Binary Exploitation on Modern Computer Systems

To what extent is binary exploitation feasible despite protections on modern computer systems?

Extended Essay
Subject: Computer Science
Session: May 2020
Author: Aaron Esau
Word Count: 3857

Contents

Binary Exploitation on Modern Computer Systems

1 Introduction

Within the realm of computer security, binary exploitation vulnerabilities involve weaknesses in binary (i.e. executable) programs rather than web applications, cryptographic functions, or other high-value targets. Binary exploitation vulnerabilities tend to exist more in programs written in lower-level (i.e. less abstract, more intentional, and often memory-unsafe) programming languages such as C, C++, and Assembly, although they can exist in some higher-level languages such as JavaScript [1].

At its core, binary exploitation is about manipulating an existing program into executing arbitrary code or escalating privileges. Because of the potential impact, these types of vulnerabilities are very valuable to attackers; an attacker could use a binary exploitation vulnerability to execute code on remote servers because of their exposure to the internet. Or, an attacker could abuse a vulnerability to take over victims' computers simply because they visit the wrong website and their web browser is not securely-written. It could even be used for local privilege escalation, from an unprivileged user to an administrator. Once an attacker can execute arbitrary code on a system, that attacker has the potential to anything to it.

As these types of vulnerabilities have such a huge impact on availability of important applications, and the integrity and confidentiality of sensitive data, programming language and operating system developers have spent a significant amount of time attempting to implement protections to prevent binary exploitation. These types of vulnerabilities were more common in the late 1990's before programming language and operating system developers began to develop various protections [2]. However, protections merely fix the symptoms and do not fix the root cause; vulnerable code is vulnerable whether or not protections are implemented to make exploitation difficult. Given a reasonable amount of time, all protections can be bypassed. Therefore, it is reasonable to argue that these protections are insufficient, and binary exploitation is still feasible in modern times despite the safeguards.

2 Background

Binary exploitation is arguably the most conceptually challenging type of software exploitation to learn because it involves very low-level memory management and central processing unit (CPU) architectural concepts. To fully understand the threat and protections against it, it is necessary to first have a firm understanding of the fundamental concepts of how computers are organized.

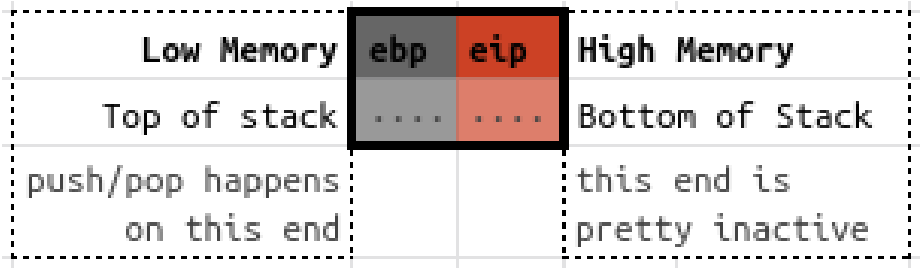

Figure 1: Stack layout

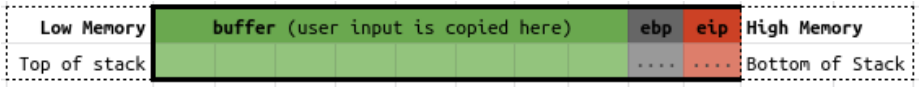

Figure 2: Allocating buffers on the stack

2.1 Low Level Concepts

2.1.1 Memory Page

A memory page is simply a dedicated, contiguous region in a computer's virtual memory that has a specific purpose (e.g. to function as a stack or heap).

2.1.2 Stack

The stack is a memory page in which a program can store data at runtime. It operates as a last-in-first-out (LIFO) list—items are added at the top of the stack ("pushed") and also removed ("popped") from the top. The most recent item that was pushed to the stack is always the first item to be removed during a pop operation. The bottom of the stack is the highest memory address associated with the stack, and the top is the lowest address. Counterintuitively, when a value is pushed to the stack, it is added to the lowest memory address associated with the stack (the top of the stack); that is, the stack grows downward [5] (in terms of memory addresses, as seen in Figure 1 where the left-hand side represents lower memory addresses).

