**Assignment 4: Injection Virus**

Atuhaire Ambala and Ricardo Escarcega

Prof. Isac Artzi

CST 315

June 23rd 2024

**Assignment 4: Injection Virus**

Command injection is a critical security vulnerability that allows attackers to execute arbitrary commands on the host operating system via a vulnerable application. This type of attack is made possible when an application passes unsafe, user-supplied data to a system shell, leading to unauthorized command execution with the privileges of the vulnerable application. Command injection attacks can cause severe damage, such as unauthorized data access, system manipulation, and even complete system compromise. This assignment aims to illustrate the concept of command injection by designing and implementing a simple command injection virus in C. Additionally, a detection program will be developed to scan and identify the presence of such malicious code, reinforcing the importance of input validation and secure coding practices as advocated by the Open Web Application Security Project (OWASP).

**Implementation approach**

The virus.c program aims to simulate a command injection attack. The idea is to modify a user’s command by injecting additional arguments without the user's knowledge, demonstrating how malicious software might alter commands to perform unintended actions.

**Virus Implementation:**

1. **Command Injection Simulation**:
   * The function executeCommand takes a user-provided command and appends the malicious string --rf \* to it. This simulates a common command injection attack where additional harmful options are added to a command.
   * The modified command is then printed to simulate its execution, rather than actually running it, to avoid any destructive consequences.
2. **Program Execution**:
   * The main function checks if the user has provided enough arguments. If not, it displays a usage message and exits.
   * If a command is provided, it calls executeCommand with the user’s command, which performs the injection and prints the modified command.
3. **Compilation and Execution**:
   * The program is compiled using gcc to produce an executable. The executable is run with a command to see the modified output, demonstrating the potential impact of a command injection attack.

**Virus detector Implementation**

The virus\_detector.c program scans files in a directory to detect the presence of a specific command injection pattern (--rf \*). It is designed to identify files that may have been compromised by the simulated virus from virus.c.

**Implementation:**

1. **Exception List**:
   * An array exceptions is created to list files that should not be scanned for infections. This helps prevent false positives, especially for files that contain the detection logic itself, like virus.c and virus\_detector.c.
2. **Checking for Exceptions**:
   * The is\_exception function iterates over the exception list to determine if a file should be ignored. It returns 1 if the file is in the exception list and 0 otherwise.
3. **File Scanning**:
   * The scanFile function opens a file, reads it line by line, and checks each line for the malicious pattern --rf \*. If the pattern is found, it prints a warning message indicating that the file is infected.
   * Before scanning, it calls is\_exception to ensure the file is not in the exception list.
4. **Directory Scanning**:
   * The scanDirectory function opens the specified directory and iterates through its contents. For each regular file, it constructs the full file path and calls scanFile to perform the scan.
   * This function ensures that all files in the directory are checked for infections, except those in the exception list.
5. **Program Execution**:
   * The main function ensures that the user has provided a directory to scan. If not, it displays a usage message and exits.
   * It then calls scanDirectory with the user-provided directory to perform the scan.
6. **Compilation and Execution**:
   * The program is compiled using gcc to produce an executable. The executable is run with a directory path to scan for infected files, displaying warnings for any files that match the malicious pattern.

**Flowchart**

A screenshot of a computer flowchart

Description automatically generated

**Code**

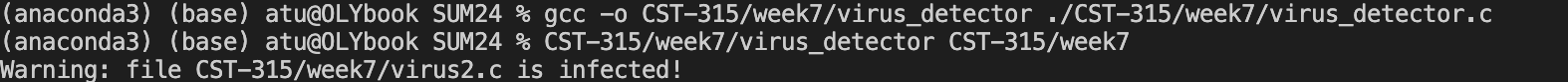
Github repo link : <https://github.com/AtuAmbala/CST-315/tree/main/Assignment4-Injection_Virus>

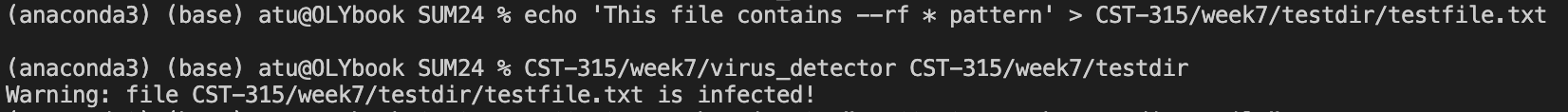
**Testing and Validation**

The purpose of testing virus.c was to ensure it correctly simulates a command injection by appending --rf \* to a user-provided command and then printing the modified command. The first test case focused on verifying the modification of a single file removal command, which is crucial as it demonstrates the basic functionality of the command injection. The second test case tested the modification of a directory removal command, which is important to show that the injection works for commands involving directories. The third test case involved modifying a wildcard removal command to ensure that the program can handle more complex and potentially more destructive commands. All these tests confirmed that virus.c performs as intended, accurately modifying and printing the user-provided commands to simulate a potential command injection.

The objective of testing virus\_detector.c was to verify its ability to accurately identify files containing the --rf \* pattern and print a warning message for each infected file. The first test case tested the program's ability to find no infections in a directory with clean files, ensuring that the detector does not produce false positives. The second test case tested the identification of a single infected file, which is essential for confirming the basic detection capability of the program. The third test case involved a directory with multiple infected files to verify that the detector lists all infected files, demonstrating its thoroughness and reliability in identifying multiple threats. These tests confirmed that virus\_detector.c effectively scans directories and identifies all files containing the malicious pattern, providing appropriate warnings without false positives.

