Class & Article Notes:

How does this help me kill people? That is why we wear the uniform.

Game Idea:

I am going to develop a game that will help players become more familiar with C code, identify known vulnerabilities, and theoretically exploit code using existing vulnerabilities. My game idea is a card-based game called “C-Sploit” for 2+ players. The game could also be used by a single individual to improve their skill in identifying vulnerabilities in code. The game will have three different types of cards. First, code snippet cards show small samples of C code that may or may not contain known vulnerabilities. The backside of each code snippet card reveals if the code is secure or not and identifies the possible exploits that could be used against the example code. The second kind of cards are exploit cards which have known exploits or attacks derived from multiple sources. Each player will have 4 of these cards in their hands. The third type of card are a type of exploit card, but instead of a specific exploit the cards will say either “The code is secure” or “the code is vulnerable.” Each player will have one of each of these cards in their hand in addition to the 4 exploit cards.

One code snippet card will be played for each round of the game. Every player will examine the code and play one exploit card (face down in front of the player). This card could be an actual exploit to attack the code snippet or state that the player thinks the code is vulnerable or not by playing the appropriate card. Once all cards are played for the round each player reveals their exploit. For greatest effect, each player should explain why they played what they did. Once this is completed, the code snippet card is flipped over revealing the answers. Points are awarded to players based on the following scale:

* Correctly identified code as secure (1 point)
* Correctly identified code as not secure (1 point)
* Played “the code is secure” card incorrectly (-2 points)
* Played “the code is vulnerable” card incorrectly (-1 point)
* Played a card containing an exploit that was incorrect (-2 points)
* Played a card containing an exploit that was correct (3 points)

After each round, all the cards are cleared to the discard pile. All players take back their secure/vulnerable cards and draw an additional two exploits cards. Each player must then discard one or two exploit cards to return to a six-card hand (4 exploits plus the secure/vulnerable pair cards). The additional step of drawing and discarding exploit cards rewards players that can identify more versatile exploits that might work against multiple code snippets.

As part of the design multiple exploits could be successful against multiple code snippets and code snippets should either be secure or vulnerable to more than one exploit.

* Focus on simple game design

4-Ps of Software Engineering:

* Project
* People
* Process (need someone else to oversee the process)
* Product

Software Engineering:

* Project Planning
* Requirements Analysis
* System Design
* Implementation
  + How do we write secure software?
  + Coding practices
* Testing
  + How do we check to see if it is right?
* Delivery
* Maintenance
  + This is hard
  + Ongoing maintenance and development is critical to using something longer than 6-12 months

CIA2 = Confidentiality, Integrity, Availability, Authentication (Non-Repudiation)

Need to be “defensive programmers”

Tuesday, 9 October:

* Topics due on Friday, 12 October
* Virtual versus Physical Addressing
  + ASLR? Address Space Layout Randomization: Make it much harder to predict a memory location for a specific application/program
* In C a string is an array of characters, it is null terminated. They will keep reading until they hit a null.
* argv[0] = name of the program
* “off by one” is a common example of Buffer overflows

Friday, 12 October: Buffer Overflows/Overrun

* What is a buffer overrun?
* Trampolining: Jump to an area of code by using another instruction to go to where we want to program to end up.
* Mostly referring to variables sitting on a stack (lesser known
  + This means we are dealing with char\* (using malloc) instead of char[]
* Null Pointers – our injected code cannot contain a null or zero
* Inexact code/buffer locations? Use “nop” [No Operation] to fill up a huge block of space and then put my arbitrary code below it. As long as the program jumps into that NOP area, then it will get to our code.
  + NOP command does nothing and moves to the next instruction
* Return Oriented Programming?
* DEP – Data Execution Protection (doesn’t let data execute for specific areas)
  + If we protect the stack with DEP, then it would be safe from running
* Attacking ASLR: brute force until you find the offset (the offset is the same for each instance of the program)
* 64-bit machines fix this issue
* There are ways to make ASLR more secure

Automated Detection of Buffer Overflow Vulnerabilities:

* Old papers, but solid analysis

StackGuard:

