GAME: Midtown Madness (Chicago Edition)
Protection: Safedisc 1.07
Written by Luca D'Amico - V1.0 beta - 29 March 2022

You will need:
- Windows XP VM (I used VMware)
- Process Hacker 2
- x64dbg (x32dbg)
- Original game disc (you need the ORIGINAL, otherwise this will not work)

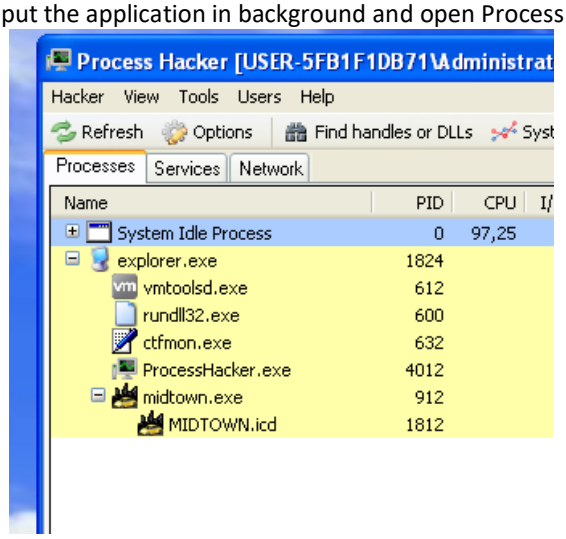
Before you start:
You need the original game disc (or a 1:1 copy) otherwise you will miss the decryption key.
We will examine how Safedisc works, and how it's possible to dump the decrypted game executable from memory and fix its import table.
Safedisc protected games will not work on Windows Vista and above, this is due to a security vulnerability in its driver. Once we will remove Safedisc, we will be able to run this game even on Windows 11.

Please take into account that you are about to fight against commercial grade copy protection, and even if this is now obsolete, back in its days this was put in place to slow down even the most skilled crackers of that time, so don't feel sad if you don't understand something even after a couple of reads. Trust me, if you spent the right amount of time, you will be able to master this.

Begin here:
First of all, install the game and select FULL install.

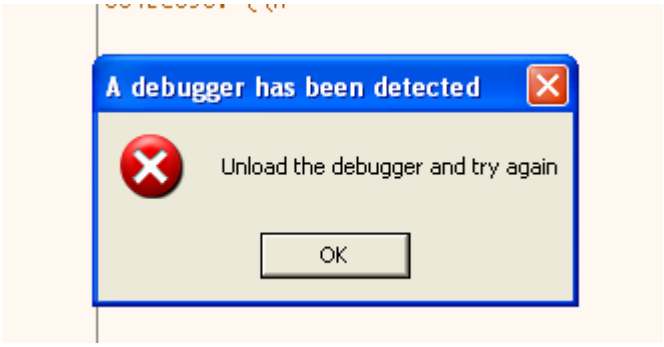
Now run the game and observe what happens by simply putting the game CD in the drive and double-click the game icon. Safedisc starts to "verify" the disc, and after about 20-30 seconds the game will start.

Once in the game menu, put the application in background and open Process Hacker 2:



Ok, we can see that midtown.exe created a new process, MIDTOWN.ICD.
What does this mean? Simple: midtown.exe is just a loader (this is why it is so small), and MIDTOWN.ICD is the real game executable, but it is encrypted (open it with any PE editor, and you will find that at least the .text section is scrambled).

Now open the loader (midtown.exe) in x32dbg. If we now try to run it, we will get this nice error message:



The loader is detecting that we are trying to run it in a debugger and it refuses to run, so, after a quick google search, I came across this very informative site with a list of common ways to detect a debugger: https://anti-debug.checkpoint.com/techniques/debug-flags.html

I tried all the proposed checks, and I found that these are used:
- IsDebuggerPresent API
- Manual checking BeingDebugged flag in PEB
- NtQueryInformationProcess() API

We can quickly patch and disable the first two checks, but for the third we need to pay attention as this API is also used to retrieve different things from the process other than just checking the presence of a debugger. We need to patch the return value of this function, ONLY when its 2nd parameter (ProcessInformationClass) is 0x7 (ProcessDebugPort). If you want to read more about this API, you can read the relevant page here: https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess

Since this API is called a lot, we can't just put a breakpoint and manually patch the return buffer when it is checking the ProcessDebugPort. We need to write a script that will automatically handle this situation for us (and also patch the PEB to defeat IsDebuggerPresent API).

At the time of writing, I've submitted my script to x64dbg scripts repository, but it isn't merged yet (you can find it here: https://github.com/x64dbg/Scripts/pull/21).

Here is a more updated version:

```
// -------------------------------------
msg "Safedisc v1.06-1.41 anti antidebugger"
run // run until 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
// -------------------------------------
```

As you can see the script is quite simple: it just clears the BeingDebugged flag from the PEB and hooks NtQueryInformationProcess. When this API is called, it will check if the 2nd parameter ([esp+8], on the stack) is 0x7 (remember 0x7 is ProcessDebugPort), and if that is the case it will patch the process information buffer returning 0 (you can check again this API and its parameter on MSDN).
Please ignore the hardware breakpoint set at WriteProcessMemory, we will get to that in a moment.

