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 (load_library_api) 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 Risoluzione meccanismi anti-debug

Nelle sezioni successive si vanno ad eliminare i meccanismi anti-debug dell'applicazione che impediscono l'utilizzo di OllyDbg per facilitare la ricerca del codice di sblocco. In molte delle funzioni che vengono descritte successivamente si fa ampio uso di meccanismi per rendere più difficile l'analisi statica, tuttavia questi meccanismi non vengono sempre citati nelle sezioni successive. Uno di questi è nascondere il codice eseguibile nella sezione data. Un altro è quello di inserire finti JUMP confizionali, utilizzando variabili nella sezione .bss, ma mai riscritte, è possibile ingannare Ghidra e mascherare delle istruzioni.

2.1 IsDebuggerPresent

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 fa chiudere l'applicazione. Questo fa 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.

2.2 Process Enviroment Block

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 fa in modo che non venga gestito. Questo fa 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.

2.3 OutputDebugStringApA

Un possibile motivo per il crash del debugger può essere dovuto ad un bug di OllyDbg. Se un processo fa una chiamata alla funzione OutputDebugStringApA con parametro una serie di %s il debugger crasha. Non vengono fatte chiamate esplicite a questa funzione, questo fa pensare che il caricamento di questa api venga offuscato. Come notato in precedenza è presente una funzione che chiama LoadLibrary e GetProcAddress, funzioni che permettono di caricare una dll. Si và ad analizzare tramite analisi statica la funzione load_library_api, si può vedere come vengano offuscati i parametri di LoadLibrary e GetProcAddress, questo fa pensare che faccia parte di un meccanismo anti-debug. Tramite analisi statica si posiziona un breakpoint in questa funzioni per vedere i parametri non offuscati per le funzioni precedenti. Si può vedere che sullo stack venga caricata la

stringa "kernel32.dll" e passata come parametro a LoadLibrary, successivamente si vede sullo stack la stringa "OutputDebugStringApA" che viene passata come parametro a GetProcAddress. Infine si salva il puntatore a questa funzione in memoria. Si vede se sono presenti riferimenti in cui questa funzione viene chiamata, tuttavia non sono presenti, questo fa pensare che i riferimenti siano nascosti in qualche modo. Si nota all'interno della WindowProc la chiamata ad una funzione, analizzandola si può vedere al suo interno le istruzioni XOR e ROR che vengono applicate in un ciclo for ad un area di memoria (0x00405020 - 004005f). Successivamente si va a fare una jump all'inizio di questa area di memoria modificata in precedenza. Questo fa pensare che all'interno di questa area sia presente codice offuscato, infatti utilizzando Ghidra non è possibile disassemblare queste istruzioni. Questa area di memoria viene deoffuscata facendo lo XOR con 0x89a3fa2b e ROR 0x9 per ogni dword di questa area di memoria. Si passa ad utilizzare OllyDbg e si mette un breakpoint al momento del jump. In questo modo è possibile analizzare l'area di memoria quando è stata già deoffuscata, continuando a seguire il flusso di esecuzione verrà chiamata la funzione OutputDebugStringApA. Analizzando il codice seguente non sembra che sia presente codice funzionale al funzionamento dell'applicazione. Si rimuove il JMP inserendo una RETN, facendo così è possibile continuare l'analisi dinamica senza che il debugger subisca un crash.

2.4 Error Box

3 Ricerca del codice si sblocco

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