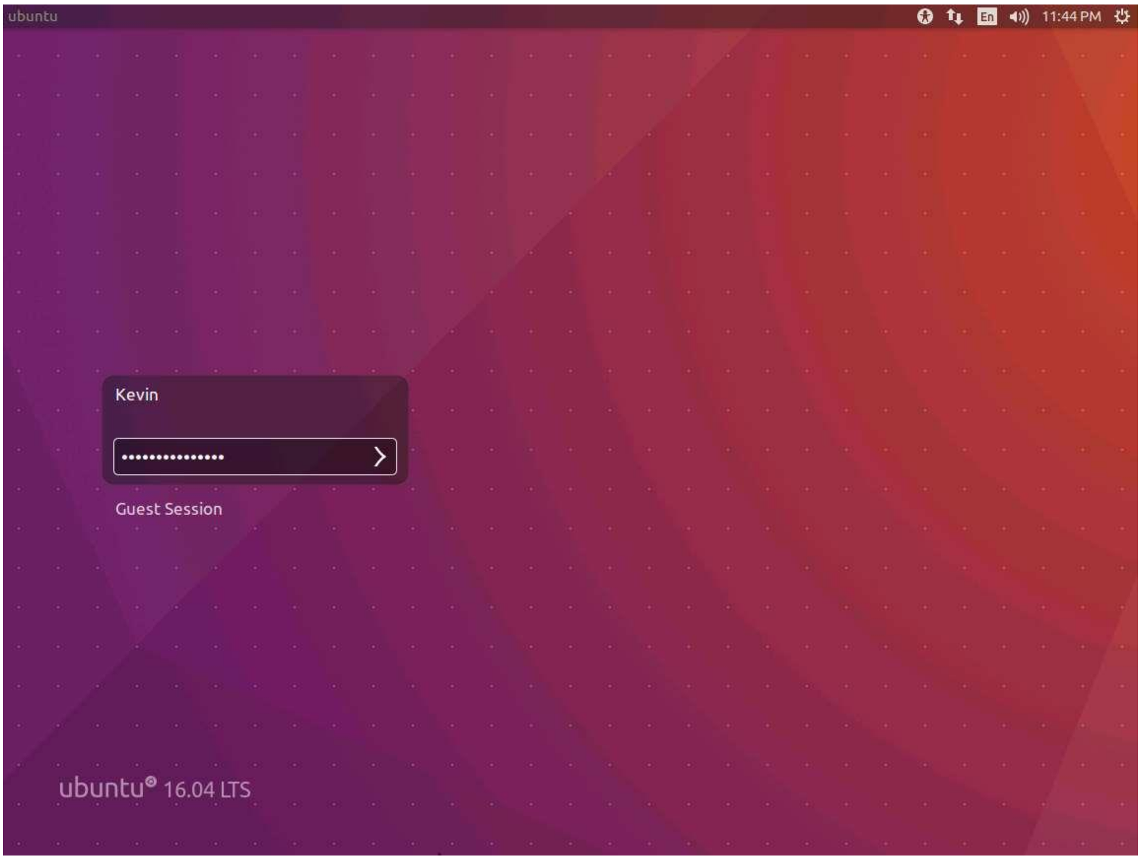
Exercise 4: Advanced Binary Analysis

Lab Objectives:

In this lab, we review the process for a customized disassembly of a binary. We will only provide a brief overview of this amazing process. Once you learn Advanced Binary Analysis, you will not face any obstacles in injecting code into the binaries that you are attempting to analyze.

Lab Tasks:

1. □ Login to the Software-Test-Linux machine using **studentpassword** as Password.



2. □ We will review a simple process of code obfuscation that deploys the method of instruction overlapping. The example we are using here is a simple reverse engineering of an exclusive or (xor) binary.

3. □ As mentioned in the slides, the assumption is that most instructions are mapped as follows:

   a. Each byte in a binary is mapped to at least one instruction.

   b. An instruction is contained in a single basic block.

