Decrypting APT33's Dropshot Malware with Radare2 and Cutter – Part 2

By Itay Cohen

Published: 2018-06-18 · Archived: 2026-04-06 15:36:01 UTC

Prologue

Previously, in the first part of this article, we used Cutter, a GUI for radare2, to statically analyze APT33's Dropshot malware. We also used radare2's Python scripting capabilities in order to decrypt encrypted strings in Dropshot. If you didn't read the first part yet, I suggest you do it now.

Today's article will be shorter, now that we are familiar with cutter and r2pipe, we can quickly analyze another interesting component of Dropshot — an encrypted resource that includes Dropshot's actual payload. So without further ado, let's start.

Downloading and installing Cutter

Cutter is available for all platforms (Linux, OS X, Windows). You can download the latest release here. If you are using Linux, the fastest way to use Cutter is to use the AppImage file.

If you want to use the newest version available, with new features and bug fixes, you should build Cutter from source by yourself. It isn't a complicated task and it is the version I use.

First, you must clone the repository:

```
 git clone --recurse-submodules https://github.com/radareorg/cutter
cd cutter
```

Building on Linux:

```
 ./build.sh
```

Building on Windows:

```
 prepare_r2.bat
build.bat
```

If any of those do not work, check the more detailed instruction page here

Dropshot \ StoneDrill

As in the last part, we'll analyze Dropshot, which is also known by the name StoneDrill. It is a wiper malware associated with the APT33 group which targeted mostly organizations in Saudi Arabia. Dropshot is a sophisticated malware sample, that employed advanced anti-emulation techniques and has a lot of interesting functionalities. The malware is most likely related to the infamous Shamoon malware. Dropshot was analyzed thoroughly by Kaspersky and later on by FireEye. In this article, we'll focus on decrypting the encrypted resource of Dropshot which contains the actual payload of the malware.

The Dropshot sample can be downloaded from here (password: *infected*). I suggest you star (★) the repository to get updates on more radare2 tutorials 🙂

Please, be careful when using this sample. It is a real malware, and more than that, a wiper! Use with caution!

Since we'll analyze Dropshot statically, you can use a Linux machine, as I did.

Getting Started

Assuming you went through the first part of the article, you are already familiar with Cutter and r2pipe. Moreover, you should already have a basic clue of how Dropshot behaves. Open the Dropshot sample in Cutter, execute in Jupyter the r2pipe script we wrote and seek to the `main` function using the "Functions" widget or the upper search bar.

A function we analyzed in the previous article | Click to enlarge

main ()

The role of the `main()` function in a program shouldn't be new to you since it is one of the fundamental concepts of programming. Using the Graph mode, we'll go thorugh `main`'s flow in order to find our target – the resource decryption routine. We can see that a function at `0x403b30` is being called at the first block of `main`.

Double-clicking this line will take us to the graph of `fcn.00403b30`, a rather big function. Going through this function, we'll see some non-sense Windows API calls with invalid arguments. When describing Dropshot, I said that it uses anti-emulation heavily – this function, for example, performs anti-emulation.

Click to enlarge

Anti-Emulation

Anti-emulation techniques are used to fool the emulators of anti-malware products. The emulators are one of the most important components of many security products. Among others, they are used to analyze the behavior of malware, unpack samples and to analyze shellcode. It is doing this by emulating the program's workflow by mimicking the target architecture's instruction set, as well as the running environment, and dozens or even hundreds of popular API functions. All this is done in order to make a malware 'think' it has been executed in a real environment by a victim user.

The emulator engine is mimicking the API or the system calls that are offered by the actual operating systems. Usually, it will implement popular API functions from libraries such as *user32.dll*, *kernel32.dll*, and *ntdll.dll*. Most of the times this will be a dummy implementation where the fake functions won't really do anything except returning a successful return value.

By using different anti-emulation techniques, malware authors are trying to fool a generic or even a specific emulator. The most common technique, which is also implemented in Dropshot's `fcn.00403b30`, is the use of uncommon or undocumented API calls. This technique can be improved by using incorrect arguments (like *NULL*) to a certain API function which should cause an Access Violation exception in a real environment.

