CS 395: Binary Exploitation in Linux

Week 10: Patching Binaries And Hooking

What is Patching?

- Patching is the act of modifying a binary file so that it does something that's advantageous for an attacker.
- This involves directly modifying the bytes inside of the binary file.
- This only works if you have write permissions on an executable.
  - For this reason, patching usually cannot be used for remote attacks.
- You are NOT allowed to patch any binaries or hook any functions for your final project unless the program specifically states that you are allowed to do so.
- You should always make a backup of the binary before patching it because you might corrupt the file.

When Is This Useful?

- In most scenarios, attackers are trying to exploit a remote target, so patching won’t be very useful.
- However, there are instances where this is useful for an attacker.
- If a product requires a license key in order to work, and if the license key is hardcoded, you could directly modify the license key to use the program (WARNING: VERY ILLEGAL!)
- You could also add your own functions to the binary that get executed instead of the original functions.
  - This technique, known as hooking, involves overwriting function pointers.
  - Hooking can be used to modify the program’s behavior.
  - Hooking usually used in real-life to intercept data and understand how it’s being used.

Points Program

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ ./points
You have 0 points.
You don't have enough points to win.
```

Points Program In Ghidra

```c

undefined8 main(void)

{
  printf("You have %d points.\n",(ulong)points);
  if ((int)points < 100) {
    puts("You don\'t have enough points to win.");
  }
  else {
    puts("Congrats, you get a shell!");
    system("/bin/sh");
  }
  return 0;
}

```

Object Dump

```
0000000000001155 <main>:
    1155:       55                      push   rbp
    1156:       48 89 e5                mov    rbp,rsp
    1159:       48 83 ec 10             sub    rsp,0x10
    115d:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
    1160:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
    1164:       8b 05 da 2e 00 00       mov    eax,DWORD PTR [rip+0x2eda]        # 4044 <points>
    116a:       89 c6                   mov    esi,eax
    116c:       48 8d 3d 95 0e 00 00    lea    rdi,[rip+0xe95]        # 2008 <_IO_stdin_used+0x8>
    1173:       b8 00 00 00 00          mov    eax,0x0
    1178:       e8 d3 fe ff ff          call   1050 <printf@plt>
    117d:       8b 05 c1 2e 00 00       mov    eax,DWORD PTR [rip+0x2ec1]        # 4044 <points>
    1183:       83 f8 63                cmp    eax,0x63
    1186:       7e 1a                   jle    11a2 <main+0x4d>
    1188:       48 8d 3d 8e 0e 00 00    lea    rdi,[rip+0xe8e]        # 201d <_IO_stdin_used+0x1d>
    118f:       e8 9c fe ff ff          call   1030 <puts@plt>
    1194:       48 8d 3d 9d 0e 00 00    lea    rdi,[rip+0xe9d]        # 2038 <_IO_stdin_used+0x38>
    119b:       e8 a0 fe ff ff          call   1040 <system@plt>
    11a0:       eb 0c                   jmp    11ae <main+0x59>
    11a2:       48 8d 3d 97 0e 00 00    lea    rdi,[rip+0xe97]        # 2040 <_IO_stdin_used+0x40>
    11a9:       e8 82 fe ff ff          call   1030 <puts@plt>
    11ae:       b8 00 00 00 00          mov    eax,0x0
    11b3:       c9                      leave
    11b4:       c3                      ret
    11b5:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
    11bc:       00 00 00
    11bf:       90                      nop
```

If Statement In Assembly

- At 0x117d in the object dump, you see the following code:

```
8b 05 c1 2e 00 00       mov  eax,DWORD PTR [rip+0x2ec1]
83 f8 63                cmp  eax,0x63
7e 1a                   jle  11a2 <main+0x4d>
```

- This is where the program checks whether the points variable is less than 100 or not.
  - If it is, then it will jump over the call to system().
- What if we replaced the first line with a MOV EAX, 0x64 instruction?
  - This would make it so that the jump never occurs.
  - Because the jump never occurs, we get to call system().

The Bytes That We’ll Use

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ cat test.asm
section .text
mov eax, 0x64

┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ nasm -f elf64 -o test.o test.asm

┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ objdump -d test.o -M intel

test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   b8 64 00 00 00          mov    eax,0x64
```

The Bytes That We'll Use (Cont.)

