Malware Analysis
Homework 3

Daniele Ferrarelli

1 Analisi iniziale

Si inizia l'analisi del programma tramite una analisi statica dell'eseguibile utilizzando Ghidra. Si può vedere tramite la lista delle dll utilizzate dal programma che si fa uso di una interfaccia gui. Per questo motivo si testa l'eseguibile su una macchina virtuale appositamente preparata. Si nota la somiglianza con l'homework precedente e si applicano metodi di analisi statica analoghi. Viene trovata una funzione con segnatura simile alla WinMain con al suo interno il message loop. Da questo si può evincere che l'applicazione, anche se non specifica esplicitamente di essere una applicazione gui, utilizza una interfaccia grafica. All'interno del WinMain è possibile trovare la struttura dati con al suo interno la WindowProc. Una volta fatta riconoscere questa funzione da Ghidra si passa ad analizzare i vari if che processano i messaggi all'interno della WindowProc. Si può procedere con metolodiga analoga all'homework precedente a trovare le funzioni:

- TimeoutProc: funzione invocata al timeout del timer
- ShutdownProc: funzione invocata per iniziare la procedura di spegnimento
- ButtonCommand: funzione che gestisce la gestione del bottone di start

Da questa prima analisi statica non si evince molto sul codice di sblocco. Tuttavia si possono notare una serie di funzioni che potrebbero interferire con l'uso del debugger. Una di queste funzioni che si trova prima del MessageLoop utilizza la funzione IsDebuggerPresent (check_debugger), mentre un'altra funzione utilizza le api LoadLibrary e GetProcAddress per mascherare il caricamento di una dll, cosa che potrebbe nascondere un altro meccanismo anti-debug. Quindi se si vuole procedere con l'analisi dinamica si necessita di eliminare i meccanismi anti-debug presenti all'interno della applicazione.

2 Analisi

All'interno della macchina virtuale si inizia ad utilizzare OllyDbg per analizzare dinamicamente il programma. Tuttavia per la presenza della funzione check_debugger il programma si chiuda con la chiamata a ExitProcess quando aperto tramite un debugger, altrimenti andrà a mostrare la finestra tramite la chiamata ShowWindow per poi ritornare. Si risolve questo problema in modo da saltare questo controllo e mostrare in ogni caso la finestra. Per risolverlo si sostituiscono le NOP alle istruzioni che fanno saltare verso la chiamata ad ExitProcess. Facendo così si può continuare con l'analisi dinamica

dell'applicazione. Si prova ad eseguire l'applicazione senza il debugger e si nota come venga mostrata una finestra di errore con scritto "InternalError" che fà chiudere l'applicazione. Questo fà pensare alla presenza di un checksum che viene impedisce all'applicazione di partire nel caso venga modificato l'eseguibile. Eseguendo il programma tramite debugger tuttavia si chiude senza mostrare le finestre della applicazione, questo sta ad indicare la presenza di altre istruzioni anti-debug.

Si continua l'analisi statica e si trova una funzione prima della gestione dei messaggi nel WinMain che potrebbe in qualche modo essere legata ai problemi incontrati nell'utilizzo del debugger, questa funzione viene rinominata a check_perb. Utilizzando il disassemblatore si può notare l'accesso al segmento FS con offset 0x30, questo può indicare l'accesso a delle informazione del processo. Tramite la documentazione di Windows si trova che all'offset 0x30 si trova il Process Enviroment Block (PEB), di questa struttura dati il terzo byte (BeingDebugger) indica se il processo corrente sta eseguendo con un debugger. Viene prelevato il 3 byte e si controlla se c'è un debugger presente, se sì si cambia il valore dell'uMsg e si fà in modo che non venga gestito. Questo fà in modo che il messaggio di creazione delle finestre non venga gestito, impedendo all'applicazione di creare e mostrare la gui. Per ovviare a questo problema si sostituisce con delle NOP la chiamata a questa funzione, facendo ciò la finestra dell'applicazione viene creta ed è possibile andare avanti con l'analisi dinamica del programma. Tuttavia ora l'esecuzione con OllyDbg porta ad un crash del debugger.

Tramite OllyDBG è possibile vedere che l'applicazione contiene qualche meccanismo anti debug. Tramite Ghidra esplorando le funzioni del WinMain è possibile vedere una funzione che contiene una chiamata alla funzione IsDeuggerPresent. Se il risultato è positivo si andrà a chiudere l'applicazione. Modificando la funzione di jump che in caso positivo chiama la funzione exit si può impedire che il programma termini quando si utilizza un debugger. Risolvendo questo problema tuttavia quando si esegue l'applicazione viene mostrata una message box che mostra internal error. Si continua ad esplorare per possibili misuri anti debug. Tramite l'esecuzione nel debugger si può vedere come la handle della window presente in ShowWindow sia nulla. Questo indica che la funzione CreateWindow fallisce essendo il suo valore di ritorno 0.