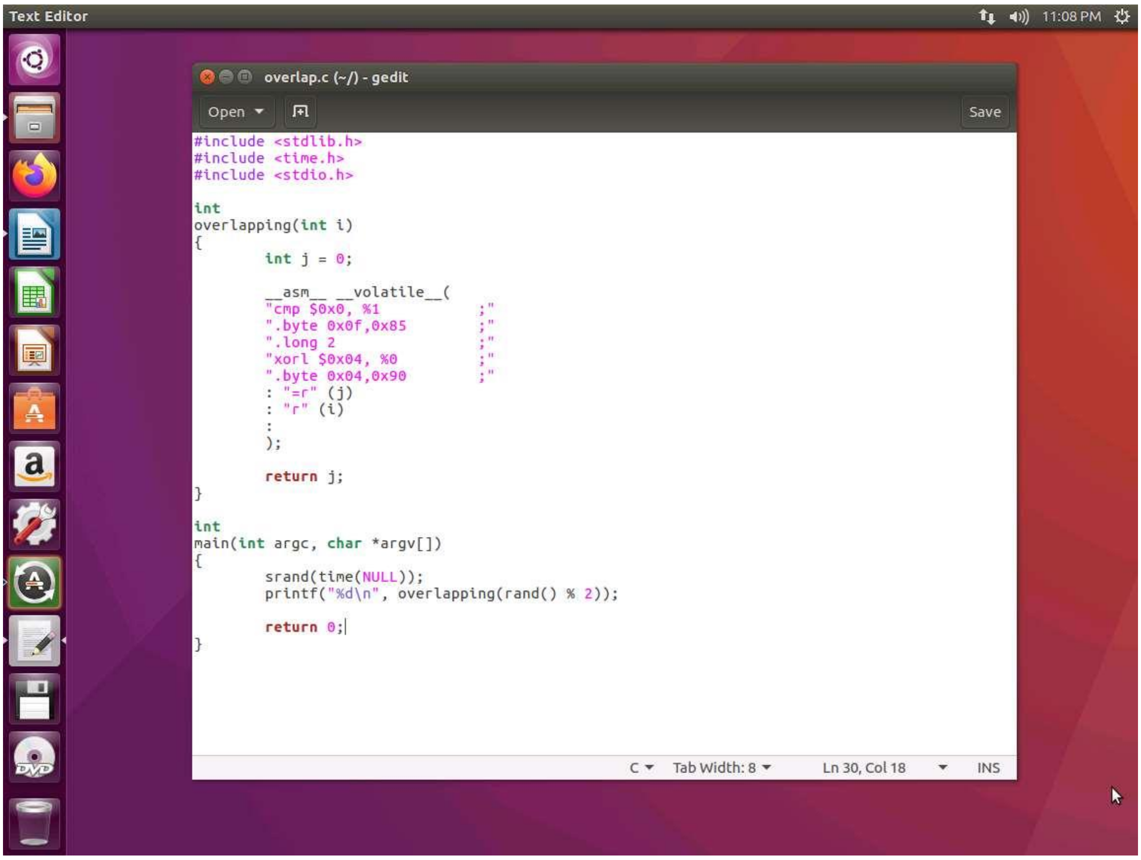
4. □ As a result of this, a disassembler does not look for chunks of code to overlap. Consequently, when instructions overlap, it makes it more difficult to reverse engineer.

5. □ We can use this technique because x86 instructions vary in length. As a result, the processor does not enforce instruction alignment, which allows for code to occupy the space of another code instruction.

6. □ You can disassemble from the middle of one instruction; this will yield another instruction that overlaps the first instruction.

7. □ Remember that this is made easy within x86, because of the dense instruction set where virtually any byte sequence corresponds to some valid instruction.

