Assignment 3: Chat Program Using Hardware Covert Channel

CS544, Spring 2019
Start: March 13th, Due Date: April 12th 11:55 EST

1 Introduction

In this lab you will build a chat client that can send messages from a sender to the receiver. The only rules are:

1. The sender and receiver must be different processes.
2. The sender and receiver may only use syscalls/shared library functions *directly accessible from the provided* `util.h`. Both sender/receiver may use *any* x86 instruction.

The twist here is that `util.h` only contains a memory allocator (like `malloc`) and some convenience functions for tracking system time. There is no way to set up a shared address space between sender and receiver, nor is there a way to call any obviously-useful-for-chat functions such as Unix sockets.

For this assignment, you must implement cross-process communication using a (very cool) notion known as *hardware covert channels*. For the duration of the lab you will have access to a processor that will be shared by the class. This processor will share its hardware resources (including cache, dram, processor pipeline, etc) with the various processes it has to serve. If the sender process places pressure on a hardware resource, it creates contention with other processes trying to use that same resource. Coupled with a mechanism to measure time between instructions, this resource contention can be turned into a reliable way to send information from process to process that violates software-level process isolation.

Please Note: that part 1 of this assignment is required for *everyone*. Part 2 of this assignment is only required for those who are *not doing the final project*. If you are doing a final project, then part 2 is *not required*. However, even if you are doing the final project, we highly recommend you to try part 2. It is really fun!

Motivation In The Matrix movie, released in 1999, there is a scene towards the beginning where Trinity secretly sends Neo a message. Here is a YouTube video of the scene. This is a secret chat program that (most likely) uses hardware covert channel.

The motivation for our chat program is this: Imagine you are Trinity (sender) trying to send a message to Neo (receiver) without letting The Matrix (OS) know what the message is. The simplest way of doing this would be to encrypt the message and send it to Neo. However, if you store the encrypted message in anywhere The Matrix can see (DRAM, Hard Disk, Network Interface Card, etc), The Matrix will know that you guys are communicating with each other and try to exterminate you. IYou can't even let OS know that you are communicating with each other.

This is where hardware covert channel chat program comes in. Your program should be written in such a way that you do not store the message (or a variation of the message, such as the ciphertext) anywhere in the memory. This includes no shared memory between the sender and the receiver, or no exchanging messages through sockets. Instead, for each bit (1 or 0) you wish to transfer, the sender will place pressure on a hardware resource (cache, dram, processor, pipeline, etc) and the receiver will decode the bit by observing the resource contention that the sender creates. Which resource to exert pressure is completely up to you.

The techniques used for creating covert channel is also quite similar to the techniques you learned for Meltdown or Spectre. As such, after you complete the first part of the assignment, you will know how to

write stealthy malware and break process isolation without tunneling through the OS. By completing the second part of the lab, you will also be able to circumvent many state-of-the-art software and hardware security mechanisms designed to ensure data privacy. If you choose to perform all the tasks in the second part of the assignment, you will be overly prepared to execute tier-1 research in shared resource attacks, which is a hot area in system/hardware security.

2 Part 1: 70 points (Required for everyone)

Reminder: Part 1 is required for everyone, whether you are doing the final project or not.

To complete the assignment, you must implement a hardware covert channel chat client which behaves similarly as the TA solution (which is provided as a pair of executables). To run the TA solution, ssh into the test machine (more details below), open two terminals and `cd` into directories containing the provided sender(`send`) and receiver(`recv`) executables:

```
On terminal A: > taskset -c i ./TA_send   # Set i to 0 through 3
On termianl B: > taskset -c j ./TA_recv    # Set j to i+4 (your setting of i plus 4)
```

Terminal A will prompt you to type a message (followed by enter). Terminal B will prompt you to press enter. Press enter for terminal B first - this will tell the receiver to start listening over hardware covert channels for message sent by the sender. Now type a message on terminal A and press enter; it should print out on the receiver side.

