computers

MDPI

Review

A Review of Memory Errors Exploitation in x86-64

Conor Pirry, Hector Marco-Gisbert * and Carolyn Begg

School of Computing, Engineering and Physical Sciences, University of the West of Scotland, High Street, Paisley PA1 2BE, UK; B00283255@studentmail.uws.ac.uk (C.P.); Carolyn.Begg@uws.ac.uk (C.B.)

* Correspondence: hector.marco@uws.ac.uk; Tel.:+44-141-849-4418

Received: 23 April 2020; Accepted: 2 June 2020; Published: 8 June 2020

Abstract: Memory errors are still a serious threat affecting millions of devices worldwide. Recently, bounty programs have reached a new record, paying up to USD 2.5 million for one single vulnerability in Android and up to USD 2 million for Apple's operating system. In almost all cases, it is common to exploit memory errors in one or more stages to fully compromise those devices. In this paper, we review and discuss the importance of memory error vulnerabilities, and more specifically stack buffer overflows to provide a full view of how memory errors are exploited. We identify the root causes that make those attacks possible on modern x86-64 architecture in the presence of modern protection techniques. We have analyzed how unsafe library functions are prone to buffer overflows, revealing that although there are secure versions of those functions, they are not actually preventing buffer overflows from happening. Using secure functions does not result in software free from vulnerabilities and it requires developers to be security-aware. To overcome this problem, we discuss the three main security protection techniques present in all modern operating system; the non-eXecutable bit (NX), the Stack Smashing Protector (SSP) and the Address Space Layout Randomization (ASLR). After discussing their effectiveness, we conclude that although they provide a strong level of protection against classical exploitation techniques, modern attacks can bypass them.

Keywords: memory errors; x86-64; stack buffer overflows; SSP; ASLR; NX

1. Introduction

Every computing device, from the servers that comprise the infrastructure of the internet to electronic cars, to the internet-of-things devices that are increasingly making their way into people's homes, has digital memory.

Unfortunately, vulnerabilities exploiting memory errors are still a major threat. In fact, in a quantitative study conducted [1], it was found that memory buffer (overflow) errors account for 14% of all vulnerabilities reported to MITRE's Common Vulnerabilities and Exposures (CVE) database and the National Institute of Standards and Technology's National Vulnerability Database (NIST NVD) from 1988 to 2012. This makes them the most common vulnerability reported throughout these years [1]. Figure 1 illustrates the top vulnerability types where buffer overflows due to memory errors are still at the top of the ranking.

Preventing memory errors and particularly buffer overflows has been a challenge since the early 1970s as James P. Anderson (1972) highlighted in his [2] in his report, "Computer Security Technology Planning Study" [3]. Those vulnerabilities are still considered as an open security problem, which requires solutions that do not introduce significant overhead and allow programmers to fully use fast languages like C.

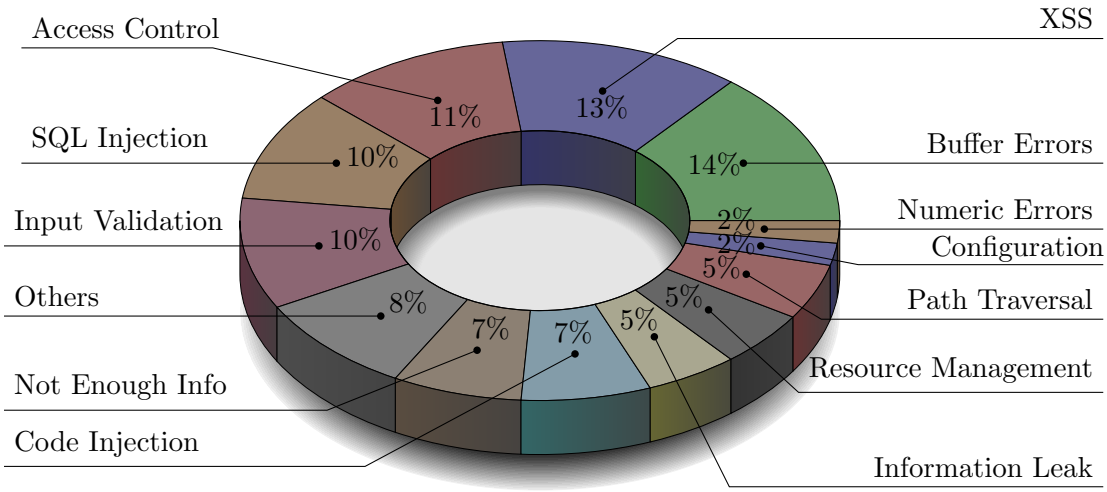

Figure 1. Top vulnerability types.

In February 2019, a CVE rated at high across all three impact metrics—(confidentiality, integrity, availability)—was released that describes a stack buffer overflow vulnerability in `libcurl`, affecting versions from 7.36.0 to 7.64.0 [4]. This library is the most used C-based highly portable transfer library in the world [5], provides functionality for end-users to carry out network-based resource transfers. Stenberg [6] claims that the library is used in every imaginable sort of embedded device where Internet transfers are needed.

Ever since Aleph One published the famous article, "Smashing The Stack For Fun And Profit" in Phrack magazine in 1996 [7], many memory protection mechanisms have been developed. The three main protection techniques employed by all modern operating systems are; the Stack Smashing Protector (SSP), which adds a canary/cookie/guard value in stack frames to mitigate against the overwriting of the frame return address; the Address Space Layout Randomisation (ASLR), which randomises the virtual memory addresses layout of a process memory to thwart attacks that relies on known memory locations; and the Non-eXecutable (NX), which enables readable memory to be non-writable, meaning that any malicious code injected for example into the stack as part of an attack payload will not be executed by the CPU. Although those protection techniques mitigates the exploitation, they do not eliminate the vulnerabilities itself but make the exploitation harder.

From the developers side, a memory corruption mitigation approach is essential for careful programming practices. That is, for example, avoiding the use of unsafe functions, checking array bounds, deallocating memory space no longer required, when possible. However, due to the need to maintain software on internet time [8], in which developers have to meet narrow deadlines, there is not much time for code and security testing. In many occasions, this leads to unnoticed, critical programming mistakes in publicly released software that are written in those languages which allow the freedom to, for example, interact directly with memory, such as C and C++.

Programmers working with other languages, such as Java or Python that cannot interact directly with memory, are not affected by these kind of memory errors because those languages have been built with protection mechanisms, such as array bounds checking and the automatic deallocation of memory space (a.k.a. garbage collection), so that memory corruption errors are less prevalent.

The main contributions of this paper are:

- A literature review on memory errors including new protection mechanisms that have not been considered in previous reviews.
- We identify the root causes that make memory errors attacks possible on modern x86-64 architecture.

- We reveal that "safe" functions are not actually preventing memory errors exploitation and developers should not wholly trust them.
- We examine the techniques developed to mitigate memory errors showing that no protection mechanism provides complete protection against known attack vectors.

This paper is organised as follows: Section 3 introduces the reader to the x86-64 architecture and provides the knowledge required to understand real memory errors exploitation. In Section 3.6 we detail both, the process and stack layout on the x86-64 architecture. In Section 4 we review the literature of buffer overflows and attack vectors used by adversaries. Section 5 presents attack approaches when probing buffer overflow vulnerabilities. Section 6 introduces the three main protection techniques present in all modern operating systems that have been developed in an effort to mitigate memory errors, including the recent literature that bypasses all those protection techniques. Finally, we conclude the paper in Section 7.

2. Literature Review

2.1. Memory Errors

The term "memory error" refers to a wide range of programming faults relating to how a processor interprets the contents of the main memory and what happens when that memory is misused or misinterpreted by a program. It is important to distinguish between physical errors caused by the underlying hardware (memory cells), due to electrical or mechanical defects, from the logical errors caused by incorrect coding. Essentially, the root cause of a memory error is a programming error and is therefore not related to hardware issues.

As cited by Steve McConnell [9] "A pair of studies performed [in 1973 and 1984] found that, of total errors reported, roughly 95% are caused by programmers, 2% by systems software (the compiler and the operating system), 2% by some other software, and 1% by the hardware. Systems software and development tools are used by many more people today than they were in the 1970s and 1980s, and so my best guess is that, today, an even higher percentage of errors are the programmers' fault." Application code is prone to containing programming errors.

