Seminar Report

On

Buffer Overflow Attacks

By

Alok Bhawankar

1032170126

Under the guidance of

Prof Geeta Sorate

MIT-World Peace University (MIT-WPU)

Faculty of Engineering

School of Computer Engineering & Technology

*** 2019-2020 ***

MIT-World Peace University (MIT-WPU)

Faculty of Engineering
School of Computer Engineering & Technology

CERTIFICATE

This is to certify that Mr. Alok Bhawankar of B.Tech., School of Computer Engineering & Technology, Trimester – IX PRN. No. 1032170126, has successfully completed seminar on

Buffer Overflow Attacks

To my satisfaction and submitted the same during the academic year 2019 - 2020 towards the partial fulfillment of degree of Bachelor of Technology in School of Computer Engineering & Technology under Dr. Vishwanath Karad MIT- World Peace University, Pune.

Prof. Geeta Sorate — Seminar Guide

Prof. Dr. M.V.Bedekar — Head, School of Computer Engineering & Technology

LIST OF FIGURES

ABBREVIATIONS

Sr No.	Abbreviation	Explanation
1.	BO	Buffer Overflow
2.	IM	Incomplete Meditation
3.	RC	Race Condition
4.	EIP	Extended Instruction Pointer
5.	EBP	Extended Base Pointer
6.	NOP	No Operation
7.	ASLR	Address Space Layout Randomization
8.	BTB	Branch Target Buffer

ACKNOWLEDGEMENT

With immense pleasure I, Mr. Alok Bhawankar of B.Tech., School of Computer Engineering & Technology, Trimester – IX PRN. No. 1032170126 express my sincere thanks to Prof. Dr. M V. Bedekar , Head of School of Computer Engineering and Technology, for providing all the necessary resources required for the completion of my seminar.

I express my profound thanks to my seminar guide, Prof. Geeta Sorate for her valuable suggestions and guidance in the preparation of seminar report.

I am also grateful to all those who have indirectly guided me and helped in the preparation of seminar.

Alok Bhawankar
PA06
Panel-1

INDEX

ABSTRACT

In recent decades, the buffer overflow has been a source of many serious security issues. Prevention against such attacks becomes more important but also costly. Over the decade it has resulted in exploitation of many secure applications and extracting sensitive information from such web servers or application servers and custom applications such as Whatsapp, Exim server, etc. It can be tremendously powerful, allowing an attacker to execute code of their choosing and completely compromise a buggy application. Buffer overflows gain their power because programs often store addresses of code adjacent to data in arrays, with the processor using these code addresses to determine which instructions to run each time a function call is completed. Overflowing the buffer allows this code address to be overwritten, which in turn means that the attacker can trick a program into running malicious code to extract useful information, gain access to the system or break the application. This study illustrates the working principle of buffer overflow using a shellcode by exploiting a vulnerable web server and also the detection of such attacks and how to prevent them.

Keywords : Buffer Overflow, Cyber Security, Exploitation, Penetration Testing

1. INTRODUCTION

The development of information technology has highlighted the discovery and exploit of software vulnerability. Many vulnerabilities can be mined effectively by mining technologies, but only a part of these vulnerabilities can be exploited, thereby causing serious consequences. The rapid and accurate analysis of the exploitability of vulnerability has come to a key problem of vulnerability analysis and detection

In the electronic world, security is the major issue on the internet, intranet, andextranet. Ethical hacking is a term which is used to increase security by identifying and overcoming those vulnerabilities on the systems owned by third parties. Attacker Uses vulnerabilities as an opportunity to attack software and web applications. Thus,the system needs to be protected from attackers so that the attacker cannot hack information and make it misbehave according to him/her. So, ethical hacking is a way to test and to identify an information technology environment for present vulnerabilities. Software is used everywhere in the digital world. But due to the flaws in soft-ware, software fails and the attacker takes this as an advantage and uses this opportunity to make software misbehave and use according to them. Flaws increase the risk to security. Some of the software developer manages the software risk by increasing the complexity of the code, but absolute security cannot be achieved. According to the literature survey, some of the software flaws which lead to security vulnerabilities are Buffer Overflow (BO), Incomplete Mediation (IM), and Race Condition (RC) [1]

This paper gives an overview of one of the flaws that exist in the software, i.e. buffer overflow and how this flaw leads to the security vulnerabilities that can be exploited and what are the preventives measures that can be taken to protect it from the attackers. This paper also explains one of the famous attacker’s techniques to access information from the database of the web application using Kali Linux. The SQL queries in Kali Linux that are used to retrieve information from the database of a web application are shown for a particular website. This paper provides a description and example with a screenshot of how these attacks can be per-formed and what will be the outcome.

2. Background

2.1 PROGRAM VULNERABILITIES

With the C language's minimal abstraction from machine-specific code, it is highly efficient. With C's own compiler handling the machine code generation, portability (independence from computer type) is achieved, thus cutting down on unnecessary steps and time. In addition, there is "at least one compiler for almost every architecture" [8], unlike assembly languages, which are processor-specific [6].

The C Standard library is a repository of built-in macros and functions that allows programmers to utilize pre-built functions within their code [4] . For example, without the addition of this library,instead of using a one-liner function like printf("HelloWorld "), you would have to write directly in assembly, with the attendant expansion factor in the number of lines of code. Common occurrences like opening files, gathering input, or calculating string lengths are simplified and standardized by this library.

```
char *fgets(char *str, int n, FILE *stream)
```

```
char *gets(char *str)
```

The Fgets() function has been corrected fromgets() to take additional arguments that support specifying the amount (intn) of characters to gather from the user. Any additional characters entered will not be recognized. Though an improvement to the previously vulnerable library functions, these additions do not provide complete safety. Between the numerous buffers that can exist between the underlying function and the input device, incorrect specifications in their length, and the existence of unprotected legacy code, vulnerabilities still exist.

