CS 33
Introduction to Computer Organization

Qi Zhao
Week 7
11/15/2018
Agenda

• Stack Exploits
Stack Exploits

- Consider a server that hosts publicly accessible server code that performs tasks A, B, and C and an attacker that has access to the server and wants to cause trouble.
- It wouldn't be terribly impressive (or convincing) if the attacker claimed that he/she took over the server and coerced it to perform tasks A, B, or C.
- However, if the attacker could get the code to perform task D (something the was not intended to be performed by the server)...
Stack Exploits

- To do this, an attacker would want to get the server code to execute code of the attacker's choice.
- The attacker cannot directly influence the operations of the CPU or swap out the code contents in memory. It'd be pretty much game over if they could.
- However, very commonly, a user of a program will have the power to provide input to the server, which can affect on the stack.
Stack Exploits

- Consider the following function:

```c
int terrible()
{
    long a = 0x7766554433221100;
    char c[16];
    gets(c);
}
```

- When compiled, the stack looked like this:
Stack Exploits

- Note: each block is 8 bytes.
- First of all, what's the blank block for?
Stack Exploits

- Note: each block is 8 bytes.
- First of all, what's the blank block for?
  - Alignment purposes. Recall that %rsp must be 16-byte aligned. The assumption is that %rsp was aligned when calling this function.
- The code looks like:
Stack Exploits

Dump of assembler code for function blah (gcc with no options):

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400528</td>
<td>push %rbp</td>
</tr>
<tr>
<td>0x400529</td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td>0x40052c</td>
<td>sub $0x20,%rsp</td>
</tr>
<tr>
<td>0x400530</td>
<td>movabs $0x7766554433221100,%rax</td>
</tr>
<tr>
<td>0x40053a</td>
<td>mov %rax,-0x8(%rbp)</td>
</tr>
<tr>
<td>0x40053e</td>
<td>lea -0x20(%rbp),%rax</td>
</tr>
<tr>
<td>0x400542</td>
<td>mov %rax,%rdi</td>
</tr>
<tr>
<td>0x400545</td>
<td>callq 0x4003b0 <a href="mailto:gets@plt">gets@plt</a></td>
</tr>
<tr>
<td>0x40054a</td>
<td>leaveq</td>
</tr>
<tr>
<td>0x40054b</td>
<td>retq</td>
</tr>
</tbody>
</table>

End of assembler dump.
Stack Exploits

- The “gets” function takes as an argument, a character pointer. Then, it asks the user to input a character string where it then copies the string into the specified character pointer.

- For example, if you called “get(c)” and the user input provided “ABCD”, then the bytes for “ABCD” would be copied from *c to *c + 3.

- The pointer in this example is -0x20(%rbp) or -32(%rbp).
Stack Exploits

- If the user typed “jonathan” (8 characters), these 8 bytes would span from $*c$ to $*c + 7$
- If the user typed “mr_super_long_name_the_third_esquire”, this would be copied from $*c$ to $*c + 35$
- However, in the C code, we only specified a character array of size 16.
- “gets” don't care
Stack Exploits

- As a result, the user can provide a string to arbitrarily write to \( *c \) to \( *c + x \).
- This means, the user could also overwrite long \( a \), the old \( rbp \), and most worryingly, the return address.
- In this instance, what would the user have to write in order to overwrite the return address with 0x400800?
Stack Exploits

- In this instance, what would the user have to write in order to overwrite the return address with 0x400800?
- At least 40 characters (which will wipe out long a and OLD RBP), and then: 0x00, 0x08, 0x40, 0x00 … (assume little endian)
- Suddenly, once the function returns, it will attempt to execute something that it did not intend to.
Stack Exploits

• Now that there is a way to change the return address, how do you get the code to do something that it didn't intend to do.

• Say... delete a file?
Stack Exploits: Jump to Existing Library

• If your code contains: “#include <unistd.h>”, this means that during runtime, the functions included in that library exist somewhere in the code segment.