2.2. Techniques to Protect Memory Errors

Compilers are now able to analyse the code and detect, alert and even generate the correct code for certain types of memory error. Most of the newer programming languages (Ada, Java, Python) try to hide the complexity of memory management all at once, by avoiding direct memory manipulation, which is an effective way to prevent most memory errors, albeit at a high cost in relation to expressiveness or overhead. Unfortunately, not all memory errors can be detected or captured before the code is released.

A decade ago, buffer overflows, especially the stack buffer overflow, was the most dangerous threat to computer system security. Over the last few years, several techniques have been developed to mitigate the ability to exploit this kind of programming fault [7,10].

The last line of defence to mitigate those programming faults is formed by a set of mitigation techniques that are active while the program is running, namely NX (Non-eXecutable), SSP (Stack Smashing Protector) and ASLR (Address Space Layout Randomisation), which do not remove the vulnerabilities per se but at least make them harder to exploit. These three mitigation techniques are very effective and easy to implement, and so most systems (Windows, Linux, Mac, Android, etc.) include most of them or slightly modified versions thereof. Most of the research effort focusing on the NX, SSP and ASLR techniques was carried out during the early 2000s. Once the techniques were consolidated and proved to be effective, they were gradually introduced into products.

2.3. Attackers Evolve and Innovate

Following the classic measure/counter-measure sequence, a few years after the introduction of each protection technique, new methods to bypass or reduce their effectiveness were introduced. The SSP can be bypassed using brute force or by overwriting non-shielded data [11–13], the ASLR can be bypassed using brute force attacks [13–15] and the NX, which effectively blocks the execution of injected code, can be bypassed using ROP (return-oriented programming) [16–18]. In spite of many existing counter-measures, these techniques are still effective protection methods, and in some cases they are the only barrier against attacks, until software is upgraded to remove a specific vulnerability.

2.4. Description of Memory Error Solutions

In response to those attacks, new and improved techniques have been proposed. The renew stack-smashing protector (RenewSSP) [19], is a new proposal which improves the stack-smashing protection (SSP) [20]. There is also a version named SSPMD [21], which is the renewSPP adapted to Android. The address-space layout randomisation next generation (ASLR-NG) [22] is the successor of the address-space layout randomisation (ASLR) [23]. Recently, Rick Edgecombe presented an improvement for the Non-eXecutable [24] technique by adding the concept of "execution-only" [25]. Those techniques do not remove the underlying error which leads to vulnerability, but they do prevent or hinder exploitation of the fault. The key idea behind the four techniques (SSP, RenewSSP, ASLR and ASLR-NG) is to introduce a secret that must be known by the attacker in order to bypass it, while the NX technique's method involves restricting the execution capabilities of processes.

2.5. Address Space Layout Randomisation (ASLR)

The ASLR is an abstract idea which has multiple implementations [22,26–29]. PaX published the first design and implementation of ASLR [23] in July 2001. The PaX project implementation is the most complete and advanced, also providing kernel stack randomisation from October 2002 onward. It also continues to provide the most entropy for each randomised layout compared to other implementations.

Two years after ASLR was invented and published as part of the PaX project, a popular security patch for Linux, OpenBSD became the first mainstream operating system to support partial ASLR (and to activate it by default) [30]. OpenBSD completed its ASLR support after Linux, in 2008, when it added support for PIE binaries [31].

Microsoft® Windows Vista® (released January 2007) was the first version of Windows® operating system to support ASLR [32]. Then all subsequent versions of Windows OS also supported ASLR [33]. There is a wide range of implementations with different levels of entropy, depending on the version and the security configuration: the Enhanced Mitigation Experience Toolkit (EMET), High Entropy ASLR or ForceASLR. For the purpose of this paper, we are only interested in the relative positions where each section of the program is loaded.

Apple® first introduced the randomisation of some library offsets in Mac OS X® v10.5 (released October 2007) [34]. However, because this initial implementation was limited to only certain system libraries, it was naturally unable to protect against many attacks that a full ASLR implementation is designed to defeat. In Mac OS X Lion 10.7, Apple expanded its ASLR implementation to also cover application code. Apple stated that "address space layout randomisation (ASLR) has been improved for all applications. It is now available for 32-bit apps (as is heap memory protection), making 64-bit and 32-bit applications more resistant to attack." As for OS X Mountain Lion 10.8, the kernel, as well as kexts and zones, are randomly relocated during the system boot. As in the case of Windows, all applications see a concrete library at the same address. In 2019, ASLR-NG, a new ASLR design was proposed [22] which highly increases the absolute entropy and removes correlation attacks, such as the offset2lib attack [15].

2.6. Stack Smashing Protector (SSP)

The other main protection technique present in all modern operating systems is the stack smashing protector (SSP). The first proposal was presented in 1998 [20] and has improved over the years. The SSP technique [35] is a compiler extension which adds a guard (the canary) between the protected region of the stack and the local buffers. Later, the RAF SSP technique [19] was introduced which greatly increases (several orders of magnitude) the difficulty of an attack when the three protection techniques (SSP+ASLR+NX) are employed, as, is the case in most systems. Regarding the problems that may cause a canary change on a running process, authors identified that the error confinement that represents each forked thread is also a de-facto stack confinement which allows to change the value of the reference canary with no impact on the correct operation of process. Recently, in 2019, the technique was applied to Android OS [21].

2.7. Non-eXecutable (NX)

The Non-eXecutable [24] is a protection technique that allows a memory region to be readable but not writable. Recently, using a similar idea, Rick Edgecombe extended this technique by adding the concept of "execution-only" [25]. The idea is quite similar to the original NX where a memory page can be either executable or writable but not both at the same time. The "execution-only" approach allows a memory page be marked as "executable only" which means that the page cannot be modified or read. This is motivated by the current knowledge about attacks that first read the memory content of the process to later exploit the target. These attacks [36] allow ROP attack techniques even in scenarios where attackers have no knowledge about the target binary.

3. Background: x86-64 Architecture Overview

In order to properly understand the memory errors and their exploitations, it is essential to choose a specific architecture as a reference because these attacks are architecture dependant. Although the same ideas and concepts can be applied to other architectures with minimal changes, this section is intended to provide the reader with the required information and knowledge to understand memory errors on the x86-64 architecture.

This section will outline some key instructions that the CPU uses during software execution. Afterwards, stack memory and stack frames will be introduced.

3.1. Endianness

Endianness is the order in which bytes are stored in memory. There are two forms: big-endian and little-endian. The x86-64 architecture uses little-endian form [37]. Little-endian means that the least-significant byte (LSB) is stored at the lowest memory address and the most-significant (MSB) at the highest, where the LSB is the first byte and the MSB is the last.

In the context of memory errors exploitation, knowing the endianness is important, specially in scenarios where attackers have the ability to produce memory overflows at byte level when exploiting vulnerabilities. For example, attacks such as return-to-csu and byte-for-byte SSP attack or even the blind ROP attack, all assume that attackers have the capability to overwrite an arbitrary number of bytes (not words or half-words) in memory. In this scenario, attackers need to know which byte they are overwriting and this depends on the endianness. It is widely know that on some architectures, one of the bytes of the Stack Smashing Protector is set to zero, and therefore attackers could also guess the endianness from that observation and quickly identify whether they are overwriting the Stack Smashing Protector, the saved base pointer, etc.

Figure 2 shows a little-endian example which stores the variable `dbeef_var` that has been initialised with the value `0xDEADBEEF`.

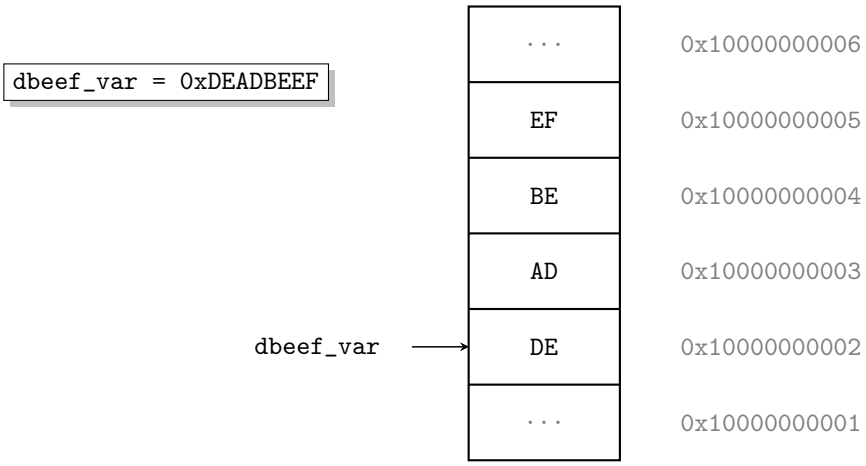

Figure 2. Little-endian memory layout example after storing `0xDEADBEEF` in memory. The variable content starts at lower addresses.