Messages may not come out perfectly every time. Hardware covert channels are noisy and the TA solution isn't perfect (yours doesn't have to be either). If the message doesn't come out correctly, try the above steps again, perhaps using different values for taskset. The most important thing is that *your only action after pressing enter on the receiver is typing a message on the sender.* When the receiver starts listening, it can't tolerate the processor hosting other applications aside from the sender.

Why does the sender/receiver require keyboard input before proceeding? Setting up a covert channel can take time and this can interfere with the other party's measurements. We allow both sender and receiver to do any setup they need to do before the key press, so that when they enter their main loops they see the system in a clean state, and any required synchronization is complete.

Lab write up. In addition to your code, please submit documentation that describes: (a) how to use your chat program, (b) a description of how you establish the covert channel, (c) which part2 project you completed (Section 3) and (d) what challenges you ran into when working through your solution. Please also feel free to provide feedback on the lab, and describe things you tried that did not work. This feedback will be very helpful in improving the lab for future classes!

Use of taskset. Notice the TA solution uses a special command `taskset` to launch the sender/receiver. By setting `-c i` and `-c i+4` for the sender/receiver, we are forcing the OS to pin the sender and receiver to different hardware threads sharing the same physical core (the test machine has 4 physical cores with two hardware threads each). Using `taskset` in this fashion dramatically reduces signal-to-noise problems for hardware covert channels, yet, does not detract from the key concepts needed to complete the lab. For part 1, your sender and receiver program can be on the same core. That being said, one of the available tasks in part 2 is to implement the program without needing `taskset`.

Checkoff procedure. Your solution will be built from source on the test machine (see below) and verify that *your* implementation of the sender works with *your* implementation of the receiver. You are not required to talk to others' solutions or to the TA solution. **Note on debugging:** while debugging your solution you may use any header files/mechanisms you like. The restriction on headers only applies to the final code you submit.

Use of helper functions (e.g., STL). The point in limiting your use of shared library/header files is to teach you how to exploit hardware covert channels for fun and profit. The point is *not* to force you to re-implement convenience functionality (e.g., STL's vector/set/etc data-structure) in raw C++. Thus, if you would like to use convenience code (e.g., STL) that keeps to the spirit of the lab, please feel free. Again, the purpose of restricting the headers files you can use is so that you use covert channel to create the chat program, not so that you have to implement everything from scratch. If you aren't sure, ask on the sakai forum and we will reply ASAP.

OS cracking. If you wish to break process isolation, you can either use hardware covert channels, or try and break into the operating system. Please do not do the latter even if you have a way, since it would denial-of-service the rest of the class. (If you have a way to do that in Ubuntu 16.04, notify the proper people through the proper channels).

Test machine. Final lab checkoff will be done on `prof2.cs.rutgers.edu`. (Specs for this machine are given at the end of the lab document.) We created an account in prof2 for each student. Your username is your netid and the password is your 9-digit Rutgers student ID. As we have all learned the importance of security, make sure to log into prof2 and change your password ASAP, to eliminate the chance of any other students stealing your work. If such event happens, you will be flagged for cheating. The simplest method of changing password is to use the command `passwd` in the terminal. Also, to coordinate students as much as possible:

1. Please test on your personal machines to the extent possible. When choosing which covert channel to use, compare your machine to the specs included at the bottom of the lab document. You can parameterize your chat client to work for multiple machines in most cases. That said, it is **definitely** a good idea to test on the prof2 machine before final submission. Anything can come up when porting a hardware covert channel to a new machine. We recommend validating your program on the prof2 machine as soon as possible to relieve last-minute pressure on that machine.

2. We created a sakai forum specifically for you to coordinate usage of prof2 machine.

3. We have two additional machines, C211-i2.cs.rutgers.edu and C211-i3.cs.rutgers.edu from ilab. Both if these machines have hyper threadding and SGX enabled. However, the machine specs, including core speed (i7-6700K), is most likely different from prof2 machine, so be careful. Also, grading will still be performed on prof2, so it is best to test on prof2 machine before final submission.

