

Analyzing Dark Crystal RAT, a C# Backdoor

fireeye.com/blog/threat-research/2020/05/analyzing-dark-crystal-rat-backdoor.html



Threat Research

Jacob Thompson

May 12, 2020

19 mins read

Malware

Threat Research

The [FireEye Mandiant Threat Intelligence Team](#) helps protect our customers by tracking cyber attackers and the malware they use. The FLARE Team helps augment our threat intelligence by reverse engineering malware samples. Recently, FLARE worked on a new C# variant of Dark Crystal RAT (DCRat) that the threat intel team passed to us. We reviewed open source intelligence and prior work, performed sandbox testing, and reverse engineered the Dark Crystal RAT to review its capabilities and communication protocol. Through publishing this blog post we aim to help defenders look for indicators of compromise and other telltale signs of Dark Crystal RAT, and to assist fellow malware researchers new to .NET malware, or who encounter future variants of this sample.

Discovering Dark Crystal RAT

The threat intel team provided FLARE with an EXE sample, believed to contain Dark Crystal RAT, and having the MD5 hash b478d340a787b85e086cc951d0696cb1. Using sandbox testing, we found that this sample produced two executables, and in turn, one of those two executables produced three more. Figure 1 shows the relationships between the malicious executables discovered via sandbox testing.



Figure 1: The first

sample we began analyzing ultimately produced five executables

Armed with the sandbox results, our next step was to perform a triage analysis on each executable. We found that the original sample and mnb.exe were droppers, that dal.exe was a clean-up utility to delete the dropped files, and that daaca.exe and fsdffc.exe were variants of Plurox, a family with existing reporting. Then we moved to analyzing the final dropped sample, which was dfsds.exe. We found brief public reporting by [@James_inthe_box](#) on the same sample, [identifying it as DCRat and as a RAT and credential stealer](#). We also found a [public sandbox run](#) that included the same sample. Other public reporting [described DCRat, but actually analyzed the daaca.exe Plurox component](#) bundled along with DCRat in the initial sample.

Satisfied that dfsds.exe was a RAT lacking detailed public reporting, we decided to perform a deeper analysis.

Analyzing Dark Crystal RAT

Initial Analysis

Shifting aside from our sandbox for a moment, we performed static analysis on `dfsds.exe`. We chose to begin static analysis using CFF Explorer, a good tool for opening a PE file and breaking down its sections into a form that is easy to view. Having viewed `dfsds.exe` in CFF Explorer, as shown in Figure 2, the utility showed us that it is a .NET executable. This meant we could take a much different path to analyzing it than we would on a native C or C++ sample. Techniques we might have otherwise used to start narrowing down a native sample's functionality, such as looking at what DLLs it imports and what functions from those DLLs that it uses, yielded no useful results for this .NET sample. As shown in Figure 3, `dfsds.exe` imports only the function `_CorExeMain` from `mscorlib.dll`. We could have opened `dfsds.exe` in IDA Pro, but IDA Pro is usually not the most effective way of analyzing .NET samples; in fact, the free version of IDA Pro cannot handle .NET Common Language Infrastructure (CLI) intermediate code.



