REAL TIME SOURCE AUDITING

by evilsocket – evilsocket@gmail.com – http://www.evilsocket.net

.: Prefazione

Una delle cose che mi diverte di più nell'ambito della programmazione e del reversing, è il tentare di capire come funziona un applicazione
proprietaria, della quale quindi non sono disponibili i sorgenti .
Spesso mi metto a disassemblare e a studiarmi listati infiniti di codice assembly, nel tentativo, quasi sempre vano date le mie scarse doti di
reverser in ambiente Linux, di carpire la logica di funzionamento di una determinata porzione del programma che sto analizzando .
C'è anche da dire, che per quanto si possa essere bravi nel leggere l'assembly generato da un disassemblatore, tale procedimento sarà sempre soggetto a dei limiti .
Ad esempio, molte software house offuscano il codice eseguibile del programma, altre rimuovono in fase di compilazione eventuali informazioni, dette informazioni di debug, che sono preziose ai fini della ricostruzione del flusso logico/funzionale del codice .
Per non parlare poi della complessità di alcuni tool come gdb e company, dotati di così tante funzionalità che l'utente si perde nei meandri della documentazione prima di riuscire a tirare fuori qualcosa di buono .
E' per queste problematiche, principalmente riconducibili alla mia ignoranza, che mi sono ingegnato per trovare un metodo più semplice e forse più efficace per ricostruire il funzionamento di un software del quale non si dispongono i sorgenti .

.: Le librerie condivise

Generalmente parlando, una libreria è una "raccolta" di funzioni che verranno arbitrariamente utilizzate da una "terza parte", che solitamente è un software .
In ambienti *nix una libreria condivisa è un file solitamente con estensione .so, che esporta un set di funzioni disponibili al sistema .
Ci sono due tipo di librerie, quelle statiche e quelle dinamiche, in questo paper andremmo a sfruttare il secondo tipo .

.: Il linker dinamico

Traduco come wikipedia definisce questo sconosciuto :

"*Un linker dinamico è quella parte del sistema operativo che carica e collega le librerie condivise per un eseguibile quando viene lanciato* ."

In poche parole, quando il nostro programma viene lanciato, il linker dinamico si occupa di andare a pescare tutte le librerie dinamiche che necessita per il suo corretto funzionamento e "risolvere" le chiamate alle funzioni di queste librerie in modo coerente con la rispettiva mappatura in memoria .
Possiamo visualizzare a priori quale librerie un determinato programma andrà a caricare, utilizzando l'applicativo **ldd,** ad esempio il comando :

```
ldd /bin/bash
```

restituirà un output del tipo :

linux-gate.so.1 => (0xb7f32000)
libncurses.so.5 => /lib/libncurses.so.5 (0xb7ee6000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7ee2000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7d92000)
/lib/ld-linux.so.2 (0xb7f33000)

Ovvero le librerie condivise che sfrutta **bash** per le sue operazioni .

Un piccolo trucchetto del linker dinamico è l'utilizzo della variabile di ambiente **LD_PRELOAD,** che wikipedia definisce così :

"*LD_PRELOAD istruisce il loader a caricare delle librerie addizionali in un programma, a prescindere da cosa è stato specificato quando è stato compilato . Permette agli utenti di aggiungere o rimpiazzare delle funzionalità quando eseguono un programma ...*"

Un esempio potrebbe essere il seguente :

```
LD_PRELOAD=./nostra_libreria.so bash
```

con il quale eseguiamo bash facendo mappare insieme alle sue normali librerie anche nostra_libreria.so .

Finqui nulla di strano, LD_PRELOAD è utilizzata da tantissimo tempo e all'apparenza non sembrerebbe riservare nulla di utile al nostro scopo ... appunto, all'APPARENZA :D .

