

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

```
shellcode.c
```

```
-----  
#include <stdio.h>  
  
void main() {  
    char *name[2];  
  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```

```
0x8000130 <main>:    pushl %ebp  
0x8000131 <main+1>:  movl %esp,%ebp  
0x8000133 <main+3>:  subl $0x8,%esp  
0x8000136 <main+6>:  movl $0x80027b8,0xffffffff8(%ebp)  
0x800013d <main+13>: movl $0x0,0xfffffff8(%ebp)  
0x8000144 <main+20>: pushl $0x0  
0x8000146 <main+22>: leal 0xffffffff8(%ebp),%eax  
0x8000149 <main+25>: pushl %eax  
0x800014a <main+26>: movl 0xffffffff8(%ebp),%eax  
0x800014d <main+29>: pushl %eax  
0x800014e <main+30>: call 0x80002bc <__execve>  
0x8000153 <main+35>: addl $0xc,%esp  
0x8000156 <main+38>: movl %ebp,%esp  
0x8000158 <main+40>: popl %ebp  
0x8000159 <main+41>: ret
```

```
-----  
static char shellcode[] =  
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"  
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

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

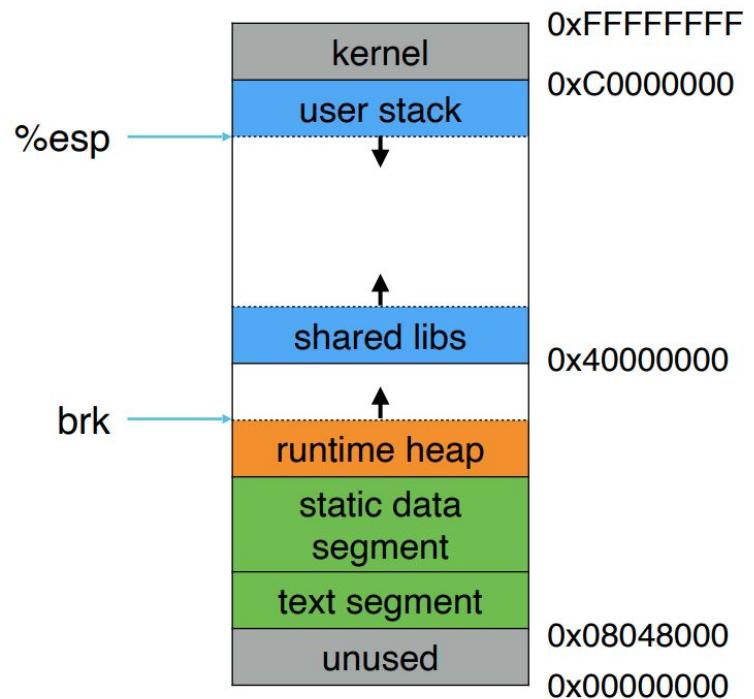
Setuid

- 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
#
```

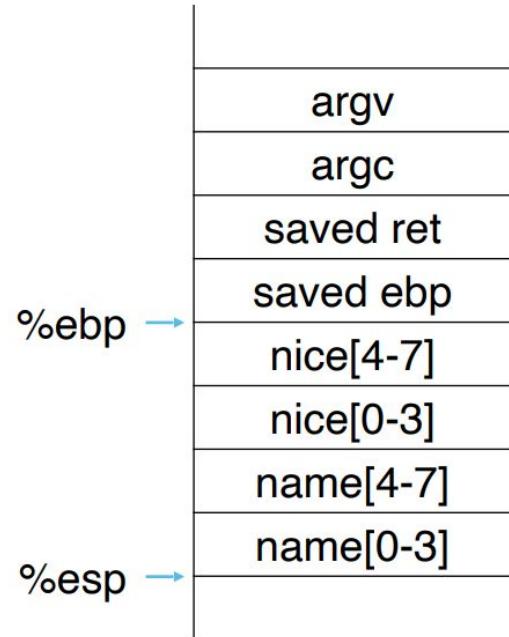
Process Memory Layout

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



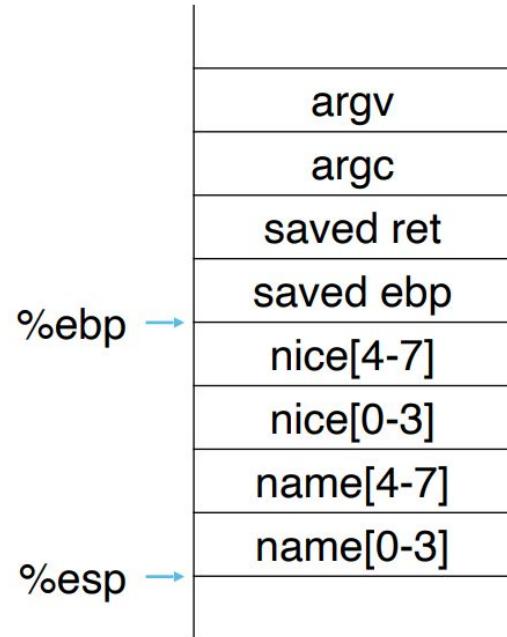
Registers to know

- %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



Function calls

