BUFFER OVERFLOW ATTACKS AND DEFENSES

Parth Laturia-180050071
Rajat Jain-180100091
Devansh Chandak-180110027
Anish Deshpande-180100013
Sunil Meena-180050107

CONTENTS

STACK BASED BUFFER OVERFLOW ATTACKS

1. Introduction

Memory copying is quite common in programs, where data from one place (source) need to be copied to another place (destination). Before copying, a program needs to allocate memory space for the destination. Sometimes, programmers may make mistakes and fail to allocate sufficient amounts of memory for the destination, so more data will be copied to the destination buffer than the amount of allocated space. This will result in an overflow. Some languages, like Java, automatically detect the problem when a buffer is over-run, but other languages such as C and C++ are not able to detect it.

Most people may think that the only damage a buffer overflow can cause is to crash a program, due to the corruption of the data beyond the buffer; however, what is surprising is that such a *simple mistake may enable attackers to gain a complete control of a program*, rather than simply crashing it. If a vulnerable program runs with privileges, attackers will be able to gain those privileges. In this section, we will explain how such an attack works.

2. Copying Data to Buffer

There are many functions in C that can be used to copy data, including strcpy(), strcat(), memcpy(), etc. In the examples of this section, we will use strcpy(), which is used to copy strings. An example is shown in the code below. The function strcpy() stops copying only when it encounters the terminating character '\0'.

```
#include <string.h>
#include <stdio.h>
void main ()
{
        char src[40]="Hello world \0 Extra string";
        char dest[40];

        strcpy (dest, src);     // copy to dest (destination) from src (source)
}
```

When we run the above code, we can notice that strcpy() only copies the string "Hello world" to the buffer dest, even though the entire string contains more than that. This is because when making the copy, strcpy() stops when it sees number zero, which is represented by '\0' in the code. It should be noted that this is not the same as character

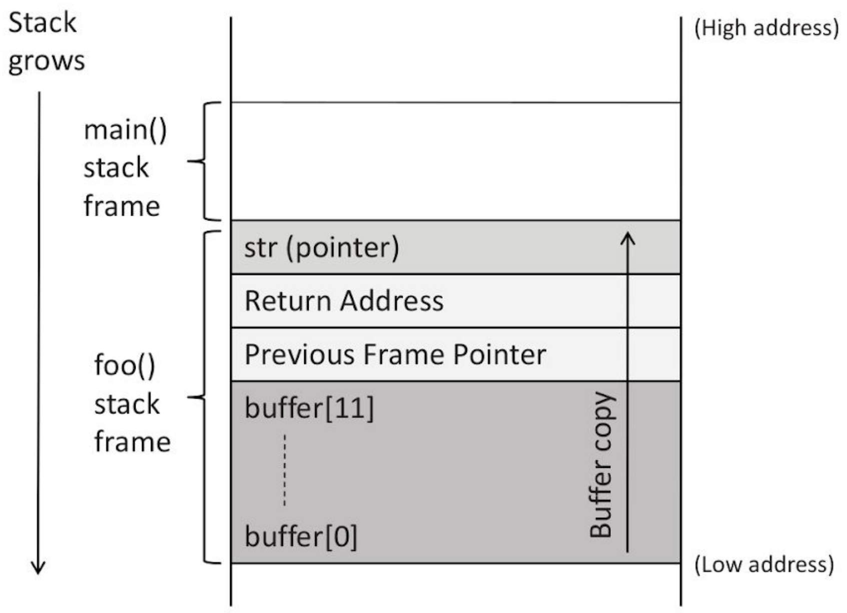
'0', which is represented as 0x30 in computers, not zero. Without the zero in the middle of the string, the string copy will end when it reaches the end of the string, which is marked by a zero (the zero is not shown in the code, but compilers will automatically add a zero to the end of a string).

3. Buffer Overflow

When we copy a string to a target buffer, what will happen if the string is longer than the size of the buffer ? Let us see the following example.

```
#include <string.h>
void foo(char *str)
{
        char buffer[12];
        strcpy(buffer, str);          // The following statement will result in buffer overflow
}
int main()
{
        char *str = "This is definitely longer than 12";
        foo(str);
        return 1;
}
```

Stack Layout of above code:



The local array buffer[] in foo() has 12 bytes of memory. The foo() function uses strcpy() to copy the string from str to buffer[]. The strcpy() function does not stop until it sees a zero (a number zero, '\0') in the source string. Since the source string is longer than 12 bytes, strcpy() will overwrite some portion of the stack above the buffer. This is called ***buffer overflow.***

Note: Stacks grow from high address to low address, but buffers still grow in the normal direction (i.e., from low to high). Therefore, when we copy data to buffer[], we start from buffer[0], and eventually to buffer[11]. If there are still more data to be copied, strcpy() will continue copying the data to the region above the buffer, treating the memory beyond the buffer as buffer[12], buffer[13], and so on.

Consequence:
As can be seen in Figure 4.4, the region above the buffer includes critical values, including the return address and the previous frame pointer. The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place.

Several things can happen:

- The new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash.
- The address may be mapped to a physical address, but the address space is protected, such as those used by the operating system kernel; the jump will fail, and the program will crash.
- The address may be mapped to a physical address, but the data in that address is not a valid machine instruction (e.g. it may be a data region); the return will again fail and the program will crash.
- The data in the address may happen to be a valid machine instruction, so the program will continue running, but the logic of the program will be different from the original one.

