Relazione homework 3

Luca Mastrobattista
Matricola: 0292461

Indice

1 Traccia dell'homework

1.1 Testo

Analizzare con Ghidra e OllyDbg il programma eseguibile hw3.exe contenuto nell'archivio hw3.zip. Determinare il codice di sblocco che rende funzionale il programma e riassumere in un documento tutte le informazioni acquisite, la metodologia adottata ed i passi logici deduttivi utilizzati nel lavoro di analisi.

1.2 Scadenza

Due settimane dalla data di assegnazione del lavoro: 13/12/2021.

1.3 Consegna

Documento in formato PDF inviato come allegato ad un messaggio di posta elettronica all'indirizzo del docente ("<cognome>@uniroma2.it"), con subject: "[AMW21] HW3: <matricola studente>"

2 Ambiente di lavoro

Il file eseguibile è stato caricato su Ghidra istallato su un sistema operativo Linux.
L'ambiente controllato di utilizzo è un sistema operativo Windows 10 virtualizzato con il software *VirtualBox*, in cui sono istallati gli strumenti di monitoraggio.

3 Metodologia

3.1 Informazioni note a priori

Dalla traccia dell'*homework* si deduce che l'eseguibile risulta *bloccato* in qualche modo, e solo con un particolare codice si riesce a farlo funzionare correttamente.

3.2 Finalizzazione dell'obiettivo

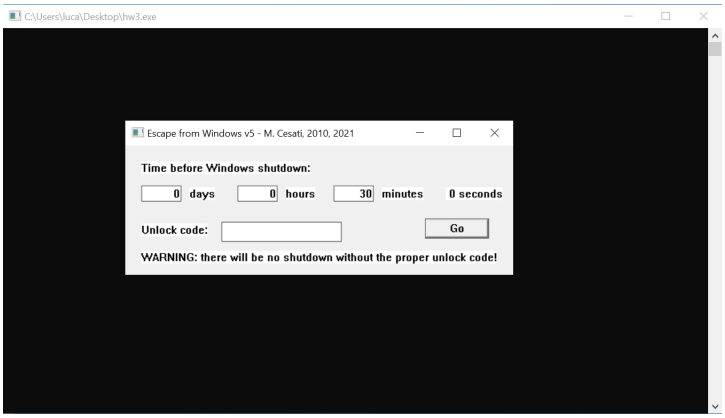
L'analisi dell'applicazione ha come obiettivo quello di individuare il codice di sblocco.

3.3 Ottenimento del codice macchina

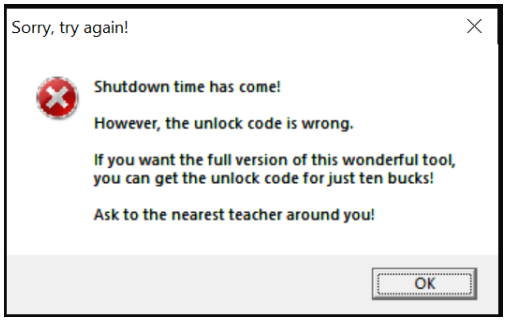
Codice macchina fornito dal professore.

3.4 Osservazione del funzionamento

L'applicazione, una volta avviata, crea una finestra con un *countdown* impostato a 30 minuti per default, ma questo tempo è modificabile. Premendo sul pulsante *Go*, il conto alla rovescia inizia. È presente una casella di testo in cui inserire il codice da trovare.