2.1.3 Buffers

Programs can allocate buffers, dedicated chunks of memory, on the stack, heap, or global data section (*bss*). If the buffer is allocated on the stack, it is always higher on the stack than the return pointer (Figure 2).

Programs often temporarily store strings read from user input or from a file by first allocating a certain number of bytes on the stack to be dedicated to a buffer on the stack, then reading into the buffer the same number of bytes from the user as fits in the buffer. If a program read more than could fit in an allocated stack-based buffer, it would overwrite other items on the stack.

2.1.4 Pointers

A pointer is a value that stores the address of another value in memory. Pointers can be stored anywhere in memory or in a register. They are used for low-level operations such as function calls, stack frame shifting, and memory allocation.

2.1.5 Registers

A register is a small, quickly-accessible storage location that is physically inside the CPU as opposed to RAM. There are two types of registers: general-use registers and registers that have specific uses. The registers can be accessed more quickly than any memory location because they are physically located inside the CPU. Because there are few general-use registers, programs move data that they need to store for longer periods of time somewhere in memory—often the stack. Hence, the general-use registers are used for—among other things—temporarily storing data for a program.

There are many specific-use registers, although the most important register for binary exploitation is the instruction pointer register (*ip*), which is called *eip* on 32-bit systems and *rip* on 64-bit systems. The instruction pointer is read-only and always points to the instruction that will be executed next by the CPU.

The CPU does not care what the instruction pointer points to; it does not have a conscience. It does not have to execute legitimate code. In fact, it can point to any piece of valid (i.e. allocated and readable) or invalid memory, and the CPU will begin execution as if it were code.

If the CPU has an issue accessing the memory the instruction pointer points to, it will interrupt code execution and kill the program (a segmentation fault or "segfault") [3]. To avoid reaching this condition, programs try to keep control of what the CPU executes so the output is intelligible and the program runs properly by carefully managing the instruction pointer and the stack.

When a function is called, the current value of the instruction pointer register (i.e. the address of the instruction that will be executed next), called the return pointer (represented by the red field in Figure 2), is pushed to the stack. The CPU then updates the instruction pointer register with the address of the function that was called. However, it is important to understand that the program itself does not directly update the register. Instead, the program tells the CPU to execute a jump instruction (*jmp*), and the CPU updates the register [4]. Once the function that was called finishes executing, the program tells the CPU to execute a return instruction (*ret*), and the CPU knows to pop the old return pointer from the top of the stack and into the instruction pointer register. Then, code execution resumes as before.

Two other specific-use registers are the base pointer register (*bp*), called *ebp* on 32-bit systems and *rbp* on 64-bit systems, and the stack pointer register (*sp*), called *esp* on 32-bit systems and *rsp* on 64-bit systems. The base pointer points to the highest memory address associated with the stack (the bottom of the stack). Likewise, the stack pointer points to the lowest memory address (the top of the stack). A stack frame is the relative positions of the base pointer and stack pointer during a function call.

Similar to how the CPU handles the instruction pointer register, it saves the stack frame when a function is called by pushing the base pointer to the

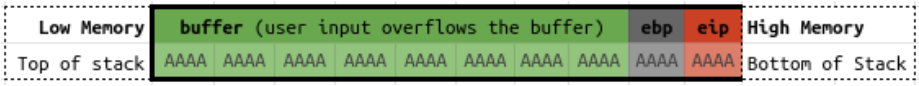

Figure 3: Filling up the buffer and overwriting pointers

top of the stack. Then, the CPU sets the value of the base pointer register to the value of the stack pointer register. At this point, there is no data stored in the current stack frame for the function call (i.e. no data between the stack and base pointers in memory), because the base pointer and stack pointer are pointing to the same location in memory. When a function returns using the *ret* instruction, the stack pointer register is set to the value of the base pointer register, and the original base pointer is popped back off the stack and into the base pointer register.

