## What is a buffer?

## <https://www.cloudflare.com/learning/security/threats/buffer-overflow/>

A buffer, or data buffer, is an area of physical memory storage used to temporarily store data while it is being moved from one place to another. These buffers typically live in RAM memory. Computers frequently use buffers to help improve performance; most modern hard drives take advantage of buffering to efficiently access data, and many online services also use buffers.

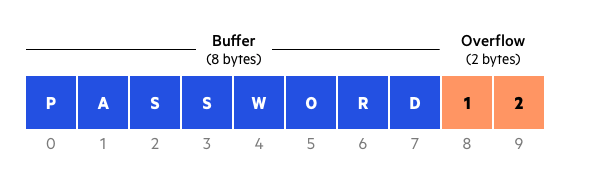
Buffers are designed to contain specific amounts of data. Unless the program utilizing the buffer has built-in instructions to discard data when too much is sent to the buffer, the program will overwrite data in memory adjacent to the buffer.

What is Buffer Overflow?

<https://www.imperva.com/learn/application-security/buffer-overflow/>

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.



## What is a Buffer Overflow Attack?

## <https://www.cloudflare.com/learning/security/threats/buffer-overflow/>

Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. For example, an attacker may introduce extra code, sending new instructions to the application to gain access to IT systems.

If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code. For example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program.

## Who is vulnerable to buffer overflow attacks?

Certain coding languages are more susceptible to buffer overflow than others. C and C++ are two popular languages with high vulnerability, since they contain no built-in protections against accessing or overwriting data in their memory. Windows, Mac OSX, and Linux all contain code written in one or both of these languages.

<https://www.ukessays.com/essays/computer-science/buffer-overflow-attacks-and-types-computer-science-essay.php>

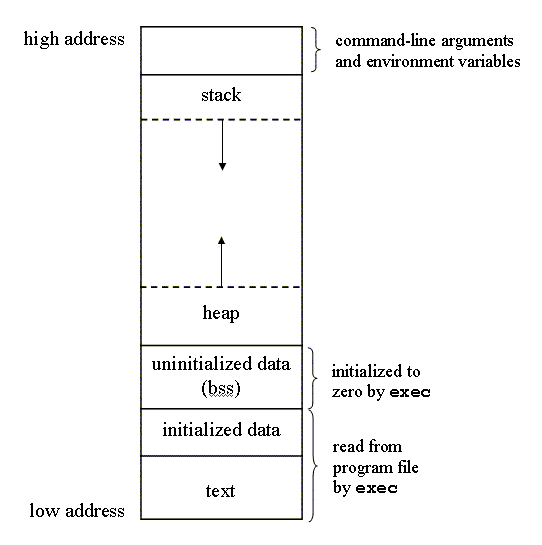
There are basically two kinds of buffer overflow attacks:

1. Heap-based attacks and

2. Stack-based attacks.

In Heap-based attack the attacker floods the memory space which is actually reserved for the program. This attacks is not exactly easy as it feels, hence the number of attacks with respect to the heap are very rare. In Stack-based attack, the attacker takes advantage of the stack, a part of the memory reserved for the program to store data or addresses. The attacker then partially crashes the stack and forces the program execution to start from a return address of a malicious program address which is actually written by the attacker.

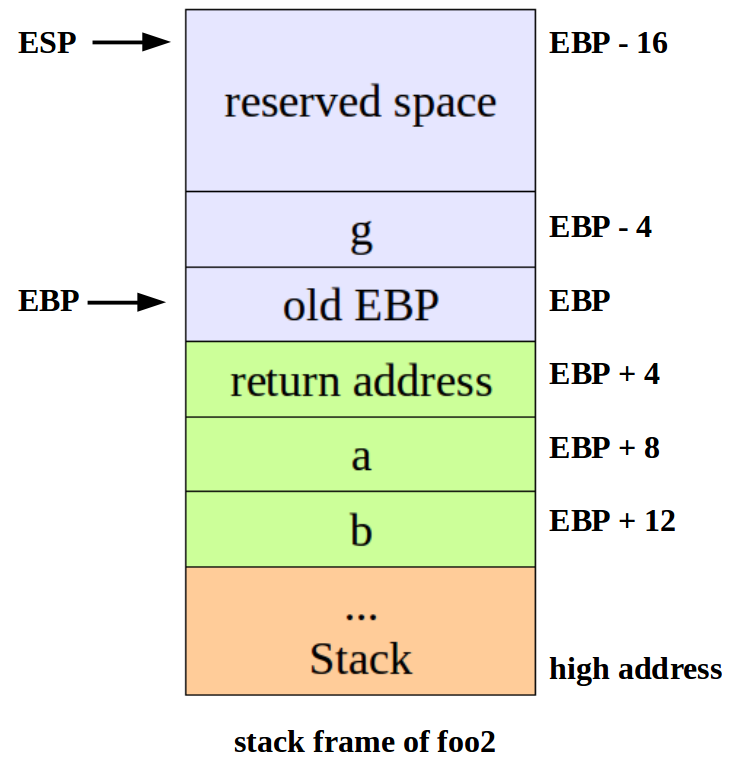
**BASIC RAM LAYOUT**



**STACK OVERFLOW**

What is a stack?

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).



What is a stackframe?

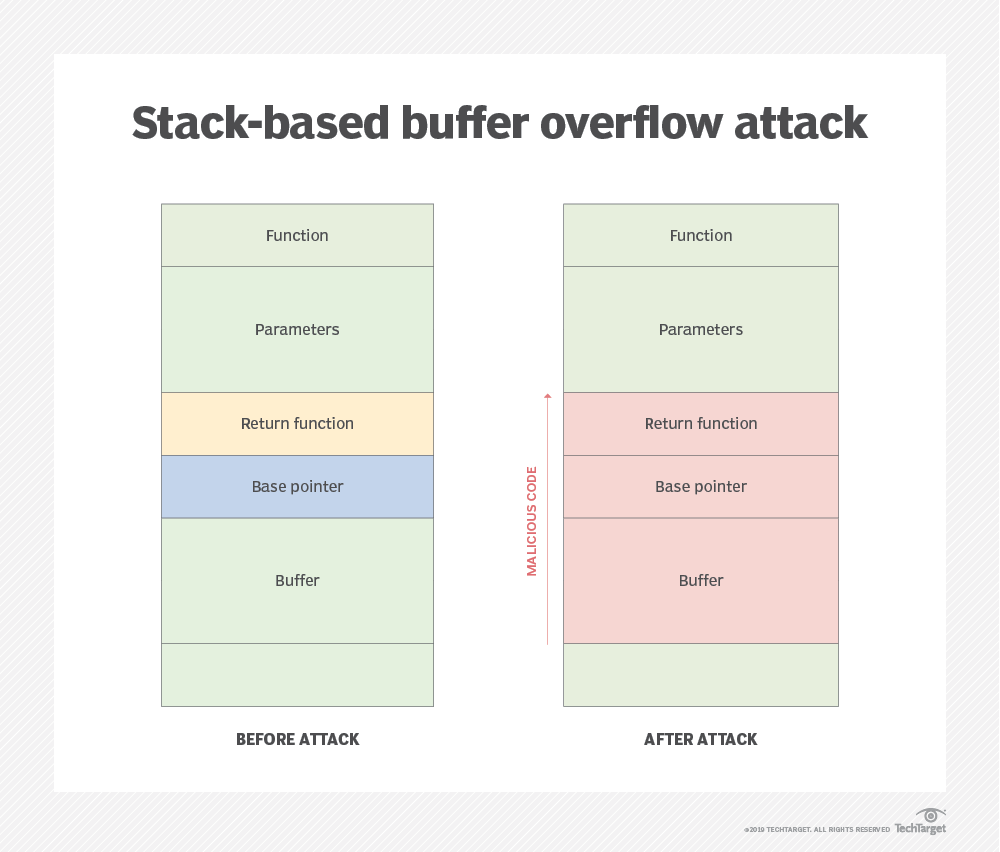
The *stack frame*, also known as *activation record* is the collection of all data on the stack associated with one subprogram call.

The stack frame generally includes the following components:

* The return address
* Argument variables passed on the stack
* Local variables (in HLLs)

What is smash the stack?

Buffer Overflow refers to a situation when we are able write past the size of a variable , which results in change of data near them , When this type of overflow occur in the stack it is called a stack overflow . With this we can change the value of sensitive variables which are adjacent to the overflow , Also since the return address of a function is stored on the stack we can change the control flow of the program .