In our case, Dropshot is calling some esoteric functions as well as passing non-sense arguments to different API functions.

More information about emulation and anti-emulation mechanisms are available in the following, highly recommended, book:

Now that we know all this, we can rename this function from `fcn.00403b30` to a more meaningful name. I used "AntiEmulation" but you can choose whatever name you want, as long as it is meaningful to you. Clicking the call instruction and then pressing Shift+N will open the Rename dialog box. Right-clicking the row and choosing "Rename" will do the job as well.

 After `main` is calling to the AntiEmulation function, we are facing a branch. Here's the assembly, copied from Cutter's Disassembly widget:

```
 |           0x004041a6           call AntiEmulation
|           0x004041ab           mov eax, 1
|           0x004041b0           test eax, eax
|       ,=< 0x004041b2           je 0x40429d
|       |   0x004041b8           push 4
```

As you can see, the code would never branch to `0x40429d` since this `test eax, eax` followed by `je ...` is basically checking whether `eax` equals 0. One instruction before, the program moved the value 1 to `eax`, thus `0x40429d` would be The Road Not Taken.

We'll skip the next block which is responsible for creating temporary files and take a look at the block starting at `0x4041f9`. In this block, we'll see that Dropshot is creating a modeless Dialog box using `CreateDialogParamA` with the following parameters: `CreateDialogParamA(0, 0x410, 0, DialogFunc, 0);`.

The DialogPrc callback which is passed to `CreateDialogParamA` is recognized by radare2 and shown by Cutter as `fcn.DialogFunc`. This function contains the main logic of the dropper and this is the function that we'll focus on. Later in this block, `ShowWindow` is being called in order to "show" the window. Obviously, this is a dummy window which would never be shown since the malware author doesn't want any artifact to be shown to the victim. `ShowWindow` will trigger the execution of `fcn.DialogFunc`.

Double-clikcing on `fcn.DialogFunc` will take us to the function itself. We can see that it is performing several comparisons for the messages it receives and then is calling to a very interesting function `fcn.00403240`.

Handling the Resources

The first block of `fcn.00403240` is pretty straightforward. Dropshot is getting a handle to itself using `GetModuleHandleA`. Then, by using `FindResourceA`, it is locating a resource with a dummy type 0x67 (Decimal: 111) and a name 0x6e (Decimal: 110). Finally, it is loading a resource with this name using `LoadResource`.

Using Cutter, we can see the content of this resource. Simply go to the Resources widget and locate the resource with the name “110”.

 As you can see in the screenshot above, the size of the resource is 28 Bytes and its Lang is Farsi which might hint us about the threat actor behind Dropshot.

Double-clicking the resource will take us to the resource’s address. Let’s click on the Hexdump widget to see its data. In the hexdump we can see that this resource contains the following bytes: `01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00`. Those of you who are familiar with radare2 may use the Console widget in the bottom left to do this quickly with `px`:

 This resource will be used later but we won’t be getting into it since it is out of the scope of this post.

After loading the resource, in the next block we can see the start of a loop:

 This loop is checking if `local_2ch` equals to 0x270f (Decimal: 9999) and if yes it exits from the loop. Inside this loop, there will be another loop of 999 iterations. So basically, this is how this nested loop looks like:

```
for ( i = 0; i < 9999; ++i )
 {
   for ( j = 0; j < 999; ++j )
   {
       dummy_code;
   }
 }
```

This is just another anti-emulation\analysis technique which is basically doing nothing. This is another example of the heavy use of anti-emulation by Dropshot.

After this loop, the right branch is taken and this is an interesting one.

Click to enlarge

At first, `VirutalAlloc` in the size of 512 bytes is being called. Next, `fcn.00401560` is called with 3 arguments. Let’s enter this function and see what’s in there:

Click to enlarge

Hey! Look who's back! We can see the 2 functions we analyzed in the previous article: `decryption_function` and `load_ntdll_or_kernel32`. That's great! Also, you can notice the comment on `call decryption_function` which is telling us that the decrypted string is `GetModuleFileNameW`. This comment is the result of the r2pipe script we wrote.

