Week 3 Discussion

PA2: Buffer Overflows
Agenda

- Go through each target
- Review concepts
- Demo
Reminder

- Due: Tuesday, January 31 at 11:59pm
- Make sure your cookie is correct
- the score on Gradescope is not your score for the assignment
Memory Layout

- Stack
- Heap
- Data segment
  - Binary instructions
- Text segment
- Shared libraries
- Runtime heap
- Static data segment
- Unused

```
KERNEL: 0xFFFFFFFF
USER STACK: 0xC0000000
SHARED LIBS: 0x40000000
RUNTIME HEAP: 0x8048000
STATIC DATA SEGMENT: 0x0
unused: 0x0
```
%eip is a register pointing to the instruction that CPU will execute in next cycle

saved %eip (aka saved return) is stored on stack when a function call is made. It has the address of where to resume execution in the caller function

When a function returns, the saved %eip value will be popped into the register %eip → control will transfer to where saved %eip points to
(after curr function return, we jump to address in saved eip)

return address == saved return address == saved %eip == %ebp+4
Stack Layout: esp

%esp aka stack pointer points to top of stack.

Stack grows down (from high to low addresses), so esp points to the bottom of our picture.

examine gdb commands (x10x address) examines memory low to high.

So to get the current stack dump, examine the %esp register!

High Memory

- arguments
- saved %eip
- saved %ebp
- Callee saved registers
- local variables

Low Memory
Target 0: Overwriting a variable on the stack

High Memory

saved %eip

saved %ebp

Irrelevant

grade[5]

name[10]

Low Memory

int _main(int argc, char *argv[])
{
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("Hi %s! Your grade is %s.\n", name, grade);
    exit(0);
}

Be careful, when name is printed, no extra garbage content should be printed. Think about when printf stops prints given a pointer.
int _main(int argc, char *argv[]) {
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("Hi %s! Your grade is %s.\n", name, grade);
    exit(0);
}
Target 1: Overwriting the return address

• What is the distance between `input` and `saved %eip`?
• How do you fill the gap in between?
• Where do you want to redirect control to?
Target 1: Overwriting the return address

```c
void print_bad_grade(void)
{
    puts("Your grade is nil.");
    exit(0);
}
void print_good_grade(void)
{
    puts("Your grade is perfect.");
    exit(1);
}
void vulnerable()
{
    char input[4];
    gets(input);
}
int _main()
{
    vulnerable();
    print_bad_grade();
    return 0;
}
```
Target 2: Redirecting control to shellcode

void vulnerable(char *arg) {
    char buf[100];
    strcpy(buf, arg);
}

int _main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}

Where do you want to transfer control to?
void vulnerable(char *arg) {
    char buf[100];
    strcpy(buf, arg);
}

int _main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}

Target 2: Redirecting control to shellcode

Where do you want to transfer control to?

shellcode
Demo
Target 3 (Hint): Overwriting the return address indirectly

```c
void vulnerable(char *arg)
{
    int *p;
    int a;
    char buf[2048];
    
    `STRNCPY`(buf, arg, sizeof(buf) + 8);
    
    *p = a;
}
```

- `strncpy(buf, arg, sizeof(buf) + 8)`; What does this mean
  - You can only overflow `p` and `a`.
- What does `*p = a` mean?
- What happens if we overflow `p` to be an address?
- Think about how we can take advantage of this
#include <stdio.h>
#include <string.h>

target vulnerable(char *arg)
{
    int *p;
    int a;
    char buf[2048];
    strncpy(buf, arg, sizeof(buf) + 8);
    *p = a;
}

target _main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```c
#include <stdio.h>
#include <string.h>

void vulnerable(char *arg) {
    int *p;
    int a;
    char buf[2048];
    strncpy(buf, arg, sizeof(buf) + 8);
    *p = a;
}

int _main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

What does this do?

- The above code snippet demonstrates a function `vulnerable` that takes a string argument and tries to directly assign a value to a location in memory without checking the validity of the pointer.

- This can lead to a buffer overflow if the input string is longer than the memory allocated for it, potentially causing memory corruption.

- The `strncpy` function is used to copy the argument string into a buffer, but it is not safe to assume that the source string will not exceed the buffer size.

- The `*_main` function checks if the correct number of command-line arguments are provided. If not, it prints an error message and returns 1. Otherwise, it calls the `vulnerable` function with the second argument.

