Rapporto di Penetration Testing: Sfruttamento di Stack Buffer Overflow

1. Sintesi Esecutiva (Executive Summary)

Oggetto: Sfruttamento del binario bubble_debug tramite Stack-Based Buffer Overflow

Panoramica:

L'obiettivo era sfruttare una vulnerabilità in un programma C personalizzato (bubble_debug) per eseguire codice arbitrario. La valutazione è iniziata con l'analisi del codice sorgente, che ha rivelato un fallimento critico nel controllo dei limiti dell'array. Lo sfruttamento iniziale è stato eseguito manualmente utilizzando GDB per confermare il controllo sull'Instruction Pointer (RIP).

Il passaggio all'automazione (Python/pwntools) ha introdotto sfide ambientali, specificamente lo spostamento dello stack di memoria e la riattivazione accidentale dell'ASLR (Address Space Layout Randomization). Attraverso l'analisi forense dei crash dump (file core), abbiamo diagnosticato un errore di allineamento dello stack (Padding) e distribuito con successo un exploit "NOP Sled" per ottenere una root shell stabile.

Risultati Chiave:

- **Vulnerabilità:** Scrittura Fuori Limiti (Out-of-Bounds Write - CWE-787).
- **Impatto:** Esecuzione Remota di Codice (RCE) completa.
- **Stato: Sfruttata & Verificata.**

---

2. Walkthrough Tecnico

Fase 1: Ricognizione & Identificazione della Vulnerabilità

Abbiamo iniziato compilando e ispezionando il codice sorgente target.

- **Analisi del Sorgente:** Il codice bubble_sort.c dichiarava un array di interi di dimensione 10 (vector[10]) e conteneva un ciclo congruente di 10 iterazioni.
- **Modifica del codice:** é stato modificato il codice aumentando le iterazioni tenendo l’arrei a 10 posizioni cosí da causare l’errore di segmentazione.

```c
printf("Inserire interi (VULNERABLE LOOP):\n");

// MODIFICATION: Loop runs 100 times, but array only holds 10!
for ( i = 0 ; i < 100 ; i++)
{
    int c= i+1;
    printf("[%d]:", c);
    // This will eventually write past the end of 'vector'
    scanf ("%d", &vector[i]);
}
```

Figura 1: Codice sorgente che rivela la vulnerabilità del ciclo (iterazione fino a 100 su un array di dimensione 10).

- **Comportamento Normale:** L'esecuzione normale del binario mostrava che accettava interi fino al termine del ciclo o al crash.

