CS 194-1 (CS 161) Computer Security

Midterm 2 Review Part 1

Todd Kosloff

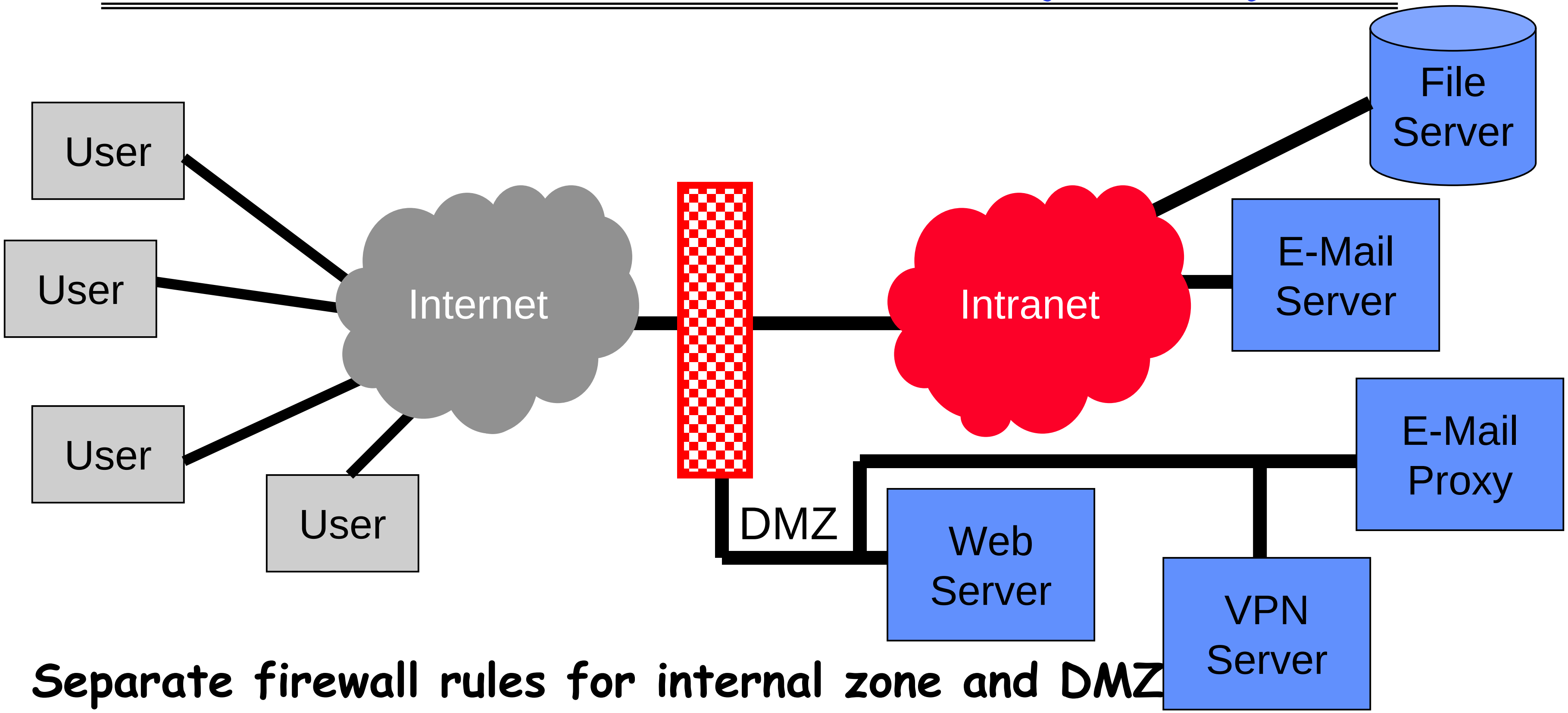
Some Web Server Threats and Attacks

- Replace static content ("defacement")
  - Exploit vulnerability to access Web or File servers
- (Distributed) Denial of Service attack
  - Request large image or emulate complex transaction
- Unauthorized database access
  - Exploit vulnerability (e.g., SQL injection) to read/write database
- Attack server OS or other services
  - Exploit vulnerability to disable server

Stopping *Some* Attacks

- Replace static content ("defacement")
  - Harden server (latest patch levels, minimum services)
  - Limit data on file server
- (Distributed) Denial of Service attack
  - Add load balancer, DNS round-robin, replicated clusters, …
- Unauthorized database access
  - Harden server (latest patch levels, min. svcs)
  - Sanity check all arguments
- Attack server OS or other services
  - Harden servers (latest patch levels, min. svcs)

DeMilitarized Zone (DMZ)



- Separate firewall rules for internal zone and DMZ
  - Internet-DMZ rules only allow web, e-mail traffic
  - DMZ-Intranet rules only allow file, e-mail, remote login *from DMZ*
  - No Internet-Intranet access
- Where to place e-mail server?
  - Add proxy to isolate e-mail access/storage from e-mail forwarding

Intrusion Detection Systems

- Detecting attempts to penetrate our systems
  - Used for post-mortem activities
  - Related problem of extrusion (info leaking out)
- In pre-network days (centralized mainframes)…
  - Primary concern is abuse and insider information access/theft
  - Reliance on logging and audit trails
- But, highly labor intensive to analyze logs
  - What is abnormal activity?
  - Ex: IRS employees snooping records
  - Ex: Moonlighting police officers

Signature vs. Anomaly Detection

- Signatures
  - Language to specify intrusion patterns
  - Packet contents
    - » Could be single or multiple packets (stream reconstruction)
- Anomalies
  - Analyze normal operation (behavior), look for anomalies
  - Uses AI techniques: Statistical Learning Techniques
  - Compute statistical properties of "features"

Some Challenges

- What is normal traffic?
  - Server, desktop, PDA, PDA/phone, …
  - My normal traffic ≠ your normal traffic
  - Lots of data for servers
- Why do we need sufficient signal and noise separation?
  - To avoid too many false alarms!
  - Legitimate IRC usage flagged as bot infection!
- What happens if signals are missed?
  - Possible intrusion!