```
1 int foobar(int a, int b)
2 {
3     int x = 1;
4     int buf[12];
5     buf[4] = 10;
6     return x;
7 }
8
9 int main()
10 {
11     return foobar(77, 88);
12 }
```



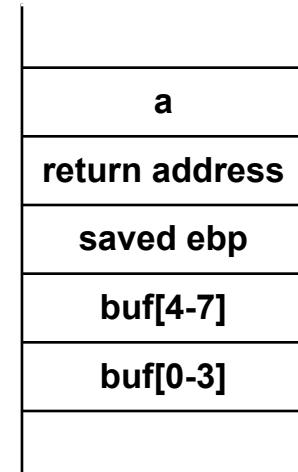
```
1 foobar(int, int):
2     pushl %ebp
3     movl %esp, %ebp
4     subl $64, %esp
5     movl $1, -4(%ebp)
6     movl $10, -36(%ebp)
7     movl -4(%ebp), %eax
8     leave
9     ret
10
11 main:
12     pushl %ebp
13     movl %esp, %ebp
14     pushl $88
15     pushl $77
16     call foobar(int, int)
17     addl $8, %esp
18     nop
19     leave
20     ret
```

<https://godbolt.org/z/xMejhv>

Basic Buffer Overflows

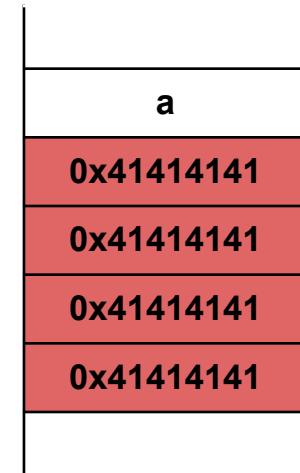
- Let's get hacking!
- What's the problem with this program?

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



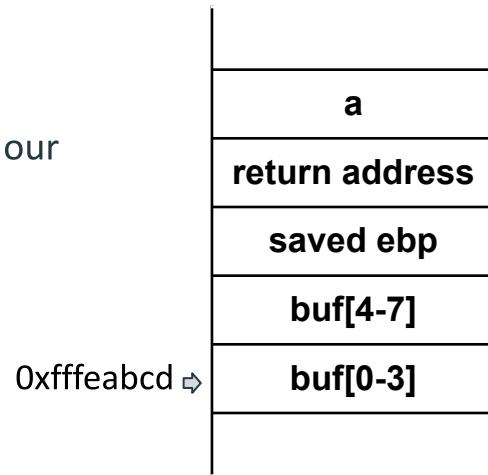
Basic Buffer Overflows

- 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!



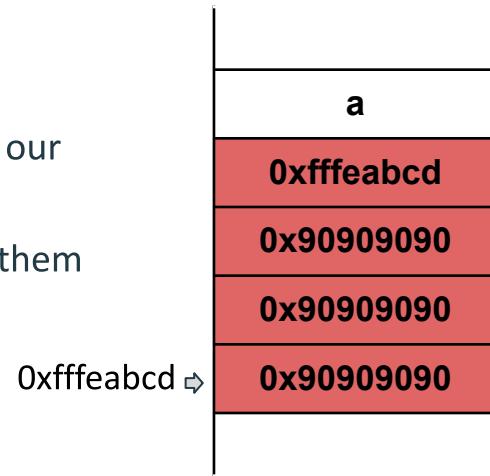
Basic Buffer Overflows

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



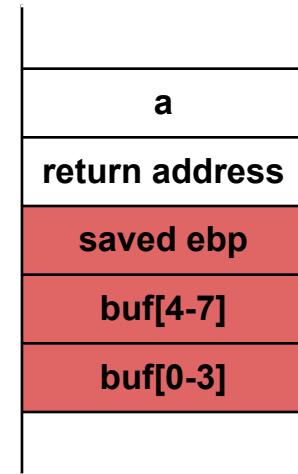
