Lab 5 - Flush+Reload Attack against AES

Due on Mar. 24^{th} (T) 2020

1 Overview

In this lab, you will learn a widely used cache side-channel attack: Flush+Reload attack. It relies on a special x86 instruction, *clflush*, and an OS optimization, *memory deduplication*. *clflush* instruction evicts cache lines corresponding to a specified memory address. *memory deduplication* merges memory pages with the same data that belong to different processes, meaning when two processes load the same file into the main memory, only one copy of the file exists in memory and both processes share it.

You will use the *clflush* instruction to monitor the usage of AES T-tables in the OpenSSL library. Based on the usage pattern, you will recover the last round encryption key.

This lab can be done on any X86-Linux platform that supports the instruction *clflush*. To test if your computer supports the instruction, run "$cpuid | grep clflush" or "lscpu |grep clflush." Note on some case-sensitive systems, you may need to use uppercase CLFLUSH. On MAC system, you may use "sysctl -a |grep CLFSH". If you don't have access to any other good machine, please contact the instructor or the TA and you will be directed to use the class workstation, which, however, may have high noise when multiple users are running their code simultaneously on the machine.

2 System Profiling

To understand the cache hierarchy of your system, you will first need to determine the timing characteristic of the main memory and caches on your testing platform. In this step, you need to create a micro-benchmark to measure the time distributions for last-level cache hits and misses, respectively. Show the timing distribution plots in the same graph with two different colors. Based on the graph, determine an appropriate threshold that can be used to separate cache hit and cache miss from the measured time.

3 Using OpenSSL Library

We have used the OpenSSL library for Lab 1. Download it again from https://github.com/openssl/openssl.git and refresh your memory. Note that we will use the library differently as Lab 1, and therefore you will need to check it out and compile again on your machine.
To checkout the repository:

```
$ git clone https://github.com/openssl/openssl.git
```

To compile it on your machine:

```
$ cd openssl
$ ./config no-hw no-asm
$ make
```

Note that it is important to have *no-hw* and *no-asm* flags during the Makefile configuration. These two flags are enforcing to use the T-table implementations and also avoid using the AES-NI instruction extension (the instruction extension activates some hardware accelerator for AES operations without involving T-tables and caches). Once it is compiled, you will see *libcrypto.so* in the OpenSSL directory. This is the shared library that will be dynamically loaded with the victim and attack binaries.

4 Preparing Flush + Reload Attack

Finding T-table Address: For Flush + Reload, the spy uses the *clflush* instruction to monitor one selected cache line of a specific T table. For example, you can choose the first cache line of the first T-table *Te0*. To do that, you will need to determine the offset of your target within the shared library. You will first disassemble the shared library and locate your target. You can use the following command to do so:

```
$ readelf -a libcrypto.so > ~/aeslib.txt
```

Then you can see:

```
73: 0000000000256720 1024 OBJECT LOCAL DEFAULT 15 Te0
74: 0000000000256320 1024 OBJECT LOCAL DEFAULT 15 Te1
75: 0000000000255f20 1024 OBJECT LOCAL DEFAULT 15 Te2
76: 0000000000255b20 1024 OBJECT LOCAL DEFAULT 15 Te3
```

where 0x256720 is the offset of Te0 etc. Note on your platform the offsets will be different from the above example.

Setting Up the Attack:

The Flush+Reload attack will involve two processes running simultaneously: victim and attacker, which are set up in a server-client model. The victim is the server, which receives the plaintext input from the client (attacker), performs the AES encryption, and returns the ciphertext to the client. The attacker collects samples of ciphertext and execution time. You are given the two c files, where the victim.c (server) file is complete but the attacker.c is incomplete and you should fill in. You still need to study victim.c file to understand what it is doing before moving onto working on attacker.c. Once you finish attacker.c, when compiling these two files, you need to compile them against the shared library (libcrypto.so) you created in the previous step.

