

GAME: Midtown Madness (Chicago Edition)
Protection: Safedisc 1.07
Author: Luca D'Amico - V1.0 - 7 Aprile 2022

Cosa ci serve:

- Windows XP VM (ho usato VMware)
- Process Hacker 2
- x64dbg (x32dbg)
- Disco di gioco originale (abbiamo bisogno del disco ORIGINALE)

Prima di iniziare:

Abbiamo bisogno del disco di gioco originale (o una copia 1:1), altrimenti ci mancherà la chiave di decifratura. Presto vedremo come funziona Safedisc e come è possibile estrarre l'eseguibile decifrato dalla memoria e sistemare la import table.

I giochi protetti da Safedisc non funzioneranno in Windows Vista o versioni più recenti a causa di una vulnerabilità di sicurezza scoperta nel suo driver. Una volta rimosso Safedisc, sarà possibile avviare il gioco anche su Windows 11.

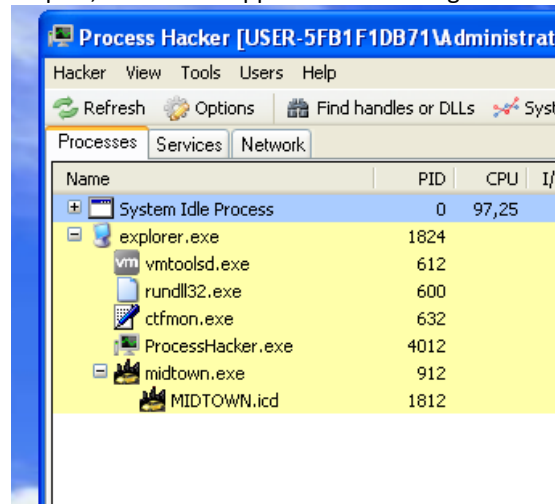
Tenete a mente che state per iniziare un combattimento contro una protezione di grado commerciale, che anche se ormai obsoleta, al tempo è stata impiegata per rallentare i cracker di talento di quel periodo. Quindi non perdere le speranze se non capisci qualcosa anche dopo un paio di letture. Fidati, se spendi il giusto quantitativo di tempo, sarai in grado di comprendere tutto.

Iniziamo:

Prima di tutto installiamo il gioco selezionando Installazione Completa.

Avviamo il gioco ed osserviamo cosa succede. Mettiamo il disco nel lettore e clicchiamo l'icona del gioco. Safedisc inizierà a fare le sue "verifiche" sul disco e dopo circa 20-30 secondi il gioco partirà.

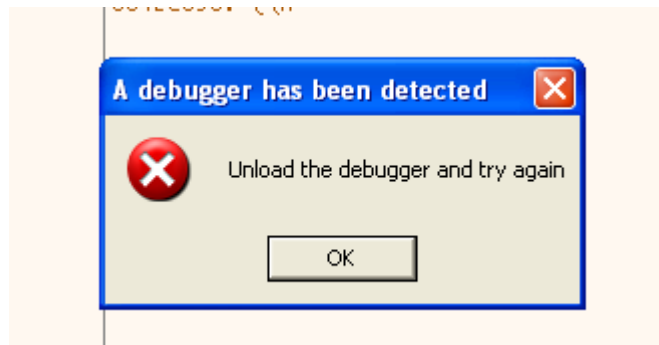
Appena siamo al menu principale, mettiamo l'applicazione in background e apriamo Process Hacker 2:



Ok, possiamo vedere che midtown.exe ha creato un nuovo processo, MIDTOWN.ICD.

Cosa significa? Semplice: midtown.exe è solo un loader (ed ecco spiegato perché è così piccolo), e MIDTOWN.ICD è l'eseguibile reale del gioco, ma è crittato (se volete verificare, provate ad aprirlo in un PE editor e noterete che almeno il segmento .text è crittato).

È giunto il tempo di aprire il loader (midtown.exe) in x32dbg. Se proviamo a avviarlo otterremo questa bella schermata di errore:



Ok, il loader riconosce che stiamo tentando di avviarlo in un debugger e si rifiuta di continuare. Dopo una breve ricerca su google, ho trovato questo sito molto ricco di informazioni, con una lista dei metodi più usati per riconoscere la presenza di un debugger:

<https://anti-debug.checkpoint.com/techniques/debug-flags.html>

Ho provato tutte le tecniche proposte e ho capito che queste sono quelle usate:

- IsDebuggerPresent API
- Controllo manuale del flag BeingDebugged nel PEB
- NtQueryInformationProcess() API

Possiamo facilmente patchare e disattivare i primi due controlli, ma per il terzo dobbiamo prestare attenzione poiché questa API è usata anche per ottenere varie informazioni dal processo, oltre che verificare la presenza di un debugger.

Dobbiamo patchare il buffer che questa funzione riempie, SOLO quando il secondo parametro (ProcessInformationClass) è 0x7 (ProcessDebugPort). Se vuoi maggiori informazioni riguardo questa API, puoi leggere la relativa pagina sulla documentazione di Microsoft:

<https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess>

Comunque, poiché questa API è chiamata molte volte, non possiamo semplicemente impostare un breakpoint e patchare manualmente il buffer di ritorno quando viene controllata la ProcessDebugPort. Dobbiamo scrivere uno script che gestisca automaticamente questa situazione (e che modifichi anche il PEB per sconfiggere i primi due controlli)

Al tempo della scrittura di questo documento, ho inviato il mio script al repository degli script di x64dbg, ma non è stato ancora accettato (puoi trovarlo qui:

<https://github.com/x64dbg/Scripts/pull/21>).