4. Exploitation

As we can see from the above consequence, by overflowing a buffer, we can cause a program to crash or to run some other code. From the attacker’s perspective, the latter sounds more interesting, especially if we (as attackers) can control what code to run, because that will allow us to hijack the execution of the program. If a program is privileged, being able to hijack the program leads to privilege escalation for the attacker.

Let us see how we can get a vulnerable program to run our code. In the previous program example, the program does not take any input from outside, so even though there is a buffer overflow problem, attackers cannot take advantage of it. In real applications, programs usually get inputs from users. See the following program example.

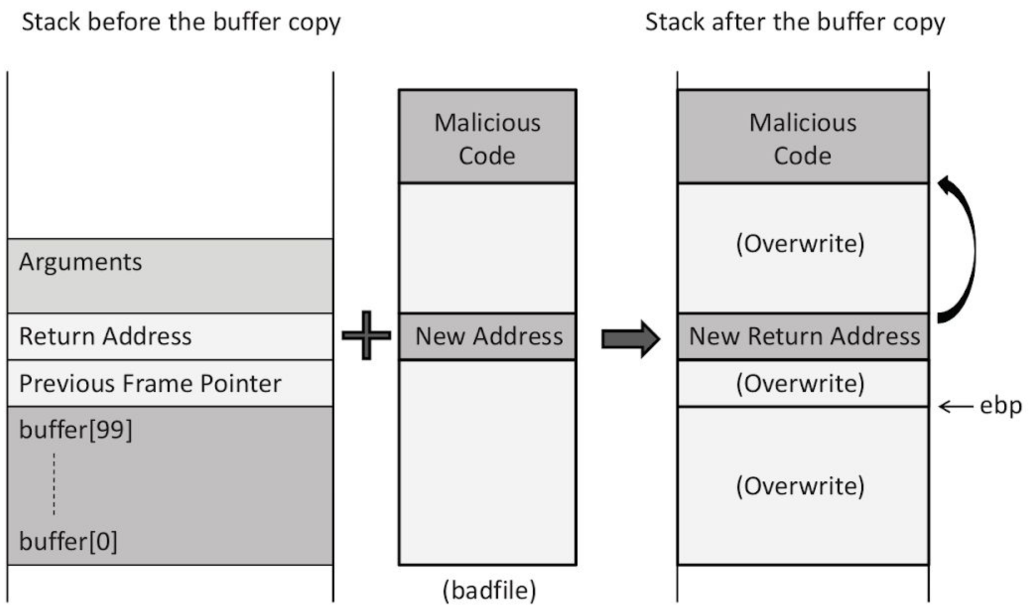
```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
        char buffer[100];
        strcpy(buffer, str);    // The following statement has a buffer overflow problem

        return 1;
}

int main(int argc, char **argv)
{
        char str[400];
        FILE *badfile;
        badfile = fopen("badfile", "r");

        fread(str, sizeof(char), 300, badfile);
        foo(str);
        printf("Returned Properly\n");
        return 1;
}
```

The above program reads 300 bytes of data from a file called "badfile", and then copies the data to a buffer of size 100. Clearly, there is a buffer overflow problem. This time, the contents copied to the buffer come from a user-provided file, i.e., users can control what is copied to the buffer. The question is what to store in "badfile", so after overflowing the buffer, we can get the program to run our code.

We need to get our code (i.e., malicious code) into the memory of the running program first. This is not difficult. We can simply place our code in "badfile", so when the program reads from the file, the code is loaded into the str[] array; when the program copies str to the target buffer, the code will then be stored on the stack. In Figure 4.5, we place the malicious code at the end of "badfile".

Next, we need to force the program to jump to our code, which is already in the memory. To do that, using the buffer overflow problem in the code, we can overwrite the return address field. If we know the address of our malicious code, we can simply use this address to overwrite the return address field. Therefore, when the function foo returns, it will jump to the new address, where our code is stored.

The figure below illustrates how to get the program to jump to our code:



In theory, that is how a buffer overflow attack works.

5. Defense Mechanisms:

The buffer overflow problem has quite a long history, and many countermeasures have been proposed, some of which have been adopted in real-world systems and software. These countermeasures can be deployed in various places, from hardware architecture, operating system, compiler, library, to the application itself.

5.1 Countermeasures: Overview

- **Hardware Architecture:**

The buffer overflow attack described in this chapter depends on the execution of the shellcode, which is placed on the stack. Modern CPUs support a feature called NX bit. The *NX bit,* standing for *No-eXecute,* is used in CPUs to separate code from data. Operating systems can mark certain areas of memory as non-executable, and the processor will refuse to execute any code residing in these areas.

Using this CPU feature, the attack described earlier in this chapter will not work anymore, if the stack is marked as non-executable. However, this countermeasure can be defeated using a different technique called *return-to-libc* attack.

- **Operating System:**

Before a program is executed, it needs to be loaded, and the running environment needs to be set up. This is done by the loader program in the operating system. The setup stage provides an opportunity to counter the buffer overflow problem because it can dictate how the memory of a program is laid out. A common countermeasure implemented at the OS loader program is referred to as *Address Space Layout Randomization or ASLR*.

