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 metodologia analoga all'homework precedente a trovare le funzioni:

- TimeoutProc: funzione invocata al timeout del timer
- shutdown_function: funzione invocata per iniziare la procedura di spegnimento
- button_command: 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.

```
typedef struct _PEB {
BYTE                          Reserved1[2];
BYTE                          BeingDebugged;
BYTE                          Reserved2[1];
PVOID                         Reserved3[2];
PPEB_LDR_DATA                 Ldr;
PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
PVOID                         Reserved4[3];
PVOID                         AtlThunkSListPtr;
PVOID                         Reserved5;
ULONG                         Reserved6;
PVOID                         Reserved7;
ULONG                         Reserved8;
ULONG                         AtlThunkSListPtr32;
PVOID                         Reserved9[45];
BYTE                          Reserved10[96];
PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
BYTE                          Reserved11[128];
```

```
PVOID                           Reserved12[1];
ULONG                           SessionId;
} PEB, *PPEB;
```

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 per il 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

Sin dalla prima modifica dell'eseguibile il programma risulta essere non funzionante. Questo perché l'eseguibile mostra una finestra di errore che fa chiudere l'applicazione, sia in caso di esecuzione con debugger che senza. Questo può indicare la presenza di un meccanismo che controlla il checksum del programma con un valore salvato in memoria. Se questo checksum è diverso mostra quell'errore. SI può notare tramite analisi statica la presenza di una funzione nel WinMain prima che venga creata la finestra (load_file_mapping). Questa funzione andrà, tramite una serie di chiamate a delle dll, a ottenere informazioni sull'eseguibile corrente, per poi creare un File Mapping che viene salvato in memoria. Da questo file mapping è possibile accedere a zone di memoria del processo corrente.

Dai riferimenti del file mapping è possibile trovare la funzione (create_checksum) che fa uso di questo dato. Questa funzione viene chiamata all'interno della gestione del messaggio di creazione nella WindowProc. Al suo interno per ogni dword del programma si applica uno XOR per creare un hash dell'eseguibile. In tal modo si genera una checksum del programma che probabilemente viene controllata in un'altra zona per verificare se ci sono state modifiche al programma. La finestra di errore viene mostrata dopo un secondo dall'avvio del timer, questo fa pensare che si trova all'interno della TimerProc. Si va ad analizzare questa funzione e si trova una chiamata ad un'altra funzione (show_debug_error_box). Al suo interno si trovano riferimenti alle stringhe mostrate nella finestra di errore, questo fa supporre che al suo interno vengano fatti i controlli del checksum, inoltre la funzione presenta meccanismi per confondere il disassemblatore di Ghidra. Si possono notare riferimenti a zone di memoria dove potrebbe essere memorizzato il checksum precedentemente generato e quello originale del programma. I riferimenti al checksum generato in precedenza vengono offuscati manipolando i puntatori in memoria, mentre quello originale di trova all'indirizzo DAT_004050a4. Dopo aver mostrato il message box di errore si andrà a chiudere il programma. Per evitare questo problema si elimina la chiamata alla funzione show_debug_error_box sostituendola con delle NOP. Ora il programma non si chiude ed è possibile continuare l'analisi statica per trovare il codice.

3 Ricerca del codice si sblocco

Avendo risolto i meccanismi che impedivano l'analisi dinamica del programma si passa alla ricerca del codice si sblocco. All'interno della shutdown_function è presente la chiamata alla funzione GetDlgItemText con indice 5, indice che indica la edit box contenente il codice di sblocco, per ottenere il contenuto della edit box. Dopo aver ottenuto la stringa contenente il codice, si passa ad una serie di istruzioni simili a quelle sopracitate per deoffuscare una zona di memoria. Si prende una zona di memoria e si applica uno XOR con 0x89a3fa2b e ROR 0x9 per ogni dword al suo interno, successivamente si va a chiamare una funzione al suo interno. Per vedere cosa si fa all'interno di questa zona si fa partire il debugger e si inserisce un codice di sblocco casuale. Utilizzando un breakpoint prima della chiamata a questa funzione offuscata è possibile analizzare passo passo le istruzioni al suo interno. Dai controlli presenti al suo interno si può evincere che il codice è di 9 caratteri. Riprovando la procedura precedente con

un codice con il corretto numero di caratteri si passa ad una serie di XOR e CMP per ogni carattere del codice inserito, questo fa pensare che il controllo del codice viene eseguito qui. Si modifica il codice dell'eseguibile in memory con OllyDbg in modo da evitare che quando il codice sia sbagliato si esca dalla funzione. In questo modo è possibile provare diversi codici senza che l'applicazione termini. Vengono caricati sullo stack la serie di byte composta da 0x3F, 0x28, 0x2F, 0xA5, 0x5D, 0x47, 0x3D, 0x4F, 0x3F. Successivamente si prende un carattere dal codice di sblocco inserito e si applica lo XOR con uno dei byte precedenti per poi confrontarlo con un byte. La serie di byte che viene usato nel confronto è 0x0C, 0x5A, 0x61, 0xC0, 0x2E, 0x13, 0x0D, 0x70, 0x1E. Essendo lo XOR una operazione idempontente se si applica con la serie di byte caricati sullo stack con la serie di byte usati per i confronti è possibile ottenere il valore dei byte del codice di sblocco necessari per superare i vari controlli. Applicando questo procedimento si trova il codice di sblocco "3rNesT0?!". Cambiando i byte nella zona di memoria dove viene inserito il codice di sblocco preso dalla edit box con quelli trovati con l'operazione in precedenza il sistema si spegnerà. Viene provato il codice trovato anche all'interno dell'appicazione originale e si osserva lo stesso comportamento, questo indica che il codice di sblocco è stato trovato con successo.