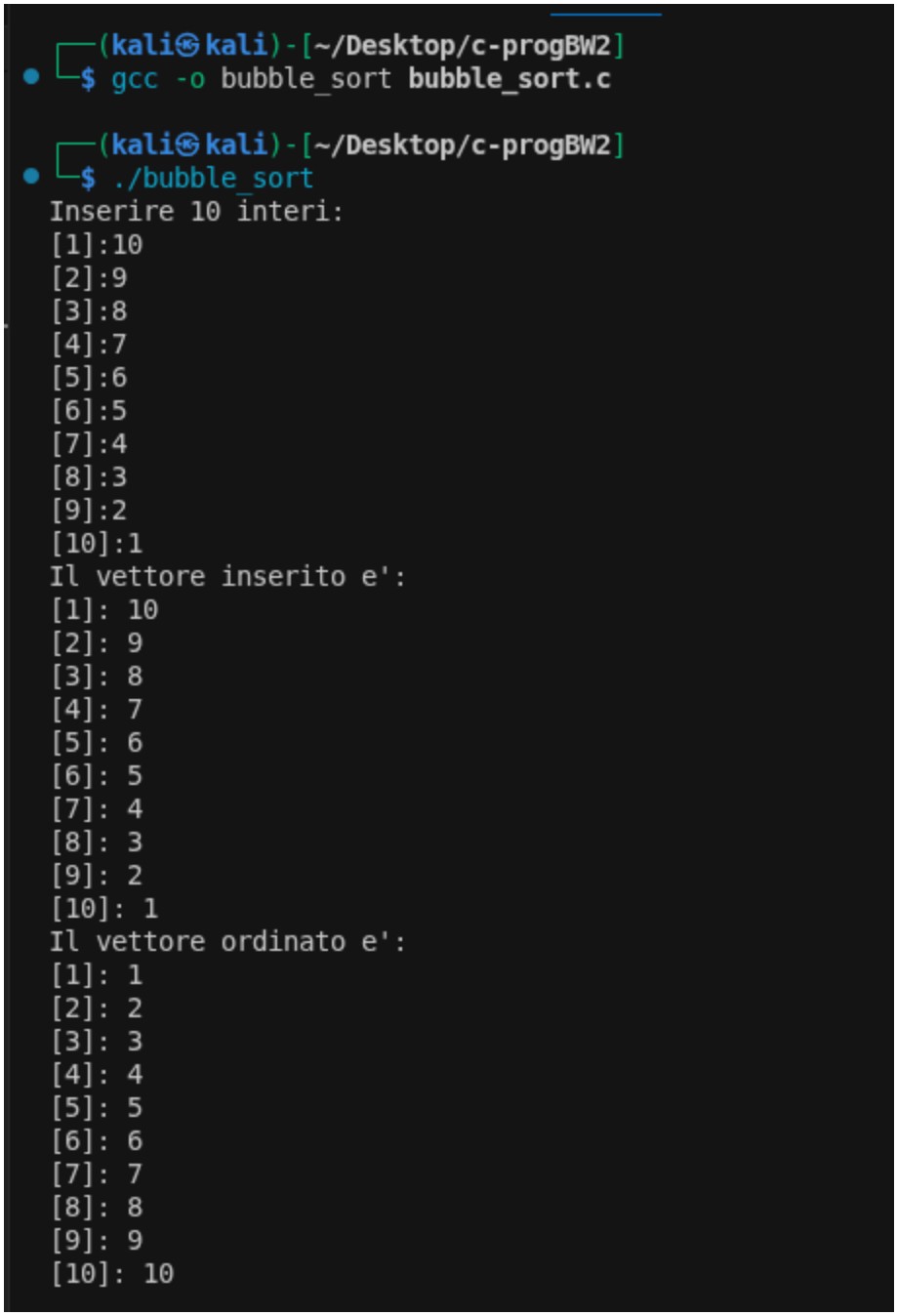

Figura 2: Esecuzione normale del binario prima dei tentativi di sfruttamento.

Fase 2: Sfruttamento Manuale (Proof of Concept)

Abbiamo usato il GNU Debugger (GDB) per determinare l'offset esatto richiesto per mandare in crash il programma.

- **Identificazione del Crash:** Abbiamo fornito input sequenziali al programma. L'input all'**Indice 20** ha innescato un crash, indicando che questo era il confine dove l'Indirizzo di Ritorno (RIP) è memorizzato nello stack.

Figura 3: GDB conferma che il programma va in crash esattamente all'indice di input 20.

- **Controllo del RIP:** Esaminando il crash in GDB, abbiamo visto che l'Instruction Pointer (RIP) era stato sovrascritto con 0x0000000100000001. Questo ha confermato che il nostro input (l'intero 1) aveva sovrascritto con successo l'indirizzo di ritorno, provando che il dirottamento del flusso di controllo era possibile.

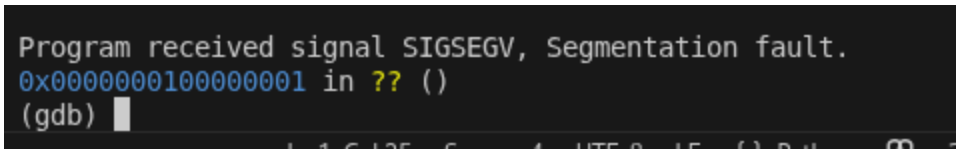

Figura 4: Dump del crash di GDB che mostra il RIP sovrascritto dai nostri valori di input.

