# Return-to-libc attacks: exploitation & protection techniques

Anda-Maria Caldare, Oana-Alexandra Coica
Computer Science Department, University of Bucharest, Romania

## Abstract

Buffer overflow attacks and the more advanced versions that derived from them (such as Return-to-libc) continue to be a problem today, despite security researchers' and enthusiasts' efforts to address the issue. Software usually depends on external data and proprieties that ultimately control its actions and final results and it is becoming more sophisticated, getting harder to predict its behavior, old code gets reused, safer programming languages and libraries can't win the popularity prize against the old (and often flawed) ones, weak programming techniques stay in place, most applications nowadays remaining still exposed to attacks related to memory safety violations. Thus, it is important to analyze technological solutions that either eliminate the flaws that could lead to buffer overflows or at least make the execution more difficult. This study will present a technical description of the attacks and analyze a few mitigation strategies, their approaches and effectiveness.

## 1 Introduction

Probably one of the most common misconceptions in the security field might be the myth that buffer overflows, the primary attack vector for the most famous and destructive worms in history (Morris, Code Red, SQL Slammer, Conficker), are obsolete. Surprisingly, this kind of memory safety issue might never vanish after all and the access error vulnerability might even regain its great popularity pretty soon among the black-hat hackers, given the recently discovered CVE-2021-3156 [1] sudo buffer-overflow exploit that managed to remain undiscovered for nearly a decade.

Writing memory safe code is a laborious task to accomplish and the consequences of spatial safety violations (such as the ones linked to buffer overflow) can have multiple undesirable consequences. As a result, researching solutions capable of thwarting buffer overflow attacks is a truly pressing matter.

## 2 Exploitation

Buffer overflow attacks fall in the category of control hijacking attacks, because the malicious actor's goal is typically to trick the target machine to execute a certain sequence of instructions by hijacking the application control flow, by violating the memory safety of the stack or heap [2]. By overrunning the buffer's limited space, an attacker will be able to reach adjacent memory areas and replace legitimate data with malicious code. What makes this type of attack possible is the combination between a programmer's (false) assumptions and a lack of automatic checking for writing beyond the bounds of an array and unsafe string handling in case of C compilers.

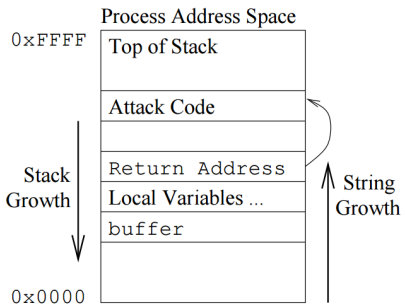


Figure 1: Stack smashing [3]

Buffer overflow vulnerabilities generally have two types of consequences – their exploitation can lead either to crashes or infinite loops, triggering denial-of-service attacks and disrupting the availability,

or to the execution of arbitrary code, ignoring the original scope of the program.

The most common type of buffer overflow attack is the one where the attacker-controlled memory is on the stack, known as stack smashing attack and pictured in Figure 1, with the return address being a very popular target. Often, a local buffer variable inside a function is overflowed, the malicious actor managing to change the return address from the stack activation record to something that points to the attack code. When the vulnerable function returns, the program passes control to the malevolent code which is executed [4].

Ideally, we should be able to find a way to prevent the code from being executed - after all, the stack should only contain data, not shellcode, right? What happens if we make the stack area not-executable? This solution is known as Data Execution Prevention (DEP) or W $\oplus$ X and it's probably what pushed the attackers to discover a new type of buffer overflow capable to bypass this protection technique - return-to-libc [2].

Return-to-libc (or ret2libc) is a special type of buffer overflow where the attacker manages to replace the return address in such a manner that it will point to a function within the libc library. The immunity against DEP derives from the fact that no shell/ attack code is needed to be injected onto the stack and from the natural possibility of legally executing existing instructions from the code segment (exec(), system()) [5].

Because ret2libc means replacing the EIP address (that points to the instruction waiting to be executed next) with the address of a library function and arranging the stack data to fit the needed arguments, as expected, the attacker's actions are limited to what kind of functions the library provides, arbitrary code execution not being an option. Still, the approach can prove itself to be destructive, despite its limitations.

