Project Write-Up

Introduction:

In this lab we will investigate some of the different types of side channel attacks and create a mock model of a possible attack on multi-user systems that contain modern computer architecture. This attack will merge two separate side channel vulnerabilities and allow for plaintext and cryptographic keys to be stolen.

Background:

Side channel attacks are a type of attack that specifically targets the normal operating behavior of a system's hardware to exploit possible leaked information as the computer performs encryption and everyday operations. There are 7 types of side channel attacks: electromagnetic, acoustic, power, optical, timing, memory cache, and hardware weaknesses. Each of these types are ways of either inferring information (electromagnetic, acoustic, power, optical, timing) or abusing a system to gather information (memory cache, hardware weakness). In many instances side channel attacks are just methods to abuse "features" that are added into hardware for espionage style attacks. This lab will specifically focus on power and hardware weakness attacks. The power analysis attack will allow cleartext to be inferred from computations during encryption and the hardware weakness attack will allow portions of the cryptographic key to be accessible inside the registers of the system.

Side channel attacks are common and frequently patched out of systems and hardware. A few of the major past attacks that have been patched out include MeltDown, FallOut, and many others. Common side channel attacks that have been patched out include FallOut [16] and MeltDown [15]. FallOut specifically investigates the information that is leaked from "Store Buffers." These buffers are used every time data is moved through the CPU and therefore can contain important information about the system [3]. FallOut is an example of a hardware weakness as it uses a bug in the store buffers to gain extra information about the system. FallOut is so volatile of an issue that it even allowed an unprivileged user to choose what information was leaked by implanting the type of command they wanted to find information about. For example, if they implanted an encryption algorithm in the system then they could get information about the encryption such as clear text, keys, and cipher text from the store buffers.

Meltdown allows a program to access the secret values contained within a different program and in the operating system [5]. Meltdown works by targeting speculative execution which is the way a computer speeds up "jump" assembly instructions. Since the program is trying to predict what will happen during compilation it will jump to where it believes it should go. When it is incorrect it reverts to where it was and follows a different path [5]. This makes it possible to trivially measure the timing differences between when a branch's speculative execution succeeds and when it fails (speculative execution is the predictions the hardware makes about the next instruction). Knowing whether the jump was taken in the code allows inferences of what information is being stored in the cache. Meltdown is an example of a memory cache timing attack vulnerability as it abuses the cache to learn extra information about the system by measuring the differences in time.

DownFall:

In the version of the attack we will perform in this lab, we will look to access the values obtained with the gather instruction during an AES encryption using OpenSSL. The gather instruction is an

instruction designed to load multiple elements at a time to make CPU vector calculations much quicker. Although this speeds up the performance of CPUs which have this instruction its introduction caused a hardware weakness vulnerability into the CPUs. This should expose portions of the cryptographic key used during encryption. Since this attack utilizes the values contained on a CPU, it is only viable for machines which have multiple users that can access it. For example, a server setup which allows many users to access the system all at the same time (Isengard, and other cloud providers are examples of this) would be vulnerable since all accounts would be using the same set of architecture. The DownFall attack requires precise control over access the vector registers for the gather instruction to work properly. This means different CPUs have different assembly code that need to be run to gain the necessary access making the attack require at least a bit of knowledge of cybersecurity and computer architecture to pull off.

Lab Start:

With DownFall being the attack performed in this lab there are still some things that must happen before the attack can begin. One is a victim encryption must be performed to ensure the registers have begun storing information while the encryption is happening. The second is knowing the plaintext used in the encryption. This is required for the attack to function so this lab begins with a power analysis attack to determine the clear text that will be used in conjunction with DownFall to steal cryptographic keys. For a power analysis attack the attacker looks at the amount of power being used by the system for specific operations. For example: a bit containing the value 1 typically uses slightly more computing power than a bit containing the value 0 so an attacker can use this information to infer what bit value is set in the system.

