CISC 3325 - Information Security

Programs and Programming

Working Towards Secure Systems

Common Software Vulnerabilities

- Memory safety vulnerabilities
- Input validation vulnerabilities
- Race conditions
- Time-of-Check to Time-of-Use (TOCTTOU) vulnerability

Memory safety vulnerabilities

- Software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers
- Java is said to be memory-safe
  - its runtime error detection checks array bounds and pointer dereferences (*p).
- In contrast, C and C++ support arbitrary pointers with no provision for bounds checking
  - thus are termed **memory-unsafe**

Input validation vulnerabilities

- Program requires certain assumptions on inputs to run properly
- When program does not validate input correctly, it may get exploited
- Buffer overflow and SQL injection are just a few of the **attacks** that can result from improper data **validation**
- Example:
  - Bank money transfer:
    - Check that amount to be transferred is non-negative and no larger than payer's current balance

Overflow Countermeasures

- Staying within bounds
  - Check lengths before writing
  - Confirm that array subscripts are within limits
  - Double-check boundary condition code for off-by-one errors
  - Limit input to the number of acceptable characters
  - Limit programs' privileges to reduce potential harm
- Many languages have overflow protections
- Code analyzers can identify many overflow vulnerabilities
- Canary values in stack to signal modification

Incomplete Mediation

- Mediation: Verifying that the subject is authorized to perform the operation on an object
- Preventing incomplete mediation:
  - Validate all input
  - Limit users’ access to sensitive data and functions
  - Complete mediation using a reference monitor