3.2. Strings

Whilst little-endian is true for how integers and memory addresses are stored, strings are not affected by the little-endian order. This is due to the way strings are created and handled, in regards to the C programming language.

In C and C++, strings are stored as a sequence of characters surrounded by double-quotes [38], which are concatenated together during compilation to make a single string. As will be discussed in Section 3.6 Stack Memory, strings (stored as local variables) grow towards higher memory addresses—each character is stored at a memory address, starting with the first character at the lowest address.

To define a string using C, the programmer creates an array of the `char` type. This is demonstrated in Listing 1:

Listing 1: Defining and initialising a string in C.

```
1 int main() {
2    char greeting[] = "Hello World\0";
3 }
```

This code creates an array of characters called `greeting` and places the characters in "Hello World" into each element of the array, such that the first element contains "H" (`0x48` in hexadecimal), the second element contains "e" (`0x65` in hexadecimal) and so on.

Listing 2 shows in hexdacimal the memory content (`string_memory`) where the `greeting[]` resides in the process memory. Note that the memory addresses (left-hand side) are in order of lowest to highest, confirming that strings grow upwards in memory.

Listing 2: String `greeting[]` process memory content growing upwards.

```
1 $ xxd -s 0x754 -l 32 c string_memory
2 00000754: 4865 6c6c 6f20 576f 726c 6400 0000 0000 Hello World.....
3 00000764: 011b 033b 3800 0000 0600 0000 ccfd ffff ...;8...........
```

3.3. Processor Registers

Processor registers are rapid-access storage locations embedded within a processor. In x86-64 systems, a processor contains sixteen general purpose registers (GPRs) [39]. Registers are utilised in Assembly languages and can be equated with variables in higher-level programming languages;

they are used as operands in an instruction to perform a task, such as store a value. Table 1 shows the sixteen general purpose registers and their sub-registers.

Table 1. The sixteen x86-64 general purpose registers and their sub-registers.

Size (in Bits)			
64	**32**	**16**	**8**
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8∼R15	R8D∼R15D	R8W∼R15W	R8L∼R15L

These registers are general purpose, and can be used in many assembly operations, some do have conventional uses [40]. The RAX stores the result of logical or arithmetic operations, the 'accumulator' register, the RBX points to data in the Data segment, the RCX is usually a counter for string and loop operations, the RDX is employed for I/O pointers, the RDI for destination pointer for string operations, the RSI for source pointer for string operations, the RBP, the stack base pointer, which points to the bottom of the stack, and finally, the RSP, the stack pointer, which to the top of the stack.

As Table 1 shows, there are many more registers within the sixteen GPRs. This is because, in x86-64, the GPRs have had their width (used to refer to storage capacity, in this case) extended from 32 bits to 64 bits, but the legacy 32 bit, 16 bit and 8 bit registers are all still accessible.

The R prefix tells the CPU to use the entire 64 bit width of the register; the E prefix tells it to use the last 32 bits. Omitting a prefix tells the CPU to use only the last 16 bits and the L and H suffixes specify to use only the first (higher) or last (lower) 8 bits of the last 16 bits, respectively. When using the sub-registers, the bits leading up to that register are zeroed out. Figure 3 visualises the RAX register along with its sub-registers.

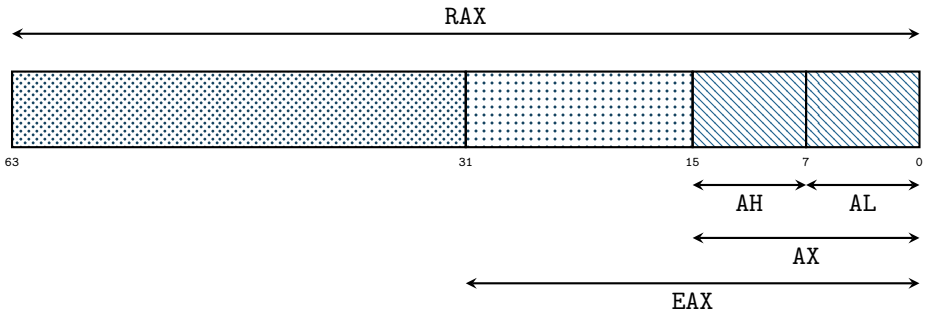

Figure 3. The RAX register and its sub-registers (in bits).

The processor needs to know which instruction is next in a program. To do this, it uses the 64 bits wide RIP, instruction pointer, register. This register stores the address of the next instruction to be executed [40] —it points to it.

Once an instruction has been executed, the RIP register (1) loads the memory address of the following instruction and jumps to that address, (2) reads (decodes) the value (an instruction opcode) stored at that address and (3) executes it. This is known as the fetch-decode-execute cycle.

3.4. System V Calling Convention

An application binary interface (ABI) is responsible for specifying how a binary executable should exchange information with some service [41]. The ABI in *nix systems is called System V. The System V ABI specifies how, amongst many other things, a function is called within software, at a low level. The function calling convention under System V is as follows [39]:

- The first argument passed to a function is moved into the `RDI` register;
- The second argument is moved to the `RSI` register;
- The third to `RDX`;
- The fourth to `RCX`;
- The fifth to `R8`;
- The sixth to `R9`;
- The system call number to `RAX`.

Any arguments beyond the sixth argument are passed on the stack, although this is a rare occurrence. It is important to note that arguments are passed in reverse order. That is, a function that requires three arguments, the third argument is moved to `RDX`, then the second to `RSI` and the first to `RDI`. In the case of system calls, the syscall number is moved to `RAX`.

3.5. Instruction Set

The x86-64 architecture includes a massive collection of instructions—discussion of every instruction is outside the scope of this paper. However, there are some instructions that are extensively used when developing exploits. Table 2 shows those instructions, including stack, data transfer and control transfer instructions.

Table 2. x86 stack instructions used in exploiting.

Instruction	Description
`push src`	Push `src` register on to the stack and increments `rsp` 8 bytes towards lower memory addresses
`pop src`	Copy the top stack value into `src` register and decrements `rsp` 8 bytes towards higher memory addresses
`mov dst, src`	Copy data stored in `src` register to `dst` register.
`jump dest`	Jump to `func`
`call func`	Push next instruction onto stack and jumps to `func`
`leave`	Carries out function epilogue
`ret`	Pops top of stack into `RIP` and jumps to address in `RIP`

Stack instructions are those which directly affect the stack. There are only two of these instructions: `push` and `pop`, which adds and removes data from the top of the stack (i.e., highest address), respectively [42].

3.6. Stack Memory

The stack is a contiguous array of memory location [40]. It is used by functions to store any local variables they may have, as well as information about how to return control to a calling function—the return address—in a structure called a 'stack frame'. Yurichev [43] describes the stack as a fundamental data structure in computer science as, without stacks, functions would not be able to call each other and, furthermore, recursion would be impossible.

Historically, the stack was also used to store any arguments passed to the function. However, this was changed under the x86-64 architecture; the first six arguments are now passed via registers.

Any arguments beyond the sixth are passed on the stack [39]. Further discussion of the calling procedure is in Section 3.4 System V Calling Convention.

On x86-64 the stack grows downwards towards lower memory addresses [39], using last-in first-out data organisation [44]. That is, when a function is called, space is made for it in the stack and its stack frame is placed after (at a lower address than) the calling function's stack frame; the stack's overall storage capacity needs increase by the size required for the new frame. When the function has finished executing, its stack frame is removed from the stack; this also means that the stack size shrinks.

As mentioned in the previous section, two GPRs are used by the stack—`RBP` and `RSP`. `RBP` serves as a static point for referencing stack-based information [45], such as local variables. It also points to the highest address in memory [46], in the context of the current stack frame. The `RSP` register always points to the top of the current stack frame (the lowest address) and is used as an operand in the calculation to allocate sufficient memory space for any local variables of a function.

Figure 4 provides a simplified view of a process memory layout where three nested functions have been called and three stack frames are stacked in the process stack.

