Introduction

- Definitions
  - Smash the stack
    - When code goes beyond allocated array space
  - Buffer
    - A contiguous block of memory that holds multiple definitions of the same data type
  - Dynamic Variable
    - A variable which is allocated at run time

Process Memory Organization

- Processes divided into 3 regions
  - Text
    - Holds code and read-only data
  - Data
    - Holds static data
    - Size doesn’t change except with system call
    - If region is exhausted, processes are blocked
  - Stack
    - The region we’ll be working with (on next slide)

Stack Region

- Size is dynamic
- Consists of stack frames
  - Each function call has a stack frame
  - Stack frame is pushed when function is called
  - Stack frame is popped when control returns from a function
  - Each stack frame holds
    - Parameters of a function
    - Local variables
    - Data necessary to recover previous stack frame
  - Stack can grow up or down towards lower memory addresses (we assume down)

Pointers used in Stack Region

- Stack pointer
  - Points to top of stack
- Frame pointer (Local Base pointer)
  - Points to fixed location within a frame
- Values in stack pointer have constant offset from FP
  - Negative offset for local variables
  - Positive offsets for actual parameters

Calling a procedure

- Procedure prolog
  - Save previous FP
  - Copy SP into FP to create new FP
  - Advance SP to reserve space for local variables
- Procedure epilog
  - Clean up stack
Buffer Overflows

- Stuff more data into buffer than it can handle
- Normally, a segment violation is returned
- But if we are careful, we can make the return value meaningful (and not what the original programmer intended!)

Example of Buffer Overflow

- This program tries to stuff a 256 byte string into a space that is only 16 bytes long
- Running this program will result in a segment violation
- Memory picture:
  - Buffer1 is 2 words, or 8 bytes long
  - Buffer2 is 1 word, or 4 bytes long
  - Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
  - When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Example

- Make return pointer point back to our buffer
- buffer1's start byte to get return sfp
- buffer1 is 2 words, or 8 bytes long
- Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
- When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Why 8 in previous example?

- Use test value (like AAA's in first example)
- Compile program
- Fire up gdb to get assembly code
- We can see that the return from function is at address 0x80004a8
- We want to skip assignment at address 0x80004b2
- We want to skip to instruction at 0x80004b2
- Math tells us that's 8 bytes

What Does the Code Look Like?

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

execute
{
  OS dependant system call to execute code
  Insert exit syscall after execute in case execute call fails
  Use gdb to create assembly code of main and execute
  Insert assembly code into buffer
  But where, exactly? (see next slide!)

Spawning a Shell

- Most useful 'arbitrary' code is a shell spawn
- Insert code we want to execute into buffer we are overflowing
- Make return pointer point back to our buffer
- Example

Example of Buffer Overflow

- This program tries to stuff a 256 byte string into a space that is only 16 bytes long
- Running this program will result in a segment violation
- Memory picture:
  - Buffer1 is 2 words, or 8 bytes long
  - Buffer2 is 1 word, or 4 bytes long
  - Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
  - When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Example

- Make return pointer point back to our buffer
- buffer1's start byte to get return sfp
- buffer1 is 2 words, or 8 bytes long
- Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
- When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Why 8 in previous example?

- Use test value (like AAA's in first example)
- Compile program
- Fire up gdb to get assembly code
- We can see that the return from function is at address 0x80004a8
- We want to skip assignment at address 0x80004b2
- We want to skip to instruction at 0x80004b2
- Math tells us that's 8 bytes

What Does the Code Look Like?

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

execute
{
  OS dependant system call to execute code
  Insert exit syscall after execute in case execute call fails
  Use gdb to create assembly code of main and execute
  Insert assembly code into buffer
  But where, exactly? (see next slide!)

Spawning a Shell

- Most useful 'arbitrary' code is a shell spawn
- Insert code we want to execute into buffer we are overflowing
- Make return pointer point back to our buffer
- Example

Example of Buffer Overflow

- This program tries to stuff a 256 byte string into a space that is only 16 bytes long
- Running this program will result in a segment violation
- Memory picture:
  - Buffer1 is 2 words, or 8 bytes long
  - Buffer2 is 1 word, or 4 bytes long
  - Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
  - When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Example

- Make return pointer point back to our buffer
- buffer1's start byte to get return sfp
- buffer1 is 2 words, or 8 bytes long
- Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
- When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Why 8 in previous example?

- Use test value (like AAA's in first example)
- Compile program
- Fire up gdb to get assembly code
- We can see that the return from function is at address 0x80004a8
- We want to skip assignment at address 0x80004b2
- We want to skip to instruction at 0x80004b2
- Math tells us that's 8 bytes

What Does the Code Look Like?

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

execute
{
  OS dependant system call to execute code
  Insert exit syscall after execute in case execute call fails
  Use gdb to create assembly code of main and execute
  Insert assembly code into buffer
  But where, exactly? (see next slide!)