- Note that we are modifying line 0x117d, which contains the bytes 8b 05 c1 2e 00 00.
  - This is six bytes long.
- Our MOV EAX, 0x64 instruction comes out to be b8 64 00 00 00 in hexadecimal, which is five bytes long.
- The fact that it is not exactly six bytes long will cause problems for us because the file gets misaligned.
- To get around this, we will insert a NOP at the end of our code.

Patching The Binary

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ python3
Python 3.8.6 (default, Sep 25 2020, 09:36:53)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> with open('./points', 'rb+') as f:
...     data = bytearray(b'\xb8\x64\x00\x00\x00\x90')
...     f.seek(0x117d)
...     f.write(data)
...
```

Executing The Patched Binary

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ ./points
You have 0 points.
Congrats, you get a shell!
$ whoami
cs395
$ ls
points  points.c  test.asm  test.o
$ ping google.com
PING google.com (172.217.164.142) 56(84) bytes of data.
64 bytes from iad30s24-in-f14.1e100.net (172.217.164.142): icmp_seq=1 ttl=119 time=5.27 ms
64 bytes from iad30s24-in-f14.1e100.net (172.217.164.142): icmp_seq=2 ttl=119 time=7.64 ms
64 bytes from iad30s24-in-f14.1e100.net (172.217.164.142): icmp_seq=3 ttl=119 time=14.8 ms
```

Patched Binary’s Object Dump

```
0000000000001155 <main>:
    1155:       55                      push   rbp
    1156:       48 89 e5                mov    rbp,rsp
    1159:       48 83 ec 10             sub    rsp,0x10
    115d:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
    1160:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
    1164:       8b 05 da 2e 00 00       mov    eax,DWORD PTR [rip+0x2eda]        # 4044 <points>
    116a:       89 c6                   mov    esi,eax
    116c:       48 8d 3d 95 0e 00 00    lea    rdi,[rip+0xe95]        # 2008 <_IO_stdin_used+0x8>
    1173:       b8 00 00 00 00          mov    eax,0x0
    1178:       e8 d3 fe ff ff          call   1050 <printf@plt>
    117d:       b8 64 00 00 00          mov    eax,0x64
    1182:       90                      nop
    1183:       83 f8 63                cmp    eax,0x63
    1186:       7e 1a                   jle    11a2 <main+0x4d>
    1188:       48 8d 3d 8e 0e 00 00    lea    rdi,[rip+0xe8e]        # 201d <_IO_stdin_used+0x1d>
    118f:       e8 9c fe ff ff          call   1030 <puts@plt>
    1194:       48 8d 3d 9d 0e 00 00    lea    rdi,[rip+0xe9d]        # 2038 <_IO_stdin_used+0x38>
    119b:       e8 a0 fe ff ff          call   1040 <system@plt>
    11a0:       eb 0c                   jmp    11ae <main+0x59>
    11a2:       48 8d 3d 97 0e 00 00    lea    rdi,[rip+0xe97]        # 2040 <_IO_stdin_used+0x40>
    11a9:       e8 82 fe ff ff          call   1030 <puts@plt>
    11ae:       b8 00 00 00 00          mov    eax,0x0
    11b3:       c9                      leave
    11b4:       c3                      ret
    11b5:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
    11bc:       00 00 00
    11bf:       90                      nop
```

LD_PRELOAD

- In order to use external functions, such as puts() or strcmp(), a program must load an external library (such as libc) and resolve those symbols at runtime.
- There is an environment variable called LD_PRELOAD, which contains a list of libraries that will be loaded before any other library.
  - In other words, these libraries have preference over other libraries.
- If the attacker modifies the LD_PRELOAD variable, the attacker can specify his own library to load.
  - If the attacker loads a library that defines a function like puts(), then whenever puts() is called by the program, the attacker’s version of puts() will be used instead of libc’s version of puts().

License Key Program

- When I programmed this, I just hardcoded the license key 123-456-789 into the program, which an attacker could easily find by reverse engineering the program.
- However, for this exercise, let’s assume that the attacker has no idea what the license key is.
  - A program in real life might try to query an internet database that the attacker doesn’t have access to.

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ ./license
Enter your license key: helloworld
Invalid license key.

┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ ./license
Enter your license key: 123-456-789
Congrats, you get a shell!
$
```

License Key Program In Ghidra

```

undefined8 main(void)

{
  char cVar1;
  char local_88 [128];

  printf("Enter your license key: ");
  fgets(local_88,0x80,stdin);
  cVar1 = check_license_key(local_88);
  if (cVar1 == '\0') {
    puts("Invalid license key.");
  }
  else {
    puts("Congrats, you get a shell!");
    system("/bin/sh");
  }
  return 0;
}

```

License Key Program Object Dump

```
00000000000011a2 <main>:
    11a2:	55                   	push   rbp
    11a3:	48 89 e5             	mov    rbp,rsp
    11a6:	48 81 ec 90 00 00 00 	sub    rsp,0x90
    11ad:	89 bd 7c ff ff ff    	mov    DWORD PTR [rbp-0x84],edi
    11b3:	48 89 b5 70 ff ff ff 	mov    QWORD PTR [rbp-0x90],rsi
    11ba:	48 8d 3d 50 0e 00 00 	lea    rdi,[rip+0xe50]        # 2011 <_IO_stdin_used+0x11>
    11c1:	b8 00 00 00 00       	mov    eax,0x0
    11c6:	e8 85 fe ff ff       	call   1050 <printf@plt>
    11cb:	48 8b 15 7e 2e 00 00 	mov    rdx,QWORD PTR [rip+0x2e7e]        # 4050 <stdin@@GLIBC_2.2.5>
    11d2:	48 8d 45 80          	lea    rax,[rbp-0x80]
    11d6:	be 80 00 00 00       	mov    esi,0x80
    11db:	48 89 c7             	mov    rdi,rax
    11de:	e8 7d fe ff ff       	call   1060 <fgets@plt>
    11e3:	48 8d 45 80          	lea    rax,[rbp-0x80]
    11e7:	48 89 c7             	mov    rdi,rax
    11ea:	e8 76 ff ff ff       	call   1165 <check_license_key>
    11ef:	84 c0                	test   al,al
    11f1:	74 1a                	je     120d <main+0x6b>
    11f3:	48 8d 3d 30 0e 00 00 	lea    rdi,[rip+0xe30]        # 202a <_IO_stdin_used+0x2a>
    11fa:	e8 31 fe ff ff       	call   1030 <puts@plt>
    11ff:	48 8d 3d 3f 0e 00 00 	lea    rdi,[rip+0xe3f]        # 2045 <_IO_stdin_used+0x45>
    1206:	e8 35 fe ff ff       	call   1040 <system@plt>
    120b:	eb 0c                	jmp    1219 <main+0x77>
    120d:	48 8d 3d 39 0e 00 00 	lea    rdi,[rip+0xe39]        # 204d <_IO_stdin_used+0x4d>
    1214:	e8 17 fe ff ff       	call   1030 <puts@plt>
    1219:	b8 00 00 00 00       	mov    eax,0x0
    121e:	c9                   	leave
    121f:	c3                   	ret
```

check_license_key()

- We would like the call to check_license_key() to return 1 instead of 0.
  - This way, we get a shell.
- Somewhere in the assembly code, we see a CALL [addr] function, where [addr] is the address of check_license_key().
- What if we were to patch this instruction so that [addr] points to the PLT of an external function?
- If we do this, then we could use the LD_PRELOAD variable to load a malicious version of that external function.

Steps of Hooking Functions Using The LD_PRELOAD Trick

1. Find the PLT entry of a legitimate library function being used by the program.
   a. It doesn't really matter which function we choose as long as it's not super important.
2. Patch a CALL instruction to call the PLT entry of the legitimate library function that we chose.
   a. In our case, this would be the CALL instruction at 0x11ea.
   b. Instead of calling check_license_key(), we will call the library function.
3. Create a malicious library that contains a function with the same name as the legitimate library function.
4. Use LD_PRELOAD to load the malicious library.

Step 1.) Selecting PLT Entries

- We have several options for PLT entries to use.
- Theoretically we could choose any function.
- However, I am going to be using the PLT entry for __cxa_finalize(void *d) because modifying it would not affect the overall logic of the program.
- https://refspecs.linuxbase.org/LSB_3.2.0/LSB-Core-generic/LSB-Core-generic/baselib---cxa_finalize.html

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ objdump -dj .text license -M intel | grep plt
    113e:       e8 2d ff ff ff          call   1070 <__cxa_finalize@plt>
    11c6:       e8 85 fe ff ff          call   1050 <printf@plt>
    11de:       e8 7d fe ff ff          call   1060 <fgets@plt>
    11fa:       e8 31 fe ff ff          call   1030 <puts@plt>
    1206:       e8 35 fe ff ff          call   1040 <system@plt>
    1214:       e8 17 fe ff ff          call   1030 <puts@plt>
```