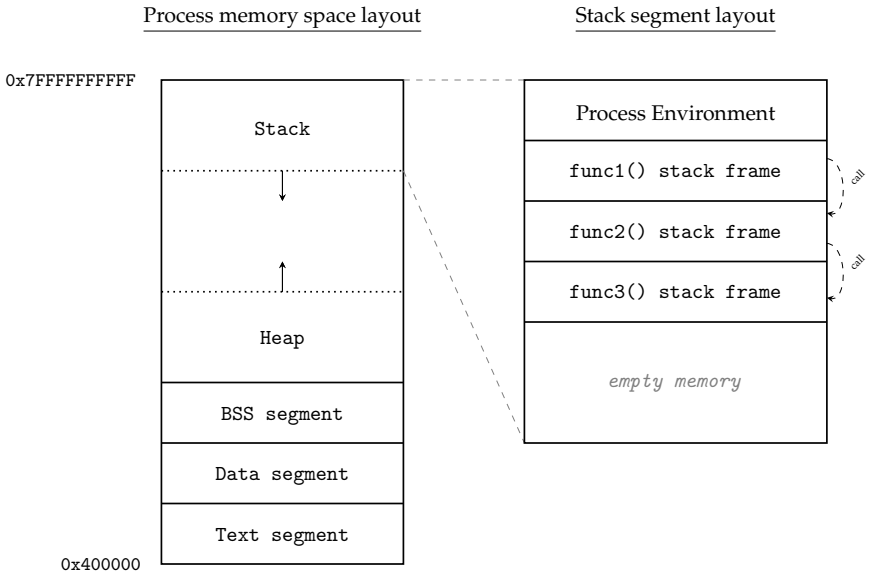

Figure 4. Process and stack layout after calling 3 nested functions.

A function that is called from another function is referred to as callee function, shown for example as `func2()` and `func3()` in Figure 4. On the other hand, a function that calls another function is referred to as caller function, shown for example as `func1()` and `func2()` in Figure 4. After each call, a new stack frame is created for the callee function and added to the stack at a lower memory address than the previous; note that the entire stack frame is not added to the stack at once and is, instead, created through register manipulation combined with assembly instructions.

After a `call`, the CPU needs to know where to continue execution from in the caller function when the callee function has finished executing. To do this, the address of the instruction immediately after the call instruction is pushed onto the stack. This is done by the caller function and is included in the functionality of the `call` instruction. In the callee function's stack frame, this will be the `return address` [7].

After the return address is pushed to the stack, the instruction pointer jumps to the address of the function and then a process known as the function prologue takes place. The function prologue is the process of creating a stack frame to hold callee function information. It is done by the callee function—the code to create the frame is located at the start of the callee function. There are three steps to the function prologue:

1. The current value of RBP is pushed onto the stack. This will allow the calling function's stack frame to be rebuilt after the callee function finish;
2. The current value of RSP is moved into RBP;
3. Space is allocated for any local variables. This is done by subtracting their collective size (in hexadecimal form) from RSP.

Similarly, there is a function epilogue which to get rid of the frame, clean up the space it occupied and return control to the calling function. The function epilogue also has three steps and is a simple reverse of the prologue:

1. RBP is moved into RSP;
2. RBP is popped from the stack;
3. The return address is read from the top of the stack (where RBP is pointing) and the instruction pointer jumps to that address.

It is important to note that at step 3, the instruction pointer will jump to whatever address is contained at the return address, be it the legitimate address to return to or not. In fact, this forms the basis of exploiting buffer overflows, as will be discussed in Section 4 Buffer Overflows. Figure 5 shows the func3 stack frame containing the local variables, the saved RBP and the return address.

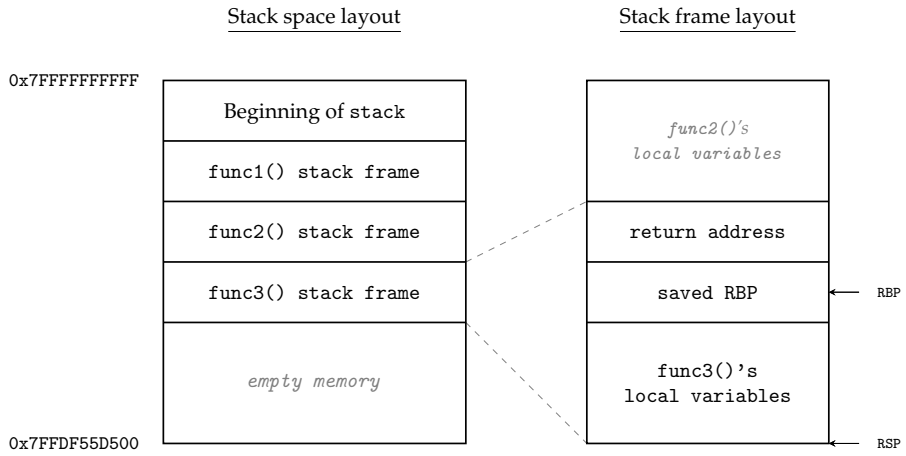

Figure 5. Layout of a stack frame.

4. Unsafe GLIBC Functions

A buffer overflow is the result of stuffing more data into a buffer than it can handle [7]. Although software defects causing buffer overflows are present in different parts of the software, the most common functions in Linux actually overwriting the memory are the functions handling memory in the GNU C Library, the GLIBC. This library is an extension of the C language, rather than being built into the language itself and is linked to C programs at compilation time. The library contains a huge collection of functions that enable a programmer to accomplish tasks—such as manage memory, receive input/send output and process strings—that are not included as built-in language functionality. These functions are declared across multiple header files (.h file format extension) that are included at the start of a C source code file (.c extension).

Unfortunately, this library contains numerous functions that are considered unsafe [47,48]; some modern compilers will warn the programmer during compilation that an unsafe function has been used.

Listing 3 shows an example of GNU Compiler Collection (GCC) detecting an unsafe function. The get_username() function uses gets(buff) to fill the buff from stdin.

Listing 3: GCC warning message: dangerous function detected.

```
$ gcc test.c -o test
/usr/bin/ld: /tmp/ccDjwSDi.o: in function ‘get_username’:
c.c:(.text+0x24): warning: the~‘gets’ function is dangerous and~should not be used.
```

GCC knows that the call to `gets(buff)` has no length about the buffer and therefore the `gets()` function cannot be check as to whether it is copying too many bytes or not and therefore overflows are possible. The C standard library contains other functions that are considered unsafe and should be avoided [47,49]. Table 3 lists the most important ones to avoid.

Table 3. Main unsafe functions included in the C standard library.

Function Signature	Description	Potential Problem
`strcpy(char *dest, const char *src)`	Copies string pointed to by `src` into buffer pointed to by `dest`	May overflow `dest`
`strcat(char *dest, const char *src)`	Appends string pointed to by `src` into buffer pointed to by `dest`	May overflow `dest`
`getwd(char *buf)`	Returns absolute path of current working directory	May overflow `buf`
`gets(char *s)`	Read a line from `stdin`, store into buffer pointed to by `s`	May overflow `s`
`fscanf(FILE *stream, const char *format)`	Reads file `stream`, formats according to `format` argument	May overflow arguments
`scanf(const char *format, ...)`	Reads from `stdin`, formats according to `format` argument	May overflow arguments
`realpath(char *path, char resolved_path[])`	Resolves symbolic links to `path` and writes canonical path to `resolved_path`	May overflow `path`
`sprintf(char *str, const char *format, ...)`	Formats a string according to `format` and writes formatted string to `str`	May overflow `str`

The first function listed, `strcpy`, is the most infamous for being the cause of buffer overflows. It copies a string, `src`, into a buffer, `dest`. However, there is no check to ensure that the `src` string is smaller, in length, than the `dest` buffer; it is up to the programmer to carry out this check and handle any errors appropriately, as is the modus operandi of the C language. The other unsafe functions can be exploited in a similar way in the sense that they also do not include the length as a parameter.

An updated version of `strcpy` exists, named `strncpy`, which attempts to prevent `strcpy`'s weaknesses. The `strncpy` function has been accepted by the C community and is included with the `GLIBC` library. This version of the function adds a third parameter, `n`. It operates in the same way as `strcpy`, except that it copies a specified amount of bytes, `n`, from `src` to `dest`. Although the `strcpy` copy could be stopped before if the source strings ends, under an attack the input length is controlled by attackers and will never happen.

Therefore, this does not make the function safe but pass to the developers responsibility to choose a proper value of `n`. If the developers incorrectly use a `n` value longer than the destination buffer then the overflow is still possible. Unfortunately, Miller and de Raadt [50] found that many programmers failed to grasp the subtleties of the API when trying to use this version and end up using it incorrectly.

