The Squirrel Strikes Back: Analysis of the newly emerged cobalt-strike loader “SquirrelWaffle”

By Eli Salem

Published: 2021-09-21 · Archived: 2026-04-05 16:52:42 UTC

Just Squirrel with waffle

Since early-mid of September 2021, a new malware loader dubbed “Squirrelwaffle” has been discovered and observed delivering the attack framework Cobalt-Strike.

In the recent cybercrime landscape, several prolific malware has either gone or been less observed. This newly created gap gives opportunities for the birth of a new malware such as Squirrelwaffle to fill the hole that others left.

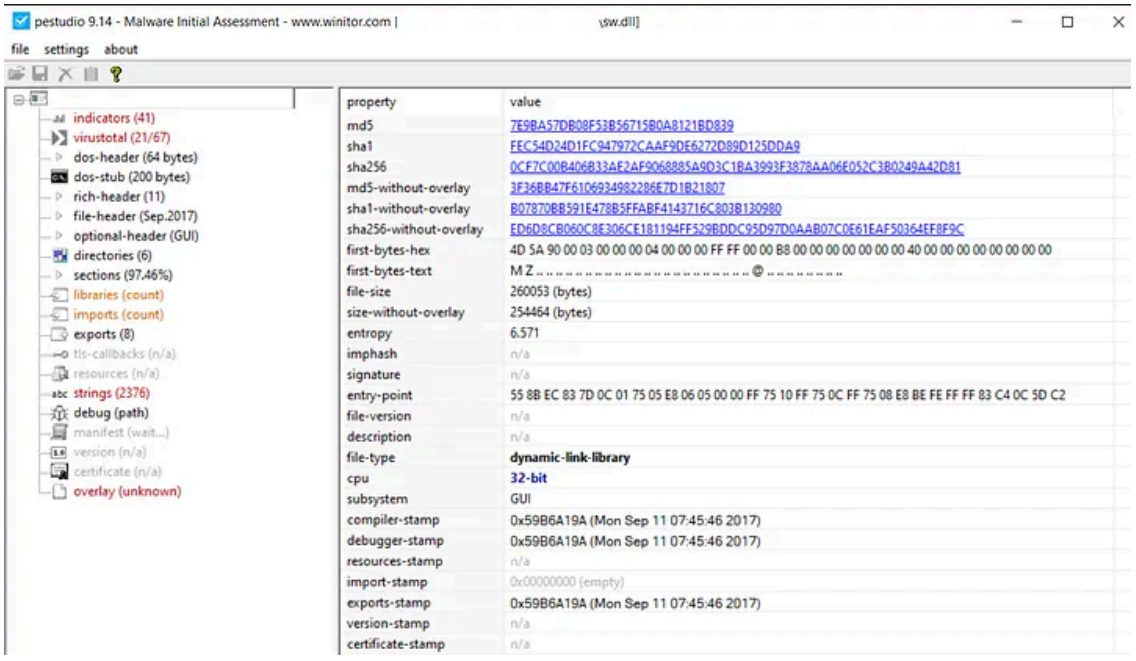
In this article, I will present an analysis of this new threat. Similar to most of my malware analysis articles, the article will be a mix between a presentation and a step by step dynamic or static analysis, with an emphasis on SquirrelWaffle download capabilities.

The dropper

In terms of the initial attack vector, the malware is being delivered by classic phishing documents and continues with dropping .vbs files and launching Powershell.
The initial access phase ends with an executable dropper being downloaded to the infected machine.

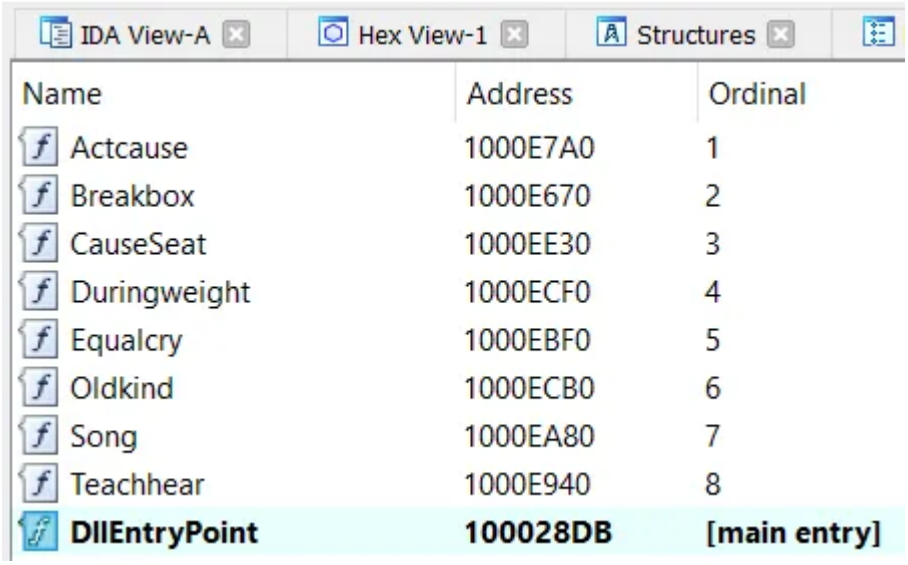
The dropper is a 32-bit DLL file, which also packed with a custom crypter.

Press enter or click to view image in full size



Dropper PEStudio

Furthermore, the dropper has 8 export functions in addition to the DllEntryPoint one. This tactic was also observed in Ursnif's dropper, which has been observed having a large number of export functions.
Usually, the reason for that is to slow down analysis.



Dropper export functions

Unpacking mechanism

To manually unpack this sample, and also observe the interesting parts of the unpacking mechanism we'll do the following:

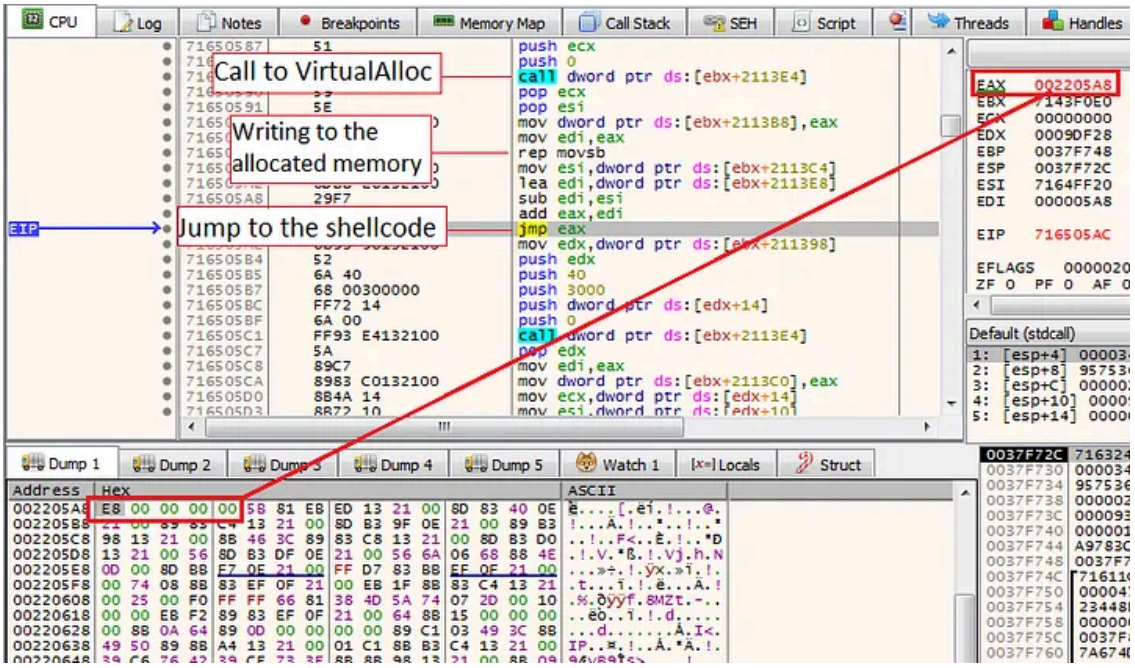
First, we'll set a breakpoint on *VirtualAlloc* and hit Run, once we reach the first breakpoint, click "Return to user code" or "execute till return + step over".
Now we can observe the following:

1. Call to *ebx+2113E4* - which is the call to *VirtualAlloc*
2. *rep movsb* -which will write shellcode to the newly allocated memory
3. *jmp eax* - execute the shellcode instructions

Because the shellcode is stored in the EAX register, we can observe it if we'll click "*follow in dump*" on the *EAX* register. we can see the bytes *E8 00 00 00 00* which is a classic trick shellcode uses to obtain the next instructions.

From this behavior, we can also assume that the entire unpacking mechanism will occur within the context of a shellcode.

Press enter or click to view image in full size

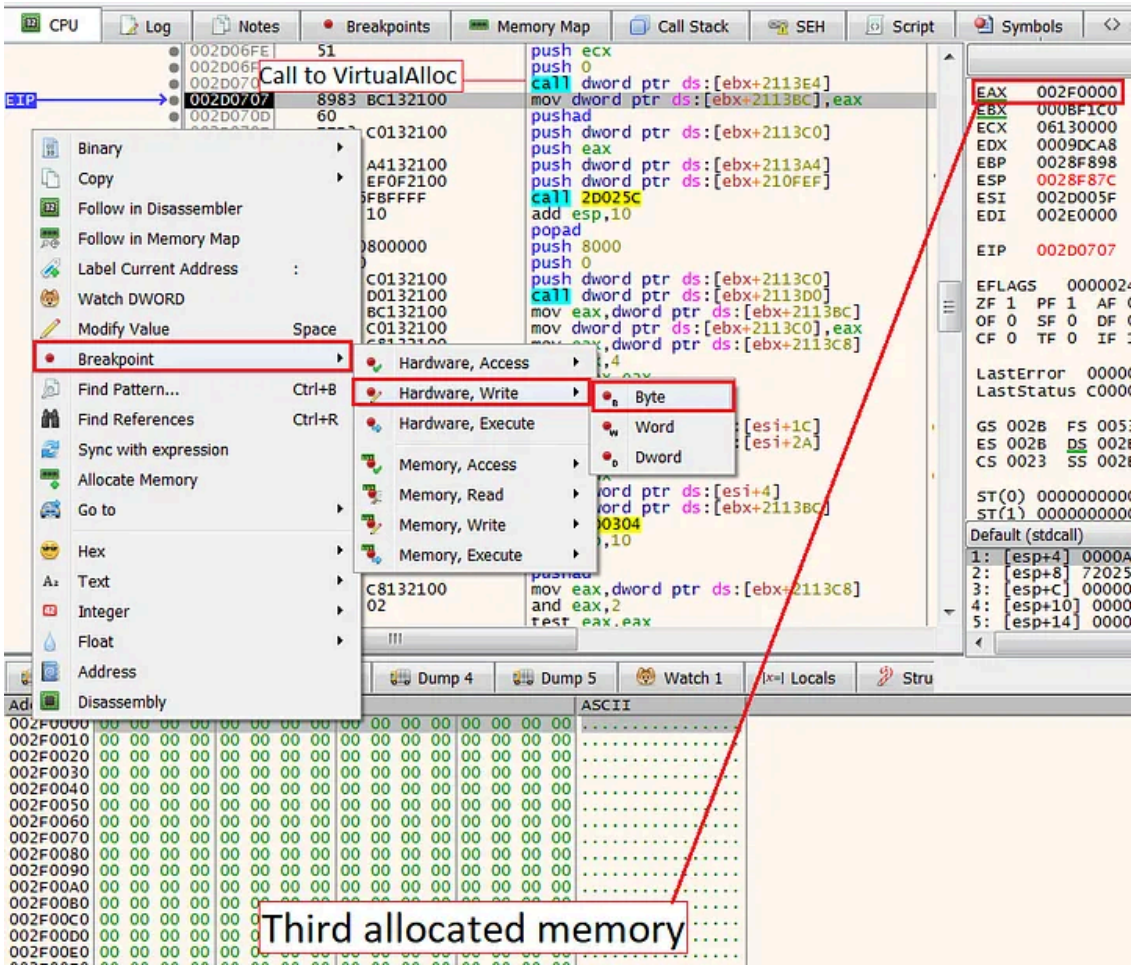


Dropper shellcode execution

Next, click Run twice until we'll reach the third instance of *VirtualAlloc*. After reaching it do the following:

1. Click "execute til return" + "step over"
2. Go to the *EAX* register and click "follow in dump"
3. Set a write hardware breakpoint on the first bytes of the newly allocated memory buffer.

