- Fall Term 2025 -

Cybersecurity -- Student Activity

ASST PROF. ESSAID MERYAM - DEPARTMENT OF ROBOTICS ENGINEERING, KMU

C++ for Cybersecurity – Hands-On Labs & Mini Project

Lab 1: Buffer Overflow Exploitation

Objective
Demonstrate how a classic stack-based buffer overflow can be exploited to hijack program control flow and redirect execution to a privileged function (e.g., `win()`).

Why This Matters
Buffer overflows are among the oldest—and most impactful—vulnerabilities in software. Understanding them helps you recognize unsafe patterns in real-world code and appreciate modern exploit mitigations like stack canaries, ASLR, and DEP.

Implementation Guide

1. **Prepare the Vulnerable Program**

   - Write a C++ program that contains a fixed-size character buffer (e.g., `char buf[64]`).
   - Use an unsafe input function like `cin.getline(buf, 256)` to read far more data than the buffer can hold.
   - Include a separate function (e.g., `win()`) that prints a success message or flag.
   - In `main()`, print the runtime address of `win()`—this simulates an information leak that attackers often rely on.
2. **Compile with Weak Protections (For Learning Only)**
   Use the following compiler flags to disable modern defenses:

   ```
   g++ -g -fno-stack-protector -no-pie -z execstack -o lab1 lab1.cpp
   ```
   - `-g`: Includes debugging symbols for GDB.
   - `-fno-stack-protector`: Disables stack canaries.
   - `-no-pie`: Makes memory layout predictable (disables ASLR for the executable).
   - `-z execstack`: Allows code execution on the stack (needed if using shellcode later).

   > **Never use these flags in production code.**

3. **Find the Exact Offset to Overwrite the Return Address**

   - Run the program in **GDB** (`gdb ./lab1`).

- Use the **Metasploit pattern tool** to generate a unique 100-byte string:
  `/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 100`
- Paste this pattern as input when the program prompts you. It will crash.
- In GDB, inspect the instruction pointer (`info registers rip` on x64).
- Use `pattern_offset.rb` to determine how many bytes are needed before overwriting the return address (e.g., 72).

4. **Craft and Deliver the Exploit**

   - Note the printed address of `win()` when you run the program.
   - Construct a payload:
     - 72 bytes of padding (e.g., `'A' * 72`)
     - Followed by the **little-endian** address of `win()`
   - Pipe this payload into your program using Python or `echo` .
   - If successful, the program will jump to `win()` and print your flag.

Learning Outcome: You'll see firsthand how memory layout, calling conventions, and unchecked input lead to full control hijacking.

Lab 2: Reverse Engineering a C++ Binary with Ghidra

Objective
Analyze a stripped C++ binary to recover hidden logic, identify a password check, and extract an encrypted flag using static and dynamic analysis techniques.

Why This Matters
Malware and proprietary software often hide secrets (passwords, keys, C2 addresses) using simple obfuscation like XOR. Reverse engineering is essential for malware analysis, vulnerability research, and digital forensics.

Implementation Guide

1. **Understand the Target Binary**

   - You'll be given (or compile) a C++ program that:
     - Prompts for a password
     - Compares it to a hardcoded value
     - If correct, decrypts and prints a flag using a simple algorithm (e.g., XOR with a constant key)
   - Compile it with symbols stripped and optimizations enabled to mimic real-world binaries:
     `g++ -o crackme crackme.cpp -s -O2`

2. **Static Analysis in Ghidra**

   - Open Ghidra and create a new project.
   - Import your binary and run the auto-analyzer.

- **Crucially, ensure “Demangle C++ Names” is enabled**—this converts mangled symbols like `_ZN6Crackme4mainEv` back to readable function names.
- Navigate to the `main` function in the decompiler view.
- Look for:
  - String comparisons (`strcmp`, `operator==`)
  - Hardcoded strings (potential password)
  - Function calls that might handle decryption

3. **Locate the Encrypted Flag**

   - Find the function responsible for flag decryption.
   - In the decompiler, identify the encrypted byte array and the XOR key (often a small constant like `0x13`).
   - Note the sequence of encrypted bytes.