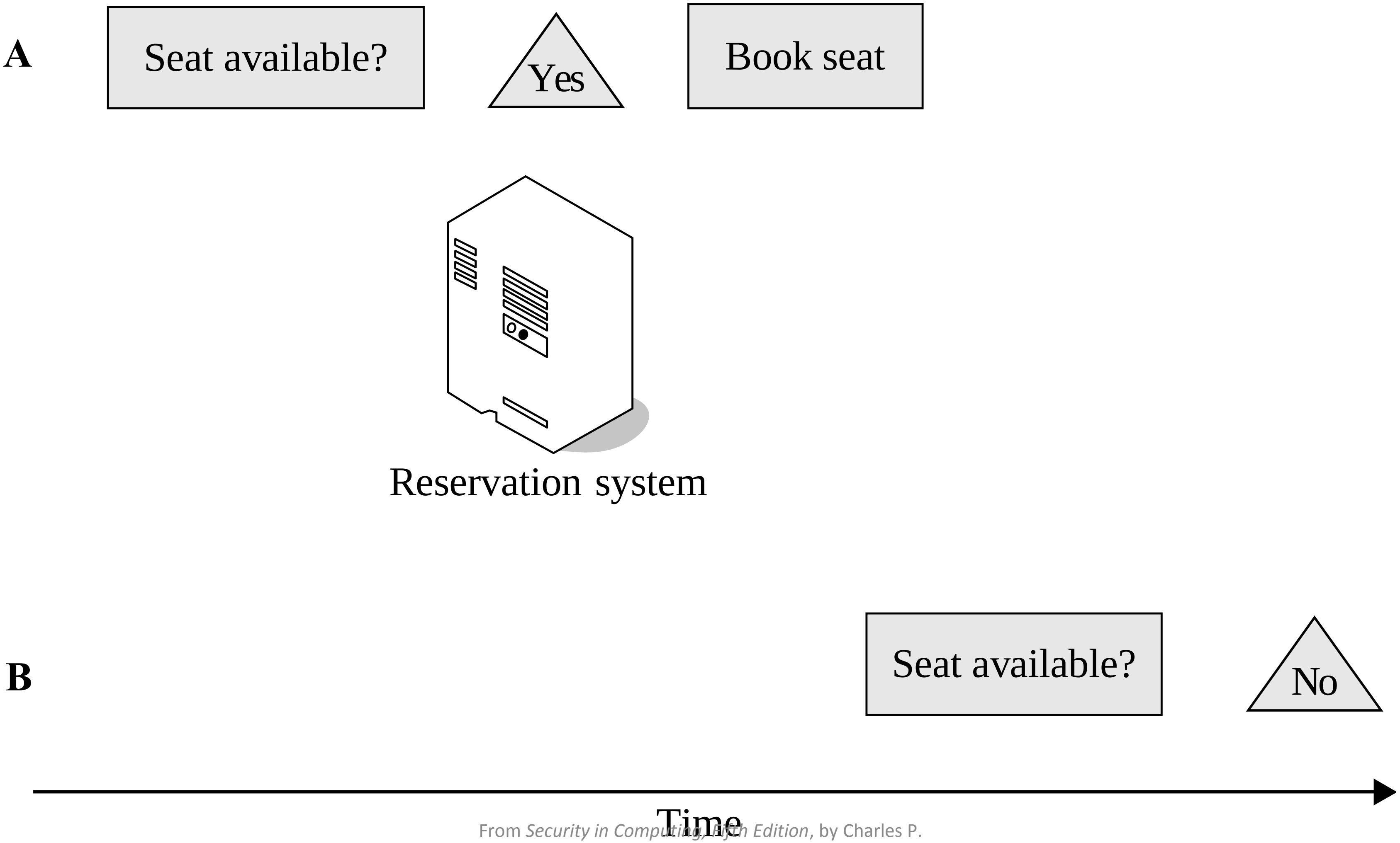
Race Condition

- Output is dependent on the sequence or timing of other uncontrollable events
- Becomes a vulnerability when events happen in a different order
  - than the programmer intended

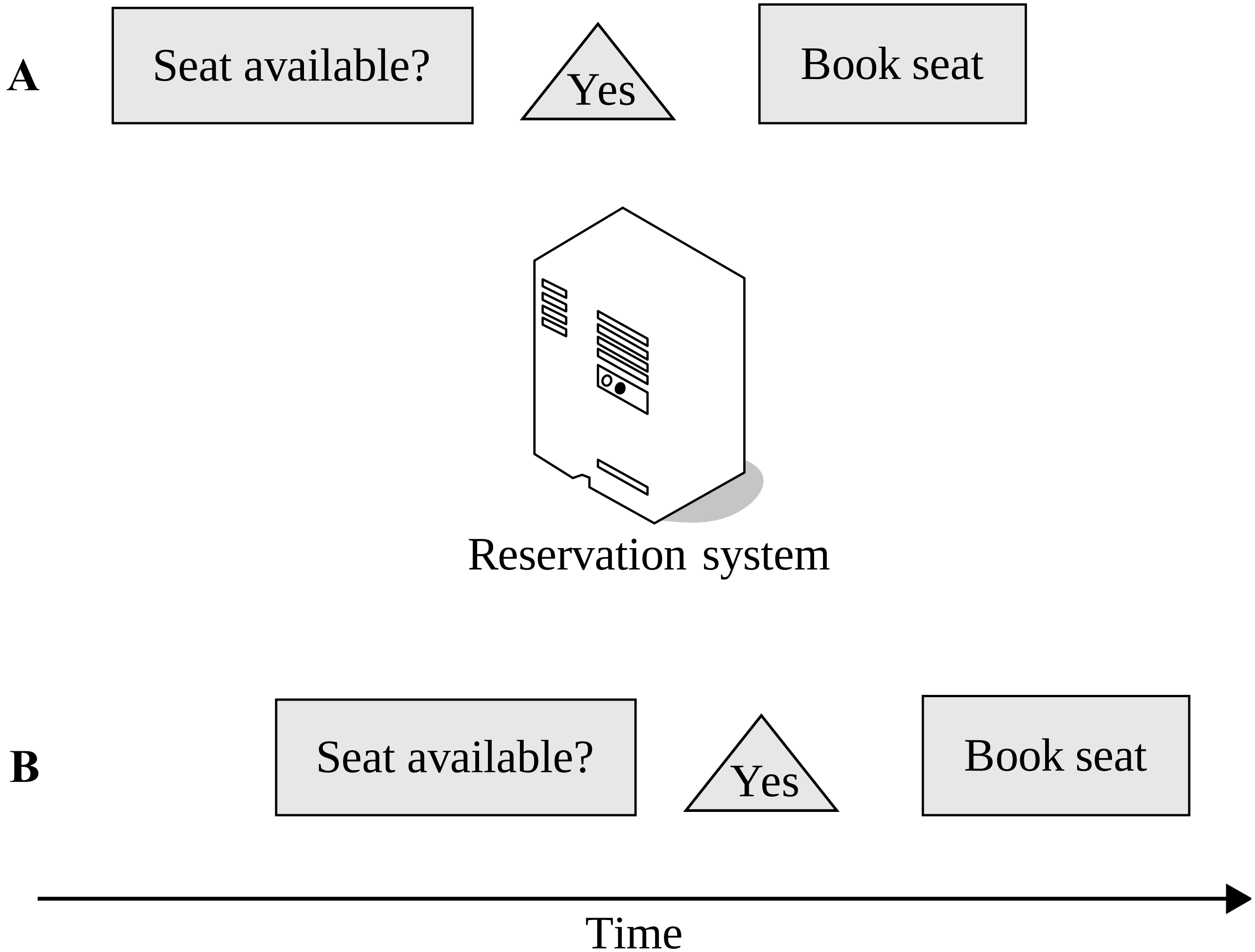
Time-of-Check to Time-of-Use (TOCTTOU) vulnerability

