Shellcoding

Robert Galvan

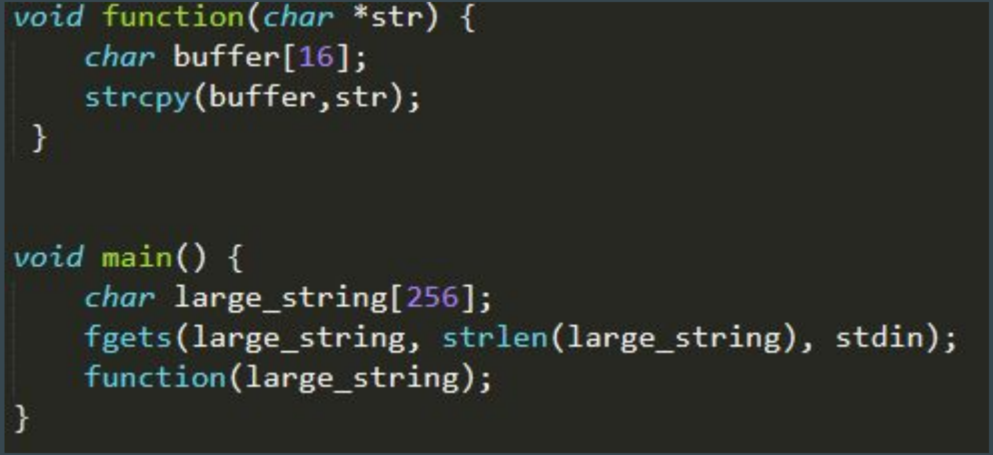
About Me

- Senior IS Major, Comp Sci Minor
- Interned at a couple of Government Contractors
  - Embedded RE
  - Vuln research
- Huntress
- CTFs
- Full time: small contractor doing RE and Development

Disclaimer

- Slides based on MBE Course from RPI
- Not a shellcoding pro

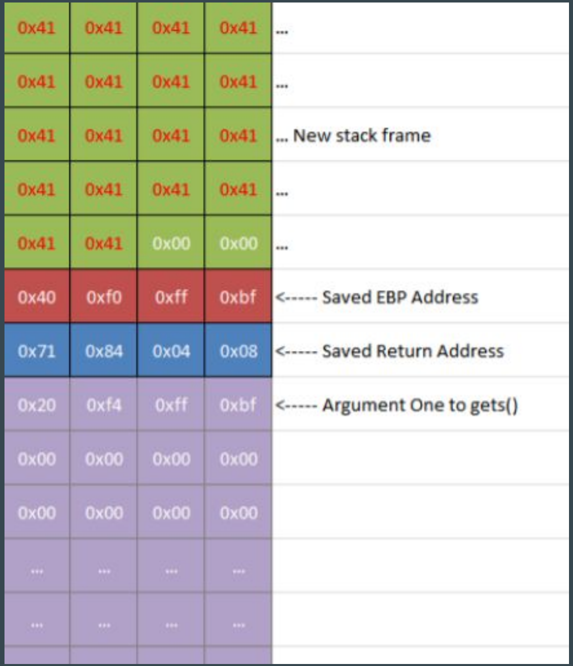
Stack Smashing Review



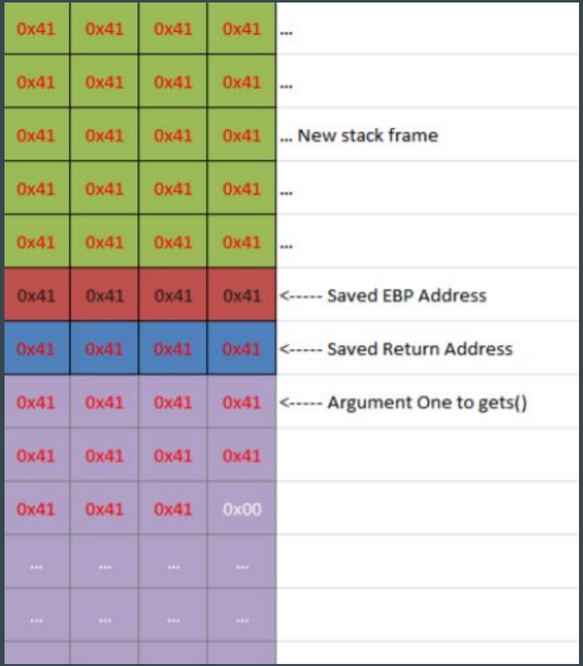
```
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
 }


void main() {
    char large_string[256];
    fgets(large_string, strlen(large_string), stdin);
    function(large_string);
}
```

