Reverse-engineering DUBNIUM

TN blogs.technet.microsoft.com/mmpc/2016/06/09/reverse-engineering-dubnium-2/

June 9, 2016

DUBNIUM (which shares indicators with what Kaspersky researchers have called DarkHotel) is one of the activity groups that has been very active in recent years, and has many distinctive features.

We located multiple variants of multiple-stage droppers and payloads in the last few months, and although they are not really packed or obfuscated in a conventional way, they use their own methods and tactics of obfuscation and distraction.

In this blog, we will focus on analysis of the first-stage payload of the malware.

As the code is very complicated and twisted in many ways, it is a complex task to reverse-engineer the malware. The complexity of the malware includes linking with unrelated code statically (so that their logic can hide in a big, benign code dump) and excessive use of an in-house encoding scheme. Their bootstrap logic is also hidden in plain sight, such that it might be easy to miss.

Every sub-routine from the malicious code has a “memory cleaner routine” when the logic ends. The memory snapshot of the process will not disclose many more details than the static binary itself.

The malware is also very sneaky and sensitive to dynamic analysis. When it detects the existence of analysis toolsets, the executable file bails out from further execution. Even binary instrumentation tools like PIN or DynamoRio prevent the malware from running. This effectively defeats many automation systems that rely on at least one of the toolsets they check to avoid. Avoiding these toolsets during analysis makes the overall investigation even more complex.

With this blog series, we want to discuss some of the simple techniques and tactics we’ve used to break down the features of DUBNIUM.

We acquired multiple versions of DUBNIUM droppers through our daily operations. They are evolving slowly, but basically their features have not changed over the last few months.

In this blog, we’ll be using sample SHA1: dc3ab3f6af87405d889b6af2557c835d7b7ed588 in our examples and analysis.

Hiding in plain sight

The malware used in a DUBNIUM attack is committed to disguising itself as Secure Shell (SSH) tool. In this instance, it is attempting to look like a certificate generation tool. The file descriptions and other properties of the malware look convincingly legitimate at first glance.

When it is run, the program actually dumps out dummy certificate files into the file system and, again, this can be very convincing to an analyst who is initially researching the file.

The binary is indeed statically linked with OpenSSL library, such that it really does look like an SSH tool. The problem with reverse engineering this sample starts from the fact that it has more than 2,000 functions and most of

them are statically linked to OpenSSL code without symbols.

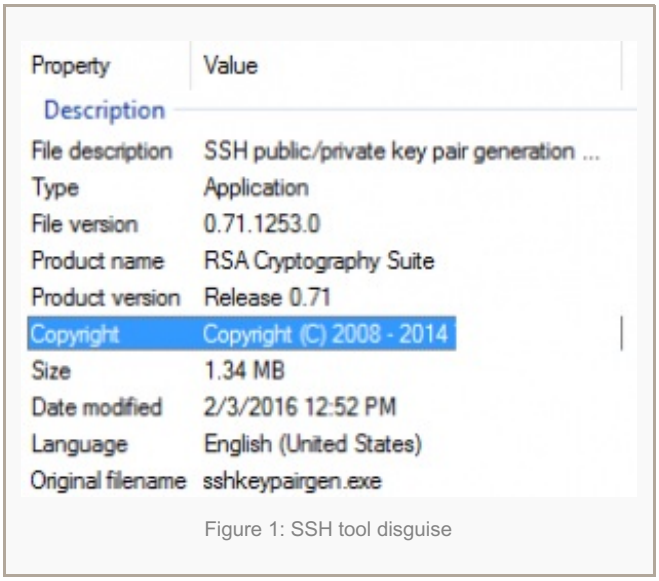

Figure 1: SSH tool disguise

The following is an example of one of these functions – note it even has string references to the source code file name.

It can be extremely time-consuming just going through the dump of functions that have no meaning at all in the code – and this is only one of the more simplistic tactics this malware is using.

We can solve this problem using binary similarity calculation. This technique has been around for years for various purposes, and it can be used to detect code that steals copyrighted code from other software.

The technique can be used to find patched code snippets in the software and to find code that was vulnerable for attack. In this instance, we can use the same technique to clean up unnecessary code snippets from our advanced persistent threat (APT) analysis and make a reverse engineer's life easier.

Figure 2 Create dummy certificate files

Many different algorithms exist for binary similarity calculation, but we are going to use one of the simplest approach here. The algorithm will collect the op-code strings of each instruction in the function first (Figure 5). It will then concatenate the whole string and will use a hash algorithm to get the hash out of it. We used the SHA1 hash in this case.