Basic Buffer Overflows

- 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?



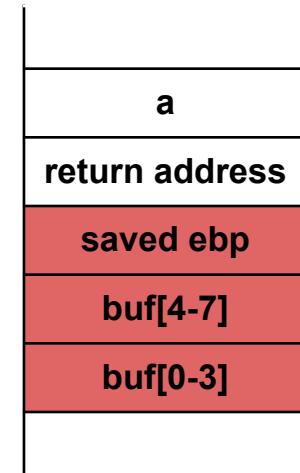
Basic Buffer Overflows

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



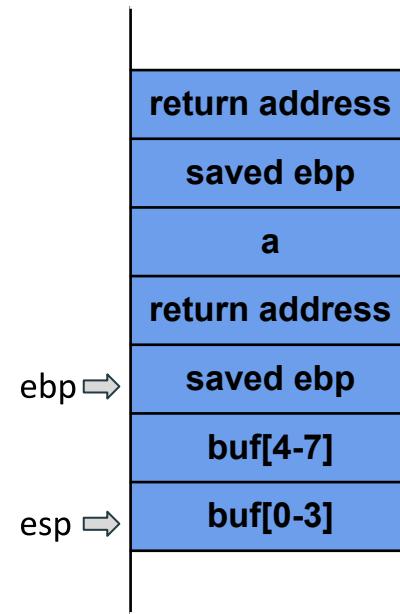
Basic Buffer Overflows

- 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?



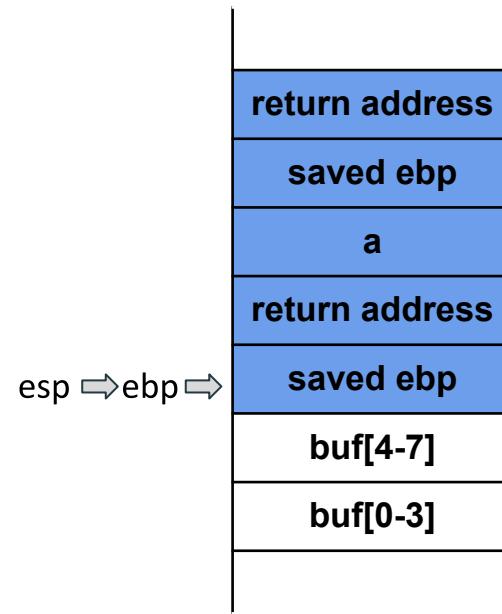
Basic Buffer Overflows

- What happens when a function returns?



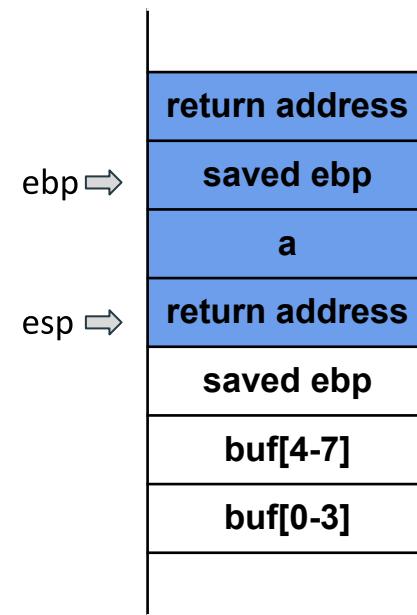
Basic Buffer Overflows

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



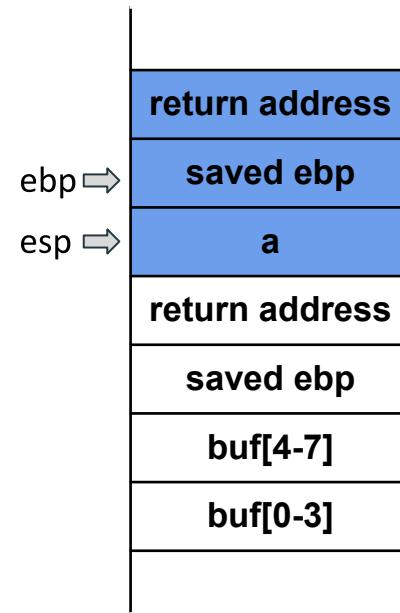
Basic Buffer Overflows

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



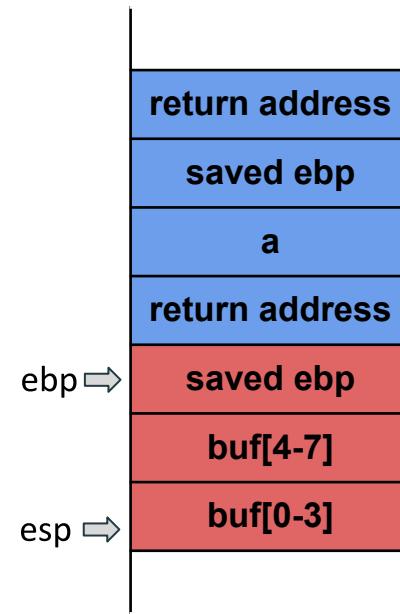
Basic Buffer Overflows

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



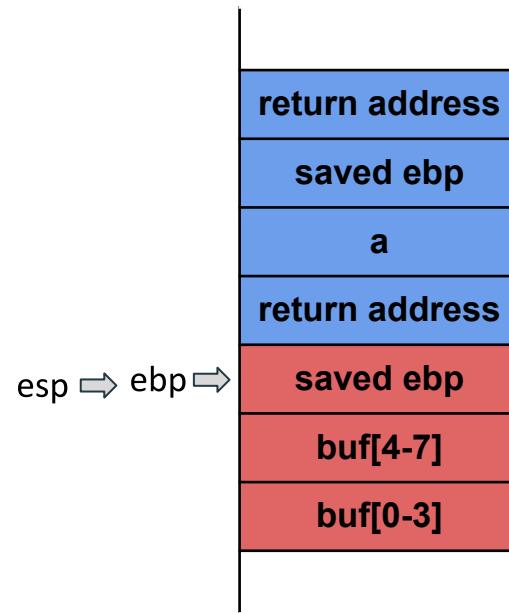
Basic Buffer Overflows

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



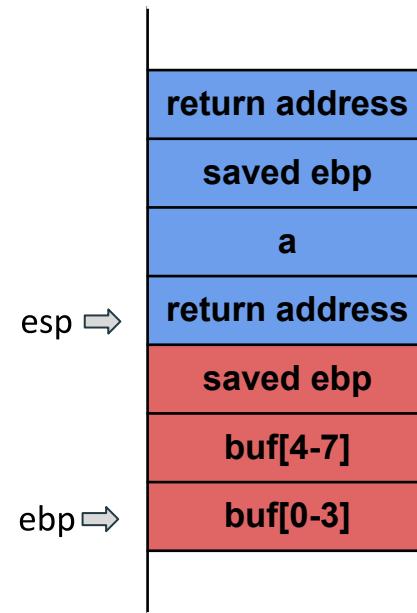
Basic Buffer Overflows

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



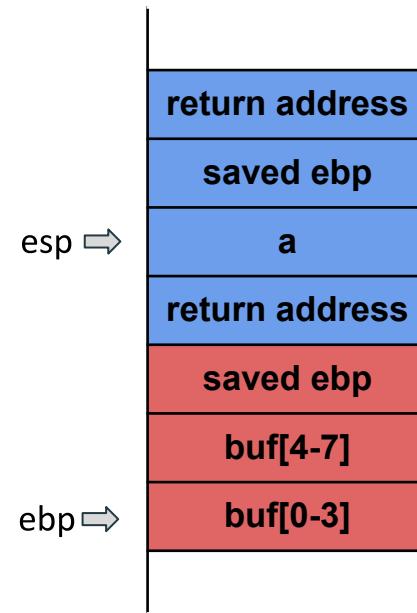
