jamk.fi

Software Exploitation

Assignment 4 Essay

Mikael Romanov

Assignment
Maaliskuu 2018
Tieto- ja viestintätekniikka
Kyberturvallisuus

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

1 Table of Contents

1 Introduction

This report is an essay about 5 topics:

- Executable space protection
- Address space
- Randomization
- Runtime checks
- Static code analysis

The essay will discuss about why countermeasures exist, what kind of vulnerabilities they target, at what level they are implemented, assumption of the countermeasures, how effective they are and what kind of conclusions I have about the topics.

2 Executable space protection

Executable space protection is a marking in the CPU for non-executable memory regions, which are defined with NX-bit in the hardware (CPU). If e.g. program runs and try to execute code in the NX regions, it will cause an error. The NX bit segregates certain areas of the memory to storage data or the processors instructions and is used in processors for security reasons. It is used to prevent malicious programs from taking over another program by inserting code into the programs storage area and running it within the section. This type of attack is known as buffer overflow, which has been one of the main focusses of this course. If an operating system marks e.g. some regions of the memory as non-executable, it could be possible to prevent those stack and heap memory areas from being exploited via buffer overflow since the memory areas cannot be accessed. Since certain buffer overflow attacks rely on the stack and heap it would mean that the exploit isn't possible if the memory area in need is non-executable or non-writable. These types

of attacks are such as injecting code and executing it, great example of the attacks are Blaster and Sasser worms.

The executable space protection can be also implemented on OS level, where the OS itself defines its own policy. Even if the NX-bit doesn't exist in a processor it can be emulated via OS or some other way (exec shield). Linux has supported NX-bit from 2004 (kernel 2.6.8) and Windows from XP SP2. Before the NX-bit was supported by the OS, there were patches like exec shield and PAX for Linux kernel for emulating the NX bit on x86 processors that didn't natively have NX-bit in hardware. Exec shield did the same thing as would the NX-bit in hardware by suppressing buffer overflows that rely on overwriting data or inserting code to those regions. The emulated NX-bit causes measurable overhead in compared to the hardware level implantation and is thus slower.

The NX-bit seems like a cheap security patch that solves part of the problem. Even thought the NX-bit is supposed to deny access to certain memory segments e.g. program that contains executable code in a DLL file, that is loaded by the program. If the DLL file contains nested functions or advertises that it needs a stack to be executable, then the stacks on the program will be marked executable. So an added library, plugin or other extensions can deactivate the NX protection by using GCC own features. The NX protection isn't good since it occurs after a buffer overflow has been done, this should be implemented in a way that doesn't allow the buffer overflow in the first place. Lastly the NX protection makes only simple buffer overflows hard to execute, but doesn't stop a good wizard.

3 Address space

The address space is a valid range in memory that is allocated for a process or a program. The memory addressed for the program can be virtual or physical and it is used for executing instructions and storing data. Each program or process are given a flat32 or flat64 address space. Virtual addresses start usually from low addresses and can extend to the highest addresses that the PC instruction set architecture and OS

allows. Example in 32-bit (4bytes 0 to 4294967295) and in 64-bit (8bytes 0 to 2^64). Each program/process flat address space is unique and is unrelated from another process that has same memory address in its address space. Two or more processes can have different data in their address spaces. If e.g. two processes want to share their address space, they can have different data at the same address spaces and share it and this is called thread.

4 Randomization

In this chapter the randomization means Address Space Layout Randomization (ASLR). The ASLR is used to prevent shellcode injections being successful by randomizing location where the executables are loaded into memory. Every time the program is run, all the components such as heap, stack and libraries are moved to different virtual memory addresses. This means that the memory addresses can't be learnt by trial and error and when an exploit occurs to an incorrect address space, the program will crash and alert the system. The ASLR was created same time as PAX to prevent stack & heap overflows and format string attacks. The ASLR has been considered as a protection against remote attacks, because some implementations change the position of libraries on system boot. A x86 system has smaller address space that limits the space of the randomization and it is easy to create a brute force attack on it. The ASLR is quite effective especially on 64-bit hardware, because the possible outcomes are numerous for the randomization, but it can be jumped over by attacking branch predictors (BTB) and getting bypass the ASLR.

5 Runtime checks

Runtime checking is a way to automatically detect runtime errors that cause the program crash or not work properly. Runtime checking is a software verification method that analyzes a program when it is executed by a debugging feature that can catch errors like stack pointer corruption, overruns of local arrays, stack corruption, depencies on unitialized local variables and loss of data on an assignment to a shorter variable, race conditions and such. The idea of runtime checking is to use it in

the earlier stages of development cycle since the errors will stack if they are not handled since it is a huge task to later on fix all the errors.

The problem with run time checking is that it cannot handle text or data areas that are larger than 8mb on hardware that is not based on UltraSPARC.

6 Static code analysis

Static code analysis is reviewing and analyzing the source code without executing it and is used to find bugs and for ensuring quality, safety and security (white box testing). The static analysis tools are usually used by integrating them in the build process (compiler). The analysis tools usually help to maintain good code quality by producing warning & error outputs that tell the developer is compiling the program. These warnings can be ignored although they shouldn't. These types of compilers are e.g. GCC or G++. The main purpose of the analysis is to test, maintenance and bug detection, although such things as race conditions are merely impossible to find with static analysis tools. The problem with the static analysis is that, you can only explore one possible execution of the program at a time. Which means that it is a rinse and repeat process. The static analysis is good for diagnosing various bugs like overflows, zero division, memory and pointer errors and runtime errors. The analysis can also warn about false positives, since the static analysis doesn't always match the developers intend.

The static code analysis is a great way to make quality code, although it isn't the all mighty holy grail, it is useful for finding bugs and patching security isssues.

7 Makefile

7.1 fno-stack-protector

Fno stack protector is off by default, although some Linux distros have the GCC turned it on by default. The problem whit the stack protector is that it doesn't allow to compile anything that is not linked against the standard userspace libraries. The

makefile needs to disable the fno stack protector when compiling the program. When the fno stack protector is disabled it allows all kinds of attacks such as buffer overflow.

7.2 Wno-deprecated

Wno deprecated is a diagnostic message that reports constructions that are risky or suggest possible errors. If the deprecated declarations are turned off, the compiling wouldn't report of risky functions such as gets() etc.

8 Sources

http://www.gutenberg.us/articles/executable_space_protection

https://news.ycombinator.com/item?id=16007703

https://en.wikipedia.org/wiki/Executable_space_protection

https://security.stackexchange.com/questions/47807/nx-bit-does-it-protect-the-stack/47825#47825

https://en.wikipedia.org/wiki/Exec_Shield

http://searchstorage.techtarget.com/definition/address-space

https://en.wikipedia.org/wiki/Static_program_analysis

https://www.owasp.org/index.php/Static_Code_Analysis

https://stackoverflow.com/questions/49716/what-is-static-code-analysis

https://docs.oracle.com/cd/E19422-01/819-3683/RunTCheck.html

http://www.cs.ucr.edu/~nael/pubs/micro16.pdf

https://notes.shichao.io/lkd/ch15/#address-spaces