Defend methods for Return-to-libc attacks

Larisa-[redacted], Miruna [redacted]
Computer Science Department, University of Bucharest, Romania

Abstract

Despite the development of software defense techniques, buffer overflow attacks are not yet obsolete and continue to be a relevant discussion topic in today's cybersecurity field. A "return-to-libc" vulnerability is a buffer overflow based attack that replaces a subroutine return address on a call stack. Thus, it is critical to investigate possible defense approaches that can defeat the weaknesses which lead to this type of attack. With this outline in mind, today's paper objective is to research and provide a detailed technical description to some famous strategies developed to defend against security breaches of type "return-to-libc".

1 Introduction

Buffer overflows rose to fame after the 1988 "Morris Worm Incident" [17] and only continued to grow in popularity: between 1988 and 2000 almost half of the attacks were of this type, per CERT/CC statistics [17]. In recent years, this infamous vulnerability was the foundation for Operation Soft Cell in 2019 or Heartbleed in 2014. Most recently, CVE-2022-1483 [13] was found in Microsoft Windows Network File System - This vulnerability is caused by the Network Lock Manager (NLM) RPC program which handles Remote Procedure Call (RPC) responses to Portmap requests inadequately.

This type of attack seeks to corrupt the memory safety of either the heap or the stack. As the name suggests, this is caused by a breach in the buffer array, arising from an absence of bounds validation on the size of input stored in it. The malicious party can overwrite the buffer and gain unauthorized access to other parts of the program, where the code will be replaced with malicious data [17]. The issue is so prevalent in today's digital era because there are a multitude of applications written in C and/or C++ and both of these programming languages lack built-in security measures against the illegal access and overwriting of data, regardless of where it is situated in the memory [4]. As mentioned above, there are 2 main types of buffer overflow techniques:

- Heap-based attacks that are not very frequent because they are more difficult to carry out and exploit. This approach implies overflowing the memory allocated to a program beyond memory used for current runtime operations.
- Stack-based ones that are the most recurrent and popular, also known as stack smashing attacks, where the targeted memory is the one present on the stack.

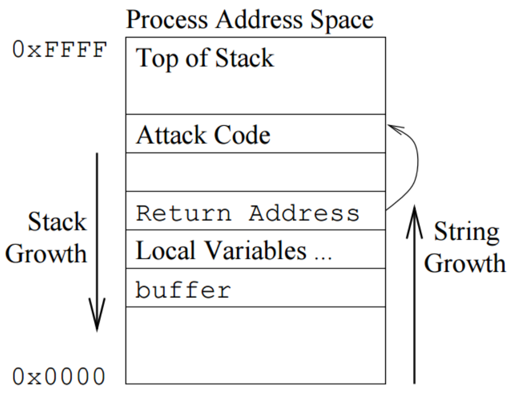

Fig.1.1: Smashing Stack scheme [2]

Usually, with stack smashing, there are 2 objectives that are sought by the attacker, concurrently: first, to inject some malicious code (in form of binary code specific to the system that is being targeted), aiming to create and open a root shell; and second, to change the return address of the called function so that it will jump to the attack code, effectively starting its execution, instead of jumping back into the program [2].

The obvious fix to such a vulnerability would imply simply restricting the stack from allowing code to be executed, considering the purpose of the stack is not to store shellcode. Such a solution thankfully exists and is implemented on all modern operating systems (OS). Known as Data Execution Prevention (DEP for short, also called "$W \oplus X$"), it's a security feature designed to avoid allowing an application to run code from a non-executable memory area [3]. This countermeasure to the buffer overflow attack is the catalyst for apparition of the return-to-libc technique.

Return-to-libc (or ret2libc) appeared as a response to the DEP protection, from a desire of the malicious entities to bypass the non-executable stack. The implementing idea is pretty similar to the stack smashing method, but, instead of pointing to the attack code in the stack, the EIP address (which points to the next instruction) will indicate to a function inside the libc library [1]. And although the attacker does not have the liberty of executing arbitrary code, even this approach can be quite destructive and thus, important for us, as programmers, to learn to protect against such techniques.

At the moment, there are no 100% secure methods, but we can rely on some approaches that help decrease the risk of performing a successful return2libc attack. These methods range from Kernel to compiler-enforced protection [1]. In the following pages, we will take a look at these techniques and try to observe their effectiveness and their limitations.

