Software Foundations of Security & Privacy 15316 Fall 2019 Lecture 1: Introduction

Matt Fredrikson
mfredrik@cs

August 26, 2019

Course Staff

Matt Fredrikson
Instructor

TBD
TA

Mundane news

Project Zero

News and updates from the Project Zero team at Google

Wednesday, January 3, 2018

Reading privileged memory with a side-channel

Posted by Jann Horn, Project Zero

Spectre & Meltdown

What's the big deal?

- "Efficiently" leak information via mis-speculated execution
- Read arbitrary virtual memory regions (including kernel)
- Bypass explicit bounds checks
- Violate browser sandboxing
- ...?

"Every Intel processor that implements out-of-order execution is potentially affected"

Timing channels

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
unsigned char value = arr1->data[untrusted_offset];
unsigned long index2 = ((value&1)*0x100)+0x200;
unsigned char value2 = arr2->data[index2];
```

Timing channels

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
unsigned char value = arr1->data[untrusted_offset];
unsigned long index2 = ((value&1)*0x100)+0x200;
unsigned char value2 = arr2->data[index2];
```

Step 1. Read some data from an arbitrary memory location

Timing channels

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
unsigned char value = arr1->data[untrusted_offset];
unsigned long index2 = ((value&1)*0x100)+0x200;
unsigned char value2 = arr2->data[index2];
```

Step 2. Isolate a bit of data from the read

- `index2` is `0x200` if bit is 0
- Otherwise, `index2` is `0x300`

Timing channels

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
unsigned char value = arr1->data[untrusted_offset];
unsigned long index2 = ((value&1)*0x100)+0x200;
unsigned char value2 = arr2->data[index2];
```

Step 3. Read from a location dependent on extracted bit

Timing channels

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
unsigned char value = arr1->data[untrusted_offset];
unsigned long index2 = ((value&1)*0x100)+0x200;
unsigned char value2 = arr2->data[index2];
```

Step 4. Time reads to `arr2->data[0x200]`, `arr2->data[0x300]`

- If `0x200` takes less time, then extracted bit was 0
- Otherwise, the extracted bit was 1

This last step is a result of the processor's data cache!

Progress

At this point, the attacker has accomplished:

1. Read an arbitrary bit of memory
2. Exfiltrate value of bit by timing cache hits & misses

Keeping track of assumptions:

1. Code doesn't check bounds on memory access
2. Code reads memory using untrusted, attacker-controlled index `untrusted_offset`
3. Targeted memory location won't cause segfault

Defensive programming: bounds checks

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
if (untrusted_offset < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

Speculative execution

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
if (untrusted_offset < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

- If `arr1->length` is not in cache, 100 cycles until it fetches
- Processor may begin executing inside branch anyway...
- If condition is false, results are rolled back like a transaction
- But not the cache!

Speculative cache leaks

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset = network_read(...);
if (untrusted_offset < arr1->length) {}
 unsigned char value = arr1->data[untrusted_offset];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

Attacker-controlled reads make measureable changes to the processor cache

Progress

At this point, the attacker has accomplished:

1. Read an arbitrary bit of memory
2. Exfiltrate value of bit by timing cache hits & misses

Keeping track of necessary assumptions:

1. ~~Process code doesn't check bounds on memory access~~
2. **Code reads memory using untrusted, attacker-controlled index** `untrusted_offset`
3. Targeted memory location won't cause segfault

Berkeley Packet Filter

Packet filters in Linux, BSD provided by usermode processes

- Filters are bytecode-interpreted or JIT-compiled, run *in kernel*
- Domain specific language for implementing filters
- Filter code can access arrays, do arithmetic, perform tests
- Triggered by sending data to associated socket

Google's Project Zero team showed how to create JITted BPF bytecode that opens a side-channel vulnerability

- Upshot: unprivileged processes can read all kernel memory
- Proof of concept demonstrated 2000 bytes/second

Javascript Interpreters

```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * 4096)|0) & (TABLE1_BYTES-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

This script causes V8 to JIT-compile vulnerable bytecode

- Leaks to cache-status of `probeTable[`n`*4096]` for $n \in [0..255]$
- Problem: Chrome degrades resolution of JS timer
- HTML5 *Web Workers* feature can open new thread, repeatedly decrement shared memory value for precise timing

Upshot: Untrusted websites can read memory of other sites (passwords, CC #'s, emails, …), extension data, browser settings, …

Mitigations

How do we fix it?

Unlike most vulnerabilities, doesn't seem patchable

- Problem enabled by both software + hardware issues
- Without hardware changes, no apparent universal fix

But there are software-based mitigations

1. Disable speculative execution *(expensive!)*
2. Disable caching *(way more expensive!)*
3. Fix speculative execution *(hardware changes?)*
4. In some settings: **don't index arrays on untrusted values**

But if you must...

```c
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* 0-padded to size 0xFF */
struct array *arr2 = ...; /* 0-padded size 0xFFF */
unsigned long untrusted_offset = network_read(...);
unsigned char value = arr1->data[untrusted_offset & 0xFF];
unsigned long index2 = ((value&1)*0x100)+0x200;
unsigned char value2 = arr2->data[index2 & 0xFFF];
```

Only when you have a good reason to require untrusted indexing,

- Make sure the target array never contains secrets
- Pad arrays and implement *logical sandboxing*
- Use a static checker to make sure you've done this correctly

Mitigations

How do we fix it?