Basic Buffer Overflows

- 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!



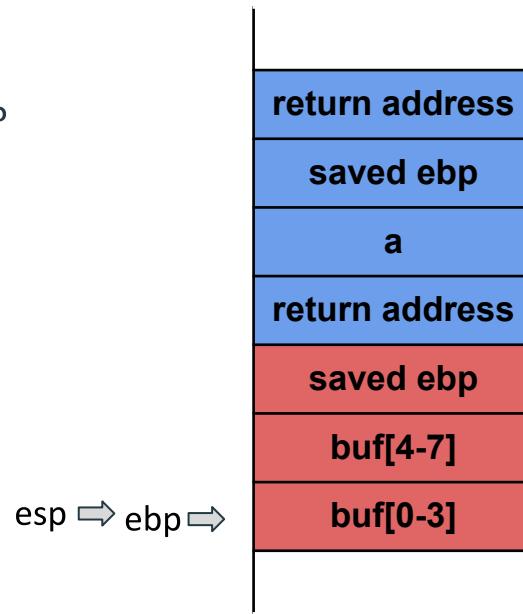
Basic Buffer Overflows

- 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...



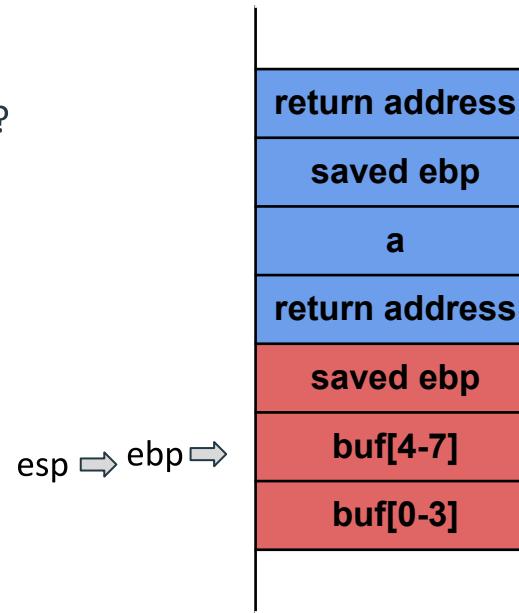
Basic Buffer Overflows

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



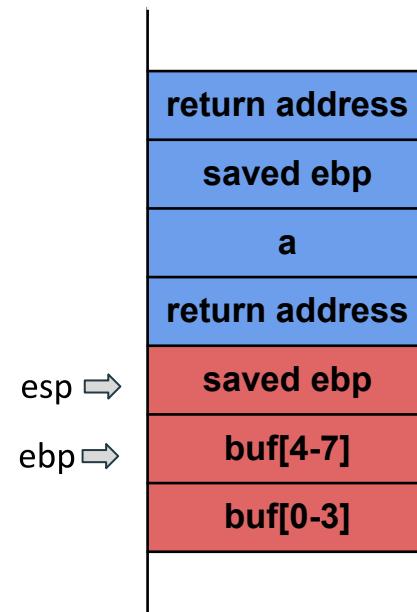
Basic Buffer Overflows

- 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
 - $0xA5 = 10100101 = 165$
- 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
 - $0xA5 = 10100101$
 - 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

Unsigned:

0000 = 0

0001 = 1

0010 = 2

0011 = 3

0100 = 4

0101 = 5

0110 = 6

0111 = 7

1000 = 8

1001 = 9

1010 = 10

1011 = 11

1100 = 12

1101 = 13

1110 = 14

1111 = 15

Signed:

0000 = 0

0001 = 1

0010 = 2

0011 = 3

0100 = 4

0101 = 5

0110 = 6

0111 = 7

1000 = -8

1001 = -7

1010 = -6

1011 = -5

1100 = -4

1101 = -3

1110 = -2

1111 = -1

Integer Representation Vulnerabilities

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

