A Study of Buffer Overflow Exploit and its prevention techniques

Shreyansh Kumar, Sunil Sikka

Amity Institute of Information Technology, Amity University Haryana

ABSTRACT

[1] On the November 2, 1988, there was one of the first computer worms distributed via the Internet, and this exploited the vulnerability from the inside of the victim system and gave access to the attacker, it exploited A hole in the debug mode of the Unix sendmail program and A buffer overflow in the fingerd network service, then in June 2000 a major new type of vulnerability called "format string" was discovered and since then these Memory error type vulnerabilities have continued to compromise the security of today's systems severely. This paper has the intention of shedding light over these memory error type vulnerabilities which have continued to compromise major organizations like **[2]** Apple who disclosed three actively exploited vulnerabilities in the year 2020 in its iOS, two of which were in the browser engine that powers Safari, and the third a buffer overflow in the Kernel. Apple has released iOS 14.4 with security fixes for three of the vulnerabilities.

1. INTRODUCTION

Programming mistakes that permit the defilement of basic bits of program memory, for example, stack and heap buffer overflows, remain a prevalent problem, these types of vulnerabilities are exploited frequently in today's day and time. Think of the most secured operating system if you think of Linux, you might get disappointed because recently a buffer overflow vulnerability in the **[3]** sudo of Linux was discovered and sudo is as old as Linux itself this vulnerability has always been there, and it gave a reverse connection to the attacker. This paper is going to be discussing about the general approach of looking at these vulnerabilities and will try to give you some sort of understanding about the memory error vulnerability or buffer overflow vulnerability. The first section will be discussing about the types and how an exploit functions on different parameters like what is a stack buffer overflow and how does a format string bug work and in the last section this paper has a practical demonstration of a stack buffer overflow of a simple C code. Buffer overflows on the stack are well-studied and have a long history of being exploited. The basic strategy is to overflow a local buffer on the stack with input data until it overwrites a code pointer (typically the return address). An arms race of ever-more sophisticated defences and attacks has led to stack exploits becoming increasingly difficult to execute against hardened programs.

Memory can be exploited in several ways the main part of exploiting a memory is reaching the buffer limit and then bypassing it to reach the OS execution level which can be done using different memory exploits.

Exploits are generally categorized using the following criteria:

1.1 Stack based exploits

[4] A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO. Several operations are defined on stacks. Two of the most important are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, reduces the stack size by one by removing the last element at the top of the stack

Attackers may manage to push some malicious code on the stack which may redirect the flow of the program and execute the malicious program which the attacker wants to execute This is done by overwriting the return pointer so that the flow of control can be passed to malicious code).

1.2 Heap based exploits

[5] Heaps stores the data which is dynamically created with functions, such as malloc ()

As this paper has discussed by now, heap is FIFO (First In First Out) and it grows towards the higher memory address in the memory, when an attacker enters the data in heap, it overflows the adjacent data that is in the lower part of the heap.

If any application takes an input from the user and copies it in the memory without checking it, a heap overflows may occur.

1.3 Format string exploits:

[6] Format String Exploits are used to crash a software or to execute a malicious code. This attack is possible because the user input is filtered as a format string parameter in certain C functions that perform formatting, such as printf(). A malicious user may use the %s and %x format tokens, among others, to print data from the stack or possibly other locations in memory. printf FormatString Vulnerabilities was the first format string bug found in 1999 by Tymm Twillman while auditing the source code for ProFTPD1.2.0pre6.

1.4 Integer bug exploits

[7] Integer bugs are exploited by passing an oversized integer to an integer variable. It may cause overwriting of valid program control data resulting in execution of malicious codes.
Each integer type in C has a fixed minimum and maximum value that depends on the type's machine represent (e.g., two's complement vs. one's complement), whether the type is signed or unsigned (called "signedness"), and the type's width (e.g., 16-bits vs. 32-bits). At a high level, integer vulnerabilities arise because the programmer does not take into account the maximum and minimum values. Integer vulnerabilities can be divided into four categories: overflows, underflows, truncations, and sign conversion errors. Integer bugs are very often found and exploited.

2. Memory Management and addressing

2.1 Importance

this paper discusses about vulnerabilities of a software so for the sake of understanding how a software works it is important to discuss how a software/process is laid out in the memory. Whenever a program is executed in a computer It is first laid out in an organized manner, various elements of the program are mapped into memory. First, the Operating System creates an address space in which the program will run. This address space includes the actual program instructions as well as any required data. Next, information is loaded from the program's executable file to the newly created address space. There are three types of segments:

.text

.bss

.data

little knowledge of assembly can help in understanding this paper better. The processes are divided into three regions: Text, Data, and the Stack. The text region is fixed by the program and includes code, or we can say the instructions and the read-only data. This region corresponds to the text section of the executable file. This region is read-only and any attempt to write to it will result in a segmentation violation or segmentation fault which you might remember from Operating System that segmentation fault happens when a program or software or any code tries to exceed the limit of memory that the operating system has given to it, when it tries to enter the territory of the operating system than segmentation fault happens.

