What will you learn, and how?

Course overview

- This course is primarily aimed at people interested in **secure software development**, who will
  - **Design** software systems that should be secure
  - **Write** code that should be secure
  - **Review** code that should be secure
  - **Test** code that should be secure
- On the design side, we will connect to other classes in the ***Coursera cybersecurity series***
  - **Usable security, cryptography, hardware security**
- Much of our **focus** will be on **the software,** and how to develop it to be **secure**

Expected background

- **Roughly:** knowledge of a **junior-level undergraduate majoring in computer science**
- **Knowledge** with the following (we will not review)
  - **Programming** in general (two semesters' equivalent of instruction, e.g., in Java)
  - **C programming language** details (one semester course), including memory allocation (both stack and heap), pointers, arrays, loops, function calls, etc.
- **Familiarity** with the following (we will do some review):
  - **Unix/Linux** including the command-line **shell** and **gdb**
  - The **web** (HTML, HTTP, TCP, network communications)
  - **Intel x86 assembly** language and architecture

Learning Software Security

- Our goal is learn how to make more **secure software**
  - **Better design**
  - **Better implementation**
  - **Better assurance**

How should we go about this?

Black Hat, White Hat

Black hat

- What are the **security-relevant defects** that constitute **vulnerabilities**?
- How are they **exploited**?

White hat

- How do we **prevent security-relevant defects** (before deploying)?
- How do we make vulnerabilities we don’t manage to avoid **harder to exploit**?

During the course we will wear both hats

Low-level Vulnerabilities

- Programs written in **C and C++** are susceptible a variety of dangerous **vulnerabilities**
  - **Buffer overflows**
    - On the stack
    - On the heap
    - Due to integer overflow
    - Over-writing and over-reading
  - **Format string mismatches**
  - **Dangling pointer dereferences**

Attacks

- *Stack smashing*
- *Format string attack*
- *Stale memory access*
- *Return-oriented Programming (ROP)*

- All **violations** of **memory safety**
  - Accesses to memory via pointers that don't *own* that memory

Ensuring Memory Safety

- The easiest way to avoid these vulnerabilities is to **use a memory-safe programming language**
    - Better still: a ***type*-safe** language
- For C/C++, use **automated defenses**
    - *Stack canaries*
    - *Non-executable data (aka W+X or DEP)*
    - *Address space layout randomization (ASLR)*
    - *Memory-safety enforcement (e.g., SoftBound)*
    - *Control-flow Integrity (CFI)*
- and **safe programming patterns and libraries**
    - *Key idea*: **validate untrusted input**

Securing the WWW

- Cybersecurity battles rage on the **world wide web**
- There are new **vulnerabilities** and **attacks**
  - *SQL injection*
  - *Cross-site scripting (XSS)*
  - *Cross-site request forgery (CSRF)*
  - *Session hijacking*

- The **defenses** have a **similar theme**
  - Careful who/what you trust: **Validate input**
  - Reduce the possible damage, make exploitation harder

Secure Software Development

- Consider security *throughout software lifecycle*
  - **Requirements**
  - **Design**
  - **Implementation**
  - **Testing/assurance**

- Corresponding activities
  - **Define** *security requirements, abuse cases,*
  - **Perform** *architectural risk analysis (threat modeling)* and *security-conscious design*
  - **Conduct** *code reviews, risk-based security testing,* and *penetration testing*

Requirements and Design

- Identify **sensitive data** and resources and define **security requirements** for them, like *confidentiality, integrity,* and *availability*
  - Consider expected **threats** and **abuse cases** that could violate these requirements
- Apply **principles for secure software design**
  - To **prevent**, **mitigate**, and **detect** possible attacks
  - Main categories: **Favor Simplicity**, **Trust with Reluctance**, and **Defend in Depth**.
- *Exemplar*: **Very Secure FTP daemon**

Rules and Tools

- Apply **coding rules** to implement secure design
  - With similar goals of *preventing*, *mitigating*, or *detecting* possible attacks

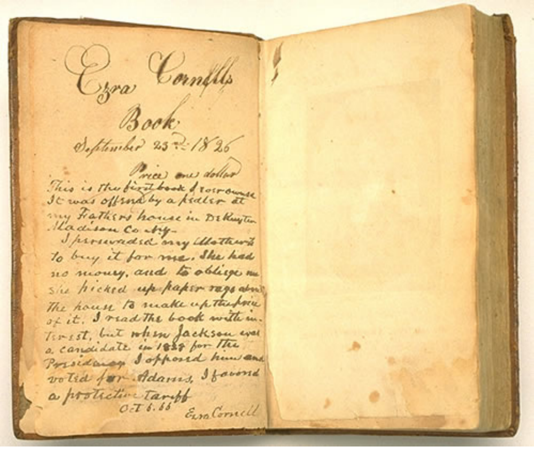
- Apply **automated code review techniques** to find potential vulnerabilities in components
  - **Static analysis**, and **symbolic execution** (which underlies whitebox fuzz testing)

- Apply **penetration testing** to find potential flaws in the real system, in a deployment environment
  - **Fuzz testing**, perhaps employing **attack patterns**

Content

6 units

- **Memory attacks**
- **Memory defenses**
- **Web security**
- **Secure design/development**
- **Automated code review**
- **Penetration testing**



Ezra Cornells
Book
September 25th 1826
Price one dollar
This is the first book I ever owned
It was offered by a pedler at
my Fathers house in DeRuyter
Madison Co. N.Y.
I persuaded my Mother
to buy it for me. She had
no money, and to oblige me
she picked up paper rags about
the house to make up the price
of it. I read the book with interest, but when Jackson was
a candidate in 1828 for the
Presidency I opposed him and
voted for Adams, I favored
a protective tariff
Oct 6.66 Ezra Cornell

~80 minutes of **video** per week, for six weeks

- Along with **supplemental readings**

Hear from the experts

Andy Chou

Static analysis expert

CTO, Coverity

Gary McGraw

Software security expert

Noted author,
CTO, Cigital, Inc.

Hear from the experts

Eric Eames
Penetration testing expert
Principal Security Consultant
FusionX

Patrice Godefroid
White-box *fuzzing* expert
Principal Researcher,
Microsoft Research

Assessment

- **Projects**
  - Learn core concepts about vulnerabilities, and building software correctly, by doing

- **Quizzes**
  - Check your knowledge of concepts and details
  - One quiz per week

Let’s begin …