CSE 127 Discussion 2

Patrick Liu
Some slides adapted from Ariana Mirian and Deian Stefan
This session is being recorded
Logistics

- PA1 due today
- PA2 released, due January 23
- Agenda
  - PA2
  - Stack Smashing!
  - Open Office Hours (if time)
PA2

- Due Saturday, January 23 at midnight
- Early due date: Wednesday, January 20
- Focused on exploit development
  - Read Aleph One!
PA2 Setup

- Download pa2box from the writeup
- When you ssh in, perform the following steps
  - Run `make generate` in the `targets` directory
  - Run `make` to compile the targets
  - Run `sudo make setuid` to give all your binaries root access
  - Programs will be randomized based on your PID
- You will be modifying ONLY the files named `sploit[1-4].c` DO NOT modify the `target[1-4].c` files or any other file
Shellcode

```c
#include <stdio.h>

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

```
http://phrack.org/issues/49/14. html#article

static char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\xb0"
"\x89\xf3\x8d\xe4\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xf2\xff\xf2\xff/bin/sh"
```
Setuid

- This bit allows for escalation of privilege
  - Since the target binaries are owned by root, this bit allows them to run with root privileges
  - Now, anything the process does has root privileges, including the shell, if you can get it(!)

```
student@CSE127:~/hw2/sploits$ ./sploit1
# whoami
root
# ```
Process Memory Layout

- Text
  - Executable code
- Data
  - .data, .rodata, .bss
- Stack
- Heap
Registers to know

- %esp, or the Stack Pointer
  - Designates the top of the stack
  - Grows from high to low memory addresses

- %ebp, or the Frame Pointer/Base Pointer
  - Points to middle of stack frame (to the saved base pointer)
  - Doesn't move as function calls are made
  - Why does it exist? (why can't we just offset from stack pointer?)
Registers to know

- %eip, or the Instruction Pointer
  - Holds the address of the next instruction to be executed

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>argv</td>
</tr>
<tr>
<td></td>
<td>argc</td>
</tr>
<tr>
<td></td>
<td>saved ret</td>
</tr>
<tr>
<td></td>
<td>saved ebp</td>
</tr>
<tr>
<td>%esp</td>
<td>nice[0-3]</td>
</tr>
<tr>
<td></td>
<td>name[0-3]</td>
</tr>
<tr>
<td></td>
<td>nice[4-7]</td>
</tr>
<tr>
<td></td>
<td>name[4-7]</td>
</tr>
</tbody>
</table>
Function calls

```c
int foobar(int a, int b)
{
    int x = 1;
    int buf[12];
    buf[4] = 10;
    return x;
}

int main()
{
    return foobar(77, 88);
}
```

```assembly
; foobar(int, int):
    pushl %ebp
    movl %esp, %ebp
    subl $64, %esp
    movl $1, -4(%ebp)
    movl $10, -36(%ebp)
    movl -4(%ebp), %eax
    leave
    ret

; main:
    pushl %ebp
    movl %esp, %ebp
    pushl $88
    pushl $77
    call foobar(int, int)
    addl $8, %esp
    nop
    leave
    ret
```
Basic Buffer Overflows

- Let's get hacking!
- What's the problem with this program?

```
int foo(int a){
    char buf[8];
    gets(buf);
    return 0
}
```
Basic Buffer Overflows

- An input larger than 8 characters will begin to overwrite the stack!
- Suppose we simply put a bunch of As
  - What happens when the program returns?
  - Program will attempt to pop 0x41414141 into eip
    - Is the data at 0x41414141 a valid instruction?
  - Crash!
Basic Buffer Overflows

- What if we wanted to do something other than crash?
  - Can we make the program execute instructions of our choice?
Basic Buffer Overflows

- What if we wanted to do something other than crash?
  - Can we make the program execute instructions of our choice?
- Let's make some NOPs, and make our program jump to them
  - Now, our input is 12 bytes of 0x90, followed by 0xcdabfeff (why?)
  - What will happen when this program returns?
Basic Buffer Overflows

- What if we can't overwrite the instruction pointer?
  - What's the first thing we can overwrite?