- This example highlights the importance of validating inputs and ensuring proper memory management to prevent vulnerabilities such as buffer overflows.
Target 3 (Hint): Overwriting the return address indirectly

Pointer subterfuge

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf, str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}
```

![Pointer subterfuge diagram](image-url)
Target 3 (Hint): Overwriting the return address indirectly

Pointer subterfuge

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf,str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}
```
Target 3 (Hint): Overwriting the return address indirectly

Pointer subterfuge

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

void foo() {
    printf("hello all!!\n");
    exit(0);
}

int i = 42;

void func(char *str) {
    int *ptr = &i;
    int val = 44;
    char buf[4];
    strcpy(buf,str);
    *ptr = val;
}

int main(int argc, char**argv) {
    func(argv[1]);
    return 0;
}
```
Target 4 (Hint): Beyond strings

- count: 32-bit unsigned integer
- What happens when I overflow an unsigned int?

Demo

```c
unsigned int maxint = 0xFFFFFFFF;
unsigned int count = 0x40000000;

// count * 4 = 0x100000000 (overflown to 0)
printf("count * 4 = %d \n", count * 4);

// count * 4 + 50 = 0x100000032 (overflown to 0x32 = 50)
printf("count * 4 + 50 = %d \n", count * 4 + 50);

return 0;
```
Target 4 (Hint): Beyond strings

- Want **count** to be bigger than **size of buf**
- This way you copy more than what buf can hold
- Figure out how big should buf be.

**Demo**

```c
unsigned int maxint = 0xFFFFFFFF;
unsigned int count = 0x40000000;

// count * 4 = 0x100000000 (overflown to 0)
printf("count * 4 = %d \n", count * 4);

// count * 4 + 50 = 0x100000032 (overflown to 0x32 = 50)
printf("count * 4 + 50 = %d \n", count * 4 + 50);

return 0;
```

```c
void read_elements(FILE *f, int *buf, unsigned int count)
{
    unsigned int i;
    for (i=0; i < count; i++) {
        if (fread(&buf[i], sizeof(unsigned int), 1, f) < 1) {
            break;
        }
    }
}

void read_file(char *name)
{
    FILE *f = fopen(name, "rb");
    ..... 

    unsigned int count;
    fread(&count, sizeof(unsigned int), 1, f);

    unsigned int *buf = alloca(count * sizeof(unsigned int));
    ..... 

    read_elements(f, buf, count);
}
```
Target 5 (Hint): Bypassing DEP

- Remember - Compiled with DEP enabled
- W^X (Write XOR eXecute)!
- You cannot put shellcode in buffer and point to it from return address.

- What can you do?
- "Return-to-libc" style of attack
- What does system("/bin/sh"); do?
- Review lecture 6, detailed walkthrough

```c
// Compiled with DEP enabled.
void greetings(void)
{
    system("echo Hello World");
}

void vulnerable(char *arg)
{
    char buf[10];
    strcpy(buf, arg);
}

int _main(int argc, char *argv[])
{
    ....
    setuid(0);
    vulnerable(argv[1]);
    greetings();
    ....
}
```
Target 5 (Hint): Bypassing DEP

Return to Libc review

- Code Reuse Attack. (Use the code that’s already there: Program + shared libraries)
- Need to find the address of:
  - system()
  - String “/bin/sh”
- Overwrite the return address to point to start of system()
- Place address of “/bin/sh” on the stack so that system() uses it as the argument
  - To be clean, you also want to push exit() on the stack so it will shut down gracefully
  - But in this target you don’t have to do that. (okay if segfault when root shell exit)
Target 5 (Hint): Bypassing DEP

Return to Libc review

- What we want to get to:
- Transfer control to address of system() in libc
- Setup stack frame to look like a normal call to system()
  - int system(const char *command);
  - &exit() system call is in the slot where the return address would be
  - &cmd is the argument
  - It points to the string “/bin/sh” stored further down the stack
Target 5 (Hint): Bypassing DEP

Return to Libc review

- Want to return to system()
  - how?
- Need to set up stack frame that looks like a legit call to system
  - Argument to be “bin/sh”
Target 5 (Hint): Bypassing DEP

Return to Libc review

- Want to return to system()
  - Overwrite eip to be &system()
- Need to set up stack frame that looks like a legit call to system
  - Argument to be &"bin/sh"
Target 6 (Hint): Variable stack position

- ASLR (address-space layout randomization) enabled
- Stack can be placed anywhere in memory (0–255 bytes)
- How can I make sure my shell code always hit?
  - Hint: NOP sleds
  - NOP = 0X90
  - If number large enough, there's a region on stack that always have nop in it.

// Compiled with -DMINIASLR.
void vulnerable(char *arg)
{
    char buf[1024];
    strcpy(buf, arg);
}

int _main(int argc, char *argv[])
{
    ...
    vulnerable(argv[1]);
    return 0;
}
Target 6 (Hint): Variable stack position
Target 7: ROP

- Return Oriented Programming
- Extra credit (4 points)
- Identical to target2, but it is compiled with DEP enabled
- Implement a ROP-based attack to bypass DEP and open a root shell.
- ROPgadget.
  - The --binary, --badbytes, --multibr, and --ropchain flags will be particularly helpful.
- Closest to real-world attacks

```c
void vulnerable(char *arg) {
    char buf[100];
    strcpy(buf, arg);
}

int _main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

```c
setuid(0); //eax = 23
execve("/bin/sh", 0, 0); //eax = 11
```