2.2 Buffer Overflow

Buffer overflow is a software flaw which is introduced unintentionally by the programmer. For example, in a program, while writing data to an allocated memory, if data is assigned to the memory which is not allocated, it overruns the boundary of the allocated memory. Most popular languages C and C++ also do not have built-in boundary check, i.e., they do not automatically check the data trying to access memory location of an array is outside the boundaries (total size) of an array, for example (Table1).Here, the total size of array is 10, but the data is assigned at location 30 which is outside the boundary of an array. Boundary check can eliminate vulnerability. Java And C # are the languages which have boundary checks but they have performance penalty for checking so the developer uses C and C++ for coding [2]. These buffer overflow vulnerabilities can be exploited by attackers in many ways:

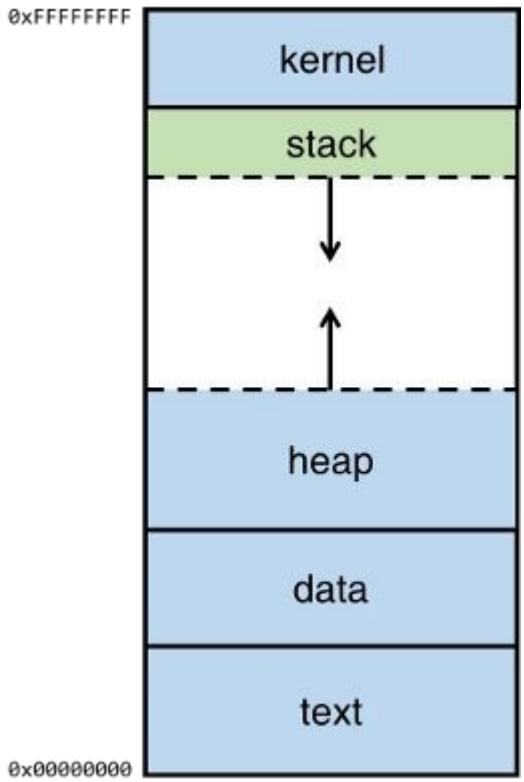
```
void main()
{
int buffer[10];
buffer[30] =45;
}
```

(a) Denial-of-Service Attack—Buffer overflow flaw may likely cause system crash,so attacker exploits this vulnerability to launch denial-of-service attack.

(b) Inject Attack Code—Attacker can manipulate the code to(i) Overwrite the system data.(ii) Overwrite the data in the memory in such a way that it transfers the code to malicious code, i.e., pointer points to injected malicious code. Buffer Overflow vulnerabilities mostly dominate in the class of remote penetration attack.Figure1 shows the structure of memory organization of CPU. Here, text stores the code of the program, the data section consists of text and static variables, heap stores dynamic data, and stack section (shown in Fig.2) stores the dynamic local variables, parameters of the functions, return address of the function call (where the control will be transferred after the function executes), stack pointer points to the top of the stack. Stack grows from high address to low address (while buffer grows from low address to high address).

3. Working

3.1 Inside the memory



The top of the memory is the kernel area, which contains the command-line parameters that are passed to the program and the environment variables.

The bottom area of the memory is called text and contains the actual code, the compiled machine instructions, of the program. It is a read-only area, because these should not be allowed to be changed.

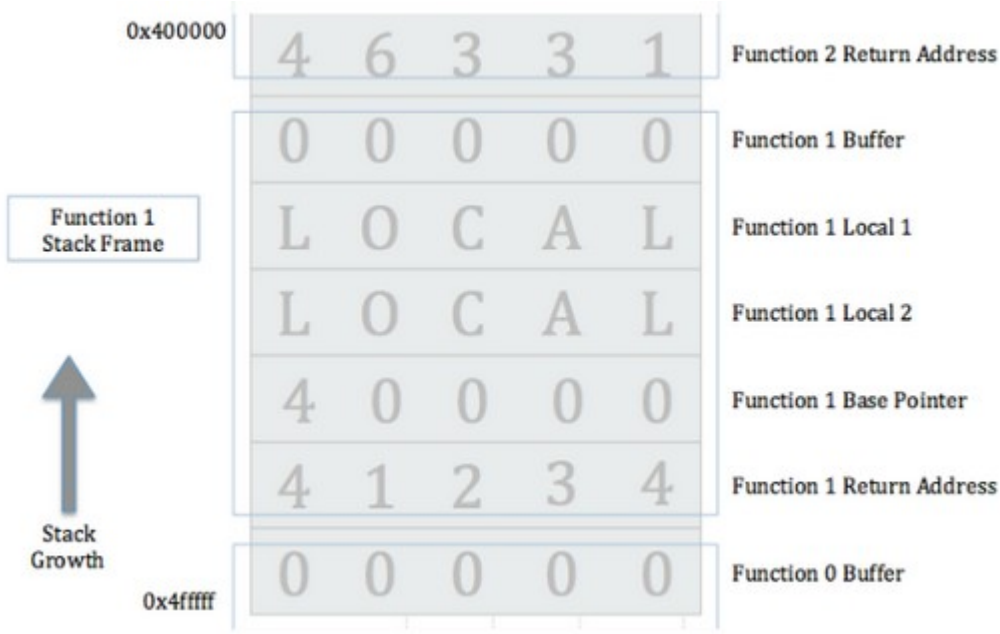
Above the text is the data, where uninitialized and initialized variables are stored.