* Sticks canaries into the code so that if they change the program stops execution because we know a buffer overrun has completed

Tools for detecting buffer overruns:

* Static code analysis
* Fuzzing
* Look at the last iteration of a function where programmers get lazy

Always test your code with memory error detection to catch mistakes (buffer overruns, memory leaks)

More research papers on BlackHat and Phrack

Format String Vulnerabilities

printf(“%s”, userString) 🡪 this is safe

sprintf(var, “5s”, userString) 🡪 this is also safe

log (var) //this may not be safe depeding on what is in the log

%x – prints the first 4 bytes to the string

Class Notes: 19 October (Integer Overflows)

* Midterm: Look at slides. Is this code secure? Why not? What could make it secure?

Usability:

* Security only works if the secure way happens to be the easy way
* Security must be part of the system requirements to be considered with usability
* Make the UI simple and clear
* If you have a password requirement; make sure it is available to the user while they are making a password. Be careful of listing the password requirements at every login
* Make security decisions for your users (as the defaults)
* Avoid too many choices
* Make selective relaxation Easy
* Make it actionable

Aligning Security & Usability

Not Updating Easily:

* Tied to poor usability
* If you annoy the end-users, they will ignore you
* Too many security updates are a problem
  + Must keep up with security fixes
  + Prompt fatigue (maybe incorporate other notification methods)
  + Updating without notifying (user needs to be informed)
* Cumbersome Updates
  + Updating one sys at a time
  + Forcing a reboot
  + Needing to read a manual
* Update Signatures – who can you trust?
  + Not DNS
  + Need to check everything (even after unzipping)

Error Handling:

* Error Messages:
  + No universal conventions
  + Return values – should be checked
    - “goto” provides try-catch functionality
  + Thrown errors – should be dealt with every time
* Handling Exceptions:
  + Do not “gobble” exceptions
  + Detail actions for each exception that you may have
  + Clean up the mess
* Testing Techniques:
  + This is hard to test

DoD Directive: Thou shalt patch every 21 days

Executing Code with Too Much Privilege:

* What’s the big deal?
* Mitigate 92% of critical security vulnerabilities in MS products could be mitigated if users used non-admin accounts
* Use least privilege
* Use elevated privileges for the shortest amount of time possible
* Other exploits become much stronger when they exploit a program running with an admin account
* Do we take this into account when handling errors/exceptions?
* Do not access the Internet while at a higher privilege.
* Don’t put executable in the user directory

Command Injection:

* Input data passed to some sort of compiler or interpreter.
* Watch for semicolons (which ends a statement)
* UNIX: semicolon, backtick, vertical bar (everything after the bar is treated as another, related process)
* O/S Command?
  + Use an API to call the command directly without going through a shell
    - UNIX box – execv() family of functions;
    - Doesn’t work all the time
  + Whitelist versus Blacklist
    - It is hard to determine everything that we don’t want to accept, but there are several different cases where different characters will be allowed.
* If there is an error…
  + Don’t give the attacker too much information
  + Don’t just copy back the bad data (invalid character in input)
  + This is a dance

SQL Injection:

* SQL code is passed to the database when not intended.
* “--" inserts a comma in SQL
* Don’t format commands based on user’s input
* MS SQL can lead to command line access
* Need to prepare a command and then stick the input into it.
  + SQLcmd x = prepart(cmd);
  + SQLset (x, 1, username); //username is from user input
  + SQL set (x, 2, userpwd); //userpwd is from
  + SQLset (x);

2 Nov: Doctoral Thesis Presentation (Mitchell)

* RAPTR Laboratory
* Avionics Cyber Vulnerabilities and Mitigations Branch:
  + Identify and Mitigate
  + Cyber Resiliency
  + New Avionics Architectures
* How to respond to cyber exploitation in aircraft?
* Applying Binary Diversity to avionics security
* Inspiration: the immune system

9 Nov:

* Windows: uses ACLs to set permissions for each user or group
* Linux: uses either individual, group, or world permissions
* FAT: does not include file/folder permissions
* Spotting the Sin: Windows (everyone, guest), Linux (world, group)

