An Overview of Prevention/Mitigation against Memory Corruption Attack

Mahmood Jasim Khalsan, Michael Opoku Agyeman
Department of Computing, University of Northampton, UK

Abstract—**One of the most prevalent, ancient and devastating vulnerabilities which is increasing rapidly is Memory corruption. It is a vulnerability where a memory location contents of a computer system are altered because of programming errors allowing execution of arbitrary codes. It particularly happens in low-level programming languages such as C, C++ because of their lack of memory safety. Many defense techniques against this kind of attacks have been presented and implemented to prevent it. However, an advanced version of the attack can bypass some of these techniques and harm the system. In this work, we present an overview of the Memory corruption attacks and the existing mitigation techniques for both compilers and operating systems. We hope that this survey will provide sufficient details that can be useful for researchers and system designer.**

Index terms: Memory Corruption,Vulnerabilities,attacks, technique to prevent hacking.

I. INTRODUCTION

In the past two decades, Memory Corruption attacks have captured the attention of security research community [1]. The first recognized worm that exploited a memory corruption attack to spread itself was called the Morris Worm. This kind of attack is well-known for its high exploitability that allows the attackers to simply execute arbitrary codes. It takes control of the remote code execution according to latest Microsoft report [2].

There are three main types of memory errors that can cause the Memory corruption: accessing uninitiated, accessing out-of-bounds as well as accessing freed memories. Also, different software bugs can cause these errors. For example, out of bound memory might be occurred due to incorrect bound check, incorrect allocation or lack of bound check or others [3]. A well-known example of memory corruption attacks is buffer overflow exploitation.

This attack happens when a program tries to read or write exceeding the end of a buffer (also known as a bounded array). Some popular string manipulation functions, which are normally used along with an array variable as their parameters, are strcpy(), memcpy() and strcmp (). These functions are generally vulnerable when a form bound checking is missing [4]. The security research community has implemented different techniques to prevent such attacks but advanced attack with a combination of complex strategies can still bypass them. In this paper, a survey of the memory corruption vulnerabilities is being presented as well as some of the existing mitigation techniques against them.

The rest of the survey is organized as the follows: Section II describes the buffer overrun attacks and Section III present the existing countermeasures. Section VI presents the summary and conclusions are concluded in Section V.

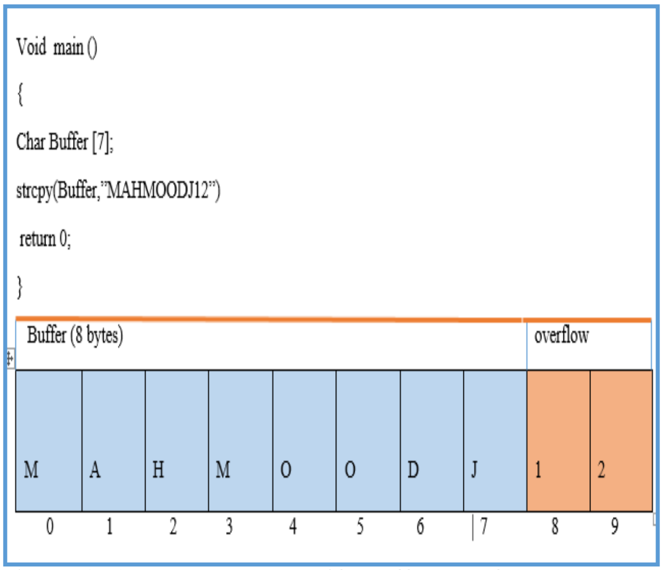
II. BUFFER OVERFLOW

A. General information about buffer overflow attacks

This section introduces the concept of buffer overflow and how malicious users or hackers can attack the Memory Corruption, it can be exploited by buffer overflow. A buffer overflow happens when the size of the data that entered to the buffer is larger than the size of the data that the buffer can handle. In another word, buffer overflow is simply occurs when data size reaches the out-of-bounds of the memory [5] Based on that, many attackers can exploit this vulnerability to force a system crash, and control-flow hijacking as well as some malicious users can run an arbitrary code. Plenty of the applications that built by some programming languages are mostly leading to buffer overflow as the buffer that has been specified is not large enough or the developers of these applications do not pay attention of checking overflow issues.