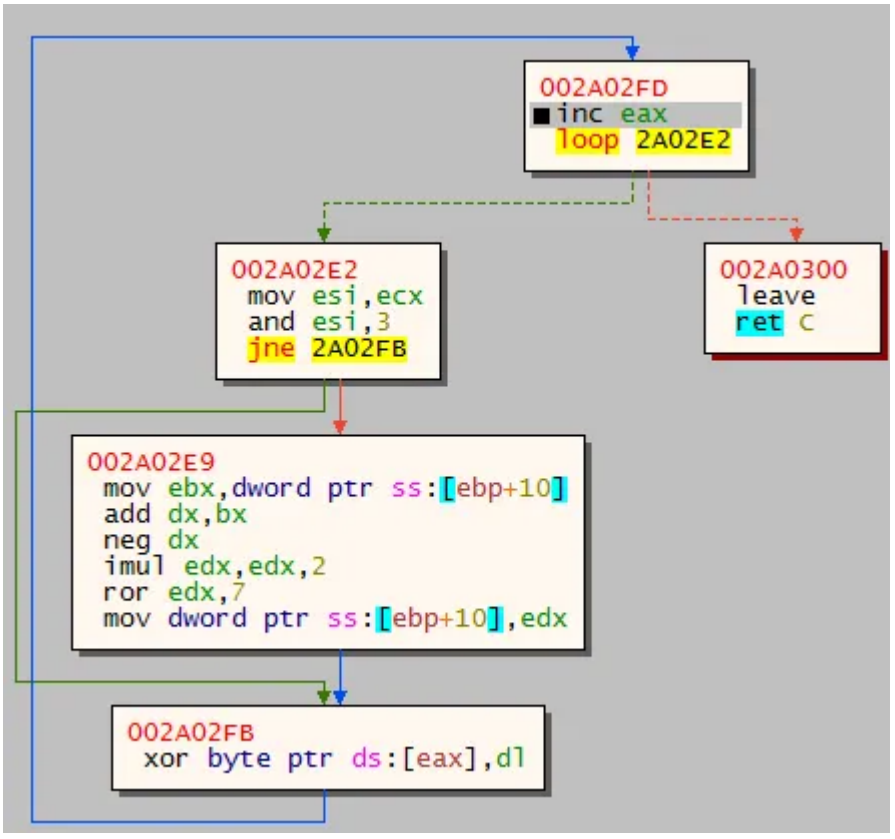
Press enter or click to view image in full size



Unpacking the dropper

After setting the breakpoint, click Run three times. We'll notice that the buffer inside the allocated memory will be filled with content.

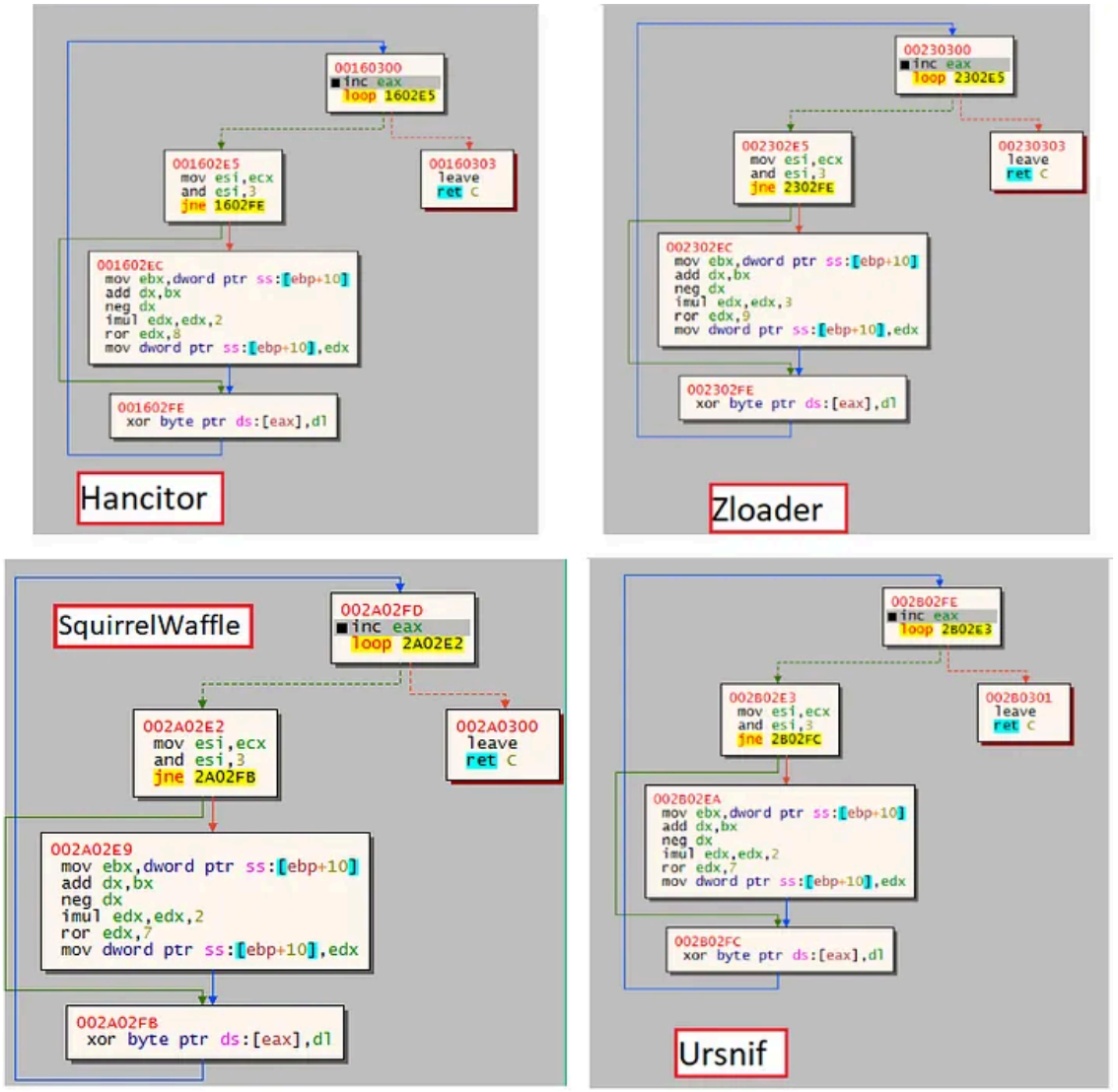
When reaching the third Run, we'll find ourselves in a small loop with some classic opcodes that we expect to see in unpacking loops such as rotate right and exclusion or (*ror* and *xor* opcodes) .



Unpacking the dropper

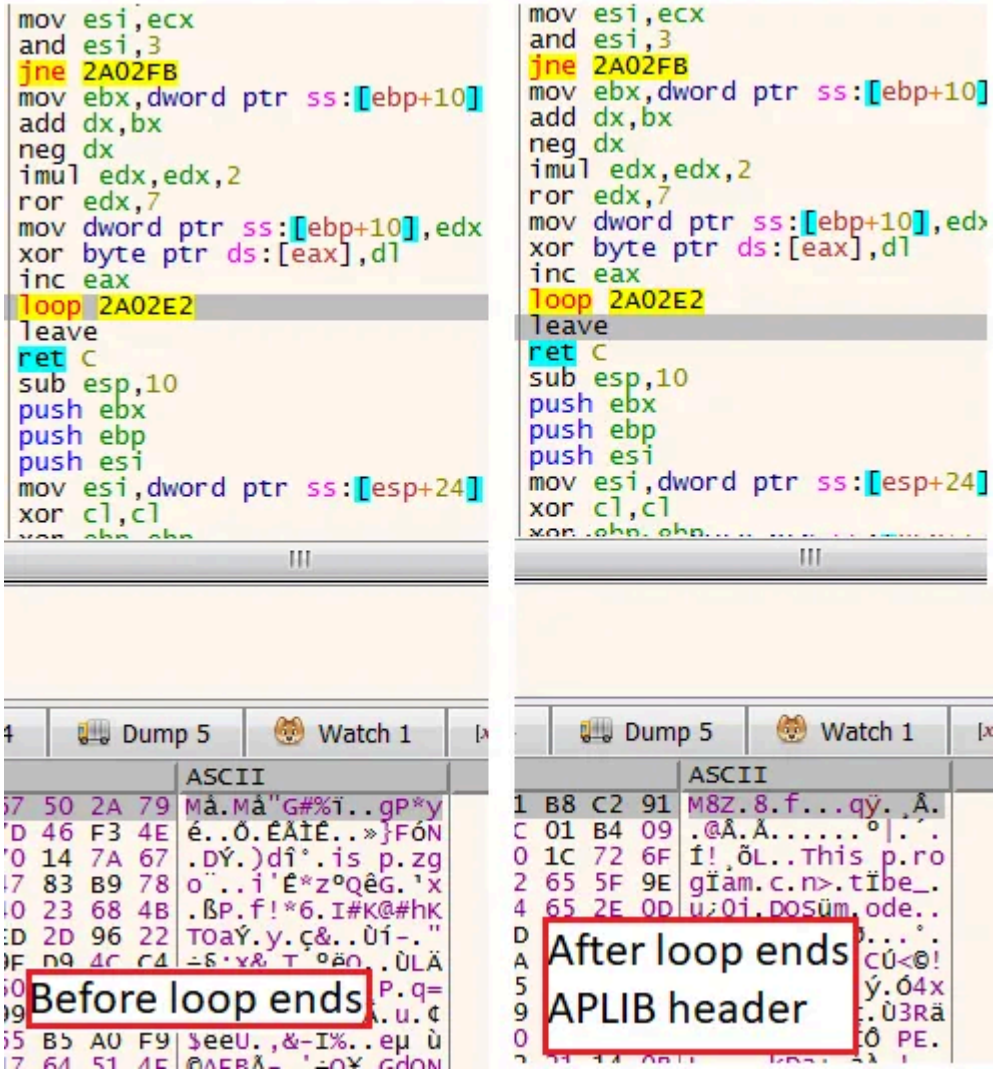
In fact, this specific loop, and the majority of this crypter were observed during the last two years in other malware droppers, such as: Ursnif, Zloader, and Hancitor.

Press enter or click to view image in full size



Dropper unpacking

By setting a breakpoint on the *leave* opcode we can go to the place where the loop ends. Once we did it, we can see the ASCII characters *M8Z* which indicates an APLIB compression.

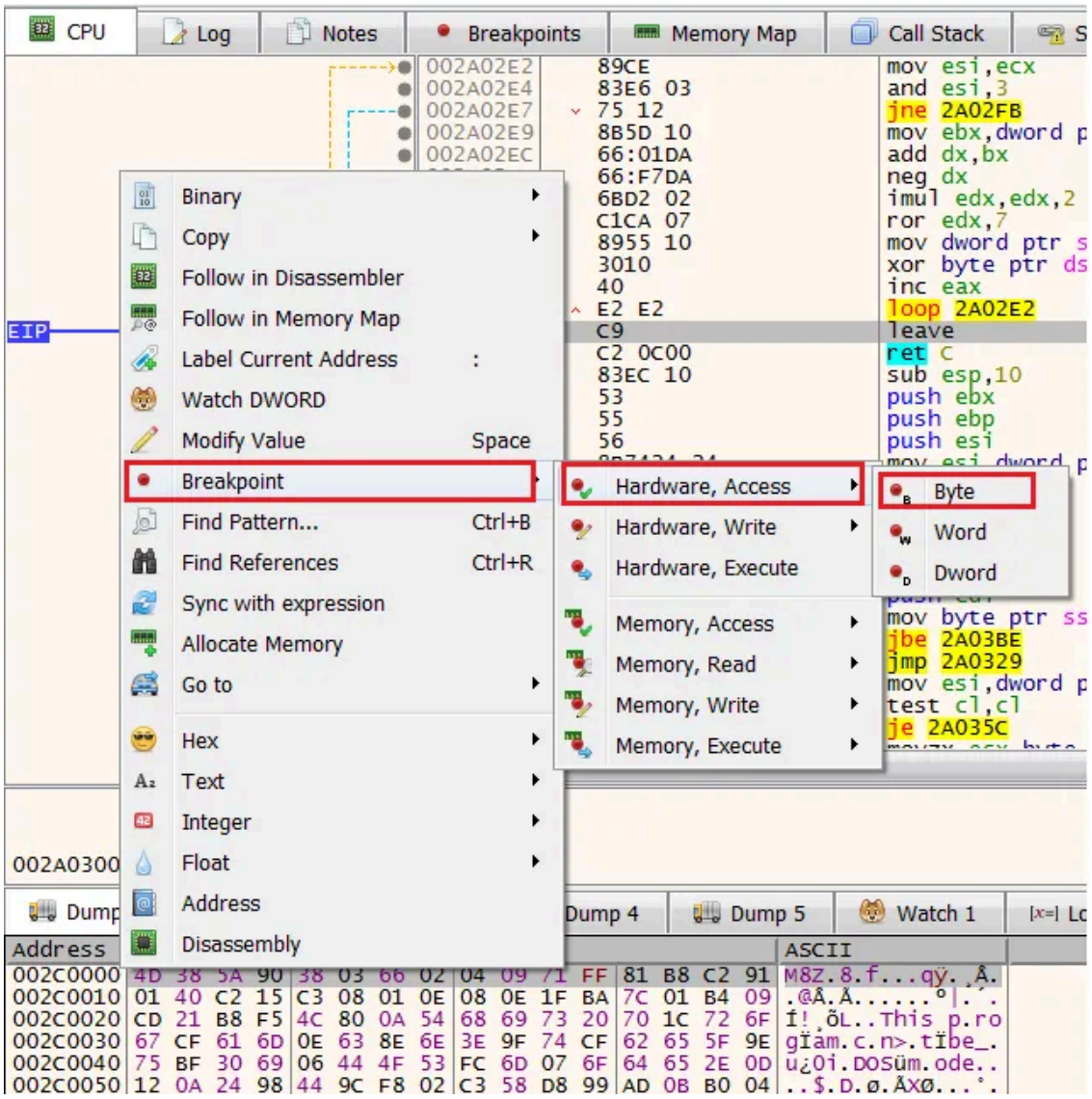


APLIB indication

Now that we know that this content is compressed with APLIB, the most logical thing to expect is a decompressing mechanism.

