Dead Drop: An Evil Chat Client

Christopher Fletcher (with major contributions from Read Sprabery and Mengjia Yan)

598CLF "Secure Processor Design" lab, Fall 2019, UIUC
Start: Sept. 19, Due: Oct. 24 11:59 CST
Document version 1.0

1 Introduction

In this lab you (and optionally a partner) 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/shared library functions *directly accessible from the provided* `util.h`. Both sender/receiver may use *any* x86 instruction.

The twist is `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.

With Dead Drop, you must implement cross-process communication using a (very cool) notion known as *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 cache, dram, processor pipeline, 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.

Why is Dead Drop evil? Once you complete this lab, you will know how to write stealthy malware and break process isolation without tunneling through the OS. By completing some of the extra credit (see below), you will also be able to circumvent many state-of-the-art software and hardware security mechanisms designed to ensure data privacy. By completing all the extra credit, you will be prepared to execute tier-1 research in shared resource attacks, which is a hot area in system/hardware security.

Lab completion. To complete the lab, you must implement a Dead Drop client which behaves in the following way. In two different terminals running on the same machine, you should be able to enter these commands:

```
1: On terminal B: > ./receiver
2: On terminal B:   Please press enter.

3: On terminal A: > ./sender
4: On terminal A:   Please type a message.

5: On terminal B: >
6: On terminal B:   Receiver now listening.

7: On terminal A: > Hello world!

8: On terminal B:   Hello world!
```

In other words: (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.[1] (3) and (4) you start the sender process in another terminal. (5) and (6) you press enter in the receiver process. It should now be ready to receive messages. (7) you type a message and hit enter on the sender side. (8) the same message appears on the receiver side.

You can optionally use `taskset` to pin the sender and receiver to a particular hardware thread context. Modern machines support multiple (2 on Intel machines) thread contexts per physical core due to SMT/Hyperthreading. By pinning sender/receiver to the same physical core, but adjacent hardware thread contexts, it becomes easier to design some covert channels. Here is an example of `taskset`:

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

This machine has 4 physical cores. This command pins the sender to physical core 0, thread 0, and receiver to physical core 0, thread 1. You will have to lookup how many physical cores your machine has and set j accordingly (see below for machine stats). Do not try to pin multiple processes to the same physical core and same hardware thread at the same time!

Messages may not come out perfectly every time. Hardware covert channels are noisy and your solution doesn't have to be perfect. If the message doesn't come out correctly, try the above steps again. Solutions will be accepted as long as the staff can type some messages of their choosing and see them come out $\sim 100\%$ correct most of the time. When running the sender/receiver, your machine should otherwise be unloaded (CPU utilization as close to 0% as possible).

Checkoff procedure. For checkoff, you will share a github with the course staff containing your code and a description of (a) how your implementation works, (b) what challenges you ran into, (c) what worked/didn't work, (d) any extra credit you performed (see below), and (d) any feedback you have for the lab for future years. You and the course staff will also setup a 5-10 minute timeslot to demo your solution (thus, the solution must run on your laptop, or be otherwise SSH accessible from your laptop).

Your sender is only expected to be able to send messages to your receiver. That is, it does not have to interoperate with other groups' receivers.

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. If you aren't sure, email the course staff. For example, it is fine to use STL vectors to pass data around, but not fine to use `sendmsg` in `socket.h` if such function is accessible from an include of an include of `vector.hpp`.

You may not use pre-packaged code from the web for building covert channels (e.g., mastik). Obviously.

2 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

[1] This feature is optional. Alternatively, your receiver can immediately start listening for messages after being launched. We have found that having the receiver wait for a signal, after the sender has launched, helps avoid synchronization issues between the sender and receiver.

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) 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.

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 the stats for your particular machine (see end of document).

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

How to build hello world. When building covert channels, it is best to crawl-walk-run: i.e., verify each level of the communications protocol piece by piece before trying to test the whole channel. For example, instead of trying to send full messages at first, start by verifying you can send one bit reliably (as discussed in class). Then build a protocol that can send multiple bits. Then apply proper handshaking so that the stream has low error rate. Etc.