PLEASE NOTE: if you don't want to use this script, you can also hide the debugger with ScyllaHide plugin. But where's the fun in doing that? :)

It is now time to re-load the executable in the debugger and launch our script: after the classic 20-30 seconds verification phase, we will be in game! (Just press run again when the WriteProcessMemory breakpoint is triggered).

Now we can start to have some fun 😊

We need to dump the decrypted ICD from memory, but to do so, we need to break at its OEP (original entry point, the first instruction that is run in the ICD process), otherwise our dump will contain garbage data of the current execution. So, the next step is to find the OEP.

How can we break at the OEP of a process that is created from another process (remember, the loader will create the ICD process)?

This is a good question, and I spent a couple of hours thinking how to do it.
After a google search, I found this quite interesting issue on x64dbg repository:
https://github.com/x64dbg/x64dbg/issues/1850

A user called blaquee suggest to patch the process using the EBFE trick (jump on the same address), in this way we can temporarily halt the ICD process and then look for the OEP. This is a great idea!
We know that the loader at some point will fill the ICD process memory with the data, so we can simply put a breakpoint on the WriteProcessMemory API (now you know why it is there in the script) and patch its buffer just before it is written in the child process.

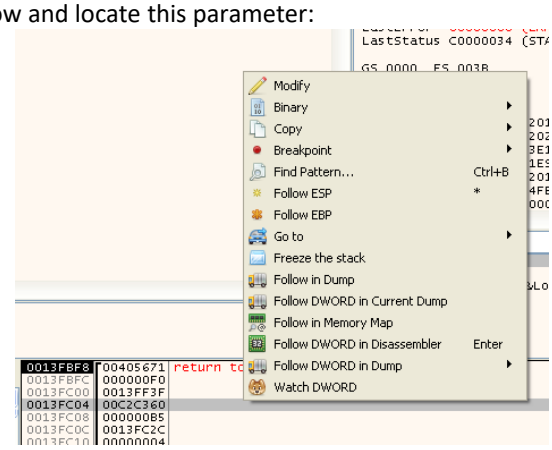
ATTENTION: do not use a software breakpoint on WriteProcessMemory. Software breakpoints will actually change the opcode (the first opcode will be changed with 0xCC), and since Safedisc checks the presence of them, it will just crash.

Because hardware breakpoints do not modify the code, they can be safely used, however, if you insist on using software breakpoints here; set one of them at WriteProcessMemory+0x2, this way it will not be detected.

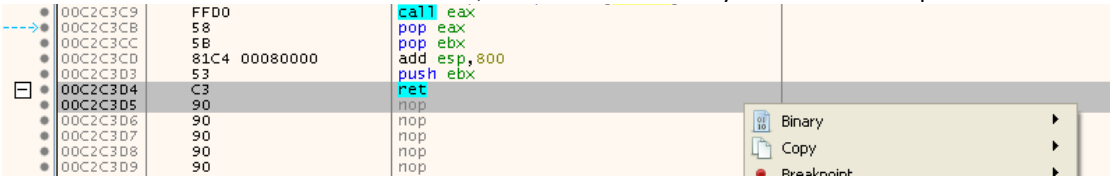
Let's reload our script and run it again. We will eventually break at WriteProcessMemory.
This API have the following signature (you can check it by yourself on MSDN):

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

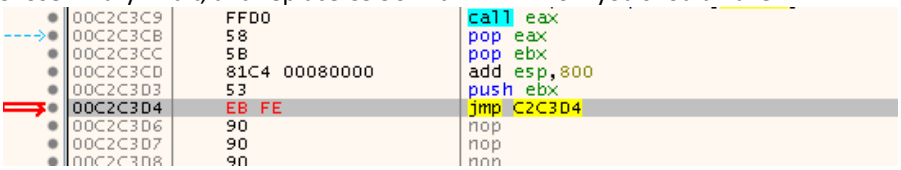
We are interested in the 3rd argument, that is the buffer.
Focus on the stack window and locate this parameter:



Click on Follow DWORD in the Disassembler, and scroll down until you find the RET opcode:

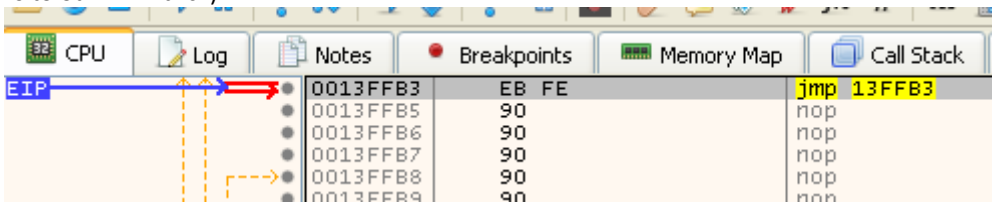


We are going to patch it with the EBFE trick. Select both addresses like in image and right-click on one of them. Chose Binary->Edit, and replace C3 90 with EB FE. Now you should have:

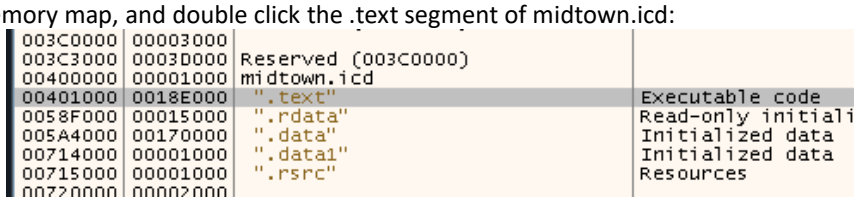


Notice the red line, meaning a jump on itself.

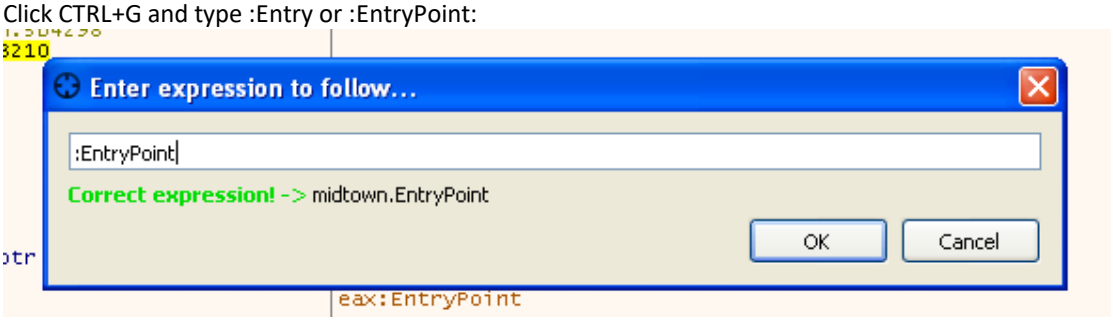
We are ready to resume the execution, so press run!
Even if you can't see it, the ICD process is running, looping on that address.
Now we can open a second instance of x32dbg (DO NOT CLOSE THIS ONE), click on File->Attach and select the ICD process from the process list.
We are now debugging the child process, and as you can see EIP is looping on the same address (thanks to our EBFE trick):



Before we restore the original RET opcode, let's locate the OEP and put a breakpoint there.
Click on Memory map, and double click the .text segment of midtown.icd:

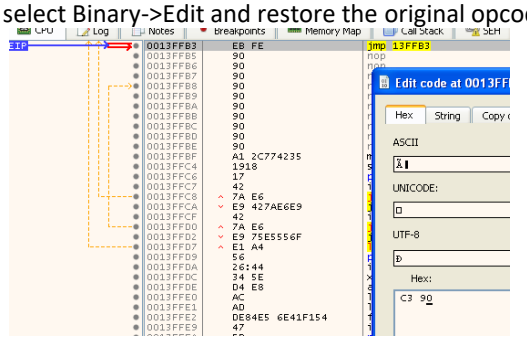


Click CTRL+G and type :Entry or :EntryPoint:

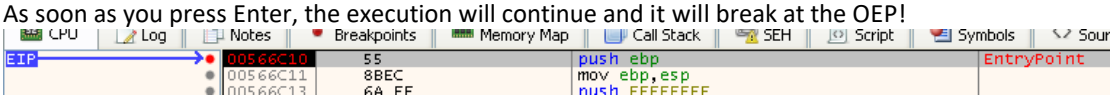


Press Enter, and finally, we are at the OEP! Put a breakpoint there, and we are ready to restore the patched RET instruction.

Click CTRL+G again and type EIP, press Enter, and we should be where the EBFE loop is in place.
Right click on that address, select Binary->Edit and restore the original opcodes: C3 90.



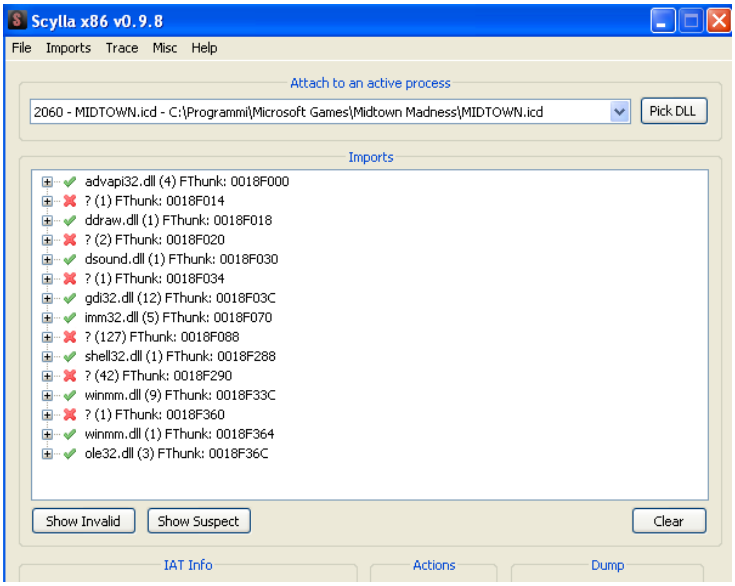
As soon as you press Enter, the execution will continue and it will break at the OEP!