To observe this mechanism do the following:

1. Remove the write hardware breakpoint from the buffer
2. Set a new Access hardware breakpoint on the first bytes of the APLIB header.
3. Click Run

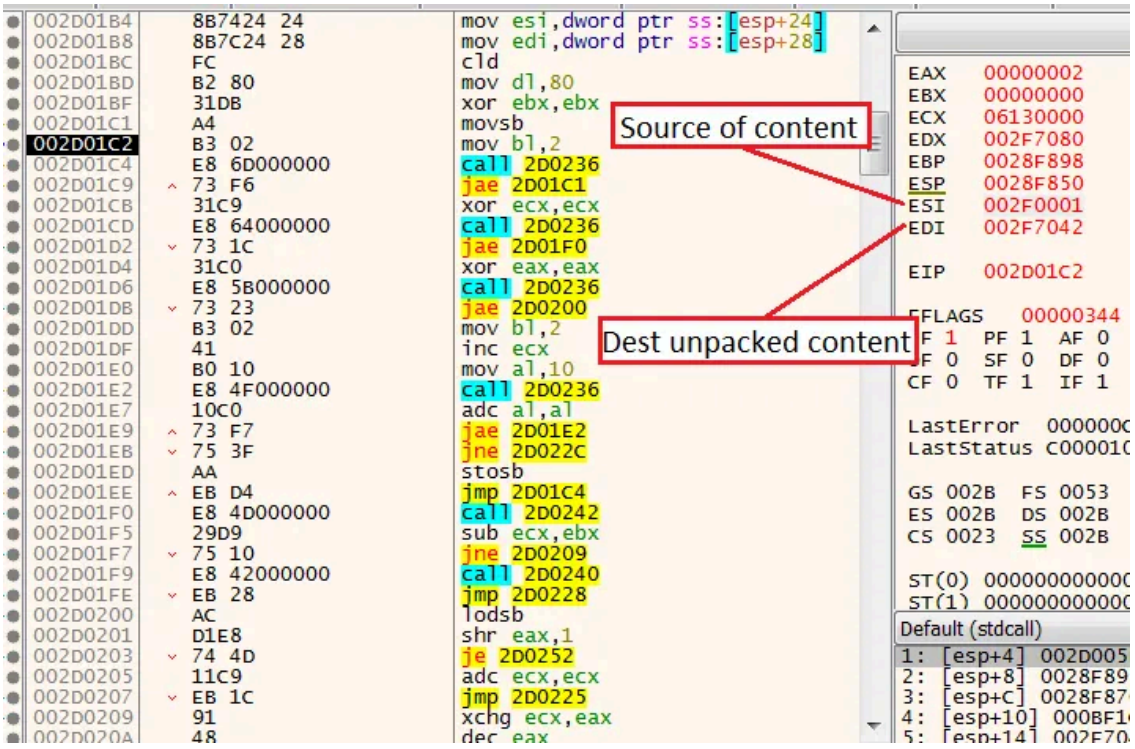


Unpacking the dropper

After clicking Run, we found ourselves in a loop that consists of several functions, this loop will be the one that decompresses the APLIB content.
In terms of decompression location, this mechanism works in the following way:

1. It will get bytes from the beginning of the APLIB content, manipulate them, and will store them in the *ESI* register.
2. It will copy the decoded content at offset 7040, the offset where the content will be written will be stored in the *EDI* register.

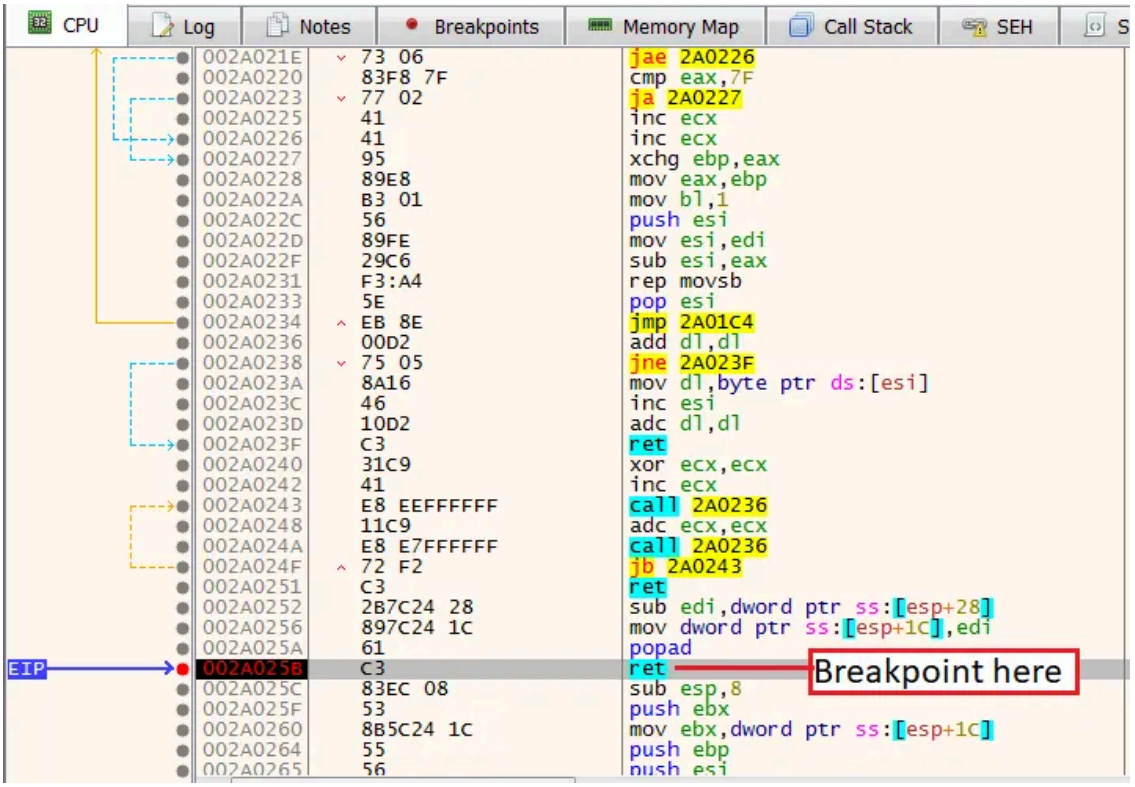
Press enter or click to view image in full size



Unpacking the dropper

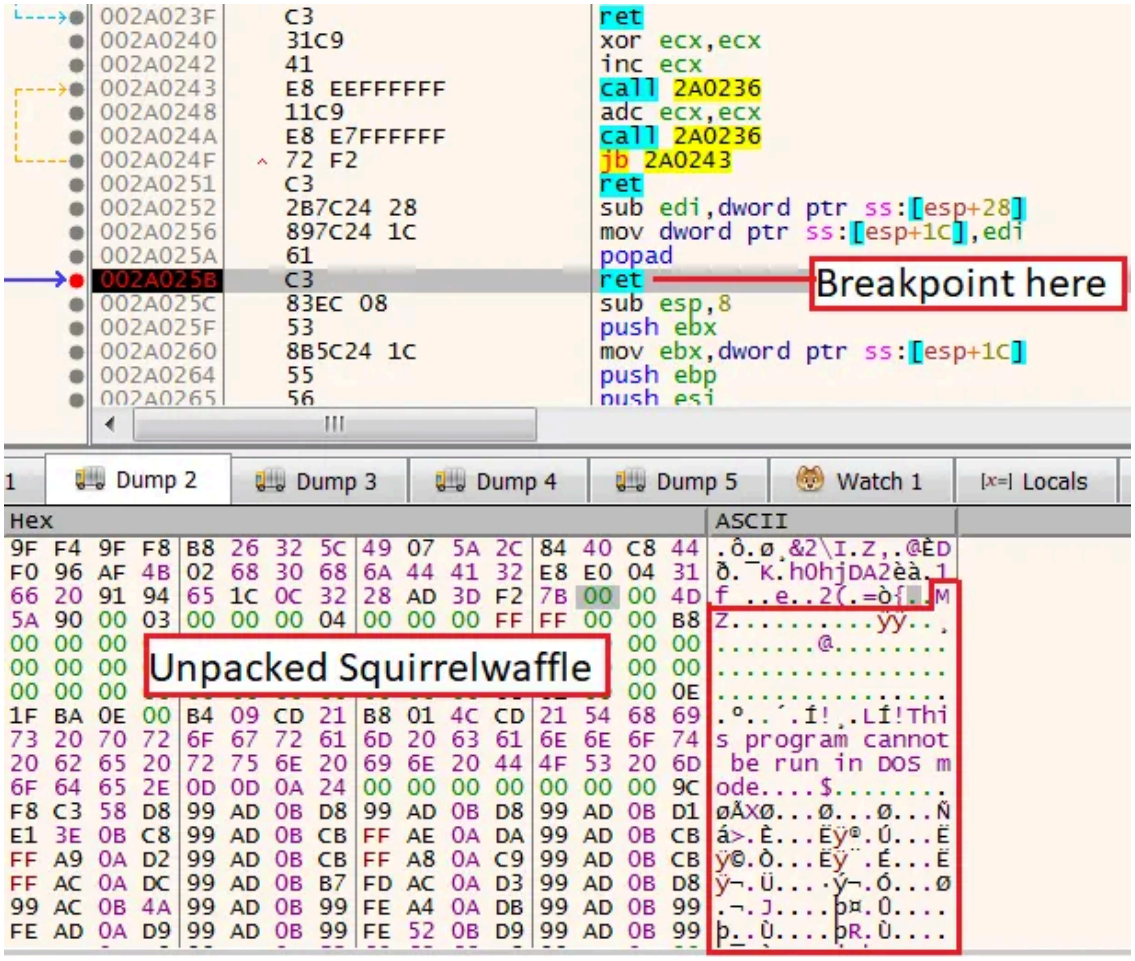
To skip the entire unpacking and decompressing process, in the loop, we can scroll down, and we'll see three ret opcodes, set a breakpoint on the third one, and hit Run.

Press enter or click to view image in full size

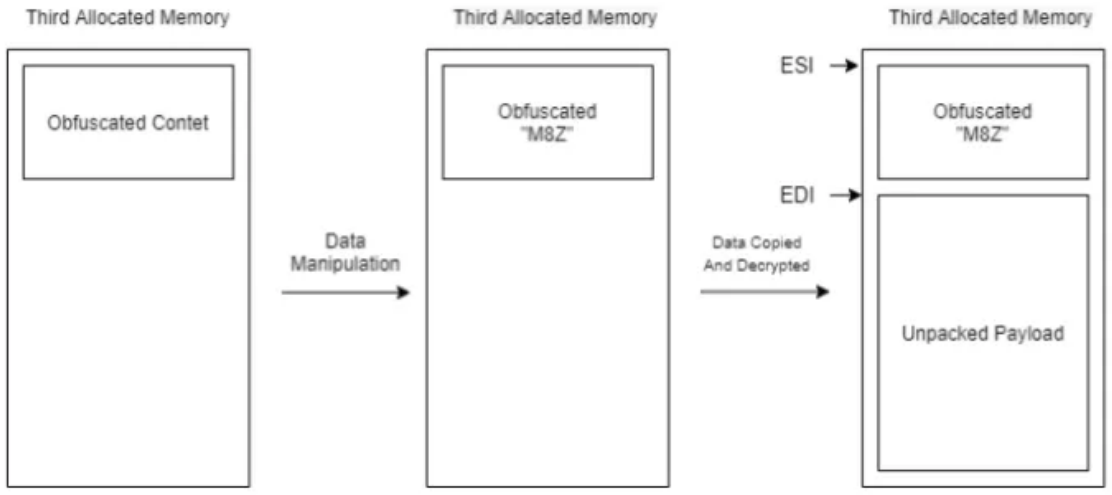


Unpacking the dropper

Now, follow in dump in the *EDI* register where we know the unpacked content should be stored. We can see now the MZ header and the unpacked SquirrelWaffle malware.



To dump it, we can mark the entire content from the MZ header until the end and save it as binary using the xdbg, or just use the pe-sieve tool.