4. **Decrypt the Flag**

   - Write a short Python script to XOR each byte with the key and convert the result to ASCII.
   - Alternatively, use **x64dbg** (on Windows) to run the binary dynamically:
     - Set a breakpoint after the password check
     - Enter the correct password
     - Step into the decryption routine and watch memory to see the flag appear in plaintext

Learning Outcome: You’ll practice mapping assembly back to high-level logic, handling C++ constructs like `std::string`, and defeating basic obfuscation.

Lab 3: Educational Keylogger (Windows)

Objective
Build a Windows keylogger that demonstrates low-level input monitoring using global hooks—while incorporating ethical safeguards to prevent misuse.

Why This Matters
Keyloggers are common in both offensive toolkits and legitimate monitoring software (e.g., parental controls). Understanding their internals helps defenders detect them and developers build responsibly.

Implementation Guide

1. **Design with Safety First**

   - Your keylogger **must not log anything by default**.
   - Only activate logging if a file named `ENABLE_LOGGING` exists in the current directory.
   - Allow the user to **press ESC to exit cleanly** at any time.

- Log output to a file (e.g., `keylog.txt`), not just the console.

2. **Use Windows API for Global Hooking**
   - Use `SetWindowsHookEx(WH_KEYBOARD_LL, ...)` to install a **low-level keyboard hook**.
   - This requires a **message loop** (typically in `main()` or a dedicated thread) to keep the hook alive.
   - Your hook callback function will receive every keystroke system-wide.

3. **Process Keystrokes Safely**
   - In the hook callback:
     - Check if `ENABLE_LOGGING` exists before logging.
     - Convert virtual key codes (`vkCode`) to readable characters (e.g., `'A'`, `[ENTER]`, `[ESC]`).
     - Handle special keys (space, enter, backspace, escape) appropriately.
     - When ESC is pressed, set a global flag to signal the program to exit.

4. **Ensure Thread Safety and Cleanup**
   - If writing to a file from the hook (which runs in a system thread), use a **mutex** to prevent race conditions.
   - Always call `UnhookWindowsHookEx()` before exiting.
   - Close file handles and clean up resources.

5. **Build and Test Responsibly**
   - Compile with:
     `g++ -o keylogger.exe keylogger.cpp -luser32`
   - **Only test on your own machine**.
   - Create the `ENABLE_LOGGING` file to start recording.
   - Press ESC to stop and verify the log file contents.

Legal & Ethical Disclaimer

> This lab is for **educational purposes only**. Unauthorized monitoring of user input violates privacy laws (e.g., CFAA, GDPR, Computer Misuse Act). You must have **explicit written permission** to run such tools on any system other than your own.

Learning Outcome: You'll gain hands-on experience with Windows internals, ethical coding practices, and defensive awareness of common surveillance techniques.

Mini Project: Build a Practical Security Tool in C++

Objective
Apply your knowledge of C++ and cybersecurity by designing and implementing a functional security tool from scratch. This project simulates real-world development tasks

performed by security analysts, red team operators, and defensive engineers. You will gain hands-on experience with low-level system interaction, memory safety, concurrency, and ethical coding practices.

> **Important**: This is not just a programming exercise—it's an opportunity to think like a security professional. Consider how your tool could be used responsibly, how it might be abused, and how you can make it robust and secure.

Choose One of the Following Projects

Option 1: Multi-Threaded Port Scanner with Timeout

Goal
Create a TCP port scanner that checks whether ports on a target machine are open, using multiple threads for speed and a configurable timeout to avoid hanging on unresponsive ports.

Why This Matters
Port scanning is a foundational technique in network reconnaissance. Real tools like Nmap use advanced timing, threading, and protocol handling—yours will implement core concepts in C++.

