Hands-On Hardware Side Channels Lab

6.888: Secure Hardware Design, Fall 2020

Mengjia Yan, Miles Dai

Due: October 14, 2020

1 Introduction

This lab explores the practical exploitation of hardware side channels. In the first part, Dead Drop, you will see how side channels can allow you to bypass process isolation guarantees by creating a covert channel to send information between two processes that are unable to communicate using traditional means. In the second part, Capture the Flag, you will fine-tune the concepts you used in Dead Drop to steal a secret from a victim program by simply observing the cache.

To get started with the lab, see the `README` in your Github repo located at

`https://github.mit.edu/6-888-fa20/lab-<kerberos>`

2 Dead Drop: An Evil Chat Client

In this lab, you will build Dead Drop: an evil chat client that can send messages between two processes running on the same machine. The only rules are:

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

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

With Dead Drop, you must implement cross-process communication using hardware covert channels. Suppose the sender and receiver run on the same physical machine at the same time. Processes on this machine share hardware resources (including caches, DRAM, processor pipelines, etc). If the sender process uses a hardware resource, it creates contention with other processes trying to use that same resource. Coupled with a mechanism to measure contention (such as a timer), this can be turned into a reliable way to send information from process to process that violates software-level process isolation.

2.1 Requirements

Allowed code: As mentioned above, the sender and receiver may only use syscalls and the functions accessible from the provided `util.h` except for `system()`. In addition, you may use any x86 instruction in your final submission except for `clflush`. The point of limiting your code usage is not to force you to re-implement convenience functions. If you would like to use some convenience code that stays within the spirit of the lab, please contact the course staff. Obviously, you may not use pre-packaged code from online for building covert channels (e.g. mastik).

Execution environment: You will be given SSH access to a lab machine to run your code on. For reasons discussed more below, you will be assigned two CPUs on this machine which are executing as SMT threads to run your code on. Please be sure to use only these CPUs to avoid interfering with other students who may be running experiments on the machine.

Expected behavior: The Dead Drop client should behave in the following way. In two different terminals running on the same machine, you should be able to run the following commands:

```
1: Terminal B: $ make run_receiver
2: Terminal B:   Please press enter.

3: Terminal A: $ make run_sender
4: Terminal A:   Please type a message.

5: Terminal B: $
6: Terminal B:   Receiver now listening.

7: Terminal A: $ 47

8: Terminal B:   47
```

In other words, in lines 1 and 2, you start the receiver process in a terminal, and it prompts you to press enter to start listening for messages. Until you press enter, the receiver is not expected to echo messages from the sender. In lines 3 and 4, you start the sender in another terminal. Lines 5 and 6 are you pressing Enter to start the receiver process. It should now be ready to receive messages. In line 7, you type a message and hit enter on the sender side. Line 8 shows the same message appearing on the receiver side.

For the basic version of Dead Drop, you only need the ability to send integers from 0-255 (inclusive). For optional extensions to the project, see Section 2.5

Checkoff: For checkoff, you will push the final version of your code to your assigned Github repository. You will also upload a description of (a) how your implementation works, (b) what challenges you ran into, (c) what worked/didn't work, and (d) any feedback you have for this lab for future years. You and the course staff will also set up a 10-minute checkoff, during which you will open up two terminals on the lab machine and run `make run_sender` in one and `make run_receiver` in the other to demo your exploit.

2.2 Restrictions

There are two notable restrictions for this lab. First, we are prohibiting the `clflush` instruction which is commonly used in flush-reload attacks. In practice, some architectures do not support this instruction. Furthermore, some side-channel mitigations suggest making `clflush` a privileged instruction or disabling it entirely. Thus, while you may use `clflush` to help you debug and experiment, your submission will need a different technique to evict cache lines.

The second restriction is that your processes will be running on two specific CPUs. This is done to simplify the conditions for this lab. The two CPUs you have been assigned are a pair of SMT (aka Hyperthreading) thread contexts which run on the same physical core and share all the hardware resources on that core such as private L1 and L2 caches. Your implementation will only need to work on your assigned cores. This is because a successful exploit across any arbitrary pair of cores would require a resource shared between all cores on the machine which makes this problem significantly more challenging. Percival describes the threats associated with SMT in more detail in *Cache Missing for Fun and For Profit* [3].