Stack Smashing Review



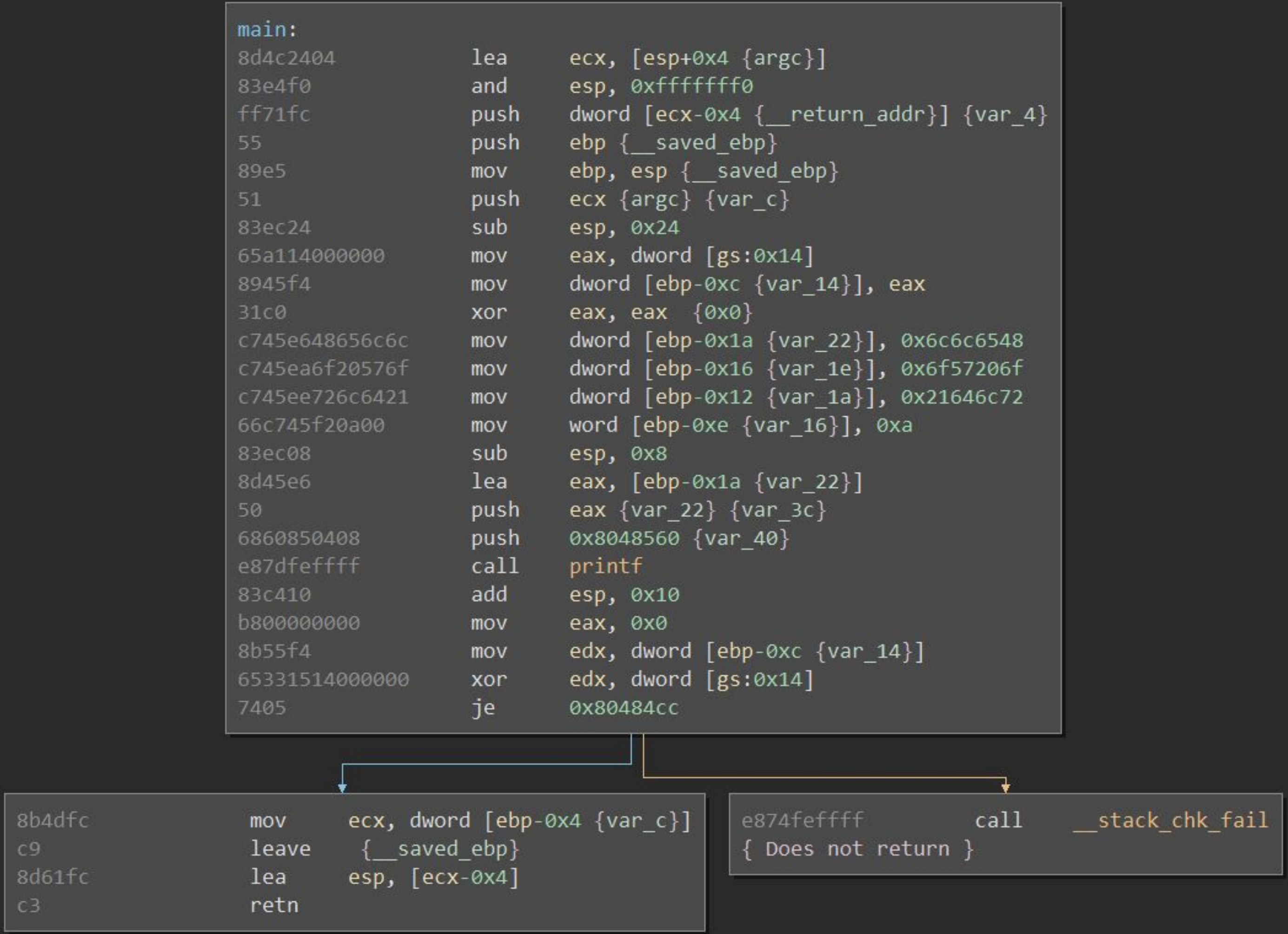
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x00	0x00	...
0x40	0xf0	0xff	0xbf	<----- Saved EBP Address
0x71	0x84	0x04	0x08	<----- Saved Return Address
0x20	0xf4	0xff	0xbf	<----- Argument One to gets()
0x00	0x00	0x00	0x00	
0x00	0x00	0x00	0x00	
...	...	...	...	
...	...	...	...	