Implementation Guide

1. **Understand the Basics**
   - Research how TCP `connect()` works. An open port accepts the connection; a closed one refuses it.
   - Learn the difference between **blocking** and **non-blocking** sockets.
2. **Set Up Networking**
   - On **Windows**: Use Winsock2 (`#include <winsock2.h>`, link with `ws2_32.lib`).
   - On **Linux**: Use POSIX sockets (`#include <sys/socket.h>`, etc.).
3. **Build a Single-Port Checker**
   - Write a function that takes an IP address and port number.
   - Attempt a TCP connection and return whether it succeeded.
   - Test on `127.0.0.1` with known services (e.g., port 80 if a web server is running).
4. **Add Connection Timeout**
   - Convert your socket to non-blocking mode.
   - Use `select()` (Linux/Windows) or `WSAEventSelect()` (Windows) to wait for connection completion with a time limit (e.g., 1 second).
   - Handle three outcomes: success, timeout, error.
5. **Introduce Multi-Threading**

- Use `std::thread` to scan multiple ports concurrently.
- Limit the number of active threads (e.g., max 50) to avoid overwhelming the system.
- Use a `std::mutex` to safely print results to the console from multiple threads.

6. **Polish and Validate**

- Accept command-line arguments for target IP, port range, and timeout.
- Validate inputs (e.g., IP format, port numbers between 1–65535).
- Test against localhost and document your results.

Option 2: File Integrity Monitor (SHA-256 Hash Watcher)

Goal
Develop a program that monitors a specified file and alerts you when its content changes by comparing SHA-256 hashes over time.

Why This Matters
File integrity monitoring is used in antivirus software, endpoint detection and response (EDR), and system auditing to detect unauthorized changes (e.g., malware tampering with system files).

Implementation Guide

1. **Understand Cryptographic Hashing**

- SHA-256 produces a unique 64-character hexadecimal string for any file. Even a 1-byte change alters the entire hash.

2. **Choose a Hashing Method**

- On **Windows**: Use the CryptoAPI (`CryptAcquireContext` , `CryptCreateHash` , etc.; link with `advapi32.lib`).
- On **Linux**: Use OpenSSL (`#include <openssl/sha.h>`) or implement SHA-256 from a trusted reference (advanced).

3. **Implement Hash Computation**

- Read the file in **chunks** (e.g., 4KB at a time) to handle large files efficiently.
- Feed each chunk into the hash function.
- Return the final hash as a lowercase hex string.

4. **Build the Monitoring Loop**

- Read the target file path from the command line.
- Compute and store the initial (“baseline”) hash.
- Enter a loop: sleep for 2 seconds → recompute hash → compare to baseline.
- If different, print an alert and update the baseline to avoid repeated notifications.

5. **Handle Edge Cases**

- What if the file is deleted or locked? Log a warning and continue.

    - What if the file is very large? Ensure your chunked reading doesn't load everything into memory.
6. **Test Thoroughly**
    - Create a test file (e.g., `test.txt`).
    - Run your monitor, then edit the file in a text editor.
    - Verify your tool detects the change and reports it with timestamps.

Option 3: Simple Debugger That Sets Software Breakpoints

> **Warning**: This project involves modifying the memory of other processes. **Use only on your own system** or with explicit written permission.

Goal
Write a program that attaches to a running process (e.g., `notepad.exe`), locates a target function (e.g., `MessageBoxA`), replaces its first instruction with a software breakpoint (`0xCC`), waits, then restores the original byte.

Why This Matters
Debuggers and malware analysis tools use breakpoints to pause execution and inspect program state. Understanding this mechanism is key to both offensive and defensive reverse engineering.

Implementation Guide

1. **Learn About Software Breakpoints**
    - The `INT3` instruction (`0xCC`) triggers a debug exception when executed.
    - Debuggers temporarily replace code with `0xCC` to pause execution.
2. **Set Up Windows API Access**
    - Include `windows.h` and `tlhelp32.h`.
    - No extra libraries are needed—everything is in the Windows SDK.
3. **Find the Target Process**
    - Use `CreateToolhelp32Snapshot` and `Process32First/Next` to locate a process by name (e.g., `"notepad.exe"`) and get its Process ID (PID).
4. **Open the Process**
    - Call `OpenProcess` with `PROCESS_ALL_ACCESS`.
    - Handle failure (e.g., insufficient privileges—run as Administrator if needed).
5. **Locate the Target Function**
    - Use `GetModuleHandle("user32.dll")` and `GetProcAddress("MessageBoxA")` to get the function's memory address in the target process.
6. **Read and Modify Memory**

- Use `ReadProcessMemory` to save the original first byte of the function.
- Use `WriteProcessMemory` to write `0xCC` to that address.
- Sleep for 30 seconds (or until user input).
- Restore the original byte using `WriteProcessMemory`.