These common mistakes particularly occur with C/C++ as these two languages have lack to build prediction against buffer overrun. According to the aforementioned, applications that build in C/C++ programming languages are more vulnerable to attack [6].

Generally, there are two most common techniques that can be utilized by an attacker to attack through buffer overflow attacks. The two techniques are stack and heap attacks.
To make the buffer overflow more understandable, an example has been used for this purpose (Figure 1).



```
Void main ()
{
Char Buffer [7];
strcpy(Buffer,”MAHMOODJ12”)
return 0;
}
```

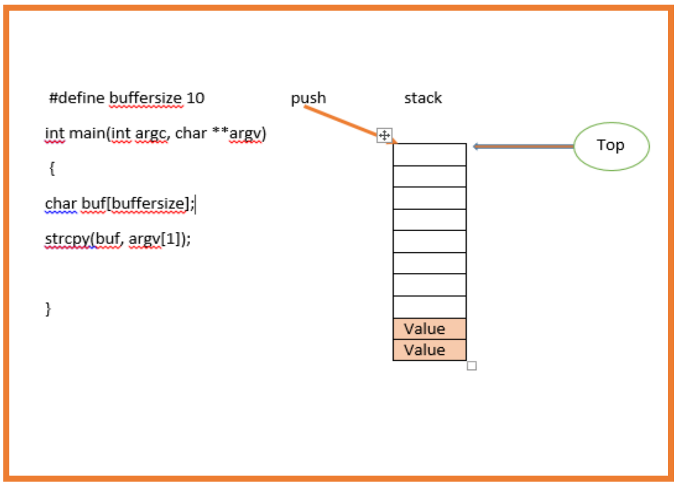
Figure 1: An example to clarify buffer overflow.

In the example the buffer is overflow with 2 bytes because the buffer is allocated for 8 bytes in the code and the data that has been entered is 10 bytes. Underlying cause that is the strcpy() does not inspect bounds so attackers can write anything outside the buffer space. Furthermore, these two bytes can be utilized to run shell code by attackers.

B.Stack–relied buffer overrun attacks.

In this part of study, we produce that how attackers can exploit stack-based buffer overflow. As we mentioned previously that a buffer overflow occurs when the input data size is which should be handled by buffer is larger than the size of buffer itself. On another word can be descried as it happens when insufficient boundary checking [7]. Consequently, an attacker can exploit this drawback point to write malicious data in the memory out of buffer,

An example has been utilized to clear stack-rely buffer overflow attacks. Suppose we have this code.



```
#define buffersize 10
int main(int argc, char **argv)
{
char buf[buffersize];
strcpy(buf, argv[1]);

}
```

Figure 2: An example to clarify buffer overflow with stack-based overrun attack.

Indeed, the code in Figure 2 demonstrates that the size of buffer is constant so very simple that the string in argv[1] might be exceeded the size of the buffer size which results stack-based buffer overrun attacks. Consequently, the code allows attackers to write malicious data to memory outside of buffer.

C. Heap-relied Buffer overrun Attack.

Heap overflow is a kind of buffer overrun. It occurs once a chunk of memory is assigned to heap also data is typing to this memory with no limitation of checking the size of the data that have been written. Hence, this vulnerable allows attackers to overwrite some important data structure in the heap for instance the heap header.

Furthermore, there are two kinds of heap with windows can be explained as follow. The first type is default heap which is utilized with windows32 to manage and specify memory for both local and global variable as well as local memory by using functions [malloc()]. Second type is dynamic heap is made by some methods like HeapCreate() which returns the address to a memory chunk that includes the heap header [8].

In the example (Figure 3) the attacker has modified the return address therefore call procedure with the new address return address as a result the attacker can control by executing the malicious code which has been allocated somewhere in the process.