Good question

- We probably don't know the full scope of the problem
- Without hardware changes, no apparent universal fix

But there are software-based mitigations

1. Disable speculative execution *(expensive!)*
2. Disable caching *(probably even more expensive!)*
3. Selectively disable spec. execution *(hardware changes?)*
4. **Never index arrays on untrusted values**
5. **Check untrusted code for side channels** *(sounds hard?)*

Ongoing research: provable side-channel security

Vale: Verifying High-Performance Cryptographic Assembly Code

Barry Bond*, Chris Hawblitzel*, Manos Kapritsos†, K. Rustan M. Leino*, Jacob R. Lorch*, Bryan Parno‡, Ashay Rane§, Srinath Setty*, Laure Thompson¶

* Microsoft Research † University of Michigan ‡ Carnegie Mellon University
§ The University of Texas at Austin ¶ Cornell University

Verifying and Synthesizing Constant-Resource Implementations with Types

Van Chan Ngo Mario Dehesa-Azuara Matthew Fredrikson Jan Hoffmann
Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
Email: channgo@cmu.edu, mdehazu@gmail.com, mfredrik@cs.cmu.edu, jhoffmann@cmu.edu

Verifying Constant-Time Implementations

José Bacelar Almeida
HASLab - INESC TEC & Univ. Minho

Manuel Barbosa
HASLab - INESC TEC & DCC FCUP

Gilles Barthe
IMDEA Software Institute

François Dupressoir
IMDEA Software Institute

Michael Emmi
Bell Labs, Nokia

Spectre & Meltdown: Takeaways

Security problems are numerous, can be subtle and challenging

- Speculative execution isn't exactly new...
- Addressing it requires deep expertise, app-specific mitigations

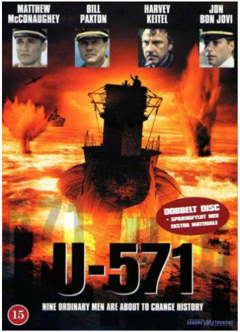
This course will teach you how to deal with hard security problems

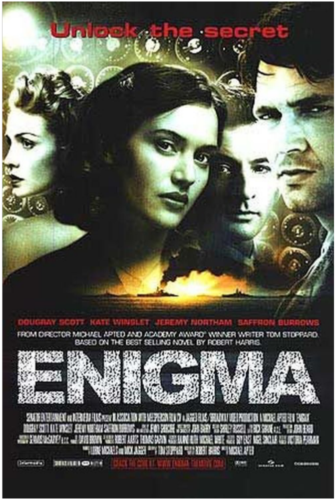
- Understand the essentials of many software security problems
- Evaluate potential solutions and their tradeoffs
- Implement strong defenses using principled techniques
- **Write code that isn't vulnerable in the first place**

Back to the course

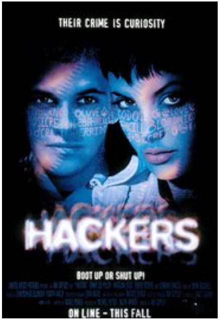
What is this course about?

This is not a course about encryption...

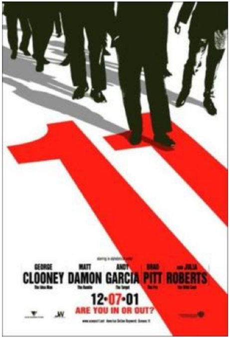


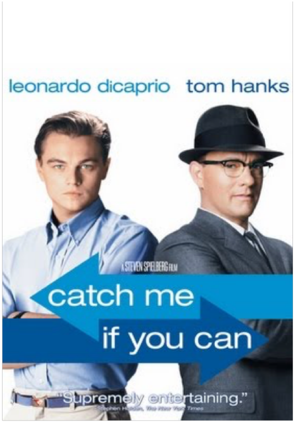


Not a course about hacking...

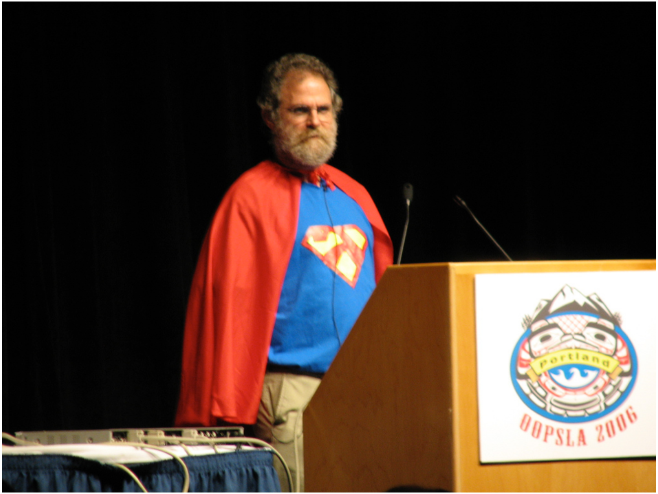


Not a course about social engineering...