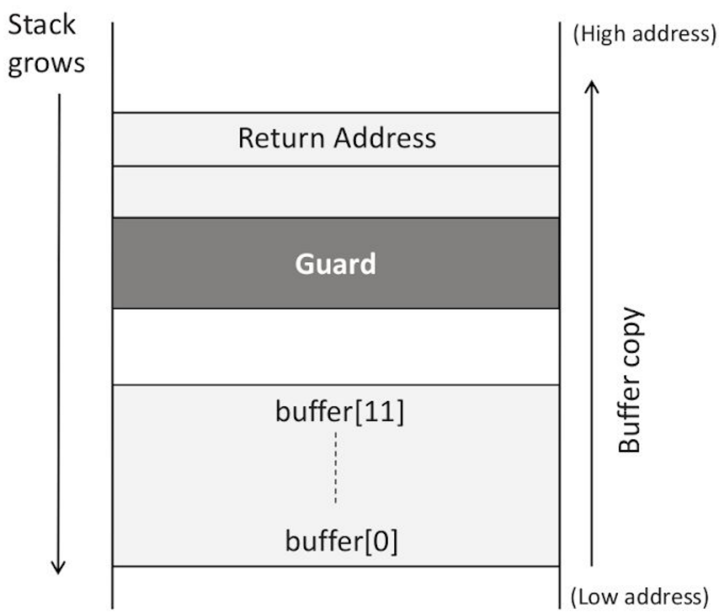
- **Compiler:**

Compilers are responsible for translating source code into binary code. They control what sequence of instructions are finally put in the binary. Ths they can control the layout of the stack. It also allows compilers to insert instructions into the binary that can verify the stack integrity, as well as eliminating the conditions that are necessary for buffer overflow attacks. Two well-known compiler-based countermeasures are Stackshield [Angelfire.com, 2000] and StackGuard [Cowa et al., 1998], which check whether the return address has been modified or not before a function returns.

5.2 StackGuard

Basic Idea:

The key observation of StackGuard is that for a buffer overflow attack to modify the return address, all the stack memory between the buffer and the return address will be overwritten. This is because the memory-copy functions, such as strcpy() and memcpy(), copy data into contiguous memory locations, so it is impossible to selectively affect some of the locations, while leaving the other intact. If we do not want to affect the value in a particular location during the memory copy, such as the shaded position marked as Guard in the above figure, the only way to achieve that is to overwrite the location with the same value that is stored there.



Based on this observation, we can place some non-predictable value (called guard) between the buffer and the return address. Before returning from the function, we check whether the value is modified or not. If it is modified, chances are that the return address may have also been modified. Therefore, the problem of detecting whether the return address is overwritten is reduced to detecting whether the guard is overwritten. These two problems seem to be the same, but they are not. By looking at the value of the return address, we do not know whether its value is modified or not, but since the value of the guard is placed by us, it is easy to know whether the guard's value is modified or not.

5.3 Address Randomization

A common countermeasure implemented at the OS loader program is referred to as Address Space Layout Randomization (ASLR). It tries to reduce the chance of buffer overflows by targeting the challenges that attackers have to overcome. In particular, it targets the fact that attackers must be able to guess the address of the injected shellcode. ASLR randomizes the layout of the program memory, making it is difficult for attackers to guess the correct address.

Basic Idea:

To succeed in buffer overflow attacks, attackers need to get the vulnerable program to “return” (i.e., jump) to their injected code; they first need to guess where the injected code will be. The success rate of the guess depends on the attackers’ ability to predict where the stack is located in the memory. Most operating systems in the past placed the stack in a fixed location, making correct guesses quite easy.

Is it really necessary for stacks to start from a fixed memory location? The answer is *No.*

When a compiler generates binary code from source code, for all the data stored on the stack, their addresses are not hard-coded in the binary code; instead, their addresses are calculated based on the frame pointer *%ebp* and stack pointer *%esp.* Namely, the addresses of the data on the stack are represented as the offset to one of these two registers, instead of to the starting address of the stack. Therefore, even if we start the stack from another location, as long as the *%ebp* and *%esp* are set up correctly, programs can always access their data on the stack without any problem.

For attackers, they need to guess the absolute address, instead of the offset, so knowing the exact location of the stack is important. If we randomize the start location of a stack, we make attackers’ job more difficult, while causing no problem to the program. That is the basic idea of the Address Layout Randomization (ASLR) method, which has been implemented by operating systems to defeat buffer overflow attacks. This idea does not only apply to stack, it can also be used to randomize the location of other types of memory, such as heap, libraries, etc.

OFF BY ONE BUFFER OVERFLOW ATTACKS

Sometimes developers don't implement length conditions correctly and as a result the off by one vulnerability exists. The off by one vulnerability in general means that if an attacker supplied input with certain length if the program has an incorrect length condition the program will write one byte outside the bounds of the space allocated to hold this input causing one of two scenarios depending on the input;

- Malicious input will overwrite an adjacent variable next to the input buffer on the stack.
- The input will overwrite the saved frame pointer of the previous function thus when returning the attacker can alter the application flow and return address.

We are more interested in the second scenario.