0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	... New stack frame
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	...
0x41	0x41	0x41	0x41	<----- Saved EBP Address
0x41	0x41	0x41	0x41	<----- Saved Return Address
0x41	0x41	0x41	0x41	<----- Argument One to gets()
0x41	0x41	0x41	0x41	
0x41	0x41	0x41	0x00	
...	...	...	...	
...	...	...	...	

Defining Shellcode

- A set of instructions that are injected by the user and executed by the exploited binary
- Generally the ‘payload’ of an exploit
- Using shellcode you can essentially make a program execute code that never existed in the original binary
- You’re basically injecting code



```
main:
8d4c2404          lea     ecx, [esp+0x4 {argc}]
83e4f0            and     esp, 0xfffffff0
ff71fc            push    dword [ecx-0x4 {__return_addr}] {var_4}
55                push    ebp {__saved_ebp}
89e5              mov     ebp, esp {__saved_ebp}
51                push    ecx {argc} {var_c}
83ec24            sub     esp, 0x24
65a114000000      mov     eax, dword [gs:0x14]
8945f4            mov     dword [ebp-0xc {var_14}], eax
31c0              xor     eax, eax  {0x0}
c745e648656c6c    mov     dword [ebp-0x1a {var_22}], 0x6c6c6548
c745ea6f20576f    mov     dword [ebp-0x16 {var_1e}], 0x6f57206f
c745ee726c6421    mov     dword [ebp-0x12 {var_1a}], 0x21646c72
66c745f20a00      mov     word [ebp-0xe {var_16}], 0xa
83ec08            sub     esp, 0x8
8d45e6            lea     eax, [ebp-0x1a {var_22}]
50                push    eax {var_22} {var_3c}
6860850408        push    0x8048560 {var_40}
e87dfeffff        call    printf
83c410            add     esp, 0x10
b800000000        mov     eax, 0x0
8b55f4            mov     edx, dword [ebp-0xc {var_14}]
653315140000 00   xor     edx, dword [gs:0x14]
7405              je      0x80484cc
```

```
8b4dfc            mov     ecx, dword [ebp-0x4 {var_c}]
c9                leave    {__saved_ebp}
8d61fc            lea     esp, [ecx-0x4]
c3                retn
```

```
e874feffff        call     __stack_chk_fail
{ Does not return }
```