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>return address</td>
</tr>
<tr>
<td>saved ebp</td>
</tr>
<tr>
<td>buf[4-7]</td>
</tr>
<tr>
<td>buf[0-3]</td>
</tr>
</tbody>
</table>
Basic Buffer Overflows

- What if we can't overwrite the instruction pointer?
  - What's the first thing we can overwrite?
  - Can you still control the instruction pointer via the saved base pointer?
Basic Buffer Overflows

- What happens when a function returns?

<table>
<thead>
<tr>
<th>return address</th>
<th>saved ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

ebp $\Rightarrow$ saved ebp

buf[4-7]

esp $\Rightarrow$ buf[0-3]
Basic Buffer Overflows

- What happens when a function returns?
  - Sets esp = ebp

```
return address
saved ebp
a
return address
saved ebp
buf[4-7]
buf[0-3]
```

esp $\Rightarrow$ ebp $\Rightarrow$
Basic Buffer Overflows

- What happens when a function returns?
  - Sets esp = ebp
  - Pop ebp
Basic Buffer Overflows

- What happens when a function returns?
  - Sets esp = ebp
  - Pop ebp
  - Pop eip
Basic Buffer Overflows

- What if we control ebp?
  - Let's try to return again
Basic Buffer Overflows

- What if we control ebp?
  - Let's try to return again
  - Setting esp = ebp functions normally...

```
  return address
  saved ebp
    a
  return address
  saved ebp
  buf[4-7]
  buf[0-3]
  esp ⇔ ebp ⇔
```
Basic Buffer Overflows

- What if we control ebp?
  - Let's try to return again
  - Setting esp = ebp functions normally…
  - What happens when we try to pop ebp?
    - We can make it point anywhere!

```
return address
saved ebp
  a
esp ↦
return address
saved ebp
buf[4-7]

ebp ↞
buf[0-3]
```
Basic Buffer Overflows

- What if we control ebp?
  - Let's try to return again
  - Setting esp = ebp functions normally…
  - What happens when we try to pop ebp?
    - We can make it point anywhere!
  - Popping eip also functions normally…
Basic Buffer Overflows

- What happens when the next function tries to return?
  - Set esp = ebp...

```
return address
(saved ebp
  a
return address
(saved ebp
(buf[4-7]
(buf[0-3]}

esp ⇔ ebp ⇔
Basic Buffer Overflows

- What happens when the next function tries to return?
  - Set esp = ebp...
  - Pop ebp...
Basic Buffer Overflows

- What happens when the next function tries to return?
  - Set esp = ebp…
  - Pop ebp…
  - And finally pop eip…
- buf[0-3] became ebp, and buf[4-7] went into eip!
Integer Representations

- C recognizes two types of integers: signed and unsigned.
- Unsigned integers are easy: simply interpret the bits as a binary number
  - $0xA5 = \text{10100101} = 165$
- Signed integers work based on two's complement
  - First bit is the sign bit. If sign bit is set, interpret as negative of two's complement
  - Two's complement rule: Flip the bits and add 1
  - $0xA5 = \text{10100101}$
    - After two's complement, this is $01011011 = 91$
    - Therefore, this represents $-91$ (for 8 bit numbers)
Integer Exercise

- Write out all the possible 4-bit values, and interpret them as signed and unsigned integers
<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 = 0</td>
<td>0000 = 0</td>
</tr>
<tr>
<td>0001 = 1</td>
<td>0001 = 1</td>
</tr>
<tr>
<td>0010 = 2</td>
<td>0010 = 2</td>
</tr>
<tr>
<td>0011 = 3</td>
<td>0011 = 3</td>
</tr>
<tr>
<td>0100 = 4</td>
<td>0100 = 4</td>
</tr>
<tr>
<td>0101 = 5</td>
<td>0101 = 5</td>
</tr>
<tr>
<td>0110 = 6</td>
<td>0110 = 6</td>
</tr>
<tr>
<td>0111 = 7</td>
<td>0111 = 7</td>
</tr>
<tr>
<td>1000 = 8</td>
<td>1000 = -8</td>
</tr>
<tr>
<td>1001 = 9</td>
<td>1001 = -7</td>
</tr>
<tr>
<td>1010 = 10</td>
<td>1010 = -6</td>
</tr>
<tr>
<td>1011 = 11</td>
<td>1011 = -5</td>
</tr>
<tr>
<td>1100 = 12</td>
<td>1100 = -4</td>
</tr>
<tr>
<td>1101 = 13</td>
<td>1101 = -3</td>
</tr>
<tr>
<td>1110 = 14</td>
<td>1110 = -2</td>
</tr>
<tr>
<td>1111 = 15</td>
<td>1111 = -1</td>
</tr>
</tbody>
</table>
Integer Representation Vulnerabilities

- What if we have this structure?
  - What assumption is made inside the if statement?
  - Can we violate this assumption?