That's GREAT! We are ready to fire up Scylla and dump our executable!

Open Scylla (the S icon on the toolbar), fill the OEP field with 00566C10, and press IAT Autosearch, click YES, OK, and then click on Get Imports.

We have a big problem!



This is the IAT (Import Adress Table) and as you can see there are many invalid entries.
We need to fix all of them so Scylla can create a valid IDT so that our executable will work once dumped. This was done by Safedisc, and with a little patience we can restore the IAT.

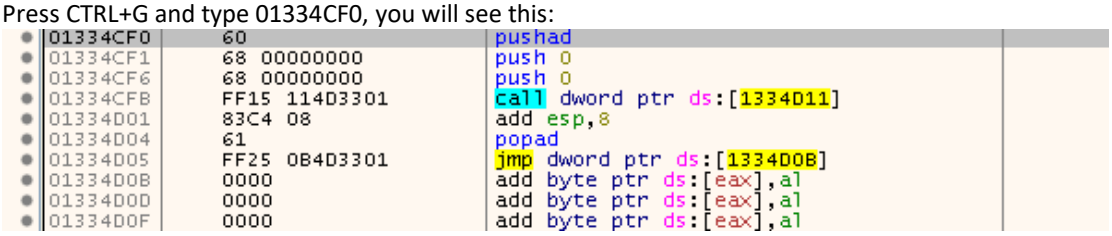
Be warned that you will have to restart the application frequently, so make sure to be comfortable with all we have done up to this point, because you will be doing all of this repeatedly (alternatively you can just follow this guide again and perform these steps once we have all the info we need).

Please ignore the entries with few broken APIs (we will fix them later), and focus on the two entries with lots of missing APIs.

The first one is missing 127 (0x7F) APIs, and the second one is missing 42 (0x2A) APIs. Take note of these values as we will use them later.

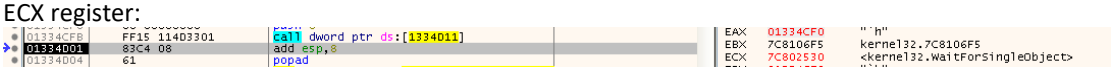
Let's start investigating the first one:

Press CTRL+G and type 01334CF0, you will see this:



Take a moment to understand what is happening here:
The first pushad stores the registers on the stack, then two values (in this case two 0s) are pushed and then a call to dplayerx.dll is performed. Interesting...

Let's try to execute that code and see what happens. Right-click on the pushad and select Set New Origin Here, step-over each instruction and stop right after the call. Now look at the value stored in ECX register:



Kernel32.WaitForSingleObject! We recovered the first API!

Now try with the second one: CTRL+G type 01334D15 and do the same thing here: set the new origin on pushad, step-over the instructions and stop right after the dxplayerx.dll call. You will be able to recover the second API (Kernel32.OutputDebugString).
Do the same with the 3rd entry, you will get a call to LoadLibraryA.

If you pay attention, you will notice that the value of the first push is incrementing, in the first call was 0, int the second call was 1:

```
01334D15    60              pushad
01334D16    68 01000000     push 1
01334D1B    68 00000000     push 0
01334D20    FF15 364D3301   call dword ptr ds:[1334D36]
01334D26    83C4 08         add esp,8
01334D29    61              popad
```

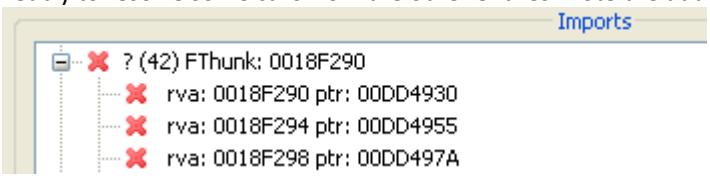
in the third call is 2:

```
01334D3A    60              pushad
01334D3B    68 02000000     push 2
01334D40    68 00000000     push 0
01334D45    FF15 5B4D3301   call dword ptr ds:[1334D5B]
01334D4B    83C4 08         add esp,8
01334D4E    61              popad
```

We can deduce that this value is actually the index of the desired kernel32.dll API!
From here we need to understand what the other pushed values represent, and come up with an automatic way to fix all these calls.

Please note that until now we have ONLY resolved kernel32.dll APIs (WaitForSingleObject, OutputDebugString and LoadLibraryA).