Viene incontrata la funzione (somethin_strange_2) che fa operazioni su creazione di file. Tramite l'esecuzione in ollyDBG è possibile vedere come vengano impostati i parametri di queste funzioni. Si fanno operazioni sulla grandezza del file.

Analizando la WindowProc si può vedere come venga eseguita una funzione (something_strange_with_peb) che lavora sul PEB del programma accedendo al FS:[30]. All'interno della PEB è presente il BYTE being debugged, questo può essere un indizio che qualche operazione anti-debug viene eseguita. Viene prelevato il valore di BeingDebugged, come parametro a questa funzione viene passato il codice del messaggio. Tramite debugger è possibile vedere come venga cambiato il valore del messaggio all'interno dell'applicazione, portando ad eseguire il gestore di default di windows, quindi il messaggio di CREATE non viene eseguito. Viene eliminata la chiamata alla funzione che fa queste operazioni, e si può vedere come viene creata la finestra dell'applicazione. Tuttavia questa finestra fa crashare il debugger.

Si decide di analizzare il resto della funzione WindowProc, si nota la pre-

senza di una funzione (debug_crasher) che può risultare interessante. Si piazza un breakpoint all'inizio di questa funzione per analizzarla tramite analisi dinamica. Eseguendo le istruzioni passo passo è possibile notare come tramite una serie di istruzioni assembler è possibile chiamare la funzione OutputDebugStringApA preparata in precedenza. In questo modo si sfrutta un bug di OllyDBG che lo fa crashare. Per risolvere questo problema si modifica il codice assembler per far in modo che al posto di chiamare la funzione della DLL problematica si fa un RETN e si ritorna al resto della WindowProc. Si sostituiscono i byte nell'indirizzo 004006 con l'istruzione RETN. Si può vedere da debugger che il blocco di codice che viene eseguito dopo la jump costruisce la stringa %s..... per poi chiamare il OutputDebugStringApA. Quindi rimuovendo il codice non si dovrebbe rimuovere funzionalità dall'applicazione. Questo codice viene offuscato tramite una serie di XOR e ROR. Tuttavia utilizzando il debugger è possibile vedere questo codice deoffuscato. Facendo in questo modo è possibile continuare l'analisi dinamica senza che il debugger subisca crash. Se si prova ad eseguire con il debugger o senza il programma si vede una message box di errore che fa chiudere l'applicazione.

Si cerca tra le funzioni presenti quando gli utilizzi di stringhe simili alla funzione di debug. Le funzioni per mostrare la message box di errore nella funzione di timeout. Questa funzione trovata quando si inizializza il timer. Viene trovata la funzione show_debug_error_box che presenta certi impedimenti al disassemblaggio che vengono risolti manualmente. Questa funzione andrà a mostrare un messaggio di errore. Da altre funzioni si può ipotizzare che venga creata una segnatura del programma in modo che non venga eseguito se vengono modificati dei byte. Per ovviare a questo problema si elimina la funzione che andrà a chiuedere il programma e mostrare la message box. Ora si prova ad eseguire il programma e si può vedere come anche essendo modificato non crashi.

Si testa il programma inserendo un codice casuale e si esegue tramite debugger per vedere come viene confrontato questo codice. Si può notare come venga di nuovo applicato un metodo di offuscamento per nascondere il codice presente nella sezione data. Si vede come viene deoffuscato e si procede ad analizzare istruzione per istruzione questo codice. Il programma resituisce l'errore senza tuttavia controllare il codice (?) questo può essere dovuto ad un problema ancora non risolto. Si suppone che la generazione di un hash della memoria del programma non sia effettivamente risolta e si continua ad analizzare la generazione dell'hash.

9 caratteri

Dalla funzione per la ricerca del codice si può ottenere che il codice è lungo 9 caratteri. Successivamente si fanno operazioni di xor su dati messi sullo stack precedentemente. Osservando il comportamento di queste operazioni di XOR possiamo dedurre il codice di sblocco. Applicando di nuovo lo XOR con il dato del compare essendo una operazione idempotente si riottiene il valore del codice oscurato.

3rNesT0?!

All'interno dell'applicazione per dare fastidio al disassemblaggio. Questi impedimenti sono stati risolti andando a decompilare e ricompilare manualmente.

La funzione (something_important) chiama la load di kernel32.dll, tramite l'utilizzo di OllyDBG. GetProcAddress con parametro OutputDebugStringApA, è un altro meccanismo anti dbg. Vado a vedere chi chiama questa funzione e provo a disattivarla modificando l'eseguibile.

Sono presenti delle sezioni aggiuntive .tls, .CRS