**Execution**





**Analysis of Malware Program**

The approach taken in virus.c is straightforward but effective for illustrating the dangers of command injection attacks. The program takes a command input from the user, appends a malicious string (--rf \*), and prints the modified command. This method is a basic example of how command injection can be performed, highlighting the simplicity with which a seemingly benign command can be transformed into a highly destructive one.

**Suitability of the Approach**

1. **Simplicity and Clarity**: The simplicity of the virus.c code makes it an excellent educational tool. It clearly demonstrates how additional arguments can be appended to user commands, leading to unintended and potentially harmful actions.
2. **Demonstration of Impact**: By simulating the modification of commands, the program effectively illustrates the potential damage of command injection attacks without causing actual harm. This approach helps users understand the risks associated with running commands from untrusted sources.
3. **Focus on Command Injection**: The specific focus on command injection is suitable for demonstrating a common type of malware attack. Command injection is a prevalent and dangerous vulnerability, making this a highly relevant example.

**Suitability for Command Injection**

The specific method used in virus.c is well-suited for demonstrating command injection for several reasons:

1. **Relevance to Real-World Scenarios**: Command injection is a real-world threat that often arises in applications that construct system commands based on user input. By focusing on this type of attack, the program highlights a critical area of concern for security professionals.
2. **Educational Value**: The straightforward nature of the program makes it easy to understand, which is essential for educational purposes. It provides a clear example of how user inputs can be manipulated to perform unintended actions.
3. **Demonstration of Potential Damage**: By appending --rf \*, the program simulates a command that could cause significant damage (e.g., deleting many files). This demonstrates the severity of command injection vulnerabilities.

Creating malware, even for educational purposes, requires careful consideration of ethical and legal implications. The approach taken here, which involves simulation rather than actual execution of harmful commands, is suitable for educational settings where the goal is to raise awareness and understanding of security vulnerabilities.

**Ethical Considerations**

1. **Education and Awareness:** The primary goal of this exercise is to educate users about the risks of command injection and how to detect and mitigate such vulnerabilities.
2. **Controlled Environment**: By not actually executing the harmful commands, the program avoids causing real damage, ensuring that the educational demonstration is safe.
3. **Responsible Disclosure**: It's important to emphasize that creating and distributing malware outside of controlled educational environments is illegal and unethical. This exercise is designed to improve understanding of vulnerabilities to better defend against them.

**Analysis of Malware detection Program**

The method used in virus\_detector.c is straightforward: it searches for a specific string pattern in the contents of files within a directory. This type of pattern matching is a common and foundational technique in malware detection, suitable for identifying simple and known threats.

**Suitability of the Approach**

1. **Simplicity and Clarity**: The detection method is simple to implement and understand, making it a good starting point for educational purposes. It clearly demonstrates how basic string matching can be used to identify potential threats.
2. **Effective for Known Patterns**: For known patterns of command injection or other simple malware, this approach can be quite effective. It quickly identifies files containing the specified malicious pattern and alerts the user.
3. **Educational Value**: The program provides a clear and straightforward example of how malware detection can be performed. It highlights the importance of scanning files for malicious content and serves as an introductory tool for learning about malware detection techniques.

**Suitability for Command Injection Detection**

The specific method used in virus\_detector.c has both strengths and limitations when applied to command injection detection:

1. **Strengths**:
   * **Targeted Detection**: By looking for a specific pattern (--rf \*), the program is effective at identifying files that have been modified by the corresponding virus.c program.
   * **Quick Implementation**: This approach can be quickly implemented and deployed, providing immediate detection capabilities for known command injection patterns.
   * **Low Overhead**: The pattern matching process is computationally inexpensive, making it suitable for regular use without significant performance impact.
2. **Limitations**:
   * **Limited to Known Patterns**: This approach only detects known patterns. If the command injection uses a different pattern, or if the malicious code is obfuscated, the program will not detect it.
   * **Lack of Contextual Analysis**: The program does not analyze the context in which the pattern appears. For example, it could flag benign comments or strings containing --rf \* as malicious.
   * **Scalability**: For larger systems with many files, a simple pattern matching approach might not scale well and could result in a high number of false positives or missed detections.

**General Approach to Malware Detection**

Detecting malware, particularly command injection vulnerabilities, requires a comprehensive approach that often includes multiple techniques to ensure robustness and accuracy.

**Advanced Techniques for Malware Detection**

1. **Signature-Based Detection**: This is the approach used in virus\_detector.c, where specific patterns or signatures are matched against the content of files. While effective for known threats, it is limited against new or unknown patterns.
2. **Heuristic Analysis**: This involves looking for behaviors or code structures that are indicative of malware, even if the exact pattern is not known. This can help detect new or obfuscated threats.
3. **Behavioral Analysis**: Monitoring the behavior of programs in real-time to identify suspicious actions, such as unexpected file deletions or network communications, can provide more comprehensive protection.
4. **Machine Learning**: Advanced malware detection systems use machine learning models trained on large datasets of both benign and malicious code to identify patterns that are indicative of malware.
5. **Contextual Analysis**: Understanding the context in which a suspicious pattern appears can help reduce false positives. For example, differentiating between a command injection in executable code and the same pattern in a comment or string literal.

**Ethical and Practical Considerations**

1. **Ethics**: It is crucial to use malware detection techniques responsibly, ensuring that legitimate software is not falsely flagged as malicious. False positives can lead to disruptions and loss of trust in the detection system.
2. **Practical Implementation**: While simple pattern matching can provide a quick and easy way to detect known threats, it should be complemented with more advanced techniques to ensure comprehensive protection.
3. **User Education**: Users should be educated about the limitations of basic detection methods and the importance of using a multi-layered approach to security.