On top of the data area, is the heap. This is a big area of memory where large objects are allocated (like images, files, etc.)

Below the kernel is the stack. This holds the local variables for each of the functions. When a new function is called, these are pushed on the end of the stack (see the stack abstract data type for more information on that).

3.2 Libraries

There are C library functions (e.g., gets(), strcpy(), or syslogd()) that operate on input without restriction, or bounds checking. These allow more data to be taken in by the program than intended. Excess input can lead to the overwriting of existing instructions and/or crashing the application all together. The main impetus for the C11 addition to the library was to create a stopping point, or bound, where no more input can be written to the memory space provided by the program. This is a sanitary means to control how much data can be collected and where that data will be stored.3 When a computer program requires user input, it creates memory space to store that information. As with any function variable, the memory space, or buffer, is inserted on the stack, as seen in Figure 2. This is placed above other functional data that tells the program what location to jump back or return to after it has finished the task.



3.3 The program

```c
#include <stdio.h>
#include <string.h>

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
}
```

At run-time, the contents of the argc and argv parameters, being parameters of the program, will be held in the kernel area.

The main-method is the entry point for the program and the first thing it does, is to call the func-function, passing the first command-line parameter to it (i.e. argv[1]).

Note that argv[0] contains a value that represents the program name (see also this stackoverflow post).

When calling the func-function, the argv[1] value is pushed onto the stack (as the name-parameter for func). When a function has multiple parameters, they are pushed onto the stack in reversed order. So with a call like foo(1, 2), '2' would be pushed first, followed by '1'.

Then, the function should know where to return when the function exits, so the address of the next instruction is pushed onto the stack as the return address. In this code

example, the next instruction is the one after func(argv[1]);, which is the memory address for line 14.

The EBP (or extended base pointer) is then pushed onto the stack. This pointer is used to refer to parameters and local variables. For the sake of keeping this explanation simple, we won't go further into details on this. However, this article has a good explanation of the EBP.

Then, a buffer of 100 bytes long is allocated in the stack, followed by a call to the string copy function (strcpy) which will copy the name-parameter into the buffer.

After this the contents of the buffer are output together with the welcome message.

3.4 Overflowing Buffer

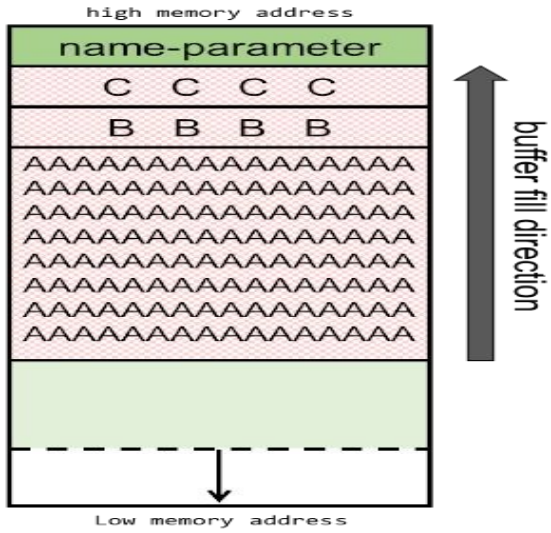
Let's see what happens when we execute the buf-program with a name-parameter of 108 characters long. The 108 characters will be built up by 100 times the letter 'A', 4 times a 'B', followed by 4 times the letter 'C'. Later, when inspecting the program's memory, this distinction of the letters will make it easier to identify the memory segments that were overwritten. We're expecting the first 100 A's to fill the buffer, the B's to overwrite the EBP and the C's to overwrite the return address.

In order to produce the string, I'll use the following python script: python -c 'print "\x41" * 100, which will generate a string with 100 times the character 'A' (0x41 is hexadecimal for 65, which is the ASCII-code for the letter 'A'). To those 100 characters, the four B's (0x42) and the four C's (0x43) will be added, producing a string with a total length of 108 bytes.

```
(gdb) run $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Starting program: /tmp/coen/buf $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

So the stack will look similar to this



The EBP-register contains the B's: 0x42424242 and the EIP-register contains the C's: 0x43434343. The EIP (Extended Instruction Pointer) contains the address of the next instruction to be executed, which now points to the faulty address.

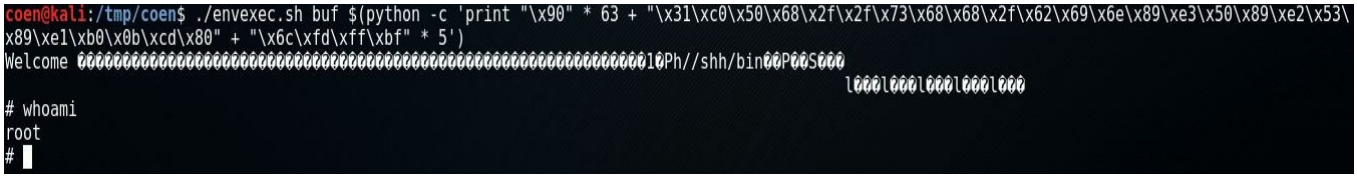
```
(gdb) info registers
eax            0x75     117
ecx            0x75     117
edx            0xb7fb3870       -1208272784
ebx            0x0      0
esp            0xbffffdc4       0xbffffdc4
ebp            0x42424242       0x42424242
esi            0x2      2
edi            0xb7fb2000       -1208279040
eip            0x43434343       0x43434343
eflags         0x10282  [ SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

3.5 Exploiting Code:

Now extract the 25 bytes of shellcode:

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe

2\x53\x89\xe1\xb0\x0b\xcd\x80



4. Buffer Overflow Mitigation

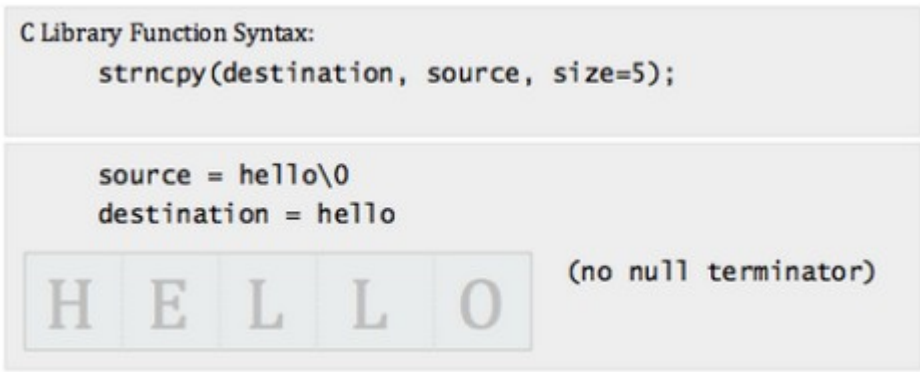
BUFFER OVERFLOW Mitigation concept of a buffer overflow has been around for nearly fifty years since theU.S. Air Force [7] published the “Computer Security Technology Planning Study "in October 1972. This paper outlined several different classes of computer attacks, one of which was titled, “Incomplete Parameter Checking.” This attack described a technique in which an attacker could provide an illegal program address as input and then redirect execution to that address. This was possible because there was no parameter validation. This may be the first development of the idea of gaining control of a program for unintended use via an injection. Just over twenty years later, there was a paper published in Phrack magazine called, “Smashing the Stack For Fun and Profit” [10]. In this paper, computer scientist Elias Levy (known by the handle, “Aleph One") defined the term “smash the stack” as being able to “corrupt the execution stack by writing past the end of an array declared auto in a routine” [10]. This is one of the reasons the modern-day buffer overflow exploit came into the public’s eye and began to be heavily analyzed and researched. Due to the fact that this concept has been widely studied for many years, there have been many fixes at the programming language and operating systems levels to address buffer overflows. Some of these fixes include address space layout randomization (ASLR), executable stack control, and stack canaries. Each of these fixes addresses a different aspect of the vulnerabilities that can lead to buffer overflow attacks. These 7 techniques of buffer overflow prevention will be briefly described and then will be followed with an explanation of how and why buffer overflow can still be a concern.ASLR is an operating-system protection against buffer overflow attacks. It works by ensuring that program code is loaded into a different section of memory every time it is executed, as well as providing randomized sections of memory for the stack, heap, and libraries. In the case where ASLR is implemented, when an attacker attempts to overwrite the return address of a function on the stack, he must also know the memory location he needs to point to in order to execute his code, which was dynamically determined at runtime. Unfortunately, there are ways of determining these

addresses. One way is by leaking memory locations with format string exploits. Other methods leak information via side channels. In 2016, researchers from the State University of New York and University of California, Riverside [11] published a means of subverting ASLR by using a side-channel attack on the branch target buffer (BTB). A second way is by utilizing a no-operation ("nop") slide/sled, which is a sequence of no-operation instructions that the machine will recognize but that will not execute any functionality. This allows the attacker to land in a range of addresses (where the "nops" exist), decreasing how specific he must be in his return address selection. Following the "nop" sled would be the attacker's code or a jump to such code. The attacker would still need to know what range of memory the code exists in (possibly via a memory leak) in order to choose a return address, but the precision required is decreased. A compiler-level defense against buffer overflow attacks is requiring that areas of memory not be writable and executable. One of these features in existence today is W^X,for "write XOR execute," which utilizes an NX bit, for "No eXecute." This allows entire pages of memory to be writable or executable but not both. Consider the following situation where this would be useful: An attacker writes code to the stack, in a dynamically allocated buffer. With the NX bit on, the attacker would not be allowed to execute such code, defeating his attempt to take control of the program. A rising workaround for non-executable stacks is ROP. Alternatively, the attacker could place hiscode in an unprotected region of memory, such as the heap, and cause it to execute fromthere.8 Another compiler-level buffer overflow prevention technique is the use of stack canaries or cookies. A canary is a random value placed onto the stack after (by order of operation) a return address and before a dynamically allocated buffer in a function. This Value is placed there by the operating system at runtime and is verified before the function returns. This value functions to prevent an attacker from arbitrarily overflowing the buffer, overwriting the canary, and overwriting the return address. For example, an attacker would overflow a buffer in order to overwrite the return address that exists below it on the stack. If a stack canary is utilized, the attacker will inadvertently overwrite this value as well. When the program goes to return using the return address on the stack (that the attacker has now replaced), it will attempt to verify the value that is located where it expects the stack canary to be. The attacker has overwritten this with his own code or junk instructions and, therefore, it will not be successfully verified. The program will then crash or exit. Unfortunately, these stack-canary values can also be

leaked or even brute-forced under certain circumstances. For example, when a process forks, the child process inherits the same canary. This situation can be exploited to leak information about the canary and/or to brute-force it. The attacker can try different values in the childprocess. If he fails, then a new child process is forked and a new value is tried. This Process continues until the child stops crashing. If the value can be obtained by the attacker, then it can be replaced in its original position and the program will not detect any corruption. The attacker proceeds with overwriting the return address following the canary.

