Introduction to Information Security

Stack Overflow
Buffer Overflow

• Low-level languages (assembly, C, C++) don't have boundary checks
• Careless programmers and large inputs may cause buffer to overflow
• This usually leads to a crash
  • Denial of Service (DoS)
• But sometimes, the input can be crafted to affect the execution flow
  • Control Hijacking

```c
if (x->field > 1) {
  ...
  x->method();
}
```
Stack Overflow

• When a function is called
  • Its return address is pushed unto the stack
  • It is jumped to, and allocates a stack frame
• When the function returns
  • It deallocates its stack frame
  • Its return address is popped off the stack and jumped to
• What if we overflow a buffer until we overflow its return address?
  • A crash!
  • Or maybe...
Stack Overflow

- The buffer is at 0x108
  - We can write code unto the buffer
  - And rewire the return address to jump to it
The Return

- But what is the return address?
  - When a program crashes, it creates a core dump –
  - Core dumps can be opened in GDB, to view the program's state.
The Code

• But what code do we write?
  • Raw opcodes (not an ELF file) that opens a shell – a shellcode

```
/JMP _GET_BIN_BASH
/_GOT_BIN_BASH:
  MOV EAX, 0x0B
  POP EBX  # and popped right off
  MOV ECX, 0x00  # This one you solve yourself 😊
  INT 0x80  # system call: execve("/bin/sh", NULL)
/_GET_BIN_BASH:
  CALL _GOT_BIN_BASH
._STRING "/bin/sh"
```

pushed unto the stack as the return address
• We don't have to jump precisely to our code
• We can land nearby and "slide" to it on NOPs
Null Bytes

- Many stack overflows are caused by use of insecure standard functions like `strcpy` or `strcat`
  - These functions copy until a null byte (`\x00`) is reached
- On one hand – the input size is not checked!
  - Today, more secure functions like `strncpy` and `strncat` are used instead
- On the other hand – the shellcode can't contain null bytes
  - What can we do with `MOV ECX, 0`?  
    - `b9 00 00 00 00`  
      - `XOR ECX, ECX`  
        - `31 c9`
  - What can we do with `.STRING "/bin/bash"`?  
    - `2f 62 69 6e 2f 62 61 73 68 00`  
  - `.ASCII "/bin/bash#"`  
    - `2f 62 69 6e 2f 62 61 73 68 23`
  - Load it, and then fix it
Addendum

Related Vulnerabilities
Format String Vulnerabilities

• Another way to find out the stack address
  • `printf("%s", input)` is OK
  • `printf(input)` is not
    • What if input is "

```c
#include <stdio.h>

int f(char* input)
{
    printf(input);
    printf("\n");
    return 0;
}

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

```c
#include <stdio.h>

int f(char* input)
{
    printf(input);
    printf("\n");
    return 0;
}

int main(int argc, char* argv[])
{
    return f(argv[1]);
}
```
Integer Overflow

• Checking the input size should do it, right?
  • Not unless the programmers keep being careless
  • If int x = 10, then x > 5 is false
  • If int x = 4294967295, is x > 5 still false?

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int size;
    char data[size];
    FILE* fp = fopen(argv[1], "r");
    fread(&size, sizeof(size), 1, fp);
    if (size > 5) {
        printf("File is too large.\n");
        return -1;
    }
    fread(data, 1, size, fp);
    return 0;
}
```
Heap Spray

- Overflows can happen on the heap as well as the stack
  - "Use after free" vulnerability
    - More common than you think, especially in complex, multi-threaded code
    - The attacker allocates lots of "mines" (that is – sprays the heap with them)
    - When the freed object is used, it may trigger a mine
      - Can be used to hijack control
      - Can be used to escape a sandbox