A simple but illustrative example of using the unsafe `strcpy` function is showed in Figure 6. If the `src` string that is 24 bytes long is copied, using `strcpy`, into a `dest` buffer that only has 8 bytes of memory allocated for it, the `dest` buffer will be overflown. If the `dest` buffer is a local variable, it will be stored in the stack frame and the extra bytes from `src` will overwrite the saved frame pointer and, potentially, the return address of the stack frame. Note that the overflow is still possible when using `strncpy`, the safe version of the string copy memory.

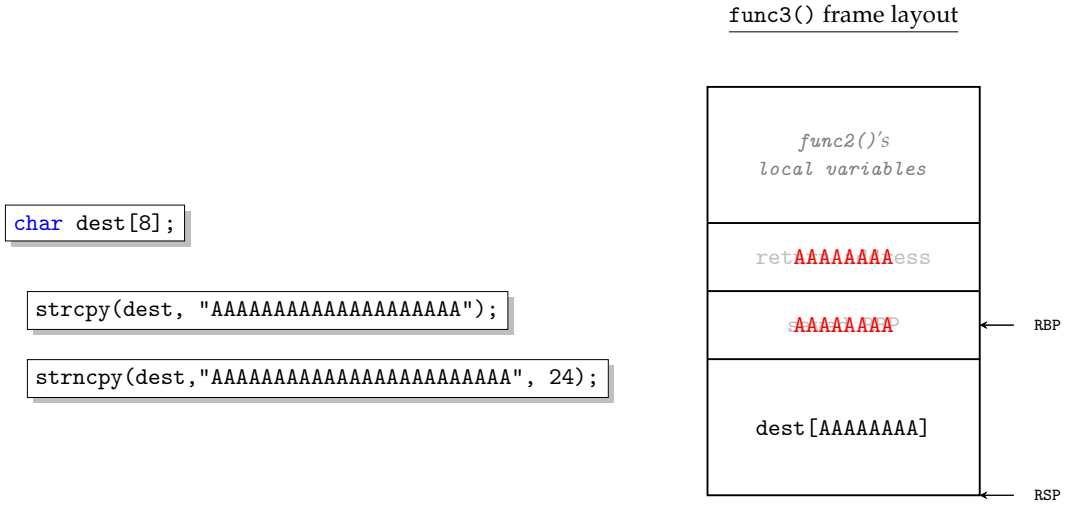

Figure 6. Buffer overflow using `strcpy()` and `strncpy` to overflow a buffer.

Although Figure 6 illustrates a simple example, real world applications contain similar bugs even when using safe functions. For example, instead of using a hard-coded 24 value, it is quite common to find code calculating the length of the source of the string using `strlen` and later use this variable as the third argument(n) of the safe `strncpy` function. If the length of the source of the string is longer than the destination buffer's size then the overflow will take place.

5. Attack Approaches

One of the first goals for attackers when probing for buffer overflow vulnerabilities is gaining the ability to overwrite the stack frame return address [51]. When it is possible to overwrite the return address of a stack frame, and an attacker does so, the CPU will jump to whatever address is stored in the return address when the function attempts to return to its caller [52].

5.1. Denial of Service

There are situations and bugs that do not allow attackers to arbitrarily overwrite the return address to redirect the execution flow to a desired location. In those situations, attackers cannot actually execute their desired code and they are limited to overwrite the return address to an invalid value.

This overwrite will cause the process to access to an invalid memory and the operating system will throw a `segmentation fault` error and halt the process's execution. By repeating this action, attackers can achieve a denial of service. This can be particularly devastating if the process provides a service that does not have a way of re-spawning.

5.2. Code Injection

On the other hand, if attackers have the capability to control the execution flow, they can try to inject and execute some code. Once the malicious code is executed, it allows an attacker to subsume their privilege level of the vulnerable process. If the process is running with root privileges, the attackers can spawn a shell and gain control of the entire system, free to upload or download data as they please, although this is just one of many things that can be done. The injected, malicious code is most commonly referred to as 'shellcode'. Listing 4 shows an example of shellcode that can be inserted into a vulnerable process to reboot a Linux x86-64 machine [53].

Listing 4: Code to reboot (`POWER_OFF`) a Linux x86-64 system.

```
char shellcode_reboot[] =
    "\xBA\xDC\xFE\x21\x43"
    "\xBE\x69\x19\x12\x28"
    "\xBF\xAD\xDE\xE1\xFE"
    "\xB0\xA9"
    "\x0F\x05";
```

The Listing 4 code is basically the assembler code of the `reboot(2)` system call. It requires three integer arguments:

1. The first argument is a `magic` number: `0x4321DEFC` moved to `EDX`.
2. The second argument is another `magic` number: `0x28121969` moved to `ESI`.
3. The third argument is another `magic` number: `0xFEE1DEAD` moved to `EDI`.
4. Finally, the `syscall` number `0xA9` for the `sys_reboot` moved to `AL`.

Tools like `rasm2` can be used to assemble instructions to opcode which greatly facilitates the task of converting from assembler to opcodes that will compose the shellcode. Listing 5 shows the `rasm2` output when converting from assembler instructions to the opcodes listed in Listing 4.

Listing 5: Reboot shellcode: From assembler to opcodes.

```
$ rasm2 -a x86 -b 64 'mov edx, 0x4321DEFC'
bafcde2143
$ rasm2 -a x86 -b 64 'mov esi, 0x28121969'
be69191228
$ rasm2 -a x86 -b 64 'mov edi, 0xFEE1DEAD'
b0a9
$ rasm2 -a x86 -b 64 'syscall'
0f05
```

In order for the injected shellcode to be executed, the attacker must redirect execution to where it ends up in memory. This can be done simply by changing the return address of a vulnerable function to the location of the shellcode in the stack. In this particular example, the shellcode is 19 bytes long so the attacker must craft the attack very precisely.

5.3. Return Orientated Programming

There are scenarios where injected code can not be executed because the system is protected against this. This protection is known as Non-eXecutable bit and is described in Section 6.1. In those scenarios the attackers need to find the shellcode assembler instructions in the already executing code. To achieve this, a very popular attacking technique known as Return orientated programming (ROP) is used.

Unlike code injection, ROP requires no code to be injected into the process and, instead, uses parts of codes that the process already has access to, "each of which ends in a "return" instruction" [18,52]. These small parts of codes are referred to as 'gadgets' and are rife within programs and libraries. In fact, Buchanan et al. [54] theorise that "(in the absence of code-flow integrity) any sufficiently large program codebase $\rightarrow$ arbitrary attacker computation and behaviour, without code injection"—that is to say, access to a large enough collection of code (the standard C library, for example) will result in an attacker being able to build a ROP exploit from the codebase [52].

The attack redirects the flow execution to the application itself or to any of its shared libraries to find those instructions or to execute different ones to create a compatible shellcode. That is, the attackers can execute selected assembler instructions (i.e., gadgets) to fill the registers into desired values to finally jump to a memory position containing the `syscall` assembler instruction.

To make use of gadgets, the attacker must first locate them within the code of the process. There have been many tools for automating this process, one of the most widely known is ROPgadget, which uses the Capstone disassembly engine to search a given binary file for snippets of assembly code that end with a `ret` instruction.

6. Memory Protection Techniques

This section examines the developments in memory protection to mitigate buffer overflow errors. We discuss the three most important protections techniques present in all modern operating systems: the Non-eXecutable bit (NX), the Stack Smashing Protector (SSP) and finally the Address Space Layout Randomisation (ASLR).

6.1. Non-Executable Bit (NX)

Non-eXectuable (NX) is a mitigation technique that seeks to thwart attacks that rely on executing injected code. To do this, memory regions are marked as writable or executable but not both at the same time. By doing this, attackers can inject code but they cannot execute it. This protection technique requires CPU support and most modern CPUs have NX support nowadays.

There is a software implementation of NX introduced by PaX [55], which implements NX on architectures where the Memory Management Unit (MMU) has no direct support. The NX protection mechanism is a very effective protection that prevents the execution of injected code but does not prevent injected data being used in return orientated programming attacks.

For example, attackers can exploit a stack buffer overflow by overwriting the return address on the stack but they also can keep overwriting more data on the stack where attackers can place arbitrary data. The overwrite of the return address will allow the attackers to redirect the control flow and the arbitrary data overwritten will assist ROP attacks. The attack idea is to use already exiting code in the application, named "ROP gadgets", to control the program state in order to execute arbitrary code.