7. **Test Safely**

   - Launch `notepad.exe` manually first.
   - Run your tool—it should report success.
   - In Notepad, go to **Help** → **About** to trigger `MessageBoxA`. The program will crash (expected behavior for an unhandled breakpoint).
   - Confirm your tool restored the original byte afterward.

Option 4: Strings Extractor (Like Unix `strings` Command)

Goal
Build a tool that reads any binary file and prints all sequences of 4 or more consecutive printable ASCII characters—mimicking the behavior of the Unix `strings` command.

Why This Matters
Malware analysts use `strings` to find hidden URLs, passwords, or commands in compiled binaries. This is often the first step in reverse engineering unknown files.

Implementation Guide

1. **Define "Printable ASCII"**

   - Printable characters are those with ASCII values from 32 (space) to 126 (`~`).
   - Anything outside this range (e.g., null bytes, control characters) breaks a string.

2. **Open File in Binary Mode**

   - Use `std::ifstream` with `std::ios::binary`.
   - Handle file-not-found errors gracefully.

3. **Stream the File Byte-by-Byte**

   - Do **not** load the entire file into memory—use a loop with `file.get(char&)`.
   - This ensures your tool works on large files (e.g., 1GB executables).

4. **Accumulate Printable Sequences**

   - Maintain a `std::string` buffer.
   - For each byte:
     - If printable → append to buffer.
     - If not → check buffer length. If ≥4, print it and clear the buffer.

5. **Handle End-of-File**

   - After the loop ends, check if the buffer still contains a valid string (≥4 chars) and print it.

6. **Add a Command-Line Interface**
   - Accept the filename as `argv[1]`.
   - Print usage help if no argument is given.
7. **Test Extensively**
   - Run your tool on:
     - A plain text file (should print all content).
     - Your own compiled C++ program (should find strings like function names or error messages).
     - A system executable (e.g., `notepad.exe`).
   - Compare output with the real `strings` command (if available on your system).

Learning Outcome

- How C++ interacts with operating system APIs (Windows/Linux)
- The importance of error handling, input validation, and resource cleanup
- Ethical considerations in tool development
- Techniques used daily by security professionals in real-world scenarios

Deliverables

You are required to complete **all three labs** and **one mini project** from the options provided. For each, submit the following:

For Labs 1–3:

1. **Lab Report (PDF)** – One combined document containing:
   - A brief explanation of what you did in each lab
   - Screenshots showing successful execution (e.g., exploit triggering `win()`, Ghidra decompiled logic, keylogger logging with `ENABLE_LOGGING` file present)
   - Answers to reflective questions (e.g., *What vulnerability did you exploit? How would you prevent it? What ethical considerations apply?*)

For the Mini Project (Choose One):

1. **Source Code** – Well-commented C++ file(s)
2. **Build Instructions** – Clear commands or a `Makefile` for compiling on Windows (MinGW/MSVC) or Linux (g++)
3. **Screenshot** – Showing your tool running successfully with sample output
4. **One-Page Project Report (PDF)** covering:
   - Design decisions and implementation approach
   - Security considerations (e.g., input validation, error handling, privilege requirements)
   - Testing methodology and edge cases handled

    - Lessons learned about C++ and cybersecurity

Submission Method

- Compress all deliverables into a **single ZIP file** named:
  `LastName_FirstName_CyberCPP.zip`
- Upload the ZIP file to the course's **Learning Management System (LMS) - CTL** under the assignment titled **"C++ for Cybersecurity – Labs & Project"**

Note: Submissions via email or other platforms will __not__ be accepted.

Deadline
Thursday, 23 October 2024, 11:59 PM
Late submissions will incur a 10% penalty per day, up to 3 days. No submissions accepted after 26 October.