5. Buffer Overflow Persistence

Despite integration of prevention techniques, buffer overflows have not been eradicated. One reason for their persistence is that, even after a programmer has enabled bounds checking to help prevent buffer overflow, he may misunderstand how input is handled thus still allowing vulnerabilities. Consider the code segment in Figure 5. When Using a function like strncpy(), the programmer must account for the input being null terminated, otherwise the code is left open to vulnerabilities.



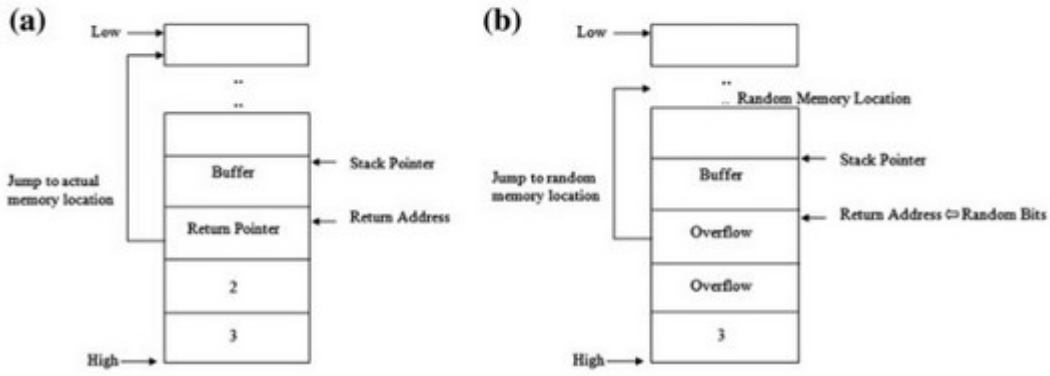
More scrupulous coding practices, keener understanding of how input is handled, and other mitigations mentioned above are key means for combating buffer overflows. Unfortunately, even with these practices employed within current code, many older devices contain outdated code. Part of the persistence of buffer overflows can be attributed to C's longevity; legacy applications utilize vulnerable functions. This is a lead actor in the industrial control system issues [18]. Because of the new trend of networking these systems, their exploitation has been on the rise over the past decade.SCADA, HMI, and controllers all fall victim to even the most trivial programming errors.12 In 2015, Kaspersky Lab [18] did a review of industrial control system (ICS)vulnerabilities

worldwide. Of the 189 vulnerabilities reported in ICS components that year, “the most widespread types are buffer overflows,” accounting for 9% of all detected vulnerabilities. Of these 17 vulnerabilities, more than half of them (8) were rated as high risk according to the Common Vulnerability Scoring System (CVSS) and half of those(4) had the highest risk level of 10. These software systems either do not take advantage of modern buffer overflow prevention techniques, or they do have protections but they are subverted by hackers via attack vectors, such as those previously described. Of note, at least a half dozen software weaknesses that can lead to buffer overflows are listed in MITRE’s Common Weakness Enumeration (CWE) database [19]. From these weaknesses, stem hundreds of fully realized buffer overflow vulnerabilities as reported in the Common Vulnerability Enumerations (CVE) database [20] in the past year alone. This translates to potentially thousands in the history of buffer overflows. (This CVEdatabase, maintained by MITRE, contains all publicly known and reported software vulnerabilities.) Many of these vulnerabilities have been patched and removed with software updates, but this gives an idea of how commonplace they are and how they might remain in software that is no longer tended to. There are alternatives to programming in C, such as using the Java and Perl Languages. They are not susceptible to buffer overflows because they implement inherent bounds checking through their use of arrays. In these languages, there are checks for abnormal conditions (i.e., exceptions) [21], which monitor data as it is put into memory. In these cases, bounds are set before memory space is allocated. If entry into this space is exceeded, the ArrayIndexOutOfBoundsException error is thrown. To allow C this type of security, these exceptions can be built within the compiler. Yet, these are not widely adopted due to the increase in overhead incurred [6]. When handling large amounts of data, these inefficiencies have great effect. Therefore, instead of switching languages,novel C -specific safeguards need to be created. In this paper, we hope to provide just that,a robust, efficient guard against buffer overflows.

6. Goals of Buffer Overflow Attack to Exploit Vulnerability

6.1 Jump to Random Memory Location

```
void area (int a, int b);
void main()
{
        area (2, 3);
}
void area (int a, int b)
{
        int buffer[10];
}
```



Attacker overwrites program with arbitrary sequence of byte with goal of corrupting the victim program. It is done by making the victim's pointer point to a random address. According to Table2, if an attacker tries to use more memory (>10), buffer overflow will overflow into the space where the return address is located. Attackers can overwrite this return address with the random bits or random address; by this, the program will jump to random memory location after function execution [4] and may lead to program crash as shown in Fig.3b.

