1 Introduzione ed obiettivi

Il seguente documento riassume le informazioni raccolte riguardo l'eseguibile "hw2.exe" a seguito del reversing del codice binario del file.
Tale eseguibile è una applicazione Windows basata su GUI, la cui funzionalità è quella di spegnere la macchina allo scadere di un timeout.
Dalla 1 si nota che vi è un edit box, all'interno del quale viene richiesto un codice per poter usare in maniera efficace l'applicativo, che senza tale codice non spegnerà la macchina. Se infatti si prova ad impostare un timeout e a farlo scadere, senza aver inserito il corretto codice di sblocco, apparirà una finestra pop-up che avverte che il codice è errato e l'applicazione termina senza espletare il suo compito, come mostrato in 2.
L'obiettivo delle attività di reverse code engineering svolte è stato quindi quello di **trovare il codice di sblocco** che rende funzionale il programma, oltre a cercare di carpire informazioni generali riguardo il funzionamento e le strutture dati del programma.

2 Ricerca del WinMain

Come prima cosa, è stata cercata la funzione **WinMain**, che è il punto di ingresso delle applicazioni Windows basate su GUI. Ogni WinMain è costruito intorno al **Message Loop**, che esegue in in ciclo while senza fine le seguenti funzioni

- **GetMessage**
- **TranslateMessage**
- **DispatchMessage**

quindi per trovare la **WinMain** è stata consultato l'albero delle chiamate a funzione di Ghidra, a partire dall'entry point: il ramo di interesse dell'albero è mostrato in 3.
Una volta individuata la **WinMain**, ne è stato analizzato il codice tramite la funzionalità disassemblatore di Ghidra:

- alla **RegisterClassExA**, che è l'API che registra la classe finestra per successive chiamate, si passa una struttura di dati di cui uno fra i campi più importanti è quello dove viene messo il puntatore alla **Window Procedure**, ovvero la routine di callback che verrà invocata per la gestione di ogni messaggio ricevuto dall'applicazione.
  Tale routine viene tradotta da Ghidra come una label, quindi è stata trasformata in funzione manualmente;
- viene creata l'applicazione finestra mediante l'API **CreateWindowEx**;
- si passa ad eseguire il Message Loop, all'interno di un while(true)

3 Analisi della Window Procedure

A questo punto, è stata analizzata la Window Procedure per vedere quali tipi di messaggi venivano gestiti. Dopo aver ricevuto un nuovo messaggio, infatti, ne viene controllato il codice in una serie di if e la procedura gestisce esplicitamente i messaggi di

- `WM_SIZE`, ricevuti dalla finestra quando cambia dimensione;
- `WM_CREATE`, tipo di messaggio che arriva all'atto della creazione della finestra;

- `WM_DESTROY`, messaggio ricevuto quando si termina l'applicazione;
- `WM_PAINT`, messaggio ricevuto nel caso in cui sia richiesto di ridisegnare la finestra;
- `WM_COMMAND`, messaggio scaturito, ad esempio, dopo l'interazione con una finestra figlia.

La gestione di tutti gli altri tipi di messaggio è delegato ad una procedura di default.

3.1 Blocco di codice per la gestione di WM_COMMAND

All'inizio del blocco di codice Assembly per la gestione del messaggio WM_COMMAND vi è l'interazione con il registro EBX, che precedentemente è stato impostato al valore del parametro di input wParam

- il contenuto di EBX viene shiftato a destra;
- si verifica se il valore ottenuto è pari a 0, facendo un test su BX che è dato dai 16 bit meno significativi di EBX;
- se il risultato non è 0, si salta ad una label di uscita.

Vi è poi l'uso dei registri EDI ed EAX:

- risalendo alle istruzioni precedenti, EDI contiene il valore di `lParam`;
- in EAX viene caricato il valore di una variabile locale, rinominata come `user_data_local_b0`;
- viene fatta una CMP fra EDI ed il valore all'indirizzo dato da EAX + offset 184

Uno spiazzamento di un certo ammontare di byte rispetto ad una base, fa pensare all'utilizzo di qualche struttura dati.
Occorre cercare di capire come viene impostata la variabile user_data_local_b0: consultando i riferimenti di Ghidra, si può notare che la scrittura avviene in un singolo punto, in cui si imposta il contenuto di EAX con il risultato ottenuto dalla **GetWindowLong**.
Tale API:

- restituisce informazioni legate ad una finestra, assieme ad una *dword* che permette di accedere a della memoria aggiuntiva associata alla finestra stessa;
- il primo parametro è l'handle alla finestra;
- il secondo parametro dipende da cosa si vuole richiedere, in questo caso c'è il codice -21 che corrisponde a `GWL_USERDATA`, quindi si richiede di accedere ai dati definiti dall'utente.

