DD2497 - Project Specification
Group 2

Christopher Gauffin
Joseph Johansson
Jesper Lagnelöv
Kevin Harrison

November 18, 2022

1 Vulnerabilities countered by project

- Buffer overflows - Mainly we want to investigate how writing data outside of the bounds of a memory buffer to corrupt memory adjacent to that buffer can be used to manipulate the stack and heap, what the consequences of that might be and what can be done to detect it.
- Memory corruption - We look at corrupting memory of the stack and heap. *An attacker may, for example, seek to modify a function pointer on the stack or heap to hijack program control flow.* [1]
- Code injections and Code-reuse - As a result of a corrupted memory, an attacker may change the destination of indirect jumps to point in order to execute newly injected malicious code or re-use code of the program in unintended ways.

2 Minimal requirements of project

We want to implement as much of the paper "Cryptographically Enforced Control Flow Integrity" [2] as possible.

Our solution will not stop an attacker from overwriting memory but will make it difficult to use the corrupted memory in a useful way, similar to ASLR. This is accomplished by, using a key not access-able to the program, encrypting (or providing a MAC of the addresses) of indirect jumps. Then, at run-time, when indirect jumps are to be resolved they are decrypted (or its MAC verified).

The minimal requirements of the project are then to provide a simplified version of this process with the following goals:

- Create a susceptible program that is susceptible to an buffer overflow attack.
- Demonstrate how a C program can be used to perform memory corruption and consequently a code injection and/or code-reuse attack.
- Encrypt the return address for every function call with AES at runtime and push it to the stack.
- We implement a Control Flow Manager (CFM) in the form of a service which can be used to handle system calls.
  - Determine what data structure the CFM should use in order to store entries as well as what data should be stored in between the function calls.
  - Create the CFM service in the form of a driver or service that handles requests and makes changes to the data structure.
  - The running program should be able to verify that its return address has not been compromised. This is done by making a system call to the CFM which then verifies the AES hash.
  - The CFM should be able to handle requests from the different processes and respond with an answer that fits the context of that specific process.
- Create all of the required system calls which will communicate with the CFM.
- Modify the source code of the susceptible program so that system calls are being made before function calls and function returns. We want to use this as a temporary solution to produce a Minimum Viable Product. (As soon as this works we want to look in to developing a LLVM pass or modifying binaries)
- Compile the susceptible program with the modified source code with the added system calls and run it with the buffer overflow exploit simultaneously as we run the CFM and prove that it is working in real-time.

3 Optional requirements of project

- Use the compiler or do manipulation of binaries and/or object files to autonomously insert cryptographic hashes of all possible return addresses of a certain function.

- Moniter more types of indirect jumps than just function calls and returns.
- Combat replay attacks in the case that an encrypted address is leaked (by a buffer overread for example).
- Use AES instructions without the need to make any system calls. May require implementation in Qemu.
- If more advanced functionality is required, for example communicating with the virtual file system we will investigate what IPC calls to intercept and modify.
- Show that stack canaries can be used but also demonstrate how they can be circumvented by an example.
- Implement minimal ASLR and show how it can be bypassed.

References

[1] Per Larsen et al. "SoK: Automated software diversity". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 276–291.

[2] Ali Jose Mashtizadeh et al. *Cryptographically Enforced Control Flow Integrity*. 2014. DOI: `10.48550/ARXIV.1408.1451`. URL: `https://arxiv.org/abs/1408.1451`.