In this function, `GetModuleFileNameW` will be decrypted, then Kernel32.dll will be loaded and `GetProcAddress` will be called to get the address of `GetModuleFileNameW` and then it will move it to `[0x41dc04]`. Later in this function, `[0x41dc04]` will be called.

Basically, this function is wrapper around `GetModuleFileNameW`, something which is common in Dropshot's code. Let's rename this function to `w_GetModuleFileNameW` where "w_" stands for "wrapper". Of course, you can choose whatever naming convention you prefer.

Right after the call to `w_GetModuleFileNameW`, Dropshot is using VirtualAlloc to allocate 20 (0x14) bytes, then there is a call to `fcn.00401a40` which is a function quite similar to `memset`, it is given with 3 arguments (*address*, *value* and *size*), just like `memset` and it is responsible to fill the range from `address` to `address+size` with the given `value`. Usually, along the program, this function is used to fill an allocated buffer with zeroes. This is quite strange to me, since VirtualAlloc is already "initializes the memory it allocates to zero". Let's name this function to `memset_` using Shift+N or via right click and move on.

Right after the program is zeroing-out the allocated memory, we see a call to another function – `fcn.00401c90`. We can see 3 arguments which are being passed to it, 0x14, 0x66, and 0x68. Since sometimes we prefer to see decimal numbers and not hex, let's use another useful trick of Cutter. Right-click on any of these hex numbers and choose "Set Immediate Base to…" and then select "Decimal".

 Now we can see that the values which are being passed to `fcn.00401c90` are 20, 102 and 104. Looks familiar? 20 was the size of the buffer that was just allocated. 102 and 104 remind us the Resource name and type that used before (110 and 111). Are we dealing with resources here? Let's see.

Moving to the Resources widget again, we can see that there's indeed a resource named "102" which "104" is its type. And yes, it is 19B long, close enough 😉

`fcn.00401c90` is one of the key functions involved in the dropper functionality of Dropshot. The thing is, that this function is rather big and quite complicated when you don't know how to look at it. We'll get back to it in one minute but before that, I want to show you an approach I use while reverse engineering some pieces of code and while facing a chain of calls to functions which are probably related to each other.

First, we saw that 20 bytes were allocated by `VirtualAlloc` and the pointer to the allocated memory was moved to `[local_ch]`. Right after that, `memset_` was called in order to zero-out 20 bytes at `[local_ch]`, i.e to zero-out the allocated buffer. Immediately after, `fcn.00401c90` was called and 3 arguments were passed to it – 104, 102 and our beloved 20. We know that 102 is a name of a resource and its size is almost 20. We don't know yet what this function is doing but we know that its return value(`eax`) is being passed along with 2 more arguments to another function, `fcn.00401a80`. The other arguments are, you guessed right, 20 and the allocated buffer. A quick look at `fcn.00401a80`, which is a really tiny function, will reveal us that this function is copying a buffer to the

allocated memory. This function is quite similar to `memcpy` so we'll rename it to `memcpy_`. So now we can do an educated guess and say that `fcn.00401c90` is reading a resource to a buffer and returns a pointer to it.

Using this approach, we can understand (or at least guess) a complicated function without even analyzing it. Just by looking at a programs chain of function calls, we can build the puzzle and save us important time.

That said, we'll still give this function a quick analysis because we want to be sure that we guessed right, and more importantly — because this is an interesting function.

Resource Parser

The next part is where things are getting more complicated. We'll start by going over `fcn.00401c90` pretty fast so try to follow. Also, you may want to make sure you fasten yourself since we are going on a rollercoaster ride through the PE structure.

