jamk.fi

Assignment 4 Essay

Software Exploitation

Joni Korpihalkola
K1625

Essay
03/2018
Information and communication technology
ICT-field

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

Contents

Figures

1 Introduction

In the previous assignments we have been trying to exploit C programs by overflowing variables that take user input and by injecting shellcode. This has been possible, because a lot of countermeasures against our exploits have been disabled in the Makefile provided with the assignments. In this essay I'm going to look up some information about the protections we have been disabling in the Makefile and some other practices that make for more secure code.

2 Countermeasures

The most common vulnerabilities in C are related to buffer overflows, so a lot of countermeasures aim to prevent code being executed from unintended locations or checking the code for functions that are considered vulnerable.

2.1 Replacing vulnerable functions

There's some secure code guidelines that should be followed in C. The gets() function doesn't check for length of the buffer and so results in a buffer overflow vulnerability. It is recommended to replace the function with fgets() and dynamically allocating memory for the length of the user's input.

String manipulation functions in C are also vulnerable, because they too do not check for buffer's length. It is recommended to for example replace strcpy() and strcat() and functions with strncpy() and strncat(). The latter functions require buffer lengths when using them.

If the program uses functions like printf or sprintf, it can be vulnerable to memory overwriting by format string attacks. So, it is always recommended to hardcore strings that are used in these functions. (CERN. Common vulnerabilities guide for C programmers.)

2.2 Stack protector

If a programmer initializes a variable in C that reads user input and uses vulnerable functions such as strcpy or printf, there's a possibility for a buffer overflow attack. In the attack, an attacker can craft an input that is longer than the initialized variable, which causes it to overwrite past the variable in the stack and towards the return address of the function.

To prevent this from happening, the stack protector assigns a variable to each function's stack area, which resides higher in the stack than the local variables. Before the function is executed, the program assigns a value for the function and after the function execution is completed, the value is checked again. If the variable's value has changed, the program is stopped.

The negative aspect to stack protector is that it requires more space in stack to store the variables assigned to functions. Stack protector can be used in different levels, where for example "f-stack-protector-all" assigns the variable to every function in the program and "f-stack-protector-explicit" only assigns the variable to functions where it is specified to be used. (Martin. Hardening C/C++ Programs Part I – Stack Protector.)

2.3 Executable space protection

Executable space protection is another countermeasure to possible overflow attacks, it was implemented to the Linux kernel in 2004. The idea is that when the program is loaded into memory, only the spaces where code is executed is given execute permissions. This is done by looking at the program's ELF headers, so execution permissions are given to the following sections:

- .init and .fini
- .plt and .plt.got
- .text

While most modern processors and operating systems take care of this automatically, one thing that may require oversight is the execution permissions for libraries. If a function in libc is used, for example “printf()”, an attacker can direct it to use a different function from the libc library to malicious intent. A programmer can check the ELF header for GNU_STACK flags to make sure that it doesn’t have execution permissions. (Martin. Hardening C/C++ Programs Part II – Executable-Space Protection and ASLR.)

2.4 Address space layout randomization

Address space layout randomization (ASLR) is a different technique to prevent shellcode being run by an attacker. The idea is to randomize the memory locations of the stack, so if an attacker is able to inject shellcode into the program, the attacker won’t be able to tell the program where to return to execute it.

ASLR has been implemented to Linux kernel since 2012, and it is determined by the “randomize_va_space” value. This is why in this course’s assignments it was necessary to disable this feature to make our shellcodes work.

There was a problem with Windows’ ASLR implementation in Windows 8 and 10, where some applications memory addresses weren’t randomized at all. The Microsoft Equation Editor that was used in the Microsoft Office 2016 software package was linked without a “DYNAMICBASE” flag, which caused it to not be loaded to a randomized location. This enabled an attack, where the attacker could craft an Office document that executed code with the privileges of the user who opened the document. (Will Dormann. Microsoft Office Equation Editor stack buffer overflow.)

2.5 Data Execution Prevention

From Windows XP and onward, Microsoft has implemented Data Execution Prevention (DEP) feature in their operating systems. DEP prevents execution of malicious code from memory with hardware-enforced DEP and software-enforced DEP, similarly to executable space protection in Linux kernel. Hardware-enforced DEP uses either AMD’s

no-execute page-protector feature or Intel's The Execute Disable Bit to raise an exception if a program tries to execute code from memory that is marked to not be executed. The software-enforced DEP helps protect system files from attacks that try to abuse Windows' exception handling system, even if the processor isn't compatible with DEP. (Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003.)

