Stack-Based Buffer Overflow

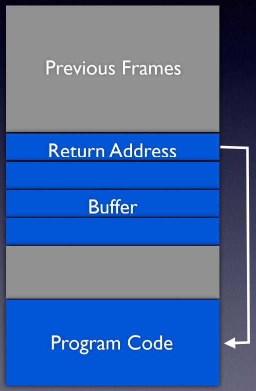
IT430 - Information Assurance
7 April 2014

Outline

- How the stack works
- Simple stack buffer overflow
- Stack smashing
- Prevention

The Stack

- Grows downward
- Organized in *frames* - each frame is associated with an active procedure call
- Frame includes *return address* and storage for parameters and local variables



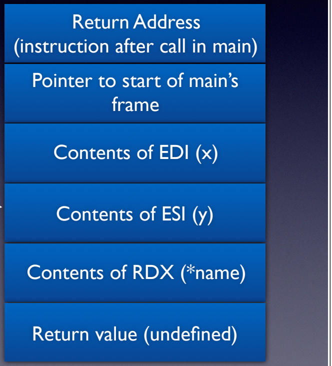
Stack Example

```
int my_func(int x, int y, char* name)
{
  int xx, yy, zz;
  float ans;
}
main()
{
  char name[6] = "chris";
  my_func(1, 2, name);
}
```

Stack Example

`x` in EDI, `y` in ESI, `*name` in RDX

```
_my_func:
   pushq   %rbp
 movq %rsp, %rbp
 movl %edi, -4(%rbp)
 movl %esi, -8(%rbp)
 movq %rdx, -16(%rbp)
 movl -20(%rbp), %eax
 popq %rbp
 ret
```


Return Address (instruction after call in main)

Pointer to start of main's frame

Contents of EDI (x)

Contents of ESI (y)

Contents of RDX (*name)

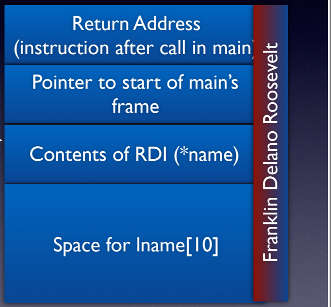
Return value (undefined)

What will go wrong?

```
#include <string.h>
int my_func(char *name)
{
  char lname[10];
  strcpy(lname, name);
  return(0);
}
main()
{
  char name[50] = "franklin delano roosevelt";
  my_func(name);
}
```

The Assembly

```
_my_func:
 pushq%rbp
 movq %rsp, %rbp
 subq $48, %rsp
 movq %rdi, -8(%rbp)
 leaq -34(%rbp), %rax
 movq -8(%rbp), %rcx
 movabsq $10, %rdx
 movq %rax, %rdi
 movq %rcx, %rsi
 callq___strcpy_chk
   ...code omitted...
 ret
```


Return Address (instruction after call in main)

Pointer to start of main's frame

Contents of RDI (*name)

Space for lname[10]

Stack Smashing

- Use the buffer overflow to change execution flow
- Clever construction of overflow to replace return address with address of malicious code
- When function returns, will jump to malicious code

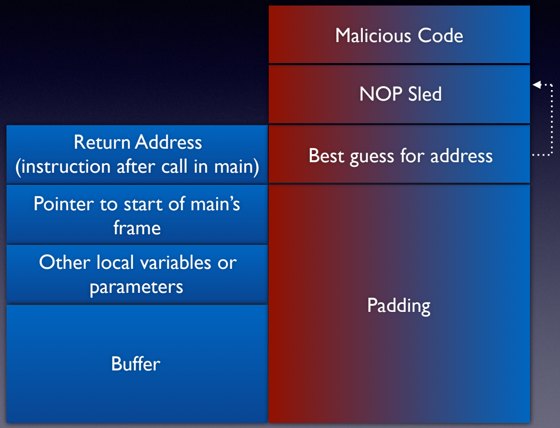
Challenges (to attacker)

- Knowing where the return address is relative to the buffer
  - Not so bad if you have source code and know the architecture of the target
- Knowing what to replace return address with
  - Can’t just put malicious code anywhere
  - Try to keep code in the stack

NOP Sledding

- Deals with ambiguity in address of malicious code
- Precede code with a bunch of no-ops (NOPs)
- If overwritten return address points to a location within the NOP block, execution will eventually reach the malicious code

NOP Sledding Picture


Malicious Code
NOP Sled
Return Address (instruction after call in main)
Best guess for address
Pointer to start of main’s frame
Other local variables or parameters
Padding
Buffer

Trampolining

- Use known location of a standard library to provide a target destination for attack
- For example, DLL known to include a jump to the address in `ESP`
  - Get address of malicious code in `ESP`
  - Overwrite return address with address of `jmp esp` in DLL

Return-to-libc

- Similar to trampolining; use known location in memory of C standard library (libc)
- Cause execution to jump to useful library function: `system()`, `execv()`, etc.

Shellcode

- Buffer overflow may allow for execution of arbitrary code
- Attacker would like to open shell with elevated privileges
- *Shellcode* is carefully crafted, malicious machine language code that is executed via the buffer overflow attack
- Shellcode is highly constrained; writing shellcode is challenging

Shellcode Constraints

- *No null bytes* - shellcode must survive string processing; nulls indicate end of string
- *Small code* - shellcode may have to fit into small portion of the stack
- Consider a small example...

Opening a Shell

- Call execve() with the following arguments
  - path - `“/bin/sh\0”`
  - argv - `[“/bin/sh\0”, 0x00]`
  - envp - `[0x00]`
- In assembly, call execve() using interrupt 0x80. Here’s one way to do it...

```
BITS 32

section .text
        global _start

_start:

cont:   xor eax, eax    ; zero contents of eax
        push eax        ; null terminator for "/bin//sh"
        push "hs//"     ; push "/bin//sh" on stack
        push "nib/"     ;    has to be byte reversed
        mov ebx, esp    ; save address of "/bin//sh" to ebx
        push eax        ; write args[1] / null envp on stack
        mov edx, esp    ; save address of envp in edx
        push ebx        ; write args[0] on stack
        mov ecx, esp    ; save args in ecx
        mov al, 0xb     ; copy execve syscall number to %al
        int 0x80        ; execute the system call
```

- This assembles to:

```
00000000  31 c0 50 68 68 73 2f 2f  68 6e 69 62 2f 89 e3 50
00000010  89 e2 53 89 e1 b0 0b cd  80
```

- Notice that we had to push “/bin//sh” with bytes reversed - that’s because the push statement reverses bytes of its argument
- The extra “/” has no effect
- No null bytes!

Putting it Together

- A successful attack combines multiple techniques
  - Overwrite return address to control execution flow
  - NOP sled to compensate for ambiguity in memory layout
  - Shellcode gives attacker access to system
- We will see an example in Friday’s Lab