```
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!

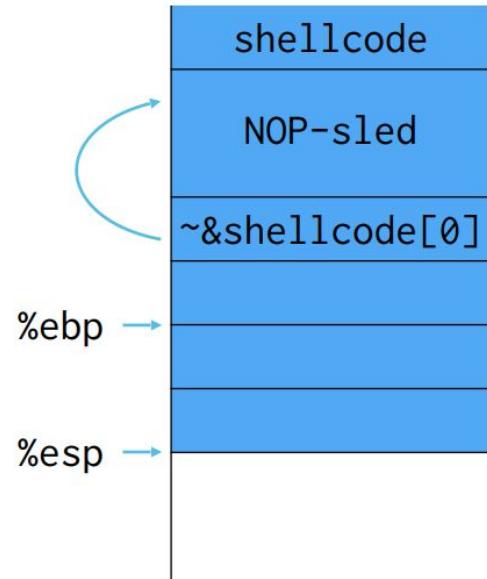
```
void *memcpy(void *dest, const void *src, size_t n);
```

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 $10101001 = -87$
 - $10101001 * 2 = 10101001 \ll 1 = 01010010 = 82$
- 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 exploit1 and exploit2
- Use memcpy and loops in exploit1-4.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(NUL)
 - This will null terminate your buffer