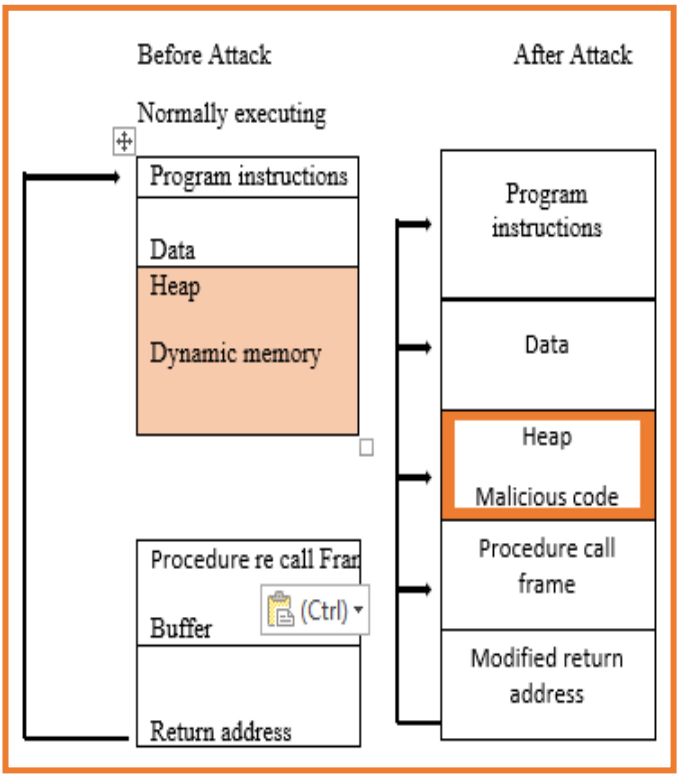


Figure 3: An example to illustrate buffer overrun with heap-based overrun attack [27].

D. Block started by symbol-relid buffer overflow attacks

The whole idea of Block started by symbol (BSS) is any program starts running, all variables either local or global which are not assigning value as initial or these variables initialized with zero for these variables, BSS will generate automatically. Consequently, the BSS area results buffer overrun in the buffer which has been created as BSS area. Subsequently, the BSS area can be exploited by some hackers to overwriting some data [9].

E. Other hacking techniques relevant Buffer Overrun.

1) Format String Exploit: The vulnerabilities of these kind of techniques are resulted by incorrect invocations of some of function like printf, sprintf and syslog. An attacker can exploit that by entering incorrect data for the first parameter of the printf () function, as we expect the first parameter has to be printf(‘’%s’’,buffer) to clearly that an example has been used for this purpose (Figure 4).

We assume that the attacker inputs in the first argument of the printf() function ‘’ "\x10\x01\x48\x08 %x %x %x %x %s"’’. This vulnerability provides to the attacker the possibility of overwriting essential program flags which leads that can control access privileges. On another hand, the attacker can write arbitrary data into memory. Moreover, overwriting the return address on the stack by using function pointer [10].

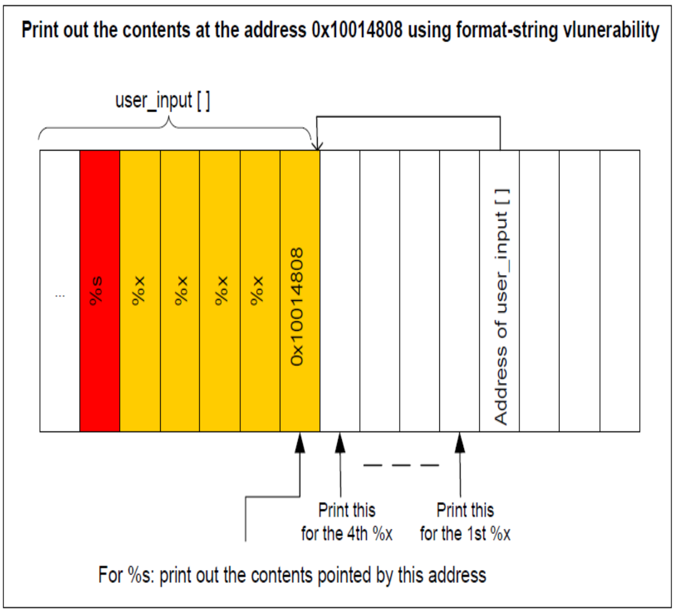

Print out the contents at the address 0x10014808 using format-string vlunerability
user_input [ ]
...
%s
%x
%x
%x
%x
0x10014808
Address of user_input [ ]
Print this for the 4th %x
Print this for the 1st %x
For %s: print out the contents pointed by this address

Figure 4: An example to clearly Format String Attack [11].