Take a look at the first block of this function. You'll see one `call` and a lot of `mov`, `add` and calculation of offsets. This is how a typical PE parsing looks like.

```
 / (fcn) fcn.00401c90 468
|   fcn.00401c90 (int arg_8h, int arg_ch, int arg_10h);
|           0x00401c90           push ebp
|           0x00401c91           mov ebp, esp
|           0x00401c93           sub esp, 0x44
|           0x00401c96           mov dword [local_40h], 0
|           0x00401c9d           push 0
|           0x00401c9f           call GetModuleHandleW
|           0x00401ca5           mov dword [local_34h], eax
|           0x00401ca8           mov eax, dword [local_34h]
|           0x00401cab           mov dword [local_20h], eax
|           0x00401cae           mov ecx, dword [local_20h]
|           0x00401cb1           mov dword [local_24h], ecx
|           0x00401cb4           mov edx, dword [local_24h]
|           0x00401cb7           mov eax, dword [edx + 0x3c]
|           0x00401cba           add eax, dword [local_24h]
|           0x00401cbd           mov dword [local_38h], eax
|           0x00401cc0           mov ecx, dword [local_38h]
|           0x00401cc3           add ecx, 0x18
|           0x00401cc6           mov dword [local_3ch], ecx
|           0x00401cc9           mov edx, dword [local_3ch]
|           0x00401ccc           add edx, 0x60
|           0x00401ccf           mov dword [local_28h], edx
|           0x00401cd2           mov eax, 8
|           0x00401cd7           shl eax, 1
|           0x00401cd9           mov ecx, dword [local_28h]
|           0x00401cdc           mov edx, dword [ecx + eax]
|           0x00401cdf           mov dword [local_44h], edx
```

```
|           0x00401ce2          mov eax, 8
|           0x00401ce7          shl eax, 1
|           0x00401ce9          mov ecx, dword [local_28h]
|           0x00401cec          mov edx, dword [local_20h]
|           0x00401cef          add edx, dword [ecx + eax]
|           0x00401cf2          mov dword [local_10h], edx
|           0x00401cf5          mov eax, dword [local_10h]
|           0x00401cf8          mov dword [local_4h], eax
|           0x00401cfb          mov ecx, dword [local_4h]
|           0x00401cfe          movzx edx, word [ecx + 0xe]
|           0x00401d02          mov eax, dword [local_4h]
|           0x00401d05          movzx ecx, word [eax + 0xc]
|           0x00401d09          add edx, ecx
|           0x00401d0b          mov dword [local_ch], edx
|           0x00401d0e          mov dword [local_14h], 0
|       ,=< 0x00401d15          jmp 0x401d20
...
...
```

At first, a handle to the current process is received using `GetModuleHandleW`. Then, the handle (`eax`) is being moved to a variety of local variables. First, it is being moved to `[local_34h]` at `0x00401ca5`. Then you can see `eax` moved to `[local_20h]` which is later being moved to `[local_24h]` using `ecx`.

So basically we have a bunch of local variables that currently hold the handle `hmodule`. We can rename all three variables to `[hmodule_x]` so it'll be easier to keep track of all the reference to `hmodule`. To rename flags you can use the Console widget and just execute `afvn old_name new_name`. For example, I executed: `afvn local_34h hmodule_1; afvn local_20h hmodule_2; afvn local_24h hmodule_3`.

`GetModuleHandle` returns a handle to a mapped module, this basically means that our `hmodule`s point to our binary's base address. In line `0x00401cb4` we can see that `[hmodule_3]` is moved to `edx`, then the value at `[edx + 0x3c]` is being moved to `eax` and `[hmodule_3]` is added to it at `0x00401cba`. Finally, `eax` is moved to `[local_38h]`. To put it simply, we can use the following pseudo-code:

```
[local_38h] = (BYTE*)hmodule + *(hmodule + 0x3c)
```

So what's in this address? Use your favorite binary structure viewer to find out. In this example, I'll use PEview but you can use any other program you prefer – including the binary structure parsing feature of radare2, if you're already a radare2 pro (see `pf?`).

Open Dropshot in PEview and inspect the DOS Header:

 As you can see, in offset `0x3c` there is a pointer (0x108) to the offset to the new EXE Header which is basically the IMAGE_NT_HEADER. Awesome! So `[local_38h]` holds the address of the NT Header. Let's rename it to `NT_HEADER` and move on.