2 Kernel-enforced protection techniques

PaX is a Linux kernel patch which provides methods to protect against unwanted code execution, as well as stack and heap buffer overflow, using the addresses randomization and control of memory access. [11]

2.1 NOEXEC

NOEXEC is a component of PaX which has the scope to protect against arbitrary code execution by managing the execution rights so that it can be done only by restricted memory pages. The strategy of giving the least-privilege can be applied based on paging - PAGEEXEC - or based on segmentation logic - SEGMEXEC in order to identify non executable pages. If the scope is to block new mappings of executable code, then the strategy can be based on the functions mprotect () and mmap () - MPROTECT - actually consisting in the careful choice of the parameters.

PAGEEXEC's goal is to use the paging logic of IA-32-based CPUs to create an implementation of the non-executable page feature. It uses TLB - Translation Lookaside Buffer - to save in two different buffers the cache for virtual-to-physical address translation. These two buffers are used for instruction fetches and data fetches, both generating page faults which are in the next step transmitted to a handler in order to determine the type of fetch. An exception is thrown when a protected page is searched in the stored addresses (both virtual and physical), so that, as the last step, a verdict can be concluded: the request either contains secure or malicious data.

A similar implementation based on segmentation logic is SEGMEXEC which uses two segments to separate the data from the code in order to stop the program if there is a request found in the first named segment. [11] [12]

2.2 ASLR

ASLR, the abbreviation for Address Space Layout Randomization has, as the name suggests, the role to randomize the layout of the virtual memory address space in order to become inaccessible to hackers. It is based on the idea that most attacks rely on static data such addresses that are guaranteed to include specified operands or pointers to a precise location of a buffer on the stack. With this strategy, ASLR effectively minimizes the possibility that an attack that depends on hardcoded addresses inside the executable binaries, locations of loaded libraries and the stack or the heap will supply the attacker buffer with redirected code execution.

An implementation of this concept can be done with PaX, which is made up of four primary components: RANDUSTACK, RANDKSTACK, RANDMMAP, and RANDEXEC. PaX's RANDUSTACK component is in charge of generating random userland stack addresses.PaX's RANDKSTACK component is in charge of adding randomization to a task's kernel stack. The component RANDMMAP is in charge of randomizing all file and anonymous memory mappings.RANDEXEC is in charge of relocating ET EXEC ELF binaries in a random order. [9]

2.3 ASCII armor

ASCII-Armor - is a technique that has the aim to defend against return-to-libc attacks by treating the character 0x00 from the beginning of an address as a NULL character marking the end, usually in the high address of the shared library (the addresses will contain a NULL byte). In this way, the functions which use NULL characters as a finish character delimiter, will terminate the execution. Such examples of functions can be: strcpy, sprintf, strlen. [15]

So, implementing a buffer overflow attack payload will be a challenge for the hackers when having ASCII-Armor, because if the payload has 0x00, the attack is stopped when this hidden NULL character is met, preventing the infected strings from being read.

2.4 Limitations

The enforcement flags of PaX are set with the scope of blocking the attacking from determining which shared library would take preference. This is the reason why these flags can impact very much the security of the entire system if they are misconfigured. Some general disadvantages of these approaches are that depend on the platform used and impact the performance of the architectures that does not support non-executable features. [11]

Even is NOEXEC seemes like an indestructible method of defense, a study [6] demonstrate that there are some strategies which can bypass the PaX security. The attacker can determine the random addresses of the library on his local system, but in order to move the attack remotely, another vulnerability is needed. (as small as it would be, such as information leak, it would be enough to help). [5] Simple, plain NOEXEC has similar problems as DEP has.

In ASLR case, if combined with Memory Management Unit Access Control Lists (MMU ACLs), the protection should be high enough against remote exploit attacks. Despite this fact, the most relevant disadvantage of simple ASLR is that, even if this technique is based on randomization, it can not provide enough randomness to guarantee protection against brute-force attacks or even timing ones.

ASCII armor is a very simple methods based on simple concepts that offers the minimum of protection. The goal is not to serve alone as a protection algorithm, so it should be merged into more sophisticated techniques. This method has a big disadvantage if the attacker overflows the NULL byte into the stack, so the whole strategy will be canceled. [15]

3 Compiler-enforced protection techniques

The goal for this type of approach is to prevent the exploitation of the buffer by controlling the data which is stored on the stack and maintaining its integrity.

3.1 StackGuard

StackGuard it's a GCC compiler patch created by Immunix Inc. and protects against return2libc attacks by either checking if the stack was not altered in some way or by ensuring that the return address is READ-ONLY. In the event of a malicious threat, both strategies will force the program to exit its execution and StackGuard can switch between the 2 of them based on the nature of the attack [2].

3.1.1 StackGuard with Canaries

First of all, what is a "canary"? A canary is a random word/value, located on the stack, right next to the return address. Any changes brought to the return address will affect the canary word, so, by monitoring it, StackGuard can prevent a return-to-libc attack. In order to accomplish this, an alteration to the `function_prologue` and `function_epilogue` (these functions have the role of saving, respectively restoring the stack when a function is called) is made: The `function_prologue` is going to place the canary word on the stack and save its value and the `function_epilogue` will check if the canary was modified. If an alteration is discovered, StackGuard will force the program to exit [2].