Figure 2: CFF

Explorer shows that `dfsds.exe` is a .NET executable

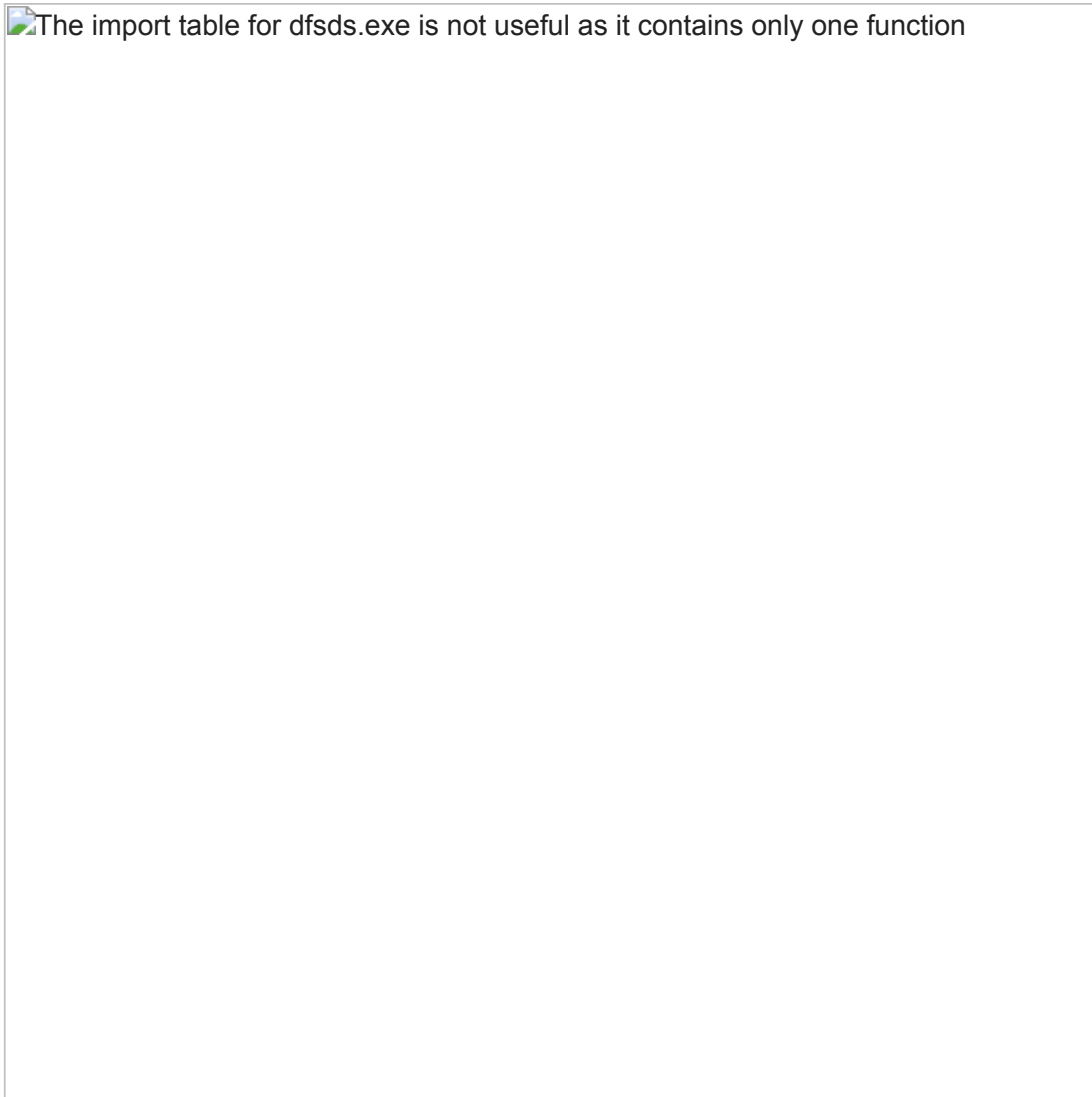


Figure 3: The

import table for dfsds.exe is not useful as it contains only one function

Instead of using a disassembler like IDA Pro on dfsds.exe, we used a .NET decompiler. Luckily for the reverse engineer, decompilers operate at a higher level and often produce a close approximation of the original C# code. dnSpy is a great .NET decompiler. dnSpy's interface displays a hierarchy of the sample's namespaces and classes in the Assembly Explorer and shows code for the selected class on the right. Upon opening dfsds.exe, dnSpy told us that the sample's original name at link time was DCRatBuild.exe, and that its entry point is at <PrivateImplementationDetails>{63E52738-38EE-4EC2-999E-1DC99F74E08C}.Main, shown in Figure 4. When we browsed to the Main method using the Assembly Explorer, we found C#-like code representing that method in Figure 5. Wherever dnSpy displays a call to another method in the code, it is possible to click on the target method name to go to it and view its code. By right-clicking on an identifier in the code, and clicking Analyze in the context menu, we caused dnSpy to look for all occurrences where the identifier is used, similar to using cross-references in IDA Pro.

 dnSpy can help us locate the sample's entry point

Figure 4: dnSpy

can help us locate the sample's entry point



Figure 5: dnSpy

decompiles the Main method into C#-like code

We went to the SchemaServerManager.Main method that is called from the entry point method, and observed that it makes many calls to ExporterServerManager.InstantiateIndexer with different integer arguments, as shown in Figure 6. We browsed to the ExporterServerManager.InstantiateIndexer method, and found that it is structured as a giant switch statement with many goto statements and labels; Figure 7 shows an excerpt. This does not look like typical dnSpy output, as dnSpy often reconstructs a close approximation of the original C# code, albeit with the loss of comments and local variable names. This code structure, combined with the fact that the code refers to the CipherMode.CBC constant, led us to believe that ExporterServerManager.InstantiateIndexer may be a decryption or deobfuscation routine. Therefore, dfsds.exe is likely obfuscated. Luckily, .NET developers often use obfuscation tools that are somewhat reversible through automated means.