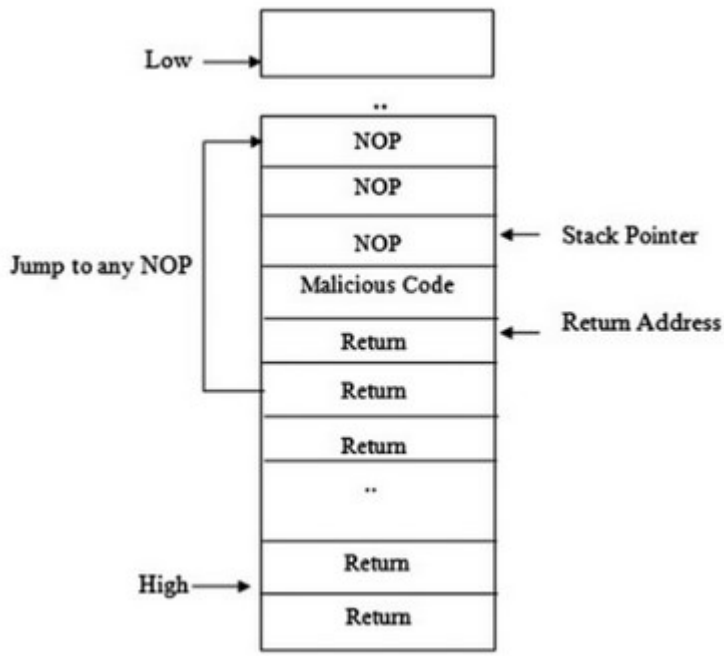
6.2 To Place Malicious Code in Victim's Program

Address Space Stack and heap are two areas of memory that a program used for reading and writing, i.e., buffer can be located in any of these two areas. Attackers provide data as input to the program to store in a buffer. This data is actually the instruction with the help of which attackers try to use victim program's buffer to store the malicious code of

his/her choice. Attacker injects this executable malicious code into the buffer and overwrites the return address with the address of this malicious code as shown in Fig.4b. This Return address can be chosen by hit-and-trial method.

7. Challenges in Buffer Overflow

There are some difficulties with buffer overflow attack, they are [5](a) Attacker may not know the exact location of malicious code injected.(b) Attacker may not know the exact location of the return address with malicious code starting address.1.3.2 These difficulties can be overcome by different methods:(a) First problem can be solved by injecting No Operation (NOP) before malicious code.(b) Second problem can be resolved by inserting the return address repeatedly. This may overwrite the actual return address with attacker's return address and will make pointer jump to any NOP address which in turn may point to next NOPand after last NOP malicious code will be executed (Fig.5)



8. Research Methodology

8.1 Protection Against Buffer Overflow

There are some ways to protect the software from buffer overflow:(a) Brute force method —to write completely the correct code but to write an error-free code is not achievable. One of the ways to achieve near to error-freecode is to introduce buffer overflow intentionally to search for vulnerable components in the program. The other way is to use debugging and analysis tools to find buffer overflow vulnerabilities. This method does not eliminate all the vulnerabilities but reduces them [6].(b) Do not allow the code to execute on stack; stack is made non-executable by using No execute bit or NX bit (supported by some hardware); memory can be flagged so that code cannot be executed in a specified location.(c) Safe program language—Java and C# have boundary checks at runtime. They Automatically check the arrays out of bound. These languages do not allow memory locations to be accessed which are out of boundary but have perfor-mance penalty for checking, due to which developer chooses C language. So in that case do not use unsafe functions, use its safe alternative. Use safe functions such as fgets, strncpy, strncat, and snprintf instead of C unsafe functions gets,strcpy, strcat, sprintf, scanf, etc. [7].(d) Runtime Stack Checking—Runtime stack checking can be introduced by pushing special value on the stack after return address. When the return address is popped off stack, the special value can be used to verify that the return address has not changed and in order to overwrite the return address, this special value also needs to be overwritten.

8.2 INSTRUCTION MODIFICATIONS

Our research employs two different methods of binary modification. The first method is to alter bytes of the target binary. We refer to this as “replace_instructions.” This involves overwriting existing bytes with desired bytes in order to provide new functionality. For example, the x86-64 instruction (in hexadecimal) “e8 98 f9 ff” (“calloffset”) could be overwritten as “e8 eb a1 01 00” (“callnew_offset”). These two instructions each contain five bytes. No bytes are added or removed from the size of the program and no instructions had their program addresses changed. The second method of binary modification involves adding new bytes where bytes did not exist or, in other

words, inserting new bytes/instructions in between already existing bytes/instructions. We refer to this functionality as “insert_instructions.” This involves opening up a space at the desired program address by shifting all of the following bytes in the program memory space down by the inserted number of bytes. For example, we might want to insert the bytes ``68 e0 87 40 00” (“pushaddress”) right before a call instruction. This Adds five new bytes to the program, so all instructions after this added one need to have their addresses shifted down in memory by five bytes. If the added instruction “pushaddress” occurs at memory address 0x408e43, then the instruction that used to exist at this address (“calloffset”) would now have the address 0x408e48(0x408e43 + 5). It is no small task to make these kinds of changes to the memory map of a compiled program. 20 There does exist, however, a Python library called mmap[26] that can be utilized to do this [27]. The Python library map provides memory mapped file I/O support in Python 3. With this, we are able to load the entire contents of the target binary file into memory and manipulate it directly, shifting and adding bytes as desired, which, in turn, has the effect of changing the file size. Two functions are written in order to replace and insert byte into the binary. The first function, insert_instructions, involves inserting a specified number of bytes into the file at a specified offset (Figure 6)

```
def insert_instructions(address, instructions):
    if address >= 0x400000:
        offset = address - 0x400000
    else:
        offset = address
    if type(instructions) == str:
        instructions = bytes.fromhex(instructions)

    insertIntoMmap(offset, instructions)
```