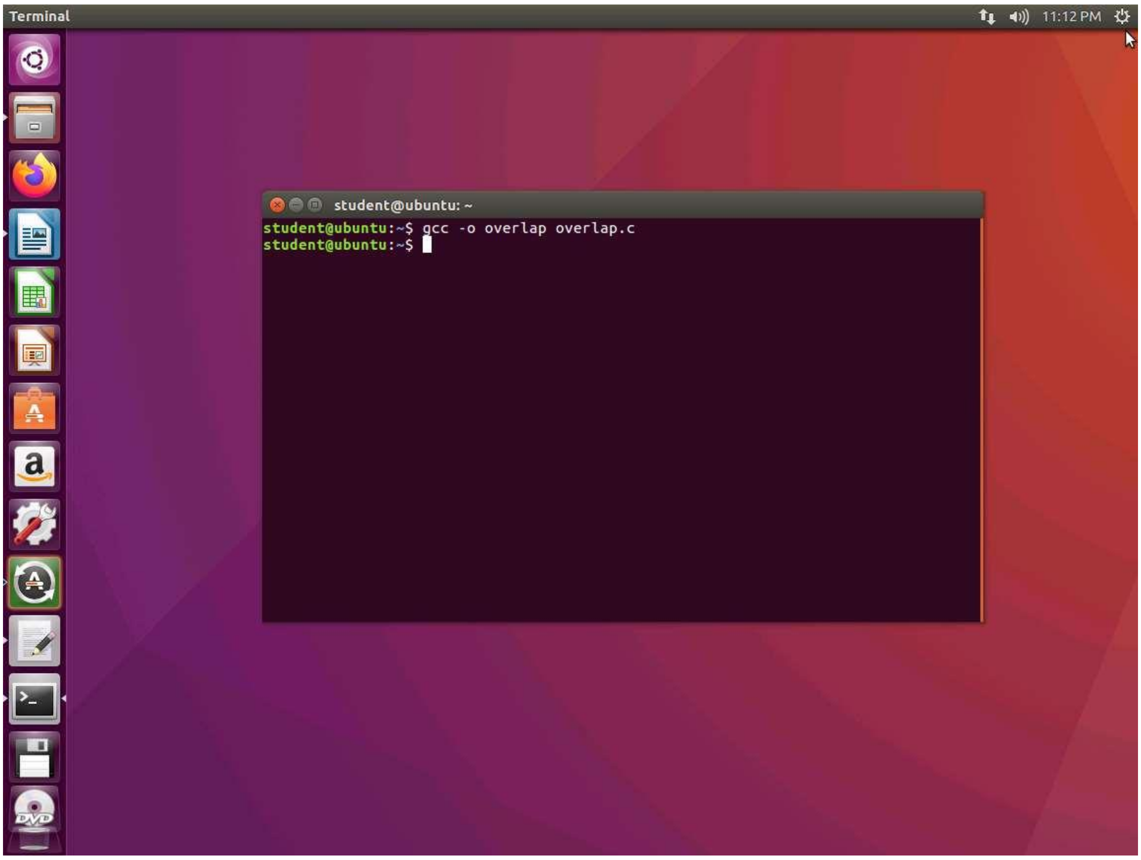
8. □ We will be using the simple code example shown in the following screenshot that uses overlapping instruction.



```c
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

int
overlapping(int i)
{
        int j = 0;

        __asm__ __volatile__(
        "cmp $0x0, %1                ;"
        ".byte 0x0f,0x85             ;"
        ".long 2                     ;"
        "xorl $0x04, %0              ;"
        ".byte 0x04,0x90             ;"
        : "=r" (j)
        : "r" (i)
        :
        );

        return j;
}

int
main(int argc, char *argv[])
{
        srand(time(NULL));
        printf("%d\n", overlapping(rand() % 2));

        return 0;
}
```

9. □ Enter the code shown above and save it as **overlap.c**.