<https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/>

Example:

#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>

int main(int argc, char \*\*argv)

{

volatile int modified;

char buffer[64];

modified = 0;

gets(buffer);

if(modified != 0) {

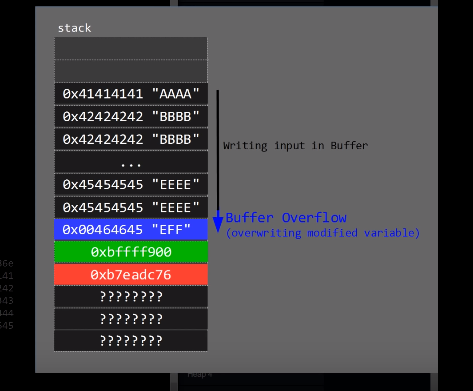
printf("you have changed the 'modified' variable\n");

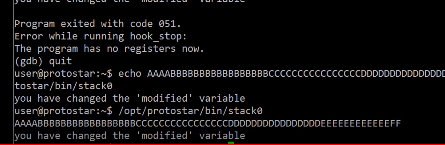
} else {

printf("Try again?\n");

}

}





Vulnerability:

Now, let’s talk about the mistakes that the programmer (me) made. First, developers should never, ever, ever use the gets function because it does not check to make sure that the size of the data it reads in matches the size of the memory location it uses to save the data. It just blindly reads the text and dumps it into memory. There are many functions that do the exact same thing—these are known as unbounded functions because developers cannot predict when they will stop reading from or writing to memory.

What happened there:we have provided an input bigger than the size of the memory alloated and the value of modified gets overwirtten.as we can see value of the local variable stored in eax is changed to EEF from 0.

**x command(GDB):**

Displays the memory contents at a given address using the specified format.

Syntax

**x** [*Address expression*]  
**x** /[*Format*] [*Address expression*]  
**x** /[*Length*][*Format*] [*Address expression*]  
**x**

Parameters

*Address expression*

Specifies the memory address which contents will be displayed. This can be the address itself or any C/C++ expression evaluating to address. The expression can include registers (e.g. $eip) and pseudoregisters (e.g. $pc). If the address expression is not specified, the command will continue displaying memory contents from the address where the previous instance of this command has finished.

*Format*

If specified, allows overriding the output format used by the command. Valid format specifiers are:

* o - octal
* x - hexadecimal
* d - decimal
* u - unsigned decimal
* t - binary
* f - floating point
* a - address
* c - char
* s - string
* i - instruction

The following size modifiers are supported:

* b - byte
* h - halfword (16-bit value)
* w - word (32-bit value)
* g - giant word (64-bit value)

*Length*

Specifies the number of elements that will be displayed by this command.

EXERCISE: <https://samsclass.info/127/proj/ED202c.htm>

CODE:

#include <stdlib.h>

#include <stdio.h>

int test\_pw() {

char password[10];

printf("Password address: %p\n", password);

printf("Enter password: ");

fgets(password, 50, stdin);

return 1;

}

void win() {

printf("You win!\n");

}

void main() {

if (test\_pw()) printf("Fail!\n");

else win();

}

VULNERABILITY:

fgets() function takes input without checking the size of array. This allows us to overflow the buffer and exploit.

COMPILE AND RUN IN TERMINAL:

To compile as 32 bit on a 64 bit machine(for ease of understanding), type the following in terminal-

>>sudo apt update

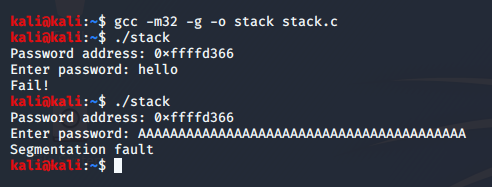
>>sudo apt install build-essential gcc-multilib gdb –y

To compile

>> gcc –m32 -g -o filename filename.c

To run

>> ./filename

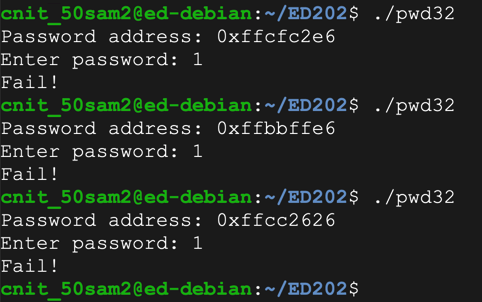


Segmentation fault means that we have gone beyond the permitted size of array.

**Disabling ASLR**

Address Space Layout Randomization is a defense feature to make buffer overflows more difficult, and all modern operating systems uses it by default.

To see it in action, run the "pwd32" program several times with a password of **1**. The password address is different every time, as shown below.



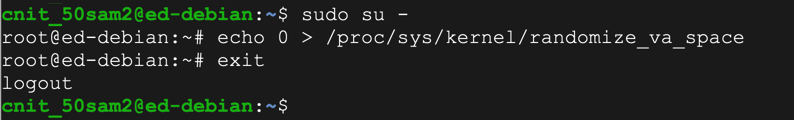
ASLR makes you much safer, but it's an irritation we don't need for the first parts of this project, so we'll turn it off.

In a Terminal, execute these commands, as shown below.

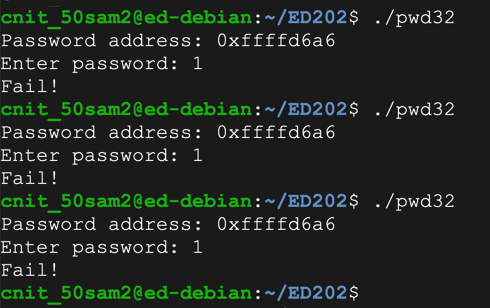
**sudo su**

**echo 0 > /proc/sys/kernel/randomize\_va\_space**

**exit**



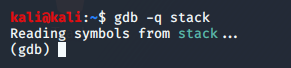
Run the "pwd32" program several times again with a password of **1**. The password address is now the same every time, as shown below.



To open program in gdb

>> gdb -q stack

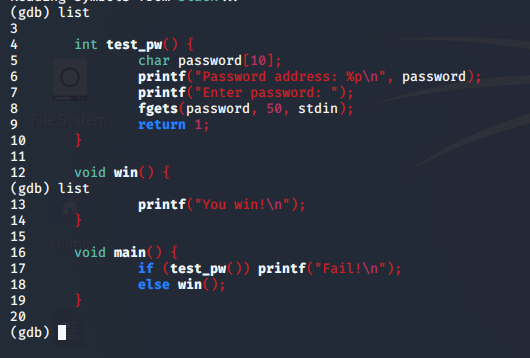
-q doesn’t print the huge text paragraph when you start gdb.



Print the source code

>>(gdb) list

it only prints ten lines, so you might have to execute list multiple times

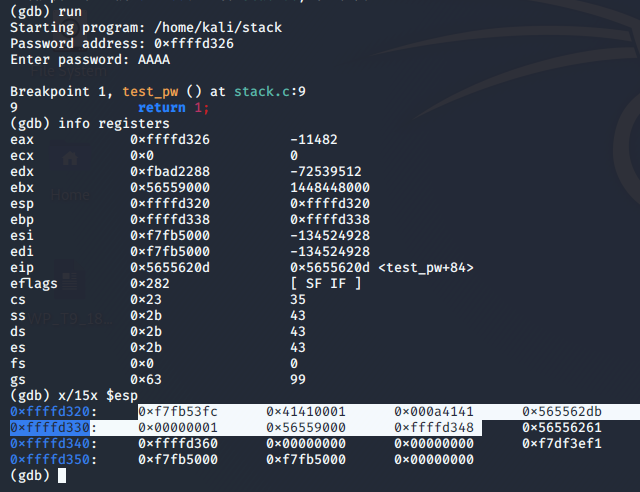


ENTER BREAKPOINTS: break/b line-number

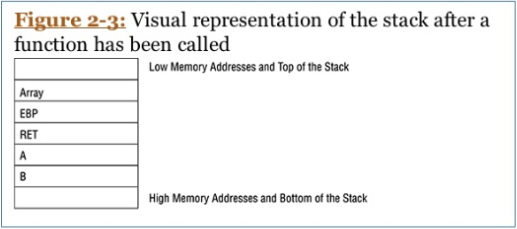
>>(gdb) b 9



GENERAL PROCESS TO EXAMINE WHAT IS HAPPENING:



1. Stops execution before line 9
2. Info registers show that stack spans from esp (320) to ebp(338)
3. Stack structure- esp -> data -> ebp-> return address(to main here).



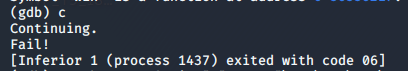
1. We can see 0x565562db is return address.
2. We can see input is filled in little endian way from 2nd slot’s second half. (input is not filled directly from esp but after a few bytes)

We need to overwrite the return address for the program execution to be redirected.