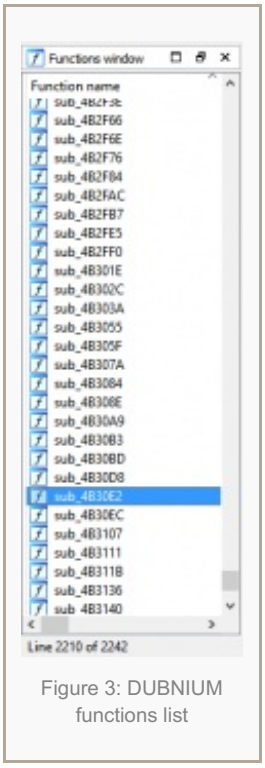

Figure 3: DUBNIUM functions list

Figure 6 shows the Python-style pseudo-code that calculates the hash for a function. Sometimes, the immediate constant operand is a valuable piece of information that can be used to distinguish similar but different functions and it also includes the value in the hash string. It is using our own utility function *RetrieveFunctionInstructions* which returns a list of op-code and operand values from a designated function.

```
01 def CalculateFunctionHash(self,func_ea):
02     hash_string=''
03     for (op, operand) in
self.RetrieveFunctionInstructions(func_ea):
04            hash_string+=op
05            if len(drefs)==0:
06                  for operand in operands:
07                        if operand.Type==idaapi.o_imm:
08                              hash _string+=('%x' % operand.Value)
09
10     m=hashlib.sha1()
11     m.update(op_string)
12     return m.hexdigest()
```

Figure 6: Pseudo-code for CalculateFunctionHash

With these hash values calculated for the DUBNIUM binary, we can compare these values with the hash values from the original OpenSSL library. We identified from the compiler-generated meta-data that the version the sample is linked to is openssl-1.0.1l-i386-win. After gathering same hash from the OpenSSL library, we could import symbols for the matched functions. In this way, removed most of the functions from our analysis scope.

(This blog is continued on the next page)

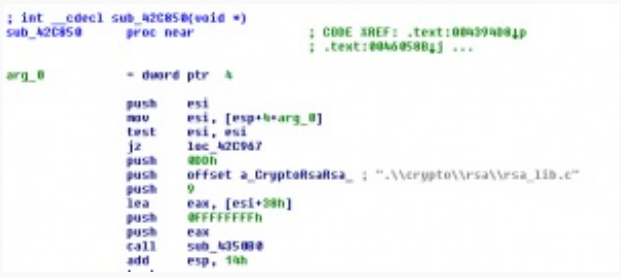

Figure 4: Code snippet that is linked from OpenSSL library

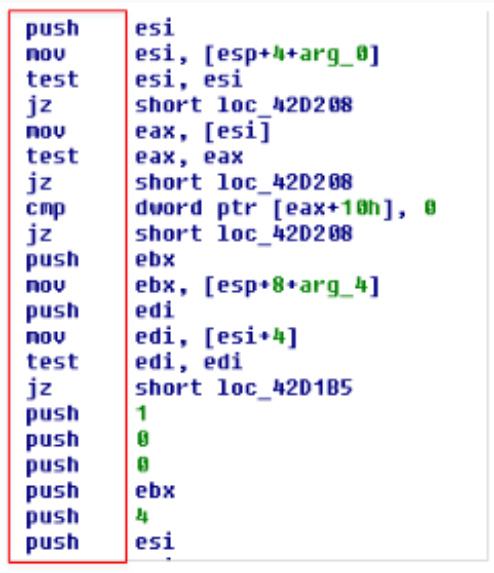

Figure 5: Op code in the instructions

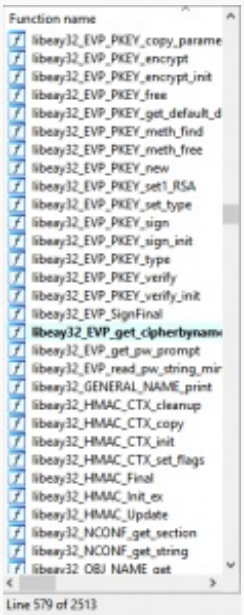

Figure 7: OpenSSL functions

Reverse-engineering DUBNIUM

TN **blogs.technet.microsoft.com**/mmpc/2016/06/09/reverse-engineering-dubnium-2/2/

June 9, 2016

Persistently encoded strings

The other issue when reverse-engineering DUBNIUM binaries is that it encodes every single string that is used in the code (Figure 8). There is no clue on the functionality of purpose of the binary by just looking at the string's table. We had to decode each of these strings to understand what the binary is intended to do. This may not be technically difficult, but it does require a lot of time and effort.

Figure 8: Encoded strings

Figure 9 shows how these encoded strings are used. For example, address 0x142C11C has an instruction that loads an encoded string which is decoded as "*hook_disable_retaddr_check*". The encoded string is passed in *ecx* register to the decoder function (*decode_string*). Note that the symbol names for the functions were made by us during the analysis.

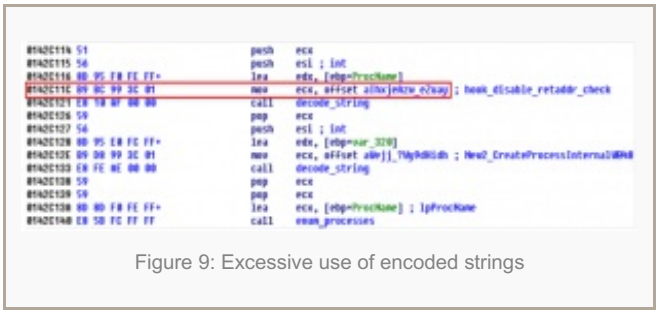

Figure 9: Excessive use of encoded strings

Because the *decode_string* function is excessively used and encoded gibberish strings are always passed to it, we can be confident that the function is truly a string decoder. The *decode_string* function looks like Figure 10. There are some approaches that can be taken for decoding these files: you could port the code to C or Python and run them through encoded strings, or you could reuse the code snippet itself and pass the encoded string to the decoder function. We took the second option and reused the existing code for decoding strings, for faster analysis of the sample.

For example, we have an encoded string at address 0x013C992C.

The *decode_string* function is located at 0x01437036 in our case. The *ecx* register will point to the encoded string and *edx* is the destination buffer address for the decoded string. We just came up with the right place on the stack with enough buffer, which in this case is esp+0x348.

`lea edx,[esp+0x348]` – pointer to stack buffer address
`mov ecx, 0x013C992C` – pointer to encoded string
`call 0x01437036` – call to decode_string

As the instructions above will decode the encoded string for us, we can use Windbg to run our code. First we prepared a virtual machine environment, because we can possibly run malicious routines from the sample. As there are some possibilities that the *decode_string* function is dependent on some initialization routines called at startup,

we put our first breakpoint to the location where the first instance of *decode_string* is called. In this way, we can guarantee that our own *decode_string* call will be surely called with proper setup. That address we came up with is 0x0142BFEE (Figure 12).

Here's where our breakpoint is hit at this address.

Now we need to write the memory over with our own code.

The memory location where *eip* is pointing looks like the following.

Basically, we put the breakpoint on the entry of the *decode_string* and exit of the function. With the entry of the function, we save the *edx* register value to a temporary register and use it to dump out the decoded string memory location at the exit point.

Figure 10: decode_string routine

Now we have a handy way to decrypt the strings we have. Just after a few IDAPython scripts that retrieve all possible encoded strings and automatically generates the assembly code that calls *decode_string*, we can come up with a new IDA listing that shows the decoded string as the comment.

Figure 11: Encoded string

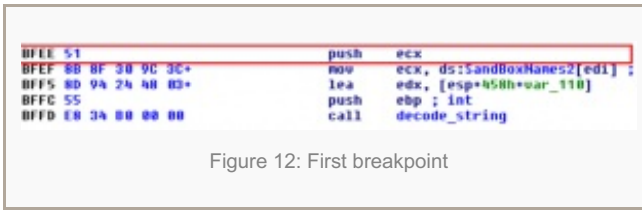

Figure 12: First breakpoint

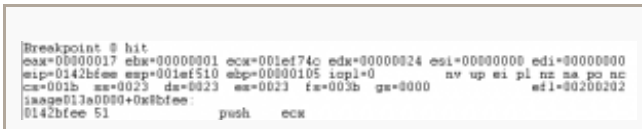

Figure 13: Breakpoint on 0142bfee hit

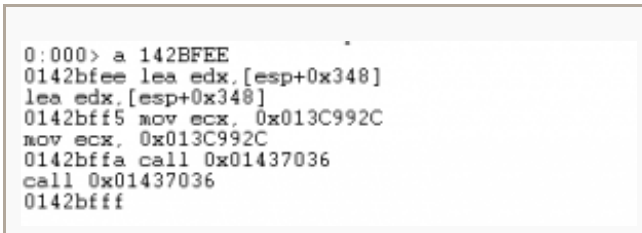

Figure 14: Use 'a' command to write instructions over the current eip location

Figure 15: New disassembly code

(This blog is continued on the next page)

