POLICIES AND ENFORCEMENT

RECAP: THREAT MODELS

- Intellectual Curiosity
- Bragging Rights
- Financial Gain
- Political Motivation
POLICIES AND ENFORCEMENT

RECAP: COMPUTER SECURITY PRINCIPLES

- 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
POLICIES AND ENFORCEMENT

RECAP: REASONABLE ASSUMPTIONS

▸ Always Assume Attackers:
  ▸ Can interact with your systems without particular notice
  ▸ Know general information about your systems
  ▸ Can obtain access to an exact copy of your system to measure and/or determine how it works
  ▸ Will make enthusiastic use of automation
  ▸ Can initiate sophisticated coordinated activity across a plethora of geographically and architecturally disparate systems
  ▸ Can bring massive resources to bear if required
  ▸ Can obtain elevated privileges if it helps them
  ▸ Are at least as smart as you are, but probably smarter
POLICIES AND ENFORCEMENT

TODAY

- Expressing and enforcing security policies
- Reasoning about software security
POLICIES AND ENFORCEMENT

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.*
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
POLICIES AND ENFORCEMENT

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
POLICIES AND ENFORCEMENT

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
CONFIDENTIALITY

- Protecting secrets as well as *inferences* about them, or even their existence
- For example

```plaintext
public := 0;
if secret == 1 then
    public := 1
```

- The contents of the variable `public` *leaks* the value of the variable `secret` (without requiring direct access)
POLICIES AND ENFORCEMENT

INTEGRITY

- “Bad things should not happen”
  - Correctness criteria
  - Absences of crashes or unexpected exits or errors
- Also for constraining how data may be modified:
  - Only a particular user or program can modify
  - Any modification must satisfy X, Y, Z constraints
  - Before running code, must pass validation
POLICIES AND ENFORCEMENT

AVAILABILITY

▶ “Good things should happen”

▶ A service that is required
  ▶ Provide access to cloud backup files
  ▶ Continually monitor for evidence of fire
  ▶ Process request in the order they are received

▶ Important for critical infrastructure and services that may be subjected to denial-of-service attacks
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.

- Ideally we want enforcement mechanisms that support a broad range of policies.
POLICIES AND ENFORCEMENT

ISOLATION

- Restricting communications with the outside world
POLICIES AND ENFORCEMENT

ISOLATION

- Example: Virtual machines
  - Execute system as if running on an isolated computer
  - Can emulate real hardware or something higher level
POLICIES AND ENFORCEMENT

ISOLATION

Example: Sandboxes

- Hides or duplicates resources, mediates access to host system
POLICIES AND ENFORCEMENT

ISOLATION

- Example: Processes

- Operating system mediates access to shared resources
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
- Sometimes, you get **neither**:
  - The **same origin policy** provides little security, but limits the kind of web applications one can easily build
POLICIES AND ENFORCEMENT

THE SANDBOXING CYCLE (XKCD)

1. "I wish these parts could communicate more easily!"
   ![Diagram showing parts communicating]

2. "Oh, this new technology makes it easy to create arbitrary connections, integrating everything!"
   ![Diagram showing increased connection]

3. "Oh, oh, this new technology makes it easy to enclose arbitrary things in secure sandboxes!"
   ![Diagram showing sandboxes]

4. "Uh-oh, there are so many connections it's creating bugs and security holes!"
   ![Diagram showing issues with connections]
POLICIES AND ENFORCEMENT

MONITORING

- Monitor interfaces to system and halt before violations
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
POLICIES AND ENFORCEMENT

SECURITY POLICIES

- Monitoring is useful for enforcing policies expressed in terms of principals and their privileges
  - Each principal is assigned a set of privileges
  - Each operation requires a set of privileges to execute
  - If a principal requests an operation without the necessary privileges, execution is halted

- *Access control lists* and *capabilities* are examples
POLICIES AND ENFORCEMENT

RECOVERY

- Reverse the damaging effects of attacks
POLICIES AND ENFORCEMENT

RECOVERY

▸ Most effective at reversing corruption or malicious modifications made by an attacker.