Here, we want to redirect it to the win function. So the return address should be replaced by address of win()

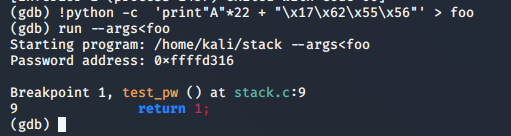


After we continue, we’ll FAIL because we have not redirected it to win().

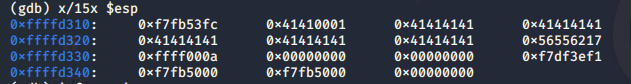


This time we run again but with our exploit.

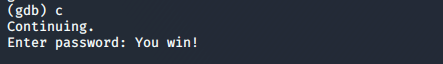
Calculating by examining the stack, we saw we need 22 As till ebp (338) and next 4 bytes will have the address of win().



What has happened to the stack because of our exploit?



Return address now has the address of win()

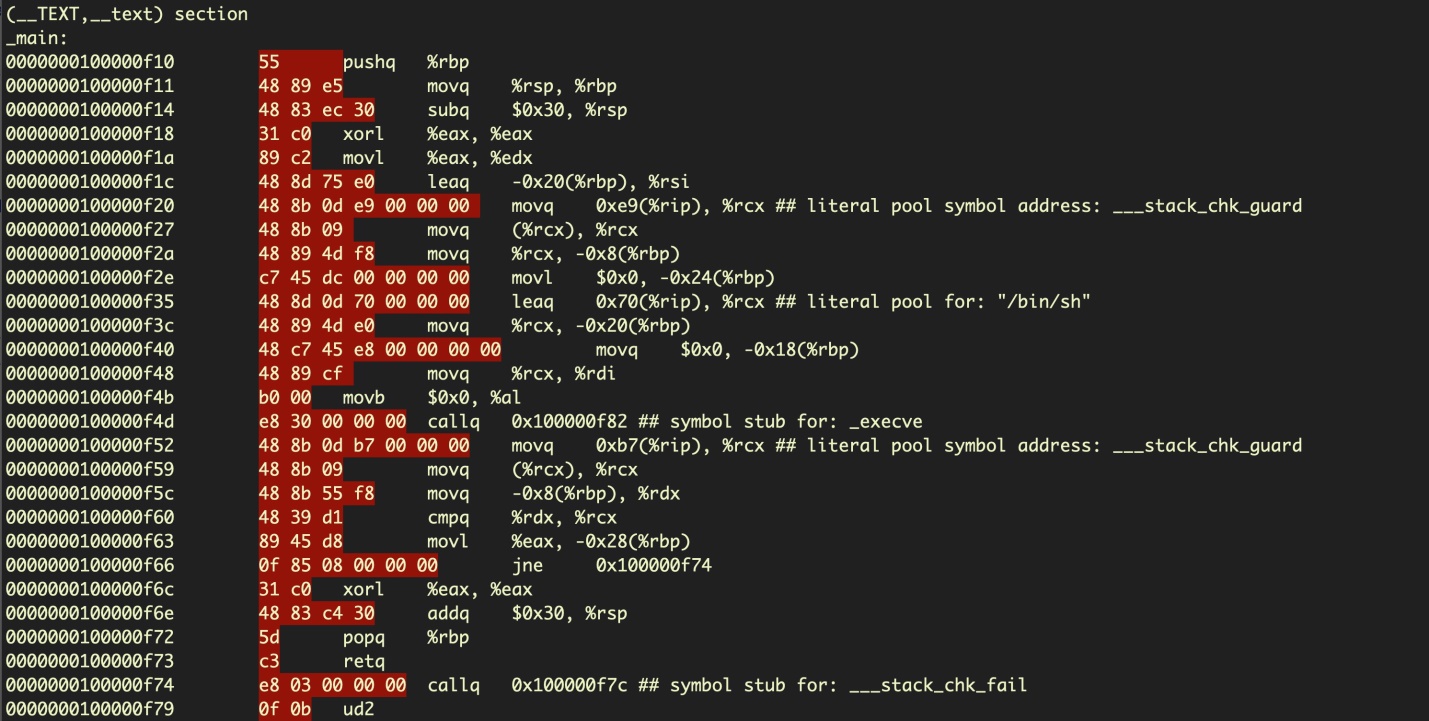


## SHELL CODE EXPLOIT

<https://www.slideshare.net/SamBowne/cnit-127-ch-3-shellcode-169145517> : FIRST 10 SLIDES

What is a shell code?

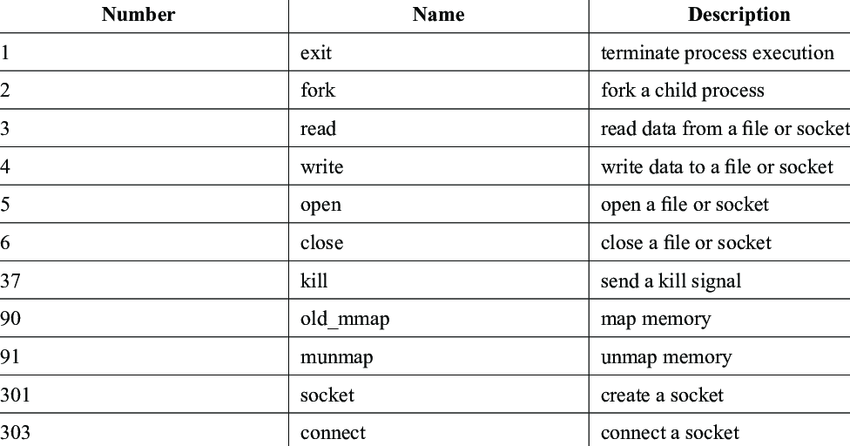
It is a set of instructions written in assembler which is then translated into hexadecimal opcodes.



Highlighted part is the converted code to hexadecimal opcodes.

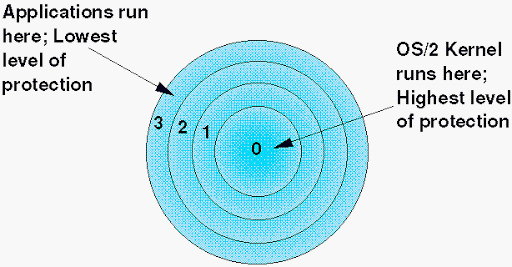
Our intention is to inject the shell code into a system by exploiting a vulnerability.

What are system calls/syscalls?

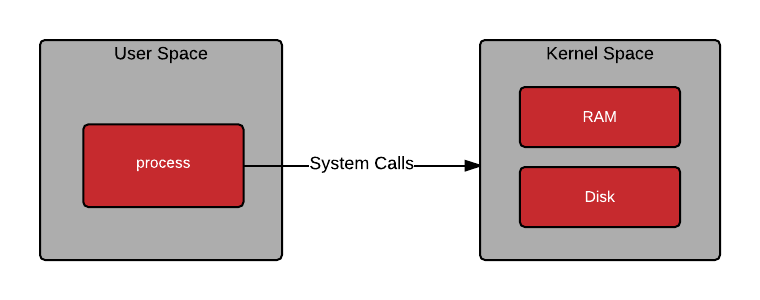


Syscalls directly access the kernel to get input/output, exit a process etc. They are the interface between protected kernel mode and user mode.

Protection rings of Linux:



Ring 3 is userland and ring 0 is kernel land.



The userland has no direct access to hardware. It has to communicate w kernel mode via syscalls.

If a user mode program attempts to access kernel memory, this generates an access exception.

Syscalls are the interface between user and kernel mode. You use syscall to call the kernel.

Kernel mode has full command. You can’t write a code or program but only call functions of OS.

Protected kernel mode prevents user applications from compromising the OS.

What are interrupts?

It is essentially an event. When an interrupt occurs, the computer stops whatever it is doing and listens to you.

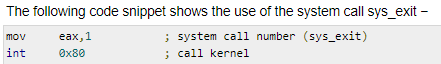
Interrupt no. 128 = 0x80 : calls the kernel.

Executed as: INT 0x80

How to execute a syscall?

1. Load the syscall number in eax register.
2. Put the arguments of the specified syscall in other registers.
3. Execute INT 0x80
4. CPU switches to kernel mode
5. Syscall loaded in eax executes.

Eg: to exit a program:



Number of exit syscall is 1. Therefore 1 is saved in eax. It does not have any parameters. INT 0x80 then calls the kernel mode.

For list of syscalls and their arguments: <http://syscalls.kernelgrok.com/>

->How to construct a shell code from scratch: not covered here but lecture CNIT 127 is good: <https://samsclass.info/127/127_S20.shtml>

What we are trying to achieve?