A potential malicious exploitation might be successful against this approach if the attacker can somehow preserve the canary (not impossible, but very difficult). In order to cover all grounds and leave no vulnerabilities, a few versions for the canaries were suggested [16]:

- Random Canary: a pseudorandom 32-bit value generated at run-time;
- Terminator Canary: a combination of the 4 types of termination strings (Null, Carriage Return - CR, Line Feed - LF, EOF - 0xFF);
- Random XOR Canary: the value is calculated by XOR-ing the random canary with the return address.

3.1.2 StackGuard with MemGuard

This approach also makes use of the `function_prologue` and `function_epilogue` but, instead of checking the integrity of the canary, in the `function_prologue` the return address is marked as READ-ONLY and then, in the `function_epilogue`, it's unmarked [2].

3.1.3 Effectiveness and Limitations

In terms of performance versus security, the canaries method obtains a better efficiency, but the MemGuard method is more safe. Because the MemGuard write is approx. 1800 times the cost of a normal write, this technique is mostly used in debugging. We can combine the 2 approaches: if the canaries variant gets compromised, we can activate the MemGuard protection [2].

The biggest concern is the ability of the attacker to get the canary word, either by information leaking of the program or by guessing it or avoiding it altogether. At the same time, StackGuard can detect the attack only after the function finishes, giving the malicious entity some time to play. Of course, there are other ways in which an attacker might be able to defend StackGuard and that involves overwriting frame pointers and/or function pointers [2] [10].

3.2 StackShield

StackShield is a security extension for GCC compilers, developed by Vendicator [14]. It has 3 different types of strategies that also make use of the `function_prologue` and `function_epilogue`: 2 methods that deal with the protection of the return address and one method that will check the integrity of function pointers [16].

3.2.1 Global Return Stack Technique

This solution implies the existence of a different stack (called Global Return Stack or `retarray` [8]) which will also store the return addresses, together with the normal stack. When the function finishes, the return address from the stack will be replaced with its equivalent from the `retarray`. It is important to underline the fact that the 2 values are not compared (prevention, not detection). This can lead to a dangerous vulnerability: the attacker might be able to modify the Global Return Stack. As a result, this method is usually used with the Return Range Check approach [16].

3.2.2 Return Range Check Technique

This method uses a global variable (in an unwriteable area [7]) which will save the return address of the function being called. Before the function is returned, its address and the global variable are compared. If the 2 values are different, then the execution stops (detection) [16].

3.2.3 Protection of Function Pointers Technique

The idea behind the implementation is simple: function pointers should only return into the `.text` segment and so, StackShield will check to see if the pointers will target an address situated below `shielddatabase` (an address which marks the beginning of a program's data [8]). If the pointers will point above `shielddatabase`, it will be assumed that an attack took place and the program will stop. This method has a limitation because the program cannot have dynamic allocation [16].

3.2.4 Limitations

Similar to StackGuard, StackShield is not fully secure, although it brings some enhancements. The attacker still has a window in which he/she can bring modifications to the program before it gets detected and there are still other possibilities for attack (such as overwriting the Global Offset Table - GOT [7]).

3.3 Stack-Smashing Protection

Stack-Smashing Protection (SSP), formerly known as ProPolice, is, perhaps, the most refined compiler extension. It borrows the concept of canaries from StackGuard and enhances it. What is new is the fact that this technique rearranges the stack in such a way that reflects the order from the below figure, regardless of the way the programmer declares the variables, pointers and buffers [16]:

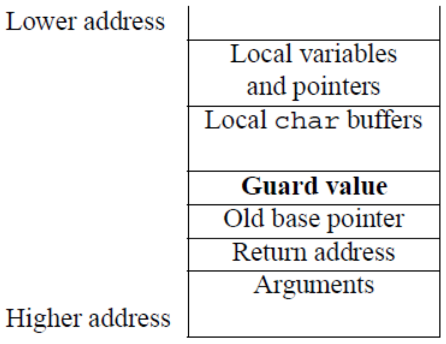

Fig3.3.1: The rearranged stack - SSP [16]

Even if the char buffers are overflown, they cannot harm any other local variables or pointers because of their position on the stack. As was the case with StackGuard, the value of the canary is checked to see if it was tampered with. If an attack is detected, then the program exits its execution [8].