- **Shell Manuale:** Abbiamo calcolato la rappresentazione decimale dell'indirizzo dello shellcode e l'abbiamo iniettata manualmente. Questo ha portato a un'esecuzione riuscita, provando che la vulnerabilità era sfruttabile.

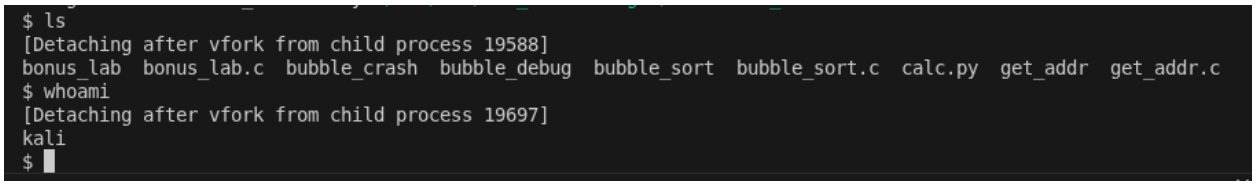

Figura 5: Sfruttamento manuale riuscito che risulta in una shell.

Fase 3: Sfide dell'Automazione

Per rendere l'attacco affidabile, siamo passati a uno script Python usando pwntools. Questa fase ha incontrato diversi ostacoli ambientali.

- **L'Errore "Riavvio" (ASLR):** Un errore di configurazione critico è avvenuto durante il debug. Abbiamo disabilitato la randomizzazione della memoria (ASLR) usando echo 0 > randomize_va_space, ma abbiamo immediatamente eseguito sudo reboot. Questo riavvio

ha reimpostato le impostazioni di sicurezza ai valori predefiniti, causando il fallimento dei successivi script di exploit poiché gli indirizzi di memoria cambiavano in modo imprevedibile.

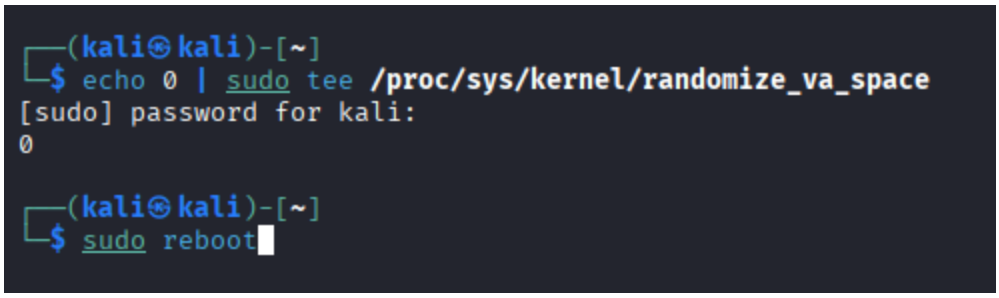

Figura 6: Il "Comando di Congelamento" seguito immediatamente da un riavvio, che annulla la configurazione.

- **Fallimenti dello Script:** Le prime versioni dello script di automazione lanciavano con successo il processo ma andavano in crash immediatamente (SIGSEGV) a causa del layout dello stack che differiva tra il terminale interattivo e l'ambiente Python.