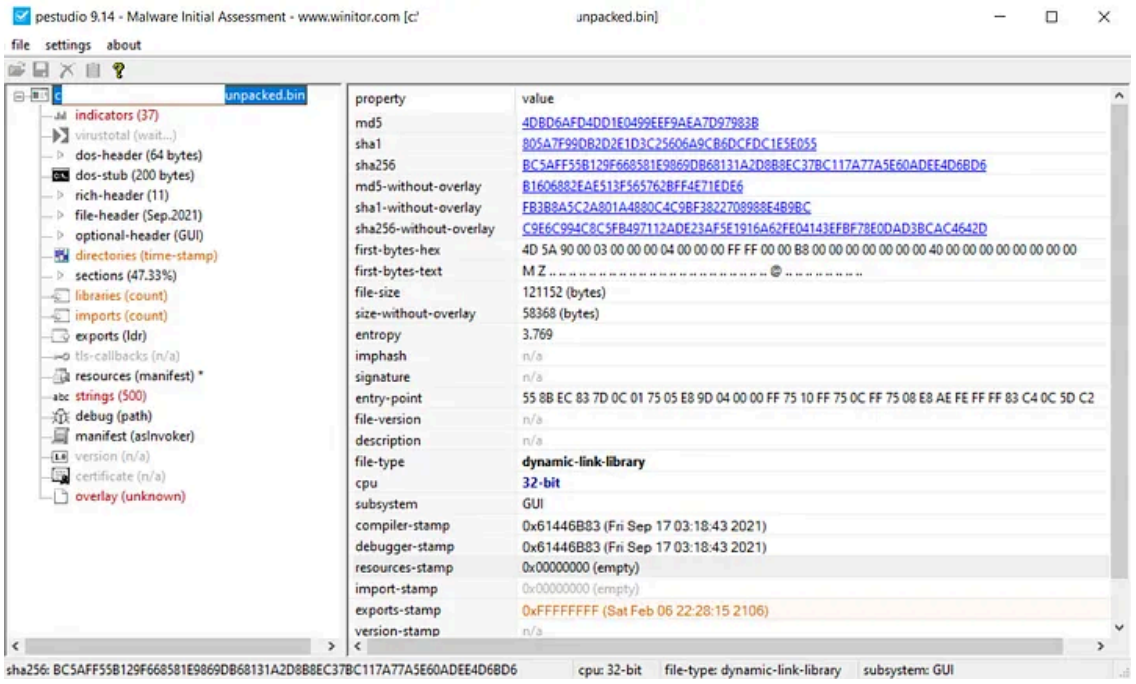


Dropper crypter unpacking

SquirrelWaffle

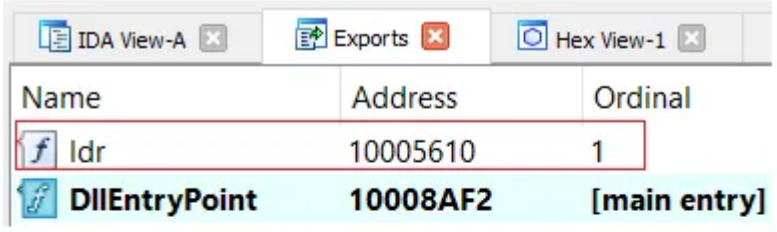
Similar to the dropper, SquirrelWaffle is also a 32-bit DLL file.

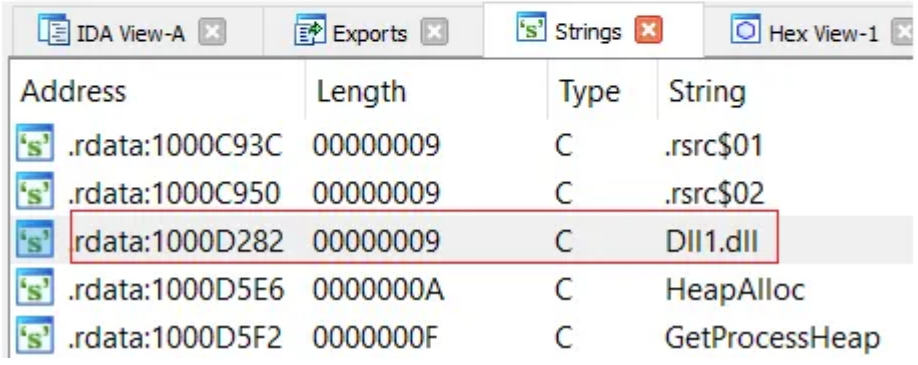
Press enter or click to view image in full size



SquirrelWaffle in PEStudio

In contrast to the dropper, this DLL file has only one export function called "*ldr*". Also, it seems that the file itself is has a DLL name in it called "*Dll1.dll*". This fixed name of a DLL file was also observed in Qbot (*stager_1.dll*), Trickbot(*templ.dll*), and IcedID (*loader_64_dll.dll*).

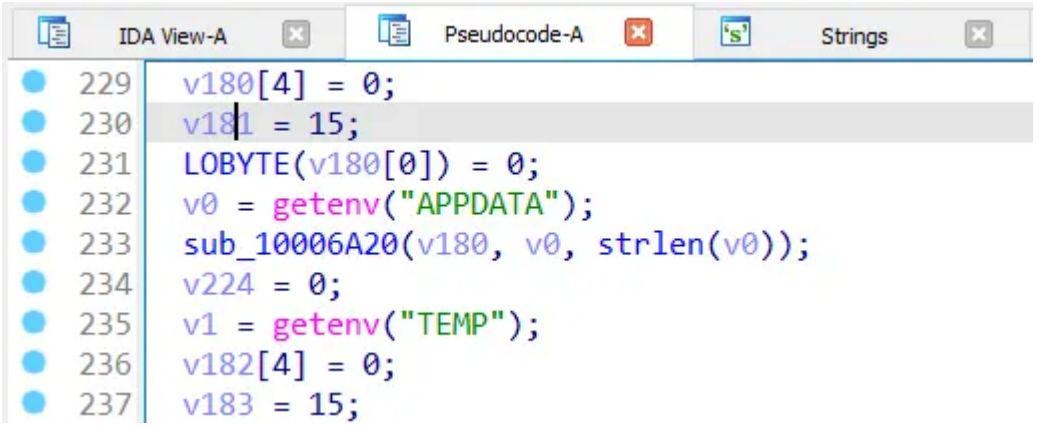




SquirrelWaffle export and DLL name

When we investigate statically the malware from the "ldr" export function, we can see that the function invokes only one very long and nested function. For this analysis, and to make tracking the article more easier, we'll

labeled it as “the core function”.
The function starts with the malware attempt to get the environment variables of the APPDATA and TEMP directories.

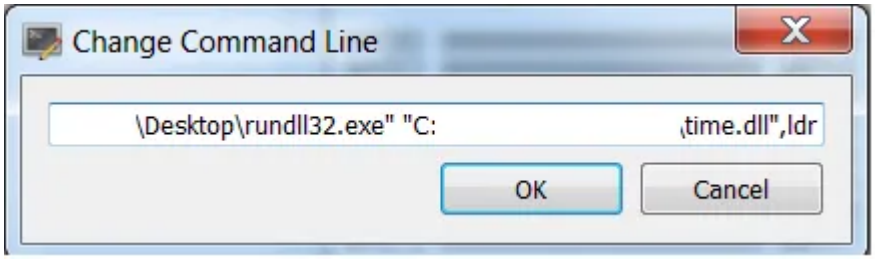


Core function begins with getting environment variables

However, some of the malware functions are not easily understandable, and some deal with content decryption. To verify it, we need to investigate dynamically.
To do so, we’ll need to start from the “ldr” export function, there are two ways to reach it.

First way

The first way will be to start the Xdbg with loading Rundll32 and assign the malware path with the *ldr* export function as an argument. (In my analysis I called the unpacked file “time.dll”)



Executing the DLL using Rundll32

However, I sometimes found this method not reliable, and the DLL file often goes to the DllEntryPoint function instead.

Second way

The second route is a cool memory trick, that will work as the following:

1. we’ll first go to the *DllEntryPoint*
2. We already know from the first glance of static investigation that the ldr function should start with *getenv()* function that searched the APPDATA and TEMP environment variables.
3. Because the APPDATA and TEMP strings are hardcoded in the malware we’ll search for their location.

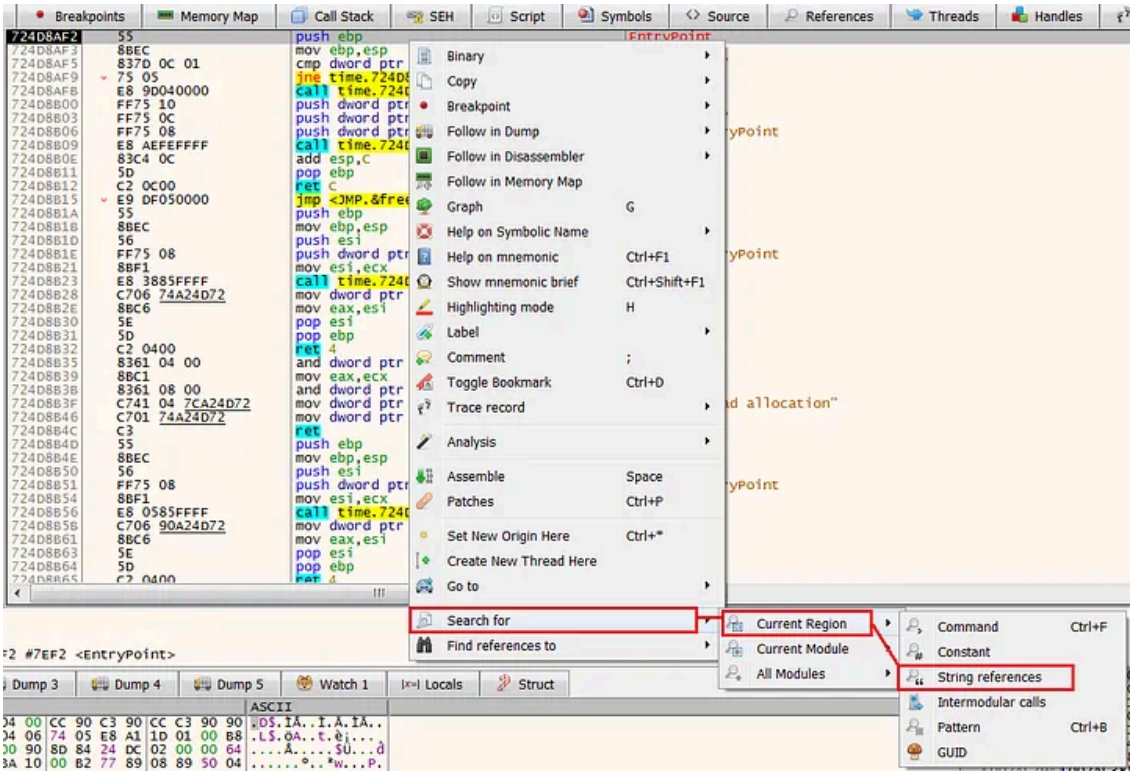
4. we'll direct our malware execution flow to go directly to the location of the function that the APPDATA and TEMP are found.

Getting the location of the APPDATA & TEMP function

To do so, once we are in the *DllEntryPoint*, do:

1. Right click
2. Search for
3. Current region
4. String references

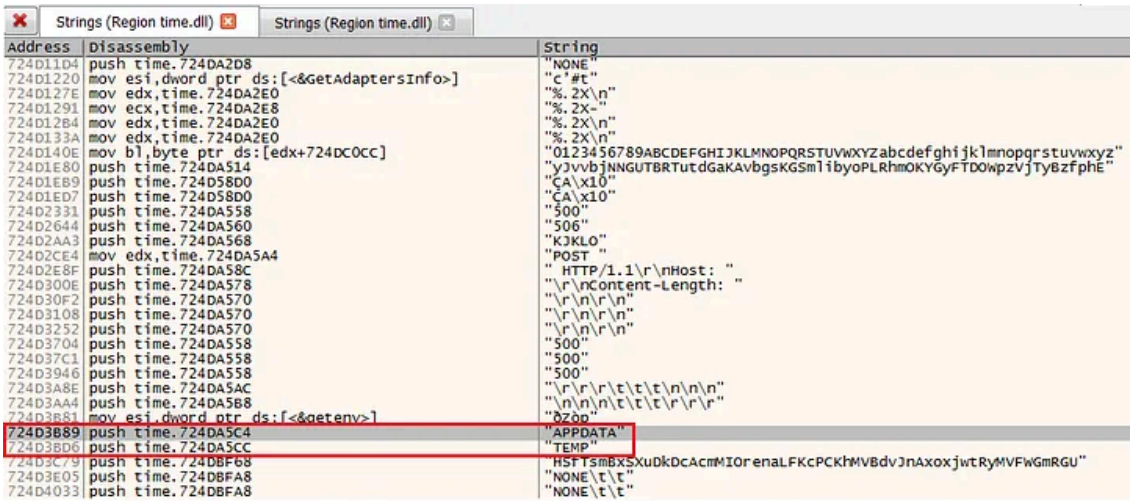
Press enter or click to view image in full size



Getting the string references

Now, we found ourselves with the entire list of the hardcoded strings of the malware, there we can also see our APPDATA & TEMP strings. let's click on one of them.

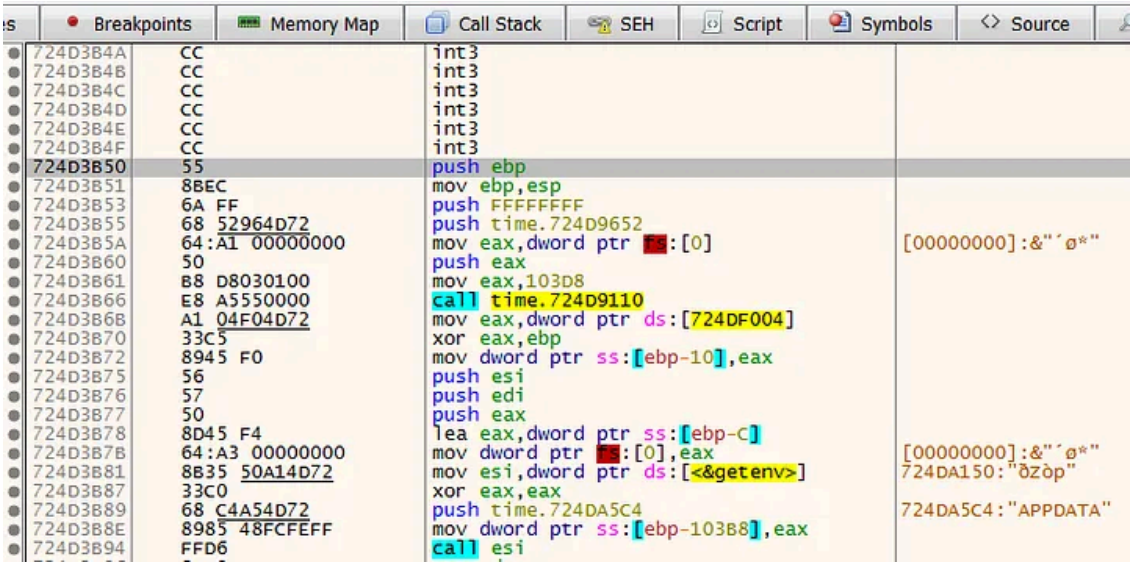
Press enter or click to view image in full size



Hardcoded strings

Once we click, we can see the places where the APPDATA and *getenv()* function will be executed, this is also the function that the ldr export function calls aka the core function.
This is the place we want to start our dynamic investigation, and therefore, we would want to direct the malware to start at the beginning of this function.