Example:

```
#include <stdio.h>
int cpy(char *x)
{
        char buff[1024];
        strcpy(buff,x);
        printf("%s\r\n",buff);
}
int main(int argc, char *argv[])
{
        if(strlen(argv[1]) > 1024)
        {
                printf("Buffer Overflow Attempt!!!\r\n");
                return 1;
        }
        cpy(argv[1]);
}
```

Basically what the code does is check if the Input size is bigger than 1024 the size of our buffer.

```
                if(strlen(argv[1])>1024)
```

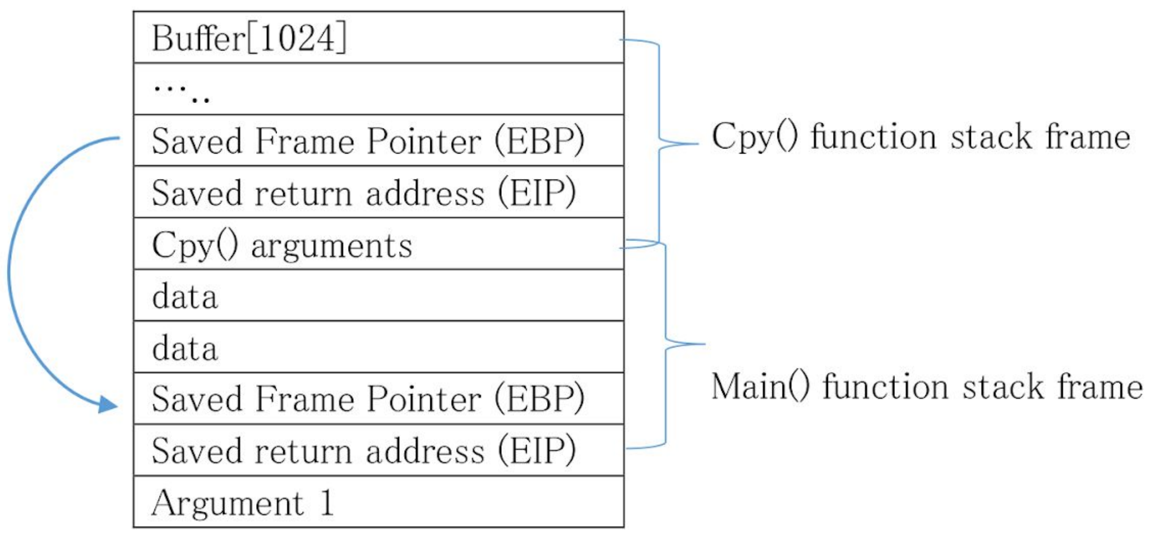
If it was larger it exits the program with an error. The error in this code is that the "strlen" function gets the length of the string without the terminating NULL byte.

The “strcpy()” function in the “cpy()” function will trigger a segmentation fault.

If the input was exactly 1024 byte; The Length check will succeed without errors. Then because *strcpy()* copies the string including the terminating NULL byte. The 1024 input string is 1025 byte long including the NULL. When the *strcpy()* function is called it will try to copy a 1025 byte string into a 1024 byte buffer triggering a segmentation fault.

Explanation:

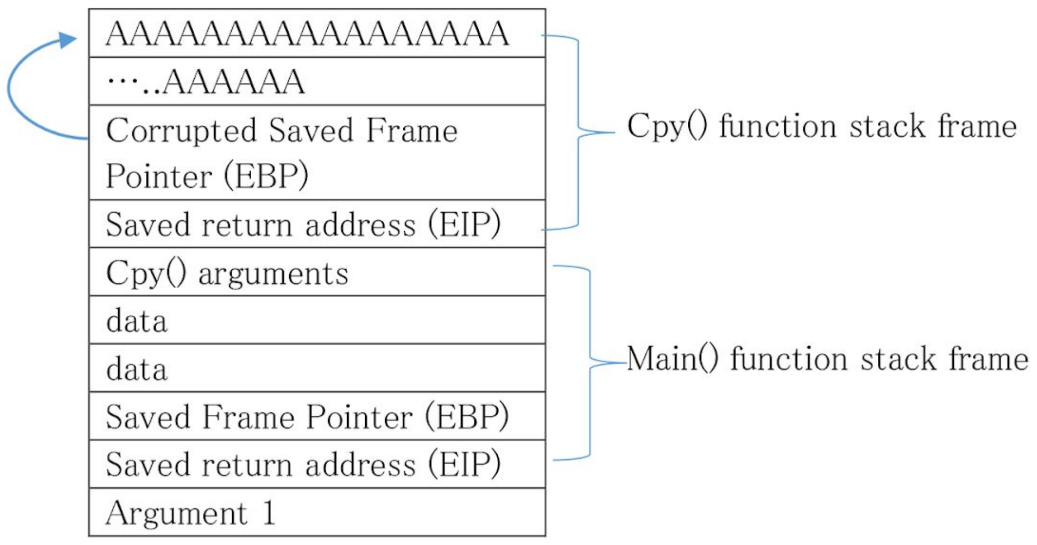
Ordinary Stack during normal function call:


Buffer[1024]
…..
Saved Frame Pointer (EBP)
Saved return address (EIP)
Cpy() arguments
data
data
Saved Frame Pointer (EBP)
Saved return address (EIP)
Argument 1
Cpy() function stack frame
Main() function stack frame

Hijack:

Let’s see what happens in the previous example if we supplied an input of 1024 byte As we know the stack grows upwards towards lower memory addresses. So if 1025 byte is copied to a 1024 byte buffer the NULL byte will be written outside the bounds of the buffer overwriting the least significant byte of the saved frame pointer (EBP). So after execution the program will not pop off the correct frame pointer to the parent function instead it will pop our modified frame pointer which will get us directly in our buffer. Then we can specify local variable values from the previous stack frame as well as the saved base pointer and return address. Therefore, when the calling function returns, an arbitrary return address will be specified, and total control over the program execution flow will be seized.