This course is about...



How logic and languages will save us (and make software secure)

Making software secure: desiderata

Central theme: *security & correctness are often two sides of a coin*

A way to specify software behaviors that are secure, i.e. *policies*

- Who can see what data, and when?
- Under what circumstances can a program execute?
- ...and what do we expect of its outputs?
- How should information flow through a system?

A way to ensure that software adheres to policy, i.e. *enforcement*

- With **convincing guarantees**, not ad-hoc arguments
- Often, without trusting developers or users

What logic & languages gives us

Precise ways to write down policies

- Types, contracts, functional specifications, customized logics
- Devised for correctness, great for security as well

Rigorous means of enforcement

- Type checking, formal verification for *static* enforcement
- Runtime monitors, sound instrumentation for *dynamic* enforcement

Convincing guarantees: can *prove* that enforcement ensures policy

Formalism & security

Why is being formal such a big deal?

Formal policies make assumptions and provisions explicit:

- **Important**: these define the attacker's capabilities
- For security, formality means *no surprises*!

Formal claims can be proven if true, and refuted if not

- "Is my program secure" should not be a rhetorical question
- ...instead, a math problem
- Without a proof, why should you trust it?

Formal techniques can often be automated

Formalism & security

Why is being formal such a big deal?

Formal policies make assumptions and provisions explicit:

- **Important**: these define the attacker's capabilities
- For security, formality means *no surprises*!

(Useful) Formal guarantees can be proven if true, and refuted if not

- "Is my program secure" is no longer a rhetorical question
- ...instead, a math problem
- If there's no proof, why should you trust it?

Formal techniques can often be automated

- While formal proof can be tedious, automation means less work
- Proof checkers mitigate human error, enable audit

What being formal doesn't give us

Formalism isn't a panacea

Proofs are relative to the formal definitions and assumptions in play

- When these aren't realistic, neither are the guarantees
- See Cormac Herley's "Unfalsifiability of security claims" in *PNAS* for a healthy dose of skepticism on this matter

Creativity, intuition, and good engineering are important for:

- Devising and validating useful definitions
- Identifying the right threat model, assumptions
- Building robust and efficient implementations

Course topics

Some of the topics that we will cover include:

- Policy models: safety, information flow, statistical privacy
- Runtime policy enforcement, reference monitoring
- Security type systems
- Isolation (SFI, CFI, hardware protections)
- Trusted computing, authorization logic
- Web app security & best practices
- Side channel vulnerabilities and defenses
- ...

Primary learning objectives

After taking this course, you should:

1. Be able to identify, formalize, and implement useful security & privacy policies
2. Understand the tradeoffs of different approaches to security & privacy, and know how to reason about which one to use
3. Understand the role of key principles like least privilege, small trusted computing base, and complete mediation in formulating effective defenses
4. Be able to use formal proof and deductive systems to reason about the security of software systems

Logistics

Website: `https://15316-cmu.github.io`

Course staff contact: Piazza
Lecture: Tuesdays & Thursdays, 9:00-10:20 HH B103

Matt Fredrikson

- Location: CIC 2126
- Office Hours: **answer Piazza poll on good times**
- Email: mfredrik@cs

Grading

Breakdown:

- 30% labs
- 30% written homework
- 30% exams (15% each, midterm and final)
- 10% participation

3-4 labs

Written homework most weeks

In-class exams, closed-book, one sheet of handwritten notes

Participation:

- Come to lecture
- Ask questions, give answers
- Contribute to discussion
- Be active and helpful on Piazza

Written homework (30% of grade)

Written homeworks focus on theory and fundamental skills

Grades are based on:

- Correctness of your answer
- How you present your reasoning

Strive for **clarity & conciseness**

- Show each step of your reasoning
- State your assumptions
- Answers without well-explained reasoning don't count!

Labs (30% of grade)

Extend HTTP server to serve answers to data queries

Incrementally add functionality while maintaining security

Grades are based on:

- Whether you implemented correct functionality
- Robustness to relevant attacks

Partial credit depending on:

- How close your impl. is to the functional spec
- How many attacks your security measures prevent

What to do before Thursday

1. Make sure that you are enrolled in the Gradescope and Piazza sections for this course
   - Gradescope entry code: **9E62JE**
   - Piazza signup link: `http://piazza.com/cmu/fall2019/15316`
2. Answer the Piazza poll about office hours time slots
3. Read the syllabus on the webpage carefully
4. Contact me (on Piazza!) if you have any questions