La funzione restituisce dati inseriti dal programmatore e tale inserimento avviene all'interno della **SetWindowLong**.
È stato visto quindi dove viene chiamata questa API, accedendo all'elenco delle funzioni tramite Ghidra: la chiamata avviene durante la creazione della finestra e fra i parametri della **SetWindowLong**, il 3° viene impostato ad `lParam`.
I parametri `wParam` ed `lParam` dipendono da cosa avviene nel **WinMain**, quindi è stata nuovamente analizzata tale funzione più nel dettaglio

3.2 Funzione init_ds

All'interno del WinMain, il parametro `lpParam` è il risultato di una funzione **FUN_00401aab** che restituisce l'indirizzo di una variabile globale.
Mediante decompilatore, è stato analizzato il contenuto di questa funzione in cui è possibile vedere che vengono manipolate diverse variabili globali, andandone ad impostare il valore, per poi ritornare il puntatore alla variabile `DAT_00406010`. La funzione è stata rinominata **init_ds** ed è probabile il riferimento ritornato punti ad una struct.
Il parametro `lpParam` viene a sua volta usato come parametro all'interno dell'API **CreateWindowExA**: consultando la documentazione di questa funzione,si scopre che nel messaggio di `WM_CREATE` tale parametro punta al campo `lpCreateParams`, che è un puntatore ai dati definiti dall'utente; `lpCreateParams` è a sua volta un campo della struttura `CREATESTRUCT`.
La struttura dati risultante dalla init_ds viene quindi passata alla Window Procedure e salvata nella memoria extra della finestra una volta, ovvero nel blocco di codice per la gestione del messaggio di `WM_CREATE`, per poi essere ri-acceduta tramite l'API **GetWindowLong** nella gestione dei messaggi successivi.
Si ricorda che, all'interno del WM_COMMAND:

- viene confrontato EDI col valore di `user_data_local_b0` ad offset 184;
- andando a ritroso, è possibile vedere che dopo la **GetWindowLong** il valore di ritorno viene messo in `user_data_local_b0`, quindi avviene una copia in locale dell'indirizzo della struttura di dati;

Viene quindi denominata la struttura dati restituita dalla **init_ds** come `app_struct`, andandola a definire in Ghidra fra i dati del programma.
Sono stati poi definiti i campi, partendo da una taglia complessiva della struttura di 188 byte, in quanto il primo campo noto è ad offset 184, di 4 byte di tipo UINT.
Tornando alla **init_ds**, dove venivano inizializzati i campi della struct, si nota che

- viene copiato uno 0 ad offset 0;
- viene copiato il valore 1000 a partire offset 8;
- viene copiato il valore 1800 a partire offset 12;
- vi sono poi due chiamate ad una funzione che, al suo interno, invoca la **svnprintf**
  - nella prima invocazione, viene passato come primo parametro il campo della struct ad offset 24, chiedendo di printare 128 byte. Quindi a tale offset c'è un array di 128 byte;
  - lo stesso accade nella seconda invocazione, dove l'offset della struct è in questo caso a 152, ed il buffer è di 16 byte;
- viene poi copiato 0 ad offset 16;
- infine, viene copiato il valore del primo parametro, che è una variabile globale, ad offset 20.