This function does some calculations to determine if the value passed to it is a file address or offset (offset is required), converts them to byte form if necessary (if passed as a string), and then callsinsertIntoMmap, which manipulates the memory mapped file data to insert the specified bytes. As an example, say we have a file of size 100 decimal(100d) bytes, and we wish to insert 25d new bytes into the file at offset 75d. WheninsertIntoMmapis called, it is passed the offset at which to place the bytes (75d in

our example) and the data to be inserted at that location. It will then acquire the current file size and the number of bytes to be added, and it will calculate the new file size. Once it has done this, we seek the desired offset into the file object and write the data. We then manipulate the memory-mapped file by moving byte 75d down to position 100d, seek to position 75d, and write our 25d bytes. We now have a file that has the original bytes 0d to21 74d, new bytes 75d to 99d, and original bytes that used to be at offsets 75d to 99d now at offsets 100d to 124d. The second function, replace_instructions, involves deleting a specified number of bytes from the file and writing new bytes in their place (Figure 7)

```
def replace_instructions(address, instructions):
    if address >= 0x400000:
        offset = address - 0x400000
    else:
        offset = address
    if type(instructions) == str:
        instructions = bytes.fromhex(instructions)

    deleteFromMmap(offset, offset+len(instructions))
    insertIntoMmap(offset, instructions)
```

This function first calls another function, delete From map, which manipulates the memory-mapped file. As another example, say we have a file size of 100d bytes, and we wish to replace bytes 50d to 74d with some other data. The first step will be to delete those bytes from the memory map of the file.delete From calculates the number of bytes to remove based on the starting and ending file offsets that are passed to it. In our example, the number of bytes to be removed will be 75d - 50d = 25d. It calculates the new file size after the bytes are removed. In our example, this will be 100d - 25d = 75d. It then moves bytes 75d to 100d up to the location of byte 50d, making byte 75d now byte50d and effectively deleting bytes 50d to 74d from the original file. Finally, it truncates the file object by calling the truncate() function on the file descriptor for the file we are working with. Once the desired bytes are deleted, thereplace_instructions function thencallsinsertIntoMmap. As previously described, this function manipulates the memory mapped file to insert the desired bytes. To continue with our example, we want to insert 25d new bytes at offset 50d, which will make the offsets of these added bytes 50d to 74d, as in the original file.insertIntoMmapis passed the offset to place the bytes (50d in our22

example) and the data to be inserted at that offset. It then uses the length of the data and the current size of the file to calculate the new file size. Once it has done this, we seek the desired offset of the file object and write the data. We then manipulate the memory mapped file by moving byte 50d down to position 74d, seek to position 50d, and write our 25d bytes. We now have a file that has the original bytes 0d to 49d, new bytes 50d to 74d, and original bytes 75d to 99d. If a file is modified in the way that was just described, so as to end up with a net gain of bytes (in our concept this is nine bytes), we decide to preserve the original file size. This helps maintain some of the integrity of the original program by reducing the number of header values that need to be updated (i.e., file size values). In order to achieve this, we have a final call to a function, delete_extra_bytes (Figure 8)

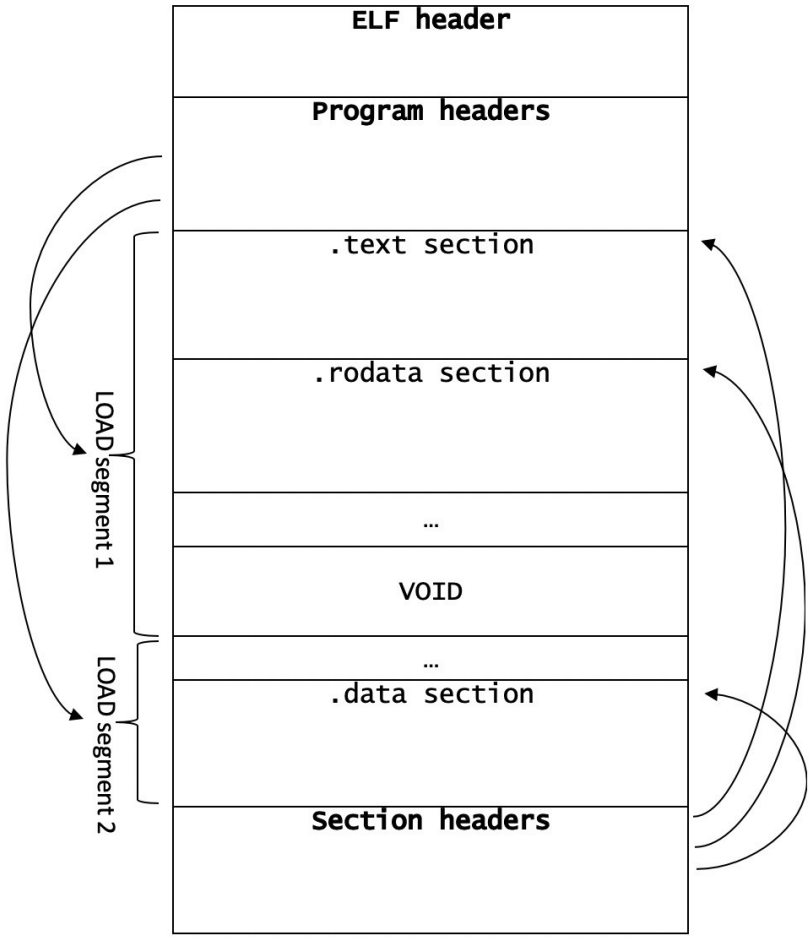
```
def delete_extra_bytes(address, num_bytes):
    if address >= 0x400000:
        offset = address - 0x400000
    else:
        offset=address

    deleteFromMmap(offset, offset+num_bytes)
```

This function performs the same data validation checks (converting an address to an offset as necessary) and then calls delete From map which performs as previously described. In this case, the offset that is passed as the location to delete bytes from is part of the extended .text section that is not utilized.