Therefore attackers just need to know the code being executed and do some offline preprocessing to find the ROP gadgets that are equivalent to the execution of the injected code. Recent attacks such as `offset2lib` [15] and `return-to-csu` [13] require to bypass the NX protection technique to have success. Later, we describe how to fully bypass the NX protection with a minimal C program, demonstrating than even a simple "hello world" application has enough ROP gadgets.

6.2. Stack Smashing Protector (SSP)

First introduced in 1997 by Immunix Inc., StackGuard is an extension of the GNU Compiler Collection (GCC) that aims to mitigate the effectiveness of buffer overflow attacks with only modest performance penalties [20].

In the first versions of StackGuard, to accomplish mitigation, a canary value was inserted next to the return address of the current stack frame to prevent an attacker from overwriting the return address. The canary value is checked before the instruction pointer loads the return address of the stack frame. If the canary value is altered, the processor knows that an attack has been attempted and execution is aborted. Figure 7 shows a stack frame protected with the SSP.

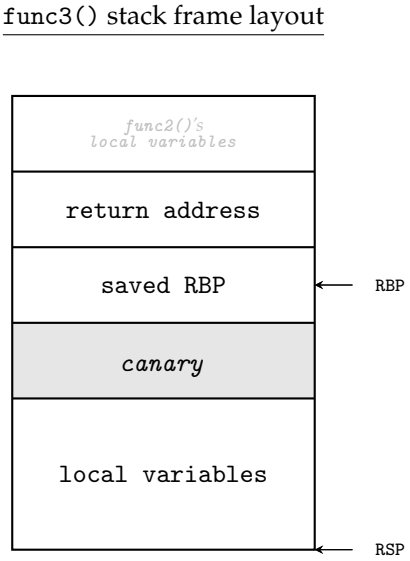

Figure 7. Layout of a stack frame with the SSP technique.

The canary value is placed on the stack, just below the saved registers by the function prologue code. StackGuard includes three potential canary value types: random XOR, random and terminator [56,57]:

Random XOR: Generated by performing a bitwise exclusive OR operation on a randomly generated canary with some or all [57] of the information on the stack used to return to a calling function, such as the return address.
Terminator: This type of canary takes advantage of the fact that strings end with a terminator value and that most stack buffer overflows involve string operations [57]. By using a canary that contains terminator values, the exploit will (should) fail as the attacker cannot write the terminator character sequence for the particular string operation being used to memory and then continue writing [57]. For example, if 1 byte of the canary is `0x00` then `strcpy` and `strncpy` library functions will stop copying after copying the `0x00` and therefore no memory beyond the canary will be overwritten.
Random: The canary is a randomly generated 64 bit value. This is the canary generation approach followed by the standard GLIBC. This could slightly vary depending on the architecture, for example, in x86_64, 7 bytes are fully random but one is `0x00`, a terminator to stop overflows from `str*-like` library functions [58].
RenewSSP: It is a modification of the Stack Smashing Protector (SSP) technique that can be applied to forking servers applications to prevent brute force attacks [19]. The technique changes the reference canary every time a new child process is created. This in practice means that each child process has a different random canary and therefore brute force attacks are not longer possible.

Unfortunately the Stack Smashing Protector (SSP) is not perfect and it can be bypassed. The SSP can not detect overflows but actually it detects whether the canary value has changed before the function returns. Therefore if the vulnerability allows to perform an overflow without overwriting the canary value then the SSP will not detect any overflow and the attack will success. Although this scenario is not very common, it is plausible [59].

The second approach to bypass the SSP is to find another vulnerability in the target application that allows attackers to perform info leak [60] attacks. Using info leaks to obtain the canary value, attackers can create a payload to overflow the buffer writing the expected canary value. The SSP will not detect any attack because the overwritten canary is the same and therefore the attack will not be detected.

The third approach to bypass SSP is to guess the canary value. Attackers can always try to guess the canary value and if the canary value guessed does not match, the SSP will send a signal to kill its own process. On remote servers, with multiple clients, the SPP will kill the process associated to the attackers' connection. However, if the target application is a forking server where a parent process launches children processes to attend clients, attackers can perform brute force attacks. Those attacks are much more efficient that just guessing [61]. Brute force attacks are possible because children always inherit the canary value from its parent and therefore the canary value remains always the same. This allows attackers to discard previously guessed values. Note that this is different from the scenario where a server is re-launched because the canary value is changed every time. This scenario is named trial and test attacks [61], and attackers cannot perform brute force attacks.

Bypassing the SSP using the brute force attacks are still being used in modern attacks because the SSP is a barrier widely used to protect applications. For example, the modern return-to-csu [13] and offset2lib [15] attacks bypasses the NX, ASLR and SSP protection techniques and both perform a SSP byte-for-byte brute force attack.

6.3. Address Space Layout Randomisation (ASLR)

Address space layout randomisation (ASLR) is a protection technique that attempts to render exploits that depend on predetermined memory addresses useless [56].

It is a protection technique that which the memory address layout to prevent attacks that relies on knowing the location of an application's memory map. Similarly to the source code analysis tools [62], the ASLR does not increase the security by removing vulnerabilities from the system but it makes more difficult to exploit existing vulnerabilities.

When a process is loaded into memory, it is mapped to an address space in virtual memory. When ASLR is enabled in the operating system, the different parts of the process, such as shared libraries, stack or HEAP are placed at random addresses in virtual memory [61]. By doing so, attacks that rely on knowing the memory locations of the application or libraries to conduct the attack will simplyfail [63].

The effectiveness of ASLR's mitigation is reliant on the amount of randomness that can be applied, a.k.a the entropy [64]. With more randomness comes the increased unlikeliness that an attacker will be able to guess an address via brute-force attacks. On 32 bit systems, this entropy is worryingly limited [15,22] because only 8 bits of an address are randomised, resulting in $2^8 = 256$ possible variations. However, on 64 bit systems, 28 bits of an addresses are randomised, resulting in $2^{28} \approx 268.5$ million possible variations. Brute-forcing an ASLR-enabled 64 bit system address is impractical for an attacker [65].

PaX ASLR implementation has the added benefit of being able to apply different amounts of randomisation [23] to these areas, as the base addresses used to map each group are not related. PaX ASLR uses 40 bits of a memory address for entropy [14,15], increasing the randomness and the time it would take for an attacker to brute-force the address. This, compared with the regular 28 bits of entropy under the vanilla Linux ASLR implementation, increases the entropy by over 4000×.

Listing 6 shows the three top memory regions addresses of two executions of Firefox. Because the ASLR is disabled in both executions, the memory regions addresses are kept across multiple executions and therefore can be pretested by attackers. However, when the ASLR is on, as shown in Listing 7, the addresses where the three memory regions are loaded are different and therefore attackers will not be able to predict the addresses and their exploits will fail.

Listing 6: ASLR disabled in two executions of Firefox.

```
  ASLR OFF (execution 1)        ASLR OFF (execution 2)
  ----------------------       -----------------------
7ffff7c2a000-7ffff7c4c000 || 7ffff7c2a000-7ffff7c4c000 r--p /usr/lib/x86_64-linux-gnu/libc-2.28.so
7ffff7fcb000-7ffff7fcc000 || 7ffff7fcb000-7ffff7fcc000 r--p /usr/lib/firefox-esr/libmozgtk.so
7fffffffde000-7ffffffff000 || 7fffffffde000-7ffffffff000 rw-p [stack]
```

Listing 7: ASLR enabled in two executions of Firefox.

```
   ASLR ON (execution 1)      ASLR ON (execution 2)
   ---------------------    ----------------------
7f494b86d000-7f494b88f000 || 7f8e4df95000-7f8e4dfb7000 r--p /usr/lib/x86_64-linux-gnu/libc-2.28.so
7f494bc0e000-7f494bc0f000 || 7f8e4e336000-7f8e4e337000 r--p /usr/lib/firefox-esr/libmozgtk.so
7ffef630d000-7ffef632e000 || 7ffd7e410000-7ffd7e431000 rw-p [stack]
```

However, the same attacks developed to bypass the NX protection, such as "offset2lib" and "return-to-csu" [13,15] can be used to bypass the ASLR. The attack takes advantage of the way Linux stores ASLR-enabled objects in memory. Authors were able to predict where those objects will be randomized in execution time by discovering that the relative distance between the executable and the libraries was always a constant.