3.3 Gestione del WM_CREATE

Si torna alla gestione del messaggio `WM_CREATE`:

- viene messo il valore puntato da EDI in ESI, per poi caricare in EDI il parametro `hwnd`, che è l'handle alla finestra;

- successivamente, si copia il puntatore all'handle accedendo ad ESI ad offset 168, quindi è stato inserito questo campo della struttura, in quanto come detto ESI contiene la **app_struct** definita nella WinMain;
- c'è poi la creazione degli edit box dell'applicazione. Ci si aspetta che vengano creati 5 box, 3 per i campi del timeout, 1 per il codice di sblocco ed un ultimo per il bottone;
- la creazione delle finestre figlie avviene in un ciclo while, dove ogni chiamata alla **CreateWindowEx** restituisce l'handle della finestra creata, tale handle viene salvato ogni volta ad indirizzo dato da ESI + EBX*4 + 168;

Successivamente, vengono creati le ultime due finestre, ovvero l'edit box che riceverà il codice di sblocco ed il bottone per avviare o fermare il timeout.
Siccome tutte le copie nella struttura dati avvengono ogni 4 byte ed i tipi di dato sono tutti `HWND`, è stato inserito nella struttura un'array di 6 elementi per contenere tutti gli handles alle finestre.
Il blocco di codice termina facendo due call:

- la prima call, dal decompilatore, mette in evidenza che vengono fatti dei conti che usano dei campi della struct;
- la seconda funzione chiama al suo interno l'API **SetTimer**

3.3.1 Gestione del timer

All'interno della prima funzione, vengono svolti alcuni conti accedendo ai campi della struttura di dati precedentemente individuati.
I conti svoli servono per aggiornare il timeout, andando a calcolare il numero di minuti, ore e giorni rimanenti, per poi impostare tali valori nelle 3 finestre edit box tramite l'API **SetDlgItemInt**; questa funzione è stata quindi rinominata come **count_rem_time**.
Dai riferimenti su Ghidra, è possibile vedere che tale funzione viene invocata anche dalla funzione che si invoca nel blocco di codice per la gestione del comando `WM_COMMAND`: all'interno di tale funzione, vengono proprio svolti i calcoli per decrementare il campo della struttura dati che mantiene il timeout, per poi passare a chiamare la **count_rem_timer**.

3.3.2 Funzione SetTimer

L'ultima funzione chiamata dal ramo di gestione della `WM_CREATE` è quella che chiama la **SetTimer**, ovvero la funzione di libreria che imposta il timeout.
La funzione prende 4 parametri ed il valore di ritorno viene inserito nel campo della struttura **app_struct** ad offset 8, che è quindi l'identificatore del timer.
È stata analizzata la Timer Proc, ovvero la funzione di callback registrata per la gestione del timer: al suo interno

- si incrementa il valore del tempo corrente;
- si verifica se il timer è ancora attivo, ovvero il campo della struttura dati ad offset 16 è diverso da 0. In tal caso, si chiama la **count_rem_time**;
- nel caso in cui lo shutdown time sia minore del tempo attuale, si effettua una CALL al campo della struttura dati ad offset 20.

Ad offset 20 è stato messo il valore della variabile `DAT_00403000`, che quindi corrisponde ad un function pointer.
Ghidra non riesce però a capire che sotto quel dato è presente del codice da disassemblare, quindi il procedimento è stato fatto manualmente:

- andando all'indirizzo del `DAT_00403000`, evidenziando la prima istruzione, è stato usato il comando "Disassemble";
- in questo modo, Ghidra ha riconosciuto e mostrato le istruzioni Assembly che componevano la routine;
- a questo punto, è stato possibile definire la funzione ed analizzare anche il risultato del decompilatore.

4 Ricerca del codice

Il parametro passato alla funzione è l'indirizzo della stessa struct, che viene usata per accedere all'array di handles, definito a partire da offset 168.
Infatti, c'è una chiamata alla API **GetDlgItemTextA** che recupera il testo contenuto in un dialogue box

- il primo parametro è l'handle alla finestra;
- il secondo parametro è 5, ovvero l'indice dell'edit box che contiene il codice di sblocco;
- il terzo parametro è il buffer che riceverà il contenuto dell'edit box, ovvero il parametro di stack `lpString`;
- il 4° parametro è il numero di byte massimo da copiare.

Il valore di ritorno della funzione è il numero di byte effettivamente copiati, escludendo il byte '\0' e tale risultato viene salvato nel parametro `UVar3`.
In seguito, a `bVar8` viene assegnato il risultato di un AND logico del seguente predicato:

$$\text{DAT\_004060d0} == 0 \;\&\&\; \text{UVar3} == 9$$

Questo permette di intuire che la lunghezza attesa per la stringa contenente il codice di sblocco è pari a 9.
Dal decompilato, si evince un predicato booleano in cui vengono effettuati diversi controlli fra caratteri, come mostrato in 4.
La stringa di sblocco sembra quindi essere

3RnESt0!?

infatti inserendola all'interno dell'applicazione questa riavvia il sistema operativo allo scadere del timeout.