8.3 BINARY EXTENSION

A secondary goal of this research is to implement the extending of the .textsection and the updating of the necessary headers, independent of the extension that is made with the Rose compiler. The first step in performing a .text section extension is to conduct a dependency analysis, locating all the headers that would be affected by adding bytes into the middle of the binary. (For purposes of this discussion, assume N bytes are added to the binary, where $N \in I$ and $N \geq 1$.) See Figure 16 for the ELF binary structure.



The ELF header is located first as it gives additional information about how to locate the rest of the headers. The ELF header always starts at offset zero in the binaryand is often a constant size of 64 bytes, as is the case for our work with Nano. The ELF30 header contains information such as the program header offset, the section header offset,the program headers' size, the number of program headers, the section headers' size, the number of section headers, and the section header string table index. The section headers are modified first. The first section header that needs changes is that of the .text section, in which we update the size value by adding N bytes. Every Action that occurs after the .text section and up to the start of the next program segment(e.g., .fini and .rodata), needs to have its starting address and offset shifted by the N bytes added to the .text section. The sections that exist in a new segment (e.g., starting afteraddress0x620000 in Nano) do not need to be modified, because there is empty, unused space between segments that we refer to as the VOID. This consists of unused null bytes at the end of the first program segment (that is aligned with address 0x400000). In order to preserve the addressing of sections in this second segment (e.g., .dynamic, .got, and.bss), we

remove bytes from this VOID area. After dealing with the section headers, we need to modify/update the program headers. Updated program headers include the first LOAD section, which contains our updated .text section, as well as any program headers that exist after the .text section and before the VOID. The load section contains a value for the size of its entry and the size of the memory space that it utilizes. As for the program headers that exist in the binary after the .text section and before the VOID (e.g., PT_GNU_EH_FRAME in Nano), their starting file offsets as well as their virtual and physical starting addresses are updated. Each of these modifications to values in the binary is performed using our map functionality that was described in Chapter III, Section A. The extended binary is then run through the UHaul program in order to get a patched executable.

CONCLUSION

Buffer overflow is still biggest security problems in software and web applications, respectively, that will exist in future for long time due to large amount of legacy code. Although various buffer overflow detection techniques have been proposed, there are few studies on comparing the effectiveness and efficiency of state-of-the-art static analysis techniques. This paper explains buffer overflow attack vulnerabilities and the preventives measures that can be taken to protect it from the attackers. we present an empirical study on both the detection and fixing of buffer overflow bugs. We also investigated the distribution of buffer overflow bugs, as well as the fixing strategies for different buffer overflow bugs.

REFERENCES

[1] T. Ye, L. Zhang, L. Wang and X. Li, "An Empirical Study on Detecting and Fixing Buffer Overflow Bugs," 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, 2016, pp. 91-101, doi: 10.1109/ICST.2016.21.

[2] M. Ma, L. Chen and G. Shi, "Dam: A Practical Scheme to Mitigate Data-Oriented Attacks with Tagged Memory Based on Hardware," *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Putrajaya, Malaysia, 2019, pp. 204-211, doi: 10.1109/APSEC48747.2019.00036.

[3] Shi, Junjing (Shanghai, CN), Long, Qin (Shanghai, CN), Gao, Liming (Shanghai, CN), Rothman, Michael A. (Puyallup, WA, US), Zimmer, Vincent J., (Federal Way, WA, US) 2020, METHODS AND APPARATUS TO PROTECT MEMORY FROM BUFFER OVERFLOW AND/OR UNDERFLOW, United States, Intel Corporation (Santa Clara, CA, US), 20200125497

[4] Zhiyong Jin, Yongfu Chen, Tian Liu, Kai Li, Zhenting Wang, and Jiongzhi Zheng. 2019. A Novel and Fine-grained Heap Randomization Allocation Strategy for Effectively Alleviating Heap Buffer Overflow Vulnerabilities. In Proceedings of the 2019 4th International Conference on Mathematics and Artificial Intelligence (ICMAI 2019). Association for Computing Machinery, New York, NY, USA, 115–122. DOI:https://doi.org/10.1145/3325730.3325738

[5] Yang Zhao, Xingzhong Du, Paddy Krishnan, and Cristina Cifuentes. 2018. Buffer overflow detection for C programs is hard to learn. In Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA '18). Association for Computing Machinery, New York, NY, USA, 8–9. DOI:https://doi.org/10.1145/3236454.3236455

[6] L. Xu, W. Jia, W. Dong and Y. Li, "Automatic Exploit Generation for Buffer Overflow Vulnerabilities," 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Lisbon, 2018, pp. 463-468, doi: 10.1109/QRS-C.2018.00085.

[7] Khurana M., Yadav R., Kumari M. (2018) Buffer Overflow and SQL Injection: To Remotely Attack and Access Information. In: Bokhari M., Agrawal N., Saini D. (eds) Cyber Security. Advances in Intelligent Systems and Computing, vol 729. Springer, Singapore

[8] Rogers, Alexis, L. and Sowers, Ryan, 2019 Calhoun, DEFENSIVE BINARY CODE INSTRUMENTATION TO PROTECT AGAINST BUFFER OVERFLOW ATTACKS, In Monterey, CA; Naval Postgraduate School.

[9] Al Sardy L., Saglietti F., Tang T., Sonnenberg H. (2018) Constraint-Based Testing for Buffer Overflows. In: Gallina B., Skavhaug A., Schoitsch E., Bitsch F. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2018. Lecture Notes in Computer Science, vol 11094. Springer, Cham

[10] Ning Huang et al 2019 IOP Conf. Ser.: Earth Environ. Sci.252 042100

BASE PAPER FIRST PAGE

PLAGIARISM CHECK REPORT