Course: DD2497
Date: 2023-03-02

Project Proposal

Project title: Code integrity and non-executable data on s3k

Author: Zacharias Terdin (zacte@kth.se)
Supervisor: Henrik Karlsson (henrik10@kth.se)
Examiner: Roberto Guanciale (robertog@kth.se)

Background

Modern software systems are filled with bugs and security flaws since they are a complex combination of many components where performance is often the focus and security is an afterthought [1]. The most privileged software, the operating system, can enforce security and hopefully prevent vulnerabilities in application and user software from resulting in a system wide compromise. This presumes that the OS itself is free from bugs which can be proven through formal methods. *S3k* (Simple Secure Separation Kernel [2]) is one such project which aims to design and implement a bug-free kernel through formal methods.

S3k is a separation kernel designed for standard RISC-V 64 bit (RV64IMA) with MPU (memory protection unit). The kernel is designed with the following security implementations in mind [3]:

- memory protection and management
- secure time management for real-time systems
- secure inter-process communication
- process monitoring.

Almost all software contains bugs and one of the most, if not the most, common type of software bug withing computer security is related to memory vulnerabilities [4]. Memory corruption, code corruption, control-flow hijacking are examples of attack which would allow an attacker to influence a system partially or completely. Fortunately, there exists protective mechanisms to counter these well-known attacks.

Problem formulation

This project will be covering the prevention of code injection and execution of arbitrary code on s3k. The problem to be solved consists of two parts, a malicious actor can write and execute arbitrary code if:

1. a program can write and execute code from the memory intended for program data.
2. a program can write code into the section of memory intended for instructions, i.e., the program can rewrite its own code.

Solution

This project aims to solve the problem by implementing the *code integrity policy* and *non-executable data policy*.

Code integrity enforces that program code cannot be writable, only readable and executable. Non-executable data enforces that memory intended for program data, such as stack and heap, cannot be executable, only readable and writable. When the policies are combined, they result in the *write xor execute policy* (also known as the *executable space protection policy* [5]), stating that a range of memory can either be writable or executable, but not both [4].

The implementation of the code integrity policy will not be as strict as described in the previous paragraph. Software updates is a legitimate reason to rewrite a programs code. This action will be allowed after a method of authentication. The authentication will check that the code comes from a trusted source and that it has not been tampered with.

Intended implementation (High-level overview)

The write xor execute policy will be enforced by a monitor process. It will be the initial user process which owns all capabilities to all user resources. All subsequent user processes that are spawned will be given capabilities to memory ranges and time by the monitor. Memory ranges for code will only be readable and executable and memory ranges for data will only be readable and writable. The capabilities will be implemented with RISC-V PMP.

The monitor will handle request to alter memory capabilities: acquire new capabilities for memory ranges, return capabilities not in use, create a shared data memory range between two or more processes (IPC). The monitor will also be responsible to handle exceptions when memory capabilities are violated.

The monitor will also be responsible for authenticating all code that it will be giving capabilities to. Code authentication will be implemented through code signatures and encryption with symmetric keys. **This is considered an optional task.**

Tasks

The tasks are categorized as main tasks and sub tasks. The order of the tasks is the preferred order of completion. Multiple sub tasks from different main tasks can be worked on in parallel since they are not necessarily a prerequisite to each other. This list should be considered dynamic, since it most probably will be modified during the course of the project.

1. **Analyse** the current state of s3k.

    a. Determine how the initializing process is booted and what the memory layout is.

    b. Determine which syscalls are relevant for the monitor process.

    c. Determine which capabilities will be relevant for the monitor process to handle.

d. Determine how exceptions exceptions are handled.

e. Determine how time scheduling and memory allocation can be handled in the monitor process.

2. **Design** the monitor process. a. Explore different implementation of the monitor process to ensure a the most seamless incorporation with s3k.

   b. Determine which capabilities will be needed and how they will be derived.

   c. Design the interface for interacting with the monitor as well as how the monitor interacts with the rest of the kernel.

   d. Identify edge cases and undefined behavior and decide how to prevent it and provide a reason for the chosen method.

   e. Design an exception handler and determine how exceptions should be resolved.

   f. Design a authentication method for verifying code integrity.

3. **Implement** the monitor process.

   a. Create a initializing process.

   b. Assign the process maximal capabilities to the memory range intended for user programs.

   c. Implement functionality to derive memory capabilities and revoke them.

   d. Implement interface to handle requests to alter memory capabilities.

   e. Create dummy processes which interact with the monitor.

   f. Implement handler for exceptions.

   g. Implement authentication method for verifying code integrity.

4. **Test** the functionality of the monitor process.

   a. Create a demonstration for all the functionality of the monitor.

5. **Deploy** the version 1.0 of the monitor process.

   a. Create a repository with instructions to build the monitor process.

   b. Write documentation for the project the monitor process.

Scope and presumptions

Detection of code injection is beyond the scope of this project. Malicious code can be written to the writable section of memory intended for data, either by intentional input methods or

malicious activity, e.g. buffer overflow attack. This project will not *detect* this data integrity violation, but it will *prevent* the code from ever being executed.

Countermeasures against code-reuse attacks is beyond the scope of this project. Code-reuse attacks are a method which can sidestep the protection provided by the write xor execute policy. ROP (return oriented programming) is one example of a code-reuse attack. It utilizes already existing code snippets loaded into memory from shared libraries and the modification of return pointers to hijack the control flow of a program and chain together code to achieve arbitrary code execution. This is possible since the accumulated code within standard libraries Turing complete [4].

References

Index	Reference
[1]	KTH-STEP Group, "Separation Kernel," n.d. [Online]. Available: https://kth-step.github.io/projects/separation-kernel/. [Accessed 28 February 2023].
[2]	KTH-STEP Group, "s3k," 2023. [Online]. Available: https://github.com/kth-step/s3k. [Accessed 28 February 2023].
[3]	H. Karlsson, "Provable Security," 27 October 2022. [Online]. Available: https://kth-step.github.io/assets/projects/separation-kernel/2022-10-27-cdis-retreat.pdf. [Accessed 28 February 2023].
[4]	L. Szekeres, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2013.
[5]	H. Chfouka, H. Nemati, R. Guanciale, M. Dam and P. Ekdahl, "Trustworthy Prevention of Code Injection in Linux on Embedded Devices," in *Computer Security -- ESORICS 2015*, G. Pernul, P. Y. A. Ryan and E. Weippl, Eds., Cham, Springer International Publishing, 2015, pp. 90-107.