Press enter or click to view image in full size



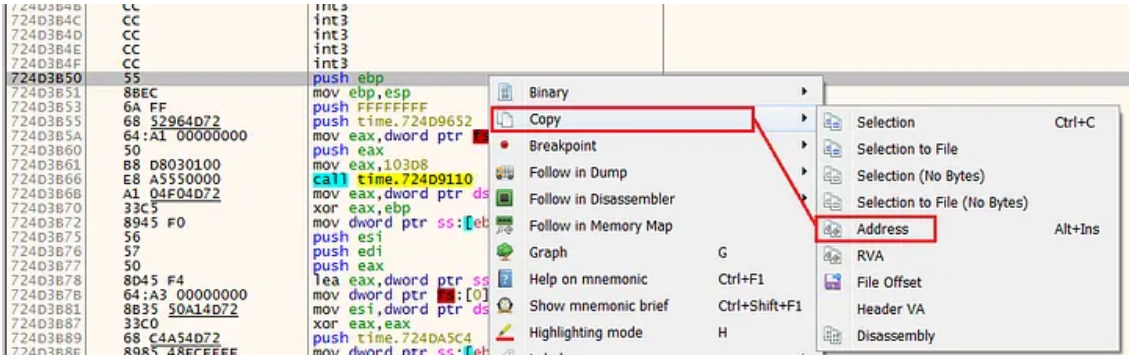
Begining of core function

Changing the malware execution flow

To change the direction of the malware to start in this function, do the following:

1. Right click on the first function line of code
2. Copy
3. Address

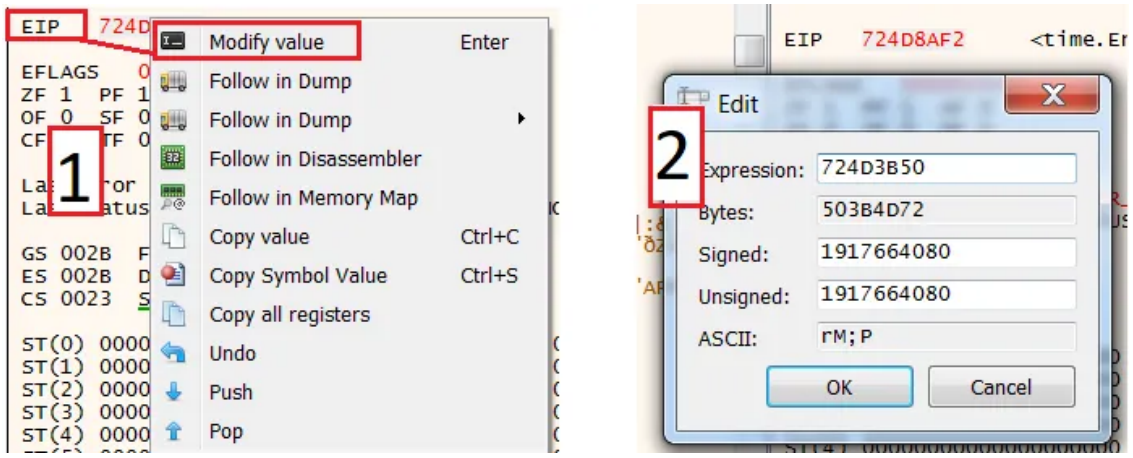
Press enter or click to view image in full size



Getting the address of the core function

Now, at the right side of the debugger, you can see the *EIP* register which is responsible for holding the next instruction to be executed, we'll want to manipulate it.
To do so, do the following:

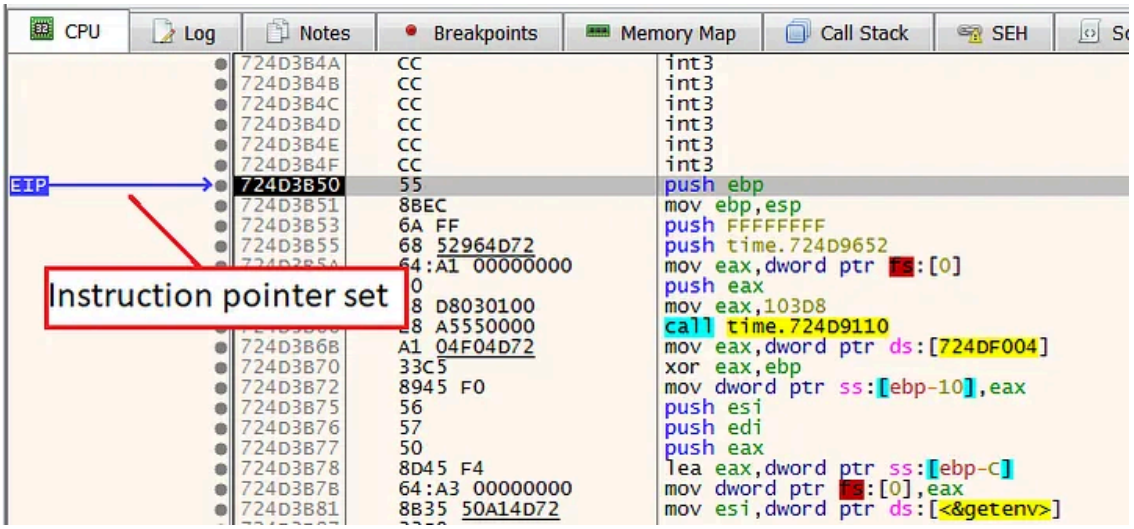
1. Right-click on the *EIP* register
2. Modify value
3. In the Expression box, paste the address you copied.
4. Click OK



Changing the EIP

After clicking OK we can see that the instruction pointer was changed to the start of the core function, now our dynamic analysis can be performed.

Press enter or click to view image in full size



Execution flow now at the start of the core function

Core function

As mentioned, the ldr export function leads us to one specific big function which we call “the core function. This function will have several objectives and will also call to other functions that will take part in this malware download mechanism.

Observable functions

Before digging into the more challenging functions, let's talk about the more visible API calls that this function consists of.
The majority of these functions are aimed to collect information about the infected machine.
As already mentioned, the malware attempt to collect information about the environment variables *c:\users\user\appdata\roaming* and *c:\users\user\appdata\local\temp* using the *getenv()* function.

```
v0 = getenv("APPDATA");
sub_10006A20(v180, v0, strlen(v0));
v224 = 0;
v1 = getenv("TEMP");
```

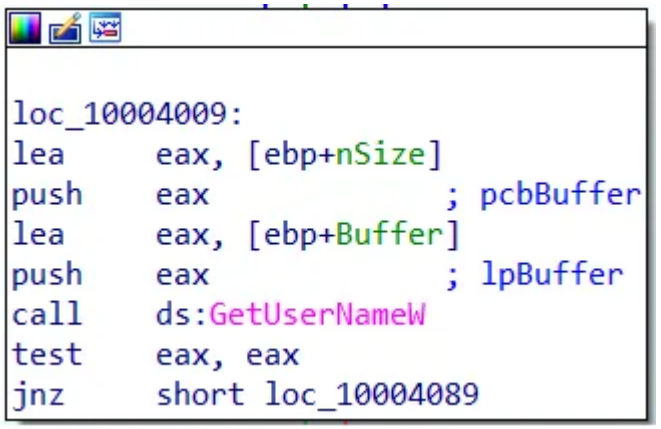
Getting environment variables

The malware also attempts to collect the name of the local computer using the *GetComputerNameW()* function.

```
push    eax             ; nSize
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
call    ds:GetComputerNameW
test    eax, eax
jnz     short loc_10003E5B
```

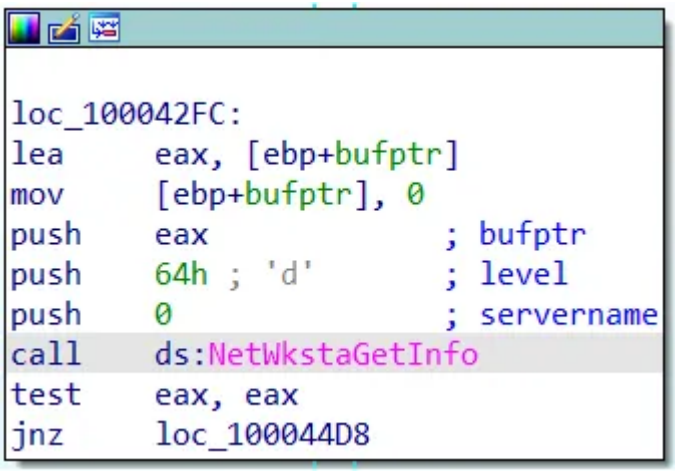
Getting the machine name

The malware then attempts to get the machine’s user name using the *GetUserNameW*() function.



Getting the user name

The malware will extract information about the configuration of a workstation using the function NetWkstaGetInfo().



Getting info on workstation's configuration

Maintenance functions

As we enter the malware's core function, we observe a function named "*sub_10006A20*" (name can change with other instances) that will repeat itself multiple times during the malware's execution.
This function receives three arguments, two pointers, and a length, the function will copy the data from one pointer and assign it to another.

Get Eli Salem's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

For example, in the first iteration, we can see sub_10006A20 do the following:

1. gets the pointer of environment variable stored in v0
2. gets the environment variable length

3. copy the data into v180

```
v0 = getenv("APPDATA");
sub_10006A20(v180, v0, strlen(v0));
```

sub_10006A20 Copy from v0 to v180

In addition, we also see the function being used four times at the beginning of the malware.

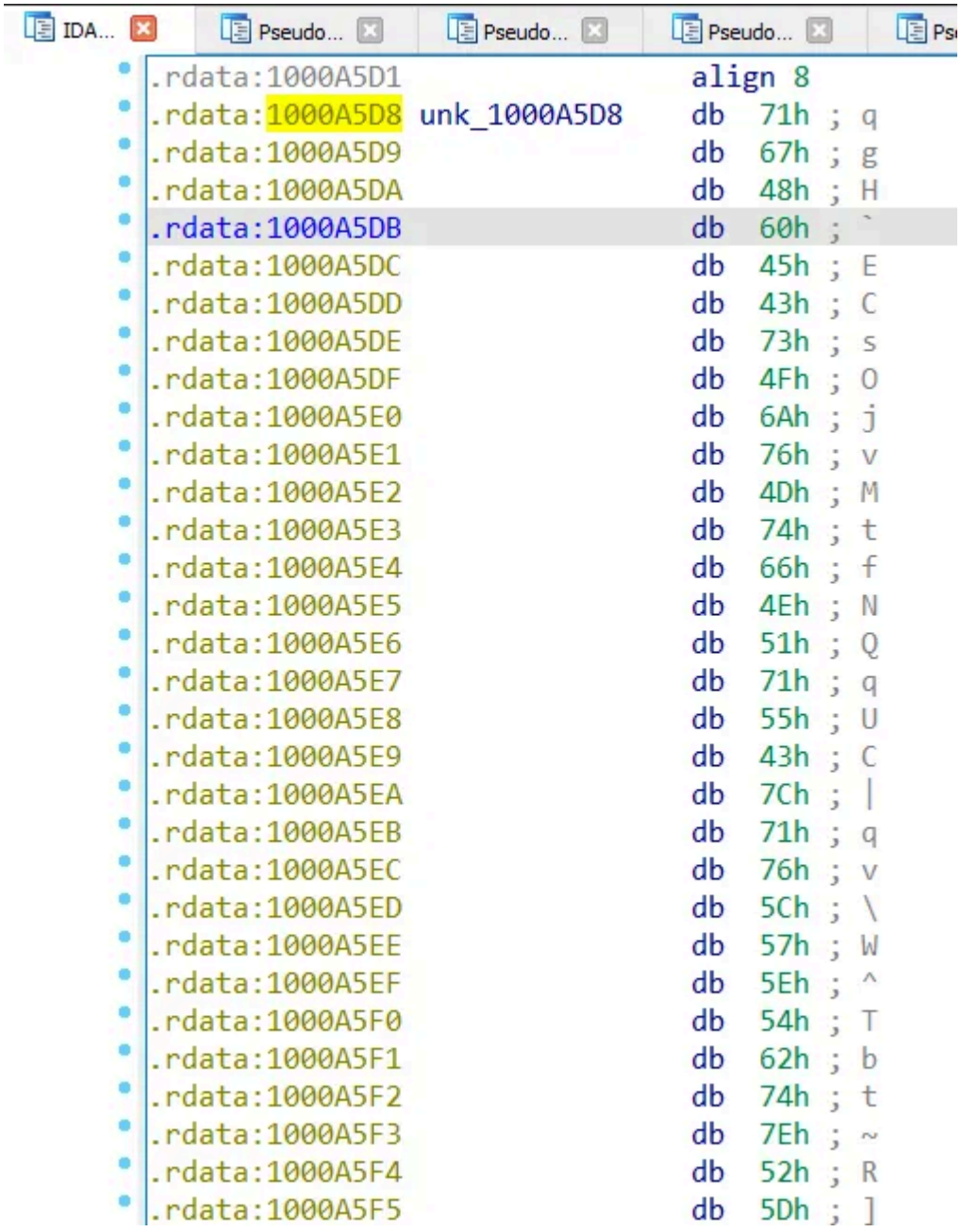
1. The first two iterations will copy the environment variables as mentioned.
2. The third iteration will copy a large chunk of code "*unk_100A5D8*" which be later discovered as the malware's config.
3. The fourth iteration will copy a hardcoded string which will take part in the config decryption part.