2) Vulnerability in Numerical Handlin: The problem of numeric treatment that when we do incorrect math operation or numeric conversion. Numeric handling includes integer Overflow or Wraparound and particularly they are utilized with buffer Overflow attacks [12]. Integer Overflow happens once doing an operation such as multiplying for two numbers and the result of the operation is too large which it exceeds the range of algorithm expression which determines for data type that cause buffer overflow. That leads to exploit by attackers as did mention in the section that has illustrated buffer overflow attacks [13].

III. THE CURRENT TECHNIQUES AGAINST MEMORY CORRUPTION ATTACKS.

This section sheds some light on the different countermeasures that have been used in operating system through complier and linkers.

A. OS countermeasures.

We consider five main techniques that have been used in the operating systems (OS) in order to prevent an attacker.

1) Data Execution Prevention (DEP): This technique is used to force the memory to be writable with no possibility to be executable or it is executable but must be read-only such as code segments. The protection technique has developed in order to hinder or at least reduce the normal exploits such as the vulnerabilities that attacked by existing code or writing malicious code inside data segments [13]. The fundamental purpose of this mechanism is preventing the program code of any possibility to execute in stack area and heap area furthermore shared libraries. DEP with windows can be divided for two types as following: Software-enforce as explained previously it assists to prevent code of executing in stack or heap area.

Software DEP can describe as blocking for hackers who use exception-handing in windows to write malicious code. Hardware-enforce DEP can be defined as making mark for memory that have to be non-executable. This enforcement helps to make all memory locations in a process are not executable excluding of that the code which includes executable code. The DEP is starting with windows XP SP2 and especially with windows 32 bits version as feature security which uses in both Advanced Micro Devices (AMD) and Intel. The processor that support this feature has No-Execute (NX) as known ADM / Execute Disable Bit (XD) as known Intel. The primary key that should be mentioned is that any processor which supports these features, the processor has to be executing when Physical Address Extension (PAE) feature is enable [14]. However, the attackers can use more intricate techniques to corrupt memory for example, using return-to-libc and return-oriented-programming [15].

2) ASCII: ASCII-armor is a prophylactic security technology which designed by evolving Exec-Shield function through Red Hat, Inc. ASCII-armor is based on the Exec-Shield where ASCII-armor becomes enable when we assign the Exec-Shield option in addition re-link the kernel. While when we assign the non-Exec-Shield in this case ASCII-armor becomes disable. In very briefly, this technique utilizes with sharing library as we know sharing library provides ability to extend some external codes which executed in run time, some attackers exploit this vulnerability to inject vector into memory. At the end, the ASCII-armor feature uses to block attackers copying malicious code into memory [16].

3) Address Space Layout Randomization (ASLR): The first starting of using ASLR was in 2001 with Linux as known Linux patch while in 2007 was the first using this protecting technology with windows operation systems and especially with Vista. Utilizing ASLR with windows operating systems provide 256 address space locations. More specifically, adding ASLR feature to Vista raises the number of probability for address space locations. Consequence of this, the possibility of pinpointing the correct location to run code with Vista after adding ASLR became very complicated so attackers have solely one chance out of 256. However, 2011 was the first beginning of using ASLR with Mac OSX and both iOS as well as google android [17]. This mechanism becomes very powerful prediction against memory misuse when it integrates with Data Execution Prevention (DEP) technology.

This technology randomizes system's virtual memory layout when each new code runs, or the system is booted. That's leading to, an attacker cannot find out the virtual address of related memory locations that required to make a control-flow or other hacking techniques. The main aim of ASLR is to prevent attackers of executing shellcode in stack area or heap area additionally shared libraries by randomizing them. That's mean, it randomizes the locations of stack area and heap area moreover share libraries.

At the end of this section, we can take two advantages of using this effective protection technology.

Firstly, it can defense against remote attacks. Secondly, local attacker would not be able to attack memory because of randomly offsetting memory structures and module base addresses. Although, ASLR does not prevent buffer overflow at all it just makes the ability of exploiting memory is very difficult [23]. Even that, ASLR has two drawbacks: Lack of entropy. Leak information.