Questa è una versione leggermente più aggiornata:

```
// -----  
msg "Safedisc v1.06-1.41 anti antidebugger"  
run // run til the EntryPoint  
  
// clear breakpoints  
bc  
bphwc  
  
bphws WriteProcessMemory // I WILL EXPLAIN THIS LATER  
  
// defeats isDebuggerPresent and manual PEB checks  
$peb = peb()  
set $peb+0x2, #00#  
  
// find and hook NtQueryInformationProcess  
nqip_addr = ntdll.dll:NtQueryInformationProcess  
bp nqip_addr  
SetBreakpointCommand nqip_addr, "scriptcmd call check_nqip"  
erun  
ret  
  
check_nqip:  
log "NtQueryInformationProcess({arg.get(0)}, {arg.get(1)},  
{arg.get(2)}, {arg.get(3)}, {arg.get(4)})"
```

```

cmp [esp+8], 7 // 0x7 == ProcessDebugPort
je patch_process_information_buffer
erun
ret

patch_process_information_buffer:
rtr
set [esp+C], #00 00 00 00#
erun
ret
// -----

```

Come puoi vedere lo script è piuttosto semplice: pulisce il flag BeingDebugged dal PEB e crea una callback su NtQueryInformationProcess. Quando questa API sarà chiamata, verrà controllato se il suo secondo parametro ([esp+8], sullo stack) è 0x7 (ricorda, 0x7 è ProcessDebugPort) e se è questo il caso, patcherà il buffer tornando 0 (puoi controllare questa API e i suoi parametri sul portale della documentazione Microsoft come al solito).

Ignora momentaneamente il breakpoint hardware impostato su WriteProcessMemory, ci torneremo tra poco.

NOTA BENE: se non vuoi usare questo script, puoi nascondere ugualmente il debugger usando ScyllaHide. Ma dov'è il divertimento nell'usare qualcosa di già pronto? :)

Bene, è tempo di ricaricare l'eseguibile nel debugger e lanciare il nostro script: dopo i soliti 20-30 secondi della fase di verifica, il gioco partirà! (premi RUN nuovamente quando l'esecuzione viene bloccata dal breakpoint su WriteProcessMemory).

Ora possiamo iniziare a divertirci :)

Dobbiamo estrarre l'ICD decifrato dalla memoria, ma per farlo, dobbiamo prima fermarci all'OEP (entry point originale, la prima istruzione che viene eseguita dal processo ICD), altrimenti il nostro eseguibile conterrà dati relativi all'esecuzione corrente. Quindi il prossimo passo è trovare l'OEP. Ma come possiamo fermarci all'OEP di un processo che viene creato da un altro processo (ricorda, il loader creerà il processo ICD)?

Questa è un'ottima domanda, e ho passato un paio d'ore a pensarci.

Dopo una ricerca su google, ho trovato questa issue ricca di informazioni sul repository di x64dbg: <https://github.com/x64dbg/x64dbg/issues/1850>

Un utente chiamato blaquee suggerisce di patchare il processo usando il trucco dell'EBFE (ovvero di jumpare sullo stesso indirizzo), in questo modo possiamo temporaneamente fermare il processo ICD e quindi cercare l'OEP. Un'ottima idea direi!

Quindi, sappiamo che il loader ad un certo punto scriverà dati sulla memoria del processo ICD, possiamo dunque impostare un breakpoint sull'API WriteProcessMemory (ora sai perché era presente nello script) e patchare il suo buffer prima che venga scritto nel processo figlio.

ATTENZIONE: non usare breakpoint software su WriteProcessMemory. I breakpoint software modificano l'opcode (il primo opcode all'indirizzo verrà rimpiazzato con 0xCC) e poiché Safedisc controlla la loro presenza, manderà in crash l'esecuzione.

D'altro canto, i breakpoint hardware non modificano il codice e possono essere usati in sicurezza. Se proprio insisti nell'usare un breakpoint software, impostane uno a WriteProcessMemory+0x2, in questo modo non sarà rilevato.

Bene, ricarichiamo il nostro script ed eseguiamolo nuovamente. Alla fine il breakpoint appena impostato scatterà e ci ritroveremo nella WriteProcessMemory.