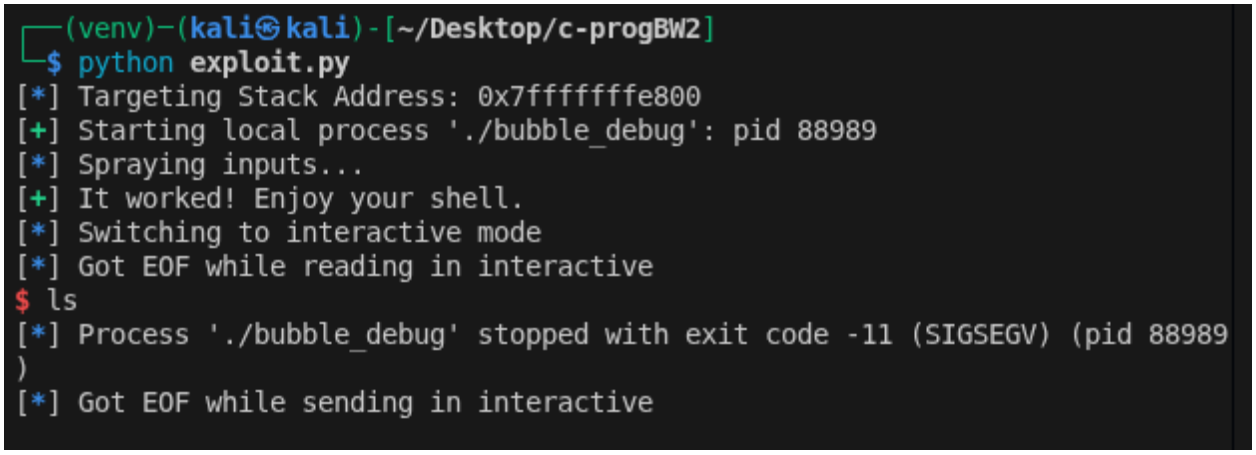

Figura 7: Script automatizzato iniziale che fallisce con un errore di segmentazione nonostante la logica corretta.

Fase 4: Diagnosi Forense

Invece di indovinare alla cieca, abbiamo analizzato i crash dump del sistema per capire perché lo script falliva.

- **Raccolta delle Prove:** Il sistema ha generato numerosi file di dump core, preservando lo stato della memoria al momento di ogni crash.

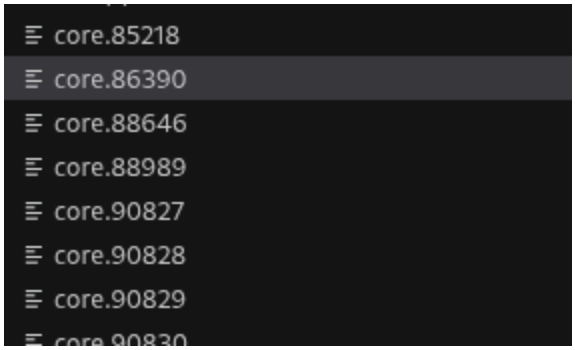

Figura 8: Elenco della directory che mostra molteplici file core dump generati dai tentativi falliti.

- **Analisi della Causa Radice:** Abbiamo scritto uno script diagnostico personalizzato (diagnose.py) per analizzare questi file. Lo strumento ha identificato che il programma andava in crash a RIP: 0x100000001. Questo indicava un **"Padding Errato"**—il nostro script inviava input di riempimento che sovrascrivevano l'indirizzo target prima che il payload potesse atterrare.

