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. 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 the power and heat output generated by a CPU to determine plaintext and see how CPU registers they behave when performing encryption on a device.

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 Spectre, Meltdown, Fallout, and many others. The Spectre attack looks to target something called branch prediction. Since branch jumps in assembly level code are expensive in the time it takes to run normally. To counteract this, computer architects have made branch predicting possible to allow a program to correctly select where it will jump to correctly. This speeds up performance greatly, but Spectre allows this to be tampered with to grab hidden information from trying to get the CPU to predict incorrectly where it is going to jump to. Meltdown is a similar attack to Spectre in that it investigates race conditions that exist within instruction execution and privilege checking in the CPU. When this race condition is triggered, information is leaked through a side channel to allow the attacker to grab any information that was given to the privilege checking systems which can be secret keys, usernames and passwords. Fallout is a vulnerability which allows malicious code to read data from the Store Buffers which typically store data when transferring between the cache and main memory. Downfall, the vulnerability we will focus on for this lab, is a combination of MeltDOWN and FALLout.

Downfall:

Downfall utilizes the ability to grab information from buffers that Fallout was known for as well as be able to utilize a bit of Meltdown's ability to access information that wouldn't normally be accessible to the attacker. The vulnerability itself takes place inside the gather register on modern CPUs. During encryption, and a few other operations, this register is used to store different parts of data flowing through the machine. This can contain things such as partial cryptographic keys, passwords, and username values, along with other random information that could be useful to an attacker. In the version of the attack we will do in this lab, we will look to access the values contained in the gather register during an AES encryption using OpenSSL. 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 is an example of this) would be vulnerable since all accounts would be using the same set of architecture. This attack is very precise on how to access the gather register so different CPUs have different assembly code that needs to run to gain the necessary access making the attack require at least a bit of knowledge of cybersecurity and computer architecture to pull off.

For this lab we will perform a known plaintext style attack for DownFall. We will however need to find this plaintext so we will do something known as frequency and power analysis to determine the value that was encrypted. In this type of attack the attacker looks at the different frequencies and power emitted from a device to learn information about the system.

Step 1:

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. 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. A bit containing the value 1 typically uses slightly more computing power than a bit containing the value 0. Call the distribution function many times 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.

Step 2:

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. Report your findings to Canvas.

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

Step 3:

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.

Step 4:

Now we will look at finding Qwords and utilize the gather register 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 gather register that is used when OpenSSL encryption occurs and happens to be where the different parts of the key is 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 gather 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 the gather register for 10 iterations of encryption. 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.

Step 5:

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.