Please use these resources and follow standard practices for sharing resources (kill your processes after you are done, etc). This lab can be a lot of fun, but if someone continually denial-of-service the prof2, C211-i2, and C211-i3 machines, it may turn into a mess.

Tips to Get You Started. Building a covert channel is analogous to building any other communications channel. At the bottom level, you need a physical layer to transport the lowest-representation of information. In traditional communications, this may be applying a potential over a wire to toggle from 1 to 0 or vice versa. At the receiving side, a circuit must be capable of sensing change on the wire. In our setting, the physical layer is the sender putting different patterns of pressure on some shared hardware resource. The receiver and sender must have agreed ahead of time on which shared resource to use and what constitutes "pressure." Then, the receiver may use a timer (we provide some examples in util.cpp) to measure this pressure. We give you complete freedom in choosing all of the above: what channel to use, how to interpret signals on that channel, etc, as long as you follow the rules at the top of the document. You may follow or suggestions or the techniques used in the academic papers we link in this description, but you may also find your own way of creating a covert channel.

After we have a physical layer, we need a protocol stack to turn our hardware covert channel "wire" into a full-fledged chat client. Protocol stacks may include a de-noising layer (e.g., a special encoding of the message, replaying noisy messages, etc) and higher layers which are optimized for the specific type of communication (e.g., chat clients). For an example of a simple higher-level protocol, we suggest reading about UART (Universal asynchronous receiver-transmitter), which is an extremely common protocol for low-speed communication in digital systems, popular for its simplicity. We also give you complete freedom in designing/choosing a protocol to use on top of your physical layer.

Both of the above components influence each other. Some physical layers may be inherently higher bandwidth, but require additional de-noising. All choices require assumptions on the underlying system, so we strongly recommend planning out your solution given prof2's stats at the end of the document.

To re-iterate: your solution does not need to interoperate with another students' solution or the TA solution. Two students' solutions will clearly not be able to speak to one another if they use a different physical or protocol layer.

3 Part2: 30 points (Required if not doing final project)

Reminder: Part 2 is required if you are not doing the final project.

The chat program that satisfies the bare minimum requirement for part 1 will grant you 70 points out of 100. In order to receive all 100 points, you have to create an improved version of the chat program. There are many ways the chat program can be improved upon. We have provided some of the improvements that you can do to the chat program below. We ordered roughly from what we think is easiest to hardest. In order to receive full credit, your chat program has to implement any one of the improvements listed below. However, we highly encourage you to implement as many improvements as you can, as these are cutting-edge exploits found in recent research.

- **Speed** Increase the bandwidth of your covert channel. When the sender hits enter to send its message, the TA solution sends at a rate of approximately 60 Bytes/second, which is pretty slow. Optimize your solution to send messages x10 faster, while still maintaining similar accuracy as the original solution (i.e., might take several tries, but does manage to get perfect accuracy).
  Note: the following code placed after the gets() call in the sender can be used to measure code execution time in seconds.

```
clock_t begin = clock();

/* put your sender loop, after the gets() call, here */

clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

- **Speed++** Further increase the bandwidth of your covert channel. Some of the academic papers listed below provides you with techniques that can increase the badwidth by x100 compared to the TA solution. For this extension, your goal is to make your program be x100 faster than TA solution.
- **SpyInEnclave**. Implement the chat program with the *sender* running in an SGX enclave. Prof2 is running the Linux SGX SDK and we have included a hello-world program for SGX programming for you to get started. It will also be beneficial to read the enclave writer's guide posted below in useful resources section. **If you manage to get this far, your code has broken/bypassed Ryoan.**
- **SpyEncrypted**. Encrypt messages within your SGX enclave and communicate ciphertexts. *Prerequisites:* SpyInEnclave.
- **SpiesEverywhere**. Extend your SGX enclave-enabled chat program so that the *receiver* also runs in an SGX enclave. **If you manage to get this far, you now know how to write cutting-edge malware.** *Prerequisites:* SpyInEnclave.
- **Portability**. The chat program for part 1 requires the sender and receiver to be run on different hyperthreads in the same processor. Hyperthread technology is not available in all commercial CPUs (i.e. Intel i3, i5, and even some version of Intel i7 do not support hyperthreadding). For this improvement, implement the chat program without using `taskset` on either sender or receiver. In another words, the chat program should be functional even if the sender and the receiver are running in different cores. For credit, your implementation needs to have similar typing accuracy as the core submission, but has no requirement on transfer rate. **If you get this far, you have implemented a cutting-edge shared resource attack.**
- **Portability++**. Implement **Portability** without using RDRAND/RDSEED (i.e. random shuffling).
- **TheMasterMind**. Implement the chat program without using `rdtsc` or other "get time" instructions. **If you get this far, your code would have bypassed most defense mechanisms proposed by the computer architecture community (based on fuzzing/blocking timers).**

These are only some of the improvements that we thought of. If you have ideas for any other improvements, please let us know. We can decide whether it will be sufficient for part 2.

Extending the lab into the final project. Alternatively, you can extend the ideas from this assignment to design your final project. A major part of doing a final project is to come up with your own idea. As such, you cannot exactly reuse the ideas from this assignment or any other assignments. However, you can create your final project ideas by extending what you can learn from this assignment.

4 Useful Resources

Here are some useful statistics regarding prof2 machine that your final submission will be tested on. Feel free to call CPUID/benchmark the machine yourself to gather additional information (as long as you play nice and announce your activity on the sakai forum). Also feel free to ask about a particular stat that isn't listed.