A modern computer makes no real distinction between instructions and data. If a processor can be fed instructions when it should be seeing data, it will happily go about executing the passed instructions. This characteristic makes system exploitation possible. This book teaches you how to insert instructions when the system designer expected data. You will also use the concept of overflowing to overwrite the designer's instructions with your own. The goal is to gain control of execution.

The Data region contains initialized and uninitialized data. Static variables are stored in this region. The data region corresponds to the data-bss section of the executable file.

If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.

Till now process's memory architecture has been discussed in this paper, the next section will discuss the stack memory and its working which is the key for exploitation.

2.2 Stack

The stack is a data structure, more specifically a Last in First out (LIFO) data structure, which means that the most recent data placed, or pushed, onto the stack is the next item to be removed, or popped, from the stack. Stack is a contiguous block of memory containing data. In x_86_64 systems there is ESP register meaning Stack Pointers which point to the top of the stack in the memory, and we have EIP register meaning Instruction Pointer which means that what instruction will run next.

Computer registers help the exploiter understand the architecture of software since the registers are like variables to a computer all the data is being passed through registers for processing and after processing. Next section of the paper will discuss about the processing of data that happens while a software/code runs.

2.3 Data bus

A data bus is a data-centric software framework for distributing and managing realtime data in intelligent distributed systems. It allows applications and devices to work together as one, integrated system. The data bus simplifies application and integration logic with a powerful data-centric paradigm.

2.4 Instruction Decoder

The Instruction Decoder is a CPU component that decodes and interprets the contents of the Instruction Register, i.e., its splits whole instruction into fields for the Control Unit to interpret. The Instruction decoder is often considered to be a part of the Control Unit.

2.5 Registers

Understanding how the registers work on an IA32 processor and how they are manipulated via assembly is essential for vulnerability development and exploitation. Registers can be accessed, read, and changed with assembly. Registers are memory, usually connected directly to circuitry for performance reasons. They are responsible for manipulations that allow modern computers to function and can be manipulated with assembly instructions. From a high level, registers can be grouped into four categories:

General purpose

These are used to perform a range of common mathematical operations. They include registers such as EAX, EBX, and ECX for the IA32, a6nd can be used to store data and addresses, offset addresses, perform counting functions, and many other things. A general-purpose register to take note of is the extended stack pointer register (ESP) or simply the stack pointer. ESP points to the memory address.

Segment

Segment registers are basically memory pointers located inside the CPU. Segment registers point to a place in memory where one of the following things begins: Data storage. Code execution.

Control

Control registers are used to control the function of the processor. The most important of these registers for the IA32 is the Extended Instruction Pointer (EIP) or simply the Instruction Pointer. EIP contains the address of the next machine instruction to be executed. Naturally, if you want to control the execution path of a program, which is incidentally what this paper is all about, it is important to have the ability to access and change the value stored in the EIP register.

3. Exploiting C

The next section of this paper is the climax and the most important section. Stack overflows are possible because no inherent bounds checking exists on buffers in the C or C++ languages. In other words, the C language and its derivatives do not have a built-in function to ensure that data being copied into a buffer will not be larger than the buffer can hold. Consequently, if the person designing the program has not explicitly coded the program to check for oversized input, it is possible for data to fill a buffer, and if that data is large enough, to continue to write past the end of the buffer. As you will see in this chapter, all sorts of crazy things start happening once you write past the end of a buffer. Look at this extremely simple example that illustrates how C has no bounds- checking on buffers.

```
#include <stdio.h>
#include <string.h> int main ()
{
int array[5] = {1, 2, 3, 4, 5};
printf("%d\n", array[5] );
}
```

In this example, there is an array in C. The array is named array and it is five elements long. There is a novice C programmer mistake here, in that programmer forgot that an array of size five begins with element zero, array[0], and ends with element four, array[4]. Programmer tried to read what he thought was the fifth element of the array, but the user was really reading beyond the array, into the "sixth" element. The gcc compiler elicits no errors, but when run this code, it gives unexpected results:

```
shellcoders@debian:~/chapter_2$ cc buffer.c
shellcoders@debian:~/chapter_2$ ./a.out
134513712
```

This example shows how easy it is to read past the end of a buffer; C provides no built-in protection. What about writing past the end of a buffer? This must be possible as well. Let's intentionally try to write way past the buffer and see what happens:

```
int main ()
{
int array[5]; int i;
for (i = 0; i <= 255; i++ )
{
array[i] = 10;
}
}
```

