to see what value was stored in the cache. This gives us access to restricted memory.

We got rid of some of the noise by adding an extra condition to the if statement that triggers the print statement. We were able to get multiple pieces of the secret message by adding values to the parameter being passed into the spectreAttack function. This attack still fails fairly often, as sometimes changes in the cache take place unexpectedly, or access time takes longer than we’d like.



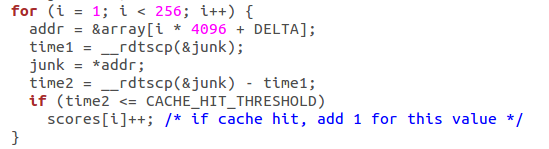


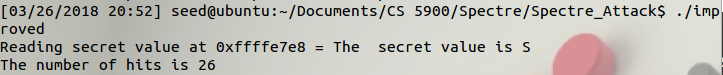


**Task 5:**

This task uses statistical analysis on the specter attack to find the most probably piece of memory that is the restricted memory. We came up with two solutions to ignore the 0 element in the statistical analysis.

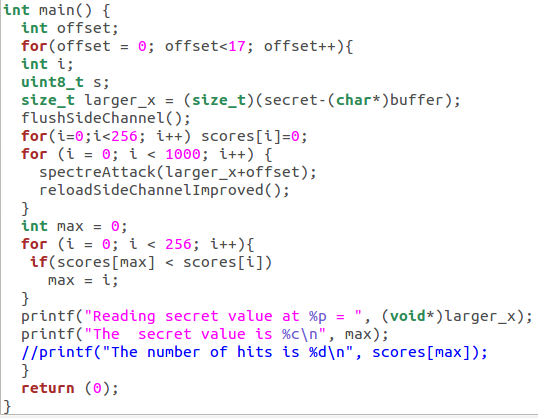
****

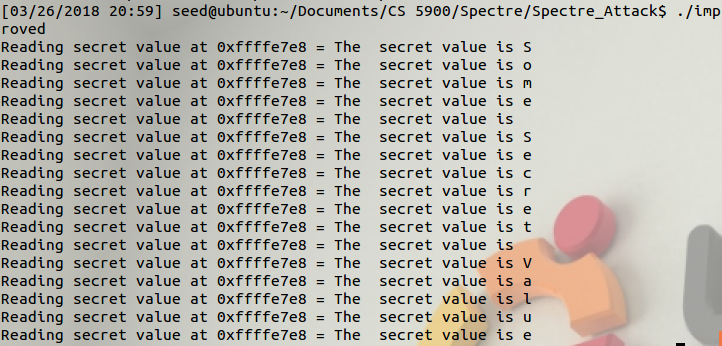
****

****

**Task 6:**

In this task we modified the statistical analysis attack to show the entire secret message.





**Thoughts on this lab:**

1. This lab is very easy to accomplish but provides a ton of insight into how these vulnerabilities work, and how one may exploit them.
2. Based on a student’s setup, this lab may be impossible for them to run in the future, as the issue is deeply rooted in the architecture of modern processors, and future versions of these should be addressing these issues
3. We’d recommend this lab as a short way of gaining insight into this situation, showing a very recent and important attack as well.