Software Foundations of Security & Privacy
15316 Fall 2020
Lecture 1:
Introduction

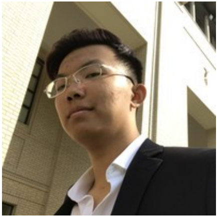
Matt Fredrikson
mfredrik@cs

September 1, 2020

Course Staff

Matt Fredrikson
Instructor

Urvi Agarwal
TA

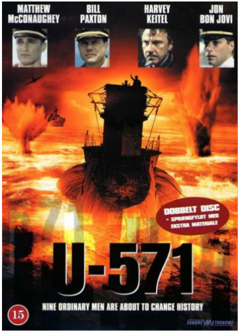


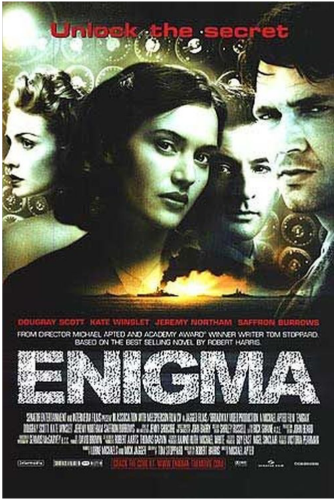
Ryan Chen
TA

Themes and anti-themes

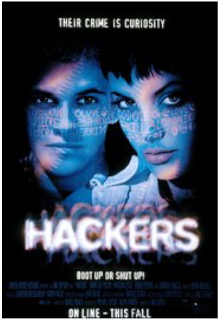
What is this course about?

This is not a course about encryption...

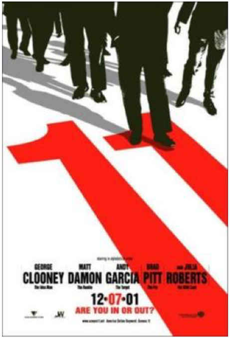


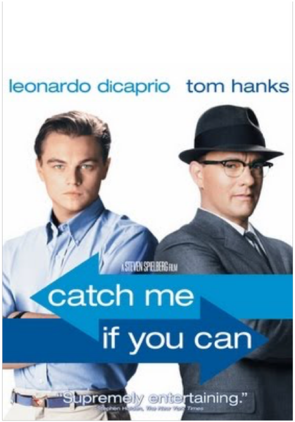


Not a course about hacking...

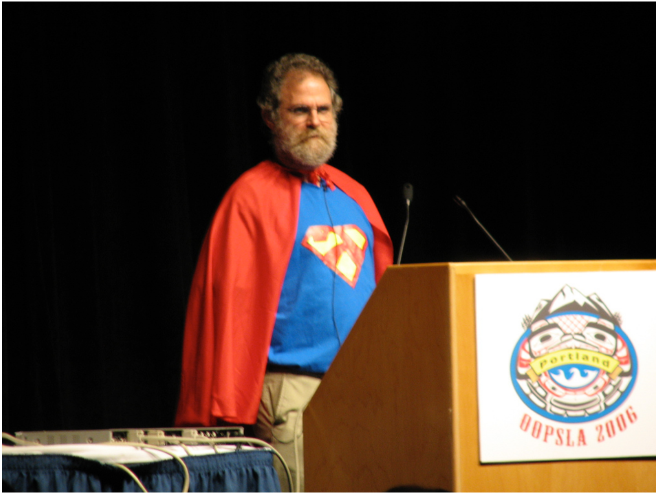


Not a course about social engineering...



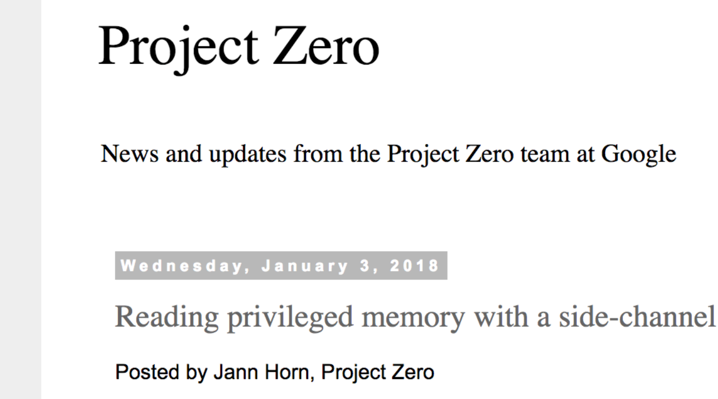


This course is about...



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

Falling sky



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?

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
- ...?

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

```
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

```
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

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 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

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

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

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

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:

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

```
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

```
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

Speculative execution

```
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...

Speculative execution

```
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

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

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

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

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

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 $n \in [0..255]$

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

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

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. Why?

Mitigations

How do we fix it?

Unlike most vulnerabilities, doesn't seem patchable. Why?

- Problem caused by both software + hardware issues

Mitigations

How do we fix it?

Unlike most vulnerabilities, doesn't seem patchable. Why?

- Problem caused by both software + hardware issues
- attacker's input → speculative cache → "normal" cache
- Without hardware changes, no apparent universal fix

Mitigations

How do we fix it?