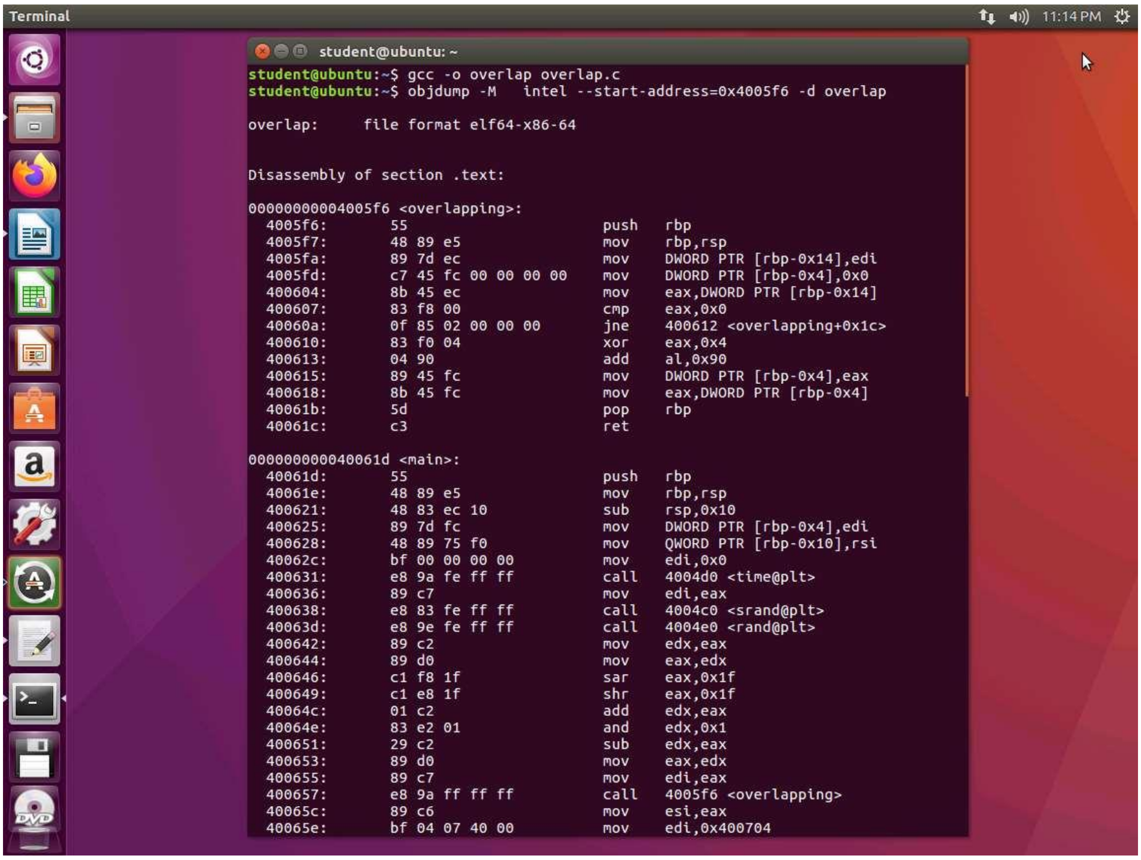
10. □ Once you have created and saved the file, compile it by entering **gcc -o overlap overlap.c**.



```
student@ubuntu:~$ gcc -o overlap overlap.c
student@ubuntu:~$ 
```

11. □ We can review the example of a simple overlap by entering the following code and examining our simple example and the corresponding object code.

12. □ Enter **objdump -M intel --start-address=0x4005f6 -d overlap**. The output of the command is shown in the following screenshot.