4) Kernel Address space layout Randomization (kASLR): This mechanism attempts to randomize both program and data locations in kernel area when the kernel start up (Figure 5). The main advantage of this technology is moving the interrupt descriptor table (IDT) in a way form most of the kernel to a location in read-only memory. As well-known SIDT instruction assists to find out the location of IDT which previously utilized to detect the location of kernel code, so the technique does give any possibility for adversary to use it because IDT has located in elsewhere.

Furthermore, kASLR is used to safeguard overwriting because it is read-only. Finally, the Kernel ASLR has been used with Linux kernel 3.14 version therefore has been developed to include randomization of the module load location In Linux kernel 3.15 [18].

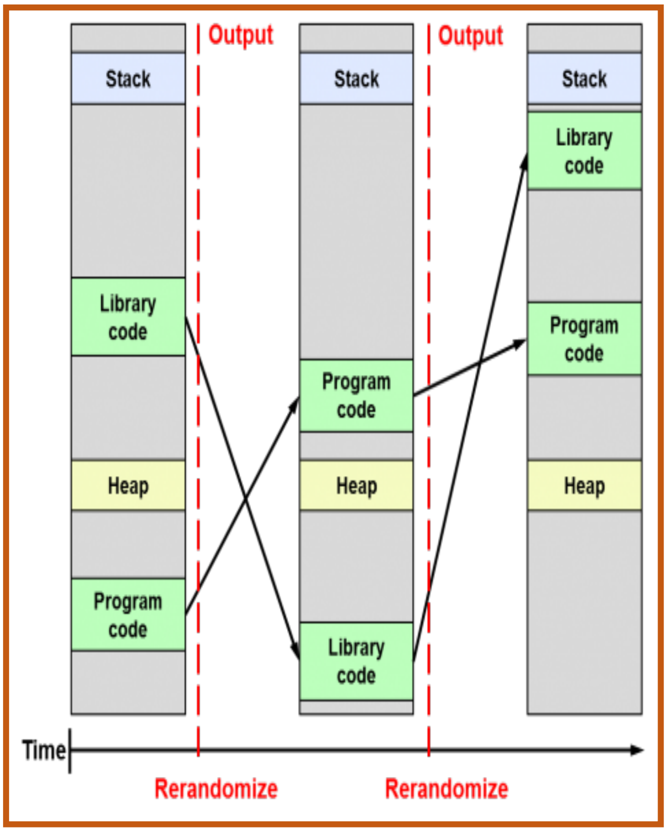


Figure 5: clarification of kernel ASLR [28].

Figure 6 gives a comparative summary of bits of entropy in Linux vs PaX vs ASLR_NG.

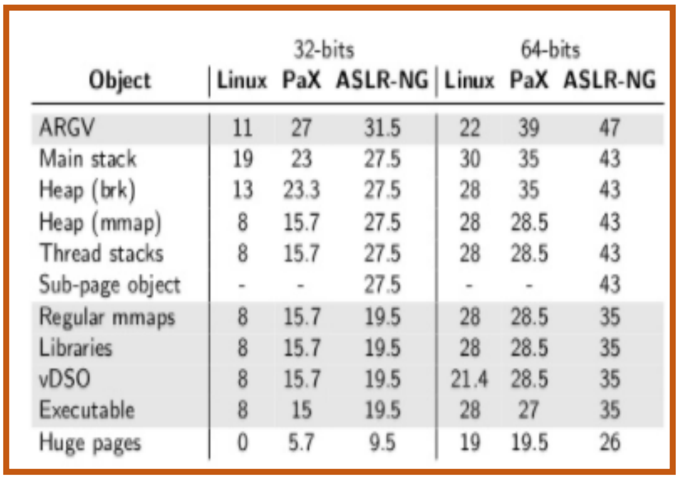
B. Compiler and Linker Countermeasures.

1) GCC Compiler Options: In this part of work, we introduce the technologies that have been added in GCC and supported by compiler options.

a) IO2BO detection.

Integer Overflow-to-buffer-overrun (IO2BO) is one of the most common vulnerabilities that used by attackers. As illustrated previously in (Vulnerability in Numerical Handling) section of the paper that Integer Overflow occurs when the output of any calculation such as (multiplication or dividing by zero) is surpass the range of the data type [19].

The fundamental reason that made Integer Overflow is extremely popular because lots of programmer have not yet pay to the magnitude of the danger that comes from integer overflow.