- Software bugs caused by changes in a system
  - Between the checking of a condition (such as a security credential) and the use of the results of that check
- An example of a race condition

Race Conditions



Race Conditions



Other Programming Oversights

- Undocumented access points (backdoors)
- Off-by-one errors
  - Occurs when an iterative loop iterates one time too many or too few
  - May occur when:
    - Using “<=" where “<" should have been used in a comparison
    - A sequence starts at zero rather than one (as with array indices in many languages)
- Integer overflows
  - Attempt to create a numeric value outside of the range that can be represented with a given number of digits
    - either larger than the maximum or lower than the minimum representable value
- Unterminated null-terminated string
- Parameter length, type, or number errors
- Unsafe utility libraries

Working Towards Secure Systems

- Along with securing individual components, we need to keep them up to date …
  - New software versions are constantly released
    - Keeping up with other competitive applications, OS changes, add new features.
- What's hard about **patching**?
  - Can break crucial functionality inadvertently
  - Management burden:
    - It never stops (the "*patch treadmill*") …

Questions?

Countermeasures for Developers

- Modular code: Each code module should be
  - Single-purpose
  - Small
  - Simple
  - Independent
- Encapsulation
  - Restrict direct access to some of the object's components
    - Using built-in language mechanism
- Information hiding
  - Segregating the program design decisions most likely to change
    - => protecting other parts of the program from extensive modification if the design decision is changed

Countermeasures for Developers

- Mutual Suspicion
  - Mutually suspicious programs operate as if other routines in the system were malicious or incorrect.
  - A calling program cannot trust its called subprocedures to be correct
    - a called subprocedure cannot trust its calling program to be correct.
  - Each protects its interface data so the other has only limited access
- Confinement
  - Isolate program data, functionality
    - For example, server should send only certain data to client

Code Testing

- Unit testing:
  - individual units/ components of a software are tested.
  - Validate that each **unit** of the software performs as designed
    - A **unit** is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.
- Integration testing:
  - individual software modules are combined and **tested** as a group. It occurs after unit **testing** and before validation **testing**
- Functional testing**:**
  - A way of checking software to ensure that it has all the required functionality that's specified within its **functional** requirements

Code Testing

- Performance testing:
  - A testing practice performed to determine how a system performs in terms of responsiveness and stability under a particular workload
- Acceptance testing:
  - Evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery
- Installation testing:
  - Most software systems have installation procedures that are needed before they can be used for their main purpose.
  - Test these procedures to achieve a usable installed software system
- Regression testing:
  - Testing changes to computer programs to make sure that the older programming still works with the new changes

Working Towards Secure Systems

- Additional approaches:
  - Use a vulnerability scanner
  - Penetration Testing
  - Design by contract approach