Honeypots and Tarpits

- Honeypots
  - Closely monitored network decoys
  - May distract adversaries from more valuable machines on a network
- Tarpits
  - Slow down scanning tools/worms to kill their performance/propagation because they rely on quick turnarounds
  - Example:
    - » Allow TCP connection to open, but don't send information through it, and don't let it close.

Buffer Overrun Vulnerabilities

- Most common class of implementation flaw
- C is basically a portable assembler
  - Programmer exposed to bare machine
  - No bounds-checking for array or pointer accesses
- *Buffer overrun* (or *buffer overflow*) vulnerabilities
  - Out-of-bounds memory accesses used to corrupt program's intended behavior

Format String Vulnerabilities

- ```
  void vulnerable() {
    char buf[80];
    if (fgets(buf, sizeof buf, stdin) == NULL)
      return;
    printf(buf);
  }
  ```
- Do you see the bug?
- Last line should be `printf("%s", buf)`
  - If buf contains "%" chars, `printf()` will look for non-existent args, and may crash or core-dump trying to chase missing pointers
- Reality is worse…
  ```

TOCTTOU Vulnerability

- In Unix, often occurs with filesystem calls because system calls are not atomic
- But, TOCTTOU vulnerabilities can arise anywhere there is mutable state shared between two or more entities
  - Example: multi-threaded Java servlets and applications are at risk for TOCTTOU

Principles of Secure Software

- Let's explore some principles for building secure systems
  - Trusted Computing Base & several principles
- These principles are neither necessary nor sufficient to ensure a secure system design, but they are often very helpful
- Goal is to explore what you can do at design time to improve security
  - How to choose an architecture that helps reduce likelihood of system flaws (or increases survival rate)

The Trusted Computing Base (TCB)

- *Trusted Component*:
  - A system part we rely upon to operate correctly for system security
  - (A part that can violate our security goals)
- *Trustworthy components*:
  - System parts that we're justified in trusting (assume correct operation)
- In Unix, the super-user (root) is trusted
  - Hopefully they are also trustworthy…
- *Trusted Computing Base*:
  - System portion(s) that must operate correctly for system security goals to be assured

TCB Definition

- We rely on every component in TCB working correctly
- Anything outside isn't relied upon
  - Can't defeat system's security goals even if it misbehaves or is malicious
- TCB definition:
  - Must be large enough so that nothing outside the TCB can violate security

Three Cryptographic Principles

- Three principles widely accepted in crypto community that seem useful in computer security
  - Conservative Design
    - » Prepare for the worst attack, just in case
  - Kerkhoff's Principle
    - » Do not rely on security through obscurity
  - Proactively Study Attacks
    - » Try to hack your own system

Principles for Secure Systems

- 1. *Security is Economics*
  - No system is 100% secure against all attacks
    - » Only need to resist a certain level of attack
- 2. Least Privilege
  - Only give a program the minimum access privileges it legitimately needs to do its job
- 3. *Use Fail-Safe Defaults*
  - *Default Deny*
- *4. Separation of Responsibility*
  - No one person or program has complete power
- 5. Defense in Depth
  - Secure redundantly
- 6 and 7. Psychological Acceptability / Usability
  - Users should want to use security

Principles of Secure Systems

- 8. Complete Mediation
  - Don't forget to always check
- 9. Least Common Mechanism
  - Be careful about reusing code
- 10. Detect if you can't prevent
- 11. Orthogonal Security
  - Mechanisms should work together
- 12. Don't rely on security through obscurity
- 13. Design security in from the start

Writing Secure Code

- Goal is eliminating *all* security-relevant bugs, no matter how unlikely they are to be triggered in normal execution
  - Intelligent adversary will find abnormal ways to interact with our code
- Different goal from software reliability
  - Focus is on most likely to happen bugs
  - Can ignore obscure condition bugs
- Dealing with malice is much harder than dealing with mischance

Three Fundamental Techniques

- (1) Modularity and decomposition for security
- (2) Formal reasoning about code using invariants
- (3) Defensive programming

Defensive Programming

- Like defensive driving, but for code:
  - Avoid depending on others, so that if they do something unexpected, you won't crash – survive unexpected behavior
- Software engineering focuses on functionality:
  - Given correct inputs, code produces useful/correct outputs
- Security cares about what happens when program is given invalid or unexpected inputs:
  - Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs
- Defensive programming
  - Apply idea at every interface or security perimeter
    - » So each module remains robust even if all others misbehave
- General strategy
  - Assume attacker controls module's inputs, make sure nothing terrible happens

Process for Writing Function Code

- First write down its preconditions and postconditions
  - Specifies what obligations caller has and what caller is entitled to rely upon
- Verify that, no matter how function is called, if precondition is met at function's entrance, then postcondition is guaranteed to hold upon function's return
  - Must prove that this is true for all inputs
  - Otherwise, you've found a bug in either specification (preconditions/postconditions) or implementation (function code)

Loop Invariant

- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - » Both a pre- and post-condition for the loop body
- Example: Factorial function code
  - ```
    /* Requires: n >= 1 */
    int fact(int n) {
        int i, t;
        i = 1;
        t = 1;
        while (i <= n) {
            t *= i;
            i++;
        }
        return t;
    }
    ```
  - Prerequisite: input must be at least 1 for correctness
  - Prove: value of `fact()` is always positive
    ```