```c
int input = //get input from user somehow
if(input < 300):
    unsigned int input2 = (unsigned int) input
    //do stuff with input2
```
Integer Representation Vulnerabilities

- What happens if we pass -10?
- -10 < 300, so we go into the if statement
- However, inside the integer is used as an unsigned int!
  - Value will be interpreted as an unsigned integer
  - Within the C standard library, size_t is an unsigned integer type!

```c
void *memcpy(void *dest, const void *src, size_t n);
```
Fun with Multiplication

- What happens when a number gets larger (or smaller) than a 32-bit integer can hold?
  - Overflow! (or underflow)
- This is particularly easy to make happen with multiplication
  - Consider the 8 bit number 10101001 = -87
  - 10101001 * 2 = 10101001 << 1 = 01010010 = 82
- What happens when there's a multiplication inside a bounds check…?
Tips and Tricks

- Read Aleph One before you start
  - This gives a blueprint for sploit1 and sploit2
- Use memcpy and loops in sploit1-4.c
  - Don't try to hardcode a 600 byte buffer
- Utilize a NOP sled
  - Prepend 0x90 (NOP instruction) to your shellcode to create a NOP sled
  - Jumping anywhere in the NOP sled will execute your shellcode
- Avoid 0x0 (NULL)
  - This will null terminate your buffer
Heap Vulnerabilities

- Dynamically allocated memory in program
- Programmer is responsible for proper use
  - Not true of stack
  - Improper use of malloc() and free() can get user into trouble
- Generally, heap is a doubly-linked list
- Double free and use after free are common vulnerabilities
Heap Structure

- Heaps are doubly linked lists
- Each block has a forward pointer and a back pointer
  - There are also "global" first and last pointers that point to the first and last block of the heap
  - Note the circular structure
Heap Exploits

```
1 a = malloc(16);  // @xa04010
2 b = malloc(16);  // @xa04030
3 c = malloc(16);  // @xa04050
4
5 free(a);          // To bypass "double free or corruption (fasttop)" check
6 free(b);          // Double Free !!
7
8 d = malloc(16);   // @xa04010
9 e = malloc(16);   // @xa04030
10 f = malloc(16);  // @xa04010  - Same as 'd'
```

1. 'a' freed.
   head -> a -> tail

2. 'b' freed.
   head -> b -> a -> tail

3. 'a' freed again.
   head -> a -> b -> a -> tail

[https://heap-exploitation.dhavalkapil.com/attacks/double_free](https://heap-exploitation.dhavalkapil.com/attacks/double_free)
Heap Exploits

```c
1. a = malloc(10); // 0xa04010
2. b = malloc(10); // 0xa04030
3. c = malloc(10); // 0xa04050

free(a);
free(b); // To bypass "double free or corruption (fasttop)" check
free(a); // Double Free !!

4. d = malloc(10); // 0xa04010
5. e = malloc(10); // 0xa04030
6. f = malloc(10); // 0xa04010 - Same as 'd' !
```

https://heap-exploitation.dhavalkapil.com/attacks/double_free
Tips for Sploit4

- Read tmalloc.c carefully
- Draw pictures!
- Can you determine what happens to the heap when malloc() and free() happen?
Open Office Hours