Dead Drop: An Evil Chat Client

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

598CLF "Secure Processor Design" lab, Fall 2017, UIUC
Start: Sept. 19, Due: Oct. 19 11:59 CST
Document version 0.1

1 Introduction

In this lab you and a partner will build Dead Drop: an evil chat client that can send messages from partner to partner. We will refer to partner A as the sender and partner B as 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 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*. For the duration of the lab you and your partner 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.

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 overly 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 same way 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 and receiver executables:

```
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)
```

Terminal A will prompt you to type a message (followed by enter/carriage return). Terminal B will prompt you to press enter/carriage return. Press enter for terminal B first - this will tell the receiver to start listening over hardware covert channels for messages 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. 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.

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. That being said, one of the extra credit tasks is to re-implement Dead Drop without needing `taskset`.

Checkoff procedure. The course staff will build your solution 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 other groups' 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. 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`.

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 `sbox.cs.illinois.edu`: the alienware box sitting in Chris' office. (Specs for this machine are given at the end of the lab document.) The whole class will get access to this box when the lab starts (the course staff will provide accounts to each group). This presents the following problem: there are more groups than resources on the processor and groups can interfere with each other. Fundamentally, we require a shared machine to launch hardware covert channels and due to the nature of the lab, we cannot use a typical batch submission system such as condor. To coordinate groups 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 sbox machine before final submission. Anything can come up when porting a hardware covert channel to a new machine. If you want to do some of the extra credit options (see below), we recommend finishing the checkoff requirement first and validating that on the sbox machine as soon as possible to relieve last-minute pressure on that machine.

2. We will be giving out single-channel access to the "Security and Privacy Research @ Illinois" Slack for everyone taking the class. We have setup a dedicated channel for people in the class to coordinate usage of the sbox machine.

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/some groups continually denial-of-service the sbox machine, it may turn into a mess.

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 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 sbox stats at the end of the document.

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

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. *Prerequisites:* Core.

1-30pts **SpeedRun.** 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 61 Bytes/second, which is pretty slow. Optimize your solution to send messages at a faster rate, while still maintaining similar accuracy as the original solution (i.e., might take several tries, but does manage to get perfect accuracy). Points are allocated as follows: 10× higher bandwidth than the TA solution 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;
```

Prerequisites: Core.

15pts **SpyInSandbox**. Implement Dead Drop with the *sender* running in an SGX enclave. The sbox machine is running the Linux SGX SDK and the course staff will be giving out some infrastructure for

using it in the coming weeks. In the meantime, reading the enclave writer's guide posted on the course website is a good primer for building SGX applications. **If you manage to get this far, your code has broken/bypassed Ryoan.** *Prerequisites:* Core.

10pts **SecureTheSpy**. Encrypt messages within your SGX enclave and communicate ciphertexts. *Prerequisites:* Core, SpyInSandbox.

20pts **TwoWaySpy**. Extend your SGX enclave-enabled Dead Drop 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:* Core, SpyInSandbox.

30pts **598CLF's version of the RSA factoring challenge.** While the lab is live, the course staff have a process pinned to core 0, thread 0 which is repeatedly sending the same message over and over on some hardware covert channel. Find the covert channel, reverse engineer the protocol being used by the pinned process to communicate bits and decode the message! As a primer to sanity check yourself, the message starts with "The Magic Words are ".
Hint: to co-locate to the same physical core as this process, use `taskset -c 4`.
Note: the process generating the secret message is not intentionally trying to obfuscate the message. The message is generated using what we believe is a reasonable and simple language for sending information. If you are familiar with digital logic analyzers, think of this assignment as constructing a digital logic analyzer for reading the pinned process' resource pressure. If you can print out the raw signal, decoding it should be within reach. *Prerequisites:* Core.

30pts **AnyCore**. Implement Dead Drop without using `taskset` on either sender or receiver. For credit, your implementation needs to have similar typing accuracy as the core submission, but has no requirement on baud rate. **If you get this far, you have implemented a cutting-edge shared resource attack.** *Prerequisites:* Core.

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. ≥ 120 points earns all of the above plus a special to-be-announced bonus prize.

Extending the lab into the final project. Covert/side channels are an active area of research. The extra credit tasks will show you where some hard problems are in this area / where there is deficiency in the literature. If this area interests you, we encourage final project ideas which extend what you have done in this lab.

4 Useful Resources

Here are some useful statistics regarding the sbox 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 class slack). Also feel free to ask the course staff about a particular stat that isn't listed.

```
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. (Assigned paper) Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures.
2. (Assigned paper) DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.
3. Cache attacks and countermeasures: the case of AES.
4. Last-Level Cache Side-Channel Attacks are Practical.
5. Malware Guard Extension: Using SGX to Conceal Cache Attacks.
6. Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations.

Additionally, the enclave writer's guide (posted at the bottom of the course website) is a nice resource for learning how to build SGX applications.

5 Document Versions

1. Version 0.0: initial version.
2. Version 0.1: revised pts / extra credit. Added another extra credit reward. Added clarification on use of STL.