We exploit a vulnerability with buffer overflow. In that exploit parameter, you include the shell code. Thus you will inject a shell code which will manipulate the OS using syscalls.

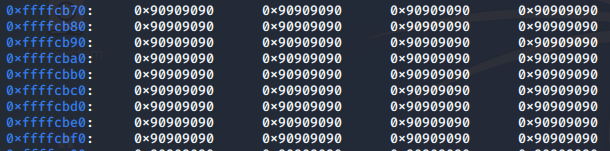
Note that you have to keep shell code as small as possible and no special characters like null, space, tab etc.

What is a NOP sled?

There are some imperfections in the debugger, so an exploit that works in gdb may fail in a real Linux shell. This happens because environment variables and other details may cause the location of the stack to change slightly. The usual solution for this problem is a NOP Sled--a long series of "90" bytes, which do nothing when processed and proceed to the next instruction.

How to add: ‘\x90’\*length

Output looks like this:



Disabling ASLR:

Remember that it is a temporary solution and needs to be disabled once in the start of your project even though you might have done it before on your machine.

**sudo su -**

**echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space**

**exit**

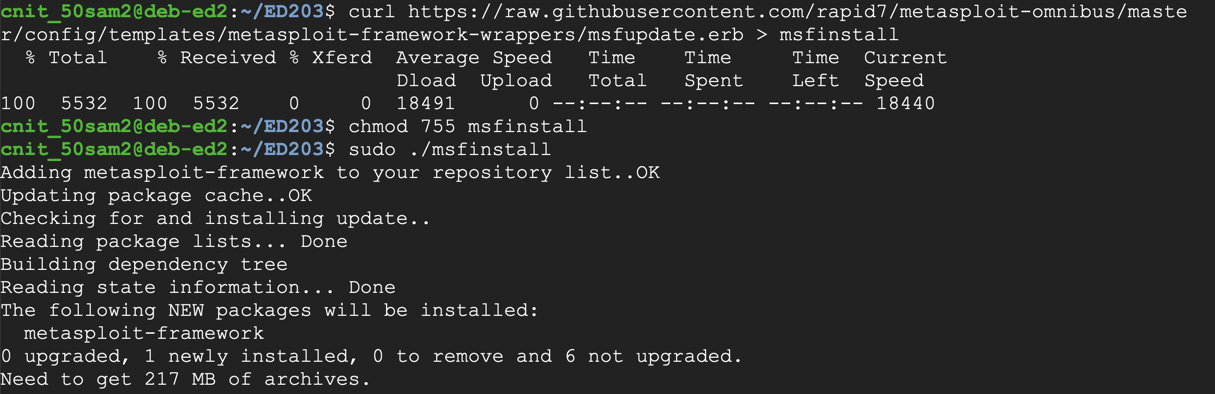
Getting your shell code:

First we need to install Metasplolit. Execute these commands:

**curl https://raw.githubusercontent.com/rapid7/metasploit-omnibus/master/config/templates/metasploit-framework-wrappers/msfupdate.erb > msfinstall**

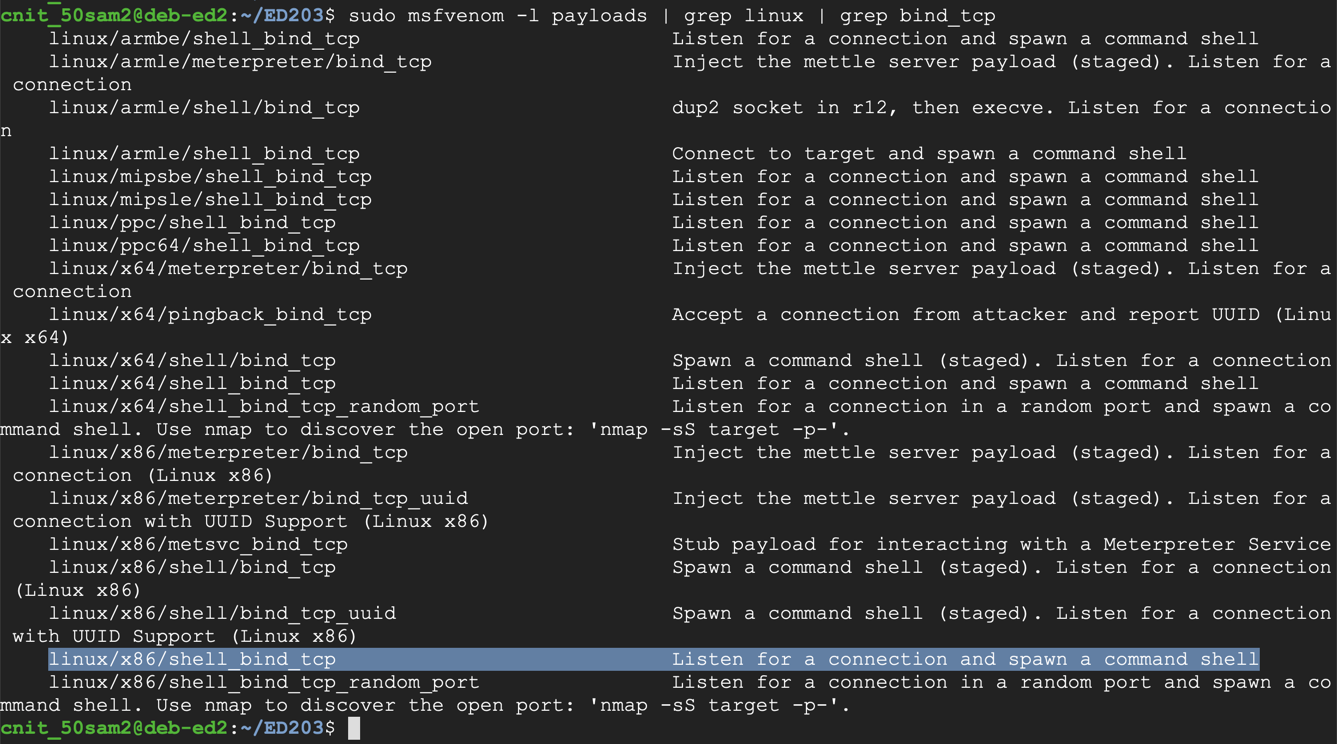
**chmod 755 msfinstall**

**sudo ./msfinstall**



Metasploit provides a tool named msfvenom to generate shellcode. Execute this command, which shows the exploits available for a Linux platform, which bind a shell to a listening TCP port:

**sudo msfvenom -l payloads | grep linux | grep bind\_tcp**

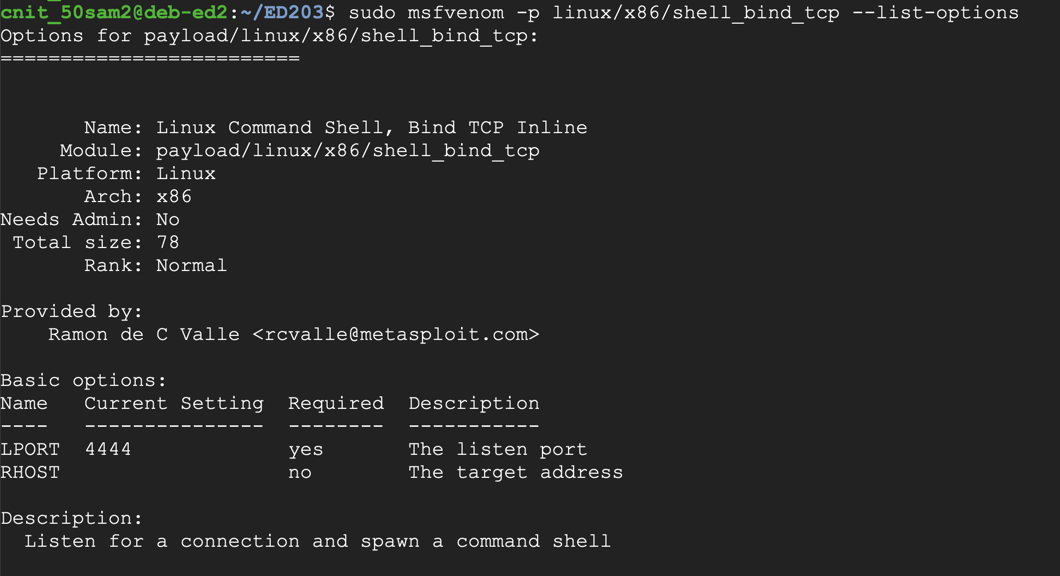


The exploit we want is highlighted above: **linux/x86/shell\_bind\_tcp**

To see the payload options, execute this command:

**sudo msfvenom -p linux/x86/shell\_bind\_tcp --list-options**

The top portion of the output shows the Basic options. The only parameter we really need is "LPORT", the port to listen on, as shown below. This port has a default value of 4444, but we'll choose a custom port.