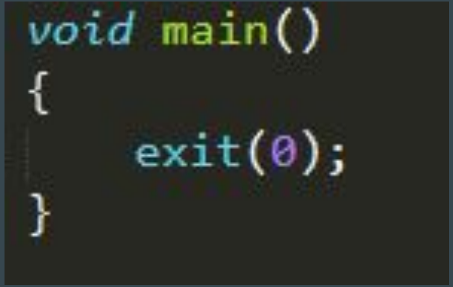
Basic Examples

https://defuse.ca/online-x86-assembler.htm

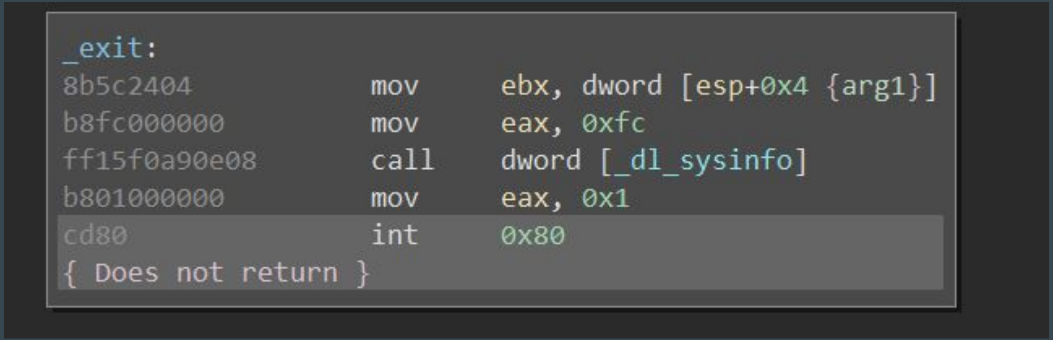
Syscalls

- How do we call functions like printf?
- System calls are how userland programs talk to the kernel to do anything interesting
- open files, read, write, map memory, execute programs, etc
- libc functions are high level syscall wrappers
    - fopen()
    - sscanf()
    - execv()
    - printf()

Example of syscall



```c
void main()
{
    exit(0);
}
```



```
_exit:
8b5c2404            mov     ebx, dword [esp+0x4 {arg1}]
b8fc000000          mov     eax, 0xfc
ff15f0a90e08        call    dword [_dl_sysinfo]
b801000000          mov     eax, 0x1
cd80                int     0x80
{ Does not return }
```

Using Syscalls in Shellcode

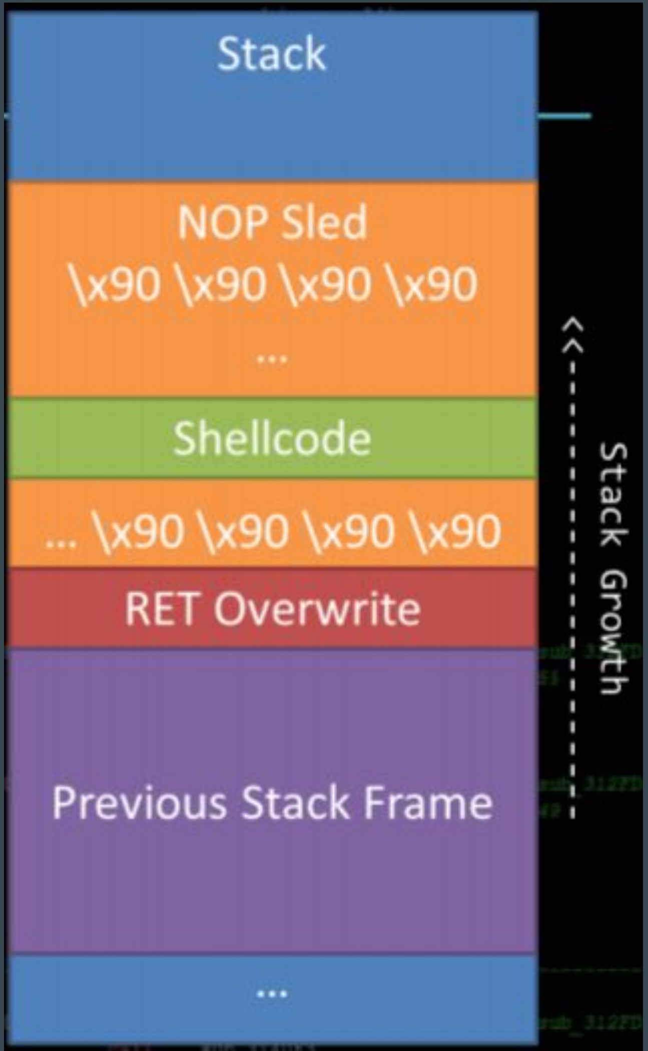
- Need syscalls to do interesting things
- Syscalls can be made in x86 using interrupt 0x80
  - int 0x80
- https://syscalls.kernelgrok.com/
- exec("/bin/sh")

Buffer overflow now what?