Press enter or click to view image in full size

```
LOBYTE(v180[0]) = 0;
v0 = getenv("APPDATA");
sub_10006A20(v180, v0, strlen(v0));
v224 = 0;
v1 = getenv("TEMP");
v182[4] = 0;
v183 = 15;
LOBYTE(v182[0]) = 0;
sub_10006A20(v182, v1, strlen(v1));
LOBYTE(v224) = 1;
v189 = 0;
v190 = 15;
LOBYTE(Src[0]) = 0;
sub_10006A20(Src, &unk_1000A5D8, 0x198Eu);
LOBYTE(v224) = 3;
v171 = &v162;
v166 = 0;
v167 = 15;
LOBYTE(v162) = 0;
sub_10006A20(&v162, "HSfTsmBxSXuDkDcAcmMIOrenaLFKcPCKhMVBdvJnAxoxjwtRyMVFWGmRGU", 0x3Au);
LOBYTE(v224) = 4;
sub_100058F0(&v156, Src);
LOBYTE(v224) = 3;
v2 = sub_100019B0(v156, v157, v158, v159, v160, v161, v162, v163, v164, v165, (int)v166, v167);
sub_100057D0(Src, v2);
```

sub_10006A20 usages

unk_100A5D8 array of bytes:



unk_100A5D8

Another function that is interesting is *sub_100058F0*. This function will copy the data from Src into the variable 156 (internally it will do it using memcpy, the *memcpy* function is very common in this sample).

We can see that the Src argument is the copied config that was stored in *unk_1000A5D8*.

Press enter or click to view image in full size

```
LOBYTE(v180[0]) = 0;
v0 = getenv("APPDATA");
sub_10006A20(v180, v0, strlen(v0));
v224 = 0;
v1 = getenv("TEMP");
v182[4] = 0;
v183 = 15;
LOBYTE(v182[0]) = 0;
sub_10006A20(v182, v1, strlen(v1));
LOBYTE(v224) = 1;
v189 = 0;
v190 = 15;
LOBYTE(Src[0]) = 0;
sub_10006A20(Src, &unk_1000A5D8, 0x198Eu);
LOBYTE(v224) = 3;
v171 = &v162;
v166 = 0;
v167 = 15;
LOBYTE(v162) = 0;
sub_10006A20(&v162, "HSfTsmBxSXuDkDcAcmMIOrenaLFKcPCKhMVBdvJnAxoxjwtRyMVFWGmRGU", 0x3Au);
LOBYTE(v224) = 4;
sub_100058F0(&v156, Src);
LOBYTE(v224) = 3;
v2 = sub_100019B0(v156, v157, v158, v159, v160, v161, v162, v163, v164, v165, (int)v166, v167);
sub_100057D0(Src, v2);
```

sub_100058F0

Then, the copied content (156) will be sent to the function *sub_100019B0* which will deal with the config decryption.

Press enter or click to view image in full size

```
sub_10006A20(Src, &unk_1000A5D8, 0x198Eu);
LOBYTE(v224) = 3;
v171 = &v162;
v166 = 0;
v167 = 15;
LOBYTE(v162) = 0;
sub_10006A20(&v162, "HSfTsmBxSXuDkDcAcmMIOrenaLFKcPCKhMVBdvJnAxoxjwtRyMVFWGmRGU", 0x3Au);
LOBYTE(v224) = 4;
sub_100058F0(&v156, Src);
LOBYTE(v224) = 3;
v2 = sub_100019B0(v156, v157, v158, v159, v160, v161, v162, v163, v164, v165, (int)v166, v167);
sub_100057D0(Src, v2);
```

sub_100019B0 deals with config decryption

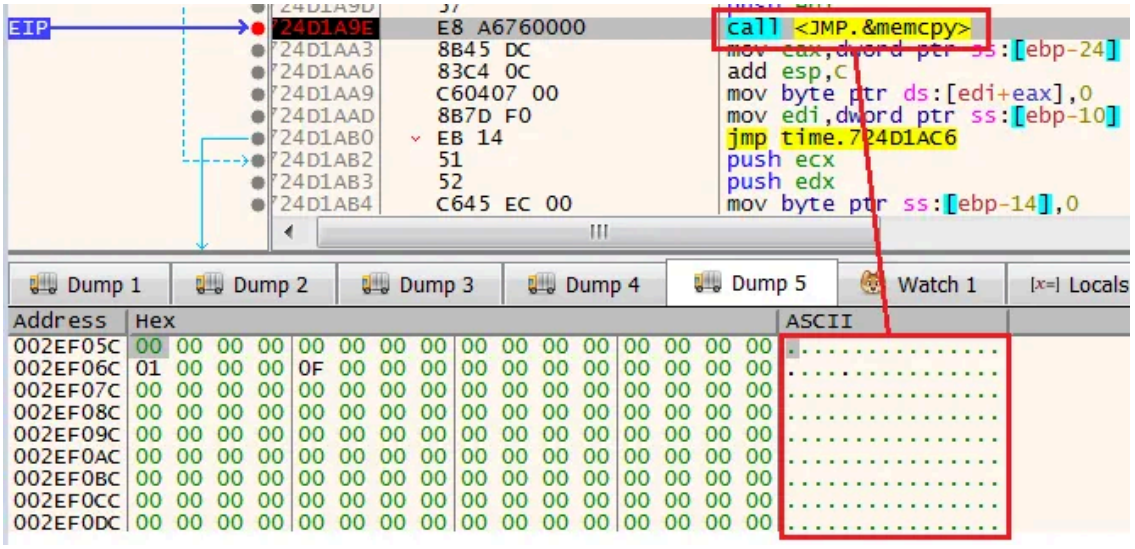
There are two ways to get the config, one of them is very trivial and easy, but where is the fun in that? (we'll discuss this way at the end of this config section).

Observing the config decryption

If we want to observe some key features of the config decryption mechanism we'll have to jump inside *sub_100019B0*.

When we step into the function dynamically, we can see several xor and copying activities, which eventually lead us to a memcpy function that will write the IP addresses.

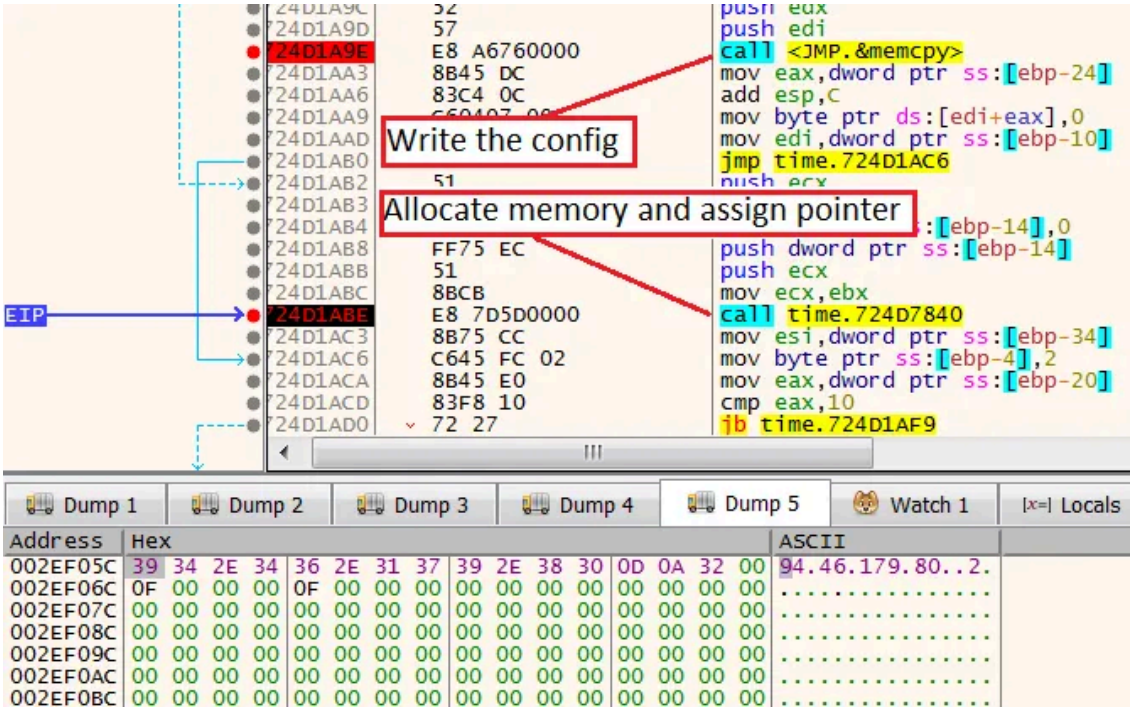
To observe it, we can just follow in dump on the *EDI* register.



memcpy writing the IP adress

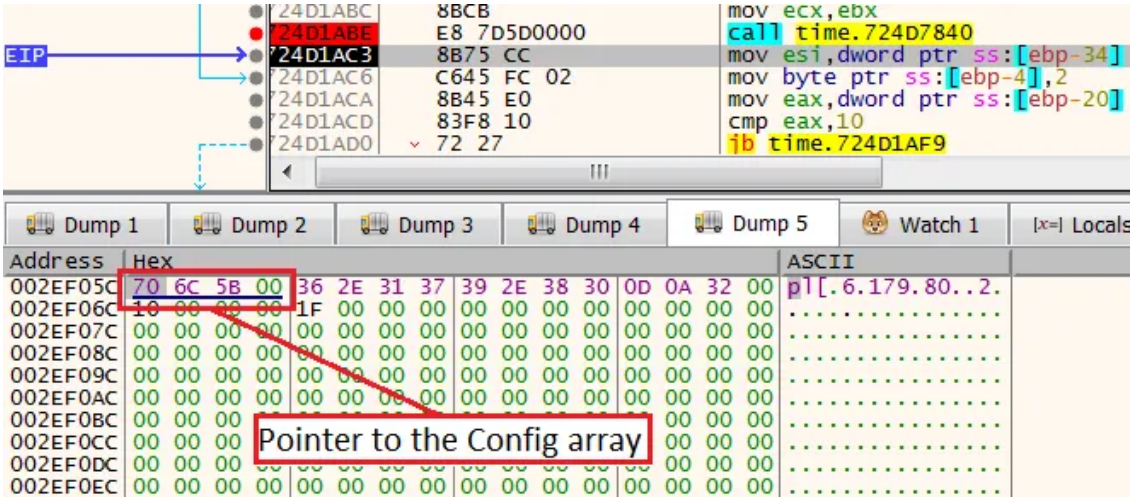
Then, the malware will check the size of the written content, allocate new memory using *Malloc* and assign pointer to it.
This small allocation and pointer assign activity will happen in the function *724D7840* (in the followed image).

Then, the first four bytes will be changed to the address that will contain the new buffer of the IP addresses.



Before executing 724D7840

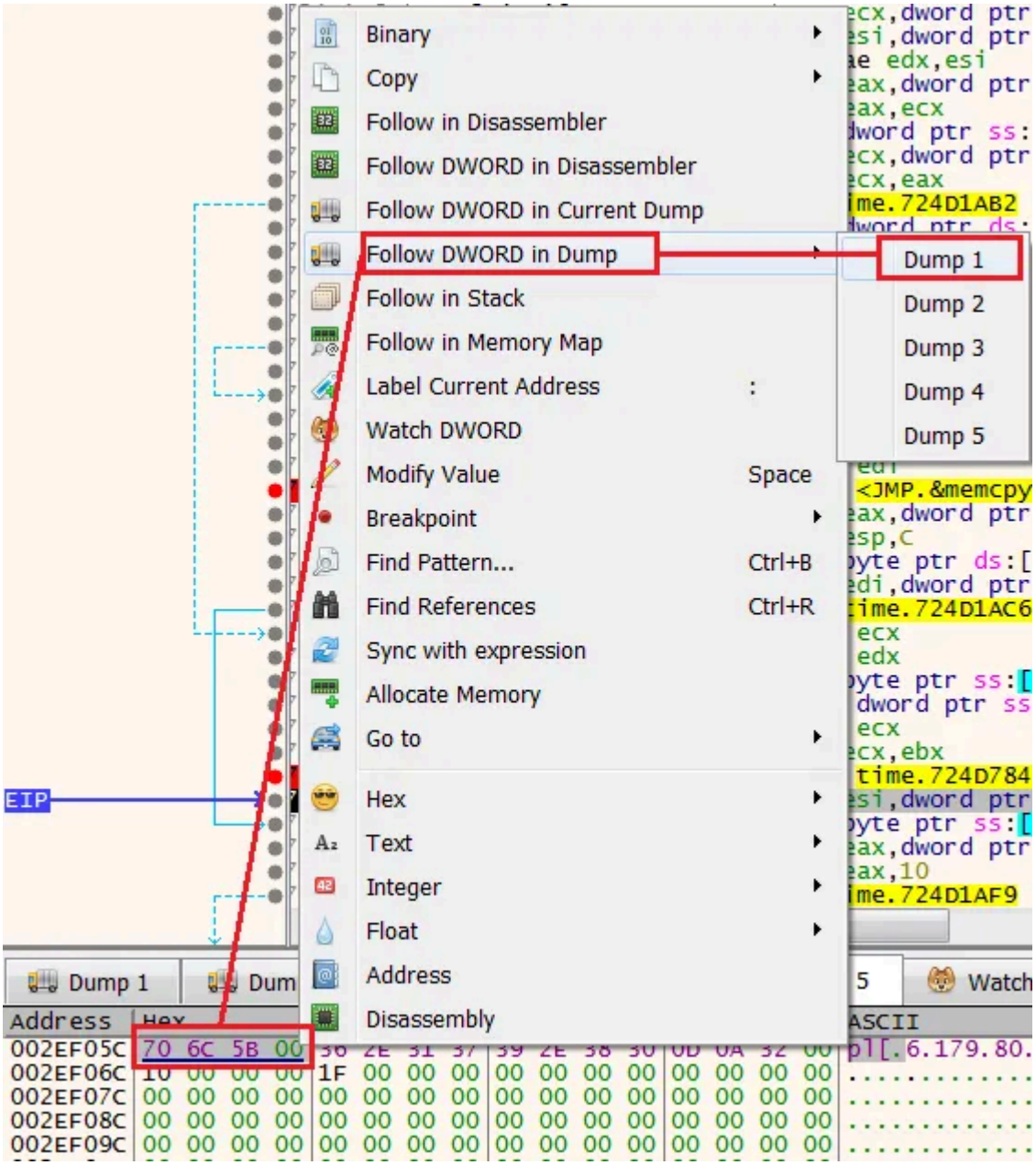
As expected, right after passing the function, we can see that the first four bytes have been changed to be a pointer.