Frame Pointer off by one corruption:


AAAAAAAAAAAAAAAAA
⋯..AAAAAA
Corrupted Saved Frame Pointer (EBP)
Saved return address (EIP)
Cpy() arguments
data
data
Saved Frame Pointer (EBP)
Saved return address (EIP)
Argument 1
Cpy() function stack frame
Main() function stack frame

Defense Mechanism:
i) Also, the designed program should validate input size and raise error immediately for the invalid ones.
ii) If the canary is disturbed, exception code is executed and the program terminates. This canary property can be used while enforcing the defense.
iii) Since this kind of overflow attack is a specific case of the generalized stack overflow attacks, the measures invoked to tackle stack overflow attacks can also be effectively utilized for tackling off-by-one buffer overflow attacks.

RETURN TO LIBC

If you face the error:

```
#include <bits/libc-header-start.h>
              ^~~~~~~~~~~~~~~~~~~~~~~~~~
    compilation terminated.
Do:
sudo apt-get install gcc-multilib g++-multilib
```

Return-to-Libc involves a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process' executable memory, bypassing the no-execute bit feature (if present). The attacker need not inject their own code.

A non-executable stack can prevent some buffer overflow exploitation, however it cannot prevent a return-to-libc attack because in the return-to-libc attack only existing executable code is used. On the other hand, these attacks can only call pre existing functions.

Address space layout randomization (ASLR) makes this type of attack extremely unlikely to succeed on 64-bit machines as the memory locations of functions are random. For 32-bit systems, however, ASLR provides little benefit since there are only 16 bits available for randomization, and they can be defeated by brute force in a matter of minutes.

For Return to LibC we need to bypass ASLR which is done using the following command:

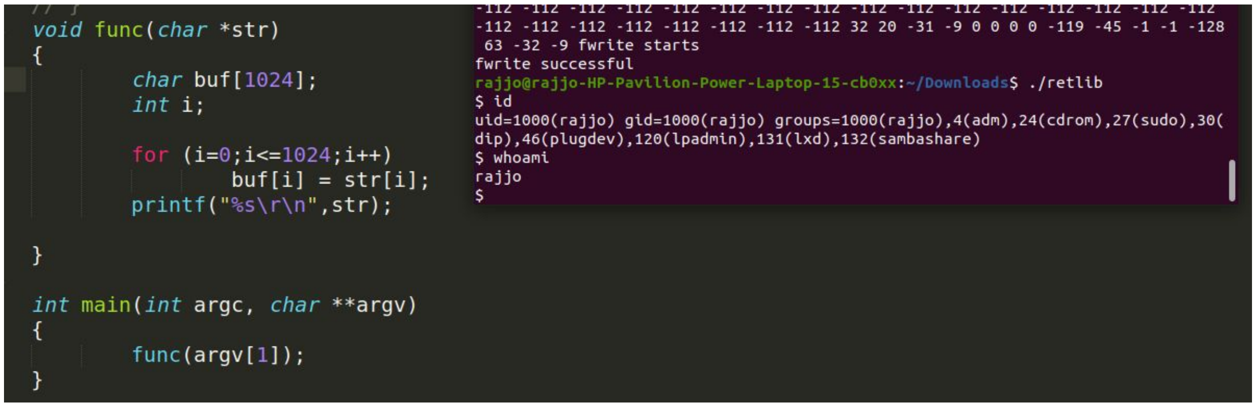
```
sudo sysctl -w kernel.randomize_va_space=0
```

The following is return-to-libc stack:

```
<- stack grows this way
   addresses grow this way ->
------------------------------------------------------------------------------
| buffer fill-up | f1 | pop-ret | f1_arg | f2 | dmm | f2_arg1 | f2_arg2 ...
------------------------------------------------------------------------------
                                    ^
                                    |
                                    - this int32 should overwrite return
address
                                        of a vulnerable function
```

So we should set `system`'s entry address at `bof`'s return address (`&buf[24]`), set `system`'s argument address at `&buf[32]`, and set `exit`'s entry address at `&buf[36]`.

```
Exploit:
A new shell is spawned by the inserted code
```

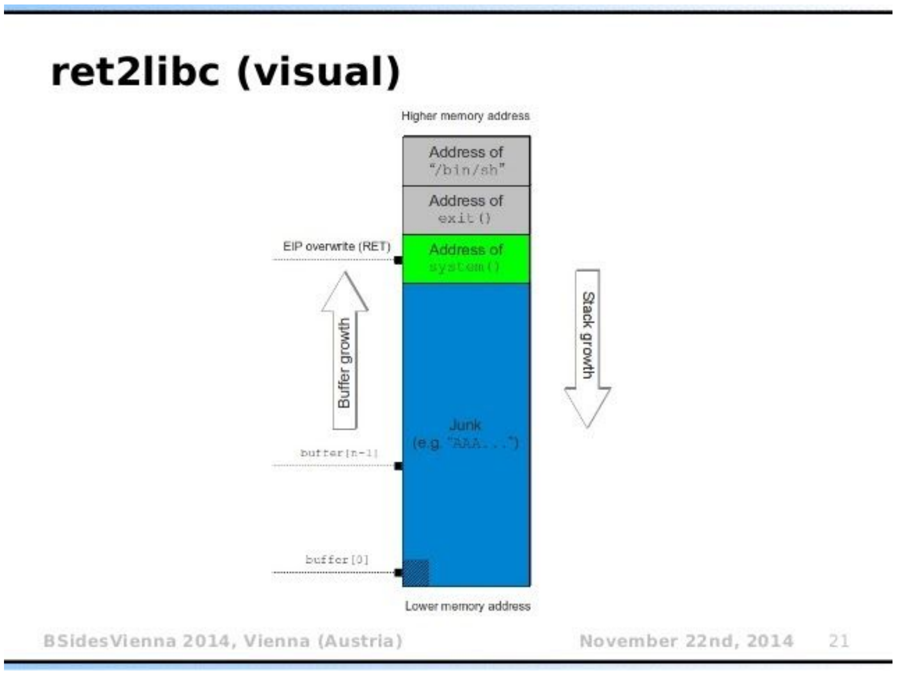


```
void func(char *str)
{
        char buf[1024];
        int i;

        for (i=0;i<=1024;i++)
                buf[i] = str[i];
        printf("%s\r\n",str);

}

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