Again, the compiler gives no warnings or errors. But, when we execute this program, it crashes.

```
shellcoders@debian:~/chapter_2$ cc buffer2.c
shellcoders@debian:~/chapter_2$ ./a.out
Segmentation fault (core dumped)
```

As you might already know from experience when a programmer creates a buffer that has the potential to be overflowed and then compiles and runs the code, the program often crashes or does not function as expected. The programmer then goes back through the code, discovers where he or she made a mistake, and fixes the bug. Let us have a peek at the core dump in gdb.

```
shellcoders@debian:~/chapter_2$gdb-q-c
coreProgramterminatedwithsignal11,Segmentationfault.#00x0000000ain??()
```

Interestingly, it is seen that the program was executing address at 0x0000000a or 10 in decimal when it crashed. So, what if user input is copied into a buffer Or, what if a program expects input from another program that can be emulated by a person, such as a TCP/IP network aware client? If the programmer designs a code that copies user input into a buffer, it may be possible for a user to intentionally place more input into a buffer than it can hold. This can have several different consequences, everything from crashing the program to forcing the program to execute user-supplied instructions. These are the situations we are chiefly concerned with.

4. Countermeasures

It is believed that if buffer overflows happen there is a way to control the application even if page protections or other mechanisms forbid for directly executing shellcode. The reason is that due to the complex nature of today's applications a lot of the shellcode is already within the application itself. SSH servers for example already carry code to execute shell because it's the programs aim to allow remote control. Nevertheless, I will discuss two mechanisms which might make things harder to exploit.

4.1Address Space Layout Randomization

[8] The code chunks borrow technique is an exact science. As you see from the exploit no offsets are guessed. The correct values have to be put into the correct registers. By mapping the libraries of the application to more or less random locations it is not possible anymore to determine where certain code chunks are placed in memory. Even though there are theoretically 64-bit addresses, applications are only required to handle 48-bit addresses. This shrinks the address space dramatically as well as the number of bits which could be randomized. Additionally, the address of a appropriate code chunk has only to be guessed once, the other chunks are relative to the first one. So, guessing of addresses probably still remains possible.

4.2 Register flushing

[9] At every function outro a xor %rdi, %rdi or similar instruction could be placed if the ELF64 ABI allows so. However, as shown, the pop instructions do not need to be on instruction boundary which means that even if you flush registers at the function outro, there are still plenty of usable pop instructions left. Remember that a pop%rdi; retq sequence takes just two bytes.

5 Conclusion

No tool can completely solve the buffer overflow problem, but with certain tools and skills it can increase the probability that a buffer overflow is detected and mitigated before it causes any damage to the users. There are a lot of countermeasures that organizations have taken for mitigating these vulnerabilities but some way or other all the countermeasures fail like SEH (Structured Exception handling) which can be bypassed by using the address of SE handler and all the countermeasures have this issue they all can be bypassed some or the other way but the purpose of this paper was to make everyone aware about the dangers of technology and to stay safe on the internet.

References

[1] Government, United States, The Morris Worm, www.fbi.gov, November,2,2018,

https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218

[2] Apple, support.apple, https://support.apple.com/en, March,26,2021, https://support.apple.com/en-us/HT212256

[3] Samedit, Baron, LiveOverflow/pwnedit, github.com, August,11,2021, https://github.com/LiveOverflow/pwnedit

[4] One, Aleph, Smashing The Stack For Fun And Profit, Phrack, 1996, https://www.cis.upenn.edu/~sga001/classes/cis331f19/resources/stack-smashing.pdf

[5] Modular Synthesis of Heap Exploits, Dusan Repel, Johannes Kinder, Lorenzo Cavallaro

https://dl.acm.org/doi/10.1145/3139337.3139346

[6] FormatGuard: Automatic Protection From printf Format String, Vulnerabilities, Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman, Mike Frantzen Jamie , Lokier

https://www.usenix.org/legacy/events/sec01/full_papers/cowanbarringer/cowanbarringer.pdf

[7] RICH: Automatically Protecting Against Integer-Based Vulnerabilities

David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, Dawn Song
https://kilthub.cmu.edu/articles/RICH_Automatically_Protecting_Against_Integer-Based_Vulnerabilities/6469253/files/11897807.pdf

[8]An Analysis of Address Space Layout Randomization on Windows Vista

Ollie Whitehouse, Architect, Symantec Advanced Threat Research

https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf

[9]x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique

Sebastian Krahmer September 28, 2005

https://trailofbits.github.io/ctf/exploits/references/no-nx.pdf