# 3 Compiler-enforced protection techniques

## 3.1 StackGuard

StackGuard is a gcc compiler patch which prevents stack smashing without altering the source code. There are 2 modes in which StackGuard works, depending on the actions taken to prevent malicious modifications to the return address of the program: detecting the attack (using canaries) and preventing the attack (using MemGuard). Both approaches force the program to exit in case of an attack, and StackGuard can shift between them, depending on the nature of the attack. [3]

### 3.1.1 Canaries variant

The idea behind this approach is to make use of a designated value on the stack (called "canary word") that must remain intact for the program to properly function and exit. When using canary words, StackGuard alters the assembly functions called prologue and epilogue, whose purpose is to save and restore states when calling a function. These functions are enhanced by saving the canary in the prologue and checking its integrity in the epilogue, which is executed before the program returns. [6] If this canary was modified, it means that the return address has been overwritten and the program is forcingly exited. The canary word is placed between the buffer and the return address, as shown in Figure 2 below.

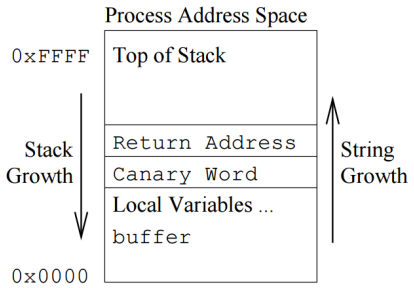


Figure 2: Placement of the canary word on the stack [3]

Since buffer overflows basically copy more data than allocated into the buffer in order to try overwriting the return address to make it execute a malicious shell code, doing so without damaging the canary word is very difficult to achieve. There are 3 types of canaries and StackGuard supports all of them: terminator, random and random XOR. A terminator canary contains string terminators (such as NULL, EOF, CR and LF) that will stop malicious string operations (like strcpy()), operations that most buffer overflows make use of. The

random canary is randomly initialized when the program starts, and the random XOR canary is the XOR result between a random value and the return address. [7]

#### 3.1.2 MemGuard variant

StackGuard offers another method, MemGuard, which is less performant but more secure compared to the canaries version. MemGuard also makes use of the assembler prologue and epilogue functions, but instead of placing and checking the canary word, it protects and unprotects the return address, by making it read-only. Since this variant of StackGuard is less performant, it is used mostly for debugging. However, it can be combined with the canaries in the following manner: the canary integrity is detected to be compromised and then MemGuard protection is activated. [3]

### 3.2 StackShield

StackShield is a tool seen as an assembler file processor that also alters the prologue and epilogue functions, but not the source code. Instead of checking the integrity of a canary word, like in StackGuard's case, StackShield saves the return address to a different stack (called retarray), located in a place unlikely to be overwritten. There are two possibilities to choose from regarding what happens before the function returns: either the saved value is compared to the actual stack value of the return address and the program is forcingly exited if the values are not the same, or the stack value is always replaced with the saved value. However, the latter version means that the stack return address will never be used, and in addition that gives opportunity to attacks in case the retarray stack has been modified.[8, 9]

### 3.3 ProPolice

ProPolice is a feature included in gcc compiler and is usually known as SSP (Stack Smashing Protection). SSP operates in two manners, that can be activated and deactivated when compiling programs.The first technique used by SSP is using random canaries, like it was previously discussed for StackGuard in 3.1.1: canary words are being placed before the return address of the functions, their integrity is checked before the return, and if the value is compromised, the program exits. The other approach of SSP consists in rearranging the stack in such a manner that if an overflow occurs, the affected data is not as important, for example buffers are being placed before other local variables. [8, 5]

### 3.4 Limitations

These 3 tools described above are considered to be effective, but all of them have their own limitations. To begin with, StackGuard does not prevent heap buffer overflows. Besides this, buffer overflows can still happen if the canary value is known by the attacker, either by guessing it or by information being leaked by the program. The canary word value can, therefore, be placed in the correct position inside the malicious code, and it would not be detected [3, 7]. Secondly, since StackShield only saves a copy of the return addresses, other data can be compromised without this being detected. Moreover, it can be possible to alter the saved copy itself, and since that copy always replaces the return address, the attacker code will be executed [6]. Lastly, ProPolice is limited by the design of the program itself, because if, for example, the buffer's length is too small, the overflow can reach the return address. Besides that, depending on the structure, the rearranging of the stack might not be always possible [5].