```
-112 -112 -112 -112 -112 -112 -112 -112 -112 -112 -112 -112 -112 -112 -112 -112
-112 -112 -112 -112 -112 -112 -112 -112 32 20 -31 -9 0 0 0 0 -119 -45 -1 -1 -128
 63 -32 -9 fwrite starts
fwrite successful
rajjo@rajjo-HP-Pavilion-Power-Laptop-15-cb0xx:~/Downloads$ ./retlib
$ id
uid=1000(rajjo) gid=1000(rajjo) groups=1000(rajjo),4(adm),24(cdrom),27(sudo),30(
dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare)
$ whoami
rajjo
$
```

Defense:

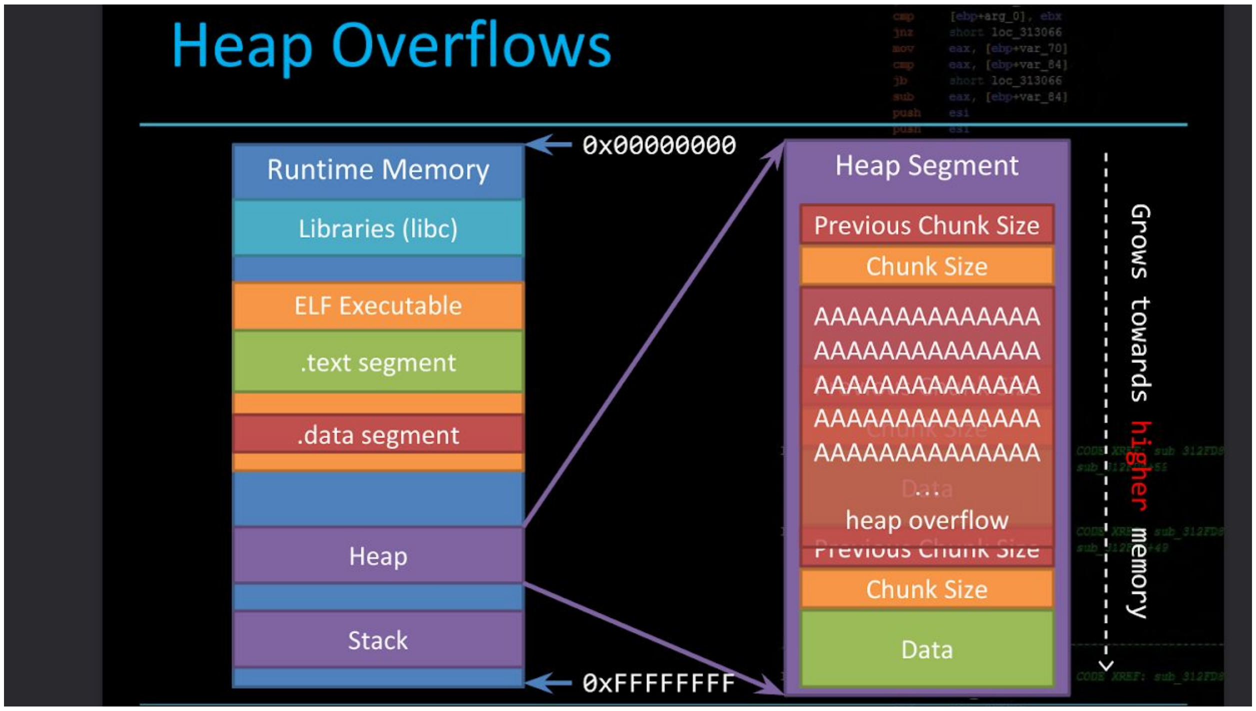
- ASLR makes this attack unlikely on 64-bit systems.
- 32-bit systems have 1 bits for randomization. So a brute force return from LibC may be used.



HEAP BASED BUFFER OVERFLOW ATTACKS

1. Introduction

A **heap overflow** or **heap overrun** is a type of buffer overflow that occurs in the heap data area. Heap overflows are *exploitable in a different manner* than stack overflows. Heap memory is dynamically allocated and typically contains program data. Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures like linked list pointers. The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as malloc metadata) and uses the resulting pointer exchange to overwrite a program function pointer.



2. Consequence

An accidental overflow may result in data corruption or unexpected behavior by any process that accesses the affected memory area. On operating systems without memory protection, this could be any process on the system.

Examples: A Microsoft JPEG GDI+ buffer overflow vulnerability allows remote execution of code on the affected machine, iOS jailbreaking uses heap overflows to gain arbitrary code execution, usually for kernel exploits to achieve the ability to replace the kernel with the one the jailbreak provides.

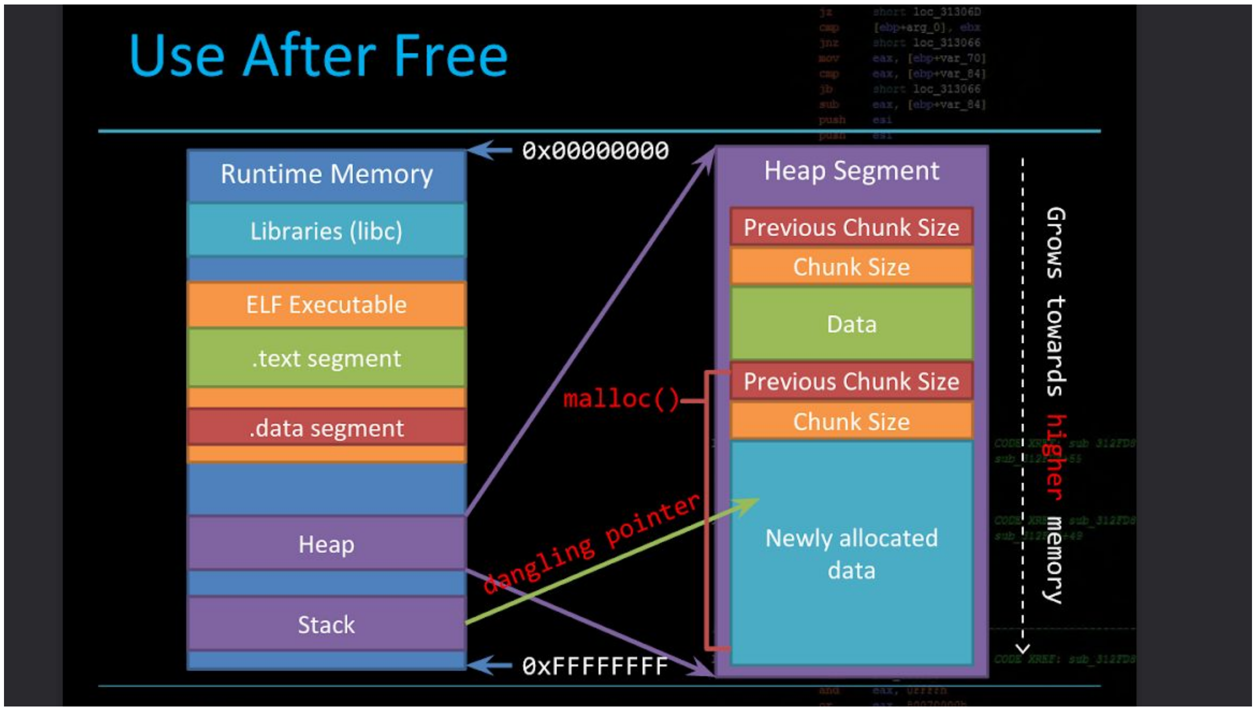
3. Exploitation: Use After Free (UAF)

We will discuss a **Use After Free (UAF)** vulnerability, which is very popular nowadays. Almost every modern browser exploit leverages a UAF.

A class of vulnerability where data on the heap is freed, but a leftover reference or ‘dangling pointer’ is used by the code as if the data were still valid

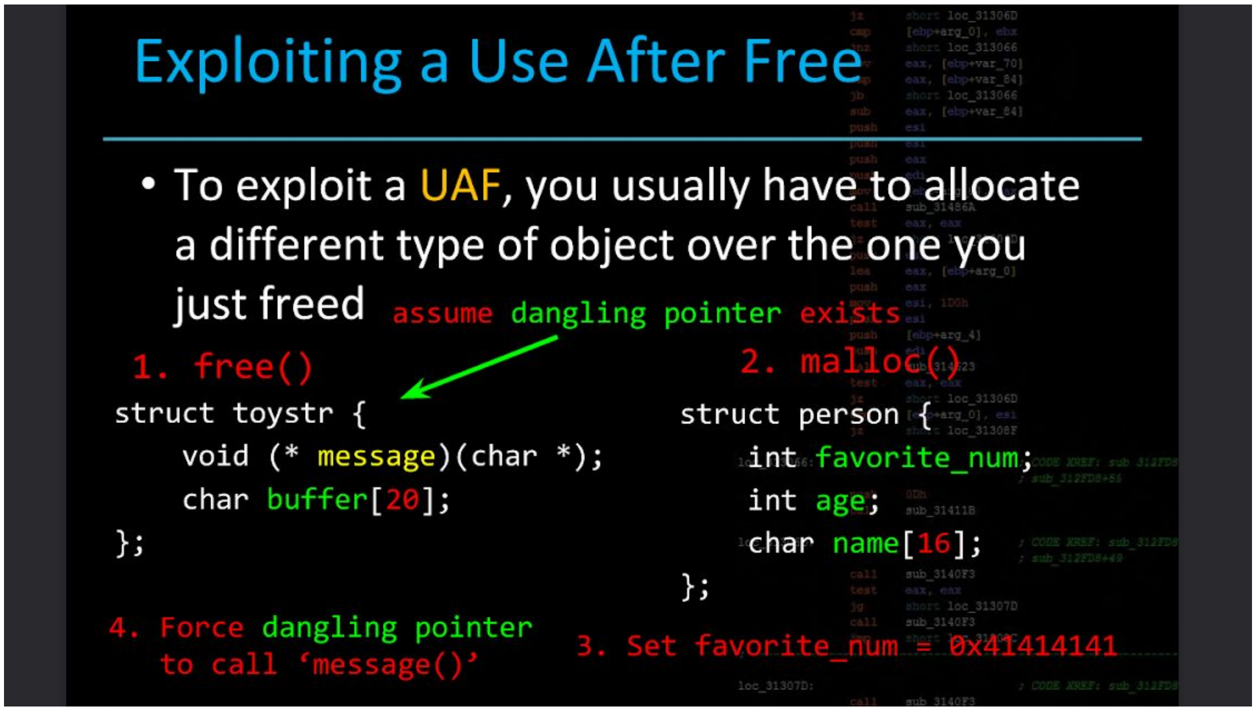
3.1 Details:

Dangling Pointer: A left over pointer in your code that references free’d data and is prone to be reused. As the memory it’s pointing at was freed, there’s no guarantees on what data is there now.



To exploit a UAF, you usually have to allocate a different type of object over the one you just freed.

You actually don’t need any form of memory corruption to leverage a use after free•It’s simply an implementation issue–pointer mismanagement.



Exploiting a Use After Free