By leaking an address belonging to the application it is possible to find out where the libraries are stored in memory [66]. This is due to the fact that the offset of the process to the libraries loaded above it (at a higher address in memory) remains constant throughout each process instance. Once the library address is known, a ROP gadget chain can be built from gadgets contained within the library, which will further defeat NX. Recently, another attack named `return-to-csu` [13], showed that even with a minimal C program there is enough gadgets to fully bypass the ASLR.

6.4. Effectiveness Summary

Besides the direct exploitation of the buffer overflow, we need to consider the presence of different attack vectors. Table 4 shows a summary of the main protection techniques and whether a particular protection technique can prevent the attack, where; **High:** the technique provides good protection; **Med:** the technique provides some protection but in some scenarios fails; **Low:** the technique provides no practical protection and **- :** the technique does not apply or provide protection.

Table 4. Main protection techniques effectiveness.

Protection Technique	Brute Force	Ret2-*	ROP
Non-eXecutable (NX)	-	Low	Low
Stack Smashing Protector (SSP) [XOR/Terminator/Random]	Med	-	-
Address Space Layout Randomization (ASLR)	Med	Low	High
Address Space Layout Randomization Next-Generation (ASLR-NG)	High	Low	High
Renew Stack Smashing Protector (RenewSSP)	High	-	-
Stack Smashing Protector for Mobile Devices (SSPFA)	High	-	-

As Table 4 shows, there is no protection mechanism that can provide effective protection against all forms of attacks. The NX [61] is considered completely defeated by ret2-* [16] and ROP attacks [15]. NX is considered as complementary protection but provides no protection on its own. The SSP is an effective protection against stack buffer overflows but it is not a mechanism to protect memory errors in general [13,61]. For example, it can deter full brute force attacks but not byte-for-byte attacks. However, the renewSSP is able to fully prevent brute force attacks against the stack smashing protector [19]. The SSPMD, a renewSSP modification for Android Operating systems, prevents all kind of brute force attacks against the SSP of individual Android applications [21].

On the other hand, ASLR is a more generic technique that can help to mitigate wider forms of attack [14,15,61]. ROP attacks require to know the code location to be bypassed and ASLR randomizes the memory layout, therefore having both techniques provide a security level higher than the sum of protection level provided individually. ASLR-NG provides stronger security since it provides more absolute and relative entropy which removes classical attacks but also attacks that exploit the correlation between virtual memory objects to de-randomize libraries [15].

7. Conclusions

In this paper, we have highlighted the importance of memory error vulnerabilities and more specifically stack buffer overflows. We have identified the root causes that make those attacks possible on modern x86-64 architecture.

We have analyzed how unsafe library functions are prone to buffer overflows, revealing that although there are secure versions of those functions, they are not actually preventing buffer overflows from happening. In fact, if they are incorrectly used, attackers can exploit them in a similar way as when unsafe functions are employed.

Therefore, using secure functions does not result in software free from vulnerabilities and it requires developers to be security-aware. Furthermore, analysis of the three main security protection techniques present in all modern operating system; the non-eXecutable bit (NX), the Stack Smashing Protector (SSP) and the Address Space Layout Randomization (ASLR), concluded that although they provide a strong level of protection against classical exploitation techniques, unfortunately, recent advanced attacks have demonstrated effective approaches to bypass them.

For the future, novel protections techniques will need to effectively protect against memory errors exploitation. The techniques should focus on attack approaches and they must be effective against attacks vectors while also fully back compatible with old, current and future applications. Since many protection techniques could coexist, they should introduce very little overhead to be accepted in the security community.

Author Contributions: Writing—original draft, C.P., H.M.-G. and C.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: Authors declare no conflict of interest.

References

1. Younan, Y. 25 Years of Vulnerabilities: 1988–2012; *Sourcefire*, 2013. Available online: https://maxedv.com/wp-content/uploads/2011/12/Sourcefire-25-Years-of-Vulnerabilities-Research-Report.pdf (accessed on 12 February 2019).
2. Meer, H. Memory Corruption Attacks: The (almost) Complete History. 2010. Available online: https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf (accessed on 12 February 2019).
3. Anderson, P.J. *Computer Security Technology Planning Study*; Deputy for Command and Management Systems HQ Electronic Systems Division (AFSC) Technical Report; 1972; Volume 2. Available online: http://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf (accessed on 12 February 2019).
4. Fowler, S. CVE-2019-3822 curl: NTLMv2 type-3 Header Stack Buffer Overflow. 2019. Available online: https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2019-3822 (accessed on 5 March 2019).
5. curl. 2019 Available online: https://curl.haxx.se/libcurl/features.html (accessed on 5 March 2019).
6. Stenberg, D. *Everything Curl*; GitBook: Lyon, France, 2015.
7. Aleph One. Smashing the Stack for Fun and Profit. *Phrack* **1996**, *7*.
8. Cowan, C.; Pu, C. Death, Taxes and Imperfect Software: Surviving the Inevitable. In Proceedings of the 1998 Workshop on New Security Paradigms, Charlottsville, VA, USA, 22–25 September 1998; pp. 54–70, doi:10.1145/310889.310915.
9. McConnell, S. *Code Complete*, 2nd ed.; Microsoft Press: Redmond, WA, USA, 2004.
10. Younan, Y.; Pozza, D.; Piessens, F.; Joosen, W. Extended protection against stack smashing attacks without performance loss. In Proceedings of the ACSAC, Shanghai, China, 6–8 September 2006.
11. Bulba; Kil3r. Bypassing StackGuard and StackShield. *Phrack* **2002**, *10*, 56.
12. Richarte, G. Four different tricks to bypass StackShield and StackGuard protection. *World Wide Web*. 2002 Available online: https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf (accessed on 12 February 2019).

13. Marco-Gisbert, H.; Ripoll-Ripoll, I. *Return-to-csu: A New Method to Bypass 64-bit Linux ASLR*; Black Hat. 2018 Available online: https://i.blackhat.com/briefings/asia/2018/asia-18-Marco-return-to-csu-a-new-method-to-bypass-the-64-bit-Linux-ASLR-wp.pdf (accessed on 12 February 2019).
14. Shacham, H.; Page, M.; Pfaff, B.; Goh, E.J.; Modadugu, N.; Boneh, D. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, Washington, DC, USA, 25–29 October 2004; pp. 298–307, doi:10.1145/1030083.1030124.
15. Marco-Gisbert, H.; Ripoll-Ripoll, I. On the Effectiveness of Full-ASLR on 64-bit Linux. In Proceedings of the In-Depth Security Conference (DeepSec), Vienna, Austria, 18–21 November 2014.
16. Tran, M.; Etheridge, M.; Bletsch, T.; Jiang, X.; Freeh, V.; Ning, P. On the expressiveness of return-into-libc attacks. In Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, Menlo Park, CA, USA, 20–21 September 2011; pp. 121–141, doi:10.1007/978-3-642-23644-0_7.
17. Wojtczuk, R. The advanced return-into-lib(c) exploits: PaX case study. *Phrack* **2001**, *58*. Available online: http://phrack.org/issues/58/4.html (accessed on 12 February 2019).
18. Roemer, R.; Buchanan, E.; Shacham, H.; Savage, S. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* **2012**, *15*, 2:1–2:34, doi:10.1145/2133375.2133377.
19. Marco-Gisbert, H.; Ripoll, I. Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers. In Proceedings of the 12th International Symposium on Network Computing and Applications, Cambridge, MA, USA, 22–24 August 2013; pp. 243–250. doi:10.1109/NCA.2013.12.
20. Cowan, C.; Pu, C.; Maier, D.; Walpole, J.; Bakke, P.; Beattie, S.; Grier, A.; Wagle, P.; Zhang, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, 26–29 January 1998.
21. Marco-Gisbert, H.; Ripoll-Ripoll, I. SSPFA: Effective stack smashing protection for Android OS. *Int. J. Inf. Secur.* **2019**, *18*, 519–532, doi:10.1007/s10207-018-00425-8.
22. Marco-Gisbert, H.; Ripoll-Ripoll, I. Address Space Layout Randomization Next Generation. *Appl. Sci.* **2019**, *9*, 2928.
23. Pax Team. PaX Address Space Layout Randomization (ASLR). 2003. Available online: http://pax.grsecurity.net/docs/aslr.txt (accessed on 17 July 2019).
24. Paulson, L.D. New Chips Stop Buffer Overflow Attacks. *Computer* **2004**, *37*, 28–30.
25. Edgecombe, R. Touch But Don'T Look: Running the Kernel in Execute Only Memory. In Proceedings of the Linux Plumbers Conference, Lisbon, Portugal, 9–11 September 2019. Available online: https://linuxplumbersconf.org/event/4/contributions/283/attachments/357/588/Touch_but_dont_look__Running_the_kernel_in_execute_only_memory-presented.pdf (accessed on 17 July 2019).
26. Xu, J.; Kalbarczyk, Z.; Iyer, R. Transparent runtime randomization for security. In Proceedings of the 22nd International Symposium on Reliable Distributed Systems, Florence, Italy, 6–8 October 2003; pp. 260–269, doi:10.1109/RELDIS.2003.1238076.
27. Zhan, X.; Zheng, T.; Gao, S. Defending ROP Attacks Using Basic Block Level Randomization. In Proceedings of the 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion (SERE-C), San Francisco, CA, USA, 30 June–2 July 2014; pp. 107–112, doi:10.1109/SERE-C.2014.28.
28. Kil, C.; Jim, J.; Bookholt, C.; Xu, J.; Ning, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In Proceedings of the Computer Security Applications Conference, Miami Beach, FL, USA, 11–15 December 2006; pp. 339–348.
29. Iyer, V.; Kanitkar, A.; Dasgupta, P.; Srinivasan, R. Preventing Overflow Attacks by Memory Randomization. In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), San Jose, CA, USA, 1–4 November 2010; pp. 339–347, doi:10.1109/ISSRE.2010.22.
30. Raadt, T.D. Exploit Mitigation Techniques (Updated to Include Random Malloc and MMAP). In Proceedings of the OpenCON 2005, Venice, Italy, 5–6 November 2005.
31. Miller, K. *OpenBSD's Position Independent Executable (PIE) Implementation*; In Proceedings of the NYCBSDCon, New York, NY, USA, 11–12 October 2008.
32. Russinovich, M. *Inside the Windows Vista Kernel: Part 3*; Microsfot, 2007 Available online: https://docs.microsoft.com/en-us/previous-versions/technet-magazine/cc162458(v=msdn.10) (accessed on 17 July 2019).