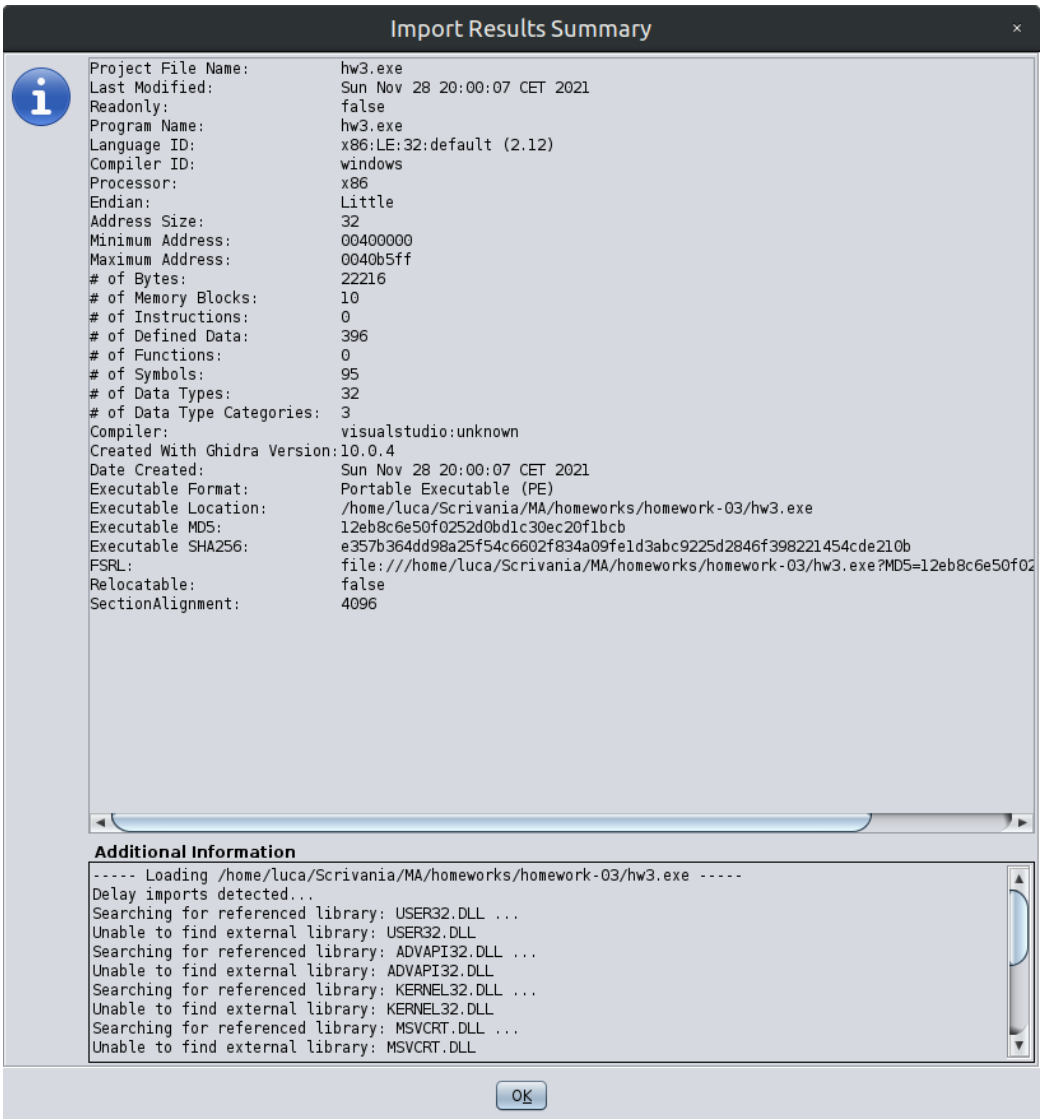
Al termine del *countdown*, se il codice inserito non è corretto, compare una finestra di avviso di codice errato:



3.5 Disassemblaggio del codice macchina

Lo strumento che si è utilizzato è il software *Ghidra*.

3.5.1 Riepilogo risultati dell'import



3.5.2 Informazioni aggiuntive

```
----- Loading /home/luca/Scrivania/MA/homeworks/homework-03/hw3.exe
-----
Delay imports detected...
Searching for referenced library: USER32.DLL ...
Unable to find external library: USER32.DLL
Searching for referenced library: ADVAPI32.DLL ...
Unable to find external library: ADVAPI32.DLL
```

```
Searching for referenced library: KERNEL32.DLL ...
Unable to find external library: KERNEL32.DLL
Searching for referenced library: MSVCRT.DLL ...
Unable to find external library: MSVCRT.DLL
Searching for referenced library: GDI32.DLL ...
Unable to find external library: GDI32.DLL
Finished importing referenced libraries for: hw3.exe
  [ADVAPI32.DLL] -> not found
  [GDI32.DLL] -> not found
  [KERNEL32.DLL] -> not found
  [MSVCRT.DLL] -> not found
  [USER32.DLL] -> not found
```

4 Analisi

Con analisi statica di base si nota subito la presenza della sezione *.tls*: sono infatti presenti due *tls_ callback*, ma dalla loro analisi statica avanzata non risultano presenti meccanismi di antidebugging al loro interno. Ricercando le funzioni presenti, invece, è presente la funzione `IsDebuggerPresent` che viene invocata in `FUN_004024a0_isDebuggerPresent`. Il controllo che viene fatto è, come sempre per questa invocazione, facilmente superabile: si esegue quindi una prima *patch* per sostituire questa `CALL` con una operazione di `XOR EAX, EAX`, in modo che il successivo `JMP` venga sempre preso e l'esecuzione continui come se il debugger non ci fosse.
Caricando il nuovo eseguibile e provando a lanciarlo senza *breakpoints*, il programma non termina ma non viene mostrata nessuna finestra: l'esecuzione rimane nello stato *running*. Si procede ad analizzare su *Ghidra* la funzione `FUN_00401de0_window_procedure`. Qui, la prima invocazione interessante è `FUN_00401dc0_check_debugger_FS[30]`, invocata a prescindere dal valore del messaggio ricevuto. Riceve in *input* l'indirizzo del messaggio `uMsg` che la `window_procedure` si prepara a gestire e ne modifica il contenuto incrementandolo di una quantità pari al valore dell'offset 2 di `FS:[30h]`. In questa zona di memoria è memorizzato il valore 1 se il processo corrente è eseguito sotto un debugger, 0 altrimenti. Quindi, se si prova a fare analisi dinamica avanzata, viene incrementato di 1 il valore del messaggio della `window_procedure` e il normale flusso della gestione dei messaggi viene alterato: si sta interferendo con il normale funzionamento dell'applicazione. Si prova quindi ad eseguire una seconda *patch* che sostitisca l'invocazione di questa funzione con una istruzione di `NOP`.
Il nuovo eseguibile sembra creare la finestra, ma senza le varie caselle di testo al suo interno e, pochi instanti dopo, *OllyDB* viene chiuso: sembra il classico funzionamento di `OutputDebugString`.

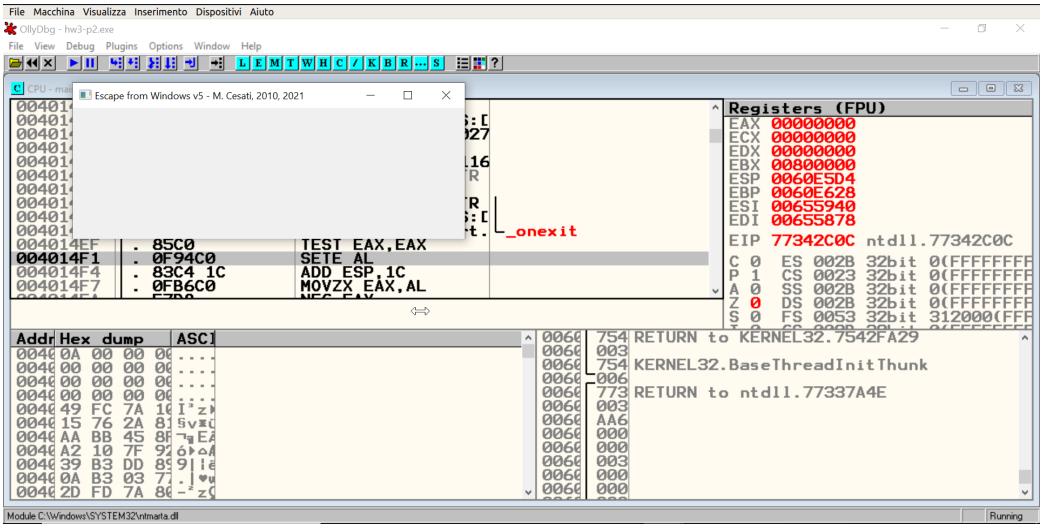


Immagine catturata pochi istanti prima del crash di OllyDB.

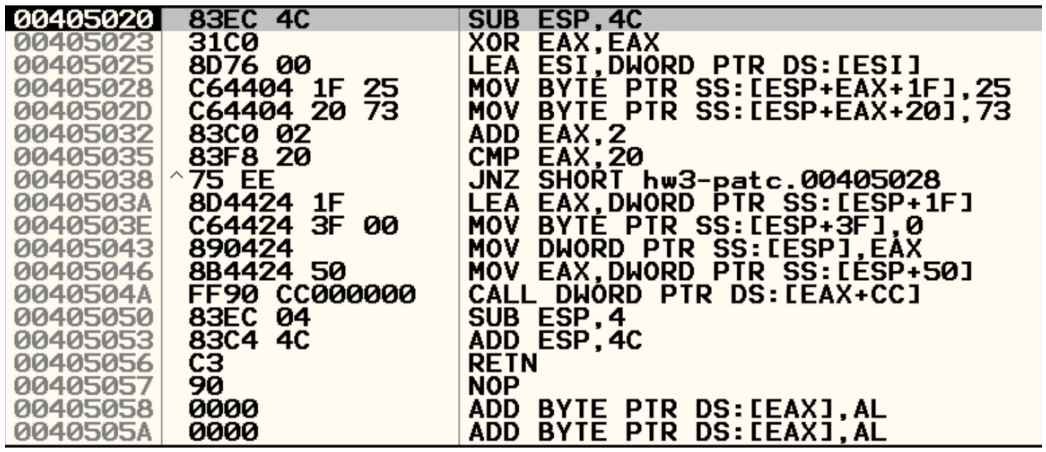
Si torna quindi su *Ghidra* alla ricerca di riferimenti all'*API* `OutputDebugString` nella lista degli *imports*, ma non vengono trovati. Si nota però la presenza delle *API* `LoadLibrary` e `GetProcAddress`, anche se sembrano non essere invocate nel codice.
Si deve capire dove potrebbe avvenire l'invocazione di `OutputDebugString`: la finestra viene mostrata ma è incompleta e perciò si procede controllando i blocchi di `WM_CREATE`, `WM_SIZE` e `WM_PAINT` della `window_procedure`. Seguendo questa strada, vengono messi 3 *breakpoints* su *OllyDB* ai seguenti indirizzi:

1. `0x401eb0`: prima istruzione del blocco `WM_CREATE`
2. `0x401fa8`: prima istruzione del blocco `WM_SIZE`
3. `0x402108`: prima istruzione del blocco `WM_PAINT`

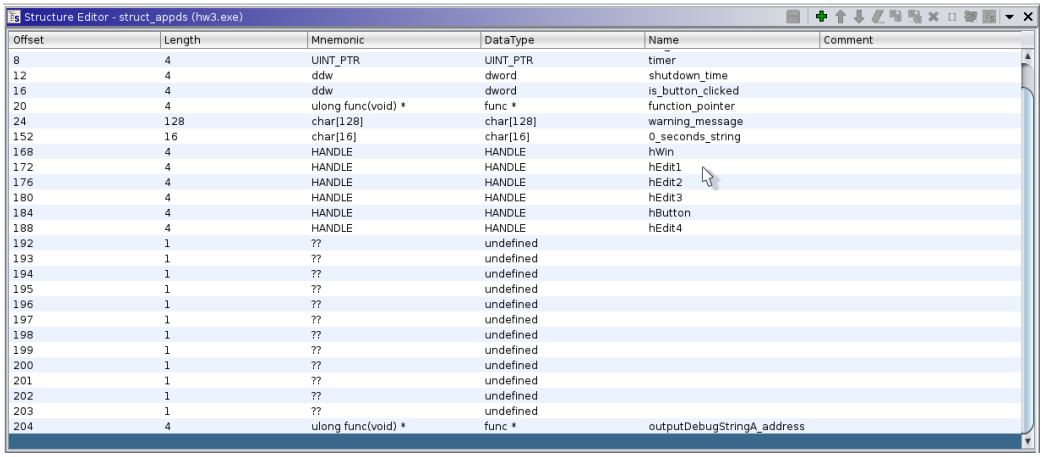
Premendo `F9`, si raggiunge il primo *breakpoint*. Premendo di nuovo `F9`, si raggiunge il secondo e la finestra viene mostrata: si deduce che l'invocazione non è all'interno del blocco `WM_CREATE`. Premendo di nuovo `F9`, il terzo *breakpoint* non viene raggiunto e l'esecuzione termina: si controlla quindi il blocco `WM_SIZE`, procedendo in parallelo con *Ghidra* e *OllyDB*.
Al termine di questo blocco c'è una funzione che *OllyDB* riesce a raggiungere: `FUN_00404000_invoke_outputDebugString`. Qui si accede all'area di memoria all'indirizzo `0x405020` e si decodificano i byte presenti. La decodifica avviene con le seguenti istruzioni:

```
XOR EDX, EDX
label:
MOV EAX, dword ptr [EDX*0x4 + 00405020]
XOR EAX, 0x89a3fa2b
ROR EAX, 0x9
MOV dword ptr [EDX*0x4 + 00405020], EAX
ADD EDX, 0x1
CMP EDX, 0xe
JNZ label
```

