Malware Analysis
Homework 3

Daniele Ferrarelli

1 Analisi iniziale

Si inizia l'analisi del programma tramite un'analisi statica dell'eseguibile utilizzando Ghidra. Si può vedere dalla lista delle dll utilizzate dal programma che si fa uso di una interfaccia GUI. Per questo motivo si testa l'applicazione 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. 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 e vengono trovate le funzioni:

- TimeoutProc: funzione invocata al timeout del timer
- initialize: funzione che prepara variabili globali per il funzionamento dell'applicazione
- initialize_timer: funzione che inizializza il timer e la sua procedura di timeout
- shutdown_function: funzione invocata per iniziare la procedura di spegnimento
- button_command: funzione che gestisce l'avvio del timer e il bottone

Da questa prima analisi statica non si evince molto sul codice di sblocco, quindi si necessita di utilizzare anche l'analisi dinamica. 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 che possono 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. 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, che impedisce a Ghidra di riconoscere le istruzioni automaticamente. Un altro è quello di inserire finti JUMP condizionali, utilizzando variabili nella sezione .bss ma mai riscritte (sempre con valore 0), è possibile ingannare Ghidra e mascherare il flusso delle istruzioni. Inoltre vengono inseriti dei byte corrispondenti ad istruzioni di CALL che interferiscono con il disassemblatore di Ghidra.

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 chiude con la chiamata a ExitProcess quando aperto tramite un debugger. Se non è presente il debugger verrà mostrata la finestra con la chiamata ShowWindow. Questa funzione utilizzerà la chiamata a IsDebuggerPresent per controllare la presenza del debugger. Si risolve questo problema in modo da saltare il controllo ed eseguire in ogni caso ShowWindow. Per risolverlo si sostituisce al JNZ (0x004024a9) delle NOP, in modo che anche se il debugger è presente si passa per ShowWindow e si ritorna senza andare ad eseguire ExitProcess. Facendo così si può continuare con l'analisi dinamica dell'applicazione. Si prova ad eseguire il programma senza il debugger e si nota come viene mostrata una finestra di errore con scritto "InternalError" che fa chiudere l'applicazione. Questo fa pensare alla presenza di un checksum che 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 altri meccanismi anti-debug.

