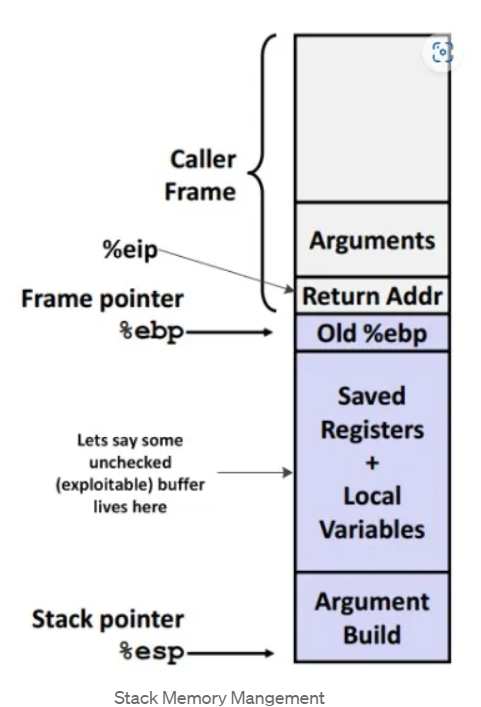
# Handout

## **Reversing ---- An Introduction**

First of all, we will discuss different components of a computer program memory map. Given is an abstract level memory map with some major components shown:



Every function in a C binary ends with a pop ret or leave ret instructions.

Pop simply removes the contents of top of the stack to the register specified in the argument.

The RET instruction pops the return address off the stack (which is pointed to by the stack pointer register) and then continues execution at that address.

pop rcx

jmp rcx

jmp instruction simply places the contents of the arg register in the rip register.

The LEAVE instruction

Following instructions are executed every time a function is called. These set of instructions are called function prologue

push rbp

move rbp,rsp

sub rsp, N

Or equivalently

enter N,0

Here N repreents the number of bytes in the stack needed by the program. The 0 is for nested functions.

Similarly, when we exit a function, following instructions are always executed:

move rsp,rbp

pop rbp

these set of instruction, also called epilogue, are equivalent to the leave instruction in assembly.

These are some basic reversing problems.

Purpose is to get some experience understand the assembly code generated from the binary. To view disassembly machine code into assembly code, you can use something like objdump.

**Example 1 --- Hello World**

#include <stdio.h>

void main(void)

{

puts("Hello World!");

}

Compile

$ gcc p1.c -o p1

Now p1 is the binary which we will view using objdump command

$ objdump -M intel -D p1 | less

0000000000001139 <main>:

1139: 55 push rbp

113a: 48 89 e5 mov rbp,rsp

113d: 48 8d 05 c0 0e 00 00 lea rax,[rip+0xec0] # 2004 <\_IO\_stdin\_used+0x4>

1144: 48 89 c7 mov rdi,rax

1147: e8 e4 fe ff ff call 1030 <puts@plt>

114c: 90 nop

114d: 5d pop rbp

114e: c3 ret

This is the assembly version of our program p1.c generated by the compiler.

In this code first two and last two lines of assembly represent the prologue and epilogue (or what ever we can call these).

The main is called by some compiler generated function.

The first line push rbp stores the rbp register of caller on stack. Now we can use rbp for our purpose.

Second line copies the value of rsp, which is pointing to the stack frame of main, to rbp. Now both rsp and rbp are pointing to the same location in memory.

The third line carries an lea instruction which is calculating an address by adding the contents of RIP with a constant. This is actually the address of the string passed to the puts() function.

What is rip+0xec0?

The rip register contains an address, and 0xec0 is the offset from this address to where the string starts. The string is stored in the .rodata (ro as in read only) section. This section is part of the binary file and is loaded in memory with the program. It usually contains program constants (i.e. strings).

The string address is calculated and stored in the rdi register using the lea instruction, therefore,

rdi = rip + 0xec0.

Let us examine the .rodata section.

$ objdump -d -s -j .rodata ./p1

This command will produce

Disassembly of section .rodata:

0000000000002000 <\_IO\_stdin\_used>:

2000: 01 00 02 00 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 ....Hello World!

Note the rodata section starts at 0x2000 while the string starts four bytes farther.

Let us calculate the address of rip, that is, the instruction in our assembly to which rip is pointing.

rip + 0xec0 = 0x2004

rip = 0x2004 - 0xec0 = 0x1144

And we can see that 0x1104 is the address of mov rdi,rax

This makes sense.