Dopo aver decodificato quel blocco di codice, si passa ad eseguirlo con una `JMP`. Il codice decodificato è il seguente:

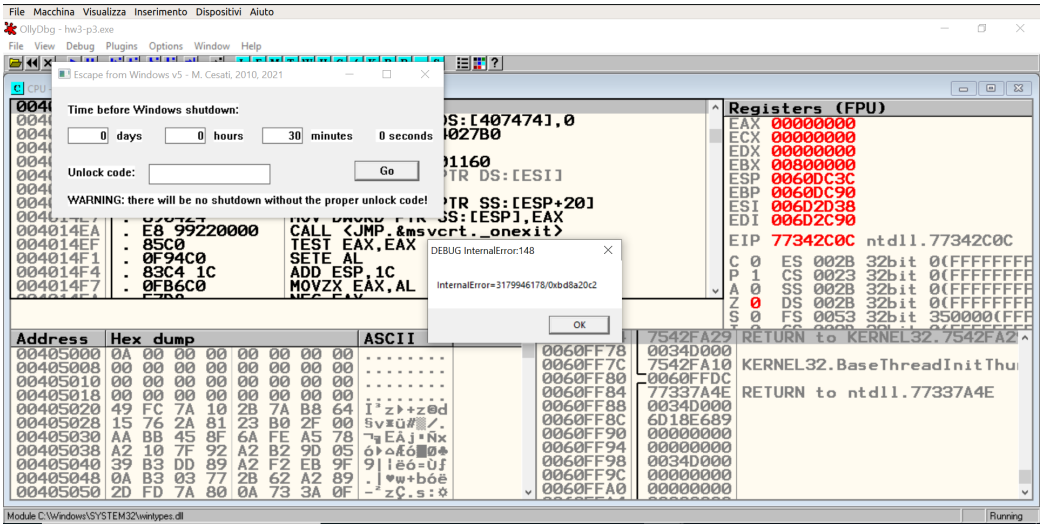


Questo set di istruzioni genera sullo stack una stringa composta da 16 `"%s"` e invoca la funzione contenuta all'indirizzo `0x407020 + 0xcc`, che sarà sicuramente `OutputDebugString`. Su *Ghidra* ci sono molte referenze all'indirizzo `0x407020`, tra cui alcune scritture nella funzione `FUN_00401830_init_appds`: è l'indirizzo della struttura dati che gestisce l'applicazione, già vista nell'*homework2*. Questa struttura dati viene qui estesa e, in particolare, all'offset 204 verrà memorizzato l'indirizzo di `OutputDebugString`, in base a quanto detto precedentemente:


Structure Editor - struct_appds (hw3.exe)