This solution is perhaps one of the most secure techniques, even though it has its own limitations. It was discovered that SSP does not protect buffers which contain less than 8 elements, thus an overflow is possible and can reach the return address of the function. It is also feasible that the reorganization of the stack may not always be achievable, depending on the design of the program itself, also leaving some structures unprotected [7].

3.4 Libsafe & Libverify

This defense uses modified C library function, made more secure, and a check of the return address and base pointer (which is placed right before the return address). The targeted C functions are those which can potentially lead to a buffer overflow: `strcpy`, `strcat`, `getwd`, `gets`, `(f/s)scanf`, `realpath`, `(v)sprintf` [16].

3.4.1 Libsafe

Libsafe ensures that all variables must stop at the beginning of the old base pointer, defending against the possible overwrite of the return address in case of an overflow. This is also underlined by the fact that the aforementioned C functions are protected by a wrapper, which calculates the size of the input and of the buffer (the length from the buffer's starting point to the old base pointer) and verifies if the input is within the boundaries. If it's not, the program is halted [16].

3.4.2 Libverify

Libverify is similar to the Libsafe, but it brings an enhancement by using the canaries concept from StackGuard: all the return addresses of the called functions will be copied into a canary stack (placed on the heap) and before their return, the 2 addresses (from the canary stack and the return one) will be compared to see if they are identical. If that's not the case, the program is halted [16].

3.4.3 Limitations

In the case of Libsafe, it is important to note that the program halts, but the potential overflow of variables and/or pointers residing on the stack is not stopped. Also, vulnerabilities might still arise if the programmer codes its own memory handling implementation, thus Libverify is the recommended choice in this matter. But Libverify does not ensure the integrity of the canary stack and a malicious party might create a heap overflow attack [16].

4 Conclusion

Taking into consideration the fact that buffer overflow attacks are still effectively used even if they are common, the security community is still studying methods to prevent this kind of attacks improving the defense. There are already strategies implemented at the kernel level (operating systems) like the ones mentioned in the second section, or the build level, like the ones detailed in the third section meant to diminish the risk of attacks. Every mentioned strategy has its own advantages and disadvantages, but the protection system that they all create disarm the attackers making the exploitation process harder. The evolution of the attacks, the computing power gives a leverage to the hackers, so that the defensive strategies are not enough anymore. Even the defense team implements traps, security breaches that lure the assailants and new counter-attacks to keep up with creative attack strategies, but to do that the known vulnerabilities, prevention methods and common issues need to be studied in detail.

References

[1] c0ntext. Bypassing non-executable-stack during exploitation using return-to-libc. `https://css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf`.

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium*, volume 98, pages 63–78. USENIX Association, 1998.

[3] David J. Day, Zhengxu Zhao, and Minhua Ma. Detecting return-to-libc buffer overflow attacks using network intrusion detection systems. In *2010 Fourth International Conference on Digital Society*, pages 172–177, 2010.

[4] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, pages 177–190. USENIX Association, 2001.

[5] Dejan Lukan. Gentoo hardening: Part 4: Pax, rbac and clamav. `https://resources.infosecinstitute.com/topic/gentoo-hardening-part-4-pax-rbac-clamav`.

[6] Nergal. The advanced return-into-libc exploits: Pax case study. 2001. `http://phrack.org/issues/58/4.html#article`.

[7] Richard Johnson Peter Silberman. A comparison of buffer overflow prevention implementations and weaknesses. 2004. `https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf`.

[8] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. Jun 2000. `https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf`.

[9] Edward J. Schwartz. The danger of unradomized code. pages 9–11, Dec 2011.

[10] Huzaifa Sidhpurwala. Security technologies: Stack smashing protection (stackguard). In *RED HAT BLOG*, Aug 2018. `https://www.redhat.com/en/blog/security-technologies-stack-smashing-protection-stackguard`.

[11] Peter Silberman and Richard Johnson. A comparison of buffer overflow prevention implementations and weaknesses. pages 2, 5–8, 2004.

[12] the PaX Team. Documentation for the pax project. `https://pax.grsecurity.net/docs/`.

[13] Trend Micro Research Team. Cve-2022-26937: Microsoft windows network file system nlm portmap stack buffer overflow. `https://www.zerodayinitiative.com/blog/2022/6/7/cve-2022-26937-microsoft-windows-network-file-system-nlm-portmap-stack-buffer-overflow`, Jun 2022.

[14] Vendicator. Stack shield: A "stack smashing" technique protection tool for linux. Jan 2000. `https://www.angelfire.com/sk/stackshield/`.

[15] David A. Wheeler. Secure programmer: Countering buffer overflows. 27 January 2004.

[16] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 01 2003.

[17] Jun Xu, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar Krishnan Iyer. Architecture support for defending against buffer overflow attacks. 2002.