Windows users can choose to use DEP on system files or on all files with exceptions in advanced system settings. (Figure 1)

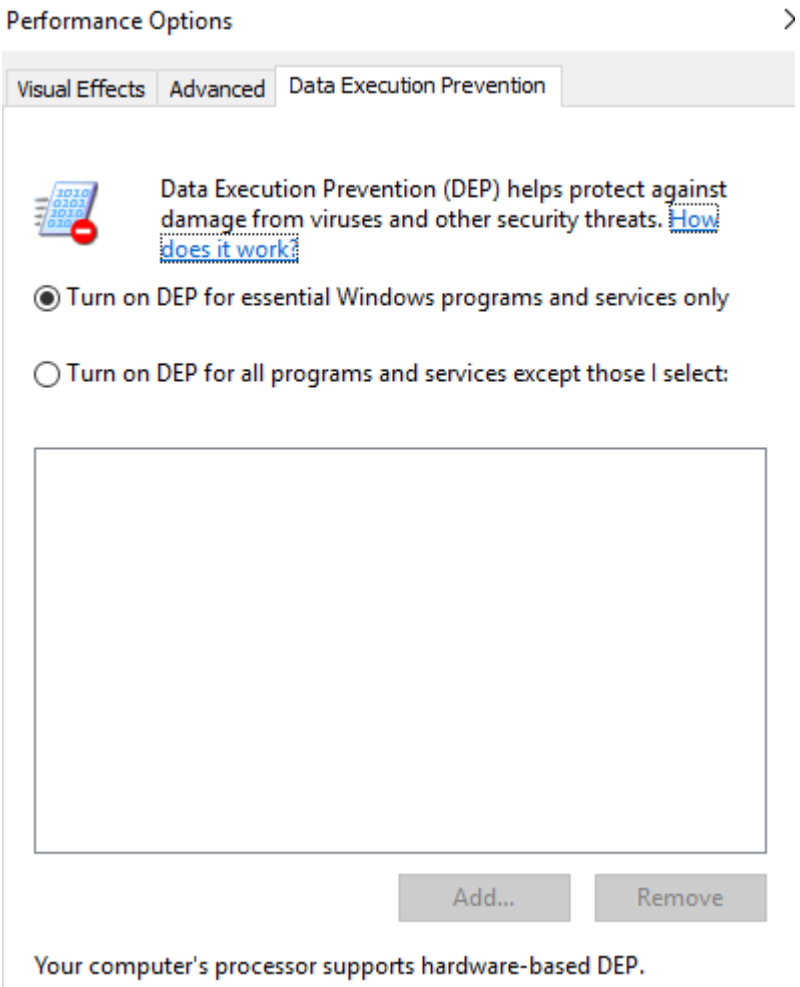

Figure 1. Windows DEP settings

2.6 Execution monitors

Another countermeasure to attacks would be to monitor your programs for events that the program shouldn't be doing and then perform an action based on the event, like shutting down the program immediately if it tries to execute shell commands with system() function.

A pattern could also be formed of a program's system calls, and then if a system call doesn't match to the pattern, it could be marked as an anomaly and the program's execution could be stopped.

The program could also be given the least privileges it needs to work properly in policy management. So, the program couldn't execute system calls that are defined as unnecessary for the program's execution. The same privileges should be applied to the directory where the program is located and the file it needs to access. (Yves Younan et. al. Execution Monitors)

2.7 Static code analysis

One way to prevent attackers from abusing your programs would be to use analysis tools to check your code for vulnerabilities. There's a long list of available tools for C, for example Microsoft's Visual Studio IDE has one in the editor and from compiler command line. In Visual Studio, a rule set can be chosen for your program. Microsoft has a few default rule sets, that either only check for security holes and potential crashes or more broadly for a range of problems in your code. Eclipse IDE has a similar code analysis tool, where you can choose what problems you want to be warned about. (Figure 2)