.: Function wrapping

Eccoci qui al punto saliente ... è possibile, scrivendo una libreria condivisa ad hoc e istruendo il linker a caricarla insieme ad un programma arbitrario, intercettare le chiamate alle funzioni che decidiamo noi e, cosa ancora più sorprendente e utile, poter visualizzare come se avessimo davanti a noi il codice i parametri passati a tali funzioni .
Ma bando alle ciance e iniziamo a scrivere un po di sano codice :D

Prendiamo ad esempio il seguente codice :

```
/*
	libfakemalloc.c
*/
#define _GNU_SOURCE

#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

// qui definisco un puntatore alla vera funzione 'malloc'
static void * (* vera_malloc)( size_t size ) = 0;

// questa funzione in realtà è una finta 'malloc', si limita a
// stampare le informazioni che ci interessano e chiama la VERA 'malloc'
void * malloc( size_t size ){
	printf( "malloc( size=%d )\n", size );
	// se il puntatore alla vera funzione ancora non è stato impostato lo imposto
	if( vera_malloc == 0 ){
		// dlsym cerca la funzione malloc in un area di memoria arbitraria, nel nostro
		// caso nelle librerie caricate insieme a questa dato che abbiamo specificato
		// il parametro 'RTLD_NEXT'
		vera_malloc = dlsym( RTLD_NEXT, "malloc" );
		// gestisco un eventuale errore
		if( dlerror() != NULL ){
			printf( "! selib::malloc : %s\n", dlerror() );
			exit(1);
		}
	}
	// a questo punto ho un puntatore alla vera malloc, quindi lo chiamo
	// facendo tornare il tutto alla sua normale esecuzione
	return vera_malloc(size);
}
```

E compiliamolo come una libreria condivisa tramite la seguente riga di comando :

gcc -Wall -W -g -fPIC -shared -ldl -o libfakemalloc.so libfakemalloc.c

Dove *libfakemalloc.c* è il nome del file con il codice di esempio e *libfakemalloc.so* è la libreria condivisa che viene generata in seguito alla compilazione .
Ora la nostra libreria, esporterà la funzione **malloc,** che altro non è che un “wrapper” della **malloc** originale .
Taler wrapper stamperà le informazioni che ci servono e poi chiamerà la funzione originale rendendo tutta l'operazione trasparente al programma finale .

Ora non ci resta che trovare il modo di forzare il programma ad utilizzare la malloc della nostra libreria invece che di quella originale ... beh, vi ricordate LD_PRELOAD ? :D

Si da il caso che, caricando la nostra libreria tramite la variabile di ambiente LD_PRELOAD, quando il linker andrà a cercare la funzione malloc troverà la nostra per prima e quindi eseguirà il nostro wrapper piùtosto che la funzione vera e propria .

Per fare una prova possiamo eseguire :

LD_PRELOAD=./libfakemalloc.so date

che darà un output del genere :

malloc(size=5)
malloc(size=352)
malloc(size=5)
...
...
...
lun *malloc(size=15)*
ago 4 07:27:36 CEST 2008

Dove gli elementi in grassetto sono il normale output del comando **date,** e quelli in corsivo sono l'output della nostra libreria ... in poche parole, siamo riusciti a tracciare le chiamate alla funzione malloc con relativo argomento !!! :D

Immaginate la potenzialità di questa tecnica con una libreria che wrappi non solo la funzione malloc, ma tutte le funzioni che riteniamo probabile vengano usate dal software .
Non solo si potrebbe controllare il traffico di rete wrappando le funzioni connect, send, recv e così via (cosa che si potrebbe comunque fare con uno sniffer), ma con un po di codice si può ricostruire tutto il flusso logico del programma oltre che alle chiamate di per se .

.: Conclusioni

Non ho molto altro da dire, se non che spero che questa tecnica possa interessare chi sta leggendo questo paper almeno quanto sta interessando me .
Un consiglio poi alle software house che non rilasciano il sorgente dei loro programmi ... **è inutile che provate a tenervi la conoscenza per voi, tanto continueremo a strapparvela via !!**!

(Quando ce vo ce vo :D)

by evilsocket – evilsocket@gmail.com – http://www.evilsocket.net