Unlike most vulnerabilities, doesn't seem patchable. Why?

- Problem caused by both software + hardware issues
- attacker's input → speculative cache → "normal" cache
- Without hardware changes, no apparent universal fix

But there are software-based mitigations

Mitigations

How do we fix it?

Unlike most vulnerabilities, doesn't seem patchable. Why?

- Problem caused by both software + hardware issues
- attacker's input → speculative cache → "normal" cache
- Without hardware changes, no apparent universal fix

But there are software-based mitigations

1. Disable speculative execution *(expensive!)*

Mitigations

How do we fix it?

Unlike most vulnerabilities, doesn't seem patchable. Why?

- Problem caused by both software + hardware issues
- attacker's input → speculative cache → "normal" cache
- Without hardware changes, no apparent universal fix

But there are software-based mitigations

1. Disable speculative execution *(expensive!)*
2. Disable the cache *(way more expensive!)*

Mitigations

How do we fix it?

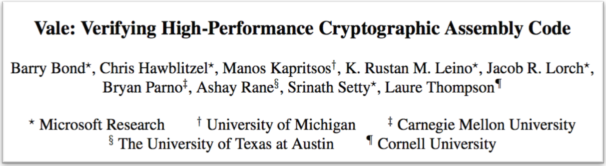
Unlike most vulnerabilities, doesn't seem patchable. Why?

- Problem caused by both software + hardware issues
- attacker's input → speculative cache → "normal" cache
- Without hardware changes, no apparent universal fix

But there are software-based mitigations

1. Disable speculative execution *(expensive!)*
2. Disable the cache *(way more expensive!)*
3. Don't index arrays on untrusted values *(hard?)*

Ongoing research: provable side-channel security



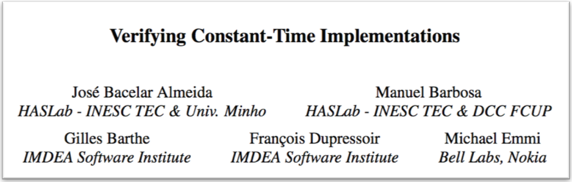
Vale: Verifying High-Performance Cryptographic Assembly Code

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

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 general principles behind vulnerabilities
- Design and critically evaluate potential solutions
- Learn a set of rigorous defense strategies, implement some
- Hopefully, write code that isn't vulnerable to begin

Making software secure: desiderata

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

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*

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

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? Who can we trust?
- 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*

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? Who can we trust?
- 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 **provable guarantees**, not ad-hoc arguments
- Often, without trusting developers or users

What logic & languages gives us

Precise ways to write down policies

What logic & languages gives us

Precise ways to write down policies

- Types, logical specification, specialized languages
- (Often) devised for correctness, perfect for security also

What logic & languages gives us

Precise ways to write down policies

- Types, logical specification, specialized languages
- (Often) devised for correctness, perfect for security also

Rigorous ways to enforce policies

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

What logic & languages gives us

Precise ways to write down policies

- Types, logical specification, specialized languages
- (Often) devised for correctness, perfect for security also

Rigorous ways to enforce policies

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

Rigorous means: we can *prove* that policy is obeyed

Formalism & security

Why is proving things important?

Formalism & security

Why is proving things important?

Proof requires formalism

Formalism & security

Why is proving things important?

Proof requires formalism

Formal policies make assumptions and provisions explicit:

- These define our goals, the attacker's capabilities
- For security, formality means *no surprises*!

Formalism & security

Why is proving things important?

Proof requires formalism

Formal policies make assumptions and provisions explicit:

- These define our goals, the attacker's capabilities
- For security, formality means *no surprises*!

Focus on refutable claims of security

- Use math to exhaust the relevant space of attacks
- Rely on formalism to make it clear what remains unknown
- Ideal: break out of vuln/patch arms race

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)
- Privacy for individuals
- Trusted computing, authorization logic
- Web app security & best practices
- Side channel vulnerabilities and defenses
- ...

Primary learning objectives

After taking this course, you should:

Primary learning objectives

After taking this course, you should:

1. Be able to identify, formalize, and implement useful security & privacy policies

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

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

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:
`15-316-course-staff@lists.andrew.cmu.edu`
Lecture: Tuesdays & Thursdays, 8:00-9:20 on Zoom

Matt Fredrikson

- Location: CIC 2126
- Office Hours: **answer Piazza poll on good times**

Grading

Breakdown:

- 35% labs
- 35% written homework
- 30% exams (15% each, midterm and final)

2-3 labs

Written homework most weeks

Exams open-book, some additional time for scanning/typesetting

Participation:

- Come to lecture if you can
- Contribute to discussion
- Answer questions on Piazza
- Ask questions **early**!

Written homework (35% 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 (35% 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, Canvas, Piazza sections for this course
    - Canvas: `https://canvas.cmu.edu/courses/19932`
    - Gradescope entry code: **9GNPXE**
    - Piazza signup link: `https://piazza.com/class/keisqg3c2w43jr`
2. Answer the Piazza poll about office hours time slots
3. Read the syllabus on the webpage carefully
4. Get started on homework (if you can)