The fourth instruction calls the puts function. This function is implemented in the GNU C library and its address is stored in the plt section, hence the puts@plt. The plt section is used to locate library functions whose addresses are not known at link time.

According to the manual (man 3 puts), the puts function takes one parameter, a string.

Given that we know that rdi contains the address of the string to be printed, we can safely assume that puts gets its parameter value from rdi. Why is it so? The simplest answer: ABI (Application Binary Interface).

An ABI generally describes the data types (size, encoding, …), the structure of functions and function calls, … relative to a low level hardware format. For example, the calling conventions of an ABI describe how a call to a function is to be performed, which registers to set according to which function parameter, which register should hold the function's return value, …

In our case, we're interested in the x86-64 SYSTEM V AMD64 ABI, the standard in the UNIX world. This ABI states that when a call to a function is placed, the following registers are to be used, in order, and depending on the number of function parameters:

RDI, RSI, RDX, RCX, R8, R9 and rest on the stack.

The sixth instruction restores the previously saved rbp value into rbp. This is useless for this code.

The last instruction sets the instruction pointer to after the function call to continue execution.

**Example 2 Hello World with PRINTF Function**

# include <stdio.h>

int main(void)

{

int x = 10;

printf("x = %d\n",x);

return 0;

}

Let us compile and generate the assembly

$ gcc p2.c -o p

$ objdump -M intel -D p | less

0000000000001139 <main>:

1139: 55 push rbp

113a: 48 89 e5 mov rbp,rsp

113d: 48 83 ec 10 sub rsp,0x10

1141: c7 45 fc 0a 00 00 00 mov DWORD PTR [rbp-0x4],0xa

1148: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

114b: 89 c6 mov esi,eax

114d: 48 8d 05 b0 0e 00 00 lea rax,[rip+0xeb0] # 2004 <\_IO\_stdin\_used+0x4>

1154: 48 89 c7 mov rdi,rax

1157: b8 00 00 00 00 mov eax,0x0

115c: e8 cf fe ff ff call 1030 <printf@plt>

1161: b8 00 00 00 00 mov eax,0x0

1166: c9 leave

1167: c3 ret

In this program we have created a variable and assigned a value to it.

Next we use printf statement to print the value.

Most of the code generated by the compiler is similar to the one from example-1.

We will discuss the lines which are new to us.

113d: sub rsp,0x10

Here compiler has generated some space (16 bytes on the stack) for the local variables etc.

1141: mov DWORD PTR [rbp-0x4],0xa

the value of variable x which is 10 is copied on the stack. The value is stored in 4 bytes since it is an integer. The integer value starts 4 bytes below the rbp and move up.

1148: mov eax,DWORD PTR [rbp-0x4]

Here the same value from stack is copied into the eax register.

Since this will be second argument to the printf() function, in the next instruction it is copied to rsi.

mov esi,eax

Next the compiler places a zero in the eax (or rax) register and then call to printf is made. Why eax is set to zero, because it is used for the return values and printf has a return value based on number of characters printed or the EOF.

Again eax is set to zero so that now it can have the return value from the leave instruction.

**leave is exactly equivalent to**

mov esp, ebp # esp = ebp,

pop ebp

**Example 3 Hello World with PRINTF Function**

#include <stdio.h>

int main(void)

{

int x = 10;

char c = 'r';

printf("x = %d and c = %d\n",x,c);

return 0;

}

First one is compiled using no optimization

Second with with a -O3 option

0000000000001139 <main>:

1139: 55 push rbp

113a: 48 89 e5 mov rbp,rsp

113d: 48 83 ec 10 sub rsp,0x10

1141: c7 45 fc 0a 00 00 00 mov DWORD PTR [rbp-0x4],0xa

1148: c6 45 fb 72 mov BYTE PTR [rbp-0x5],0x72

114c: 0f be 55 fb movsx edx,BYTE PTR [rbp-0x5]

1150: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

1153: 89 c6 mov esi,eax

1155: 48 8d 05 a8 0e 00 00 lea rax,[rip+0xea8] # 2004 <\_IO\_stdin\_used+0x4>

115c: 48 89 c7 mov rdi,rax

115f: b8 00 00 00 00 mov eax,0x0

1164: e8 c7 fe ff ff call 1030 <printf@plt>

1169: b8 00 00 00 00 mov eax,0x0

116e: c9 leave

116f: c3 ret

$ gcc p3.c -O3 -o p

$ objdump -M intel -D p | less

000000000001050 <main>:

1050: 48 83 ec 08 sub rsp,0x8

1054: ba 72 00 00 00 mov edx,0x72

1059: be 0a 00 00 00 mov esi,0xa

105e: 31 c0 xor eax,eax

1060: 48 8d 3d 9d 0f 00 00 lea rdi,[rip+0xf9d] # 2004 <\_IO\_stdin\_used+0x4>

1067: e8 c4 ff ff ff call 1030 <printf@plt>

106c: 31 c0 xor eax,eax

106e: 48 83 c4 08 add rsp,0x8

1072: c3 ret

1073: 66 2e 0f 1f 84 00 00 cs nop WORD PTR [rax+rax\*1+0x0]

107a: 00 00 00

107d: 0f 1f 00 nop DWORD PTR [rax]

**Example 4 Hello World with arrays**

#include <stdio.h>

int main(void)

{

int x[4] = {1,2,3,4};

printf("x[1] = %d\n",x[1]);

return 0;

}

Compile and display the binary

$ gcc p4.c -o p

$ objdump -M intel -D p | less

0000000000001139 <main>:

1139: 55 push rbp

113a: 48 89 e5 mov rbp,rsp

113d: 48 83 ec 10 sub rsp,0x10

1141: c7 45 f0 01 00 00 00 mov DWORD PTR [rbp-0x10],0x1

1148: c7 45 f4 02 00 00 00 mov DWORD PTR [rbp-0xc],0x2

114f: c7 45 f8 03 00 00 00 mov DWORD PTR [rbp-0x8],0x3

1156: c7 45 fc 04 00 00 00 mov DWORD PTR [rbp-0x4],0x4

115d: 8b 45 f4 mov eax,DWORD PTR [rbp-0xc]

1160: 89 c6 mov esi,eax

1162: 48 8d 05 9b 0e 00 00 lea rax,[rip+0xe9b] # 2004 <\_IO\_stdin\_used+0x4>

1169: 48 89 c7 mov rdi,rax

116c: b8 00 00 00 00 mov eax,0x0

1171: e8 ba fe ff ff call 1030 <printf@plt>

1176: b8 00 00 00 00 mov eax,0x0

117b: c9 leave

117c: c3 ret

Each array value is stored on the stack. Note the location where first index is stored and so on.

Next second index of array is copied in eax and then in esi. (since this is the second argument to printf)

Then instruction LEA is used to compute the address of string, which is then stored in rdi as it is first argument to printf.

Finally the printf function is called.

**Example 5 Hello World with if-else**

#include <stdio.h>

int main(void)

{

int x = 10;

int y = 5;

if(x<=100)

{

y = y - 3;

printf("Less than\n");

}

else

{

y = y + 3;

printf("Greater than\n");

}

return 0;

}

Compile and generate binary using objdump.

0000000000001139 <main>:

1139: 55 push rbp

113a: 48 89 e5 mov rbp,rsp

113d: 48 83 ec 10 sub rsp,0x10

1141: c7 45 fc 0a 00 00 00 mov DWORD PTR [rbp-0x4],0xa

1148: c7 45 f8 05 00 00 00 mov DWORD PTR [rbp-0x8],0x5

114f: 83 7d fc 64 cmp DWORD PTR [rbp-0x4],0x64

1153: 7f 15 jg 116a <main+0x31>

1155: 83 6d f8 03 sub DWORD PTR [rbp-0x8],0x3

1159: 48 8d 05 a4 0e 00 00 lea rax,[rip+0xea4] # 2004 <\_IO\_stdin\_used+0x4>

1160: 48 89 c7 mov rdi,rax

1163: e8 c8 fe ff ff call 1030 <puts@plt>

1168: eb 13 jmp 117d <main+0x44>

116a: 83 45 f8 03 add DWORD PTR [rbp-0x8],0x3

116e: 48 8d 05 99 0e 00 00 lea rax,[rip+0xe99] # 200e <\_IO\_stdin\_used+0xe>

1175: 48 89 c7 mov rdi,rax

1178: e8 b3 fe ff ff call 1030 <puts@plt>

117d: b8 00 00 00 00 mov eax,0x0

1182: c9 leave

1183: c3 ret

**Example 6 Hello World with for loop**

#include <stdio.h>

int main(void)