33. Whitehouse, O. *An Analysis of Address Space Layout Randomization on Windows Vista*; Technical Report; Symantec Advanced Threat Research, Black Hat, 2007. Available online: https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf (accessed on 12 February 2019).
34. Ruoho, C. *ASLR: Leopard Versus Vista*; Laconic Security: Broomfield, CO, USA, 2008
35. 'xorl'. Linux GLibC Stack Canary Values **2010**.
36. Bittau, A.; Belay, A.; Mashtizadeh, A.; Mazières, D.; Boneh, D. Hacking Blind. In Proceedings of the 35th IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014. Available online: http://www.ieee-security.org/TC/SP2014/papers/HackingBlind.pdf (accessed on 17 July 2019).
37. *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*; AMD, 2017. Available online: https://www.amd.com/system/files/TechDocs/24592.pdf (accessed on 17 July 2019).
38. ISO/IEC. Working Draft, Standard for Programming Language C++ [Online, C++ International Standard, N4800]. Available online: https://www.iso.org/standard/74528.html (accessed on 18 February 2019).
39. Matz, M.; Hubička, J.; Jaeger, A.; Mitchell, M. System V Application Binary Interface. 2014 Available online: https://uclibc.org/docs/psABI-x86_64.pdf (accessed on 17 July 2019).
40. Intel. Intel 64 and IA-32 Architecture Software Developer's Manual. 2016. Available online: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (accessed on 17 July 2019).
41. Kerrisk, M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*; No Starch Press: San Francisco, CA, USA, 2010.
42. Anley, C.; Heasman, J.; Lindner, F.; Richarte, G. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, 2nd ed.; John Wiley & Sons, Inc.: New York, NY, USA, 2007.
43. Yurichev, D. Reverse Engineering for Beginners (Understanding Assembly Language). 2018. Available online: https://yurichev.com/writings/RE4B-EN.pdf (accessed on 17 July 2019).
44. Eilam, E. *Reversing: Secrets of Reverse Engineering*; Wiley Publishing, Inc.: Hoboken, NJ, USA, 2005.
45. Foster C., J.; Osipov, V.; Bhalla, N.; Heinen, N. *Buffer Overflow Attacks: Detect, Exploit, Prevent*; Syngress Publishing Inc.: Rockland, MA, USA, 2005.
46. Weidman, G. *Penetration Testing: A Hands-On Introduction to Hacking*; William Pollock: San Francisco, CA, USA, 2014.
47. Baratloo, A.; Singh, N.; Tsai, T. Transparent Run-time Defense Against Stack Smashing Attacks. In Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, USA, 18–23 June 2000.
48. Wheeler A.D. Secure Programming HOWTO. 2015. Available online: https://www.tldp.org/HOWTO/pdf/Secure-Programs-HOWTO.pdf (accessed on 17 July 2019).
49. Gustedt, J. *Modern C*; Manning Publications: Helter Island, NY, USA, 2016.
50. Miller C., T.; Raadt de, T. strlcpy and strlcat—Consistent, Safe, String Copy and Concatenation. In Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, Monterey, CA, USA, 6–11 June 1999.
51. Kc S., G.; Keromytis D., A. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. In Proceedings of the 21st Annual Computer Security Applications Conference, Tucson, AZ, USA, 5–9 December 2005.
52. Sayeed, S.; Marco-Gisbert, H.; Ripoll-Ripoll, I.; Birch, M. Control-Flow Integrity: Attacks and Protections. *Appl. Sci.* **2019**, *9*, 4229.
53. "zbt". Linux/x86-64—reboot(POWER_OFF)—19 Bytes. Available online: http://shell-storm.org/shellcode/files/shellcode-602.php (accessed on 16 February 2019).
54. Buchanan, E.; Roemer, R.; Savage, S.; Shacham, H. Return-Orientated Programming: Exploitation without Code Injection. *Black Hat*, 2008. Available online: https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf (accessed on 12 February 2019).
55. PaX. NOEXEC. 2003 Available online: https://pax.grsecurity.net/docs/noexec.txt (accessed on 12 February 2019).
56. Silberman, P.; Johnson, R. A Comparison of Buffer Overflow Prevention Implementations and Weaknesses. *Black Hat*, 2014. Available online: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf (accessed on 12 February 2019).
57. Wagle, P.; Cowan, C. StackGuard: Simple Stack Smash Protection for GCC. In Proceedings of the GCC Developers Summit, Ottawa, ON, Canada, 25–27 May 2003.

58. Cowan, C.; Beattie, S.; Pu, C.; Wagle, P.; Walthinsen, E. *Protecting Systems from Stack Smashing Attacks with StackGuard*; Institute of Science & Technology: Hillsboro, Or, USA, 1999.
59. Marco, H. Root Shell on Sniffit. Available online: http://hmarco.org/bugs/CVE-2014-5439-sniffit_0.3.7-stack-buffer-overflow.html (accessed on 25 May 2020).
60. Huawei Technologies Co., L. Information Leak Vulnerability in Some Huawei Products. Available online: https://www.huawei.com/en/psirt/security-advisories/huawei-sa-20191030-01-phone-en (accessed on 25 May 2020).
61. Marco-Gisbert, H.; Ripoll-Ripoll, I. *On the Effectiness of NX, SSP, RenewSSP and ASLR against Stack Buffer Overflows*; IEEE: New York, NY, USA, 2014.
62. Jelinek, J. Object Size Checking to pRevent (Some) Buffer Overflows (GCC FORTIFY). 2004. Available online: http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html (accessed on 17 July 2019).
63. Kc S., G.; Keromytis D., A.; Prevelakis, V. Countering Code-Injection Attacks With Instruction-Set Randomization. In Proceedings of the 10th ACM Conference on Computer and Communications Security, Washington, DC, USA, 27–31 October 2003.
64. Marco-Gisbert, H. Cyber-Security Protection Techniques to Mitigate Memory Errors Exploitation. Ph.D. Thesis, Universitat Politècnica de València, València, Spain, 2015.
65. Gras, B.; Ravazi, K.; Bosman, E.; Bos, H.; Giuffrida, C. *ASLR on the Line: Practical Cache Attacks on the MMU*; Network and Distributed System Security Symposium (NDSS): San Diego, CA, USA, 2017.
66. Göktas, E.; Athanasopoulos, E.; Bos, H.; Portokalidis, G. Out of Control: Overcoming Control-Flow Integrity. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014.