2.2 Attacks and Vulnerabilities

The goal of binary exploitation is to manipulate an existing program into doing what an attacker wants—which is, almost always, redirecting code execution. The method by which an attacker can achieve a goal such as code execution redirection is called an attack (or exploit vector). An exploit is the intentional act of performing an attack. Finally, types of vulnerabilities are characterized by their potential impacts and the method of abuse.

2.2.1 Buffer Overflow Attacks

Often, code execution redirection entails overwriting the return pointer that is stored on the stack and pointing it to code that is controlled by the attacker, which is also often on the stack [6]. Attacker-controlled code on the stack (or another executable memory page) is known as shellcode. Once the attacker can control where the CPU is executing instructions from, the only challenge is to determine the beginning of the arbitrary attacker-controlled code.

If a vulnerable function is called which reads more data from the attacker than can fit in a buffer that it allocated on the stack, then the attacker could potentially fill up and write past the buffer (as shown in Figure 3 where the attacker filled up the buffer with the *A* character and overwritten the base pointer and return pointer).

The value that is stored after the buffer in memory from the function call is the return pointer, which points to the location that the CPU should jump to once execution of the function completes. The CPU will execute code anywhere in memory where it is told. Once execution of the vulnerable function is complete, the CPU will pop the attacker-overwritten return pointer from of the stack and back into the instruction pointer register, and hence the attacker will have control over the flow of code execution. This type of attack, under the category of binary exploitation, is appropriately named a buffer overflow attack because the buffer is filled up and data is written past the buffer [6].

2.3 Impact of Vulnerabilities

There are many types of vulnerabilities that could enable an attacker to write arbitrary data to arbitrary locations in memory. The obvious impact of these vulnerability types is that attackers would be able to overwrite the return pointer and redirect code execution in the same way that a buffer overflow attack does.

In addition, there are many vulnerability types that allow an attacker to read values from arbitrary locations in memory [7]. While vulnerabilities that may enable an attacker to leak sensitive information such as passwords or private encryption keys may at first seem relatively harmless compared to vulnerabilities that enable attackers to redirect code execution, they have a huge impact on the attacker's ability to bypass protections that require a secret (e.g. a stack canary or memory offset for ASLR) to be known, as explained later.

3 Preventing Buffer Overflow Attacks using Stack Canaries

3.1 Introduction

In an attempt to stop buffer overflow attacks, programming language developers created a protection strategy called StackGuard [8]. In this protection, a randomly-generated, secret value called a stack canary is pushed to the stack immediately after the return pointer is pushed when calling a function. Its value is checked against before returning from the function, and if it does not match its original value, the program immediately exits—preventing the return pointer from being popped into the instruction pointer register and the flow of code execution from being changed. When the return pointer is overwritten during a buffer overflow attack, the stack canary is between the buffer and the return pointer, so it is overwritten as well. Thus, to exploit a buffer overflow vulnerability, an attacker must first know the value of the stack canary.

3.2 Bypass

However, an attacker's goal using a buffer overflow attack may not be to redirect code execution. For example, if a variable that determined a user's privilege levels is stored on the stack after a buffer but before the return pointer, the attacker may want to only overwrite the variable to escalate privileges. And, as the stack canary is placed immediately before the return pointer, the canary may not be overwritten.

Also, given a vulnerability that allows an attacker to leak memory, the attacker may be able to leak the stack canary before performing a buffer overflow attack. Or, the attacker may find another way to write to arbitrary locations in memory and can simply only overwrite the return pointer and not the stack canary. Once the return pointer has been overwritten, the attacker may change the flow of code execution to anywhere in memory, including the stack, where his/her malicious code may be placed. Therefore, stack canaries are far from the end-all solution, and they may give the programmer a dangerous false sense of security.

4 Disabling Executability with NX and DEP

4.1 Introduction

To make it as difficult as possible to execute malicious code, operating system developers implemented a feature known as non-executable (NX) on Linux and hardware-enforced data execution prevention (DEP) on Windows which allows a program to mark memory pages as not-executable. After setting the NX bit on the stack's memory page, if the attacker attempts to redirect code execution to the stack, the CPU will interrupt execution and kill the program—thus preventing execution of potentially malicious shellcode on the stack [9].