2.3 Getting Started

The following are some suggested steps to help you get started which guide you towards one possible solution, namely a cache-based side channel. There are many other hardware side channels that can be exploited.

You are free to explore other options if you so choose, but using the cache side channel will be more useful for the next part of the lab.

2.3.1 Setting Up Your Environment

Before starting, test your SSH access to the lab machine. Next, modify the `SENDER_CPU` and `RECEIVER_CPU` variables in your `Makefile` to your assigned CPUs. Using tmux, screen, or simply two SSH connections, you should be able to run `make run_sender` in one terminal and `make run_receiver` in another terminal. These two make recipes will compile your code and start the two processes on the correct CPUs using `taskset`. **IMPORTANT: Set `SENDER_CPU` and `RECEIVER_CPU` to your assigned values before running code for any portion of this lab. Otherwise, you may be interfering with another student's lab.**

The files included for the Dead Drop portion of the lab are

- `sender.c, receiver.c` - template code for your sender and receiver respectively
- `util.h, util.c` - utility functions and library imports you are permitted to use (except for `clflush`)

2.3.2 Determine the Machine Architecture

Before exploiting any hardware side channels, you must first understand the machine architecture. There are a few commands that can help with this on Linux.

- `lscpu`: provides information on the type of processor and some summary information about the architecture in the machine.
- `less /proc/cpuinfo`: provides detailed information about each logical processor in the machine.
- `getconf -a | grep CACHE`: displays the system configurations related to the cache. This will provide detailed information about how the cache is structured.

In addition, WikiChip provides a lot of information specific to each processor and architecture. You may find the pages on the Intel Xeon Gold 5220R processor and the Cascade Lake architecture useful.

2.3.3 Timing a Memory Access

Take a look at the provided `measure_one_block_access_time` utility function in `utils.c` which measures the length of time required to access a particular memory address.

Try using this function to measure the access time of a memory address. Then try using `clflush` to evict the memory address from the cache and into DRAM, and measure the access time again. You should notice a large ($> 100\%$) increase in the latency. (Remember that you should not use `clflush` in your solution; this is just to demonstrate the memory access latency difference.) Can you achieve a similar result without using the `clflush` instruction? In other words, can you create a cache eviction without `clflush`?

2.3.4 Detecting a Cache Eviction Across Processes

Because we are unable to create a shared memory space between the two processes, traditional flush+reload or evict+reload attacks cannot be used. Instead, we can explore the possibility of using a prime+probe style attack. Can you implement a basic prime+probe function to detect cache evictions? You may find the `clflush` function helpful for debugging. In addition, you may find the C5 Covert Channel paper helpful in guiding your methodology [2].

2.4 Common Pitfalls

Virtual Address Abstraction: Keep in mind that the addresses you are dealing with in your C code are virtual addresses. However, physical addresses (which you will not have access to within your code) may be used when indexing into lower level caches. You may need to consider how address translation interacts with cache indexing/tagging as well as run some experiments to see if this presents a problem for your cover channel implementation.

For a review of virtual memory, please refer to the lecture slides or to past 6.004 lectures (FA19-L18, SP17-L16).

2.5 Optional: Dead Drop Extensions

There are several extensions you can implement for bonus points that are not required for the checkoff or for full points on the lab.

- Reduce the transmission time by using a single cache set as a covert channel instead of the entire cache. Note that this will require understanding how set indexes are derived from addresses.
- Add support for text transmission instead of just the integers from 0-255.

2.6 Acknowledgments

The original Dead Drop lab was developed by Christopher Fletcher for CS 598CLF at UIUC. The starting code and lab handout are both heavily adapted from his work.

3 Capture the Flag (CTF)

In this part of the lab, you will be attempting to extract a secret value from a victim program. To keep the lab manageable, we have created a simple program, `victim.c`, that performs a secret-dependent memory access (pseudocode in Listing 1). Your task will be to write a program, `attacker.c` that determines this secret value.

```
// Set flag to random integer in the
// range [0, NUM_L2_CACHE_SETS)
int flag = random(0, NUM_L2_CACHE_SETS);
printf(flag);

// Allocate a large memory buffer
char *buf = get_buffer();

// Find four addresses in buf that all map to the cache set
// with an index of flag to create a partial eviction set
char *eviction_set[4];
get_partial_eviction_set(eviction_set, target);

// Main loop
while (true) {
    for (int i = 0; i < 4; i++) {
        // Access the eviction address
        (*(eviction_set[i]))++;
    }
}
```

Listing 1: victim.c process pseudocode

3.1 Requirements

With the description of the victim program provided above, you should be able to extract the secret flag value from a second program running on the same core.