- To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

assume dangling pointer exists

1. free()

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

4. Force dangling pointer to call 'message()'

3.2 Reasons for Wide Use:

- Doesn't require any memory corruption to use
- Can be used for info leaks
- Can be used to trigger memory corruption or get control of EIP

3.3 Detection:

From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is very difficult, even in industry.

Why ?

- UAF's only exist in certain states of execution, so statically scanning source for them won't go far
- They're usually only found through crashes, but symbolic execution and constraint solvers are helping find these bugs faster.

4. General Detection and Defense

As with buffer overflows there are primarily three ways to protect against heap overflows. Several modern operating systems such as Windows and Linux provide some implementation of all three.

- Prevent execution of the payload by separating the code and data, typically with hardware features such as NX-bit
- Introduce randomization so the heap is not found at a fixed offset, typically with kernel features such as ASLR (Address Space Layout Randomization)
- Introduce sanity checks into the heap manager

Since version 2.3.6 the GNU libc includes protections that can detect heap overflows after the fact, for example by checking pointer consistency when calling unlink. However, those protections against prior exploits were almost immediately shown to also be exploitable. In addition, Linux has included support for ASLR since 2005, although PaX introduced a better implementation years before. Also Linux has included support for NX-bit since 2004.

Microsoft has included protections against heap resident buffer overflows since April 2003 in Windows Server 2003 and August 2004 in Windows XP with Service Pack 2. These mitigations were safe unlinking and heap entry header cookies. Later versions of Windows such as Vista, Server 2008 and Windows 7 include: Removal of commonly targeted data structures, heap entry metadata randomization, expanded role of heap header cookie, randomized heap base address, function pointer encoding, termination of heap corruption and algorithm variation. Normal Data Execution Prevention (DEP) and ASLR also help to mitigate this attack.

References

https://en.wikipedia.org/wiki/Code_Red_(computer_worm)
https://www.researchgate.net/publication/220269622_Code-Red_a_case_study_on_the_spread_and_victims_of_an_Internet_worm
https://www.sans.org/security-resources/malwarefaq/code-red
https://en.wikipedia.org/wiki/Heap_overflow