4.2 Bypass

NX and DEP are fairly easy to bypass using a technique called Return-Oriented Programming (ROP) (ROP). If the attacker has control of a portion of a stack with a return pointer, it is possible to place pointers to code that must be in a memory page marked as executable (e.g. the program code and libraries) on the stack as if they were return pointers from actual function calls [9].

Once the vulnerable function returns, the CPU will pop the first ROP gadget—the fake return pointer—into the instruction register. The CPU will begin executing code wherever the first fake return pointer points to and will continue until it hits the next *ret* instruction. Then, the CPU will pop the address of the next gadget off the stack and into the instruction pointer as if it is returning from one function and calling the next.

Thus, an attacker can execute arbitrary code simply by returning to various places in memory where pre-existing code is. The list of return pointers that the attacker places on the stack is called a ROP chain [10].

On 32-bit systems, programs pass parameters to functions by pushing the parameters to the stack before calling a function. The function accesses them by simply popping them off the stack. In ROP chains, attackers pass parameters to the functions that are called by simply appending the arguments immediately after their respective gadgets [12].

However, on 64-bit systems, programs pass the first six parameters through the general-use registers *rdi*, *rsi*, *rdx*, *rcx*, *r8*, and *r9*, and the remaining parameters, if any, are passed through the stack as before [12]. Attackers can simply use other gadgets first to pop a value off of the stack and into a register used for function parameters, then call the gadget.

On Windows, even if the DEP is enabled (i.e. stack executability is disabled) and the ROP gadgets that an attacker needs to accomplish a goal are not available, the attacker could potentially use a gadget that re-enables executability and then simply return to the stack where shellcode exists [11].

Additionally, there are tools that convert regular, executable code into ROP chains for bypassing NX and DEP to simplify the process of crafting payloads [12]. The presence of NX and DEP are typically only an inconvenience to attackers and do not hinder exploitability.

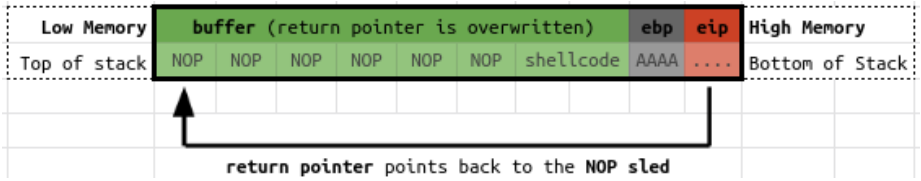

Figure 4: Redirecting code execution to a NOP sled

5 Location Randomization with ASLR and PIE

5.1 Introduction

To know where to set fake return pointers, Return-Oriented Programming attacks require the attacker to know where the gadgets—the sections of pre-existing code—are in memory. Moreover, buffer overflow attacks require the attacker to know where shellcode is in memory. So, to complicate these attacks as much as possible, operating system developers created a feature known as Address Space Layout Randomization (ASLR). On all modern operating systems, it is enabled by default. ASLR randomizes the positions in memory of the libraries, stack, and heap by an arbitrary offset that is chosen at runtime [13].

In addition, they created Position Independent Executable (PIE), a feature that, similarly to ASLR, randomizes the position of the program code in memory at runtime [14]. However, unlike ASLR, PIE is a feature that a program is compiled with and is not specific to the operating system.

5.2 Bypass

There are a few ways to bypass ASLR and PIE. First, if the attacker's goal is to execute shellcode and bypass ASLR, it is possible to reduce the number of possibilities where ASLR could have placed the stack and guess the location of the shellcode in memory using a no-operation sled (NOP sled) [15]. As the range of addresses in which the shellcode could be in memory is very large, the attacker can greatly increase chances of guessing the correct location of the shellcode by prepending hundreds or thousands of no-operation (*nop*) instructions to the shellcode. The *nop* instruction, as the name suggests, does not perform any operation, and the CPU will simply move to the next instruction. Thus, if the return pointer lands somewhere on the NOP sled, the shellcode at the end of the sled will execute (as seen in Figure 4). On 32-bit systems, the address space is small enough that this style of attack is very feasible; there are only 232 ($\approx$ 4.2 billion) possible addresses in a 32-bit address space. However, on 64-bit systems, chances that the return pointer will happen to point on the NOP sled are very slim as there are 264 ($\approx 1.9\times1019$) possible addresses [16].