Allowed code: The code constraints for the CTF are identical to that of Dead Drop. In addition, you are allowed to use the `sys/mman.h` library for the purpose of allocating memory only (i.e. `mmap()` or `posix_memalign()`). Once again, use of the `clflush` instruction and the `system()` function are not allowed.

Execution environment: The execution environment for CTF is identical to that of Dead Drop. You will use two terminals, one to run the victim binary and one to run your attack code. They will be running on the same pair of SMT thread contexts as in Dead Drop.

Expected behavior: The victim code will print out the randomly chosen flag before it enters the infinite loop. Your code, upon running, should perform some operations to determine the flag, then clearly print the flag as shown in the starter code. While you are free to print whatever debugging information you may find useful, please try to keep your terminal output relatively clean for the checkoff.

Checkoff: As with Dead Drop, you will upload your code to your assigned Github repository. In your scheduled checkoff time, you will open up two terminals on the lab machine, run `make run_victim` on one, and run `make run_attacker` on the other to demo your exploit and show that the two printed flag values are the same.

3.2 Setting Up Your Environment

You will use the same setup as the Dead Drop portion of the lab. **IMPORTANT: Make sure `SENDER_CPU` and `RECEIVER_CPU` are set to your assigned values in the Makefile before running code for any portion of this lab. Otherwise, you may be interfering with another student's lab.** The files included for the CTF portion of the lab are

- `attacker.c` - template code for your attacker
- `victim` - victim binary that you will be stealing a secret from. Note that you will not have access to the victim source code, only the pseudocode provided above.
- `util.c, util.h` - utility functions and library imports you are permitted to use (except `clflush`)

Using tmux, screen, or simply two SSH connections, you should be able to run `make run_victim` in one terminal and `make run_attacker` in another terminal. These two make recipes will compile your code and start the two processes on the correct CPUs using `taskset`. If you have problems running the `victim` binary, you may need to run `chmod +x victim` within your lab directory.

3.3 Common Pitfalls

Virtual Address Abstraction: Similarly to the situation in Dead Drop, address translation may present an issue with your attack. To help mitigate some of the virtual address problems, we have enabled hugepages on the machine. You are free to use 2 MiB hugepages if you need them for the CTF portion of the lab only. See Appendix B for help on how to allocate a hugepage.

Cache Replacement Policy: While least recently used (LRU) is the most common example of a cache replacement policy, practical processor implementations often use much more complex policies. You may need to experiment a bit to roughly figure out how eviction works on your processor.

Furthermore, be sure to carefully consider how you are accessing your cache lines. With prime+probe attacks, it is common to encounter *thrashing*, a situation in which the attacker primes the cache set only to evict their own data with subsequent accesses while probing. One way to avoid this problem is to prime the cache by accessing the eviction set in one direction and probe in the reverse direction, as described by Tromer et al. [4].

Hardware Prefetching: Modern processors have hardware that can prefetch data into the cache before it is required as it attempts to anticipate future accesses. This may cause spurious cache evictions or mask evictions caused by the victim. This hardware can also be triggered by linear, fixed-stride access patterns within a page. To avoid these issues, one technique is called "pointer-chasing," in which the eviction set addresses are randomly ordered and then organized as a linked list in which each each address stores the address of the next member of the eviction set [4]. Liu et al. concretely show how to traverse this list quickly with a few lines of assembly [1][1].

Cache Pollution: While prime+probe is a simple attack in theory, it suffers from high levels of noise partially due to cache pollution issues. Keep in mind that everything involving a memory access will use the cache, not only data but also instruction fetches. You will need to carefully manage the interval between when you prime and when you finish probing to minimize the number of cache evictions caused by accesses other than the victim process.

Noisy or Inconsistent Results: Because side channels exploit shared resources in unintended ways, the resulting covert channels can be quite noisy. Modern processors often contain optimizations that make them behave differently from the simplified architectures taught in class. This lab requires experimentation to find working approaches and values. You should not expect your solution to work on the first attempt, so be sure to incrementally build up your solutions and verify that each step is working before proceeding.