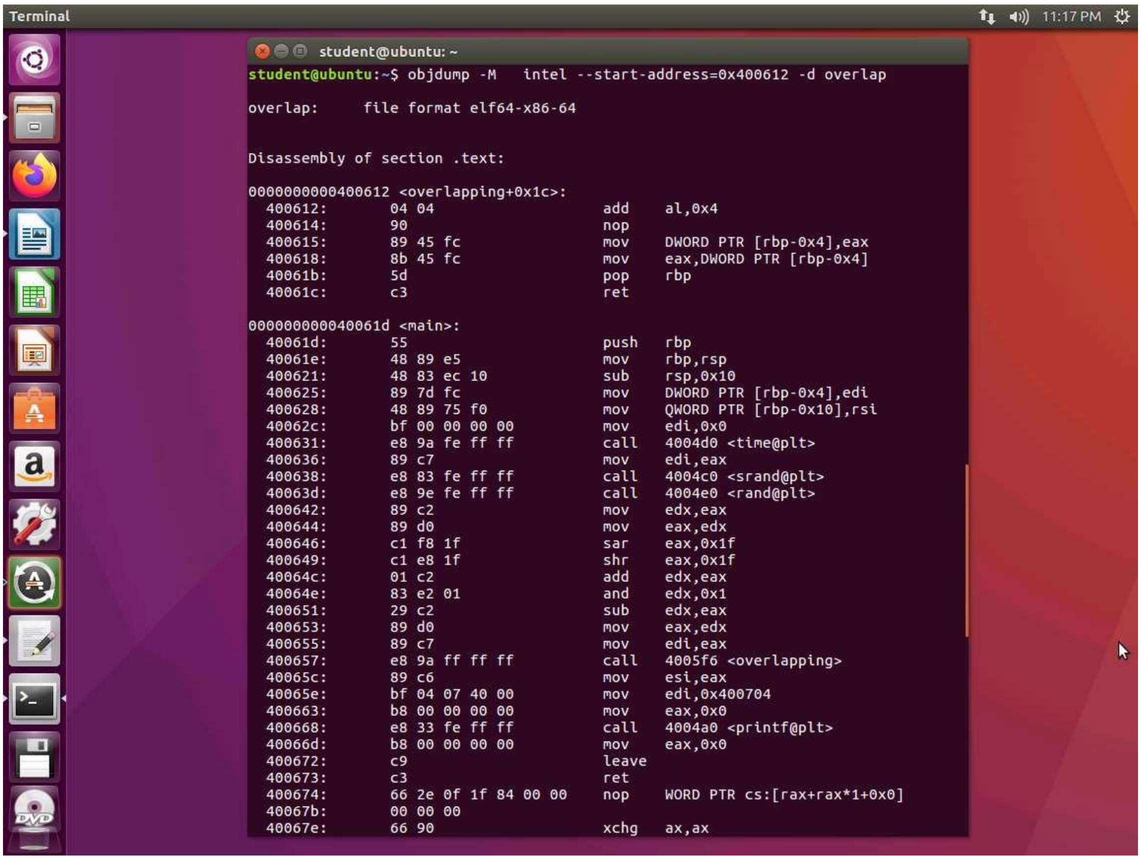
```
student@ubuntu:~$ gcc -o overlap overlap.c
student@ubuntu:~$ objdump -M   intel --start-address=0x4005f6 -d overlap

overlap:     file format elf64-x86-64


Disassembly of section .text:

00000000004005f6 <overlapping>:
  4005f6:       55                      push   rbp
  4005f7:       48 89 e5                mov    rbp,rsp
  4005fa:       89 7d ec                mov    DWORD PTR [rbp-0x14],edi
  4005fd:       c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
  400604:       8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
  400607:       83 f8 00                cmp    eax,0x0
  40060a:       0f 85 02 00 00 00       jne    400612 <overlapping+0x1c>
  400610:       83 f0 04                xor    eax,0x4
  400613:       04 90                   add    al,0x90
  400615:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  400618:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  40061b:       5d                      pop    rbp
  40061c:       c3                      ret

000000000040061d <main>:
  40061d:       55                      push   rbp
  40061e:       48 89 e5                mov    rbp,rsp
  400621:       48 83 ec 10             sub    rsp,0x10
  400625:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  400628:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  40062c:       bf 00 00 00 00          mov    edi,0x0
  400631:       e8 9a fe ff ff          call   4004d0 <time@plt>
  400636:       89 c7                   mov    edi,eax
  400638:       e8 83 fe ff ff          call   4004c0 <srand@plt>
  40063d:       e8 9e fe ff ff          call   4004e0 <rand@plt>
  400642:       89 c2                   mov    edx,eax
  400644:       89 d0                   mov    eax,edx
  400646:       c1 f8 1f                sar    eax,0x1f
  400649:       c1 e8 1f                shr    eax,0x1f
  40064c:       01 c2                   add    edx,eax
  40064e:       83 e2 01                and    edx,0x1
  400651:       29 c2                   sub    edx,eax
  400653:       89 d0                   mov    eax,edx
  400655:       89 c7                   mov    edi,eax
  400657:       e8 9a ff ff ff          call   4005f6 <overlapping>
  40065c:       89 c6                   mov    esi,eax
  40065e:       bf 04 07 40 00          mov    edi,0x400704
```

13. □ As the above screenshot shows, we have the block of code. As you review it, the first address ending in **5f6** is our parameter "**i**". The local variable "**j**" is located at an address ending in **60a**. Again, the instruction for the jne located at address ending in **60a** jumps into the middle of the instruction instead of into an address of the instruction. For example, the instruction that performs the **xor** is located at the address ending in **610**.