Another method to bypass PIE is to overwrite only the less significant bytes of the return pointer; PIE only randomizes the more significant bytes of the address [17], so if it is possible for the attacker to selectively overwrite bytes of the return pointer, it is trivial to bypass PIE.

Both ASLR and PIE use the Global Offset Table (GOT)—which is at a fixed position in memory—to map functions to the randomized memory addresses.

Instead of calling the address of a function in memory directly (as the address is randomized at runtime), the program code simply calls the value of the pointer stored in the GOT for the particular function, as the position in the GOT where the pointer exists is fixed [18].

If attackers are able to leak a pointer to a function in memory from the stack or any value from the GOT they can simply calculate ASLR or PIE's fixed memory offset and completely defeat both protections.

Use of the GOT introduces another attack vector; as pointers to functions are always stored in the table, if a vulnerability enables an attacker to write to an arbitrary location in memory, simply overwriting the GOT entry for a particular function could result in code execution redirection; next time the function is called, it would instead jump to an address of the attacker's choosing.

Furthermore, if the GOT is positioned somewhere after the stack, in a buffer overflow attack, the attacker could potentially write past the stack, into the GOT, and over GOT entries.

6 Protecting the GOT with RELRO

6.1 Introduction

Partial Relocation Read-Only (Partial RELRO) is a feature that forces the GOT to be positioned before the stack so that it cannot be overwritten in a stack-based buffer overflow attack [17]. It does nothing, however, to randomize the location of the GOT or prevent the attacker from overwriting entries in the GOT.

Full Relocation Read-Only (Full RELRO), in addition to placing the GOT before the stack, sets the non-writable (NW) bit on the memory page containing the table so that GOT overwrite attacks cannot occur. To disable writability of the GOT, all functions that a program uses must be resolved and entered into the table before runtime before writability is disabled. Such runtime analysis of a program significantly slows down the initialization of larger programs [19]. Because it has such a drastic impact on performance, it is rarely enabled by default on larger programs, which tend to be the more vulnerable ones [19].

6.2 Bypass

Even if Full RELRO is enabled with all other protections and randomizations, it is still possible for an attacker with an arbitrary write vulnerability to overwrite an execution hook. For example, if the attacker overwrote the *malloc* execution hook, code execution would be redirected as soon as the *malloc* function is called; the *malloc* execution hook is a pointer to a function in memory that libc dereferences (i.e. executes) whenever memory is allocated using the *malloc()* function in C [20]. It is intended to be used by a common library on Linux called glibc, but it must be writable at runtime for glibc to function properly due to its nature. Thus, nothing can stop an attacker from taking advantage of execution hooks.

The only challenge in this attack vector is identifying the location of the *malloc* hook when ASLR is enabled, because the location of glibc in memory is randomized. Of course, as before, if a pointer to anywhere inside of glibc can

be leaked from the stack or the GOT, then it is trivial to calculate the offset, find the *malloc* allocation hook, and to take advantage of it.

7 Conclusion

Given enough time and effort, it is possible to bypass all modern protections. Stack canaries do not protect against arbitrary write vulnerabilities, can be bypassed with a memory leak vulnerability, and do not prevent other values on the stack from being overwritten. The non-executable bit and data execution prevention successfully prevent code on the stack from being executed, but do nothing to protect against Return-Oriented Programming attacks. Address Space Layout Randomization and Position Independent Executable can be bypassed using NOP sleds, by leaking addresses from the stack or Global Offset Table, or by simply overwriting only the less significant bytes of the return pointer. ASLR and PIE introduce the GOT which can be used as another vector to redirect code execution. And even the protections for the GOT are insufficient as code execution can still be redirected by overwriting execution hooks.

Compilers for memory-unsafe programming languages typically warn against the use of vulnerable functions that could potentially copy more data from a user than fits in a buffer or enable attackers to write to arbitrary locations in memory. However, most modern vulnerabilities in software applications exist because of the improper implementation of complex libraries, not because of the use of insecure functions. For example, heap corruption attacks, a common type of modern binary exploitation, abuse the improper implementation of the memory allocation libraries (e.g. *malloc.c* on Linux).

Furthermore, programmers who work on complex software which deals with low-level memory operations are likely not well aware of all of the security implications of their work. This will likely leave vulnerabilities as software developers may not fully understand the ways that their code interacts with other programmers' code. The Linux kernel is a good example of how lack of complete understanding of software leads to vulnerabilities—programmers who create kernel vulnerabilities often simply do not completely understand the inner workings of the kernel and how their code should be integrated [21].

Anyone is prone to write vulnerable code. By using protections to fix symptoms of vulnerable code, the problem itself remains unsolved. Furthermore, protections may provide software developers with a false sense of security. A sufficiently motivated attacker can bypass any of these protections, and motivation is easy to obtain when it comes to binary exploitation; the rewards are very high.

Modern protections may make exploitation difficult, but they will never be enough—binary exploitation will always be a threat. It is the responsibility of the developers of programs, libraries and frameworks, and programming languages to write secure code and make it as difficult as possible to write vulnerable code.

Glossary

Address Space Layout Randomization (ASLR)

a protection that randomizes the positions in memory of the libraries, stack, and heap by an arbitrary offset that is chosen at runtime.

buffer

dedicated chunks of memory often on the stack or heap, often temporarily store strings read from user input or from a file.

buffer overflow attack

an attack in which a buffer is filled and information stored after the buffer is overwritten (typically the return pointer).

data execution prevention (DEP)

on Windows, allows a program to mark a memory page as non-executable.

execution hook

pointer to a function that is dereferenced (i.e. executed) when some function is called; an event handler.

Full Relocation Read-Only (Full RELRO)

a protection that enables Partial RELRO and sets the non-writable (NW) bit on the memory page containing the GOT so that GOT overwrite attacks cannot occur.

glibc

GNU's libc library.

Global Offset Table (GOT)

a table that maps functions to their ASLR or PIE randomized memory addresses.

libc

a common library for programs written in C.

memory page

a dedicated, contiguous region in memory that has a specific purpose.

no-operation sled (NOP sled)

hundreds or thousands of no-operation instructions that are prepended to shellcode to increase the chances of guessing a correct memory location when ASLR is enabled.

non-executable (NX)

on Linux, allows a program to mark a memory page as non-executable.

non-writable (NW)

allows a program to mark a memory page as non-writable.

Partial Relocation Read-Only (Partial RELRO)

a protection that forces the GOT to be positioned before the stack so that it cannot be overwritten in a stack-based buffer overflow attack.

pointer

a value that stores the address of another value in memory.

Position Independent Executable (PIE)

a protection feature programs are compiled with that, similarly to ASLR, randomizes the position of the program code in memory at runtime.

register

a small, quickly-accessible storage location that is physically inside the CPU as opposed to RAM.

return pointer

the address of the instruction that will be executed next that is pushed to the stack when calling a function.

Return-Oriented Programming (ROP)

the use of a list of return pointers to pre-existing code, or ROP chain.

ROP chain

a list of ROP gadgets, and sometimes arguments for the gadgets.

ROP gadget

a pointer to pre-existing code in an executable memory page such as program code or libraries used for bypassing NX or DEP.

shellcode

attacker-controlled code, typically on the stack or another executable memory page.

stack

a structured list containing return pointers, arguments, and local variables.

stack canary

a randomly-generated, secret value that is pushed to the stack immediately after the return pointer is pushed when calling a function to prevent buffer overflow attacks.

stack frame

the relative positions of a base and stack pointers during a function call.

References

[1] A. Burnett, P. Biernat, and M. Gaasedelen, "Weaponization of a JavaScript-Core Vulnerability," *RET2 Systems Blog*, 11-Jul-2018. [Online]. Available: https://blog.ret2.io/2018/07/11/pwn2own-2018-jsc-exploit/. [Accessed: 25-Nov-2019].