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf
https://bufferoverflows.net/use-after-free-vulnerability-uaf/
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf
https://blog.exodusintel.com/2019/03/20/cve-2019-5786-analysis-and-exploitation/
https://resources.infosecinstitute.com/topic/heap-overflow-vulnerability-and-heap-internals-explained/
https://blog.rapid7.com/2019/06/12/heap-overflow-exploitation-on-windows-10-explained/
https://github.com/LauraWartschinski/overflow_with_joy
https://stackoverflow.com/questions/47607333/illegal-instruction-when-trying-to-get-shell-from-a-simple-stackoverflow
https://stackoverflow.com/questions/2500362/running-32-bit-assembly-code-on-a-64-bit-linux-64-bit-processor-explain-the
https://github.com/npapernot/buffer-overflow-attack
https://seedsecuritylabs.org/lab_env.html
https://resources.infosecinstitute.com/topic/heap-overflow-vulnerability-and-heap-internals-explained/
https://blog.rapid7.com/2019/06/12/heap-overflow-exploitation-on-windows-10-explained/
https://blog.exodusintel.com/2019/03/20/cve-2019-5786-analysis-and-exploitation/
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf
https://bufferoverflows.net/use-after-free-vulnerability-uaf/
https://sploitfun.wordpress.com/2015/06/07/off-by-one-vulnerability-stack-based-2/
https://nixhacker.com/exploiting-off-by-one-buffer-overflow/
https://www.exploit-db.com/docs/english/28478-linux-off-by-one-vulnerabilities.pdf
https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/off_by_one/
https://csl.com.co/en/off-by-one-explained/
http://www.cis.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf
https://www.thegeekstuff.com/2013/06/buffer-overflow/
https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/
https://github.com/Subangkar/Buffer-Overflow-Attack-Seedlab
https://dhavalkapil.com/blogs/Shellcode-Injection/

https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/#:~:text=Buffer%20overflow%20vulnerability,of%20it's%20alloted%20memory%20space.
https://nixhacker.com/exploiting-off-by-one-buffer-overflow/
https://csl.com.co/en/off-by-one-explained/
https://vulncat.fortify.com/en/detail?id=desc.internal.cpp.buffer_overflow_off_by_one
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf
https://www.geeksforgeeks.org/heap-overflow-stack-overflow/
http://www.cis.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf
https://github.com/yehiahesham/Buffer-Overflow-Attack
https://github.com/npapernot/buffer-overflow-attack
https://github.com/RihaMaheshwari/Buffer-Overflow-Exploit
https://github.com/shashijangra22/Buffer-Overflow-Attack
https://github.com/hackutk/overflow-example