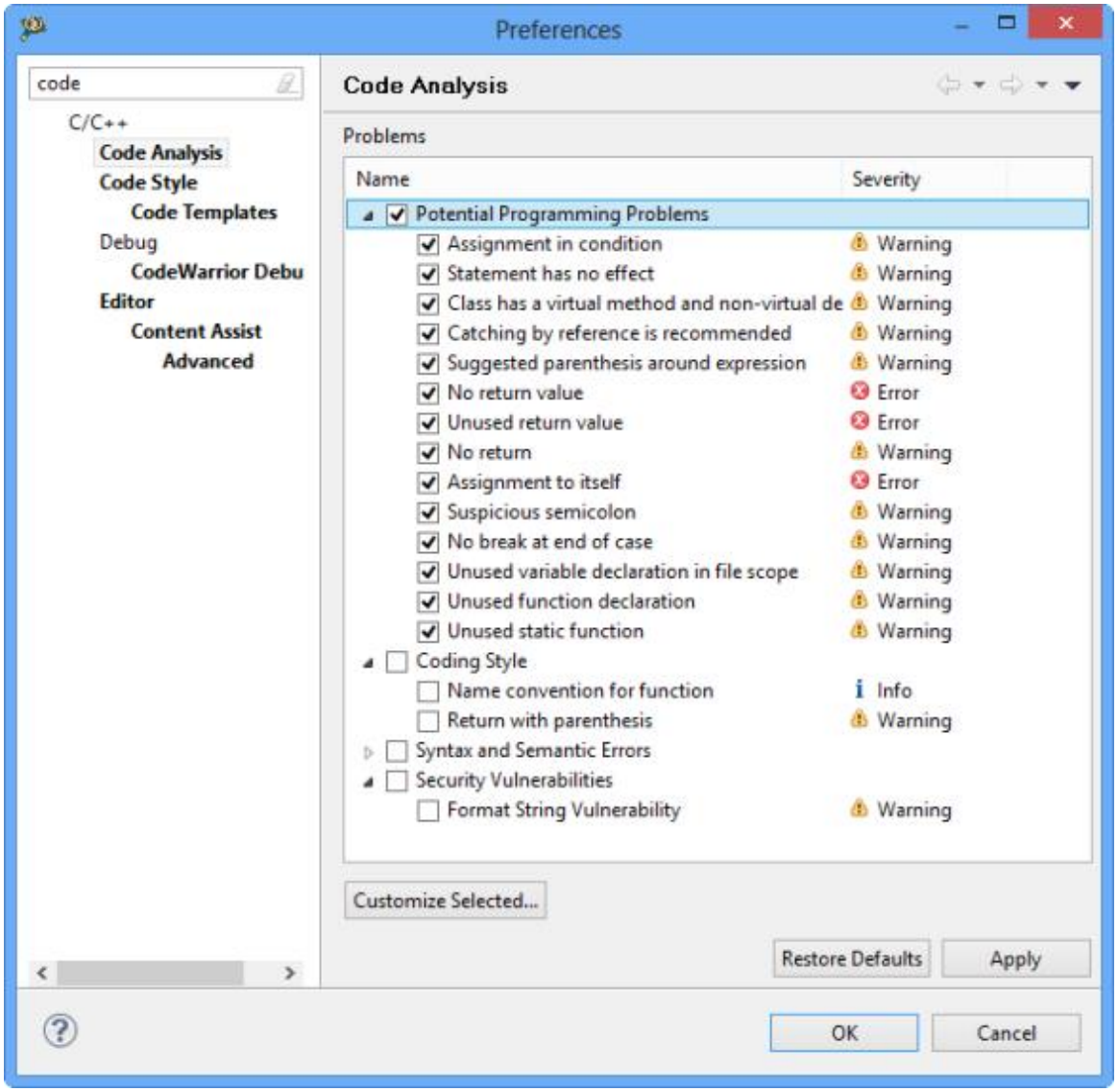

Figure 2. Eclipse Code Analysis tool (Erich Styger. Free Static Code Analysis with Eclipse.)

In our previous assignment's makefiles, there have been such flags as "Wno-deprecated" and "Wno-deprecated-declarations". The first one disables warnings about usage of what are considered now deprecated features. For example, minimum and maximum operators "<?" and ">?" should be replaced with std::min and std::max. The "Wno-deprecated-declarations" flag causes the compiler not to warn about the use of deprecated functions, variables and types. (Free Software Foundation. Using the GNU Compiler Collection (GCC).)

Sources

Erich Styger. Free Static Code Analysis with Eclipse. 2013. https://mcuoneclipse.com/2013/01/06/free-static-code-analysis-with-eclipse/

Free Software Foundation. Using the GNU Compiler Collection (GCC). 2008. https://gcc.gnu.org/onlinedocs/gcc/Deprecated-Features.html#Deprecated-Features

Martin. Hardening C/C++ Programs Part II – Executable-Space Protection and ASLR. 2017. http://www.productive-cpp.com/hardening-cpp-programs-executable-space-protection-address-space-layout-randomization-aslr/

Martin. Hardening C/C++ Programs Part I – Stack Protector. 2017. http://www.productive-cpp.com/hardening-cpp-programs-stack-protector/

Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

Will Dormann. Microsoft Office Equation Editor stack buffer overflow. 2017. https://www.kb.cert.org/vuls/id/421280

Yves Younan, Wouter Joosen, Frank Piessens. Runtime countermeasures for code injection attacks against C and C++ programs. 2010. http://younan.ca/files/csurv.pdf