At address `0x00401cc0` we can see that `NT_HEADER` is moved to `ecx` and then the program is adding 0x18 to `ecx`. Last, the value in `ecx` is moved to `[local_3ch]`. Just as before, let's open again our PE parser and check what is in `NT_HEADER + 0x18`. Adding 0x18 to 0x108 will give us 0x120. Let's see what is in this offset:

Click to enlarge

Nice! 0x120 is the offset of the IMAGE_OPTIONAL_HEADER as can be seen in the image above. Let's rename `local_3ch` to `OPTIONAL_HEADER`. To cut a *long story short*, Dropshot is then parsing the IMAGE_DATA_DIRECTORY structure (OPTIONAL_HEADER + 0x60), the RESOURCE_TABLE, and last it iterates through the different resources and compares the resource type and the resource name to the function's arguments. Finally, it uses `memcpy_` to copy the content of the required resource to a variable and returns this variable.

Now we can rename the function to `get_resource` and the arguments to the corresponding meaning of them by executing `afvn arg_8h arg_rsrc_type; afvn arg_ch arg_rsrc_name; afvn arg_10h arg_dwsize`.

Now that we are sure about what this function does, we can see where else it is referenced. Right-click on the function and choosing "Show X-Refs" (or simply pressing 'x') will take us to the X-Refs window. We can see that `get_resource` is being called from two locations. One (`0x0040336d`) is already familiar to us, it is called to get resource "102". The second call (at `0x00403439`), a few instructions later, is new to us — it is called to get the content of another resource, named "101" (0x65).

Remember the screenshot of the Resources widget from before? We can see there the resource named "101". What makes "101" so interesting is that it is **much** bigger than the other — its size is 69.6 KB! This is Dropshot's payload. By going to the Resources widget and double-clicking "101" will take us to the resource's offset in the binary. In the Hexdump widget, we can see that the content of this resource makes no sense and has a really high entropy (7.8 out of the maximum 8):

Click to enlarge

This data is compressed/encrypted somehow so we need to decrypt it. Let's continue our analysis to find out how.

How To Decrypt The Resource

In order to decrypt the resource, we should follow the program's flow to see how and where the payload is being used. Right after `get_resource` is being called with "101" and "103", the resource is copied to `[local_20h]` using `memcpy_` (at `0x00403446`). Let's call it `compressed_payload`. The compressed buffer is then passed to `fcn.00401e70` which is a function that performs dummy math calculations on the resource's data. Probably another Anti-Emulation technique or simply a way to waste our time. I'll rename it to `dummy_math`. Next, `compressed_payload` is being passed to `fcn.00401ef0` along with another buffer `[local_54h]`.

The analysis of this function is out of the scope of this article but this function is responsible to decompress a buffer using zlib and put the decompressed buffer in `[local_54h]`. You can see, for example, that `fcn.00401ef0` is calling to `fcn.004072f0` which contains strings like "unknown compression method" and "invalid window size" which can be found in the file inflate.c in the zlib repository. I renamed `fcn.00401ef0` to `zlib_decompress` and `local_54h` to `decompressed_payload`.

I'll tell you now that simply a decompression of the buffer isn't enough since there's still another simple decryption to do. Straight after the decompressing, we can see more of the Anti-Emulation which we are already familiar with. Finally, our decompressed buffer is being passed to `fcn.00402620`. This function is responsible for the last decryption of the resource and then it performs a notorious technique known as "Process Hollowing" or "RunPE" in order to execute the decrypted payload.

So how `fcn.00402620` decrypts the decompressed payload? Simply, it uses `ror 3` to rotate-right each byte in the decompressed buffer. 3 stands for the number of bits to rotate.

The rest of this function is interesting as well but it has nothing to do with decrypting the resource so I'll leave it to you.