Spawning a Shell

- Most useful 'arbitrary' code is a shell spawn
- Insert code we want to execute into buffer we are overflowing
- Make return pointer point back to our buffer
- Example

Example of Buffer Overflow

- This program tries to stuff a 256 byte string into a space that is only 16 bytes long
- Running this program will result in a segment violation
- Memory picture:
  - Buffer1 is 2 words, or 8 bytes long
  - Buffer2 is 1 word, or 4 bytes long
  - Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
  - When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Example

- Make return pointer point back to our buffer
- buffer1's start byte to get return sfp
- buffer1 is 2 words, or 8 bytes long
- Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
- When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Why 8 in previous example?

- Use test value (like AAA's in first example)
- Compile program
- Fire up gdb to get assembly code
- We can see that the return from function is at address 0x80004a8
- We want to skip assignment at address 0x80004b2
- We want to skip to instruction at 0x80004b2
- Math tells us that's 8 bytes

What Does the Code Look Like?

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

execute
{
  OS dependant system call to execute code
  Insert exit syscall after execute in case execute call fails
  Use gdb to create assembly code of main and execute
  Insert assembly code into buffer
  But where, exactly? (see next slide!)

Spawning a Shell

- Most useful 'arbitrary' code is a shell spawn
- Insert code we want to execute into buffer we are overflowing
- Make return pointer point back to our buffer
- Example

Example of Buffer Overflow

- This program tries to stuff a 256 byte string into a space that is only 16 bytes long
- Running this program will result in a segment violation
- Memory picture:
  - Buffer1 is 2 words, or 8 bytes long
  - Buffer2 is 1 word, or 4 bytes long
  - Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
  - When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Example

- Make return pointer point back to our buffer
- buffer1's start byte to get return sfp
- buffer1 is 2 words, or 8 bytes long
- Total is 12 bytes, so we add 12 to buffer1's start byte to get return address
- When the program tries to return, we want it to skip the x=1 assignment, so we add 8 to ret's pointer value (why 8? Next slide!)

Why 8 in previous example?

- Use test value (like AAA's in first example)
- Compile program
- Fire up gdb to get assembly code
- We can see that the return from function is at address 0x80004a8
- We want to skip assignment at address 0x80004b2
- We want to skip to instruction at 0x80004b2
- Math tells us that's 8 bytes
Where do we put our code?

- Use JMP and CALL.
  - Put a Call instruction right after our code and a Jump instruction right before it
  - Since these can use IP relative addressing, we don’t need to know absolute addresses
  - When the call is executed, our code will be pushed onto stack as return address
  - Copy return address into register
  - Call instruction can call start of our code

Memory map

```text
DDDDDDDDEEEEEEEEEEEEE EEEE FFFF
89ABCDEF01234567890AB CDEF 0123 4567 89AB CDEF
```

Finding where to start our code

- Pad front of overflow buffer with NOPs
- Fill about half the buffer with NOPs, then our code, then the return addresses
- Now we can return anywhere in the front half of the buffer, and we’ll be okay
- A good size for our buffer is about 100 bytes more than the buffer we’re trying to overflow
- Use trial and error to find right buffer size and offset
- You’ll know you’re right when you get the shell prompt

Writing an Exploit
(or how to mung the stack)

- Fill large_string with address of buffer
  - This is where our code is
- Copy shellcode into beginning of large_string
  - steady will copy without doing bounds checking
  - Will overflow into return address
- Once it reaches end of main, it will try to return, but it will ‘return’ to our code!

```c
char shellcode[] = "(\x90)*ret\x90...";
char large_string[128];

void main() {
    char buffer[128];
    long long_ptr = (long *)large_string;
    for (i=0; i<12; i++) {  //overflow
        long_ptr[i] = (int)buffer;
    }
    strcpy(buffer, large_string);
}
```

Next Steps

- Calculate offsets from jmp to call, and from call to the first instruction in our code
- Problem: Where we want to insert our code is designated read-only!
  - Solution: place code in stack or data segment and transfer control to it
  - We will place our code in global array in data segment
    - Need hex representation of binary code
    - Copy code and write hex to gdb

Where does the buffer start?

- The stack starts at the same address every time
  - We can write a little program to output the stack pointer
- Usually no more than a few hundred or thousand bytes are pushed onto the stack at one time
  - We can try to guess where the buffer starts
  - Very inefficient and nearly impossible!
  - Problem is we need to know exactly where the address of our code will start

Small Buffer Overflows

- There are times when buffer is so small that
  - Our code won’t fit
  - We’ll overwrite the return address with our instructions
  - The number of NOPs in our code doesn’t help us
- Solution
  - Place code in environment variable (must have access, of course, to environment variables)
  - Overflow buffer with the address of our chosen variable
Finding Buffer Overflows

- Need a function that does no bounds checking
  - Examples in C are:
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets
    - while loops that read one character at a time
- Use the source d00d
  - Many closed source OSs are very similar to open source OSs

Summary

- Components that make buffer overflow possible
  - C functions that don’t bounds check
  - The buffer is a known (or discoverable) distance from the return pointer in the stack
  - Being able to store code in writable memory and forcing ‘return’ to that memory
  - Using gdb to disassemble code and calculate certain lengths
- Components that make buffer overflow easier
  - Using NOPs so we don’t need to know where the buffer starts
  - Using environment variables in cases of buffers that are too small

The End

- Questions or comments?