Final report - Group 1

Joakim Loxdal, Douglas Fischer,
Oussama Kaddami, Abdelmoujib Megzari

November 12th 2021

1 Introduction

Malware is harmful software, designed to cause some form of damage to a system without the consent of the system user. Some malware infects benign binaries to hide themselves from discovery.

2 Targeted Vulnerabilities

In our project we aim to implement a detection system into the OS level by storing signatures (hashes) of accepted binaries in a database and make sure that the signatures still hold before executing the binaries. We assume that the OS kernel is trusted.

Signing user binaries in the system and doing signature verification at loading time will prevent execution of modified binaries.

Our strategy is to create hashes of known binaries and store these in a table/database (that is assumed secure), and verify that the binary has not been altered when they are executed. This can be used to secure certain important binaries, or all binaries that are known to be used in the system.

3 Minimal Requirements

1. Hash user binaries for which integrity is sought.
2. Implement a storage for known trusted hashes
3. Check signature on demand at loading time of user binary code
4. Alert the user about the potential malware and stop the execution of the binary

4 Optional Requirements

1. Use our technique on both user level binaries and kernel modules
2. Optimize the overhead due of the signature check
3. Add an integrity check of the disk once in a while
4. Secure the storage of hashes

5 Final solution

Our final solution implements:

1. A code signature checker service (CSC) that compares signaturers of ELF memory pages with a database of signatures.
2. A call to the CSC when a pagefault happens in the VM (Virtual Memory).
3. A syscall from the CSC to the PM (Process Manager) that replies with the name of a process.
4. A syscall from the CSC to the VFS (Virtual File System) that replies with a magic grant id giving access to the page with pagefault.
5. Benchmarking?
6. Generating signatures from ELF files (outside of minix)

6 Implementation Details

6.1 Our solution (Figure 1)

The CSC service can be found in minix/servers/csc. It has the files and main.c, signcheck.c as well as some header files. The service has a syscall implemented, CSC_CODECHECK. When a different process/server (in our case, the VM from mem_file.c) makes the syscall (with an endpoint and a virtual memory address), CSC will catch this message and do the following:

In the codecheck function in the CSC, our server has to ask the pm (process manager) for the name of the process being signature checked. To do this, an IPC call is implemented in the PM, which replies with the name of the process. To perform the signature check we need to fetch the memory page of the running process in the CSC. To do this, we implemented a syscall in the VFS. This syscall creates a magic grant and replies to the CSC with the grant id so that the memory can be read in the CSC.

After this, we have implemented a simple xor signature calculation that in the end creates a signature.

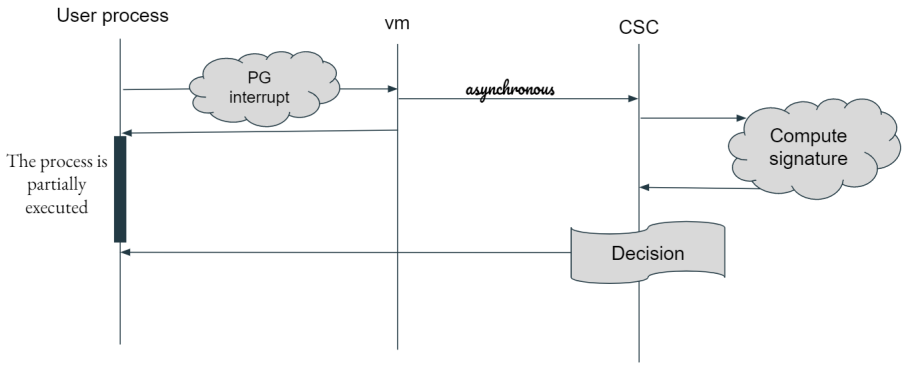

Figure 1: Diagram of our implementation

If the binary is supposed to be signature checked and the signature is in the whitelist of accepted signatures, the binary will be allowed to run as normal.

If the binary is supposed to be signature checked and its signature is not in the whitelist, the process will be killed.

In this solution, the binary is allowed to run for a moment before the CSC decides if the binary should be allowed to run.

6.2 Requirements fulfilled

All minimal requirements are partially or fully fulfilled.

We hash (non cryptographically) a couple of binaries that we want to check the signature for. We have a storage for the signatures, but it is just an array for now. We check the signatures on demand. We alert the user if the signature has changed on execution, and kill the binary process.

The optional requirements are not fulfilled at all.

7 Design choices

We chose to verify the signature page by page whenever a page of the program is loaded in memory so that our verification can be more secure. That way it's impossible for an attacker to change the code between the verification of the program and the loading of the program into memory. The other choice was to verify the signature once in a while for the whole ELF file but we judged that it wasn't secure enough.