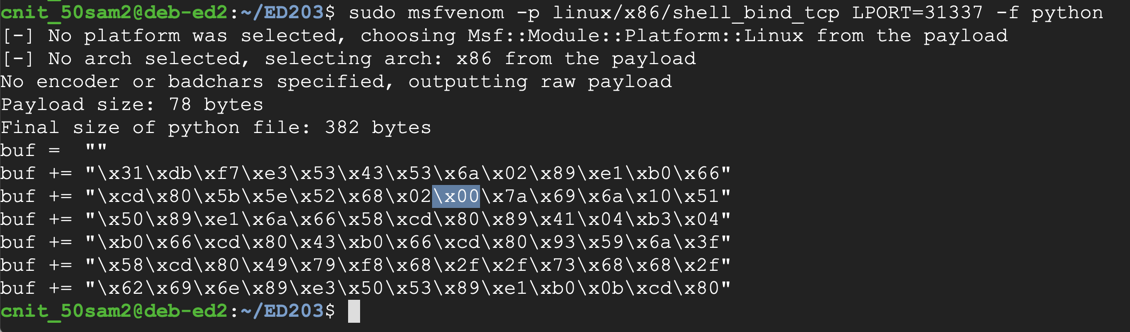


To generate Python exploit code, execute this command:

**sudo msfvenom -p linux/x86/shell\_bind\_tcp LPORT=31337 -f python**

The resulting payload isn't useful for us, because it contains a null byte ("\x00"), as shown below.

That null byte will terminate the string, preventing the shellcode after it from being processed by C programs.



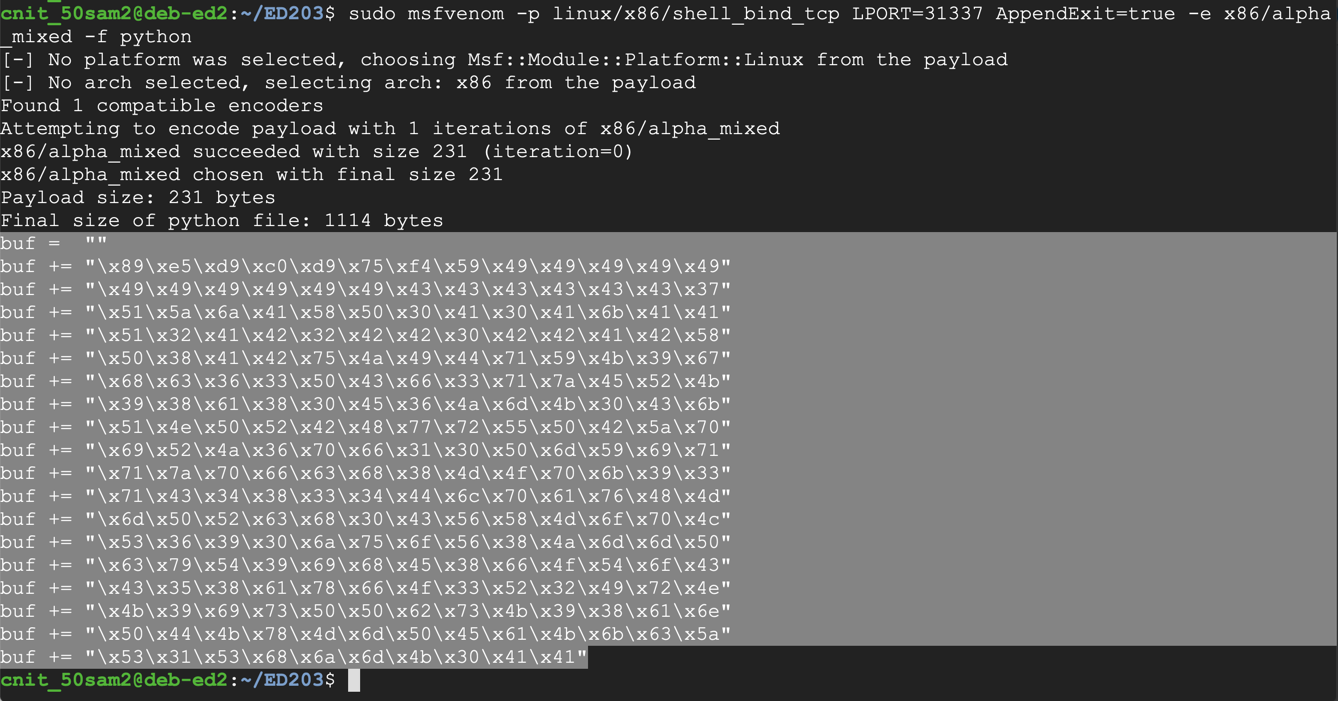
We could use the "-b '\x00'" switch to avoid null characters, but since we have plenty of room (1000 bytes or so), we can use the "**-e x86/alpha\_mixed**" switch, which will encode the exploit using only letters and numbers.

The '**AppendExit=true**' switch f makes the shellcode more reliable.

Execute this command:

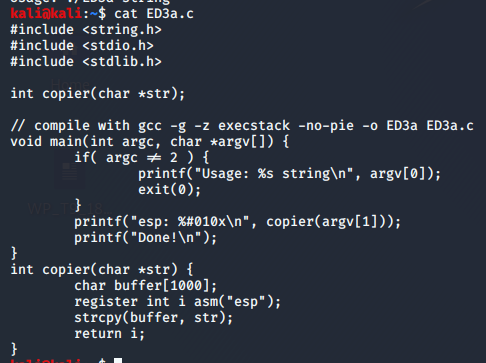
**sudo msfvenom -p linux/x86/shell\_bind\_tcp LPORT=31337 AppendExit=true -e x86/alpha\_mixed -f python**

This payload is longer--approximately 230 bytes (the exact length varies). Highlight the Python code and copy it to the clipboard, as shown below:



Exploit exercise using above shell code: <https://samsclass.info/127/proj/ED203.htm>

1. Program



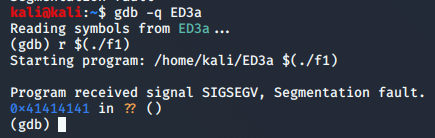
Vulnerability: in copier function, we have got a strcpy which won’t limit the input to 1000 bytes. Therefore we can perform buffer overflow.

1. F1 file:



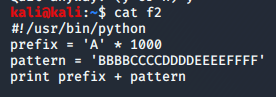
20 As are beyond designated memory space.

1. When run with 1020 A :



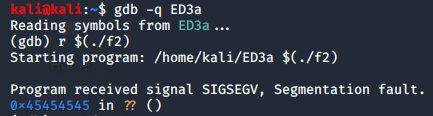
AAAA out of last 20 As manipulate the eip.

1. F2 file:



To identify which bytes manipulate the eip, we put in the pattern.

1. When run with f2:

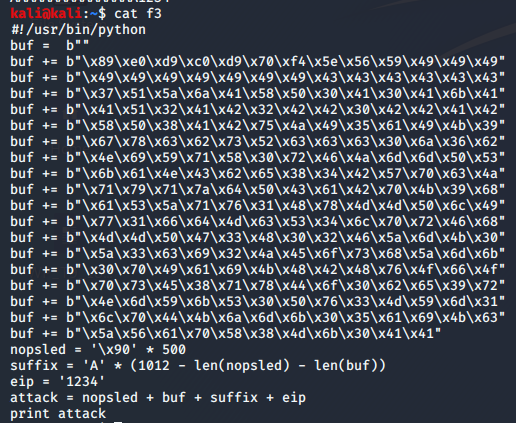


Ascii value of E=45. Therefore after 1012 , next 4 bytes manipulate eip.