Offset	Length	Mnemonic	DataType	Name	Comment
8	4	UINT_PTR	UINT_PTR	timer	
12	4	ddw	dword	shutdown_time	
16	4	ddw	dword	is_button_clicked	
20	4	ulong func(void) *	func *	function_pointer	
24	128	char[128]	char[128]	warning_message	
152	16	char[16]	char[16]	0_seconds_string	
168	4	HANDLE	HANDLE	hWin	
172	4	HANDLE	HANDLE	hEdit1	
176	4	HANDLE	HANDLE	hEdit2	
180	4	HANDLE	HANDLE	hEdit3	
184	4	HANDLE	HANDLE	hButton	
188	4	HANDLE	HANDLE	hEdit4	
192	1	??	undefined		
193	1	??	undefined		
194	1	??	undefined		
195	1	??	undefined		
196	1	??	undefined		
197	1	??	undefined		
198	1	??	undefined		
199	1	??	undefined		
200	1	??	undefined		
201	1	??	undefined		
202	1	??	undefined		
203	1	??	undefined		
204	4	ulong func(void) *	func *	outputDebugStringA_address	

Questo nuovo campo della struttura dati viene inizializzato proprio nella funzione `FUN_00401830_init_appds`, impostandolo al valore di ritorno di `FUN_004016f0_retrieve_OutputDebugStringAddress`. Sono presenti meccanismi di anti-disassemblaggio ma, alla fine, viene caricata la libreria `kernel32.dll` per recuperare l'indirizzo di `OutputDebugString`; questo indirizzo viene infine restituito. Le stringhe usate vengono costruite sullo stack un carattere alla volta in modo da nosconderle all'analisi statica di base.
Continuiamo quindi l'analisi applicando una terza *patch* all'eseguibile, sostituendo l'invocazione della funzione `FUN_00404000_invoke_outputDebugString` all'interno del blocco di gestione di `WM_SIZE` della `window_procedure` con una istruzione di `NOP`.
Con questa *patch*, *OllyDB* non crasha, ma i meccanismi di anti-debugging non sono finiti: viene infatti mostrata una finestra di errore.

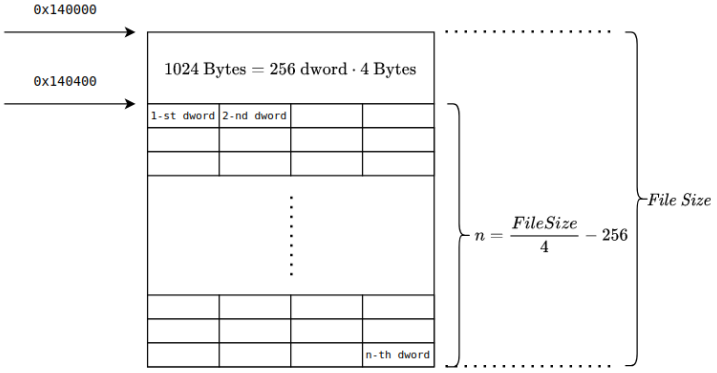


A questo punto rimangono solo due funzioni sospette che non sono ancora state analizzate:

- `FUN_00401560_create_and_load_file_mapping` invocata nel `FUN_004024e0_win_main`. Qui viene aperto l'eseguibile con `CreateFileA` per poi creare un suo *file mapping object* con `CreateFileMapping`. Il valore di ritorno di quest'ultima invocazione è passato alla funzione `MapViewOfFile` per caricare il *file mapping object* nell'*address space* del processo. Il valore restituito è l'indirizzo in cui viene caricato e viene memorizzato all'*offset* 196 della struttura dati `struct_appds`; quel campo viene rinonimato `starting_file_view_address`. In questa funzione, viene inizializzato anche un altro campo della struttura: il campo all'*offset* 200 viene impostato pari al valore del campo `nFileSizeLow` della struttura dati `BY_HANDLE_FILE_INFORMATION` passata all'invocazione `GetFileInformationByHandle`. Questo campo, nel nostro caso, è in grado di memorizzare l'intera taglia dell'eseguibile. Il campo della struttura dati viene rinonimato in `nFileSizeLow`.

- la funzione `FUN_004016b0_compute_checksum` invocata nel blocco `WM_CREATE` in `FUN_00401de0_window_procedure`. In questa funzione si parte dall'indirizzo memorizzato nel campo `starting_file_view_address` e ci si sposta di 1024 bytes: si ricade all'inizio della sezione `.text`. Si memorizza in `EAX` il valore del campo `nFileSizeLow` della struttura dati `struct_appds`, lo si divide per 4 e si sottrae `0x100`: si sta contando il numero di `dword` presenti del

file, escludendo quelle contenute nei primi 1024 bytes. Poi parte un ciclo in cui si calcola lo `XOR` di tutte queste `dword`: si sta praticamente calcolando un **checksum**. Il valore finale viene salvato all'offset 192 della struttura dati; quel campo viene quindi chiamato `checksum`.