## 4 Kernel-enforced protection techniques (PaX)

### 4.1 NOEXEC

NOEXEC is a PaX approach that basically refers to giving executable rights only to the memory pages that need such rights. It is further split into: the actual implementation of tagging the pages as non-executable (through PAGEEXEC and SEGMEXEC) and the enabling of restrictions (through MPROTECT). PAGEEXEC makes use of two memory cache tables that store virtual addresses-physical addresses mappings and, when trying to look up a protected page in the tables, it will result in a page fault exception, at which moment it will

be decided if the request was a legitimate data access or a malicious code (resulting in exiting). SEGMEXEC is based on separating the address ranges for code and data segments, thus enabling the program to be exited if there is an executable request found in the data range. MPROTECT basically consists of separating the writable and executable rights, to prevent writing new malicious code to pages that are tagged as executable (thus giving it default permission to execute it). This is done by carefully choosing certain flags used in mmap() and mprotect() interfaces. [10, 5]

### 4.2 ASLR

ASLR (Address Space Layout Randomization) is a randomization technique that deters attackers from jumping to the location of libc functions by mapping the shared libraries (also stack, heap and binaries) to an unknown (pseudo) random location in memory. Effectively, every time the process executes, the virtual memory layout is changed, so the function address is going to be different [10, 11].

### 4.3 Limitations

The two mitigations provided by the PaX patch are critically codependent. If any of those is intentionally or accidentally deactivated, the solution isn't safe anymore. NOEXEC, by itself, has the same problems as DEP (discussed in Section 2), while ASLR can't provide enough randomization, so it might be vulnerable to brute force, timing attacks or "return-to-PLT" attacks [11].

Otherwise, according to [5], the PaX patch offers important protection against all traditional types of buffer overflows on both the stack and the heap when all its security functions are active.

## 5 Conclusions

Given the popularity of the buffer overflow (especially seen as a common attack vector for internet worms), a lot of protection methods were implemented to prevent the exploitation of this kinds of vulnerabilities - either pre-design, through build approaches (such as StackGuard), or operational, at the operating system level (such as the mitigations provided by PaX).

These solutions have made exploitation a lot more difficult for the attacker, but they do have their limitations. Also, protections are sometimes switched off intentionally to allow certain applications to run properly, sometimes deactivated by default (and not very popular among users) or they do not even exist on specific platforms. The offensive techniques are evolving constantly, attackers discovering ways to bypass countermeasures or even use those protection methods in place to find a new way in. It is important to realize that evading defenses is possible and consequences can be tough.

A "defense in depth" approach (consisting of activating all existing protections, all the run-time detection and prevention techniques, based on signatures or machine learning) might lower the chances of a successful attack [2], but the final layer of responsibility might actually turn out to rest on the software developer's shoulders.

## References


[1] NIST: National Institute of Standards and Technology, "National Vulnerability Database - CVE-2021-3156." `https://nvd.nist.gov/vuln/detail/CVE-2021-3156`.

[2] D. J. Day, Z. Zhao, and M. Ma, "Detecting return-to-libc buffer overflow attacks using network intrusion detection systems," in *2010 Fourth International Conference on Digital Society*, pp. 172–177, IEEE, 2010.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX security symposium*, vol. 98, pp. 63–78, San Antonio, TX, 1998.

[4] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[5] P. Silberman and R. Johnson, "A comparison of buffer overflow prevention implementations and weaknesses," *IDEFENSE, August*, 2004.

[6] J. N. Gupta and S. Sharma, *Handbook of research on information security and assurance*. IGI Global, 2008.

[7] Sidhpurwala, Huzaifa, "Security Technologies: Stack Smashing Protection (StackGuard)." `https://access.redhat.com/blogs/766093/posts/3548631`, 2018.

[8] G. Richarte *et al.*, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, vol. 1, 2002.

[9] "Stack Shield - A "stack smashing" technique protection tool for Linux." `https://www.angelfire.com/sk/stackshield/`, 2000.

[10] "Documentation for the PaX project." `https://pax.grsecurity.net/docs/index.html`.

[11] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, vol. 70, 2001.