1. Shell code:

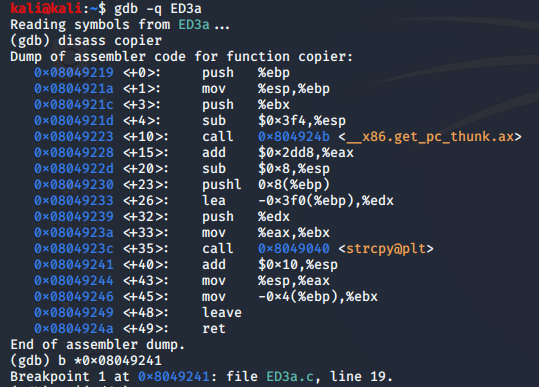


1. F3 file:



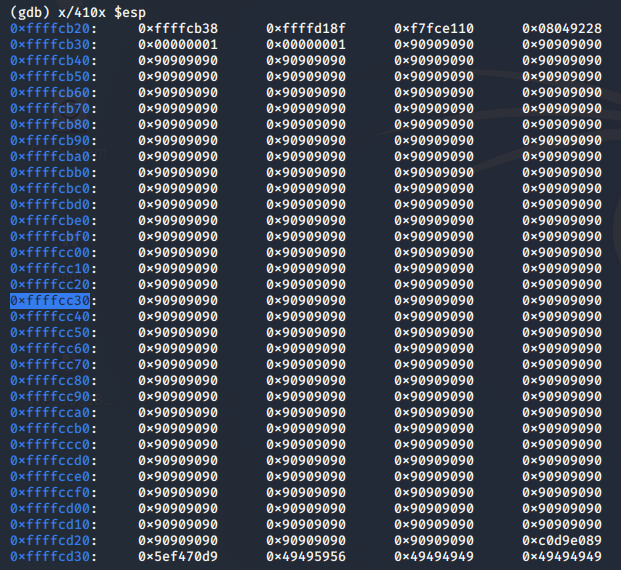
We will have 500 nop sled then the shell code then space left till 1012 filled with As and finally the bytes that will manipulate eip.

1. Setup breakpoint: after strcpy is done.(+40)



1. Choose address to redirect in nop sled:

You need to choose an address to put into $eip. If everything were perfect, you could simply use the address of the first byte of the shellcode. However, to give us some room for error, choose an address somewhere in the middle of the NOP sled



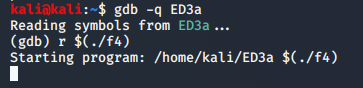
1. F4 file:



1. Success:

The program runs, and never returns a prompt, as shown below.

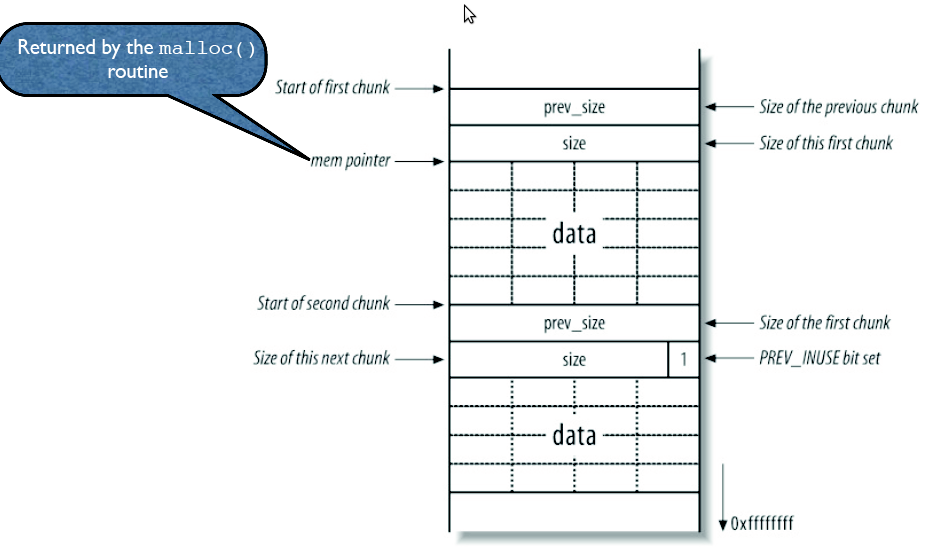
This is because it worked, and it's now running the payload.



## HEAP OVERFLOW

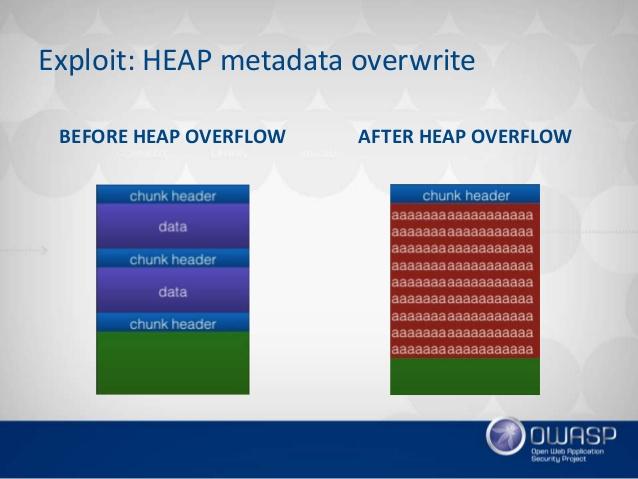
## What is Heap?

Heap is a memory region allotted to every program. Unlike stack, heap memory can be dynamically allocated. This means that the program can 'request' and 'release' memory from the heap segment whenever it requires. Also, this memory is global, i.e. it can be accessed and modified from anywhere within a program and is not localized to the function where it is allocated.



What is a heap overflow?

A heap overflow is a form of buffer overflow; it happens when a chunk of memory is allocated to the heap and data is written to this memory without any bound checking being done on the data. This is can lead to overwriting some critical data structures in the heap such as the heap headers, or any heap-based data such as dynamic object pointers.

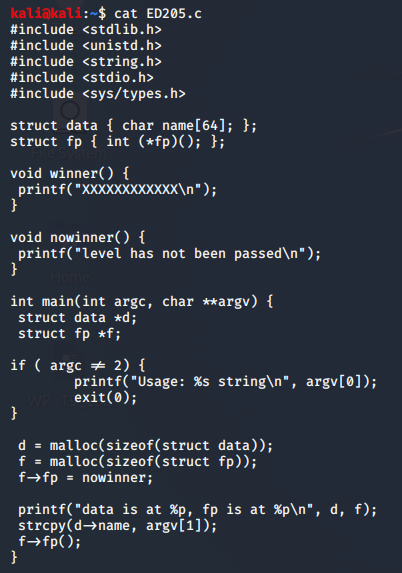


## What is heap and what does malloc() do(the process)?

<https://www.youtube.com/watch?v=HPDBOhiKaD8&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN&index=22&t=3s>

**Type 1: by overwriting some pointer to redirect code execution:**

Program:



We have a string: data and a pointer : fp on the heap.

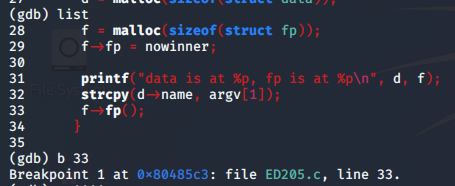
The program reads string using strcpy (which won’t limit the size to 64) into data.

After that we have the fp pointer pointing to nowinner function.

**Aim**: make fp point to winner by overflowing the heap.

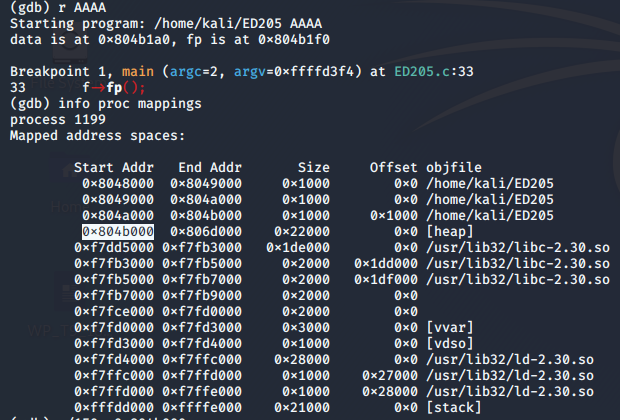
Observing the memory space:

1. Inserting Breakpoint



Put one after the strcpy i.e line 33

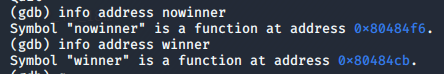
1. Run with AAAA and find the address of heap



Locate [heap] and the highlighted part is the starting address of the heap.