 SchemaServerManager.Main makes many calls to
ExporterServerManager.InstantiateIndexer

Figure 6:

SchemaServerManager.Main makes many calls to ExporterServerManager.InstantiateIndexer



Figure 7:

ExporterServerManager.InstantiateIndexer looks like it may be a deobfuscation routine
Deobfuscation

De4dot is a .NET deobfuscator that knows how to undo many types of obfuscations. Running `de4dot -d` (for detect) on `dfsds.exe` (Figure 8) informed us that .NET Reactor was used to obfuscate it.

```
> de4dot -d dfsds.exe
```

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com  
Latest version and source code: https://github.com/0xd4d/de4dot
```

```
Detected .NET Reactor (C:\...\dfsds.exe)
```

Figure 8: `dfsds.exe` is obfuscated with .NET Reactor

After confirming that `de4dot` can deobfuscate `dfsds.exe`, we ran it again to deobfuscate the sample into the file `dfsds_deob.exe` (Figure 9).


```
> de4dot -f dfsds.exe -o dfsds_deob.exe
```

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com  
Latest version and source code: https://github.com/0xd4d/de4dot
```

```
Detected .NET Reactor (C:\Users\user\Desktop\intelfirst\dfsds.exe)  
Cleaning C:\Users\user\Desktop\intelfirst\dfsds.exe  
Renaming all obfuscated symbols  
Saving C:\Users\user\Desktop\intelfirst\dfsds_deob.exe
```

Figure 9: de4dot successfully deobfuscates dfsds.exe

After deobfuscating dfsds.exe, we ran dnSpy again on the resulting dfsds_deob.exe. When we decompiled SchemaServerManager.Main again, the results were much different, as shown in Figure 10. Contrasting the new output with the obfuscated version shown previously in Figure 6, we found the deobfuscated code much more readable. In the deobfuscated version, all the calls to ExporterServerManager.InstantiateIndexer were removed; as suspected, it was apparently a string decoding routine. In contrast, the class names shown in the Assembly Explorer did not change; the obfuscator must have irrecoverably replaced the original class names with meaningless ones obtained from a standard list. Next, we noted that ten lines in Figure 10 hold base64-encoded data. Once the sample was successfully deobfuscated, it was time to move on to extracting its configuration and to follow the sample's code path to its persistence capabilities and initial beacon.

