HACKING DELLO STACK E I DEI SUOI MEANDRI

evilsocket
http://evilsocket.altervista.org/

In questo paper andremo ad analizzare uno degli aspetti più importanti della programmazione C "hacking-oriented" .
Quante volte avrete sentito parlare del misterioso stack ? Se siete dei veri smanettoni come me suppongo tante :D .
Tanto per introdurre il concetto, fatemi dire COSA è lo stack in poche parole .
Generalmente parlando, lo stack è una struttura a "pila" dove si immagazinano dei dati .
Dico a pila perchè segue il metodo "Last In First Out" (il cosiddetto LIFO), ovvero il primo dato ad essere estratto è l'ultimo che è stato inserito nella pila .
Nel nostro caso, andremo a vedere come lo stack viene usato quando viene chiamata una funzione in C .

Prendiamo una funzione di questo tipo :

```
void fai_qualcosa( int numero )
{
 ...
}
```

Tradotto nel linguaggio del processore, una chiamata a questa funzione si traduce come :

```
mov eax, numero
push eax
call fai_qualcosa
ret
```

Ok ok, lo so, molti di voi erano spiazzati sul codice C, figuriamoci sull assembler XD ... don't worry guys, ora mi accingo a spiegare .
L'operazione "mov eax, numero" non fa altro che mettere il valore della variabile "numero" dentro il registro eax (registro a 32 bit) del processore .
Continuando, vediamo "push eax" ... è qui che vi volevo ! :)
Questa linea salva il valore di eax (ovvero il valore di "numero") nello stack ... e si, proprio quello di cui parlavo ^^ .

Le linee

```
call fai_qualcosa
ret
```

come appare evidente chiamano la funzione "fai_qualcosa" e ritornano dalla chiamata .

Ok, a questo punto possiamo trarre alcune conclusioni :

- Gli argomenti di una funzione vengono "pushati" nello stack in ordine inverso, questo perchè essendo estratti seguendo il LIFO devono poter essere prelevati nel giusto ordine .

- La funzione viene cercata in memoria e una volta ottenuto un puntatore ad essa viene eseguita una chiamata .

- La chiamata ritorna e salva eventuali valori di return del registro eax .

Dopo questa breve (insomma :D) introduzione, andiamo a vedere come implementare

queste deduzioni in qualcosa di interessante .
Chi tra di voi si è cimentato almeno una volta nel caricare una funzione da una dll sotto windowz, sicuramente riconoscerà il seguente frammento di codice :

```
typedef int (*LPMESSAGEBOX)( HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT
uType );

HMODULE hMod = LoadLibrary( "user32.dll" );

LPMESSAGEBOX lpMsgBox = (LPMESSAGEBOX)GetProcAddress( hMod, "MessageBoxA" );

lpMsgBox( 0, "Hello World !", "Hello World !", 0 );
```

Senza dilunguarmi troppo nei dettagli, riassumo in parole povere quanto appena scritto :

- Viene definito un tipo, come puntatore a funzione .

- Viene mappata nella memora del processo user32.dll

- Viene risolta la funzione "MessageBoxA" in base al suo symbol-name, ottenendo così un puntatore al relativo indirizzo di memoria .

- Viene chiamata la funzione .

La cosa che dovrebbe incuriosirvi (a suo tempo a me incuriosì tantissimo ^^) è il cast che viene fatto per ottenere il puntatore :

```
LPMESSAGEBOX lpMsgBox = (LPMESSAGEBOX)GetProcAddress( hMod, "MessageBoxA" );
```

Ma perchè questo cast ?!?!? Andiamo un po a vedere come è definito il prototipo della funzione GetProcAddress per capire un po che cavolo di tipo di puntatore ritorna ...
Spulciando nelle MSDN troviamo questo :

```
FARPROC GetProcAddress( HMODULE hModule, LPCWSTR lpProcName );
```

Mh, e mo che cavolo è sto FARPROC ?!??!?! Continuando a spulciare nelle dozzine di headers del windows sdk, finalmente troviamo che altri non è che un "__stdcall *", ovvero una specie di "puntatore generico" a funzione ... ecco perchè il casting !
Di per se, GetProcAddress restituisce un puntatore generico ma, per poterlo utilizzare in base alla funzione che abbiamo caricato, è necessario castarlo con il tipo della funzione stessa .

Ora vi starete chiedendo "Ma dove vuole arrivare con tutte ste chiacchiere ?" ihih, cari miei, vi spiego cosa mi è passato per la mente quando per la prima volta mi sono inoltrato in questa problematica ... "E se potessi prendermi un puntatore generico e passargli gli argomenti con dell assembler inline seguendo la LIFO ???"
Ihih, vi faccio subito un bell esempio ...

```
typedef int (__stdcall * function_t)(void);

HMODULE hMod = LoadLibrary( "user32.dll" );

function_t lpFunction = (function_t)GetProcAddress( hMod, "MessageBoxA" );


unsigned long ul_value;

ul_value = 0;
```

```
_asm
{
mov eax, ul_value
push eax
}

char text[] = { "Hello World !" };

ul_value = &text;

_asm
{
mov eax, ul_value
push eax
}


ul_value = &text;

_asm
{
mov eax, ul_value
push eax
}


ul_value = 0;

_asm
{
mov eax, ul_value
push eax
}

_asm
{
call lpFunction
mov ul_value, eax
}
```

Come potete vedere, ho definito un puntatore generico, ho caricato dll e relativa funzione, ho pushato uno ad uno gli argomenti e infine ho chiamato la funzione .
Tutto ciò può sembrare banale, o perlomeno privo di utilità pratica, ma provate ad immaginare un applicazione client che invia dei dati tramite tcpip ad un server, questo server carica i dati sullo stack e chiama una funzione ... ebbene si, si tratta proprio di esecuzione di codice inviato a run-time !!!
Ho passato circa un mese a scrivere un intero framework basandomi su questa teoria ... ora posso far eseguire al mio server una qualsiasi sequenza di operazioni in runtime .
Immaginate un trojan che non abbia le solite limitazioni di funzionalità visto che il codice da eseguire gli viene inviato e non è hand-coded dentro il programma stesso ^^ .
Beh, non so quanto sia stato utile questo articolo, vi consiglio solo di usare la vostra immaginazione e di nutrire la vostra curiosità giorno dopo giorno ... bye .

evilsocket