Week 7 - Friday

COMP 4290

Last time

- What did we talk about last time?
- Therac-25
- Malicious code
- Viruses

Questions?

Assignment 3

Project 2

Virus Case Studies

Brain virus

- The Brain virus is one of the oldest known
  - It changed the label of disks it attacked to "BRAIN"
  - It was written by two brothers from Pakistan
- It copies itself to the boot sector in MS-DOS
- It rewrites the system interrupt for disk reading so that it controls reads
  - If you try to look at the boot sector, it will lie to you about what's there
- Anytime it sees an uninfected disk, it infects it
- It doesn't otherwise do anything malicious

The Internet Worm

- In 1988 Robert Morris, a Cornell graduate student, wrote an worm that infected a lot of the Internet that existed at that time
- Serious connectivity issues happened because of the worm and because people disconnected uninfected systems
- He claimed the point was the measure the size of the Internet
- The worm's goal:
  1. Determine where it could spread to
  2. Spread its infection
  3. Remain undiscovered

Determining where to spread

- It tried to find user accounts on the host machine
  - It tried 432 common passwords and compared their hash to the list of password hashes
  - Ideally, this list should not have been visible
- It tried to exploit a bug in the **`fingerd`** program (using a buffer overflow) and a trapdoor in the **`sendmail`** mail program
  - Both were known vulnerabilities that should have been patched

Spreading infection

- Once a target was found, the worm would send a short loader program to the target machine
- The program (99 lines of C) would compile and then get the rest of the virus
- It would use a one-time password to talk to the host
- If the host got the wrong password, it would break connection
- This mechanism was to prevent outsiders from gaining access to the worm's code

Remain undiscovered

- Any errors in transmission would cause the loader to delete any code and exit
- As soon as the code was successfully transmitted, the worm would run, encrypt itself, and delete all disk copies
- It periodically changed its name and process identifier so that it would be harder to spot

What happened

- The worm would ask machines if they were already infected
- Because of a flaw in the code, it would **reinfect** machines 1 out of 7 times
- Huge numbers of copies of the worm started filling infected machines
  - System and network performance dropped
- Estimates of the damage are between $100,000 and $97 million
  - Morris was fined $10,000 and sentenced to 400 hours of community service
- The CERT was formed to deal with similar problems

Code Red

- Code Red appeared in 2001
  - It infected a quarter of a million systems in 9 hours
  - It is estimated that it infected 1/8 of the systems that were vulnerable
- It exploited a vulnerability by creating a buffer overflow in a DLL in the Microsoft Internet Information Server software
- It only worked on systems running a Microsoft web server, but many machines did by default

Versions

- The original version of Code Red defaced the website that was being run
- Then, it tried to spread to other machines on days 1-19 of a month
- Then, it did a distributed denial of service attack on whitehouse.gov on days 20-27
- Later versions attacked random IP addresses
- It also installed a trap door so that infected systems could be controlled from the outside

Countermeasures

Countermeasures for developers

- Write **modular code**
  - Robust independent components
- Components should meet the following criteria:
  - **Single-purpose:** Perform one function
  - **Small:** Short enough to be understandable by a single human
  - **Simple:** Simple enough to be understandable by a single human
  - **Independent:** Isolated from other modules

Modularity

- Modular components have many advantages
- Maintenance
  - It's easy to replace a modular component
- Understandability
  - It's easier to understand a large system made out of simple components
- Reuse
  - Modular components can be reused in other code
- Correctness
  - It's easy to see which component is failing
- Testing
  - Each component can be tested exhaustively on its inputs and outputs

Encapsulation

- Components should hide their implementation details
- Only the smallest number of public methods should be kept to allow them to interact with other components
- This information hiding model is thought of as a **black box**
- For both components and programs, one reason for encapsulation is **mutual suspicion**
  - We always assume that other code is malicious or badly written

Testing

- **Unit testing** tests each component separately in a controlled environment
- **Integration testing** verifies that the individual components work when you put them together
- **Regression testing** is running all tests after making a change, verifying that nothing that used to work is now broken
- **Function** and **performance tests** sees if a system performs according to specification
- **Acceptance testing** give the customer a chance to test the product you have created
- The final **installation testing** checks the product in its actual use environment

Secure design principles

- Saltzer and Schroeder wrote an important paper in 1975 that gave eight principles that should be used in the design of any security mechanisms
  1. Least privilege
  2. Fail-safe defaults
  3. Economy of mechanism
  4. Complete mediation
  5. Open design
  6. Separation of privilege
  7. Least common mechanism
  8. Psychological acceptability
- These principles will be part of Project 3

Principle of least privilege

- The **principle of least privilege** states that a subject should be given only those privileges that it needs in order to complete its task
- This principle restricts how privileges are granted
- You're not supposed to get any more privileges than absolutely necessary
- Examples
  - Banner
  - Unix systems
  - Windows systems?