```
Processor: Intel i7 7700K @ 4.20GHz
Extensions: SSE/MMX variants, AES, SGX, RDRAND/RDSEED
4 physical cores, 2 HW threads per core
Cache hierarchy:
  Per physical core:
    L1 instr Cache:   32 KB, 8-way
    L1 data Cache:    32 KB, 8-way
    L2 unified Cache: 256 KB, 8-way
  Shared last-level Cache: 8 MB, 16-way, 4 slices
Main memory: 2 x 16 GB DDR4 @ 2400 MHz
Prefetching @ L1/L2 enabled
Turboboost: Disabled in BIOS
Page size: 4 KB, hugepage can be used (by using madvise() function)
OS: Ubuntu 16.04 LTS
Graphics: Nvidia GP107GL (Quadro P1000)
```

The following academic papers may also prove useful in learning about different hardware covert channels:

1. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures; R. Guanciale and H. Nemati and C. Baumann and M. Dam; Oakland'16.
2. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack
3. Last-Level Cache Side-Channel Attacks are Practical; F. Liu, Y. Yarom, Q. Ge, G. Heiser, R. B. Lee; Oakland'15.
4. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks; P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard; Security'16.
5. Malware Guard Extension: Using SGX to Conceal Cache Attacks.
6. Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations; D. Evtyushkin, D. Ponomarev; CCS'16.

Of course, there are many other academic papers related to side-channel covert channel attack. You can refer to any other academic papers for guidance as long as it follows our rules specified in the Introduction.

5 Building SGX Enclaves

We have installed the Intel SGX SDK/PSW on prof2 machine so that you can build and experiment with hardware debug enclaves. "Hardware debug" means the enclaves you write use the real SGX hardware extensions, however, have mechanisms for debugging. Thus, these enclaves are good for testing covert channels but not fit for production. Here are some resources that may be useful to start building programs in SGX Enclave:

1. The enclave writer's guide (posted at the bottom of the course website) is a nice resource for learning how to build SGX applications.
2. Alongside the lab starter code, we have included "hello world" enclave code that you can use to get started

To compile the enclaves in prof2, you first have to add the SGX SDK to your PATH by using this command:

```
source /opt/intel/sgxsdk/environment
```

You only have to use this command once per log in. Verify that you can build and run the hello world enclave without errors. **Please be aware of the following:** SGX enclave stack/heap size is controlled in `hello-enclave/Enclave/Enclave.config.xml`. If you try to allocate/run enclaves beyond that size, you will see strange undefined behavior. If you need more space, just change that file.