▸ Examples:
  ▸ Running browser in a VM to avoid malware
  ▸ Reverting to known-good backups after a compromise
  ▸ Transactional processing of concurrent sequences of operations (revert on conflicts)
POLICIES AND ENFORCEMENT

RECOVERY

- Some attacks cannot be reversed (unless you are a Time Lord):
  - Secrets cannot be un-leaked
  - Missiles cannot be un-fired
SOFTWARE SECURITY

REASONING ABOUT SOFTWARE SAFETY

- 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
- **Modularity** provides boundaries for our reasoning:
  - **Preconditions**: must hold for function to operate correctly
  - **Postconditions**: should hold after function completes
SOFTWARE SECURITY

REASONING ABOUT SOFTWARE SAFETY

- We describe a **contract** for using the function
- Notions also apply to individual statements
  - What must hold for correctness?
  - What holds after execution?
  - Statement 1’s postcondition should imply Statement2’s precondition
- **Invariants**: conditions that always hold at a given point in a function (this particularly matters for loops)
SOFTWARE SECURITY

REASONING ABOUT MEMORY SAFETY

- Prevent access to undefined memory
  - “Undefined” with respect to the semantics of the programming language
- Prevent unauthorized access to defined memory
  - “Unauthorized” with respect to system policy
- Where “access” = read / write / execute
// requires: p != NULL
// requires: p is a valid pointer
int deref(int *p) {
    return *p;
}

**Precondition:** That which needs to hold for the function to operate safely and correctly on a COMPUTER.

Needs to be expressed in a way that a HUMAN writing code to call the function can easily evaluate.
SOFTWARE SECURITY

```c
// ensures: return value != NULL
// ensures: return value is a valid pointer
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p){
        perror("malloc");
        exit(1);
    }
    return p;
}
```

**Postcondition:** What the function promises will hold upon its return when executed by a COMPUTER.

Also expressed in a way that a HUMAN using the call in their code can make sense of and act on.
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}

**Precondition(s)?**
```c
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}
```

General correctness proof strategy for memory safety:
1) Identify each point of memory access
2) Write down preconditions
3) Promote preconditions to function comment
4) Identify invariants
```c
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}
```

General correctness proof strategy for memory safety:
1) **Identify each point of memory access**
2) Write down preconditions
3) Promote preconditions to function comment
4) Identify invariants
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        // ??
        total += a[i];
    }
    return total;
}

General correctness proof strategy for memory safety:
1) Identify each point of memory access
2) Write down preconditions
3) Promote preconditions to function comment
4) Identify invariants
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        // requires: a != NULL
        // requires: i >= 0
        // requires: n <= no. of elements in a
        total += a[i];
    }
    return total;
}

General correctness proof strategy for memory safety:
1) Identify each point of memory access
2) Write down preconditions
3) Promote preconditions to function comment
4) Identify invariants
// requires: a != NULL

int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        // requires: i >= 0
        // requires: n <= no. of elements in a
        total += a[i];
    }
    return total;
}

General correctness proof strategy for memory safety:
1) Identify each point of memory access
2) Write down preconditions
3) **Promote preconditions to function comment**
4) Identify invariants
// requires: a != NULL
// requires: n <= no. of elements in a
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        // requires: i >= 0
        total += a[i];
    }
    return total;
}

General correctness proof strategy for memory safety:
1) Identify each point of memory access
2) Write down preconditions
3) Promote preconditions to function comment
4) Identify invariants
// requires: a != NULL
// requires: n <= no. of elements in a
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        // invariant: i >= 0 (from code: i = 0)
        total += a[i];
    }
    return total;
}

General correctness proof strategy for memory safety:
1) Identify each point of memory access
2) Write down preconditions
3) Promote preconditions to function comment
4) Identify invariants
// requires: a != NULL
// requires: n <= no. of elements in a
int sum(int a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}

And we’re done!

Or are we?
// requires: n <= no. of elements in a
text sum(int a[], size_t n) {
    assert(a != NULL);
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += a[i];
    }
    return total;
}
// requires: a != NULL
// requires: n <= no. of elements in a
// requires: ???
int sumderef(int *a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += *a[i];
    }
    return total;
}
// requires: a != NULL
// requires: n <= no. of elements in a
// requires: for all j in 0..n-1, a[j] != NULL
int sumderef(int *a[], size_t n) {
    int total = 0;
    for(size_t i = 0; i < n; i++) {
        total += *a[i];
    }
    return total;
}
// requires: n <= no. of elements in a

int sumderef(int *a[], size_t n) {
assert(a != NULL);
int total = 0;
for(size_t i = 0; i < n; i++) {
    assert(a[i] != NULL);
    total += *a[i];
}
return total;
}
char *tbl[N];  // N > 0, type int

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);
}
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;
}

What is the correct postcondition for hash()?