Principle of fail-safe defaults

- The **principle of fail-safe defaults** states that, unless a subject is given explicit access to an object, it should be denied access to an object
- This principle restricts how privileges are initialized
- A subject should always be assumed not to have access
- Examples
  - Airports
  - Unix systems
  - Windows systems?

Principle of economy of mechanism

- The **principle of economy of mechanism** states that security mechanisms should be as simple as possible
- This principle simplifies the design and implementation of security mechanisms
- The more complex a system is, the more assumptions that are built in
- Complex systems are hard to test
- Examples
  - Die Hard
  - Houdini

Principle of complete mediation

- The **principle of complete mediation** requires that all access to objects be checked to ensure that they are allowed
- This principle restricts the caching of information (and also direct access to resources)
- The OS must mediate all accesses and make no assumptions that privileges haven't changed
- Examples
  - Banks
  - Unix systems

Principle of open design

- The **principle of open design** states that the security of a mechanism should not depend on the secrecy of its design or implementation
- "Security through obscurity" fallacy
- Examples
  - Enigma
  - RSA
  - Lock-picking

Principle of separation of privilege

- The **principle of separation of privilege** states that a system should not grant permission based on a single condition
- Security should be based on several different conditions (perhaps two-factor authentication)
- Ideally, secure mechanisms should depend on two or more independent verifiers
- Examples
  - Nuclear launch keys
  - PhD qualifying exams
  - Roaccutane (used to be Accutane)

Principle of least common mechanism

- The **principle of least common mechanism** states that mechanisms used to access resources should not be shared
- Sharing allows for channels for communication
- Sharing also lets malicious users or programs affect the integrity of other programs or data
- Examples
  - Virtual memory
  - File systems

Principle of psychological acceptability

- The **principle of psychological acceptability** states that security mechanisms should not make the resource (much) more difficult to access than if the security mechanisms were not present
- Two fold issues:
  - Users must not be inconvenienced or they might fight against the system or take their business elsewhere
  - Administrators must find the system easy to administer
- Examples
  - Windows UAC
  - Retina scans
  - Changing your password all the time

Secure coding practices

- Top 10 Secure Coding Practices from the CERT

1. Validate input
2. Heed compiler warnings
3. Architect and design for security policies
4. Keep it simple
5. Default to deny
6. Adhere to the principle of least privilege
7. Sanitize data sent to other systems
8. Practice defense in depth
9. Use effective quality-assurance techniques
10. Adopt a secure coding standard

Penetration testing

- **Penetration testing** is when a team that didn't design or implement the software tries to break into it
- Also called **tiger team analysis** or **ethical hacking**
- It's a great tool, but there's no guarantee it will work quickly
- Also, there's no guarantee that all vulnerabilities will be found
- The Google Vulnerability Reward Program (VRP) is a crowd-sourcing approach to penetration testing Google
  - You can make $200 to $101,010 per vulnerability you find

Formal verification

- It is possible to prove that *some* programs do specific things
  - You start with a set of preconditions
  - You transform those conditions with each operation
  - You can then guarantee that, with the initial preconditions, certain postconditions will be met
  - Using this precondition/postcondition approach to formally describe programming languages is called **Hoare semantics**
- Proving things about complex programs is hard and requires automated use of programs called **theorem provers**

Validation

- **Validation** is checking the design against the requirements
  - Verification is checking the implementation against the design
- Program validation is often done in the following ways:
  - Requirements checking
  - Design and code reviews
  - System testing

Defensive programming

- **Defensive programming** assumes any input could be bad
- Types of input to watch out for:
  - Value inappropriate for data type
  - Value out of range
  - Value unreasonable
  - Value out of scale or proportion (similar to unreasonable)
  - Incorrect number of parameters
  - Incorrect order of parameters

Design by contract

- Programming by contract is related to formal verification
- Each module of code should have preconditions, postconditions, and invariants
- One way to check that conditions are not met is with an **assertion**
- Assertions are statements in a language that will throw an error if they are not true

```
double findHypotenuse(double a, double b) {
      assert a > 0 && b > 0; // Assertions must be on
      return Math.sqrt(a*a + b*b);
}
```

Countermeasures that don't work

- Penetrate-and-patch
  - Fixing a fault can have non-obvious side-effects
  - Focusing too narrowly on one fault may ignore deeper problems
  - Fixing a problem isn't workable because of performance
- Security by obscurity
  - Example: don't tell people what encryption algorithm is being used
  - If internals leak out, security is useless
- A perfect bad code detector
  - Impossible because of the halting problem

Ticket out the Door

Upcoming

Next time…

- Web security
- Obtaining user or website data
- E-mail attacks
- OS background
- Hussein Alani presents

Reminders

- Reading section 4.1 – 4.4
- Work on Assignment 3
- Work on Project 2