2.2 Process Enviroment Block

Si continua l'analisi statica e si trova una funzione prima della gestione dei messaggi nella WindowProc, che potrebbe in qualche modo essere legata ai problemi incontrati nell'utilizzo del debugger, questa funzione viene rinominata a check_peb. 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), il terzo byte di questa struttura dati (BeingDebugged) indica se il processo corrente sta eseguendo con un debugger.

```
typedef struct _PEB {
BYTE                          Reserved1[2];
BYTE                          BeingDebugged; <- Controllato dalla funzione
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 all'interno degli if della WindowProc. 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 creata ed è possibile andare avanti con l'analisi dinamica del pro-

gramma. 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 subisce un crash. Non vengono trovati riferimenti 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 funzione di 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 dinamica si posiziona un breakpoint in questa funzione per vedere i parametri non offuscati per le funzioni precedenti. Si può vedere che viene caricata sullo stack 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 viene salvato il puntatore a questa funzione in memoria. Si ricercano riferimenti al puntatore della funzione ma non vengono trovati, questo fa pensare che le chiamate siano nascoste in qualche modo. Si nota, all'interno della gestione del messaggio WM_SHOW, la chiamata ad una funzione (0x00404000, rinominata ad anti_debugger); 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 viene fatto un 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 dal programma applicando, per ogni dword, uno XOR con 0x89a3fa2b e ROR 0x9 in modo da ottenere il valore deoffuscato di questa serie di istruzioni. Si passa ad utilizzare OllyDbg e si mette un breakpoint al momento del jump in questa area di memoria. 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 l'applicazione. Si rimuove il JMP all'area di memoria con la chiamata ad OutputDebugStringApA 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'applicazione mostra una finestra di errore che fa chiudere il processo, 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. 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 sul processo corrente, per poi creare un File Mapping che viene salvato in memoria, inoltre viene ottenuta anche la grandezza del file in byte. Da questo file mapping è possibile vedere l'eseguibile corrispondente al 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 si calcolerà l'hash del programma con una serie di XOR. Utilizzando il file mapping precedentemente ottenuto si andrà a prendere le dword del programma per poi applicare uno XOR tra di loro. Nel calcolo dell'hash tuttavia vengono ignorati i primi 1024 byte dell'eseguibile, zona di memoria corrispondente all'header del programma (trovata tramite Memory Map su Ghidra). In tal modo si genera una checksum del programma che probabilmente verrà controllato in un'altra zona per verificare se ci sono state modifiche al programma, questo checksum viene salvato come variabile globale all'indirizzo 0x004070e0.

La finestra di errore viene mostrata dopo un secondo dall'avvio del timer, questo fa pensare che si trova all'interno della TimeoutProc, essendo eseguita ogni secondo. 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 interferire con il disassemblatore di Ghidra. Si possono notare riferimenti a zone di memoria dove potrebbero essere memorizzati il checksum precedentemente generato e quello originale del programma. I riferimenti al checksum creato in precedenza vengono offuscati manipolando i puntatori in memoria, mentre quello originale di trova all'indirizzo DAT_004050a4. Posizionando un breakpoint in accesso alla memoria nella zona dove è mantenuto l'hash generato è possibile vedere che la funzione show_debug_error_box accede a questa area per fare il confronto. Se il valore dell'hash non è uguale a quello originale viene mostrato il message box di errore e 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 dinamica per trovare il codice.

3 Ricerca del codice di 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 GetDlgItemTextA con indice 5 per ottenere il contenuto della edit box. L'indice 5 identifica la edit box contenente il codice di sblocco. Dopo aver ottenuto la stringa contenente il codice, vengono eseguite una serie di istruzioni simili a quelle sopracitate per deoffuscare una zona di memoria. Viene presa una zona di memoria e viene applicato uno XOR con 0x89a3fa2b e ROR 0x9 per ogni dword al suo interno, successivamente viene chiamata una funzione al suo interno. Per vedere cosa viene fatto 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. All'interno di questa funzione viene confrontato il risultato di GetDlgItemTextA, che indica il numero di caratteri ottenuti dalla edit box, con il numero 9. Da questo si può dedurre che il numero di caratteri del codice di sblocco è 9. 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 modificano le istruzioni del processo in memory con OllyDbg in modo da ricominciare il confronto dei caratteri quando un carattere è errato, facendo in modo di non chiudere il processo, questo viene fatto per facilitare la fase di ricerca del codice. Questo si fa andando a sostituire le istruzioni con indirizzo 0x004050FB (ADD ESP, 1C) e 0x004050FE (RETN) con NOP, in questo modo le istruzioni di JNZ, che fanno terminare l'applicazione in caso di carattere errato, riportano all'inizio dei vari confronti. In tal modo è possibile provare diversi codici senza che il processo termini.

Continuando l'analisi si può vedere che vengono caricati sullo stack la serie di byte composta da 0x3F, 0x28, 0x2F, 0xA5, 0x5D, 0x47, 0x3D, 0x4F, 0x3F. Successivamente viene preso 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 l'operazione inversa di se stessa, se si applica con la serie di byte caricati sullo stack e 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 ottiene la serie di byte corrispondente al codice di sblocco "3rNesT0?!". Tramite il debugger viene modificata l'area di memoria contenente il codice si sblocco con quello appena ottenuto, successivamente si fà riprendere l'esecuzione del processo e si può vedere che andrà ad avviare lo spegnimento Windows.

Viene testato il codice trovato anche con l'eseguibile originale, questo porterà allo spegnimento della macchina virtuale, ciò indica che il codice di sblocco è stato trovato con successo.