Questa API ha la seguente firma (puoi controllarla anche sul portale Microsoft):

```

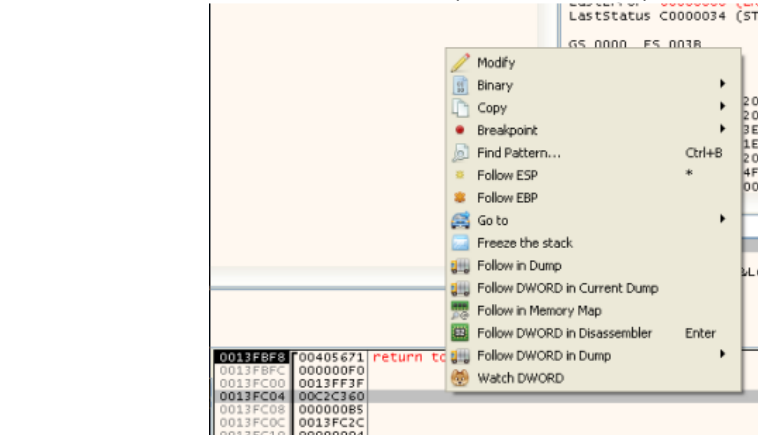
BOOL WriteProcessMemory(
    [in] HANDLE hProcess,
    [in] LPVOID lpBaseAddress,

```

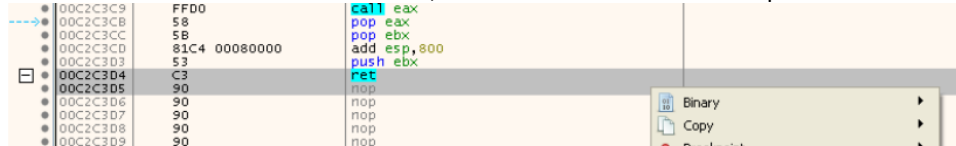
```
[in] LPCVOID lpBuffer,
[in] SIZE_T nSize,
[out] SIZE_T *lpNumberOfBytesWritten
);
```

Siamo interessati al terzo argomento, cioè il buffer.

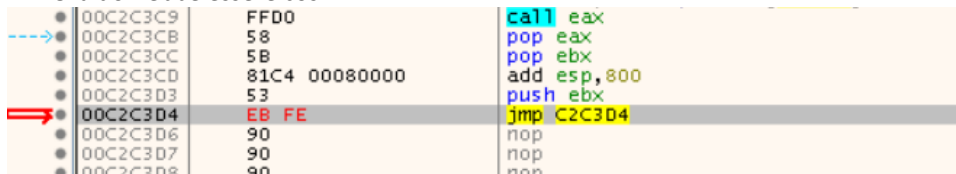
Guardiamo attentamente la finestra dello stack (in basso a destra) e troviamo questo parametro:



Clicchiamo su Follow DWORD in Disassembler, e scorriamo in basso sino all'opcode RET:



Andiamo a patcharlo per effettuare il trucco dell'EBFE. Selezioniamo entrambi gli indirizzi come in figura e clicchiamo col destro del mouse su uno di essi. Scegliamo Binary->Edit, e rimpiazziamo C3 90 con EB FE. Ora dovrebbe essere così:



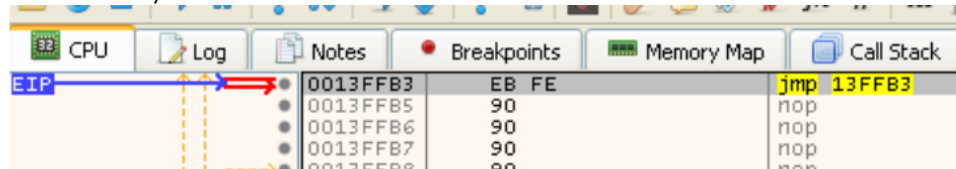
Se presti attenzione alla linea rossa, noterai che salta su sé stessa.

Siamo pronti a riprendere con l'esecuzione del programma, premendo su RUN!

Anche se non riesci a vederlo, il processo ICD è in esecuzione, in loop su quell'indirizzo.

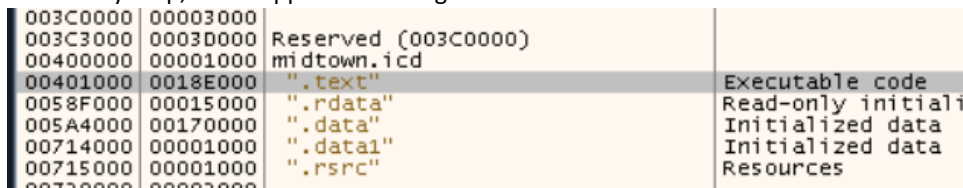
Possiamo aprire una seconda istanza di x32dbg (NON CHIUDERE QUELLA ATTUALMENTE APERTA), clicca su File->Attach e seleziona il processo ICD dalla lista.

Siamo quindi dentro il processo figlio e come puoi vedere EIP è in loop sullo stesso indirizzo (grazie al truccetto dell'EBFE):

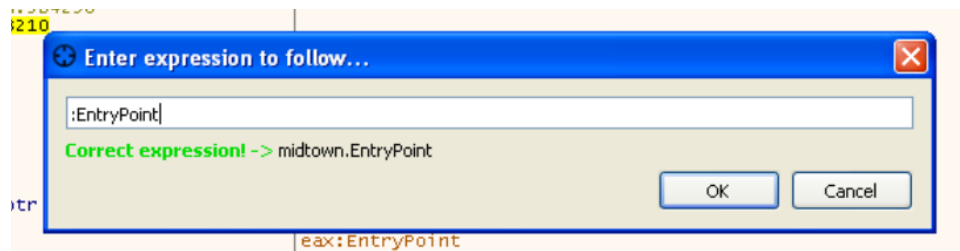


Prima di ripristinare l'opcode originale (RET), cerchiamo l'OEP e settiamogli un breakpoint.