Vulnerability scanner

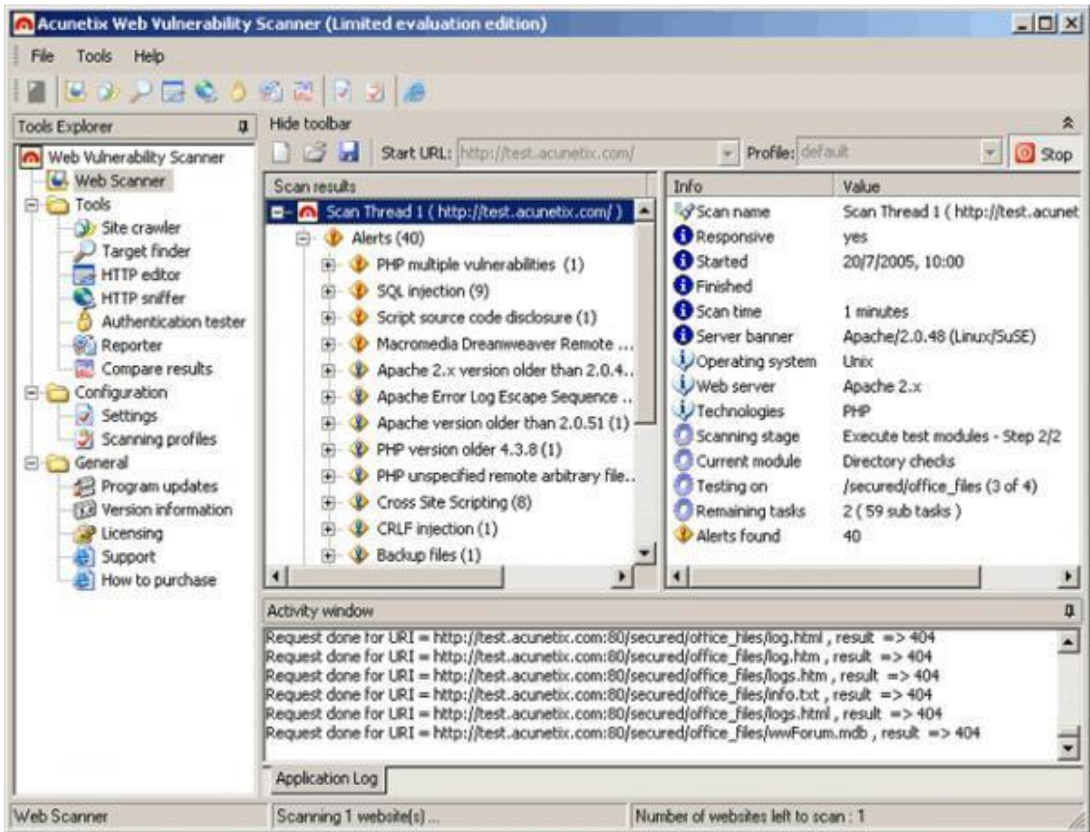
- A computer program designed to assess computers, computer systems, networks or applications for known weaknesses
  - probe your systems/networks for known flaws
- Typically used to detect vulnerabilities in software that runs on a network component
  - firewall, router, web server, application server, etc.

https://www.businesscomputingworld.co.uk/6-reasons-every-business-needs-a-network-vulnerability-scanner/

Vulnerability scanner

- Two types of scans:
  - Authenticated scans: scanner may access low-level data, provide detailed info
    - about OS, installed software, missing security patches, etc.
  - Unauthenticated scans: unable to provide detailed information, may result in high false positives
    - used to determine the security posture of externally accessible assets
      - By attackers or security analysts

Vulnerability scanner



http://acunetix-web-vulnerability-scanner.networkice.com/

Penetration testing (“*pen-testing*”)

- Authorized simulated attack on a computer system
- **Hire** someone to break into your systems …
  - Act as if they were in fact an adversary trying to compromise the integrity of their target organization
  - Attacker should remain stealthy and undetected
    - while executing targeted attack patterns
      - observed in real-world corporate breaches
  - Provide detailed documentation about the attack and the vulnerabilities exploited!