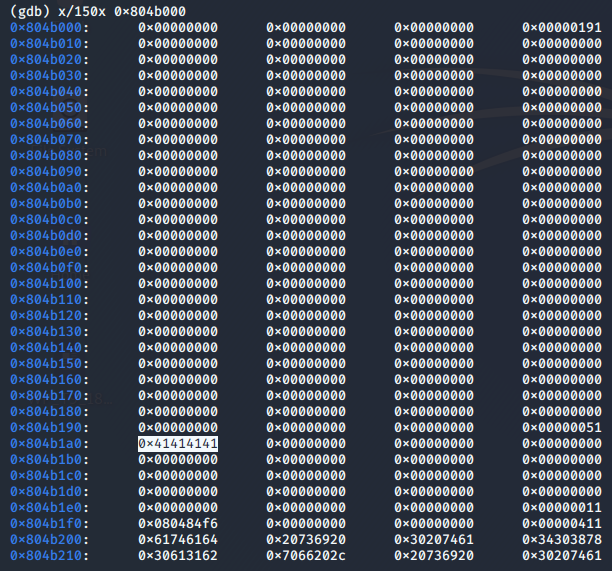
\*\*use info proc mappings to locate the heap.

1. To locate winner() and nowinner():



Note these both down.

1. To understand how and where data is stored on the heap:



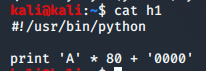
We see that string starts loading in heap from 0x804b1a0 (refer highlighted part)

You can notice that at 0x804b1f0( the address of nowinner) is saved. That must be the fp pointer that redirects the code to nowinner.

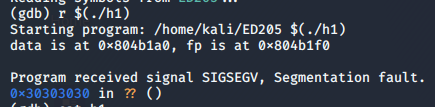
If you want to replace fp, calculating from 0x804b1a0 to 0x804b1f0: results in 80 bytes of overflow and next 4 containing replacement of nowinner.

h1 file:

We put 0000 in place of address of nowinner.



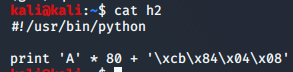
Run with h1:



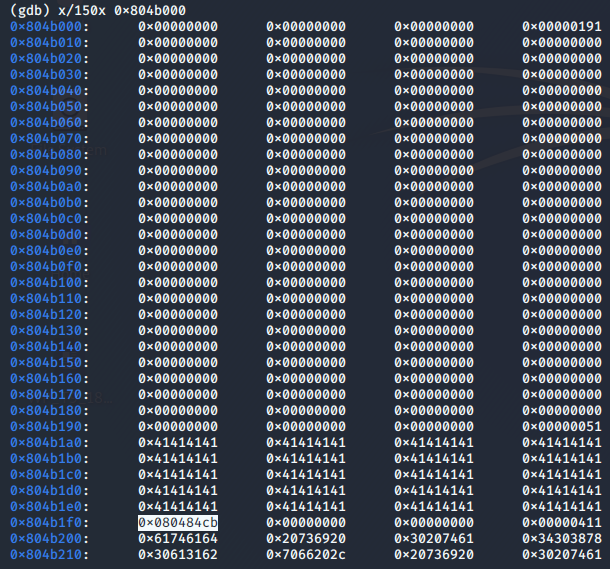
Eip is redirected successfully to 30303030 (ascii of 0 is 30) instead of nowinner.

h2 file:

We have to replace the address of nowinner to winner.

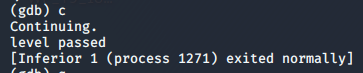


Run with h2 having breakpoint at line 33 and examine the heap:



Address of winner is stored where previously nowinner was stored.

Continuing,



Thus program is redirected to winner() and we have passed the level.

TYPE 2: Heap Overflow via Data Overwrite

<https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning>

There’s two types of binaries on any system: **statically linked** and **dynamically linked**.

Statically linked binaries are self-contained, containing all of the code necessary for them to run within the single file, and do not depend on any external libraries.

Dynamically linked binaries (which are the default when you run gcc and most other compilers) do not include a lot of functions, but rely on system libraries to provide a portion of the functionality. For example, when your binary uses printf to print some data, the actual implementation of printf is part of the system C library. Typically, on current GNU/Linux systems, this is provided by libc.so.6, which is the name of the current GNU Libc library.

In order to locate these functions, your program needs to know the address of printf to call it.

A strategy was developed to allow looking up all of these addresses when the program was run and providing a mechanism to call these functions from libraries. This is known as **relocation.**

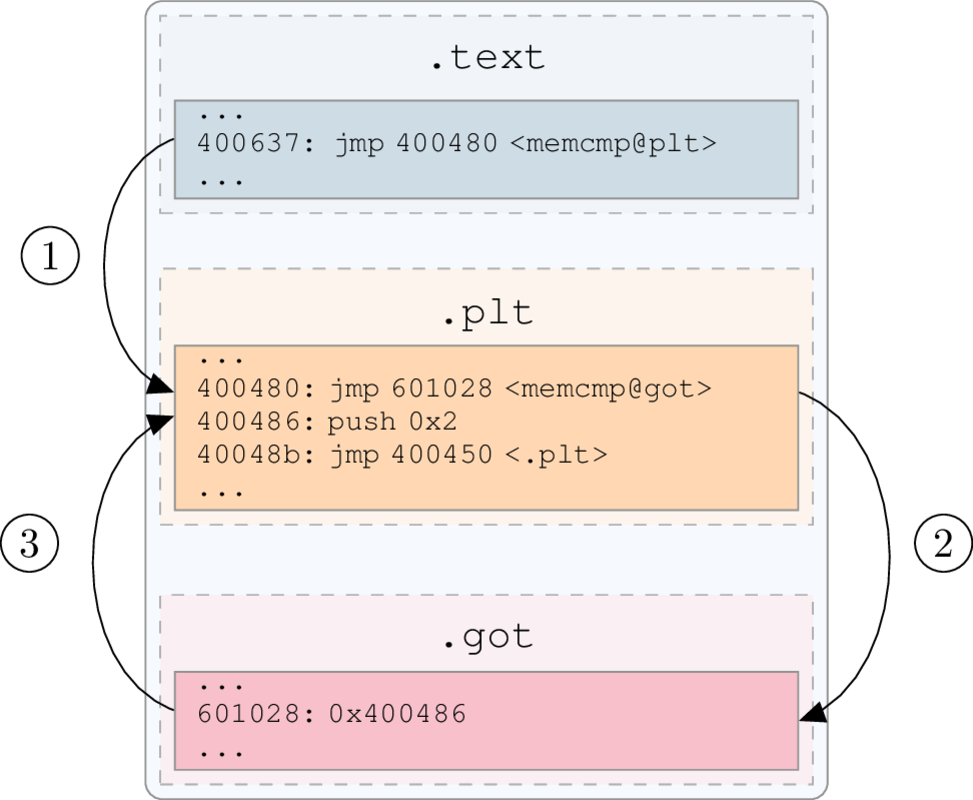
Looking at an ELF(Executable and Linkable Format) file, you will discover that it has a number of sections, and it turns out that relocations require *several* of these sections.

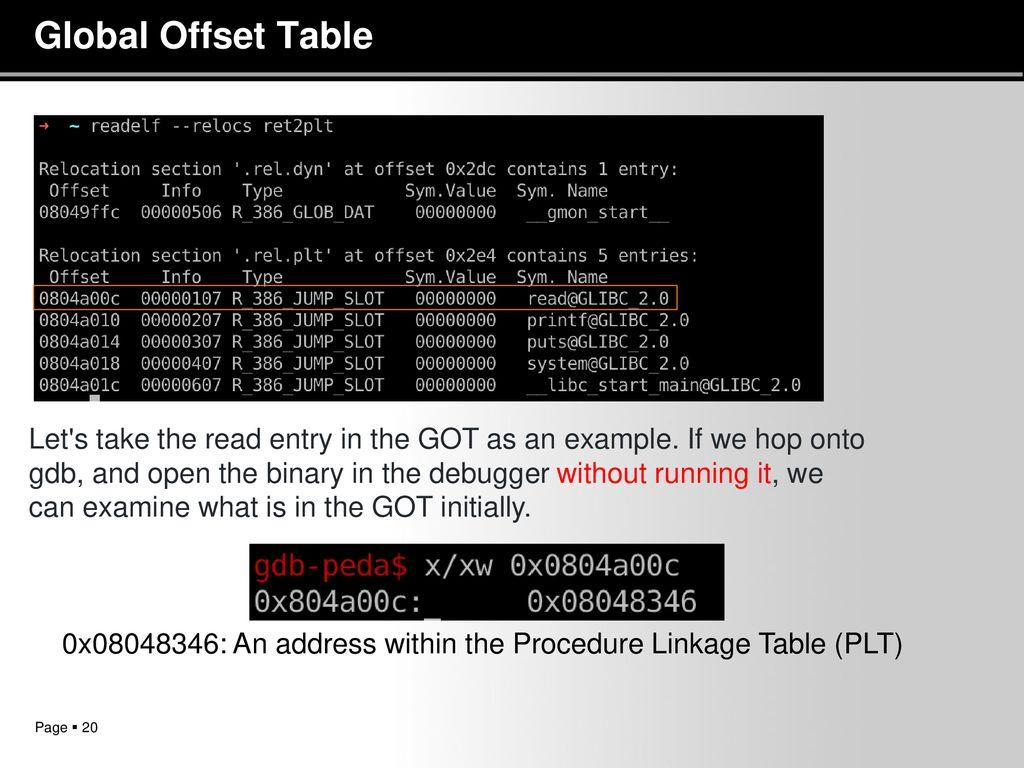
**.got**

This is the GOT, or **Global Offset Table** is used to resolve addresses.