Object	32-bits			64-bits		
	Linux	PaX	ASLR-NG	Linux	PaX	ASLR-NG
ARGV	11	27	31.5	22	39	47
Main stack	19	23	27.5	30	35	43
Heap (brk)	13	23.3	27.5	28	35	43
Heap (mmap)	8	15.7	27.5	28	28.5	43
Thread stacks	8	15.7	27.5	28	28.5	43
Sub-page object	-	-	27.5	-	-	43
Regular mmaps	8	15.7	19.5	28	28.5	35
Libraries	8	15.7	19.5	28	28.5	35
vDSO	8	15.7	19.5	21.4	28.5	35
Executable	8	15	19.5	28	27	35
Huge pages	0	5.7	9.5	19	19.5	26

Figure 6: comparative summary of bits of entropy [29].

Based on the threads of integer overflow compiling has supported by –ftrapv Option which uses to detect integer overflow with addition, subtraction, multiplication, and division for signed integers when a program is running.
Obviously, when a program identify integer overflow must be stop. The first appearing was in GCC version 3.4.
However, it is not possible way to detect dividing by zero, so it is not completely detection for integer overflow [20].

b) *Address Sanitizer.*

This tool is very fast way to detect an error in memory. Basically, it was evolved by Google. In short, this option has been implemented in GCC to detect six type of ways that used by attackers to hack the memory.
The six types can be listed as following: Out-of-bounds, Use-after-free, Use-after-return, Use-after-scope, Double-free, invalid free and Memory leaks. Defiantly, when these kind of hacking techniques detected, the program will stop immediately to prevent attackers of exploiting these vulnerabilities by corrupting the memory [21].

c) *Stack Smashing Protector (SSP).*

This option was released in GCC as a patch before 4.1 and has been applied in 4.1 version. Most importantly, this security feature has been developed to defense against memory attacks such as return address, frame pointers as well as pointers in stack area. This mechanism was added to the compiler to protect program against stack mashing attacks. The main idea of SSP is to detect that the return address has been modified before the function returns by inserting ''canary'' word underneath return address on the stack as shown in (Figure 7). Therefore, checking whether the canary is intact or not before moving to return address [22].
In fact, canary can be used merely when the buffer size is 8 bytes or larger while a canary would be able to enter in buffer when the size is 4 bytes or over with Ubuntu 10.10. Additionally, to prevent hackers of finding the canary value or its location by buffer overflow. For this purpose, 32-bits random number has been used as value for the canary to make the possibility of guessing canary value by attackers is very complicated. These 32-bits random numbers will be selected when the program starts up.

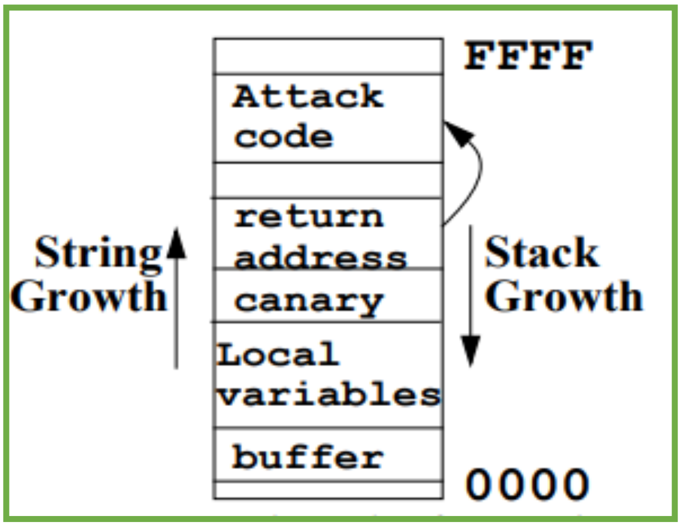


Figure 7: using Canary to protect against Stack smashing attacks [30 [1] [1] [1]].

d) *Automatic Fortification.*

Automatic Fortification is a function has been implemented in GCC especially in 4.0 version. Most importantly, the Automatic Fortification has been added to prevent Format String attacks.
As well-known, Format String attacks is one of ways that results buffer overflow (illustrated in Format String Exploit of this paper) and that assists attackers to write or insert malicious code. Depending on that, any function has a buffer must be substituted with safe functions. Automatic Fortification due inspecting string size, if it is larger than buffer size that has been determined for storing the string at runtime. That's mean, checking whether the buffer size is sufficient or not before calling the function that results buffer overrun.