Understanding and employing a power analysis attack:

For the first step in the lab, you have been given access to the file “bellcurve.py”. In this file there is a function called “distribution” that will return a value which will represent power consumed by the CPU at the time the function is called. Every time this function is called, we simulate the ability to test a CPU’s power consumed when inputting a specific bit value (commonly in real world attacks the information that can be inferred comes in full bytes or words not single bits). These values will differ slightly each time the function is run as in the real world the power value recorded won’t be the exact same for each call to perform operations on bits. Call the distribution function many times (at least 1000) and store each value in a list to create a distribution of values that can be graphed to determine relations. Now create a histogram using the pyplot.hist function to see the pattern created and report your findings to Canvas.

Demonstrate understanding of the power analysis by deciphering the plaintext used in the encryption:

We have provided a table of power usage values in the plaintext.txt file. These represent the values recorded from a target machine after an encryption has been performed using OpenSSL. Using the pattern you found above, convert the values to plaintext to find the plaintext used in the encryption. This can be done manually or with a small script to convert the power values to plaintext. Report your findings to Canvas.

Hint: the plaintext you create above should be a familiar word or phrase not random characters.

Understand why a victim encryption needed to happen for the attack to work on the same system:

In step 3 of the lab, we need to perform our own encryption to create the encryption we will attempt to attack. In the terminal run: make clean && make. This will reset and update all of the files and allow the next command to run. Now run ./openssl_encrypt_128.sh. This will perform the encryption on the plaintext value you found in the first part and store the encrypted value at the path: /tmp/downfall.txt.encrypted.

Understand why a victim encryption needed to happen for the attack to work on the same system:

Now we will look at finding Qwords and utilize the gather instruction properties to determine the encryption key that was used on the plaintext found in part 1. A Qword is a value representing 8 bytes or 64 bits of data. This matches the size of the register that is used when OpenSSL encryption occurs. This also is one of the locations in which parts of the key will be transferred through to perform the encryption. Lots of data moves through this register so it's not as simple as accessing the register and seeing what values exist in it to find the key; we first must do an encryption ourselves using OpenSSL to force the machine to start trying to encrypt a new value. By doing this the encryption will require a new key and force out the previous key through the register. We simulate this action with the "GetQwords.py" file. Inside of this file there are two functions for you to use: getQword1 and getQword2. Each of these will simulate the information gained from different registers for 10 iterations of the gather instruction being called. Most of the time this will force the first and second portions of the key to be pushed out through this register and be collected by our program. The first Qword represents possible matches for the first 8 bytes of the encryption key and the second Qword represents possible matches for the last 8 bytes. In between these being called the system will push the information for the first half of the key out of the system and begin attempting to find the second portion of the key. Call the getQword1() function and store the returned values in a list. These represent the possibilities for the first half of the encryption key. Then call the getQword2() and store the returned values in another list. These represent the possibilities for the second half of the encryption key.

Perform the combination of Qwords and compare encryptions to show understanding of Qwords

After retrieving the list of the first and second Qwords, we have no way of knowing for certain which combination of the Qwords would represent the key. However, since we have nailed down the space to be a relatively small number of combinations, we can do a brute force solution to find the key. Write a program to use each combination of Qwords (of the form "[Qword1][Qword2]") to create a key for the encryption. Then encrypt the values using this command:

subprocess.open(['openssl', 'aes-128-cbc', '-salt', '-e', '-in', '/tmp/downfall.txt', '-out', '/tmp/downfall.txt.encrypted2', '-K', KEY,'-iv', '11111111111111111111111111111111'])

*In the function above we use /tmp/downfall.txt rather than the plaintext you found since the process expects a file not plaintext. The file's contents contain the same plaintext you found above.

This command will perform an OpenSSL encryption with AES-128 cbc mode and store the value in downfall.txt.encrypted2. The original encrypted value is stored in downfall.txt.encrypted so comparing these values to one another will determine what the key is. When opening the files for both encryptions make sure to include the parameter for encoding="latin1" in the open statement so it knows how to interpret the files correctly. Report your findings to Canvas.