Let try to resolve some calls from the other entries. Note the address in Scylla:



Press CTRL+G and type 00DD4930, we are now here:

```
00DD4930    60              pushad
00DD4931    68 00000000     push 0
00DD4936    68 01000000     push 1
00DD493B    FF15 5149DD00   call dword ptr ds:[DD4951]
00DD4941    83C4 08         add esp,8
00DD4944    61              popad
```

Please note how this function is similar to the one already encountered, except for the push 1. Let's set a new origin in the pushad and step-over each instruction until just after the call. We will get user32.SetFocus in ECX.

Do the same with the second call:

```
00DD4955    60              pushad
00DD4956    68 01000000     push 1
00DD495B    68 01000000     push 1
00DD4960    FF15 7649DD00   call dword ptr ds:[DD4976]
00DD4966    83C4 08         add esp,8
00DD4969    61              popad
```

And the third call:

```
00DD497A    60              pushad
00DD497B    68 02000000     push 2
00DD4980    68 01000000     push 1
00DD4985    FF15 9B49DD00   call dword ptr ds:[DD499B]
00DD498B    83C4 08         add esp,8
00DD498E    61              popad
```

All of them resolve user32.dll functions.

Have you begun understanding what is happening?

The first push is pushing the index of the desired call and the second push is pushing the index of the desired library.

In every case of Safedisc that I've analysed only 2 libraries are altered that way:
0 is for kernel32.dll
1 is for user32.dll

If you are interested in how the correct API name is retrieved, feel free to jump into that dplayerx function and follow the disassembly, however, the code is very obfuscated (lots of meaningless jumps to confuse anyone attempting to reverse it!).

Now, we can think of a way to automatically fix these calls, since doing this manually (place a new origin on each call, run until after the call, read the API in ECX and set it in Scylla) is too time consuming and it is easy to make mistakes.

The method that I prefer is the one explained by W4kfu (he is a very skilled at reverse engineering and also a very nice person) in his blog (http://blog.w4kfu.com/): we will use the same dplayerx function to calculate the APIs for us!

Before we write some ASM code to achieve our result, we have to understand two concepts:

1) Once we retrieve the API address, we will need to write it in the IAT (so Scylla can identify it), so we have to make sure that this section is writable.
2) We need some space where to write our little ASM code, and we have to make sure that this area is executable.

Let's quickly fix the first one by opening the Memory Map, locate the ".rdata" segment just after the .text segment, and right click on it. Select Set Page Memory Rights, click on Select All, then select Full Access and finally click on Set Rights. The first problem is now resolved, so we now need to find some space for our code.

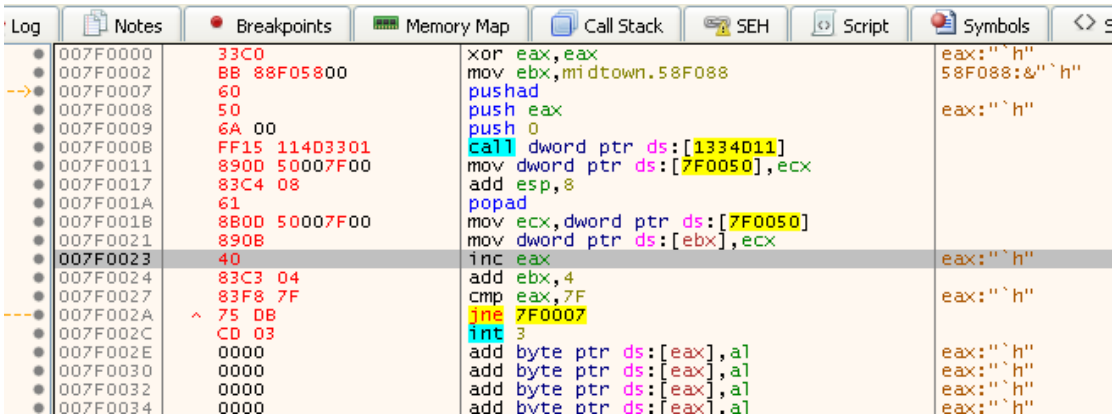
While in the Memory Map, try to find some empty memory area that is marked as PRV.
In this specific case I found 7F0000 is suitable for this:

Again, make sure we have full access rights by clicking on Set Page Memory Rights, click on Select All, select Full Access and finally click on Set Rights.

Double click on that memory area and the hex-view window will blink red, right click the first address and select Follow in Disassembly. We are ready to write and execute our IAT fixing code!

We will do this in 2 steps: the first will fix the kernel32 imports, and the second one will fix the user32 imports.

Carefully, write our code starting from 0x7F0000 (don't worry I'll explain you every single line of it in a moment). This one will fix the kernel32 imports.



Here is what this code does:

First of all, we clean the EAX register: this will hold the index of the API we are trying to retrieve. We then store the EBX address of the first API in the IAT. This is located in the .rdata segment, and you can get this address using the First Thunk RVA got using Scylla (in the case of Kernel32, it is 0x18F088) and since this is RVA, you have to add the imagebase address to get the correct VA. So: 18F088 (RVA of the first thunk) +400000 (imagebase) = 0x58F088.

We can reuse the same code that Safedisc is using to resolve the APIs. Notice that we pushed EAX, that is the index of the API we are retrieving. Just after the call to dplayerx, we store ECX in a temporary memory address (0x7F0050), REMEMBER that in ECX there is the correct API address now! We then restore the registers using the popad opcode, and then we load the previously stored value (the API address) in ECX again.

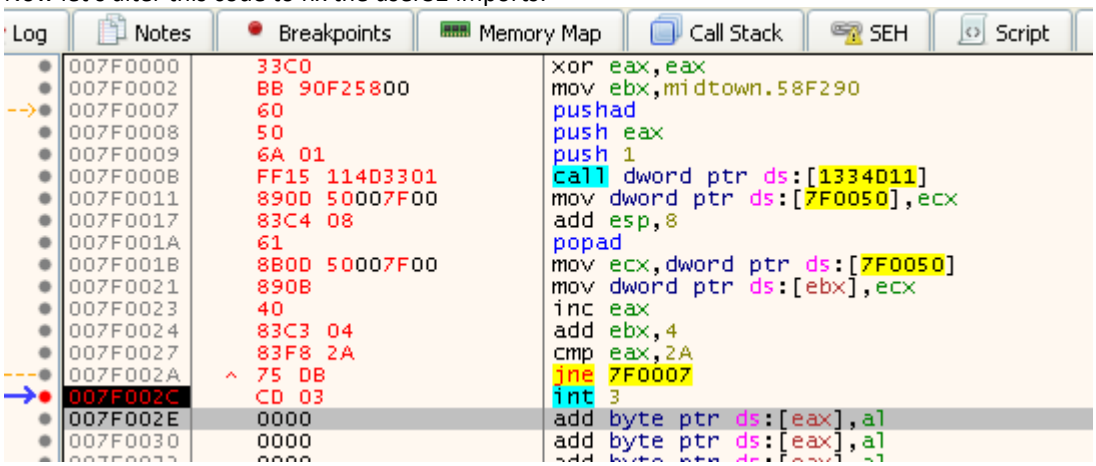
Now move that value to the correct address in the IAT (remember that in EBX there is the address of current API index in the IAT). We increment EAX by 1 (so we can retrieve the next API), and we increment EBX by 4 (because each Thunk in the IAT is 4 bytes long).

Finally, we can then compare EAX with 0x7F (remember 0x7F is the number of kernel32 imports that we got earlier from Scylla): and if it is not equal, we jump back to 0x7F0007 and we fix the next API. Once all 0x7F APIs are solved, we will break at 0x7F002C.

Now we are ready to execute it so set a new origin at 0x7F0000, put a breakpoint at 0x7F002C, and press RUN!

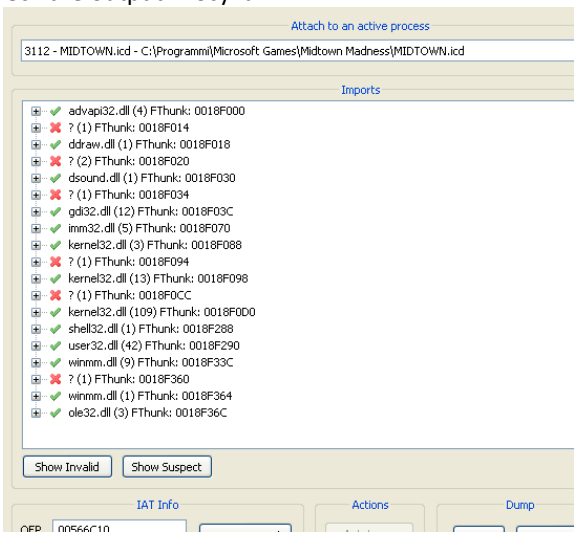
If you have done everything correctly, you will be now located at 0x7F002C due to the breakpoint.

Now let's alter this code to fix the user32 imports:



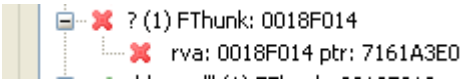
The code is quite similar to the previous one, but we changed the address of the first Thunk in the .rdata segment, in this case it is 0x58F290 (again, you can get this address by sum the RVA of the first Thunk of this entry you got from Scylla, 0x18F290 + the imagebase = 0x58F290). We also change the second push to 0x1 since we are now resolving the user32 imports (remember: 0x0 -> Kernel32, 0x1 -> User32). The last edit is the CMP at 0x7F0027, we use the value 0x2A because this is the amount of user32 APIs we need to retrieve (we got this value from Scylla earlier).