Finally, this feature is not effective way to prevent Format String attacks at all, but it is only reducing it. Because this feature is not being able to detect the Format String attacks when a function passes a buffer for another function as a parameter.

2) Stack-Shield protection.: This technology is one of the runtime techniques which has been implemented to guard return address. Basically, the key idea is copy the return address and keep it in somewhere that should not be overflowed memory space at runtime. Thus, the return address that has been saved somewhere will be used instead of utilizing the return address on the stack that could be exploited by an attacker. In this case, this tool can reduce risk magnitude that comes from the exploiting return address [24].

IV. Other technologies *against memory corruption attacks.*

Instruction Set Randomization (ISR) is one of the techniques that has been implemented to defense against code injection attacks by randomly modifying the instructions. More specifically, this feature is used to prevent Return-Oriented Programming attacks.

Ultimately, ISR is not enough way to avoid control flow hijacking attacks because some of techniques are not necessary require to know the Instruction Set such as return-to-libc attack. Moreover, using LibsafePlu for Runtime Buffer Overflow Protection [25]

Pin, this tool has been implemented for security reason at runtime which used to defense against return address. It works as following, when the program starts running, it is potentially analysis and create Binary Translation for the code that has been inserted by user.

Eventually, Pin is supported by these operating systems Linux, Windows, and OS X [26].

V. summary

We derive some generic techniques that have been implemented to defend against memory corruption attacks. These technologies are implemented either with operating systems by default or with Compiler and Linker as can be illustrated as following. 1) Operating system: One most common options that have been provided by operating system to defense against memory corruption attacks is kernel ASLR. Basically, ASLR is supported by new operating systems such as Linux, windows and OS. Even if ASLR is made great protection but is not efficiency technique because it has lack of entropy moreover leak of information.

2) Compiler and Linker: There are many techniques such as (*Address Sanitizer, SSP, IBO and etc.)* have been implemented with Compiler and Linker by default to defense against buffer overflow but they are enough to prevent attackers for attacking memory. Thus, the programmer must be pay attention to secure his/her program because there are lots vulnerabilities that can be exploited by attackers.

VI. CONCLUSION

In this paper, we highlight most common technologies that can be exploited by attackers to hack memory. Additionally, we present existing countermeasures that have been used to prevent or minimize memory corruption attacks. Our aim in this paper is to assist researchers and system designers who wants to improve the current countermeasures for memory corruption attacks either by making combination for available countermeasures or by adding new feature(s).

VII. References

[1] L. Szekeres, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, pp. 48-62. doi: 10.1109/SP.2013.13.

[2] C. Song, "Preventing Exploits Against Memory Corruption Vulnerabilities," Georgia Institute of Technology, 2 8 2016.

[3]Tim Rains,Matt Miller,David Weston, "Exploitation Trends:From Potential Risk to Actual Risk," San Franisco, 24 April 2015.

[4] M. Alam, D. B. Roy, S. Bhattacharya, V. Govindan, R. S. Chakraborty and D. Mukhopadhyay, "SmashClean: A hardware level mitigation to stack smashing attacks in OpenRISC," ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, Kanpur, 2016, pp. 1-4.*

[5] A. Kundu and E. Bertino, "A New Class of Buffer Overflow Attacks," *International Conference on Distributed Computing Systems*, Minneapolis, MN, 2011, pp. 730-739.

[6] "CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')," 18 January 2018.

[7] G. Chen *et al.*, "SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities," in *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 6, pp. 368-379, Nov.-Dec. 2013.

[8] M. Mouzarani, B. Sadeghiyan and M. Zolfaghari, "A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes," *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, Zhangjiajie, 2015, pp. 42-49.

[9] ARichard Stevens, Stephen A. Rago, "Advanced Programming in the UNIX® Environment: Second Edition," 17 June 2005.

[10] John Wilander, Mariam Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow," Dept. of Computer and Information Science, Link¨opings universitet.