- Control EIP
- Input generally stored on stack
- Point EIP to location of shellcode
- Stack is often unreliably

NOP Sleds

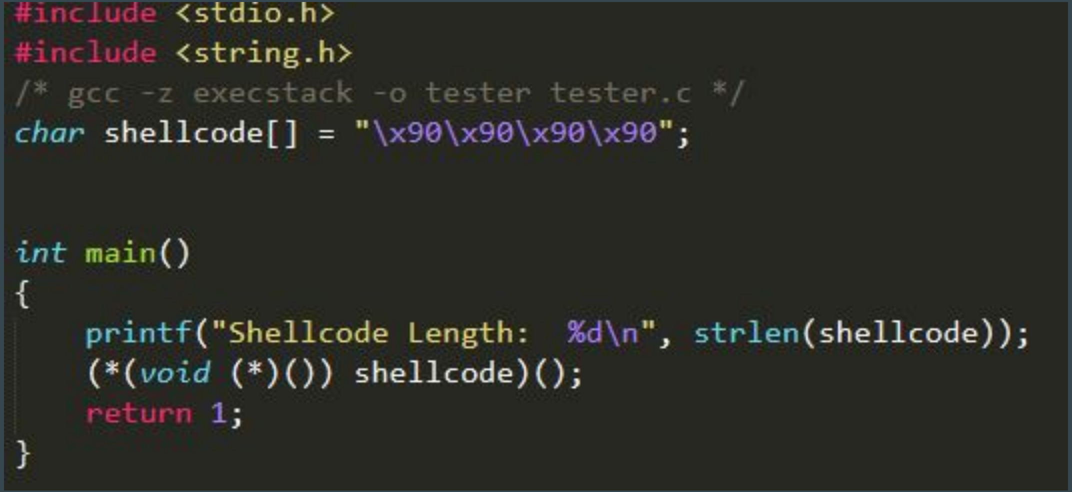
- “Nop” = (\x90) is an instruction that does nothing
- We can pad our shellcode with nops



Things to keep in mind

- \x00 (null) byte stops most string functions
  - strcpy(), strlen(), strcat(), strcmp()
  - The instruction mov eax, 4 ; “\xB8\x04\x00\x00\x00”
  - can be replaced by: mov al, 4 ; “\xb0\x04”
  - xor eax,eax ;clears register
- \x0A (newline) bytes causes gets(), fgets() to stop reading
  - Not nulls
- Endianness
- Stack addresses changes inside of GDB
  - We can attach with gdb after the program has begun
- Nx Bit (DEP)?
  - Why can't we always use shellcode?
  - checksec

Shellcode tester



```c
#include <stdio.h>
#include <string.h>
/* gcc -z execstack -o tester tester.c */
char shellcode[] = "\x90\x90\x90\x90";


int main()
{
    printf("Shellcode Length:  %d\n", strlen(shellcode));
    (*(void (*)()) shellcode)();
    return 1;
}
```

First Challenge

http://165.227.113.74:8080/

- Create a flag.txt file in the same directory as the binary
- export WUNTEE_CHALLENGE_FLAG=flag.txt
- Start in gdb
- Set breakpoint at 0x80488e9
- r B001CD80   ;exit

Tools

- http://shell-storm.org/
- http://www.exploit-db.com/shellcode/
- You should reuse shellcode
- Pwntools asm()
- Passing hex to input:
    - `(python -c 'print "\x90"*20 + "\x31\xc0"'; cat -) | nc pwn.me.org 555`
    - `(python -c 'print "\x90"*20 + "\x31\xc0"'; cat -) | ./level1`

If you finished MBE try ORW from pwnable.tw