Exam Review:

* Short Answer
* 1 or more code review questions (find the vulnerabilities)
* Software Engineering & Security
  + Through the cycle (req to mnx)
  + CIA^2
  + Security is emergent property of software system (bake it in)
  + Security Feature != Secure Software
  + Requirements:
    - Multi-user (permissions, race conditions, executing with too much privilege)
    - Usability
    - Updating easily
    - Uptime rate
  + Design:
    - Feedback loop with requirements/design
  + Implementation:
    - Buffer Overflows, Format Strings, Command Injection, error messaging, race conditions, SQL injection
  + Re-entry of code (it is safe to have multiple people enter the code)
  + Testing:
    - Use cases, abuse cases, misuse cases, fuzzing, think like attacker
    - Use multiple different perspectives
  + Delivery:
    - Signing code, ensuring code is what it is supposed to say
    - Ensuring encryption
  + Mnx
  + Different ways to classify vulnerabilities?
    - Attack Vectors
    - How are you vulnerable
    - Where are you vulnerable
    - Developing use cases
  + How to execute a buffer overflow/overrun?
* Legacy code is a big problem in secure software design.

For Every vulnerability:

* What to look for in code review
* Dynamic testing? What sorts of tests should you run
* Mitigation techniques

Buffer Overflows:

* This is a continued problem
* Build it: find a way to stick more into the buffer than designed
* Overwrite return address on stack
* Could overwrite return address
* Return to libc (?)
* Imbed code in overflow and then return to that code
* Java is not as vulnerable, but there are still attack vectors.
* Java itself is not vulnerable to buffer overflows (will not execute arbitrary code)
* Heap: dynamically allocated variables
* Stack: local variables
* Off by one (able to know the difference between string length and buffer size
* Itanium Chip? The return address is now in a register not
* Non-Executable pages, Address space randomization (64 bit good for this over 32 bit)
  + ASLR
* Canaries: segments spaced in the code that if they are overwritten then something went wrong.
* Mitigation Techniques: NX pages, canaries, randomization (ASLR?)
* What is the heap NX?

Format Strings:

* Format strings crawl up the stack, read what is in memory and can rewrite (%n can rewrite).
* What does %x do?
* %s can print the string that is held at that pointer location
* Languages: C, C++
* How does FormatGuard work? What does it do? (Check number of arguments in a format string)
* Legacy code does not let us change this for the language
* Not everyone used FormatGuard or other protections
* Type qualifiers: taint/untaint?
* How to do static analysis? Tracking data as it flows through the code.
* Code Review:
  + Printf(); scanf()
  + Printf() of user input that I am not checking
* Dynamic testing: Fuzzing, adding %x to username/password
* 32-bit vs 64-bit: Null byte makes it harder to input memory addresses in the stack. Null character will end the printf() statement.
* More data in registers versus in the stack (less for an attacker to work with)

Integer Overflows

* Java is vulnerable to these
* Upcasting information in C
* Can this code overflow? (typecasting)
* What languages are vulnerable: pretty much every one
* Signed/unsigned integers
* Use standard types for sizes; use unsigned when reasonable
* Code Review:
  + Counters
  + Math
  + Operations with different types
  + User input
* Dynamic testing
  + Data fuzzing (around 0, 255, boundary cases)

**Design Vulnerabilities:**

Usability

* Secure way happens to be the easy way
* Passwords are a big problem
* Making the unsecure way the easy way
* Admin accounts
* Different types of users (developer, end user, admin user)
* Admins: doing things in bulk, scripting, remote accessing boxes
* Security versus admonition or designation? Different security paradigm? What would a new OS look like?