{

int i = 0;

int limit = 5;

for(i = 0;i<limit; i++)

{

printf("%d ",i);

}

printf("\n");

return 0;

}

0000000000001149 <main>:

1149: 55 push rbp

114a: 48 89 e5 mov rbp,rsp

114d: 48 83 ec 10 sub rsp,0x10

1151: c7 45 fc 00 00 00 00 mov DWORD PTR [rbp-0x4],0x0

1158: c7 45 f8 05 00 00 00 mov DWORD PTR [rbp-0x8],0x5

115f: c7 45 fc 00 00 00 00 mov DWORD PTR [rbp-0x4],0x0

1166: eb 1d jmp 1185 <main+0x3c>

1168: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

116b: 89 c6 mov esi,eax

116d: 48 8d 05 90 0e 00 00 lea rax,[rip+0xe90] # 2004 <\_IO\_stdin\_used+0x4>

1174: 48 89 c7 mov rdi,rax

1177: b8 00 00 00 00 mov eax,0x0

117c: e8 bf fe ff ff call 1040 <printf@plt>

1181: 83 45 fc 01 add DWORD PTR [rbp-0x4],0x1

1185: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

1188: 3b 45 f8 cmp eax,DWORD PTR [rbp-0x8]

118b: 7c db jl 1168 <main+0x1f>

118d: bf 0a 00 00 00 mov edi,0xa

1192: e8 99 fe ff ff call 1030 <putchar@plt>

1197: b8 00 00 00 00 mov eax,0x0

119c: c9 leave

119d: c3 ret

* Places the values of i and limit on stack
* Jumps to 1185
* Here it compares the value placed in eax with the limit value placed on stack
* If less, jumps to 1168
* Here it places the arguments to printf in appropriate registes, and calls it
* Then it increments the value of i placed in eax.
* Compares and if less than the value of limit jumps to 1168.
* Now if value in i is equal to limit , it moves down.
* Here it places 10, ascii of \n in edi, and calls putchar, why not printf? Just an innocent optimization 😊

**Example 7 Hello World with function call**

include <stdio.h>

int foo(int,int);

int main(void)

{

int val1 = 10;

int val2 = 20;

int sum = foo(val1,val2);

printf("sum is: %d\n",sum);

return 0;

}

int foo(int a, int b)

{

int out = a + b;

return out;

}

Compile and generate the objdump. Here we have two binaries: one for main and other for foo

0000000000001139 <main>:

1139: 55 push rbp

113a: 48 89 e5 mov rbp,rsp

113d: 48 83 ec 10 sub rsp,0x10

1141: c7 45 fc 0a 00 00 00 mov DWORD PTR [rbp-0x4],0xa

1148: c7 45 f8 14 00 00 00 mov DWORD PTR [rbp-0x8],0x14

114f: 8b 55 f8 mov edx,DWORD PTR [rbp-0x8]

1152: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

1155: 89 d6 mov esi,edx

1157: 89 c7 mov edi,eax

1159: e8 23 00 00 00 call 1181 <foo>

115e: 89 45 f4 mov DWORD PTR [rbp-0xc],eax

1161: 8b 45 f4 mov eax,DWORD PTR [rbp-0xc]

1164: 89 c6 mov esi,eax

1166: 48 8d 05 97 0e 00 00 lea rax,[rip+0xe97] # 2004 <\_IO\_stdin\_used+0x4>

116d: 48 89 c7 mov rdi,rax

1170: b8 00 00 00 00 mov eax,0x0

1175: e8 b6 fe ff ff call 1030 <printf@plt>

117a: b8 00 00 00 00 mov eax,0x0

117f: c9 leave

1180: c3 ret

0000000000001181 <foo>:

1181: 55 push rbp

1182: 48 89 e5 mov rbp,rsp

1185: 89 7d ec mov DWORD PTR [rbp-0x14],edi

1188: 89 75 e8 mov DWORD PTR [rbp-0x18],esi

118b: 8b 55 ec mov edx,DWORD PTR [rbp-0x14]

118e: 8b 45 e8 mov eax,DWORD PTR [rbp-0x18]

1191: 01 d0 add eax,edx

1193: 89 45 fc mov DWORD PTR [rbp-0x4],eax

1196: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

1199: 5d pop rbp

119a: c3 ret