After executing 724D7840

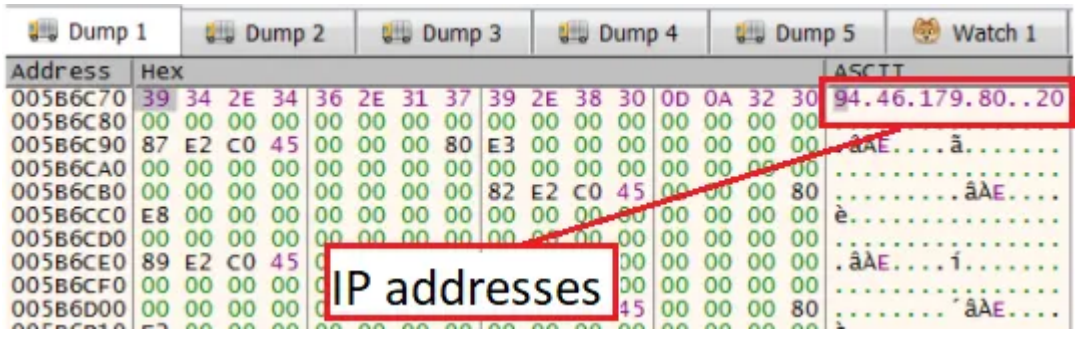
If we want to see the array of IP addresses that this pointer points to, do the following:

1. Right-click on the pointer
2. Follow DWORD in Dump
3. Select your preferable dump



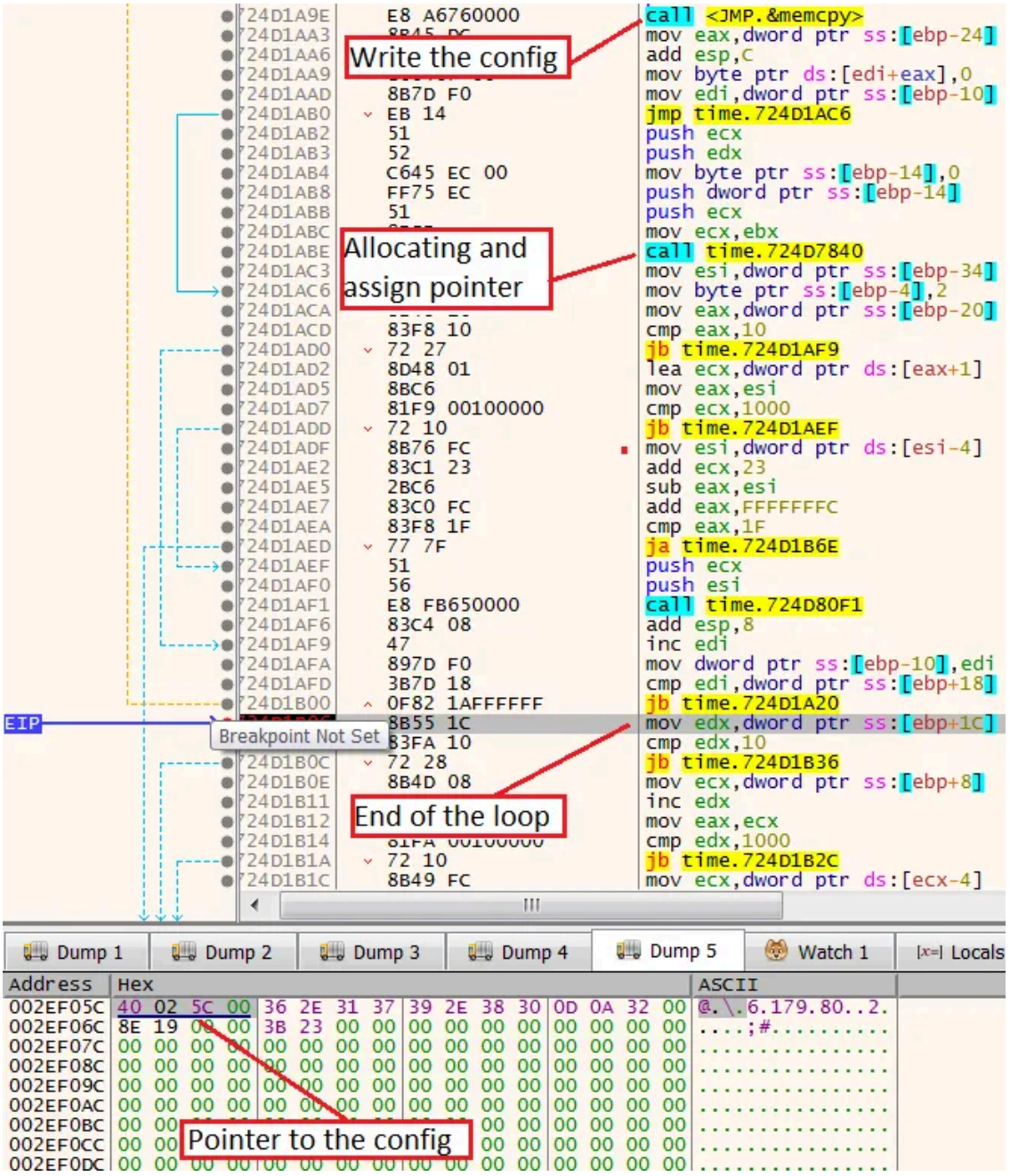
Following in the pointer dump

Once we click, we could see the array of IP addresses



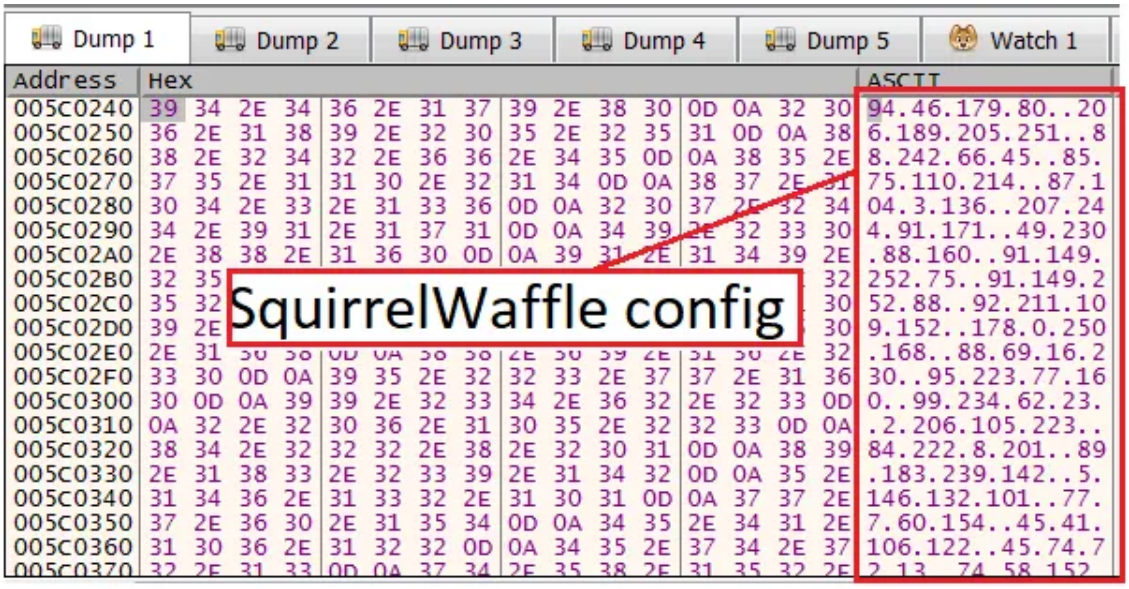
IP addresses array

This activity of writing and assigning a new address for the config will happen several times inside a loop, therefore, to skip it we would want to set a breakpoint right after the loop.



Breaking after the config decryption loop ends

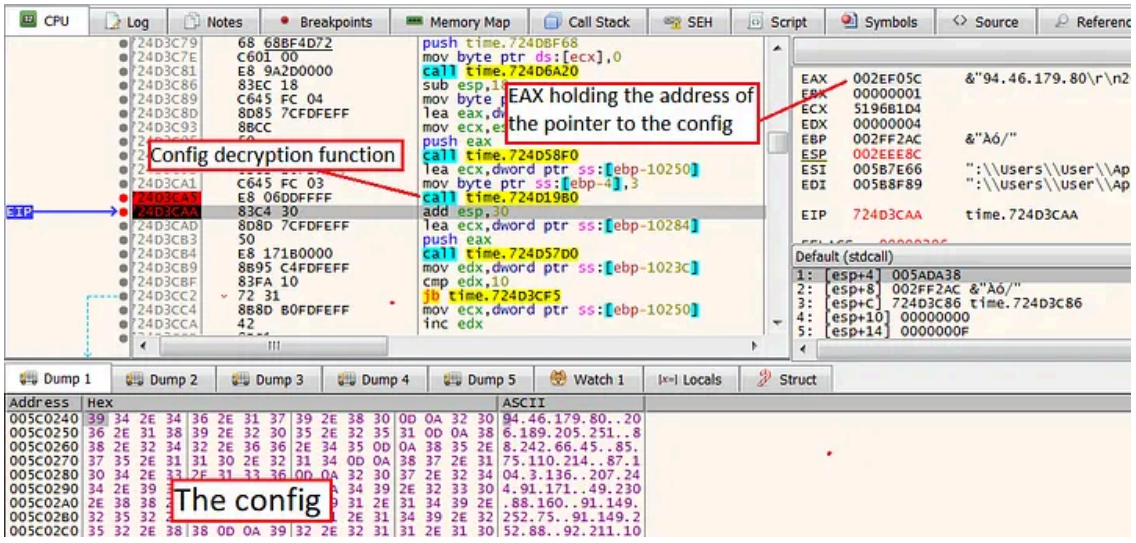
And just like before, we can click on the pointer and follow in dump to see the config.



After getting the config, we can get out of the entire function.

Remember I said there is an easier way to get the config? well, when *sub_100019B0* ends, it returns (in the *EAX* register) the address of the pointer to the config.

Press enter or click to view image in full size



After sub_100019B0 ends, the EAX register holds the config

To recap, the start of the core function and config extraction can be seen in this pseudo code.

Press enter or click to view image in full size

```
v0 = getenv("APPDATA");                 // return c:\users\username\appdata\roaming
sub_10006A20(v180, v0, strlen(v0));     // copy v0 to v180
v224 = 0;
v1 = getenv("TEMP");                    // return c:\users\username\appdata\local\temp
v182[4] = 0;
v183 = 15;
LOBYTE(v182[0]) = 0;
sub_10006A20(v182, v1, strlen(v1));     // copy v1 to v182
LOBYTE(v224) = 1;
v189 = 0;
v190 = 15;
LOBYTE(Src[0]) = 0;
sub_10006A20(Src, &unk_1000A5D8, 0x198Eu);    // unk_1000A5D8 is the obfuscated config
                                              // It will be copied to Src
LOBYTE(v224) = 3;
v171 = &v162;
v166 = 0;
v167 = 15;
LOBYTE(v162) = 0;
sub_10006A20(&v162, "HSfTsmBxSXuDkDcAcmMIOrenaLFKcPCKhMVBdvJnAxoxjwtRyMVFWGmRGU", 0x3Au);// The string will be copied to v162
LOBYTE(v224) = 4;
sub_100058F0(&v156, Src);               // copy config to 156
LOBYTE(v224) = 3;
v2 = sub_100019B0(v156, v157, v158, v159, v160, v161, v162, v163, v164, v165, (int)v166, v167);// Config Decryption
                                              // V2 (EAX register) will store the address of the pointer to the config
```

Recap

Now that we are more familiar with the malware “maintenance functions”, we can speed things up.