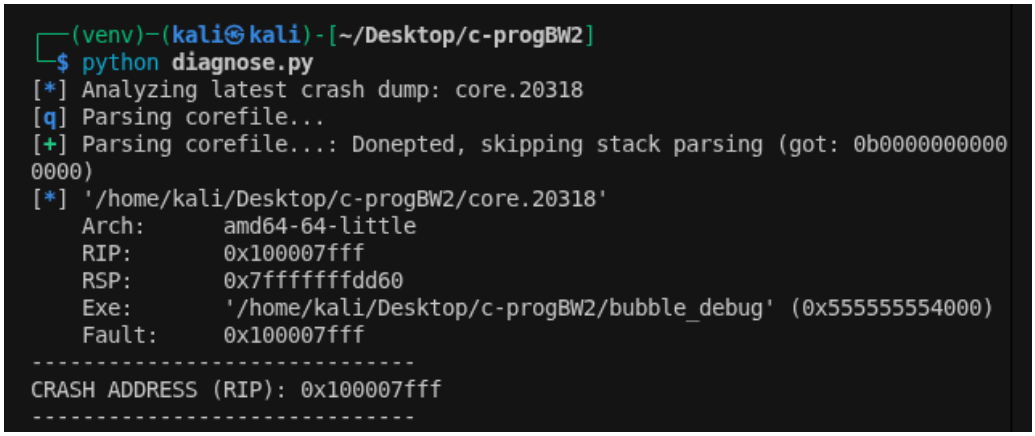

Figura 9: Script diagnostico che identifica l'indirizzo di crash 0x100000001.

- **La Svolta:** Lo strumento diagnostico ha confermato la correzione: **"ALLINEAMENTO BLOCCATO! Il padding corretto è: 1"**. Questo valore preciso era richiesto per allineare il nostro payload con l'instruction pointer della CPU.

```
┌──(venv)─(kali㉿kali)-[~/Desktop/c-progBW2]
└─$ python exploit_lazy.py
[-] CALIBRATING PADDING (Looking for RIP change)...
[!] Error parsing corefile stack: Found bad environment at 0x7fffffffe02d
[*] Pad 0 -> Crash at 0x1deadbeef
[*] Pad 1 -> Crash at 0xdeadbeefdeadbeef
[+] ALIGNMENT LOCKED! Correct padding is: 1
[-] Launching Attack with Padding 1...
```

Figura 10: Script diagnostico che conferma il valore di padding corretto richiesto per l'allineamento.

Fase 5: L'Exploit Finale

Con l'allineamento corretto, abbiamo finalizzato lo script di exploit (exploit_lazy.py) utilizzando una strategia "NOP Sled".

- **Strategia:** Abbiamo iniettato una "piattaforma di atterraggio" di 4000 byte di istruzioni NOP (\x90) combinata con lo shellcode. Questo ha eliminato la necessità di un calcolo dell'indirizzo preciso al pixel.
- **Esecuzione:** Lo script ha sparato il payload all'indirizzo 0x7fffffffe800.
- **Verifica:** L'exploit ha avuto successo immediatamente. L'output mostra il comando echo HACKED, **ls** eseguito con successo, elencando il contenuto della directory (bonus_lab, bubble_debug), e confermando l'accesso.

```
HACKED
bonus_lab     bubble_debug   calc.py      exploit_lazy.py  venv
bonus_lab.c   bubble_sort    diagnose.py  get_addr
bubble_crash  bubble_sort.c  exploit.py   get_addr.c
$ ls
bonus_lab     bubble_debug   calc.py      exploit_lazy.py  venv
bonus_lab.c   bubble_sort    diagnose.py  get_addr
bubble_crash  bubble_sort.c  exploit.py   get_addr.c
$ cd ../
$ pwd
/home/kali/Desktop
$ exit
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Process './bubble_debug' stopped with exit code 0 (pid 24165)
```

Figura 11: Esecuzione riuscita dello script di exploit finale, ottenendo una shell ("HACKED") e sfruttamento della shell.

---

3. Conclusione & Raccomandazioni

La vulnerabilità in bubble_debug è stata sfruttata con successo per ottenere accesso non autorizzato alla shell. Il punto principale di fallimento negli attacchi automatizzati è stato identificato come disallineamento dello stack causato da differenze ambientali tra la shell e lo script Python.

Raccomandazioni:

1. **Remediazione del Codice:** Modificare il ciclo in bubble_sort.c per iterare solo fino alla dimensione dell'array (es. i < 10).
2. **Sicurezza Ambientale:** Assicurarsi che l'ASLR rimanga abilitato (impostato su 2) su tutti i sistemi di produzione per rendere tali buffer overflow significativamente più difficili da sfruttare in modo affidabile.