4.1 Risoluzione degli indirizzi per le sub-routine

All'interno del codice Assembly, vi sono diverse CALL a subroutine che hanno indirizzi non presenti in memoria, infatti Ghidra se ci cerca di saltare all'indirizzo di tali funzioni risponde col messaggio di errore "Address not found in program memory: "; in 5, viene mostrato un esempio di come Ghidra interpreta le istruzioni errate e del codice decompilato risultante.
Consultando il listato Assembly, si può notare che tutte le CALL a queste presunte sub-routine scaturiscono dal fatto che si cerca di saltare ad una certa label, ma il salto effettivo che viene tradotto da Ghidra finisce nel mezzo dell'istruzione e quindi viene deassemblato in maniera non

corretta.
Per risolvere questi problemi e cercare di ottenere il reale codice decompilato, sono stati effettuati i seguenti passi:

- è stata selezionata l'istruzione di CALL ed è stata applicata la funzionalità di Ghidra "Clear flow and repair";
- a questo punto, il JMP alla label viene correttamente tradotto da Ghidra, e vengono prodotte due istruzioni macchina sottostanti, corrispondenti ai codici operativi E8 e 63;
- queste due istruzioni sono state patchate all'interno di Ghidra, sostituendovi una NOP (codice operativo 90).

A seguito della risoluzione dei problemi, il codice decompilato ottenuto viene riassunto in 6

4.2 Analisi completa della funzione di spegnimento

Dopo aver reso il codice decompilato più leggibile, è stato possibile analizzare le altre API chiamate all'interno della funzione che gestisce lo spegnimento della macchina, in modo da completare l'analisi del programma. Nel caso in cui il codice di sblocco inserito nel box sia corretto, la funzione chiama le seguenti API:

- **GetCurrentProcess**, che restituisce uno pseudo-handle al processo corrente;
- **OpenProcessToken**, che restituisce l'access token del processo;
- **LookupPrivilegeValueA**, per ottenere l'identificativo locale dell'utente ed a cui viene passato come secondo parametro la stringa "SeShutdownPrivilege", ovvero viene richiesto di poter effettuare lo shut down del sistema;
- c'è poi la **AdjustTokenPrivileges**, che permette di attivare i privilegi richiesti, associandoli allo specifico token;
- infine, c'è la chiamata alla **ExitWindowsEx**, che effettua il log off dell'utente e spegne il sistema.

Nel caso in cui il codice di sblocco non sia corretto, viene invocata una funzione, denominata con **show_err_win**, in cui si apre la finestra pop-up mostrata in 2

