CS 426 Lab1

Tyler Wykoff
Task

- Understand buffer overflow
- Exploit some bugs

Environment

- Linux
- Targets: C language
- Exploits: in C or script
Outline

- Function call
- Examples
- Targets
- Useful tools
- Environment setup
Memory Layout Overview

- High Address
- Stack
  - Unused memory
  - Heap
  - Data
  - Text
- Low Address
- %esp
- %eip
```c
#include <stdio.h>
#include <string.h>

void foo(char * a, char * b)
{
    char x[8];
    char y[8];

    strcpy(x, a);
    strcpy(y, b);

    printf("x=%s y=%s\n", x, y);
}

int main(int argc, char ** argv)
{
    foo("Good", "Luck");
    return 0;
}
```
Function Call(2)  

<table>
<thead>
<tr>
<th>Frame for main</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Addr</td>
</tr>
<tr>
<td>%ebp</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frame for foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Addr</td>
</tr>
<tr>
<td>%ebp</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>
Function Call(3)  what happens?

- **Caller**
  - Push parameter(s) on stack
  - Push return addr
  - Jump to start addr of callee

- **Callee**
  - Push %ebp, %ebp ← %esp
  - Allocate space for local variables
  - ...
  - %esp ← %ebp, Pop %ebp

- **Return**
  - Pop return addr, jump to the addr
  - Restore %esp
### Function Call(4)

Assembly code:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048400</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x8048401</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x8048403</td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td>0x8048406</td>
<td>mov 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048409</td>
<td>push %eax</td>
</tr>
<tr>
<td>0x804840a</td>
<td>lea 0xffffffffffffff8(%ebp),%eax</td>
</tr>
<tr>
<td>0x804840d</td>
<td>push %eax</td>
</tr>
<tr>
<td>0x804840e</td>
<td>call 0x8048340 &lt;strcpy&gt;</td>
</tr>
<tr>
<td>0x8048413</td>
<td>add $0x8,%esp</td>
</tr>
<tr>
<td>0x8048416</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048419</td>
<td>push %eax</td>
</tr>
<tr>
<td>0x804841a</td>
<td>lea 0xffffffffffffff0(%ebp),%eax</td>
</tr>
<tr>
<td>0x804841d</td>
<td>push %eax</td>
</tr>
<tr>
<td>0x804841e</td>
<td>call 0x8048340 &lt;strcpy&gt;</td>
</tr>
<tr>
<td>0x8048423</td>
<td>add $0x8,%esp</td>
</tr>
<tr>
<td>0x8048426</td>
<td>lea 0xffffffffffffff0(%ebp),%eax</td>
</tr>
<tr>
<td>0x8048429</td>
<td>push %eax</td>
</tr>
<tr>
<td>0x804842a</td>
<td>lea 0xffffffffffffff8(%ebp),%eax</td>
</tr>
<tr>
<td>0x804842d</td>
<td>push %eax</td>
</tr>
<tr>
<td>0x804842e</td>
<td>push $0x80484c0</td>
</tr>
<tr>
<td>0x8048433</td>
<td>call 0x8048330 &lt;printf&gt;</td>
</tr>
<tr>
<td>0x8048438</td>
<td>add $0xc,%esp</td>
</tr>
<tr>
<td>0x804843b</td>
<td>leave</td>
</tr>
<tr>
<td>0x804843c</td>
<td>ret</td>
</tr>
</tbody>
</table>
Buffer Overflow

- C doesn’t check boundaries!

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

void foo(char * a, char * b)
{
    char x[8];
    char y[8];

    strcpy(x, a);
    strcpy(y, b);

    printf("x=%s y=%s\n", x, y);
}

int main(int argc, char ** argv)
{
    foo("Good", "Luck_____Bad");
    return 0;
}
```
Example 2

```c
#include <stdio.h>

void foo(int a)
{
    char x;
    unsigned int * ret;

    ret = (unsigned int *)(&x + 5);
    *ret += 10;
}

int main(int argc, char ** argv)
{
    int x;

    x = 10;
    printf("x=%d\n", x);

    foo(23);
    x = 20;
    printf("x=%d\n", x);

    return 0;
}
```
Example 3

```c
#include <stdio.h>

void foo(char * arg)
{
    char buf[56];
    strcpy(buf, arg);
}

int main(int argc, char ** argv)
{
    if (argc < 2)
    {
        return 0;
        foo(argv[1]);
    return 0;
    }
```
Example 3

- Load some code to the buffer
- Modify the return addr to execute our code

```
Frame for main

| argv    |
| argc    |
| Return Addr |
| %ebp    |
| arg     |
| Return Addr |
| %ebp    |
| buf     |
| ...     |
```

Frame for foo
Example 3 preparation

- Need to know
  - Address of the buf
  - Address of the return addr
  - Distance between buf and return addr
  - Length of the buffer

- Insert code in the buffer
  - The code to launch a shell
  - Reading: Smashing The Stack For Fun And Profit by Aleph One
  - Provided in exploits/shellcode.h
Example 3: 

- Insert shellcode at the beginning of the buffer
- Put the addr of buf somewhere in the buffer
- Execute the target program
Return to lib-c attack

- Defense against buffer overflow
  - Stack data are not executable
  - Attack cannot provide code in the stack

- Attacker can still modify the return address
  - Return to some system library
  - For example, `system(const char * string)`
Target1

- A program to check the correctness of the password

- Goal: Make the program accept your `password`

- Exploit1.sh: a shell script

- Credit: 20%
Target2

- A program to print a coupon
- Goal: to print a lot coupons!
- Exploit2.c: c program
- Credit: 10% will be given if you can print two coupons (only launching the target program once) 20% will be given if you can print more than twenty coupons
A program to check if a password is strong or weak

Goal: to start a shell, by using a buffer overflow and shellcode

Exploit3.c: C program

Credit: 30%
Target4

- A program to check if a password is strong or weak
- Goal: to start a shell, using a return-to-libc attack
- Exploit4.c: c program
- Credit: 30%
Useful tools

○ GDB
  - Start: gdb ./example1
  - Source: list linenum
  - Assembly: disassemble func
  - Step: step/steppi
  - Memory: x addr
  - Variables/registers: print var/reg

○ Will give GDB tutorial in PSO this week!
Warming up

- Understand what is going on
  - The assembly code
  - The memory (stack)
  - The registers
  - The variables
  - What does LEAVE/RET do
  - ...

Environment Setup

- The OS is running in a virtual machine
- Login
  - Connect to the VM
    - `ssh cs426vm1.cs.purdue.edu`
- Tools available
  - `gcc, make, gdb, vim, emacs`
Submission

- Deadline is 11:59pm Oct 8th (two weeks from Friday)
- Just leave your solution files (including answers to questions) in ./exploits of your home directory
- .c files should be compiled and ready to run without any arguments
Team Details

- Email me (twykoff@purdue.edu):
  - Who you are working with (both of your names)
  - What your requested login name is
- If you don't yet have a partner
  - Email me and I'll pair you up
Other stuff

- Exploits codes are short
- Several ways to exploit
- Start early
- Codes from others may *not* work
- Backup files often (outside the virtual machine)
- Make your exploits stable
Other stuff

- Don’t use the machines for other purposes
- Updates may be available through mailing list
- Have fun 😊
Questions?