Finally, set a new origin at 0x7F0000, put a breakpoint at 0x7F002C, and press RUN.
Once finished, we can check the output in Scylla:

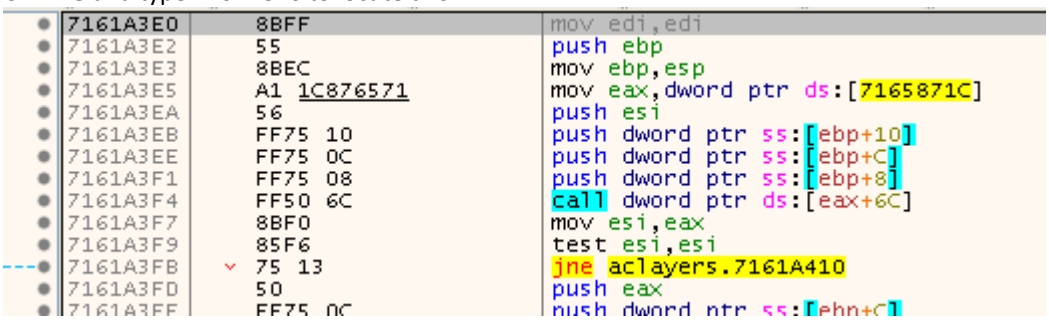


Great, only five manual fixes left, and we will be able to finally dump the executable from memory!

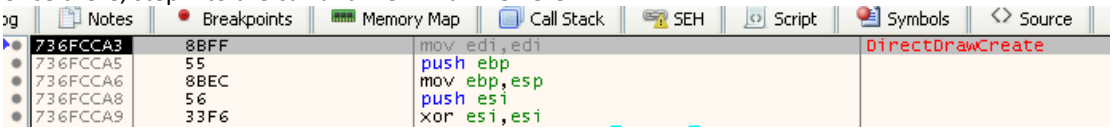
Let's fix the first one:



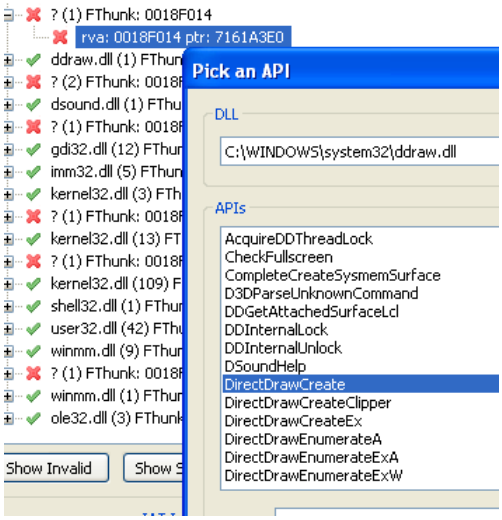
CTRL+G and type 7161A3E0 to locate this:



As usual, Set New Origin on the first instruction (0x7161A3E0) and start to step-over until the CALL, once there, step into the call and we will arrive here:

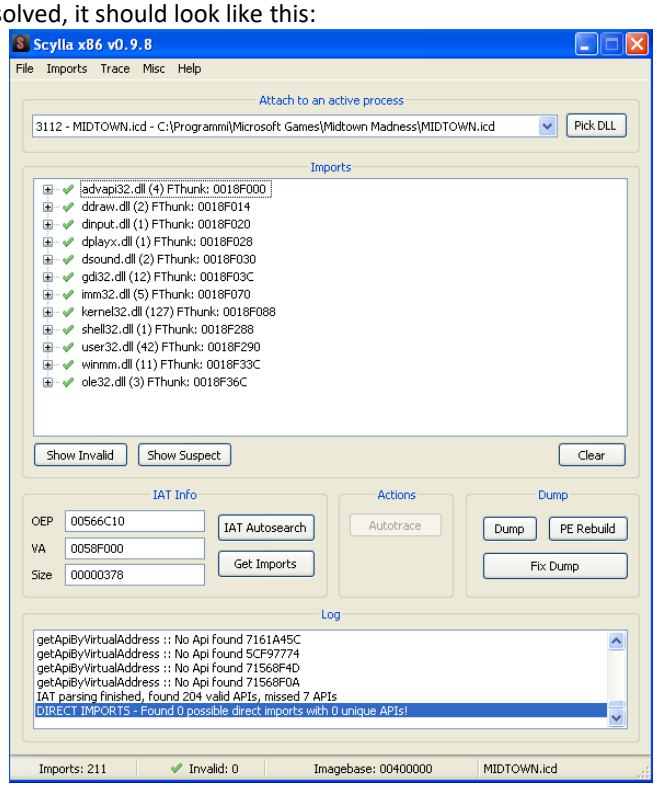


This is DirectDrawCreate (from DDraw.dll), we can manually fix it in Scylla by double clicking it and selecting the correct dll (DDraw.dll) and the correct call:



Do the same for the remaining calls, in this case the only tricky one is at 0x5CF97774, but you can clearly see from the first call that it is indeed kernel32.GetProcAddress.

Once every entry is solved, it should look like this:



CONGRATULATIONS, you did it! The IAT is now fixed, and we can proceed to dump the process (Dump button) and then fix it (Fix Dump button).