Step 2.) Patching The Call Instruction

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ python3
Python 3.8.6 (default, Sep 25 2020, 09:36:53)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> with open('./license', 'rb+') as f:
...     data = bytearray(b'\xe8\x81\xfe\xff\xff')
...     f.seek(0x11ea)
...     f.write(data)
...
```

Object Dump With Modified Call Instruction

```
00000000000011a2 <main>:
    11a2:       55                      push   rbp
    11a3:       48 89 e5                mov    rbp,rsp
    11a6:       48 81 ec 90 00 00 00    sub    rsp,0x90
    11ad:       89 bd 7c ff ff ff       mov    DWORD PTR [rbp-0x84],edi
    11b3:       48 89 b5 70 ff ff ff    mov    QWORD PTR [rbp-0x90],rsi
    11ba:       48 8d 3d 50 0e 00 00    lea    rdi,[rip+0xe50]        # 2011 <_IO_stdin_used+0x11>
    11c1:       b8 00 00 00 00          mov    eax,0x0
    11c6:       e8 85 fe ff ff          call   1050 <printf@plt>
    11cb:       48 8b 15 7e 2e 00 00    mov    rdx,QWORD PTR [rip+0x2e7e]        # 4050 <stdin@@GLIBC_2.2.5>
    11d2:       48 8d 45 80             lea    rax,[rbp-0x80]
    11d6:       be 80 00 00 00          mov    esi,0x80
    11db:       48 89 c7                mov    rdi,rax
    11de:       e8 7d fe ff ff          call   1060 <fgets@plt>
    11e3:       48 8d 45 80             lea    rax,[rbp-0x80]
    11e7:       48 89 c7                mov    rdi,rax
    11ea:       e8 81 fe ff ff          call   1070 <__cxa_finalize@plt>
    11ef:       84 c0                   test   al,al
    11f1:       74 1a                   je     120d <main+0x6b>
    11f3:       48 8d 3d 30 0e 00 00    lea    rdi,[rip+0xe30]        # 202a <_IO_stdin_used+0x2a>
    11fa:       e8 31 fe ff ff          call   1030 <puts@plt>
    11ff:       48 8d 3d 3f 0e 00 00    lea    rdi,[rip+0xe3f]        # 2045 <_IO_stdin_used+0x45>
    1206:       e8 35 fe ff ff          call   1040 <system@plt>
    120b:       eb 0c                   jmp    1219 <main+0x77>
    120d:       48 8d 3d 39 0e 00 00    lea    rdi,[rip+0xe39]        # 204d <_IO_stdin_used+0x4d>
    1214:       e8 17 fe ff ff          call   1030 <puts@plt>
    1219:       b8 00 00 00 00          mov    eax,0x0
    121e:       c9                      leave
    121f:       c3                      ret
```

Step 3.) Creating Malicious Library

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ cat lib.c
#include <stdio.h>
#include <stdlib.h>

int __cxa_finalize(void *d) {
    return 1;
}

┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ gcc -shared -fPIC -o lib.so lib.c
```

Step 4.) Executing Program With Malicious Library

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week10]
└─$ LD_PRELOAD=$PWD/lib.so ./license
Enter your license key: a
Congrats, you get a shell!
$ whoami
cs395
$ ping google.com
PING google.com (172.217.164.142) 56(84) bytes of data.
64 bytes from iad30s24-in-f14.1e100.net (172.217.164.142): icmp_seq=1 ttl=119 time=5.59 ms
64 bytes from iad30s24-in-f14.1e100.net (172.217.164.142): icmp_seq=2 ttl=119 time=12.0 ms
64 bytes from iad30s24-in-f14.1e100.net (172.217.164.142): icmp_seq=3 ttl=119 time=7.00 ms
```