Not Updating Easily:

* Cumbersome updates
* Prompt fatigue
* Updating without notifying
* Forcing reboots
* Signatures – who can you trust?
* Recovery plans – don’t use the code itself for updates. Use a separate install prgm
* Need to trust where zipped files are stored or “unpacked”

Error Handling:

* No universal standard for what is returned in a failure
* How much data do you present to your “user”
* Catching errors – handle every error gracefully
  + Cleanup actions
  + Exit cleanly
* Danger of copy/pasting sample code is dangerous. Not posting error handling code

Executing Code w/ Too Much Privilege:

* Must use admin account to run? Probably a bad design
* Correctly handle admin username/password
* Keep elevated privilege to a minimum. What happens with error handling while privilege is elevated.
* Do not access the internet with an admin account

Command & SQL Injection:

* User passes input that is interpreted as a command
* Not checking user input
* Co-locating data and code (using input to build a command/instruction)
* Languages: all; SQL instructions
* Usually after a command shell;
* Error statements are a problem with SQL injections
* Dynamic Testing: system commands, SQL commands, Fuzzing

Information Leakage:

* Leakage by accident (spillage)
* Leakage by intent (mis communication between developers and designers)
* Side channel attacks.
  + What are timing attacks?
    - Does it take longer/shorter for error to return (provide more information than intended by the designer/developer)
  + Storage channels:
* Errors and Logs – store them far away where they can’t be accessed easily.

Race Conditions:

* Two different processes trying to access the same resource at the same time
* Some instructions take a lot of cycles (system functions), leaving files vulnerable (especially temp files).
* “Re-entrant” code (static local variables, …)
* Signal handlers
  + Mouse/keyboard processing
  + Error handling
  + Must be re-entrant, yes. Look for the variables they manipulate.
* Code Review:
  + Verifying re-entrant code, atomic code
  + Hitting global variable
  + Is there a chance for multi-threaded code?
  + Hard to test dynamically
  + Maybe use by fast multi-processing device

Failing to Protect Stored Data:

* Windows and Linux access controls
* NTFS vs FAT (FAT does not have file protection)
* Threat Modeling?
  + Misuse cases
  + Looking for vulnerable data

Weak Passwords:

* What is a weak password?
* Throttling login attempts? (1, 5, 10 minute lockouts)
* Handling password resets
* Multi-factor authentication

Strong Random Numbers and Crypto:

* Originally used for testing (needed to duplicate process)
* Language RNGs are not good
* Need a good seed for Crypto Pseudorandom number generators
* How does a block cipher work?
* Asymmetric versus symmetric crypto (when would you use them)
  + “A” when passing a symmetric key
  + ‘symmetric’ when doing large amounts of data

Software Security Testing:

* Designers, Developers, Testers
* Think like an attacker (is DoS enough)
* Misuse cases, abuse cases, use cases
* Flaws created by security features
* Pen-testing: looking at the seems

**20 Nov 2018: Mobile Code and Web Applications**

Attack Surface – Keep in mind, similar to ORM

Mobile and WebApps are the future

Mobile Code:

* Inherently put code and data in the same place
* Watch for code escaping its container (like escaping a VM)
* Watch for extensions (on browsers, etc)

Two aspects to consider:

1. Container
   1. The nature of this is that you are downloading and executing arbitrary code
   2. Must keep malicious code contained
   3. Balancing usability and security
2. Code
   1. Probably not designed by a professional coder

Web Server Vulnerabilities:

* The problem is with the weblinks that are sending malicious code to the webserver to be returned to the person who clicked on the link.
* The attacker adds malicious code to the link
* Magic URLs add code to the end of a link to give instructions to the web server
* HTTP is stateless

Examples:

* Replay Attack: mitigate by encrypting traffic
* Prediction Attacks: only works if things are sent in the clear
* Integrity Attacks: hidden fields are changed to send malicious code

**Paper Feedback:**

* Do not use first person
* Literature review needs to be in paragraph form.
* The goal of the lit review is answer the why question and motivate/lay the foundation for the “how” question
* Minimize quotes. Don’t look at the source paper while you are writing yours.
* The abstract needs to summarize the whole paper. Do not use quotes or citations in the abstract
* Use present tense and active voice.
* Do not use emotional words like very, obviously, “astute reader”
* Any non-cited sentence is the author’s opinion/believe. There’s no reason to say “I believe” or “I think.”

Presentation Questions:

* Intended Audience, will they be able to play?
* Objectives and how the game met the objectives