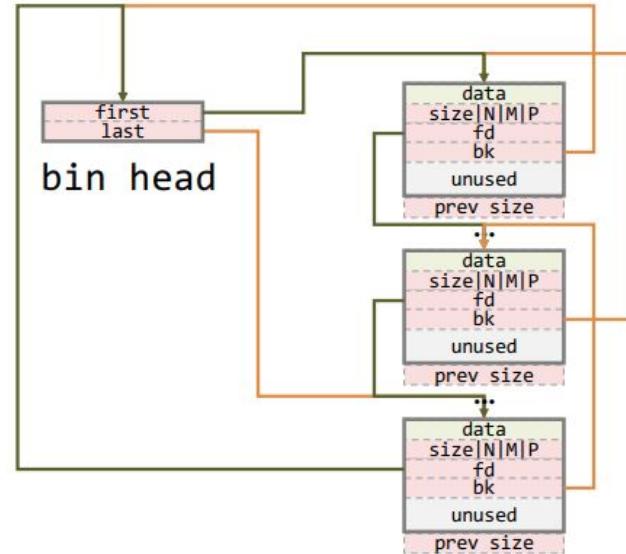


Heap Vulnerabilities

- 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(10);      // 0xa04010
2 b = malloc(10);      // 0xa04030
3 c = malloc(10);      // 0xa04050
4
5 free(a);
6 free(b); // To bypass "double free or corruption (fasttop)" check
7 free(a); // Double Free !!
8
9 d = malloc(10);      // 0xa04010
10 e = malloc(10);     // 0xa04030
11 f = malloc(10);     // 0xa04010 - Same as 'd' !
```



https://heap-exploitation.dhavalkapil.com/attacks/double_free

Heap Exploits

```
1 a = malloc(10);      // 0xa04010
2 b = malloc(10);      // 0xa04030
3 c = malloc(10);      // 0xa04050
4
5 free(a);
6 free(b); // To bypass "double free or corruption (fasttop)" check
7 free(a); // Double Free !!
8
9 d = malloc(10);      // 0xa04010
10 e = malloc(10);     // 0xa04030
11 f = malloc(10);     // 0xa04010 - 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

Heap Exploits

```
1 a = malloc(10);      // 0xa04010
2 b = malloc(10);      // 0xa04030
3 c = malloc(10);      // 0xa04050
4
5 free(a);
6 free(b); // To bypass "double free or corruption (fasttop)" check
7 free(a); // Double Free !!
8
9 d = malloc(10);      // 0xa04010
10 e = malloc(10);     // 0xa04030
11 f = malloc(10);     // 0xa04010 - 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

4. 'malloc' request for 'd'.

head -> b -> a -> tail ['a' is returned]

5. 'malloc' request for 'e'.

head -> a -> tail ['b' is returned]

6. 'malloc' request for 'f'.

head -> tail ['a' is returned]

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