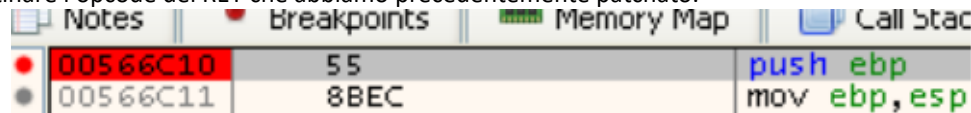
Clicca su Memory Map, e fai doppio click su segmento .text di midtown.icd:



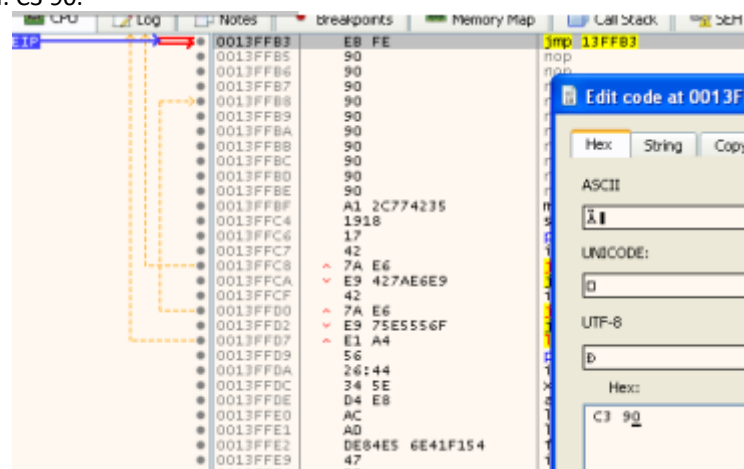
Premi CTRL+G ed inserisci :Entry (o :Entrypoint)



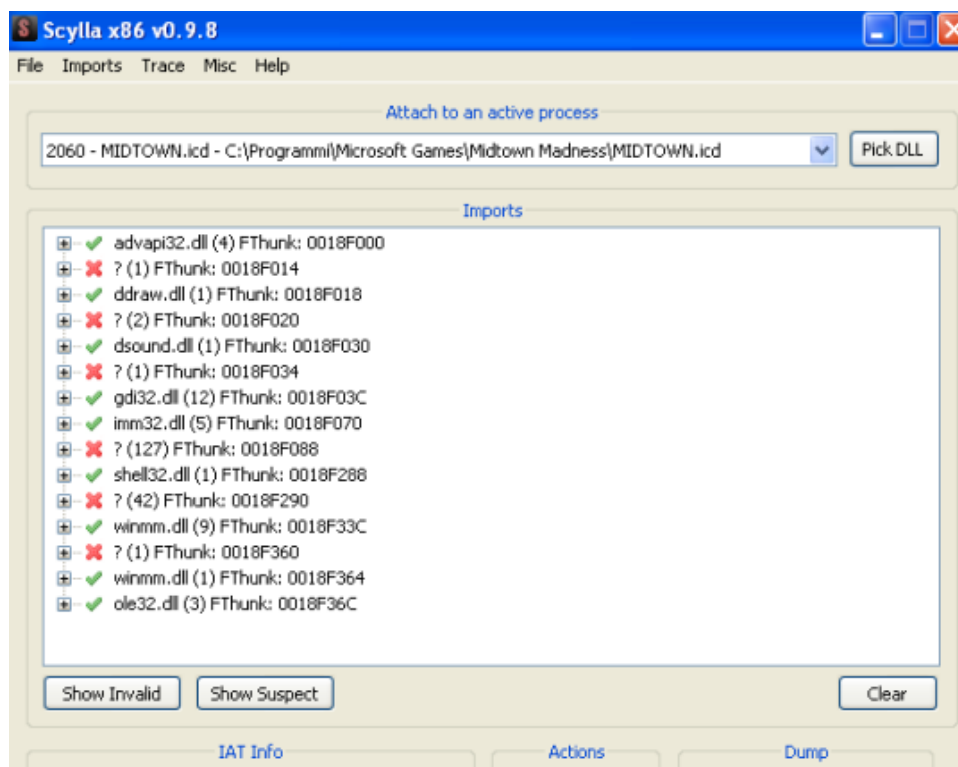
Premi Invio, e finalmente ci troveremo sull'OEP! Impostiamo un breakpoint e saremo pronti per ripristinare l'opcode del RET che abbiamo precedentemente patchato.



Premi CTRL+G nuovamente e inserisci EIP, premi Invio e ci ritroveremo al punto del loop dovuto all'EBFE. Clicca con il destro su questo indirizzo e seleziona Binary->Edit, dunque ripristiniamo gli opcode originali: C3 90.



Appena premiamo Invio, il flusso dell'esecuzione continuerà e ci fermeremo direttamente sull'OEP! Apriamo quindi Scylla (l'icona a forma di S sulla barra degli strumenti), riempiamo il campo OEP con 00566C10 e clicchiamo su IAT Autosearch, su YES, su OK e per finire su Get Imports. Maledizione! Abbiamo un grosso problema!

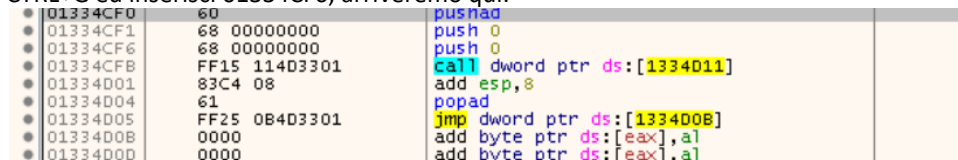


Questa è la IAT (l'address import table) e come è possibile vedere contiene molte voci invalide. Il nostro compito è quello di ripararle in modo che Scylla possa creare una IDT valida e il nostro eseguibile possa funzionare una volta estratto dalla memoria. Il responsabile di questo problema è Safedisc... ma con un po' di pazienza possiamo ripristinare la IAT. Ti avviso: da ora in poi ci sarà la necessità di riavviare spesso l'applicazione e ripetere tutto quello che abbiamo fatto sino ad ora da capo, quindi assicurati di aver capito bene tutti i passaggi che abbiamo svolto (in alternativa puoi continuare a seguire questo documento e procedere solo dopo che abbiamo recuperato tutte le informazioni che ci servono). Ignora momentaneamente le voci che presentano poche API invalide (ci torneremo dopo su quelle) e concentrati sulle due voci che presentano il maggior numero di API mancanti. Nella prima voce mancano 127 (0x7F) API, mentre alla seconda ne mancano 42 (0x2A). Prendi nota di questi valori perché dopo ci serviranno.

Iniziamo a controllare la prima:



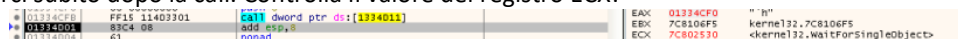
Premi CTRL+G ed inserisci 01334CF0, arriveremo qui:



Fermiamoci un attimo a capire cosa sta succedendo in queste poche righe:

Il primo pushad conserva i registri sullo stack, poi due valori (in questo caso entrambi 0) vengono pushati e viene effettuata una chiamata a dplayerx.dll. Interessante....

Proviamo ad eseguire questo codice e vediamo cosa succede. Clicchiamo con il destro sull'istruzione pushad e selezioniamo Set New Origin Here, continuiamo cliccando step-over su ogni istruzione sino a fermarci subito dopo la call. Controlla il valore del registro ECX:



Kernel32.WaitForSingleObject!

Abbiamo recuperato la prima API! Proviamo a recuperare anche la seconda: CTRL+G e inseriamo 01334D15. Effettuiamo gli stessi passaggi anche qui: settiamo una nuova origine su pushad,

Se hai fatto attenzione, avrai notato che il primo valore che viene pushato nello stack incrementa progressivamente ad ogni API che recuperiamo: nella prima chiamata era 0, nella seconda 1:

Nella terza 2:

Possiamo dedurre che questo valore è l'indice dell'API di Kernel32 desiderata!

Adesso ci rimane da capire il significato del secondo valore pushato e finalmente possiamo cercare il modo di automatizzare il recupero di queste API.

Nota bene che sino ad ora abbiamo recuperato SOLO API appartenenti a kernel32.dll (WaitForSingleObject, OutputDebugString and LoadLibraryA).

Proviamo a recuperare qualche API dall'altro gruppo. Prendiamo nota dell'indirizzo in Scylla:

Premiamo CTRL+G and inseriamo 00DD4930, arriveremo qui:

Nota bene come questa funzione è molto simile a quella già incontrata, con la sola differenza che qui viene pushato 1. Selezioniamo il primo pushad e scegliamo Set New Origin Here, e steppiamo tutte le istruzioni sino a subito dopo la call. Otterremo user32.SetFocus nel registro ECX. Facciamo lo stesso per la seconda e la terza call:

e

Noteremo che tutte e tre risolvono funzioni appartenenti ad user32.dll.

Iniziate a capire cosa sta succedendo? :)

Il primo push seleziona l'indice della funzione desiderata

Il secondo push seleziona l'indice della libreria desiderata

In tutte le versioni di Safedisc che ho analizzato, solo 2 librerie vengono proxate in questo modo:

0 equivale a kernel32.dll

1 equivale a user32.dll

Se sei interessato a scoprire come vengono calcolati i nomi delle chiamate, sentiti libero di entrare nella chiamata a dplayerx e di seguire il disassemblato. Ti avverto, il codice è molto offuscato (ci sono una marea di salti privi di senso per farti confondere).

Adesso possiamo pensare ad un modo per sistemare automaticamente tutte queste funzioni, poiché effettuare questa operazione manualmente (impostare una nuova origine su ogni chiamata, eseguire il codice fino a dopo la call, leggere il nome dell'API nel registro ECX e impostarlo correttamente su Scylla) richiede una quantità enorme di tempo ed è anche facile commettere errori.

Il metodo che personalmente preferisco è quello spiegato da W4kfu (è un reverser di talento e anche una persona molto disponibile) nel suo blog (<http://blog.w4kfu.com/>): useremo la funzione stessa del dplayerx per farci calcolare le API corrette!

Prima di scrivere qualche riga di codice asm per ottenere quanto desiderato, dobbiamo capire bene due concetti chiave:

- 1) Una volta ottenuto l'indirizzo dell'API, dobbiamo scriverlo nella IAT (così Scylla può identificarlo), quindi dobbiamo assicurarci che la sezione contenente la IAT sia scrivibile.
- 2) Abbiamo bisogno di un po' di spazio dove scrivere il nostro codice asm e assicurarci che questa area sia marcata come eseguibile.

Risolviamo al volo il primo problema: apri la Memory Map, trova il segmento ".rdata" subito dopo il segmento ".text" e cliccalo con il destro. Seleziona Set Page Memory Rights e clicca su Select All, su Full Access e quindi su Set Rights. Il primo problema è risolto. Adesso dobbiamo trovare lo spazio per il nostro codice.

Mentre siamo su Memory Map, cerchiamo un'area di memoria libera che sia marcata come PRV.

Per questo caso specifico, ho trovato l'area a 0x7F0000 perfetta per i nostri scopi.

007E0000	00002000		MAP	ER---
007E2000	00006000	Reserved (00720000)	MAP	
007F0000	00001000		PRV	-RW--
00800000	00005000		PRV	-RW--
00805000	00008000	Reserved (00800000)	PRV	
00810000	00002000		PRV	-RW--

Ripeto: assicurati di avere settato i diritti correttamente cliccando su Set Page Memory Rights e su Select All, su Full Access e quindi su Set Rights.

Fai doppio click su quest'area di memoria e la finestra con l'hex-view lampeggerà di rosso, clicca con il destro sul primo indirizzo e seleziona Follow in Disassembly. Siamo pronti per scrivere ed eseguire il codice che sistemerà la IAT!

Eseguiamo questa operazione in 2 passi: prima sistemeremo gli import di kernel32, e dopo quelli di user32.

Quindi iniziamo a scrivere attentamente il nostro codice partendo da 0x7F0000 (non preoccuparti, ne analizzeremo ogni singola riga tra un attimo)

Questo sistema gli import di kernel32:

Log	Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols
007F0000	33C0						eax: "h"
007F0002	BB 8BF05800						58F088:&"h"
007F0007	60						
007F0008	50						eax: "h"
007F0009	6A 00						
007F000B	FF15 11403301						
007F0011	890D 50007F00						
007F0017	83C4 08						
007F001A	61						
007F001B	8B0D 50007F00						
007F0021	890B						
007F0023	40						eax: "h"
007F0024	83C3 04						
007F0027	83F8 7F						eax: "h"
007F002A	75 0B						
007F002C	CD 03						
007F002E	0000						eax: "h"
007F0030	0000						eax: "h"
007F0032	0000						eax: "h"
007F0034	0000						eax: "h"

Ecco cosa fa questo codice:

Prima di tutto, puliamo il registro EAX, poiché lo useremo per tenere l'indice dell'API che cercheremo di recuperare. Impostiamo su EBX l'indirizzo della prima API (di kernel32) nella IAT: questa si trova nel segmento .rdata, puoi ottenere questo indirizzo dal First Thunk RVA ottenuto da Scylla (nel caso di kernel32 è 0x18F088) e poiché questo è un RVA, dobbiamo sommarlo all'ibase per ottenere il VA corretto.

Quindi: 18F088 (RVA del primo thunk)+400000(imagebase)=0x58F088. Possiamo a questo punto sfruttare lo stesso codice che utilizza Safedisc per risolvere le API. Fai attenzione che in questo caso abbiamo pushato EAX, che è l'indice dell'API che stiamo recuperando.

Subito dopo la call a dplayerx, conserviamo il valore nel registro ECX in un indirizzo di memoria temporaneo (0x7F0050), RICORDA che in questo momento in ECX c'è l'indirizzo corretto dell'API!

A questo punto ripristiniamo i registri con l'istruzione `popad` e ricarichiamo il valore precedentemente conservato (l'indirizzo dell'API) nuovamente in ECX. Possiamo quindi scrivere questo valore all'indirizzo corrispondente nella IAT (ricorda che in EBX c'è l'indirizzo dell'API corrente nella IAT). Incrementiamo EAX di 1 (così possiamo recuperare la prossima API) e incrementiamo EBX di 4 (poiché ogni thunk nella IAT è grande 4 byte). Confrontiamo EAX con 0x7F (ricorda che 0x7F è il numero degli import di kernel32 che abbiamo ottenuto prima da Scylla): se non è uguale, saltiamo a 0x7F0007 e sistemiamo la prossima API. Quando tutte le 0x7F API saranno risolte, ci fermiamo a 0x7F002C. Siamo pronti per eseguire il nostro codice. Settiamo l'origine a 0x7F0000, impostiamo un breakpoint a 0x7F002C e premiamo su RUN.

Se hai fatto tutto correttamente, ti troverai a 0x7F002C fermo sul breakpoint.

Modifichiamo il codice per sistemare gli import di user32:

Address	Disassembly
007F0000	33C0 xor eax, eax
007F0002	BB 90F25800 mov ebx, midtown.58F290
007F0007	60 pushad
007F0008	50 push eax
007F0009	6A 01 push 1
007F000B	FF15 114D3301 call dword ptr ds:[1334D11]
007F0011	890D 50007F00 mov dword ptr ds:[7F0050], ecx
007F0017	83C4 08 add esp, 8
007F001A	61 popad
007F001B	8B0D 50007F00 mov ecx, dword ptr ds:[7F0050]
007F0021	890B mov dword ptr ds:[ebx], ecx
007F0023	40 inc eax
007F0024	83C3 04 add ebx, 4
007F0027	83F8 2A cmp eax, 2A
007F002A	75 DB jne 7F0007
007F002C	CD 03 int 3
007F002E	0000 add byte ptr ds:[eax], al
007F0030	0000 add byte ptr ds:[eax], al
007F0032	0000 add byte ptr ds:[eax], al