**.plt**

This is the PLT, or **Procedure Linkage Table** which is used to call external procedures/functions whose address isn’t known in the time of linking, and is left to be resolved by the dynamic linker at the run time.

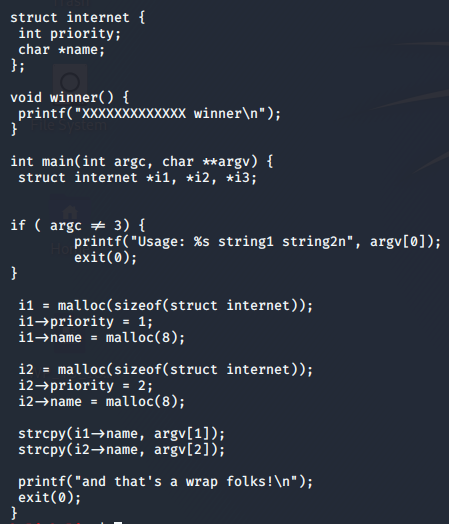




The .got.plt section is basically a giant array of function pointers. Maybe we could overwrite one of these and control execution from there. It turns out this is quite a common technique.

Essentially, any memory corruption primitive that will let you write to an arbitrary (attacker-controlled) address will allow you to overwrite a GOT entry.

Program:



A structure named "internet" is defined, which contains an integer (4 bytes) and a pointer to a string (4 bytes).

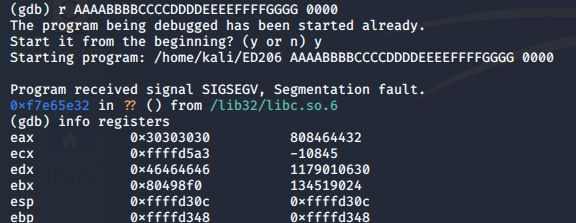
There's a function named winner(). As you might expect, our goal is to execute that function.

The main() routine creates two objects of type "internet" on the heap with malloc().

Then it copies the two command-line arguments into the strings in those objects without checking the input length.

Observe crash:





See that eax has 0000 and Edx has FFFF.

To see at which instruction program crashed:



Attempting to move the contents of $eax into RAM at the address in $edx

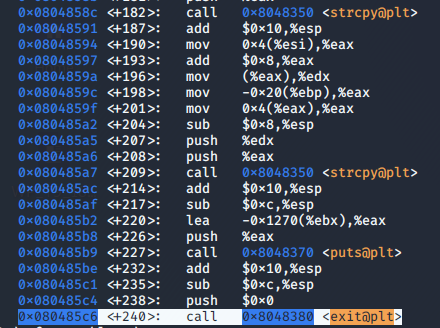
This means we can write to any memory location we wish, putting the data in place of '0000' and the address in place of 'FFFF'.

GETTING THE ADDRESS:

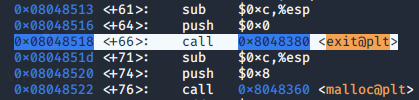
If we can write to that address, we can take over the program's execution when it calls "exit@plt".

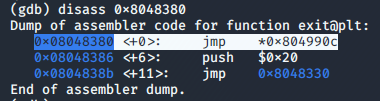
*Note: the "puts" address won't work because it contains "28" which is ascii for "(" and breaks the bash command line.*

1. By plt and got



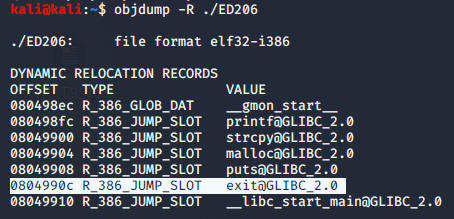
Disass 0x080485c6:





Therefore real address of exit is 0x804990c

1. Directly by obj dump



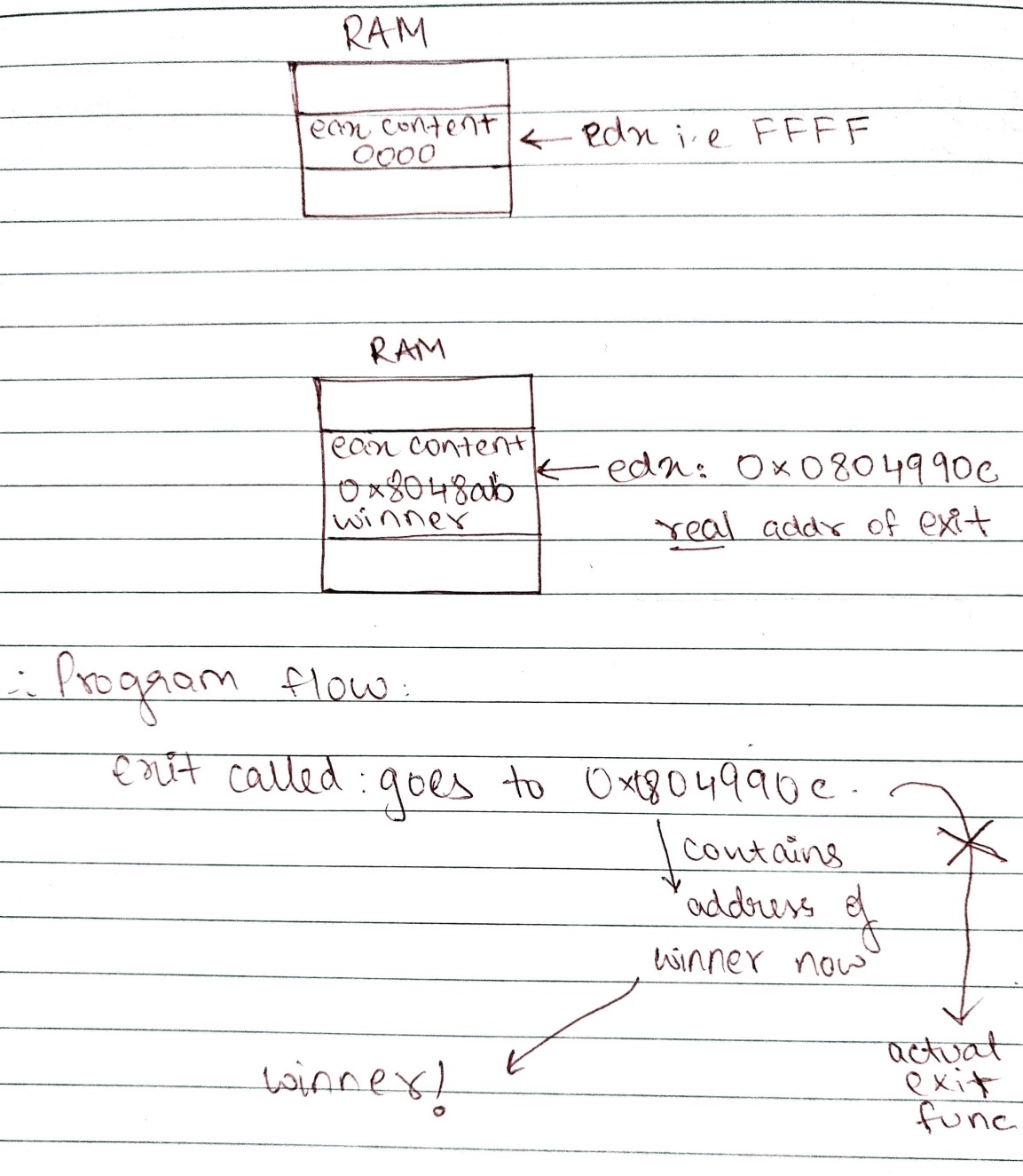
So, we have to replace FFFF with 0x0804990c

GETTING DATA:



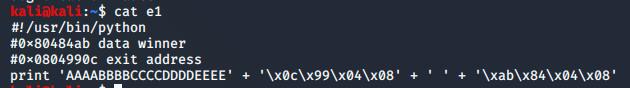
Replace ‘0000’ with address of winner.

VISUAL REPRESENTATION OF WHAT WE ARE DOING:



Writing exploit e1:

Buff + address of exit + ‘ ‘ + address of winner



Run:

