CEN429 Secure Programming Week-4

Code Hardening Techniques

Author: Dr. Ã–ÄŸr. Ãœyesi UÄŸur CORUH

Contents

List of Figures

List of Tables

1 CEN429 Secure Programming

1.1 Week-4

1.1.0.1 Code Hardening Techniques Download

- PDF[1]
- DOC[2]
- SLIDE[3]
- PPTX[4]

1.1.1 Outline

- Code Hardening Techniques
- Code Hardening for Native C/C++
- Code Hardening for Java and Interpreted Languages

1.2 Week-4: Code Hardening Techniques

1.2.0.1 1. Code Hardening Techniques for Native C/C++

In low-level languages like C and C++, various techniques are used to write secure code and make it resistant to attacks. These techniques aim to make code analysis and reverse engineering more difficult.

[1] pandoc_cen429-week-4.en_doc.pdf
[2] pandoc_cen429-week-4.en_word.docx
[3] cen429-week-4.en_slide.pdf
[4] cen429-week-4.en_slide.pptx

1.2.0.2 a) Opaque Loops Theoretical Explanation:

Opaque loops are loops that, when viewed externally, have no clear purpose. These loops make code analysis more difficult. Attackers struggle to understand the function of the loop, making the code harder to reverse-engineer.

Example Applications:

1. Adding loops created with random conditions to complicate code analysis.
2. Introducing loops that do not affect program functionality but confuse the analysis.
3. Using opaque loops to increase program runtime, misleading attackers.

1.2.0.3 b) Hiding Shared Object Symbols Theoretical Explanation:

Hiding symbols used in shared objects makes external access to these objects difficult. This process is used to prevent analysis and reverse engineering.

Example Applications:

1. Restricting symbol visibility with compiler options.
2. Only exposing necessary symbols and hiding others to improve security.
3. Concealing critical functions in shared libraries to enhance protection.

1.2.0.4 c) Obfuscation of Arithmetic Instructions Theoretical Explanation:

Arithmetic operations are fundamental to a program. Making these operations more complex makes the code harder to analyze and understand.

Example Applications:

1. Replacing simple addition operations with more complex mathematical expressions.
2. Adding unnecessary steps to arithmetic operations to maintain functionality while complicating code analysis.
3. Using bit manipulation on arithmetic operations to make them more complex.

1.2.0.5 d) Obfuscation of Function Names Theoretical Explanation:

Changing function names to random character strings makes the code harder to understand. This technique is especially useful for preventing reverse engineering.

Example Applications:

1. Changing function names to meaningless strings of characters.
2. Generating different function names for each compilation to confuse static analysis tools.
3. Randomizing critical function names to make them difficult for attackers to identify.

1.2.0.6 e) Obfuscation of Source File Names Theoretical Explanation:

Obfuscating source file names makes it difficult to understand which function or class the file belongs to.

Example Applications:

1. Changing source file names to random characters.
2. Hiding relationships between source files to obscure code structure.
3. Obfuscating file names without affecting the source code by modifying structures.

1.2.0.7 f) Obfuscation of Static Strings Theoretical Explanation:

Static strings are important information sources for attackers. Encrypting and hiding these strings increases code security.

Example Applications:

1. Encrypting static strings and decrypting them at runtime.
2. Applying random string masks to obscure the meaning of the strings.
3. Reducing static string usage by eliminating string constants.

1.2.0.8 g) Other Code Hardening Techniques

1. **Opaque Boolean Variables:** Making conditional statements more complex.
2. **Function Boolean Return Codes:** Making function return values more complex.
3. **Obfuscation of Function Parameters:** Hiding function parameters.
4. **Bogus Function Parameters & Operations:** Adding meaningless parameters and operations to complicate analysis.
5. **Control Flow Flattening:** Flattening control flow to make it unpredictable.
6. **Randomized Exit Points:** Randomizing exit points to reduce code predictability.
7. **Logging Disabled on Release:** Disabling logging in the final release version.

1.2.1 2. Code Hardening Techniques for Java and Interpreted Languages

In Java and other interpreted languages, code hardening techniques are used to reduce security vulnerabilities and complicate reverse engineering efforts.

1.2.1.1 a) Code Obfuscation and Shrink Protection with Proguard

Theoretical Explanation: Proguard shrinks, optimizes, and obfuscates Java code, making it harder to analyze.

Example Applications:

1. Using a Proguard configuration file to shrink and optimize the code.
2. Testing obfuscated code and resolving errors.
3. Analyzing Proguard reports to identify which elements have been obfuscated.

1.2.1.2 b) Separated Fingerprint Storage for Device Binding

Theoretical Explanation: This technique uses a device's unique properties to ensure that the application only runs on specific devices.

Example Applications:

1. Encrypting the device fingerprint and storing it securely.
2. Using fingerprint verification to ensure the application runs on the designated device.
3. Protecting fingerprint data from attacks.

1.2.1.3 c) Native Library JNI API Obfuscation

Theoretical Explanation: Obfuscating native libraries called through the Java Native Interface (JNI) makes reverse engineering more difficult.

Example Applications:

1. Randomizing JNI function names.
2. Hiding JNI parameters to complicate understanding.
3. Using error management in JNI to prevent attackers from analyzing errors.

1.2.1.4 d) Obfuscation of Static Strings

Theoretical Explanation: Static strings contain important information that attackers can use during reverse engineering. Obfuscating these strings increases security.

Example Applications:

1. Encrypting static strings and decrypting them at runtime.
2. Obfuscating strings to obscure their meaning.
3. Using random string generation and manipulation techniques to enhance security.

1.3 Summary of the Week and Next Week

1.3.1 This Week:

- **Code Hardening Techniques (C/C++ and Java)**
- **Obfuscation Techniques and Applications**

1.3.2 Next Week:

- **Attack Trees and Security Models**
- **Attack Methods and Secure Communication**

$$End - Of - Week - 4$$