• Even if the original code didn't use them, somewhere, there exists many fun functions (or functions) such as:
  - unsigned int alarm(unsigned int)
  - int pause()
  - int chown(const char *path, uid_t owner, gid_t group)
  - int unlink(const char *path) – Deletes a file specified by path.
Stack Exploits: Jump to Existing Library

- Why write the code to implement behavior when the code already exists in the executable?
- Thus, if you know where the “unlink” function is loaded into the executable during runtime, you can change the return address to &unlink and then “return” to unlink.
- How would you know the address of unlink?
- We make the assumption that you have a copy of the code and can therefore run it and see where it will be loaded. Thus, you can use gdb to find and disassemble unlink.
Stack Exploits: Jump to Existing Library

• Can you see a problem with this approach?

• An obvious one is that you're limited to the functions that are provided by the libraries. In particular, you can only call functions provided by the linked libraries. That can be limiting.

• However, there's another, more pressing weakness.

• Hint: The signature of unlink is:
  – int unlink(const char *path);
Stack Exploits: Jump to Existing Library

- Can you see a problem with this approach?
- In order to get unlink to work, we need to specify a pointer to a string (which is the path to the file we want to delete) as an argument.
- In x86-64, we specify arguments via registers. If all we do is overwrite the return address, then we will effectively call unlink on whatever is currently in the %rdi register, which is probably not the path that we want.
Stack Exploits: Jump to Existing Library

- The fact is, this USED to work on x86. Why?
Stack Exploits: Jump to Existing Library

- The fact is, this USED to work on x86. Why?
- x86 specifies its arguments by passing them on to the stack. More specifically, when a function is entered the arguments are expected to be at (%esp + X).
- Thus, we could previous write the argument to the stack.
- Now thanks to the cutting edge 2003 x86-64 technology, this is no longer a possibility.
Stack Exploits: Injecting Code

- Knowing that we are no longer able to rely on existing code, we will somehow need to supply our own.
- Then, we can jump to our own code when we return.
- How exactly do we specify code? It's not exactly like the program is going to be asking us for code to inject. Don't be ridiculous.
Stack Exploits: Injecting Code

- Actually... it sort of it
- “gets” gives us direct access to writing into memory; whatever string of bytes is provided as an input is saved into memory/the stack.
- Sure it's not the code segment, but we can provide a string that contains the code that we want to execute,
- Then in the return address, write the address of the code that we provided.
Stack Exploits: Injecting Code

• Let's say as our vicious attack to take over the server is to execute the following instructions:

    48 89 f8     mov     %rdi,%rax
    48 83 c0 01  add     $0x1,%rax
Stack Exploits: Injecting Code

- The function gets(c) is called.
- Say %rsp = 0x7fffffffffe1d0.
- Then the memory before calling “gets” looks like this:
Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffe1d0 (and &c).

```
0x7fffffffe1d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffe1e0: 0x00400570 0x00000000 0x33221100 0x77665544
0x7fffffffe1f0: 0xfffffe200 0x00007fff 0x00400560 0x00000000
```

- First, some notes about this little endian GDB format.
Stack Exploits: Injecting Code

- `(gdb) x/40xw $rsp`
- Stack frame for “terrible” before calling gets.
- `%rsp` is `0x7fffffffe1d0` (and &c).

- `0x7fffffffe1d0`: `0x00000000 0x00000000 0x00000000 0x00000000`
- `0x7fffffffe1e0`: `0x00400570 0x00000000 0x33221100 0x77665544`
- `0x7fffffffe1f0`: `0xfffffe200 0x00007fff 0x00400560 0x00000000`

- Each column is a 4-byte word.
- `0x00400570` is the 4-byte word at address `0x7fffffffe1e0`.
- `0x00000000` is the word at address `0x7fffffffe1e4`
- `0x33221100` is the word at address `0x7fffffffe1e8`
- `0x77665544` is the word at address `0x7fffffffe1ec`
Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7ffffffffe1d0 (and &c).