Importantly, each of you will be running your code on a different machine. Each machine might be slightly different in its physical characteristics, so it is best to write some simple tests to see what the important parameters are for your machine (e.g., cache miss latencies).

3 Extensions

We hope that you have a lot of fun implementing Dead Drop, as we did. The lab completion requirements (referred to as "Core") are the minimum you have to do to get full credit. We have put together the following extra credit extensions that you can complete if you wish (ordered roughly from what we perceive to be easiest to hardest):

10pts **TRX**. Implement bi-directional communication. I.e., both processes can simultaneously function as sender and receiver.

1-30pts **SpeedRun.** Increase the bandwidth of your covert channel. When the sender hits enter to send its message, our (simple) solution sends at a rate of approximately 61 Bytes/second, which is pretty slow. Optimize your solution to send messages at a faster rate, while still maintaining 100% accuracy. Points are allocated as follows: 10× higher bandwidth is 10 pts, 100× is 20 pts. The last 10 pts are awarded at the course staff's discretion.
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;
```

30pts **AnyCore**. Implement Dead Drop without using `taskset` on either sender or receiver, and without RDRAND/RDSEED or mmap (see below regarding partial credit). For credit, your implementation needs to have similar typing accuracy as the core submission, but has no requirement on baud rate. This represents a state-of-the-art microarchitectural covert channel: one which works across physical cores, and does not rely on any shared addresses.

30pts **Timeless**. Implement Dead Drop 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).

If you have other lab extension ideas: email cwfletch@illinois.edu. Ideas may be added to this document!

Scoring policy. Per the syllabus, the lab is worth 15% of the course grade. If your group earns ≥ 10 points of extra credit, you will earn 1% extra on your *overall course grade* (i.e., 6.6% of the lab's credit). ≥ 30 points earns 3% extra on your course grade, plus a special prize that will be announced in class. ≥ 60 points earns 5% extra on your course grade (1/2 a letter grade) as well as the above special prize.

Extending the lab into the final project. Covert/side channels are an active area of research. If this area interests you, we encourage final project ideas which extend what you have done in this lab.

4 Partial Credit

Building a covert channel is difficult. If you are unable to get your channel working, we offer an alternative submission plan which will get 80% of lab credit. In this submission plan, you are allowed to use `mmap`, or other facility, to share read-only memory between sender/receiver. You cannot write to this shared memory (which would make the lab trivial), only read. This allows you to build a FLUSH+RELOAD channel which has lower noise than some other channels. If you feel the need to take this option, please let the course staff know.

5 Useful Resources

A good starting point to build covert channels is to figure out the hardware stats for the machine you are running on. You can get this information from `/proc/cpuinfo` or by looking up your processor on ark.intel.com. For example, some statistics for the course staff machine:

```
Processor: Intel Skylake i7 6700K
Extensions: SSE/MMX variants, AES, SGX, TSX, 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, 8 slices
Main memory: 16 GB DDR4 @ 1066 MHz
Prefetching @ L1/L2 enabled
All cores overclocked up to 4 GHz
Turboboost: Unknown/not exposed in BIOS
Processor C-states: enabled
Page size: 2 MB (huge pages enabled)
OS: Ubuntu 16.04
Graphics: Nvidia GTX 970/Intel Integrated Graphics 530
```

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. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks; P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard; Security'16.

3. Last-Level Cache Side-Channel Attacks are Practical; F. Liu, Y. Yarom, Q. Ge, G. Heiser, R. B. Lee; Oakland'15.

4. Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations; D. Evtyushkin, D. Ponomarev; CCS'16.

6 Document Versions

1. Version 0.0: initial version. Lab originally administered for 598clf "Secure Processor Design", Fall 2017, UIUC (`http://cwfletcher.net/598clf.html`).

2. Version 0.1: revised pts / extra credit. Added another extra credit reward. Added clarification on use of STL.

3. Version 0.2: multiple revisions.

4. Version 1.0: document revisions for Fall 2019 CS598clf.