Il codice è molto simile a quello precedente, tranne che abbiamo cambiato l'indirizzo del primo thunk nel segmento `.rdata`, che in questo caso è 0x58F290 (nuovamente, puoi ottenere questo indirizzo dalla somma dell'RVA del primo thunk di user32 ottenuto da Scylla, 0x18F290 + l'ibase = 0x58F290). Abbiamo inoltre cambiato il secondo push a 0x1 poiché stiamo sistemando gli import di user32 (ricorda: 0x0 -> kernel32, 0x1 -> user32). L'ultima modifica effettuata è il CMP a 0x7F0027, impostando il valore a 0x2A poiché questo è il numero delle API di user32 che dobbiamo sistemare (abbiamo ottenuto questo valore prima, con Scylla).

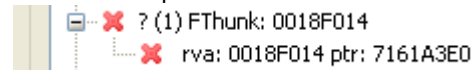
Nuovamente, settiamo una nuova origine a 0x7F0000, impostiamo un breakpoint a 0x7F002C e clicchiamo su RUN.

Una volta terminato, controlliamo la situazione su Scylla:



Ottimo, solo 5 API ancora da sistemare e potremo finalmente dumpare l'eseguibile dalla memoria. Procediamo sistemandole manualmente.

Risolviamo la prima:



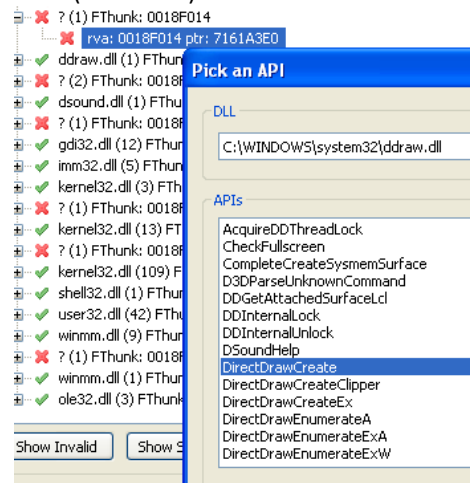
CTRL+G ed inseriamo 7161A3E0, ci troveremo qui:

7161A3E0	8BFF	mov edi,edi
7161A3E2	55	push ebp
7161A3E3	8BEC	mov ebp,esp
7161A3E5	A1 1C876571	mov eax,dword ptr ds:[7165871C]
7161A3EA	56	push esi
7161A3EB	FF75 10	push dword ptr ss:[ebp+10]
7161A3EE	FF75 0C	push dword ptr ss:[ebp+C]
7161A3F1	FF75 08	push dword ptr ss:[ebp+8]
7161A3F4	FF50 6C	call dword ptr ds:[eax+6C]
7161A3F7	8BF0	mov esi,eax
7161A3F9	85F6	test esi,esi
7161A3FB	75 13	jne aclayers.7161A410
7161A3FD	50	push eax
7161A3FF	FF75 0C	push dword ptr ss:[ebp+C]

Come di consueto, settiamo una nuova origine sulla prima istruzione (0x7161A3E0) e iniziamo a steppare le istruzioni sino alla call, una volta li portiamoci dentro e arriveremo qui:

736FCCA3	8BFF	mov edi,edi	DirectDrawCreate
736FCCA5	55	push ebp	
736FCCA6	8BEC	mov ebp,esp	
736FCCA8	56	push esi	
736FCCA9	33F6	xor esi,esi	

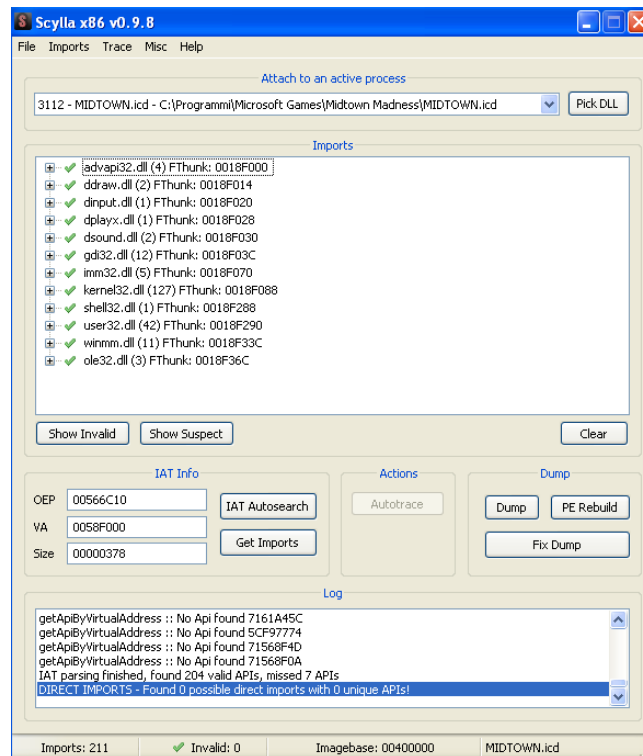
Ok, questa è DirectDrawCreate (di DDraw.dll), sistemiamola manualmente in Scylla cliccandola due volte e selezionando la dll corretta (DDraw.dll) e la relativa funzione:



Facciamo la stessa cosa con le chiamate rimanenti.