SAFEDISC HAS BEEN DEFEATED!

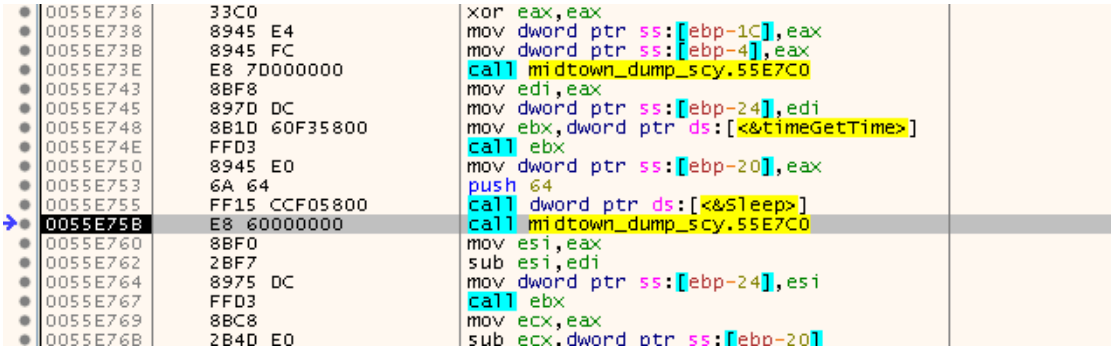
Let's put the executable in the game directory and try to run it:

"WHAAAT?! It is not working?! WHAT THE HELL?!"

Ok, keep calm 😊

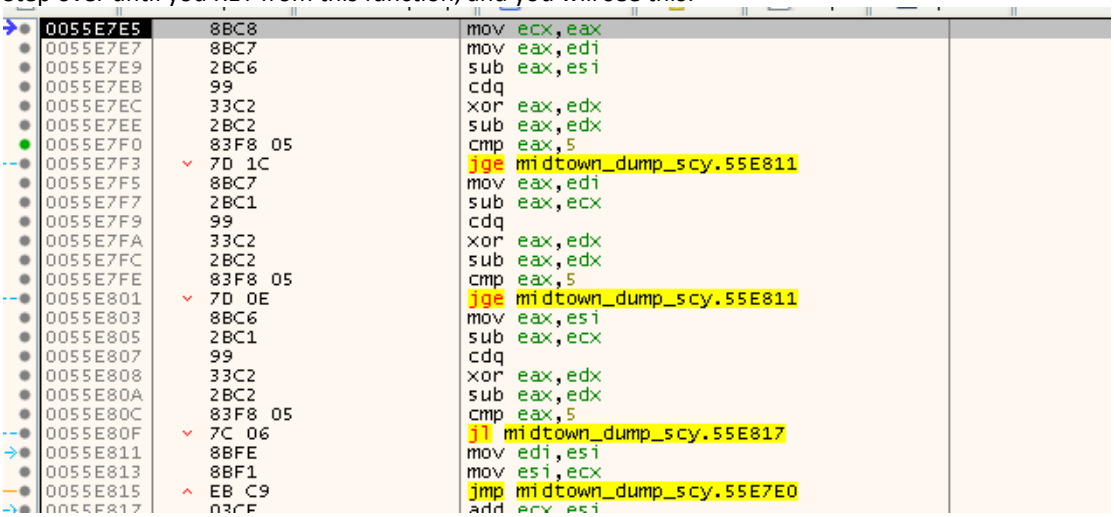
If we wait a random amount of time, it will eventually start, but this is not normal behaviour so there may still be some unwanted checks in our dumped exe.

Let's open our fixed executable in the debugger and check what is happening.
The game seems to loop without starting, so click pause on the debugger, click on Run-To-User-Code, here:



Notice the two suspicious calls to get the time and sleep?

Step over until you RET from this function, and you will see this:



If you follow the code, the conditional jump at 0x55e7f3 will be taken, and finally the jump at 0x55e815 will make the code loop again. Simply NOP this jump and the game will begin. I believe this is a sort of timing-based protection, but it is not part of Safedisc.

You can now enjoy Midtown Madness on Windows 11 and keep the original disc in a secure place 😊

Credits:
I would like to thank the following resources/people:
- The admin of https://anti-debug.checkpoint.com/ for his informative website
- blaquee from x64dbg issue repo for the EBFE trick
- W4kfu for his very informative writeups about various versions of safedisc
- mrexodia for x64dbg (probably the best ring3 debugger in my opinion)
- NtQuery for Scylla
- KiiWii (AKA DefaultDNB) for fixing the spelling and layout of this file (THANK YOU!)

Conclusion:
With the knowledge you got from following this guide you should be able to defeat Safedisc from 1.06 to 1.11. From Safedisc 1.30 things are a little more complicated, but it's still easy compared to the newer versions.

I hope you enjoyed this technical paper. It was my first public one!

Luca