14. □ Most of the disassemblers will only disassemble these instructions that are shown in the screenshot. As a result, they would miss the overlapping instruction that is located at the address ending in **612**.

15. □ As the code shows, if **i=0**, then the jump is not taken, and it falls the rest of the way through the code. The other option is that when **i != 0**, the hidden code will execute. So how do you determine this? Hopefully, you said by entering the address that the jne is jumping to and if you did, you are correct!

16. □ To see this, enter **objdump -M intel --start-address=0x400612 -d overlap**. An example of the output of this is shown in the following screenshot.



```
student@ubuntu:~$ objdump -M  intel --start-address=0x400612 -d overlap

overlap:     file format elf64-x86-64


Disassembly of section .text:

0000000000400612 <overlapping+0x1c>:
  400612:	04 04                	add    al,0x4
  400614:	90                   	nop
  400615:	89 45 fc             	mov    DWORD PTR [rbp-0x4],eax
  400618:	8b 45 fc             	mov    eax,DWORD PTR [rbp-0x4]
  40061b:	5d                   	pop    rbp
  40061c:	c3                   	ret

000000000040061d <main>:
  40061d:	55                   	push   rbp
  40061e:	48 89 e5             	mov    rbp,rsp
  400621:	48 83 ec 10          	sub    rsp,0x10
  400625:	89 7d fc             	mov    DWORD PTR [rbp-0x4],edi
  400628:	48 89 75 f0          	mov    QWORD PTR [rbp-0x10],rsi
  40062c:	bf 00 00 00 00       	mov    edi,0x0
  400631:	e8 9a fe ff ff       	call   4004d0 <time@plt>
  400636:	89 c7                	mov    edi,eax
  400638:	e8 83 fe ff ff       	call   4004c0 <srand@plt>
  40063d:	e8 9e fe ff ff       	call   4004e0 <rand@plt>
  400642:	89 c2                	mov    edx,eax
  400644:	89 d0                	mov    eax,edx
  400646:	c1 f8 1f             	sar    eax,0x1f
  400649:	c1 e8 1f             	shr    eax,0x1f
  40064c:	01 c2                	add    edx,eax
  40064e:	83 e2 01             	and    edx,0x1
  400651:	29 c2                	sub    edx,eax
  400653:	89 d0                	mov    eax,edx
  400655:	89 c7                	mov    edi,eax
  400657:	e8 9a ff ff ff       	call   4005f6 <overlapping>
  40065c:	89 c6                	mov    esi,eax
  40065e:	bf 04 07 40 00       	mov    edi,0x400704
  400663:	b8 00 00 00 00       	mov    eax,0x0
  400668:	e8 33 fe ff ff       	call   4004a0 <printf@plt>
  40066d:	b8 00 00 00 00       	mov    eax,0x0
  400672:	c9                   	leave
  400673:	c3                   	ret
  400674:	66 2e 0f 1f 84 00 00 	nop    WORD PTR cs:[rax+rax*1+0x0]
  40067b:	00 00 00
  40067e:	66 90                	xchg   ax,ax
```

17. □ As the above screenshot shows, when **i != 0** then we take the hidden branch. This result adds the value in **al** to **0x4**, which results in a significant change in the code and changes the return value.

18. □ While we did reveal the instruction at the addresses ending **612** and **614** with our reverse engineering, we have now hidden the instructions located at the addresses ending in **610** and **611**.

19. □ Due to this, we now face another challenge. This is why this process can be time consuming, especially when there are multiple locations where obfuscation is deployed. As with anything else, it takes extensive practice.

20. □ The lab objectives have been achieved.