L'unica leggermente più complicata è quella a 0x5CF97774, ma possiamo chiaramente vedere dalla prima call che si tratta di kernel32.GetProcAddress.

Appena tutte le API sono state risolte, avremo questo risultato:



CONGRATULAZIONI, ci sei riuscito! La IAT è stata aggiustata e possiamo procedere dumpando il processo (tasto Dump) e quindi a fixarlo (tasto Fix Dump).

SAFEDISC È STATO SCONFITTO.

Posizioniamo l'eseguibile nella cartella del gioco e proviamo ad avviarlo.....

COOOOSA?! NON FUNZIONA?!?!? CHE DIAMINE!!!!!!

Ok, manteniamo la calma.

Se aspettiamo un po' di tempo, alla fine si avvierà, ma ovviamente questo non è ciò che si dovrebbe verificare. Quindi, ci deve essere qualche altro controllo nell'eseguibile che abbiamo dumpato.

Apriamo il nostro eseguibile nel debugger e controlliamo cosa sta succedendo.

Il gioco sembra bloccato in un loop senza riuscire a partire. Clicchiamo su PAUSE nel debugger e su Run-To-User-Code.

Ci troveremo qui:

0055E736	33C0	xor eax,eax
0055E738	8945 E4	mov dword ptr ss:[ebp-1C],eax
0055E73B	8945 FC	mov dword ptr ss:[ebp-4],eax
0055E73E	E8 7D000000	call midtown_dump_scy.55E7C0
0055E743	8BF8	mov edi,eax
0055E745	897D DC	mov dword ptr ss:[ebp-24],edi
0055E748	8B1D 60F35800	mov ebx,dword ptr ds:[&timeGetTime]
0055E74E	FFD3	call ebx
0055E750	8945 E0	mov dword ptr ss:[ebp-20],eax
0055E753	6A 64	push 64
0055E755	FF15 CCF05800	call dword ptr ds:[&sleep]
0055E75B	E8 60000000	call midtown_dump_scy.55E7C0
0055E760	8BF0	mov esi,eax
0055E762	2BF7	sub esi,edi
0055E764	8975 DC	mov dword ptr ss:[ebp-24],esi
0055E767	FFD3	call ebx
0055E769	8BC8	mov ecx,eax
0055E76B	2B4D E0	sub ecx,dword ptr ss:[ebp-20]

Nota le due chiamate sospette che prendono il tempo di sistema e si fermano.

Steppiamo sino ad arrivare all'istruzione RET di questa funzione e ci troveremo qui:

0055E7E5	8BC8	mov ecx, eax
0055E7E7	8BC7	mov eax, edi
0055E7E9	2BC6	sub eax, esi
0055E7EB	99	cdq
0055E7EC	33C2	xor eax, edx
0055E7EE	2BC2	sub eax, edx
0055E7F0	83F8 05	cmp eax, 5
0055E7F3	7D 1C	jge midtown_dump_scy.55E811
0055E7F5	8BC7	mov eax, edi
0055E7F7	2BC1	sub eax, ecx
0055E7F9	99	cdq
0055E7FA	33C2	xor eax, edx
0055E7FC	2BC2	sub eax, edx
0055E7FE	83F8 05	cmp eax, 5
0055E801	7D 0E	jge midtown_dump_scy.55E811
0055E803	8BC6	mov eax, esi
0055E805	2BC1	sub eax, ecx
0055E807	99	cdq
0055E808	33C2	xor eax, edx
0055E80A	2BC2	sub eax, edx
0055E80C	83F8 05	cmp eax, 5
0055E80F	7C 06	j1 midtown_dump_scy.55E817
0055E811	8BFE	mov edi, esi
0055E813	8BF1	mov esi, ecx
0055E815	EB C9	jmp midtown_dump_scy.55E7E0
0055E817	03CF	add ecx, esi

Come possiamo vedere, il salto condizionale a 0x55E7F3 sarà seguito e alla fine il salto a 0x55E815 manderà il codice in loop.

Noppiamo quest'ultimo jump e il gioco si avvierà. Credo che questa sia una sorta di protezione basata sul tempo, ma non fa parte di Safedisc.

Hai finito, divertiti con Midtown Madness su Windows 11 e tieni il disco originale conservato in un posto sicuro :)

Credits

Voglio ringraziare le seguenti persone/risorse:

- L'admin di <https://anti-debug.checkpoint.com/> per questo sito estremamente ricco di informazioni
- blaquee dal repository di x64dbg per il trucco dell'EBFE
- W4kfu per i suoi documenti interessantissimi sulle varie versioni di Safedisc
- mrexodia per x64dbg (probabilmente il miglior debugger ring 3 in mia opinione)
- NtQuery per Scylla

Conclusione

Con quello che hai imparato da questo documento dovresti essere in grado di sconfiggere Safedisc dalla versione 1.07 alla 1.11.

Da Safedisc 1.30 le cose si complicano leggermente, ma rimangono sempre relativamente semplici se confrontate con le versioni più alte.

Spero che ti sia piaciuto questo documento tecnico. È il mio primo pubblico.

Luca