To sum things up, and before we adding the logic for the resource decryption inside the script we wrote in the previous article – let's sketch how the decryption function should look like. It should be something like this:

```
 rsrc_101 = get_resource("101")

decompressed_payload = decompress(rsrc_101)

decrypted_payload = []

for b in decompressed_payload:
   decrypted_payload.append(ror3(b))
```

Scripting time! Decrypting the resource

Scripting radare2 is really easy thanks to r2pipe. It is the best programming interface for radare2.

> The r2pipe APIs are based on a single r2 primitive found behind `r_core_cmd_str()` which is a function that accepts a string parameter describing the r2 command to run and returns a string with the result.

r2pipe supports many programming languages including Python, NodeJS, Rust, C, and others.

Luckily, Cutter is coming with the python bindings of `r2pipe` integrated into its Jupyter component. We'll write an r2pipe script that will do the following:

- Save the compressed resource into a variable

- Decompress the resource using zlib
- Perform ror3 on each byte in the decompressed payload
- Save the decrypted resource to a file

Just as in the previous part, let's go to the Jupyter widget and open the script we wrote when decrypted the strings (part1).

The first thing to do is to read the content of the encrypted and compressed resource to a file:

```
 rsrcs = cutter.cmdj('iRj')
rsrc_101 = {}

# Locate resource 101 and dump it to an array
for rsrc in rsrcs:
    if rsrc['name'] == 101:
        rsrc_101 = cutter.cmdj("pxj %d @ %d" %
                               (rsrc['size'], rsrc['vaddr']))
```

> `iR` was used to get the list of resources and their offsets in the file. Next, we iterate through the different resources untill we find a resource named "101". Last, using `px` we are reading the resource's bytes into a varibale. We appended `j` to the commands in order to get their output as JSON.

Next, we want to decompress the buffer using zlib. Python is coming with "zlib" library by default which is great news for us. Add `import zlib` to the top of the script and use this code to decompress the buffer:

```
 # Decompress the zlibbed array
decompressed_data = zlib.decompress(bytes(rsrc_101))
```

Now that our buffer is decompressed in `decompressed_data`, all we left to do is to perform right rotation on the data and save it to a file.

Define the following `ror` lambda:

```
 def ror(val, r_bits, max_bits): return \
    ((val & (2**max_bits-1)) >> r_bits % max_bits) | \
    (val << (max_bits-(r_bits % max_bits)) & (2**max_bits-1))
```

And use it in your code like this:

```
 decrypted_payload = []

# Decrypt the payload
for b in decompressed_data:
    decrypted_payload.append((ror(b, 3, 8)))
```

Last, save it to a file:

```
 # Write the payload (a PE binary) to a file
open(r'./decrypted_rsrc.bin', 'wb').write(bytearray(decrypted_payload))
```

Now let's combine the script from the previous article to the one we created now and test it in Jupyter. The combined script should first decode the encrypted scripts, and then it should decrypt the resource and save it to the disk.

The final script can be found here.

Copy it and paste it into your Jupyter notebook. You can also execute your version of the code to see if you got it right by yourself.

Seems like our script was executed successfully and "Saved the PE to ./decrypted_rsrc.bin". Great!

The last thing we want to do is to open `decrypted_rsrc.bin` in a new instance of Cutter in order to verify that this is indeed a PE file and that we didn't corrupt the file in some way.

Click to enlarge

Awesome! Cutter recognized the file as PE and seems like the code is correctly interpreted. This binary we just decrypted and saved is the Wiper module of Dropshot – a quite interesting piece of malware on its own. This module, just as its dropper, is using heavy anti-emulation and similar technique to decrypt its strings. You can give it a try and analyze it on your own using Cutter, radare2, and r2pipe. Good Luck!

Epilogue

Here comes to an end the second and the last part of this series about decrypting Dropshot with Cutter and r2pipe. We got familiar with Cutter, radare2 GUI, and wrote a decryption script in r2pipe's Python binding. We also analyzed some components of APT33's Dropshot, an advanced malware.

As always, please post comments to this post or message me privately if something is wrong, not accurate, needs further explanation or you simply don't get it. Don't hesitate to share your thoughts with me.

Subscribe on the left if you want to get the next articles straight in your inbox.