[11] W. Han, M. Ren, S. Tian, L. Ding and Y. He, "Static Analysis of Format String Vulnerabilities,"*First ACIS International Symposium on Software and Network Engineering*, Seoul, 2011, pp. 122-127.

[12] M. Qingkun, W. Shameng, F. Chao and T. Chaojing, "Predicting integer overflow through static integer operation attributes,"*International Conference on Computer Science and Network Technology (ICCSNT)*, Changchun, 2016, pp. 177-181.

[13] W. Dietz, P. Li, J. Regehr and V. Adve, "Understanding integer overflow in C/C++," *2012 34th International Conference on Software Engineering (ICSE)*, Zurich, 2012, pp. 760-770..

[14] "Microsoft Windows Server 2003 Service Pack 1Microsoft Windows Server 2003 Web Edition," 12 Jul 2017.

[15] N. Stojanovski, M. Gusev, D. Gligoroski and S. J. Knapskog, "Bypassing Data Execution Prevention on MicrosoftWindows XP SP2," *Availability, Reliability and Security,ARES. The Second International Conference on*, Vienna, 2007, pp. 1222-1226.

[16] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen and A. R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization,"*IEEE Symposium on Security and Privacy*, Berkeley, CA, 2013, pp. 574-588.

[17] Ryohei Watanabe,Shuta Kondo, "A Survey of Prevention/Mitigation against Memory Corruption Attacks," International Conference on Network-Based Information Systems, 19 Dec 2016 .

[18] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen and A. R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization,"*IEEE Symposium on Security and Privacy*, Berkeley, CA, 2013, pp. 574-588.

[19] D. M. Stanley, D. Xu and E. H. Spafford, "Improved kernel security through memory layout randomization," *IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, San Diego, CA, 2013, pp. 1-10..

[20] B. Zhang, C. Feng, B. Wu and C. Tang, "Detecting integer overflow in Windows binary executables based on symbolic execution," *IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Shanghai, 2016, pp. 385-390.

[21] Chao ZhangTielei WangTao WeiYu ChenWei Zou, "IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time," Institute of Computer Science and Technology, Peking University, 2010. [22] K. Serebryany, "Continuous Fuzzing with libFuzzer and AddressSanitizer,"*IEEE Cybersecurity Development (SecDev)*, Boston, MA, 2016, pp. 157-157.

[23] Y. Younan, D. Pozza, F. Piessens and W. Joosen, "Extended Protection against Stack Smashing Attacks without Performance Loss," *Annual Computer Security Applications Conference (ACSAC'06)*, Miami Beach, FL, 2006, pp. 429-438.

[24] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall and J. W. Davidson, "ILR: Where'd My Gadgets Go?," *IEEE Symposium on Security and Privacy*, San Francisco, CA, 2012, pp. 571-585. doi: 10.1109/SP.2012.39.

[25] Zhiqiang Lin, Bing Mao and Li Xie, "LibsafeXP: A Practical and Transparent Tool for Run-time Buffer Overflow Preventions,"*IEEE Information Assurance Workshop*, West Point, NY, 2006, pp. 332-339..

[26] Naftaly, "Pin - A Dynamic Binary Instrumentation Tool," 13 June 2012.

[27]"Home Buffer Overflow," [Online]. Available: http://globaltechconsultants.org/?q=content/buffer-overflow.

[Accessed 3 April 2018].

[28] D. L. Azaña, "Differences between ASLR, KASLR and KARL," 2015. [Online].

Available:http://www.daniloaz.com/en/differences-between-aslr-kaslr-and-karl/. [Accessed 3 April 2018].

[29]Dr. Hector Marco-Gisbert ; Dr. Ismael Ripoll,

"Exploiting Linux On 32-bit and 64-bit Systems," 16 Jun 2016. [Online]. Available: https://www.slideshare.net/

AlesJohn/exploiting-linux-on-32bit-and-64bit-systems. [Accessed 4 April 2018].

[30]Crispin Cowan, Steve Beattie, Ryan Finnin Day, "Protecting Systems from Stack Smashing Attacks with StackGuard," 2013.

[Online].Available:https://pdfs.semanticscholar.org/9d92/fa9eaa6ca12888d303deffe8bc392b85c09f.pdf. [Accessed 5 April 2018].