Appendix A Useful Functions

There are some useful functions and libraries included in the starter code.

- `time.h` provides the `clock()` function which returns the current number of clock ticks. Thus two calls to `clock()` can be used to measure elapsed CPU time.
- `clflush()` is a wrapper around the `clflush` x86 instruction which will flush a particular memory address from all levels of the cache meaning that the next fetch to that memory address must come from DRAM.
- `measure_one_block_access_time()` is a utility function written in assembly that measures the amount of time required to access a particular memory address using the `rdtsc` instruction. Note that additional `lfence` instructions are used to prevent the processor from reordering the timing-critical instructions.

Appendix B Linux Hugepages

The default page size used by most operating systems is 4K (2^{12}) bytes. You may find it useful to use a larger page size to guarantee consecutive physical addresses. In particular, Linux (as of 2.6.38) supports *transparent huge pages* (THP) for private anonymous pages, which will allow programs to allocate 2 MiB pages, ensuring 2^{21} consecutive physical addresses.

Allocating Hugepages

THP relies on the `madvise` syscall to tell the kernel to back the virtual pages with hugepages. The following sample code demonstrates how to allocate a single hugepage.

[1]You do not need to write your own assembly to complete this lab, but it can make for cleaner results.

```c
1     void *buf = NULL;
2     int buf_size = 1 << 21; // one hugepage
3     if (posix_memalign(&buf, 1 << 21, buf_size)) {
4         perror("posix_memalign");
5     }
6     madvise(buf, buf_size, MADV_HUGEPAGE);
7     // the buffer can now be used
8     *((char *)buf) = 1;
9
```

Listing 2: allocating a hugepage

Note that if you are monitoring the use of hugepages on the system, you will not see anything until you actually write to the hugepage (i.e. line 8) due to copy-on-write mechanisms.

Monitoring Hugepage Usage

The implementation of THP is rather complex but essentially involves the kernel scanning the areas marked as hugepage candidates to replace them with huge pages. Thus hugepages may not be immediately available when requested by the software.[2]

As such, it may be useful to monitor the hugepage usage on your system. One approach is to use `getchar()` to pause program execution after allocating the hugepage and monitoring the system-wide hugepage usage.

To get more information about hugepages on a Linux system, you can use `cat /proc/meminfo | grep Huge`. This will give you information about the size of hugepage configured on the system. Furthermore, if any hugepages are currently allocated, they will be visible under `AnonHugePages`.

You can use the `watch` command to continuously monitor the output of a command. For example, the following command will poll the output of the `cat /proc/meminfo` command above every 0.5 seconds and highlight the changes (Ctrl-C to quit):

```
watch -d -n 0.5 "cat /proc/meminfo | grep Huge"
```

In addition, you can get a list of all your processes that are using hugepages with the following shell command

```
grep -e AnonHugePages -s /proc/*/smaps | awk '{if($2>0) print $0}'
```

The victim code uses one hugepage. You can use this to test the above commands. Run the commands with and without the victim code in a separate terminal. You should see an increase in the number hugepages used.

References

[1] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy (SP). San Jose, CA: IEEE, May 2015, pp. 605–622. ISBN: 978-1-4673-6949-7. DOI: `10.1109/SP.2015.43`. URL: `https://ieeexplore.ieee.org/document/7163050/` (visited on 09/18/2020).

[2] Clémentine Maurice et al. "C5: Cross-Cores Cache Covert Channel". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Magnus Almgren, Vincenzo Gulisano, and Federico Maggi. Vol. 9148. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 46–64. ISBN: 978-3-319-20549-6 978-3-319-20550-2. DOI: `10.1007/978-3-319-20550-2_3`. URL: `http://link.springer.com/10.1007/978-3-319-20550-2_3` (visited on 09/18/2020).

[3] Colin Percival. "CACHE MISSING FOR FUN AND PROFIT". In: (), p. 13.

[2] `madvise` only advises the kernel to take action but does not guarantee that the kernel will follow the advice.

[4] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23.1 (Jan. 2010), pp. 37–71. ISSN: 0933-2790, 1432-1378. DOI: `10.1007/s00145-009-9049-y`. URL: `http://link.springer.com/10.1007/s00145-009-9049-y` (visited on 09/18/2020).