Pages: Page 1, Page 2, Page 3

Figure 16: Breakpoints and dump of decoded string

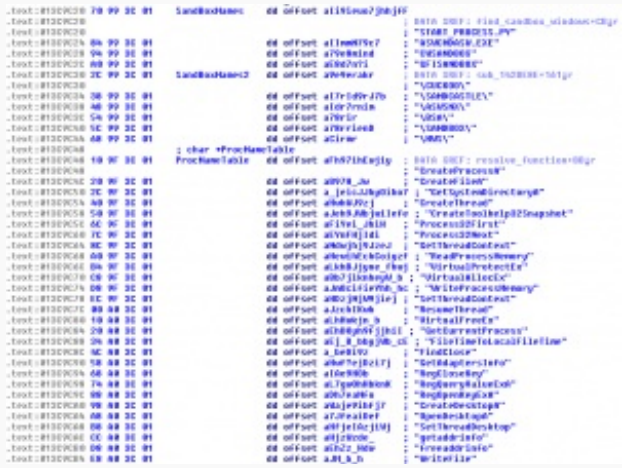

Figure 17: Decoded strings

Reverse-engineering DUBNIUM

TN blogs.technet.microsoft.com/mmpc/2016/06/09/reverse-engineering-dubnium-2/3/

June 9, 2016

Memory cleanup

Even after encoding every single string related to malicious code, the DUBNIUM malware goes one more step to hide its internal operations. When it calls *decode_string* to decode an encoded string, it will use the local stack variable to save the decoded string. Whenever the function returns, it calls *fill_memory_with_random_bytes* function for every local variable it used, so that the stack is cleared from decoded strings.

The memory cleaner function generates random bytes and fills the memory area. This can be very simple, and but still can be very annoying to malware analysts because, even with memory snapshot, we can't acquire any meaningful strings out of it. It's not easy to get a clue of what this binary is doing internally by just skimming through a memory snapshot.

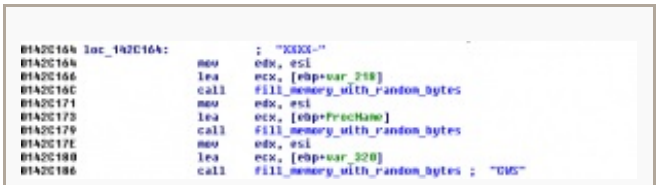

Figure 18: Calling memory cleaner function

Various environment check

Once we have decoded the string, further reverse engineering becomes trivial. It is no more complicated than any other malware we observe on a daily basis. The DUBNIUM binary checks for the running environment very extensively. It has a very long list of security products and other software it detects, and it appears that it detects all major antimalware and antivirus vendor process names.

Figure 18b: Calling memory cleaner function

One other very interesting fact is the presence of process names that are associated with software mainly used in China. For example, *QQPCRTP.exe* and *QQPCTray.exe* are from a messaging software by a company based in China. Also, *ZhuDongFangYu.exe*, *360tray.exe* and *360sd.exe* process names are used by security products that originate from China. From the software it detects, we get the impression that the malware is focusing on a specific geolocation as its target.

Aside from security programs and other programs used daily that can be used to profile its targets, the DUBNIUM malware also checks for various program analysis tools including Pin and DynamoRIO. It also checks for a virtual machine environment. If some of these are detected, it quits its execution. Overall, the malware is very cautious and deterministic in running its main code.

Figure 19: Extensive list of process names

The following figure shows the code that checks for the

existence of the Fiddler web debugger, which is very popular among malware analysts. As we wanted to use Fiddler to get a better understanding on the network activity of the malware, we manually patched the routine so it would not detect the Fiddler mutex.

Figure 20: Fiddler mutex check

Second payload download

The DUBNIUM samples are distributed in various ways, one instance was using a zero-day exploit that targets Adobe Flash, in December 2015. We also observed the malware is distributed through spear-phishing campaigns that involve social engineering with LNK files.

After downloading this payload, it would check the running environment and will only proceed with the next stage when it determines the target is a valid one for its purpose.

If software and environment check passes, the first stage payload will try to download the second stage payload from the command and control (C&C) server. It will pass information such as the IP, MAC address, hostname and Windows language ID to the server, and the server will return the encoded second stage payload.

Figure 21: 2nd payload download traffic

The way the first stage payload downloads the second payload is both interesting and unique. It doesn't access the Internet directly from the code, but it uses the system-installed *mshta.exe* binary. *Mshta.exe* is often used by malware to run VBscript for malicious purposes, but using it for downloading a general purpose payload is not so common. This is because *mshta.exe* doesn't support downloading URL contents directly to an arbitrary location.