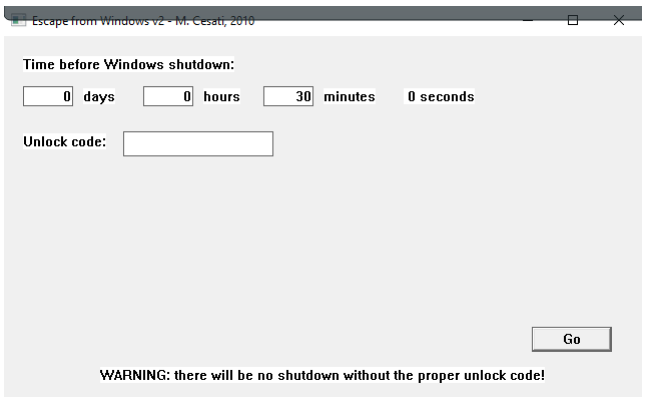

Figure 1: Schermata iniziale dell'applicativo

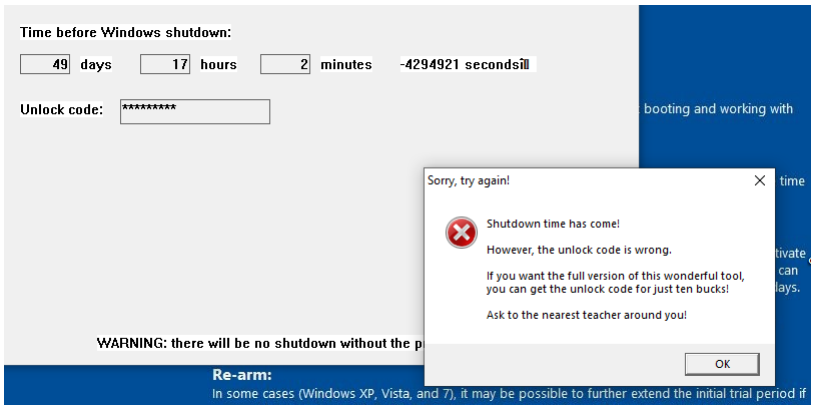

Figure 2: Messaggio di errore

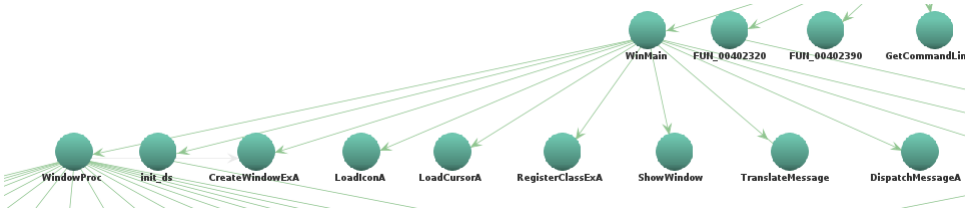

Figure 3: Ramo dell'albero dove viene chiamato il WinMain

```
if ((((((bVar8) && (cStack54 == '3')) && (cStack53 == 'R')) &&
     ((cStack52 == 'n' && (cStack51 == 'E')))) &&
    ((cStack50 == 'S' && ((cStack49 == 't' && ((char)uStack48 == '0')))))) &&
   ((uStack48._1_1_ == '!' && (uStack48._2_1_ == '?')))) {
```

Figure 4: Controllo del valore recuperato dalla GetDlgItemText

```
00403055 85 c0          TEST        EAX,EAX
00403057 74 02          JZ          LAB_00403059+2
                    LAB_00403059+2                          XREF[0,1]:   00403057(j)
00403059 e8 63 83       CALL        SUB_0a3ab3c1
         fa 09
0040305e 74 15          JZ          LAB_00403075
```

```
  func_0xf4e0192a();
                /* WARNING: Bad instruc ncating control flow here */
  halt_baddata();
}
uVar4 = func_0x00402568();
```

Character
Length: 1

Figure 5: Istruzioni non disassemblate correttamente da Ghidra

```
void __cdecl FUN_00403000(int param_1)

{
  UINT UVar1;
  HANDLE ProcessHandle;
  BOOL BVar2;
  DWORD DVar3;
  char cStack54;
  char cStack53;
  char cStack52;
  char cStack51;
  char cStack50;
  char cStack49;
  char cStack48;
  char cStack47;
  char cStack46;
  _TOKEN_PRIVILEGES _Stack24;
  HANDLE pvStack8;

  *(undefined4 *)(param_1 + 0x10) = 0;
  UVar1 = GetDlgItemTextA(*(HWND *)(param_1 + 0xa8),5,&cStack54,0x1e);
  if ((((((UVar1 == 9) && (cStack54 == '3')) && (cStack53 == 'R')) &&
       ((cStack52 == 'n' && (cStack51 == 'E')))) &&
      ((cStack50 == 'S' && ((cStack49 == 't' && (cStack48 == '0')))))) &&
     ((cStack47 == '!' && (cStack46 == '?')))) {
    ProcessHandle = GetCurrentProcess();
    BVar2 = OpenProcessToken(ProcessHandle,0x28,&pvStack8);
    if (BVar2 == 0) {
      PostQuitMessage(0);
    }
    LookupPrivilegeValueA((LPCSTR)0x0,"SeShutdownPrivilege",(PLUID)_Stack24.Privileges);
    _Stack24.PrivilegeCount = 1;
    _Stack24.Privileges[0].Attributes = 2;
    AdjustTokenPrivileges
              (pvStack8,0,(PTOKEN_PRIVILEGES)&_Stack24,0,(PTOKEN_PRIVILEGES)0x0,(PDWORD)0x0);
    DVar3 = GetLastError();
    if (DVar3 == 0) {
      ExitWindowsEx(5,0);
    }
    PostQuitMessage(0);
  }
  else {
    show_err_win();
  }
  return;
}
```

Figure 6: Risultato della decompilazione