Penetration testing ("*pen-testing*")

- Skilled Penetration testing engineers may be hard to distinguish from real-world hackers
  - Other than first obtaining written permission before engaging their targets.
- Goal is to detect both vulnerabilities and strengths
  - enabling a full risk assessment

Design By Contract Approach

- How can we *verify* that our code executes in a safe (and correct, ideally) fashion?
  - Build up confidence on a function-by-function module-by-module basis
    - Modularity: the extent to which a **software**/Web application may be divided into smaller module
  - By using the **Design By Contract** approach

Design By Contract Approach

- Modularity provides boundaries for our verification:
  - Preconditions: what must hold for function to operate correctly
  - Postconditions: what holds after function completes
  - Invariants: a condition that is true BEFORE and AFTER running the code
- Notions also apply to individual statements (what must hold for correctness; what holds after execution)
  - Statement #1’s postcondition should logically imply statement #2’s precondition

Software Examples

```
/* requires: p != NULL
                    (and p a valid pointer) */
int deref(int *p) {
return *p;
}
```

- What is the precondition here?
  - What needs to hold for function to operate correctly?

Software Examples

```
/* requires: p != NULL
                    (and p a valid pointer) */
int deref(int *p) {
return *p;
}
```

- What is the precondition here?
  - What needs to hold for function to operate correctly?
  - P needs to be a valid pointer
    - Otherwise, return call fails

Software Examples (cont.)

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

- Postcondition?
  - what does the function promise will hold upon its return

Software Examples (cont.)

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

- Postcondition?
  - what does the function promise will hold upon its return
  - Function returns a valid pointer
    - With n bits allocated

Software Examples (cont.)

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

- Precondition?

Software Examples (cont.)

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

- Precondition?
  - a is a valid pointer to a buffer of size n or larger

Increase Memory Safety

- General correctness proof strategy for memory safety:
  - Identify each point of memory access
  - Write down precondition it requires
  - Propagate requirement up to beginning of function
    - Document accordingly
  - Write down post-conditions
    - Verify that function fulfills them

Software Examples (cont.)

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

- Precondition?
  - Identify each point of memory access

Software Examples (cont.)

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

- Precondition?
    - Identify each point of memory access
    - Write down precondition it requires?

Software Examples (cont.)

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

- Precondition?
  - Identify each point of memory access
  - Write down precondition it requires?

Software Examples (cont.)

```
/* ?? */
int sum(int a[], size_t n) {
        int total = 0;
        for (size_t i=0; i<n; i++)
        /* requires: a != NULL && 0 <= i && i < size(a) */
                total += a[i];
        return total;
}
```

- Precondition?
  - Identify each point of memory access
  - Write down precondition it requires?
  - Propagate requirement up to beginning of function?

Software Examples (cont.)

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
	int total = 0;
	for (size_t i=0; i<n; i++)
	/* requires: a != NULL && 0 <= i && i < size(a) */
		total += a[i];
	return total;
}
```

- Precondition?
  - Identify each point of memory access
  - Write down precondition it requires?
  - Propagate requirement up to beginning of function?
    - Is that sufficient? How do we combine both requirements?

Software Examples (cont.)

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
        int total = 0;
        for (size_t i=0; i<n; i++)
        /* requires: a != NULL && 0 <= i && i < size(a) */
                total += a[i];
        return total;
}
```

- Precondition?
  - Identify each point of memory access
  - Write down precondition it requires?
  - Propagate requirement up to beginning of function?
    - Is that sufficient? How do we combine both requirements?
    - At this point the proposed invariant will always hold
      - Invariants: conditions that always hold at a given point in a function

Increase Software Security

- Induction:
  - Another methods of verifying correctness.
  - In case of a more complicated loop
  - Consists of 2 steps:
    - Step 0: verify conditions upon entering the loop
    - Step 1 - Induction: show that the *postcondition* of last statement of loop plus loop test condition implies invariant

Increase Software Security

- Perform both verification and validation
  - Validation: checking whether the specification captures the customer's needs
  - Verification: checking that the software meets the specifications