Figure 22: Encoded strings of the client information

DUBNIUM spawns the *mshta.exe* process with the URL to download and waits for some time, after that it opens the *mshta.exe* process and goes through open file handles to find a handle for the temporary file that is associated with the downloaded contents.

This is a very inconvenient way to download a payload from the Internet, but it is useful for hiding the originating process for network activities. Sometimes network security programs check for the process name and their digital signature to check if they have the right to access outside the network. In that case, this feature will be very handy for the malware.

Figure 23: mshta.exe execution code

As you can see from the figures below, it uses process-related documented and undocumented APIs to retrieve file handles from the *mshta.exe* process, resolves their names and uses filename heuristics to check if it is a response file or not.

The cache filename will be retrieved and opened to retrieve the payload from the C&C server.

Figure 24: API calls to retrieve handle file name in mshta.exe process

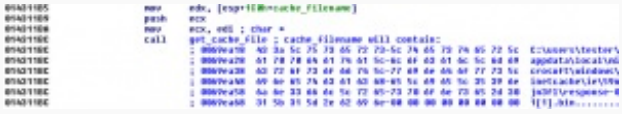

Figure 25: Cache filename

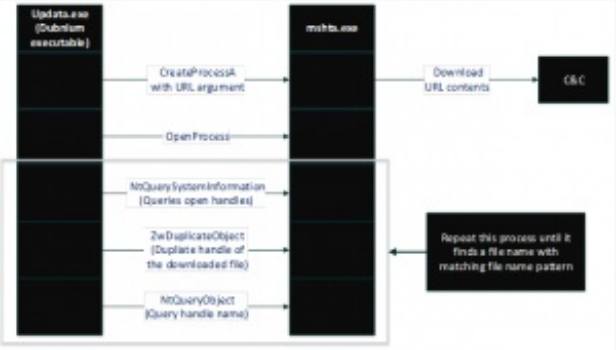

Figure 26: Using mshta.exe to download additional payload

Conclusion

Overall, the functionality of the DUBNIUM first stage payload is not so advanced in its functionality. It is a very simple downloader for the second stage payload.

However, the way it operates is very strategic:

- It hides in plain sight.
- It is very careful in initiating the next stage of the attack.
- It checks many different security products and user-installed programs that are bound to specific geolocations and cultures.
- It encodes every string that can be useful for quick analysis.
- It encodes outbound web traffic.
- It doesn't use high class encryption – but it does use an excessive amount of in-house string scrambling algorithms.
- It checks for many popular virtual environments and automatic analysis systems that are used for malware analysis, including VMware, Virtualbox and Cuckoo Sandbox
- It checks for popular dynamic analysis tools like PIN tool, DynamoRIO and other emulators.

In conclusion, this is the first stage payload with more of reconnaissance purpose and it will trigger next stage attack only when it decides the environment is safe enough for attack.

Appendix – Indicators of compromise

We discovered the following SHA1s in relation to DUBNIUM:

- 35847c56e3068a98cff85088005ba1a611b6261f
- 09b022ef88b825041b67da9c9a2588e962817f6d
- 7f9ecfc95462b5e01e233b64dcedbcf944e97fca
- cad21e4ae48f2f1ba91faa9f875816f83737bcaf
- ebccb1e12c88d838db15957366cee93c079b5a8e

- aee8d6f39e4286506cee0c849ede01d6f42110cc
- b42ca359fe942456de14283fd2e199113c8789e6
- 0ac65c60ad6f23b2b2f208e5ab8be0372371e4b3
- 1949a9753df57eec586aeb6b4763f92c0ca6a895
- 259f0d98e96602223d7694852137d6312af78967
- 4627cff4cd90dc47df5c4d53480101bdc1d46720
- 561db51eba971ab4afe0a811361e7a678b8f8129
- 6e74da35695e7838456f3f719d6eb283d4198735
- 8ff7f64356f7577623bf424f601c7fa0f720e5fb
- a3bcaecf62d9bc92e48b703750b78816bc38dbe8
- c9cd559ed73a0b066b480902434361O3eb52cc45
- dc3ab3f6af87405d889b6af2557c835d7b7ed588
- df793d097017b90bc9d7da9a85f929422004f6b6
- 8ff7f64356f7577623bf424f601c7fa0f720e5fb
- 6ccba071425ba9ed69d5a79bb53ad27541577cb9

-Jeong Wook Oh

MMPC