We also chose to let the program be executed while we are verifying the signature so that we don't slow the executed program performance. However, this raises a security issue since some malicious code can be executed before the signature is checked and the program is killed. Hence we tried to implement

another version that made the program wait while the signature is verified. We thought we could join the two methods by specifying in addition to the programs that should be checked whether they should wait until the verification is complete or not. We began the implementation of the synchronous verification but we couldn't deliver it in time due to a deadlock that we failed to debug.

8 Security Analysis of our Solution

Our solution manages to secure against malicious altering of binaries that have been signed previously, by checking the signatures each time a memory page is loaded.

However, their are some security concerns. Our solution defaults to allowing a binary to be executed. If a user downloads a new binary with a different name from any signed binary, it will be allowed to run without any restrictions. Same thing if a binary loads a page that we have not signed, it will be allowed by default. So our solution basically only applies to known binaries and known memory pages that have been altered after signing takes place. A better opinion would be to default to blocking any binary from executing, and using a whitelist instead with binaries and the signatures of the binaries.

In our solution the binaries will have a small time window before they are killed (our experiments show that this time window is probably very small, but still). Instead we should block the executable from running before the signature check is done and we can decide if it should be killed or allowed to execute.

Our signature hashing function is not secure. This function should be implemented with a trusted cryptographic hash function, e.g. SHA-2.

Our solution does not secure the storage of the signature since that was outside the scope of the project. In the case of someone being able to alter files in your file system, running the signature checker as a hypervisor for the operating system is a more secure solution, as done in [1].

9 Benchmarking

A benchmarking tool was built, basically working like the time function in bash. It is a binary that takes another binary as argument, calls it 1000 times and calculates the average time of execution using both the clock-function and the gettimeofday-function to give to different results.

The results With signature checking:

Running binary demo01 (correctly signed):

10 Future Work

In our current solution, binaries may be able to run for a small amount of time while the CSC service verifies the hash of the binary. In the second version of our solution we wanted to kill a malicious binary before it has started executing. We

worked on this solution, but got stuck in the development because of a deadlock that we were not able to solve.

There is also work that is still to do on the signing function since we do consider that the one used right now isn't secure enough. The signature function should be secure and fast at the same time.

Securing the signatures storing is also one of the future perspectives of the project.

Another future work could be to default to blocking all binaries and only allow signed binaries to run. In our solution we only kill binaries that are signed and have been altered.

11 Individual contributions Joakim Loxdal

11.1 Setting up the CSC

Me and Douglas set up the CSC service foundation, without any implementation details. We tried to look at how other services were implemented and copied this to our service.

11.2 Adding demo binaries

To test our implementation we needed at least two demo binaries. This was no difficult task, but I added demo binaries to be able to test our solution.

11.3 Debugging the first deadlock

In our solution we first had synchronous messaging, which lead to a deadlock between the VFS, PM and the CSC. This was solved after conversing with Roberto and lead to our solution, where the messaging is asynchronous. It brought up some other issues (mainly security related), which were brought up in the security analysis section of the report.

11.4 The benchmarking tool

Me and Douglas worked on several benchmarking solutions. One idea was a benchmarking server, to make the benchmarking modular. This was tested, but because of overhead and missing libraries in the server we had trouble with this solution and gave up on it. The work can be seen on the branch called benchmarking.

Another idea was to have a benchmarking program running as a binary, similar to the time binary that can be run from bash. We did work on this, and in the end had a solution that could perform benchmarking on a program. However, as the time differed a lot between different runs, and even when taking the average, the benchmarking did not result in any interesting findings.

It would have been more interesting to perform the benchmarking in the CSC server, seeing which part took most time. Attempts were made at this, but in the there were libraries missing that was needed for it to work. We could have reimplemented these libraries, but because of lack of time and belief it did not happen.

11.5 Writing the report

I have contributed a lot to the report, especially the parts about security analysis and implementation details.

12 Individual contribution Abdelmoujib Megzari

12.1 Caling the CSC from the VM when a page fault is detected

This was a common part to which most members participated in peer programming. In this part we tried to explore the execution steps of a program in detail and find the best moment to call the server to check the signatures. We got a good help from Roberto on this task when he explained in details the steps of a program execution and helped us understand more key parts of minix code.I also participated to understanding and sending the first messages between different servers.

12.2 Code signing and comparing the signature

In this part I created the program responsible for signing the ELF file outside minix which required a deeper understanding of the ELF file structure and understanding where to find the necessary elements such as the program size and the used virtual address.The program also generated a file called "signatures.h" that would help access these signatures inside minix.

In the other hand inside signcheck, I managed to sign the page that was granted to the CSC and compare it to the stored signatures in case the programmed was to be checked. In case the signatures are different I called the syscall that was furnished to me by my colleagues and that is responsible.

I want to point out that even though this part was appointed to me I got help from my team to debug and to address certain issues that we have faced in understanding the ELF file structure.

References

[1] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries.," in *USENIX Security Symposium*, vol. 22, p. 70, 2008.