- 0x7ffffffffe1d0: 0x00000000 0x00000000 0x00000000 0x00000000
- 0x7ffffffffe1e0: 0x00400570 0x00000000 0x33221100 0x77665544
- 0x7ffffffffe1f0: 0xfffffe200 0x00007fff 0x00400560 0x00000000

- Each word is presented as the actual value. In a little endian machine.
- 0x70 is the byte at address 0x7ffffffffe1e0.
- 0x05 is the byte at address 0x7ffffffffe1e1
- 0x40 is the byte at address 0x7ffffffffe1e2
- 0x00 is the byte at address 0x7ffffffffe1e3
Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffde1d0 (and &c).

- 0x7fffffffde1d0: 0x00000000 0x00000000 0x00000000 0x00000000
  0x7fffffffde1e0: 0x00400570 0x00000000 0x33221100 0x77665544
  0x7fffffffde1f0: 0xfffffe200 0x000007fff 0x00400560 0x00000000

- Remember, we want to execute:
  - 48 89 f8     mov    %rdi,%rax
  - 48 83 c0 01  add    $0x1,%rax

- Because of gets(c), we have the power to write bytes into the 'c' buffer.
Stack Exploits: Injecting Code

Dump of assembler code for function blah (gcc with no options):

0x400528 <+0>:  push   %rbp
0x400529 <+1>:  mov    %rsp,%rbp
0x40052c <+4>:  sub    $0x20,%rsp
0x400530 <+8>:  movabs $0x7766554433221100,%rax
0x40053a <+18>: mov    %rax,-0x8(%rbp)
0x40053e <+22>: lea    -0x20(%rbp),%rax
0x400542 <+26>: mov    %rax,%rdi
0x400545 <+34>: callq  0x4003b0 <gets@plt>
0x40054a <+34>: leaveq
0x40054b <+35>: retq

End of assembler dump.
Stack Exploits: Injecting Code

• We need to do two things:
  – Inject the code.
  – Overwrite the return address to point to the injected code.

• One way of doing this is to inject the code with “gets” at the beginning of “c” and then have the return address point to address of buffer “c”.

• In this example, the return address is at *c + 40 (&c is a -0x20(%rbp). Since %rbp was the first thing pushed to the stack, the return address is at 0x8(%rbp). Thus, the return address is *c + 0x20 + 0x8)
Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” before calling gets.
- %rsp is 0x7fffffffle1d0 (and &c).

- 0x7fffffffle1d0: 0x00000000 0x00000000 0x00000000 0x00000000
  0x7fffffffle1e0: 0x00400570 0x00000000 0x33221100 0x77665544
  0x7fffffffle1f0: 0xfffffe200 0x00007fff 0x00400560 0x00000000

- Red: The return address.
- Blue: The first 4-bytes of c (&c is ...e1d0). The return address is at ...e1d0 + 0x20 + 0x8 = e1f8.
Stack Exploits: Injecting Code

- Thus, if we provided the exploit string:

  48 89 f8 48 83 c0 01 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
d0 e1 ff ff ff 7f 00 00

- 48 89 f8 48 83 c0 01 is the code we want to execute.

- d0 e1 ff ff ff 7f 00 00 is the return address.

- After the “gets”, the stack will look like this:
Stack Exploits: Injecting Code

- (gdb) x/40xw $rsp
- Stack frame for “terrible” AFTER calling gets with our exploit string.
- %rsp is 0x7fffffffe1d0.

```
0x7fffffffe1d0: 0x48f88948 0x0001c083 0x00000000 0x00000000
0x7fffffffe1e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffe1f0: 0x00000000 0x00000000 0xffffe1d0 0x00007fff
```

- Now when we return, the code will “return” to 0x7ffffffffe1d0 and execute accordingly:

Dump of assembler code for function... huh? Was this here before?:

```
0x7ffffffffe1d0 <+0>: 48 89 f8 mov %rdi,%rax
0x7ffffffffe1d3 <+3>: 48 83 c0 01 add $0x1,%rax
```

End of assembler dump. Man I'm losing it.
End of Week 7
Only 3 weeks more