Network function

One of the interesting functions is *sub_10001DB0*, which appears within the largest loop in the core function. To easily locate this function in your code, search for a *sleep()* function with *0x5DC0* as an argument.
sub_10001DB0 will be the function that responsible for SquirrelWaffle’s network activity.

Press enter or click to view image in full size

```
while ( 1 )
{
  Sleep(0x5DC0u);
  v172 = &v162;
  sub_100058F0(&v162, v197);
  LOBYTE(v224) = 21;
  v160 = 0;
  v161 = 15;
  LOBYTE(v156) = 0;
  sub_10006A20(&v156, &unk_1000A2D5, 0);
  LOBYTE(v224) = 20;
  v60 = (void **)sub_10001DB0(v156, v157, v158, v159, v160, v161, v162, v163, v164, v165, (size_t)v166, v167);// Network Function
  if ( v200 != v60 )
  {
    if ( HIDWORD(v201) >= 0x10 )
    {
```

Network function

Right after entering the function, we encounter the familiar function *sub_100019B0*, which as we remember also used to decrypt the config.
It also seems to follow a similar pattern of the config extraction:

1. *sub_10006A20* receive embedded hardcoded content and copy it to he memory.
2. Another *sub_10006A20* function recieve long hardcoded string.
3. *sub_100058F0* take the copied content and assign it
4. *sub_100019B0* take the pointer of the obfuscated content, and the hardcoded string as an arguments.

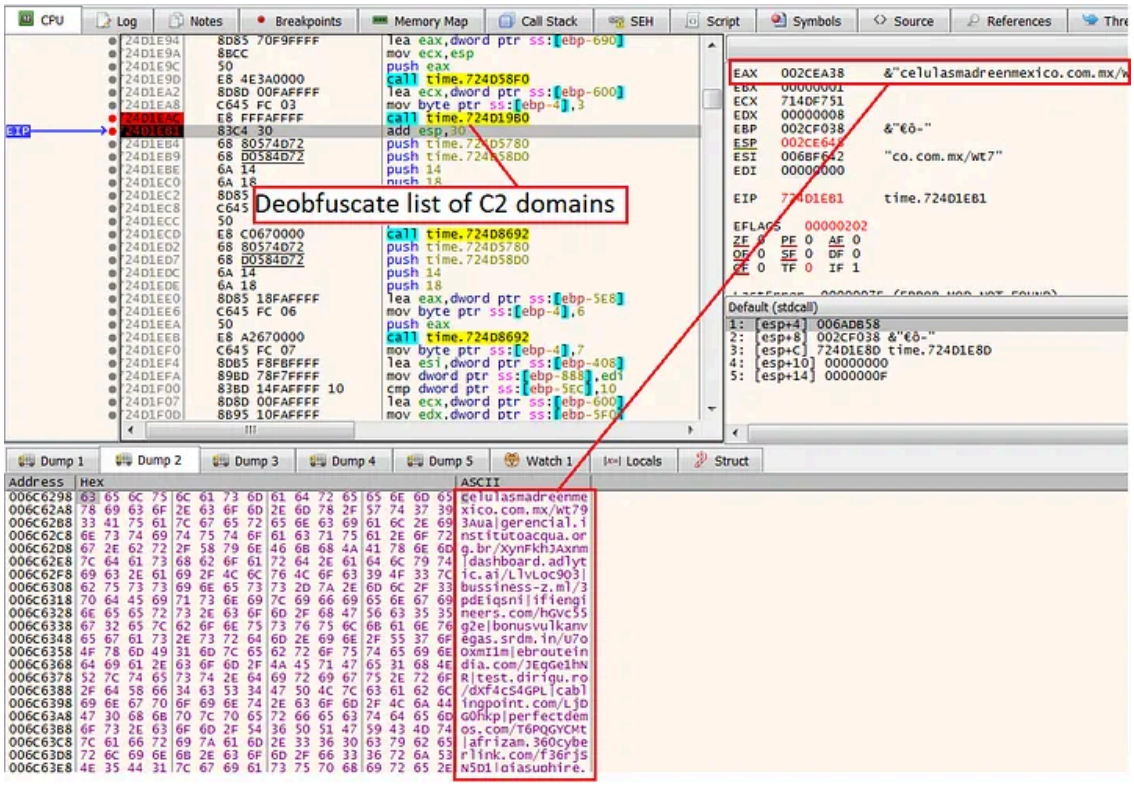
Press enter or click to view image in full size

```
Time = time64(0);
buf = (char *)localtime64(&Time);
s = -1;
ppResult = 0;
v252[4] = 0;
v253 = 15;
LOBYTE(v252[0]) = 0;
sub_10006A20(v252, &unk_1000A338, 0x1DBu);
LOBYTE(v272) = 3;
v239 = &v205;
v209 = 0;
v210 = 15;
LOBYTE(v205) = 0;
sub_10006A20(&v205, "yJvvbjNNGUTBRTutdGaKAvbgsKGSmlibyoPLRhmOKYGyFTDOWpzVjTyBzfphE", 0x3Du);
LOBYTE(v272) = 4;
sub_100058F0(&v199, v252);
LOBYTE(v272) = 3;
sub_100019B0(v199, v200, v201, v202, v203, v204, v205, v206, v207, v208, (int)v209, v210);// Config decryption
                                                // This time Domains instead of IP addresses
```

sub_100019B0 in the Network Function

Because we have already seen this pattern, we remember that if we step over *sub_100019B0* we would see some content returned. Interestingly, now the content is a list of C2 domains.

Press enter or click to view image in full size



Returned list of C2 domains

After collecting the C2 domains and IP addresses the malware can finally communicate externally. The communication is done using the classic *WS_32* API calls.

The malware first create a *socket*, send the data using *send()* and receive information from the C2 using *recv()*.

```
v192 = strlen(buf);
v193 = s;
if ( send(s, buf, v192, 0) == -1 || shutdown(v193, 1) == -1 )
{
  closesocket(v193);
  WSACleanup();
  sub_10005890(v240, "500");
  v243 = v157 | 1;
}
else
{
  sub_10005890(v248, &unk_1000A2D5);
  LOBYTE(v272) = 39;
  while ( 1 )
  {
    v194 = recv(v193, v270, 512, 0);
    if ( v194 <= 0 )
      break;
    for ( i = 0; i < v194; ++i )
    {
      LOBYTE(buf) = v270[i];
      sub_10006860(v248, (char)buf);
```

SquirrelWaffle communication functions

Once the network function finishes its activity its returns to the core function, then we start to see signs and clues about the content to be download.

Final payload

As mentioned by several security researchers, SquirrelWaffle aims to download the Cobalt-Strike framework. Once downloaded, the SquirrelWaffle store the binary as a .txt file. A maybe possible indication of this activity can be seen in the code with the hardcoded “.txt” strings.

```
if ( v117 - v118 < 4 )
{
  LOBYTE(v169) = 0;
  v116 = sub_10007840(v116, 4, v169, (int)".txt", 4u);
}
else
{
  v116[4] = v118 + 4;
  v119 = v116;
  if ( v117 >= 0x10 )
    v119 = (_DWORD *)*v116;
  v120 = (char *)v119 + v118;
  memmove((char *)v119 + v118, ".txt", 4u);
  v120[4] = 0;
}
```

Indication of .txt extension

The malware has execution capabilities using the WinExec function. The function itself can be executed in three different locations during the malware execution.

```
push    0               ; uCmdShow
push    eax             ; lpCmdLine
call    ds:WinExec
```

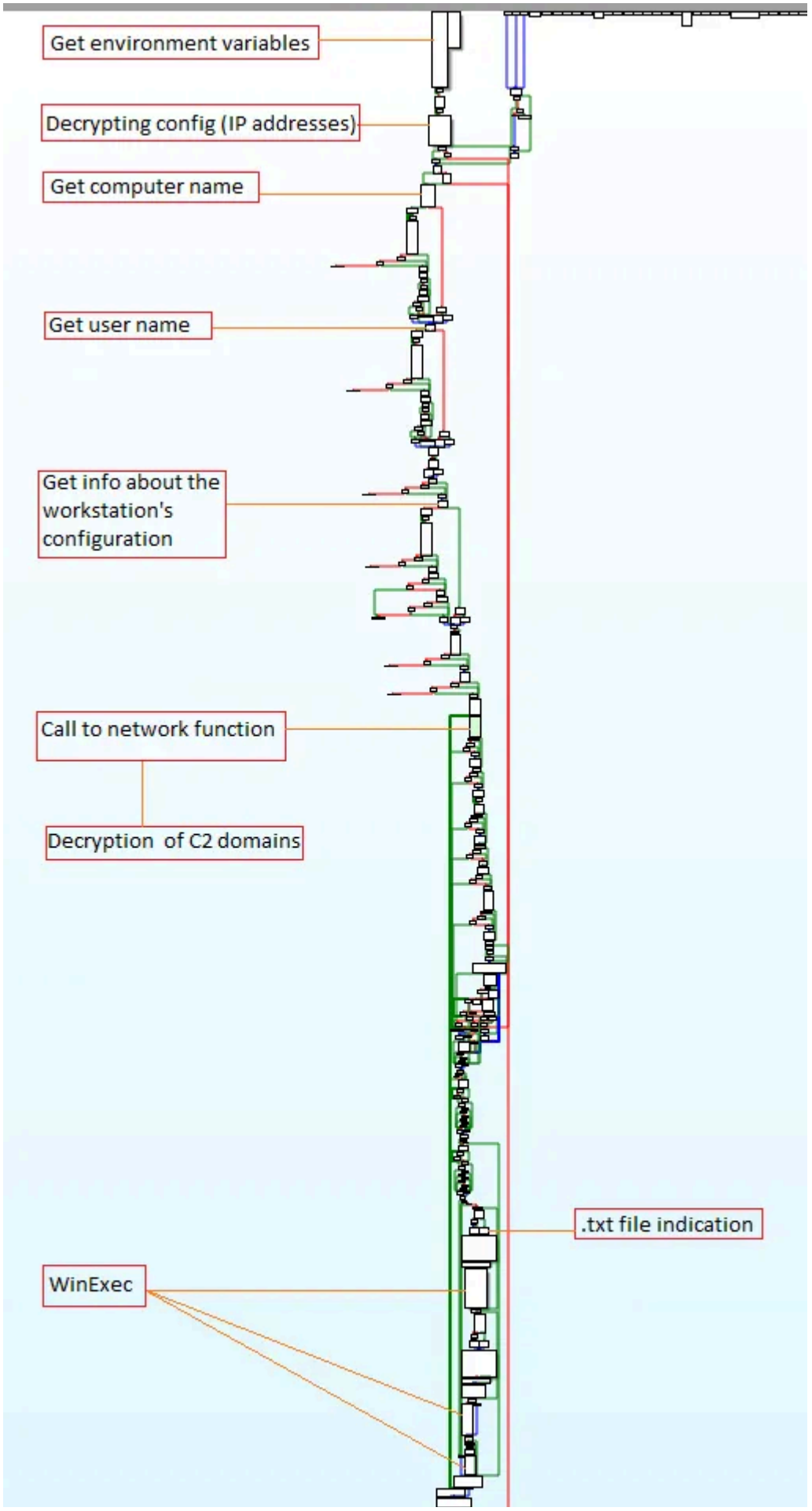
WinExec

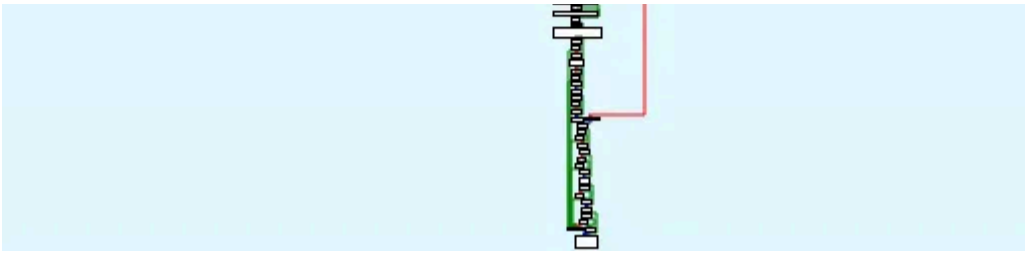
Recap

In this technical analysis, we discussed multiple topics

1. SquirrelWaffle dropper and how to unpack it
2. SquirrelWaffle core function
3. SquirrelWaffle network capabilities as a downloader
4. How to observe the SquirrelWaffle list of C2 domains and IP addresses

The entire analysis flow can also be seen in the following graph:


Get environment variables
Decrypting config (IP addresses)
Get computer name
Get user name
Get info about the workstation's configuration
Call to network function
Decryption of C2 domains
.txt file indication
WinExec



Conclusion and thoughts

In this article, I presented the newly emerged malware downloader SquirrelWaffle. Although SquirrelWaffle is “the new kid on the block” in the cybercrime ecosystem, its dropper already using a very known crypter that was used by other famous malware.
These findings raise the question of whether the threat actor behind SquirrelWaffle is an already known group. Also, many ransomware attacks have initially started after a successful deployment of the Cobalt-Strike framework, it will be interesting to see how many ransomware attacks will happen because of infiltration by SquirrelWaffle, and which ransomware group will operate with it.
Furthermore, because this malware is new, more features are most likely to be discovered in the near future. It will be interesting to track this malware evolution as times goes on.

References:

1. https://twitter.com/malware_traffic/status/1439052358437253123
2. https://www.malware-traffic-analysis.net/2021/09/17/index.html
3. https://security-soup.net/squirrelwaffle-maldoc-analysis/
4. https://twitter.com/Max_Mal_/status/1439415164605018113