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

Discuss with a partner...
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// 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);
}
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
int hash(char *s) {
    int h = 17; // 0 <= h
    while (*s) { // 0 <= h
        h = 257*h + (*s++) + 3; // 0 <= h
    }
    return h % N; // 0 <= return value < N
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
// what happens if we do this?
char c = -1;
int h = hash(&c);
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
int hash(char *s) {
    int h = 17; // 0 <= h
    while (*s) { // 0 <= h
        h = 257*h + (*s++) + 3; // 0 <= h
    }
    return h % N; // 0 <= return value < N
}

What is the correct postcondition for hash()?

a) 0 <= return value
b) return value < N
c) a) and b)
d) none of the above
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
unsigned int hash(char *s) {
    unsigned int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    unsigned int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
Discuss for two minutes with your classmates:

- What mechanisms could we use to build more secure software?
SOFTWARE SECURITY

WHY DOES SOFTWARE HAVE VULNERABILITIES?

- Programmers are humans, mostly
  - Use tools!
  - Automate!
- Lack of security awareness
  - Education!
- Low-level languages (ahem, C) aren’t designed for security
  - Higher-level languages (e.g., Java, OCaml, Haskell, Python) give more guarantees
SOFTWARE SECURITY

CMU SEI CAPABILITY MATURITY MODEL

- **Initial**
  - Undocumented and in a state of dynamic change

- **Repeatable**
  - Some processes are repeatable, possibly with consistent results

- **Defined**
  - Defined and documented standard processes established and subject to some degree of improvement over time

- **Managed / Capable**
  - Using process metrics, effective achievement of the process objectives can be evidenced across a range of operational conditions

- **Optimizing**
  - Continually improving process performance through both incremental and innovative technological changes/improvements

In the 90s SEI determined that ~70% of organizations were stuck here
SOFTWARE SECURITY

TESTING FOR SECURITY

▸ How to test for the **absence** of something?
  ▸ Security is (often) a **negative** property
  ▸ “Normal” inputs rarely stress security-vulnerable code

▸ Testing via **randomized input generation** (aka “fuzzing”) helps
  ▸ Few false positives: anything you can generate, so can an attacker
  ▸ Huge search space, hard to get past first layers of code

▸ When are you done? Code coverage is a proxy, but not perfect

The birth of fuzzing: a dark and stormy night
SOFTWARE SECURITY

TESTING FOR SECURITY

- Vulnerability scanning: probe systems for known flaws
  - Automate!
- Penetration testing ("pen-testing")
  - Pay an expert to try to break into your system
  - Assess how they did it and recommend fixes
SOFTWARE SECURITY

KEEPING SOFTWARE UP TO DATE

▸ What is so hard about patching?
  ▸ Can require restarting systems
  ▸ Can break crucial functionality
  ▸ Management burden
    ▸ When/where to patch
    ▸ Tension between risk of exploitation and preventing regressions
  ▸ They keep coming!
SOFTWARE SECURITY

TECHNIQUES FOR BUILDING SECURE SYSTEMS

- **Goals:** try to prevent, otherwise mitigate, at least detect

- **Run-time checks / monitoring:**
  - Bounds checking, stack inspection
  - What happens on a failed check? Performance overhead?

- **Address randomization (ASLR)**
  - Make it hard for attacker to predict memory layout
  - Not perfect: vulnerabilities could reveal the layout, or exploits could reduce entropy (e.g., “heap spraying”)

- **Non-executable stack and heap**
  - Some legacy code issues, plus some programs need to generate code (JITs)
  - Not perfect: return-oriented programming (ROP)
SOFTWARE SECURITY

TECHNIQUES FOR BUILDING SECURE SYSTEMS

- **Coding standards**
  - Defensive programming: extra sanity checks to guard against broken requirements or corrupted memory
  - Safe libraries: `strlcpy` vs `strcpy`, `snprintf` vs `sprintf`
  - **Code reviews** to enforce standard + find other problems

- **Bug-finding tools**
  - Static analysis of source code (most common)
  - Dynamic analysis of runtime code
  - Good, but often many false positives. (Avoiding FPs == coding standard?)
SOFTWARE SECURITY

TECHNIQUES FOR BUILDING SECURE SYSTEMS

▸ Use a safe language
  ▸ Safe ~ memory safety, strong typing, garbage collection
  ▸ Strong typing = programs can’t “go wrong” at runtime
  ▸ Static type checking makes runtime errors into compile-time errors

▸ Constrain user inputs
  ▸ Prevent untrusted inputs from flowing to security sensitive operations: e.g., SQL injection, XSS, etc
  ▸ “Sanitize” inputs to prevent code injection

▸ Contain damage
  ▸ Isolate system components in VMs or chroot jails, separate and minimize privileges
SOFTWARE SECURITY

NEXT TIME

- More on Software Security