Most of the code for attacker.c is already given except for one line to set the monitoring target and a function called doTrace(). The target setting line is in function of init(), which is to figure out the memory address for the given offset in the shared library (e.g., the first cache line for Te0). Check out the source code of attacker.c and find the function to use for setting the target memory address.

```
// setup the target for monitoring
    printf("setting up target\n");
    target = ;
```

Function doTrace() implements the major steps for one round of Flush+Reload operations, and is called in an iterative loop (e.g., 1M times). For collecting one time sample, the operations are: 1. generate a random plaintext; 2. flush the corresponding memory target address (using clflush); 3. send the plaintext to the server (victim) for encryption; 4. on receiving the ciphertext, reload the memory address and time it; 5. save the trace (the ciphertext and timing sample). All the function macros are already included in the attacker.c file and you just need to instantiate them in doTrace appropriately. Hints for steps are given in the function definition place.

```
void doTrace()
{
    // generate a new plaintext
```

```
    // set the cache to a known state
    // ask victim for an encryption
    // check current cache state
    // record timing and ciphertext
#ifdef DEBUG
    printText(ciphertext, 16, "ciphertext");
    printf("Timing: %i\n", *timing);
#endif
}
```

Note this lab involves socket programming. If you are not sure how to edit the victim's listening port, it would be better if you could run everything on your own machine.

Once you have completed the attacker source code, you can proceed to compile both attacker and victim sources. Note that you need to link the shared library you just created. The compilation example given below assumes you have put the attacker.c and victim.c in one directory above the root directory of OpenSSL (you have downloaded and compiled).

```
$ gcc attacker.c -Iopenssl/includes -Lopenssl/ -o attacker -lcrypto
$ gcc victim.c -Iopenssl/includes -Lopenssl/ -o victim -lcrypto
```

When you run an executable (e.g., victim or attacker) compiled with a shared library (also called a dynamic library), you will need to tell the loader where to find that library. You can set the *LD_LIBRARY_PATH* environment variable in this way: export LD_LIBRARY_PATH=[openssl root directory]. You can refer to https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html if you don't know how to run the binary.

5 Attack - Online Phase: Collect Time Samples

You need two terminals: on one terminal, you run the victim, and on the other terminal, you run the attack to collect 1 million samples. First, for the samples collected you can run the analysis program on them to identify information leakage (on which cipher/key bytes there is information leaked - with outliers). The code analysis.c is already given to you and you need to compile it and run it on your samples. The analysis program calculates the average reload time for each cipher value at each byte position. Use *plot_analysis.py* to plot the result from the analysis program. If you have successfully monitored one cache line of a lookup table, you should observe 16 outliers as shown on Slide 54 of Lecture 5. The Python environment is the same as the previous lab.

Question 1. For your samples, are the outliers in the same column of subplots (on four cipher/key bytes), or actually in two columns? If you observe the latter case, explain it and describe how you can utilize such information leakage to recover additional key bytes (refer to slide 63 of Lecture 5)?

6 Attack - Offline Analysis: Recover Last Round Key Bytes

To recover key bytes, you will need the threshold you found in Section 2 and samples you collected in Section 5. For each sample, first, you use the threshold to determine whether to keep this sample or not - you can either keep the samples with lower timing or with higher timing (attack strategy 1 and 2, respectively as shown on Slide 61 and 62 of Lecture 5). Discuss the pro and con of each method. If you choose samples with lower timing, use the following algorithm to recover each key byte value individually. The k value with the highest counter is the correct key.

For the positions with information leakage j (e.g., j = 2/6/10/14 when monitoring Te0), set a counter Cnt[j]=0;
For each kept sample {C^q, T^q}, the ciphertext is c_j^q

```
    For k_j = 0...255
       1.
       Calculate Inverse_Sbox[k_j xor c_j^q] => s^q[k_j]
       2.
       Check s^q[k_j] < 16?
       3.
       If so, increment the counter Cnt[j][k_j]
k_j = argmax Cnt[j][k_j]
```

By monitoring one address of one T table, you should be able to recover four key bytes.

You can refer to https://eprint.iacr.org/2014/435.pdf and https://ieeexplore.ieee.org/document/8203771 for AES key recovery methods.

7 What You Need to Turn In

Turn in two files. One is a zipped folder of all your code, with clear README (how to run your code) and Makefile.

Another is a PDF report with the following items:

1. Figures that show the timing distributions for L3 cache hit and miss. Your timing threshold
2. The plots generated using *plot_analysis.py* on the samples your collect.
3. Answer to Question 1.
4. Report 4 last round key bytes you have recovered together with corresponding plots.

8 Extra Credits

1. (5 points) For scenario where information is leaked on two columns of cipher/key bytes, recover other four key bytes corresponding to another T table (e.g., Te1 and therefore key bytes 3, 7, 11, 15 as shown on Slide 58).
2. (5 points) Keep changing the T tables to monitor and repeat the attack until you recover all the last-round 16 key bytes.
3. (5 points) Implement a different attack strategy from the one you have chosen: keep samples with lower or higher timing, and use argmax or argmin for the counters. Discuss the pro and con of each strategy.