[2] *An Introduction to Modern Binary Exploitation.* [Online]. Available: https://recon.cx/2019/montreal/training/trainingmodern.html. [Accessed: 25-Nov-2019].

[3] "Call and Return Instructions and the Stack." [Online]. Available: https://cs.nyu.edu/courses/fall04/V22.0201-003/ia32_chap_03.pdf. [Accessed: 25-Nov-2019].

[4] "Call and Return Instructions and the Stack." [Online]. Available: https://cs.nyu.edu/courses/fall04/V22.0201-003/ia32_chap_03.pdf. [Accessed: 25-Nov-2019].

[5] R. Han, "Stacks and Frames Demystified," 2005. [Online]. Available: https://www.cs.colorado.edu/~rhan/CSCI_3753_Spring_2005/CSCI_3753_Spring_2005/Lectures/02_03_05_Stacks_and_Frames.pdf. [Accessed: 25-Nov-2019].

[6] A. One, "Smashing the Stack for Fun and Profit." [Online]. Available: http://www-inst.eecs.berkeley.edu/\textasciitilde{}cs161/fa08/papers/stack_smashing.pdf. [Accessed: 25-Nov-2019].

[7] "Format String Vulnerability." [Online]. Available: http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf. [Accessed: 25-Nov-2019].

[8] D. Fleck, "Stack Smashing Attacks." [Online]. Available: https://cs.gmu.edu/~astavrou/courses/ISA_564_F15/StackSmashing.pdf. [Accessed: 25-Nov-2019].

[9] M. Gaasedelen, "DEP & ROP," 10-Mar-2015. [Online]. Available: http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/11/07_lecture.pdf. [Accessed: 25-Nov-2019].

[10] S. Staniford, "Defending Computer Networks." [Online]. Available: http://www.cs.cornell.edu/courses/cs5434/2014fa/Lecture4-defenses-vulnerabilities.pdf. [Accessed: 25-Nov-2019].

[11] E. J. Schwartz, "The Danger of Unrandomized Code." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.418.8542&rep=rep1&type=pdf. [Accessed: 25-Nov-2019].

[12] J. Stewart and V. Dedhia, "ROP Compiler." [Online]. Available: https://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf. [Accessed: 25-Nov-2019].

[13] P. Biernat, "Address Space Layout Randomization," 31-Mar-2015. [Online]. Available: http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/15/09_lecture.pdf. [Accessed: 25-Nov-2019].

[14] R. Turner, "Heap Corruption," 2018. [Online]. Available: https://cseweb.ucsd.edu/classes/sp18/cse127-b/cse127sp18.6.pdf. [Accessed: 25-Nov-2019].

[15] G. Wang, "Buffer Overflow." [Online]. Available: http://people.cs.vt.edu/~gangwang/class/cs4264/4-buffer-overflow.pdf. [Accessed: 25-Nov-2019].

[16] H. Shacham, E.-J. Goh, M. Page, N. Modagugu, B. Pfaff, and D. Boneh, "On the Effectiveness of Address-Space Randomization." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.6727&rep=rep1&type=pdf. [Accessed: 25-Nov-2019].

[17] A. D. Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "How the ELF Ruined Christmas," 12-Aug-2015. [Online]. Available: https://sefcom.asu.edu/publications/how-the-elf-ruined-christmas-usenix2015.pdf. [Accessed: 25-Nov-2019].

[18] C. Zhang, L. Duan, T. Wei, and W. Zou, "SecGOT: Secure Global Offset Tables in ELF Executables." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1008.4167&rep=rep1&type=pdf. [Accessed: 25-Nov-2019].

[19] X. Ge, M. Payer, and T. Jaeger, "An Evil Copy: How the Loader Betrays You." [Online]. Available: http://www.cse.psu.edu/~trj1/papers/ndss17.pdf. [Accessed: 25-Nov-2019].

[20] G. Richarte, "Four different tricks to bypass StackShield and StackGuard protection," 03-Jun-2002. [Online]. Available: https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf. [Accessed: 25-Nov-2019].

[21] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems." [Online]. Available: https://pdos.csail.mit.edu/papers/chen-kbugs.pdf. [Accessed: 25-Nov-2019].