Week 3 Discussion

Leo Cao
Karthik Mudda
Sumanth Rao
%eip is a register pointing to the instruction that CPU will execute in next cycle

Saved %eip references to a 4-byte address value stored on stack

Saved %eip 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

Return address == saved return address == saved %eip == %ebp+4
int _main(int argc, char *argv[]) {
    char grade[5];
    char name[10];
    strcpy(grade, "nil");
    gets(name);
    printf("Hi %s! Your grade is %s.
", name, grade);
    exit(0);
}

- How does `gets` copy stdin into `name`?
- When does `gets` stop copying?
- Will `gets` check the boundary between `name` and `grade`?

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);
}
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;
}

- 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?
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;
}
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?
```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;
}
```

Where do you want to transfer control to?

- `shellcode`
Shellcode - Simple examples and explanations

- [https://www.youtube.com/watch?v=1S0aBV-Waeo](https://www.youtube.com/watch?v=1S0aBV-Waeo)

- [https://www.youtube.com/watch?v=hJ8IwyhqzD4](https://www.youtube.com/watch?v=hJ8IwyhqzD4)
void vulnerable(char *arg) {
    int *p;
    int a;
    char buf[2048];
    
    strncpy(buf, arg, sizeof(buf) + 8);
    
    *p = a;
}

- Figure out what goes in ‘p’ and ‘a’
- *p = a is basically dereferencing ‘a’
- Think about how dereferencing can be taken advantage of
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);
}

int main()
{
    unsigned int a = pow(2, 30) + 127;
    unsigned int b = pow(2, 31) + 100;

    printf("%u\n", a * 4);
    printf("%u\n", b * 2);
}
Helpful links

- https://devblogs.microsoft.com/oldnewthing/20050107-00/?p=36773
- https://www.acunetix.com/blog/web-security-zone/what-is-integer-overflow/
Target 5 (Hint)

- 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 6 (Hint)

- 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
- Make sure you can open a root shell 100% of the time

// Compiled with -DMINIASLR.

```c
void vulnerable(char *arg) {
    char buf[1024];
    strcpy(buf, arg);
}

int _main(int argc, char *argv[]) {
    ...
    vulnerable(argv[1]);
    return 0;
}
```
Target 7

- Return Oriented Programming
- Extra credit (15 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
```