CEN429 Secure Programming Week-5

RASP Techniques for Native C/C++

Author: Dr. UÄŸur CORUH

Contents

List of Figures

List of Tables

1 CEN429 Secure Programming

1.1 Week-5

1.1.0.1 RASP Techniques for Native C/C++ Download

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

1.1.1 Outline

- What is RASP (Runtime Application Self-Protection)?
- RASP Techniques for Native C/C++
- Caller APK Hash Verification
- Root Detection and LD Preload Protection

1.2 Week-5: RASP (Runtime Application Self-Protection) for Native C/C++

Runtime Application Self-Protection (RASP) is a security approach that allows applications to protect themselves during runtime. In Native C/C++ applications, various security checks can be applied using RASP. In this session, RASP techniques will be explained in detail, along with practical examples.

1.2.0.1 1. Runtime CodeBlock Checksum Verification **Theoretical Explanation:** During runtime, the hash or checksum values of specific code blocks are verified to detect if the code has been tampered with. This method protects against code manipulation and malicious interventions.

Practical Examples:

[1] pandoc_cen429-week-5.pdf
[2] pandoc_cen429-week-5.docx
[3] cen429-week-5.pdf
[4] cen429-week-5.pptx

1. Calculate the checksum of any code block and compare it during execution.
2. Terminate the program or produce incorrect results when tampering is detected.
3. Protect important functions and critical code parts with checksum verification.

1.2.0.2 2. Caller APK Hash and Signature Verification Theoretical Explanation:

By verifying the hash and signature of APK files, only trusted and signed APKs are allowed to execute the application. This prevents modified or fake APKs from running the application.

Practical Examples:

1. Verify the hash value of the APK file during runtime.
2. Check the signature information of the APK, allowing only original signed APKs to run.
3. Store hash and signature values for dynamic verification processes.

1.2.0.3 3. Rooted Device Detection Theoretical Explanation:

Devices with root privileges pose security risks. Detecting rooted devices can prevent the application from running on such devices.

Root Detection Methods:

1. **/dev/kmem File:** Check if this file exists on the system. If present, syscall table may be hooked, indicating the device has root access.
2. **/proc/kallsyms File:** Verify if sys_call_table and compat_sys_call_table addresses are empty.
3. **/default.prop and /system/build.prop Files:** If these files are readable, the device may be rooted.
4. **Other Root Detection Methods:**
   - The presence of Superuser.apk.
   - Connecting to port 27047 to search for a running Frida server.

Practical Examples:

1. Detect root by checking for the presence of the specified files.
2. Test for tools like Frida and detect them.
3. Prevent the application from running on rooted devices.

1.2.0.4 4. Advanced LD Preload Attack Detection Theoretical Explanation:

LD_PRELOAD is a method used to manipulate dynamically loaded libraries. This technique is a vector used by malware. Detecting LD_PRELOAD attacks improves the security of the application.

Practical Examples:

1. Check LD_PRELOAD environment variables during runtime.
2. Use specific algorithms to detect LD_PRELOAD attacks.
3. Protect the application when such attacks are detected.

1.2.0.5 5. GDB, Tracers, and Emulator Detection Theoretical Explanation:

Detecting debugging tools like GDB, tracers, and emulators prevents attackers from analyzing and modifying the application.

Practical Examples:

1. Detect if the GDB environment is present and prevent the application from running in this environment.
2. Detect and block the usage of tracers such as ltrace or strace.
3. Make the application close or behave differently when running in an emulator environment.

1.2.0.6 6. Debugger Attachment Check Theoretical Explanation:

By detecting whether the application is attached to a debugger, attackers can be prevented from analyzing the application.

Practical Examples:

1. Add code snippets that detect debugger attachment.
2. Stop the application or make it behave differently when a debugger is detected.
3. Use anti-debugging techniques to enhance the application's security.

1.2.0.7 7. Memory Protection Theoretical Explanation:

Memory protection techniques control memory access. These techniques protect against manipulations in memory. Clang's SafeStack feature makes memory access traceable.

Practical Examples:

1. Activate memory protection mechanisms using SafeStack.
2. Detect any manipulation in memory.
3. Enhance application security with memory protection mechanisms.

1.2.0.8 8. Other RASP Techniques

1. **LD Preload Custom Environment Detection:** Detect customized LD_PRELOAD environment variables.
2. **Tamper Device Detection:** Check if the device running the application has been tampered with.
3. **Control Flow Counter Checking:** Use counters to monitor control flow and detect code manipulation.
4. **Device Binding:** Ensure the application runs only on a specific device.
5. **Version Binding:** Ensure the application runs only on a specific version.

$$End - Of - Week - 5$$