Solo le n parole rappresentate vengono incluse nel calcolo del checksum.

Structure Editor - struct_appds (hw3.exe)

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	ddw	dword	elapsed_seconds	
4	4	ddw	dword	init_1000	
8	4	UINT_PTR	UINT_PTR	timer	
12	4	ddw	dword	shutdown_time	
16	4	ddw	dword	is_button_clicked	
20	4	ulong func(void) *	func *	function_pointer	
24	128	char[128]	char[128]	warning_message	
152	16	char[16]	char[16]	0_seconds_string	
168	4	HANDLE	HANDLE	hWin	
172	4	HANDLE	HANDLE	hEdit1	
176	4	HANDLE	HANDLE	hEdit2	
180	4	HANDLE	HANDLE	hEdit3	
184	4	HANDLE	HANDLE	hButton	
188	4	HANDLE	HANDLE	hEdit4	
192	4	ddw	dword	checksum	
196	4	addr	pointer	starting_file_view_address	
200	4	ddw	dword	nFileSizeLow	
204	4	ulong func(void) *	func *	outputDebugStringA_address	

Struttura dati completa

Avendo *patch-ato* il file, il `checksum` calcolato sarà sicuramente diverso da quello atteso, ovunque questo venga controllato. Per questo motivo, si recupera il file originale per cercare di ottenere il valore valido e applicare in seguito una nuova *patch*. Nonostante i *breakpoints* software modifichino il codice sostituendo il codice operativo dell'istruzione in cui fermarsi con quello di `INT3`, si possono comunque usare in questo caso: infatti la modifica avviene al codice caricato in memoria, mentre il `checksum` è calcolato sul *file mapping object* ottenuto leggendo il file sul disco. Sono inoltre necessari soltanto 2 *breakpoints*:

- in `FUN_00401dc0_check_debugger_FS[30]`, all'indirizzo `0x401dce`. Dopo le istruzioni:

```
MOV EDX, dword ptr FS:[0x30]
MOV EDX, dword ptr [EDX + 0x2]
```

in `EDX` ci sarà il valore `FFFF0001` e dovrà essere cambiato in `FFFF0000`.

- in `FUN_004016b0_compute_checksum`, all'indirizzo `0x4016e2`: qui si potrà leggere in `ECX` il `checksum` valido.

Il valore letto è `74ee8f1f`. Si procede ricaricando l'eseguibile con le 3 *patches* fatte per poi aggiungerne una quarta: si sostituiscono le prime istruzioni di `FUN_004016b0_compute_checksum` con:

```
MOV dword ptr [4070e0], 74eebf1f
RETN
```

Questa *patch*, insieme a tutte le altre, sblocca l'esecuzione dell'applicazione con *OllyDB*.

Si procede quindi mettendo un *breakpoint* sulla prima istruzione della funzione che viene invocata al termine del timer, memorizzata nel campo `function_pointer` della struttura `struct_appds`. Questo campo viene inizializzato nella funzione `FUN_00401830_init_appds` con l'indirizzo della funzione `FUN_004040e0_timer_proc_end`. Anche qui sono presenti molti meccanismi anti-disassemblaggio, quindi è conveniente procedere in parallelo con *OllyDB* e *Ghidra*. All'indirizzo `0x4041f7` c'è l'invocazione dell'*API* `GetDlgItemTextA`, utilizzata per recuperare il codice inserito dall'utente. Il buffer specificato è il registro `ESI`, mentre il numero di bytes letti verrà inserito in `EAX`. L'esecuzione continua poi effettuando la stessa operazione di decodifica vista nel caso di `FUN_00404000_invoke_outputDebugString`, accedendo però all'area di memoria in `0x4050c0`:

```
XOR ECX, ECX
label:
MOV EDX, dword ptr [ECX*0x4 + 004050c0]
XOR EDX, 0x89a3fa2b
ROR EDX, 0x9
MOV dword ptr [ECX*0x4 + 004050c0], EDX
ADD ECX, 0x1
CMP ECX, 0x34
JNZ label
```