Increase Software Security

- Use formal verification
  - Prove or disprove the correctness of the algorithm
    - Using analysis and mathematical tools
    - Asser that the algorithm is correct with respect to the software specifications

Bad Practices

- Penetrate-and-patch
  - Fixing problems only after the product has been publicly (and often spectacularly) broken by someone
  - Why is it bad?
    - security should not be an add-on feature
    - problem that is being actively exploited by attackers
- Security by obscurity

Security through Obscurity

http://www.treachery.net/articles_papers/tutorials/why_security_through_obscurity_isnt/index.html

Security through Obscurity

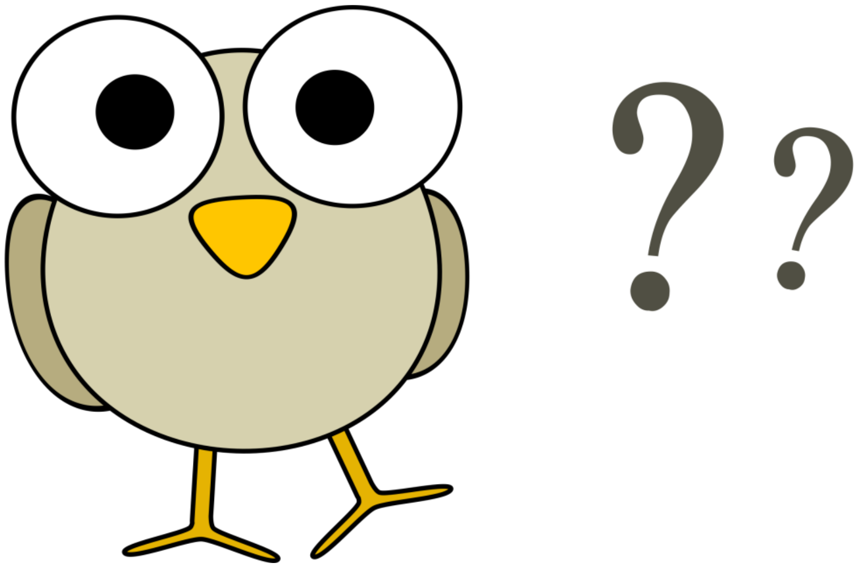
- Reliance on the secrecy of the design or implementation
  - as the main system security method
    - or component of a system
- System may have security vulnerabilities
  - System designers believe that if the flaws are not known, it prevents a successful attack
- Rejected by security experts!
  - obscurity should never be the only security mechanism!
  - has been historically used without success by several organizations

http://ithare.com/advocating-obscurity-pockets-as-a-complement-to-security-part-ii-deployment-scenarios-more-crypto-primitives-and-obscurity-pocket-as-security/

Summary

- Buffer overflow attacks can take advantage of the fact that code and data are stored in the same memory in order to maliciously modify executing programs
- Programs can have a number of other types of vulnerabilities, including off-by-one errors, incomplete mediation, and race conditions
- Developers can use a variety of techniques for writing and testing code for security

- Questions?



Computer Security Quiz

What is penetration testing?

- A. A procedure for testing libraries or other program components for vulnerabilities
- B. Whole-system testing for security flaws and bugs
- C. A security-minded form of unit testing that applies early in the development process
- D. All of the above

What is penetration testing?

- A. A procedure for testing libraries or other program components for vulnerabilities
- B. Whole-system testing for security flaws and bugs
- C. A security-minded form of unit testing that applies early in the development process
- D. All of the above

Which of the following are benefits of penetration testing?

- A. Results are often reproducible
- B. Full evidence of security: a clean test means a secure system
- C. Compositionality of security properties means tested components are secure even if others change
- D. They specifically consider adversarial thinking, which is not usually necessary for normal tests

Which of the following are benefits of penetration testing?

- A. Results are often reproducible
- B. Full evidence of security: a clean test means a secure system
- C. Compositionality of security properties means tested components are secure even if others change
- D. They specifically consider adversarial thinking, which is not usually necessary for normal tests

- Questions?