A questo punto, bisogna proseguire usando soltanto *OllyDB*: le istruzioni su *Ghidra* sono codificate. Dopo l'operazione di decodifica, l'esecuzione continua con una `CALL` a quell'indirizzo. Qui vengono fatti dei controlli in modo sequenziale: prima si controlla se la lunghezza della stringa inserita dall'utente sia uguale a 9, poi ne vengono controllati i caratteri uno alla volta, terminando al primo esito errato. Tuttavia, questi 9 controlli non sono fatti *in chiaro*: ogni carattere del codice dell'utente viene messo in `XOR` con uno specifico byte e si controlla che il risultato dell'operazione sia uguale a un certo risultato atteso. In particolare, la lista dei byte con cui il codice dell'utente è messo in `XOR` è:

```
{ 0x3f, 0x28, 0x2f, 0xa5, 0x5d 0x47, 0x3d, 0x4f, 0x3f }
```

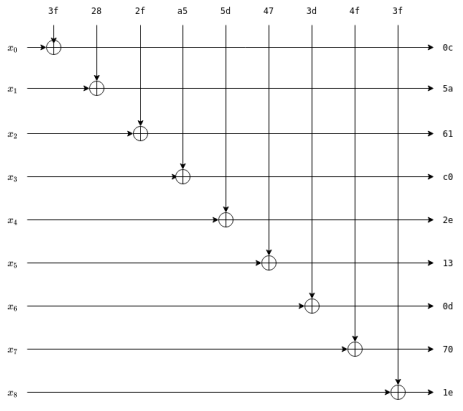
Chiamiamo questi byte k_i $\forall i \in [0..8]$.
La lista degli output attesi, invece, è:

```
{ 0x0c, 0x5a, 0x61, 0xc0, 0x2e 0x13, 0x0d, 0x70, 0x1e }
```

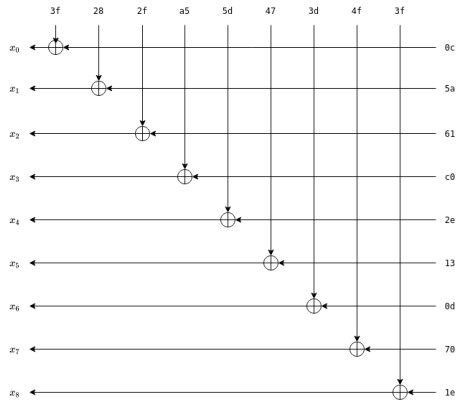
Chiamiamo questi byte y_i $\forall i \in [0..8]$.
Chiamiamo inoltre i caratteri della password attesa x_i $\forall i \in [0..8]$.

Le precedenti informazioni bastano a ricostruire la password attesa: per le proprietà dello `XOR`, infatti, si ha che:

$$y_i = x_i \oplus k_i \implies y_i \oplus k_i = x_i \oplus k_i \oplus k_i = x_i \oplus 0 = x_i \implies y_i \oplus k_i = x_i$$



Schema con cui funziona l'applicazione.



Invertendo la direzione, si può risalire alla password che sblocca l'applicazione.

5 Verifica

Per verificare che le informazioni raccolte siano giuste, è stato creato un semplice script *python* per ricostruire la password:

```
def decrypt_byte(hex1, hex2):
    tmp = int(hex1, 16) ^ int(hex2,16)
    return chr(tmp)

if __name__ == '__main__':
    hex_key = ['3f', '28', '2f', 'a5', '5d', '47',
            '3d', '4f', '3f']

    expected_outputs = ['0c', '5a', '61', 'c0', '2e',
            '13', '0d', '70', '1e']
    s = ''
    for k_i,y_i in zip(hex_key, expected_outputs):
        s += decrypt_byte(k_i, y_i)
    print('The password is:', s)
```

La password che viene stampata dallo script è **3rNesT0?!**. Inserendola nell'apposita *edit text*, la macchina virtuale *Windows* viene effettivamente arrestata al termine del *countdown*.