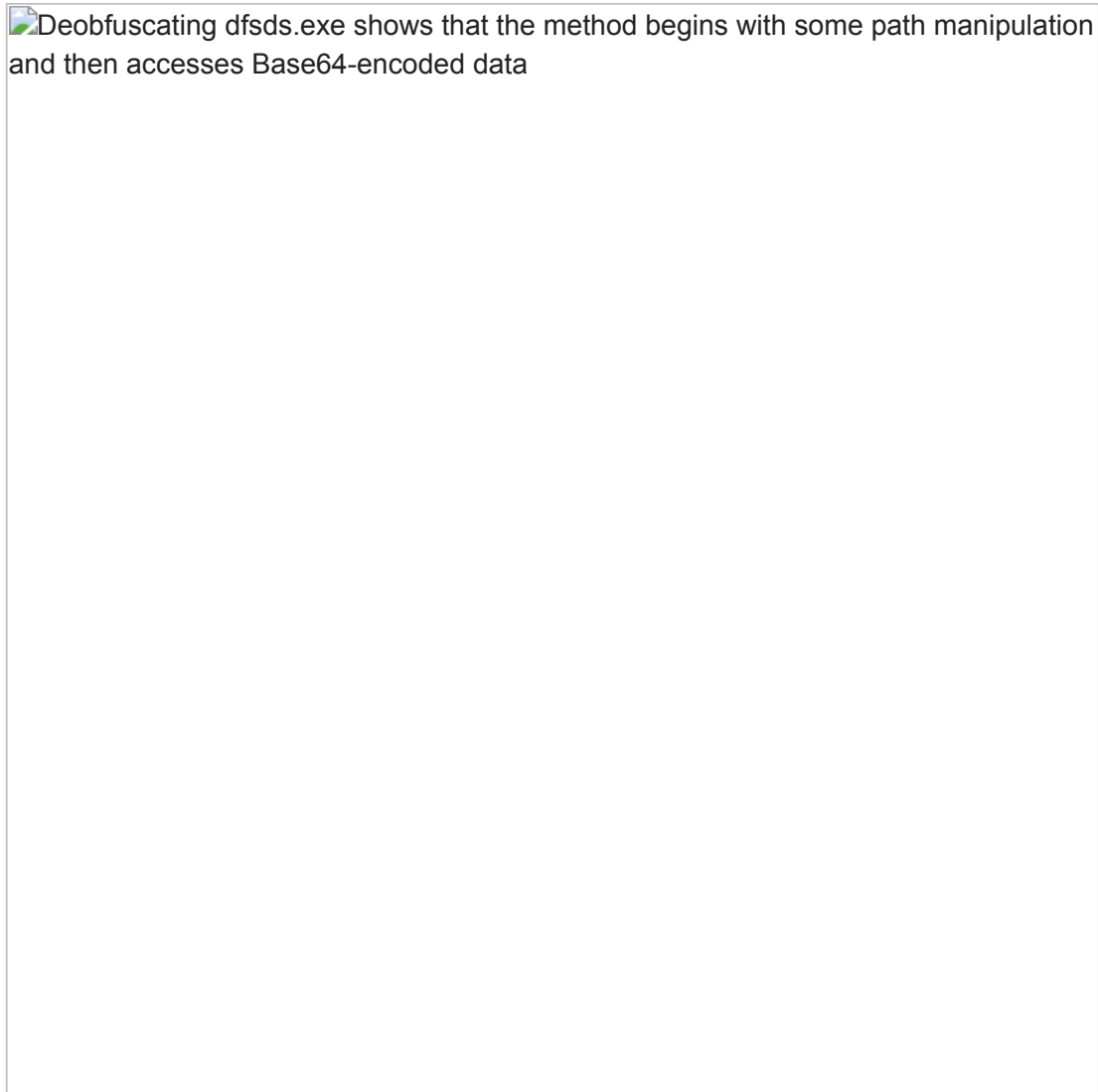


Figure 10:

Deobfuscating dfsds.exe shows that the method begins with some path manipulation and then accesses Base64-encoded data

Configuration, Persistence and Initial Beacon

Recall that in Figure 10 we found that the method SchemaServerManager.Main has a local variable containing Base64-encoded data; decoding that data revealed what it contains. Figure 11 shows the decoded configuration (with C2 endpoint URLs de-fanged):

```
> echo
TUhvc3Q6aHR0cDovL2RvbWFSby5vbmhpbmUva3NlemJseGx2b3Uza2NtYnE4bDdoZjNmNGN5NXhnZW
80dWRsYTkxZHVldTNxYTU0LzQ2a3FianZ5a2x1bnAxejU2dHh6a2hlbjdnamNpM2N5eDhnZ2twdHgy
NWk3NG1vNm15cXB4OWtsdnYzL2FrY2lpMjM5bXl6b24weHdqbnHxbm4zYjM0dyxCSG9zdDpodHRwOi
8vZG9tYWxvLm9ubGluZS9rc2V6Ymx4bHZvdTNRy21icThsN2hmM2Y0Y3k1eGdlbzR1ZGxhOTFkdWV1
M3FhNTQvNDZrcWJqdnlrbHVucDF6NTZ0eHpraGVuN2dqY2kzY3l4OGdna3B0eDI1aTc0bW82bXlxcH
g5a2x2djMvYWtjaWkyMzlteXpvcjB4d2pseHFubjNiMzR3LE1YOkRDUI9NVVRFWC13TGNzOG8xTIZF
VXRYeEo5bjl5ZixUQUc6VU5ERUY= | base64 -d
```

```
MHost:hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/
46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjl
xqnn3b34w,BHost:hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91
dueu3qa54/46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239
myzon0xwjlxqnn3b34w,MX:DCR_MUTEX-wLcs8o1NVEUtXxJ9n9yf,TAG:UNDEF
```

Figure 11: Decoding the base64 data in SchemaServerManager.Main reveals a configuration string

Figure 11 shows that the data decoded to a configuration string containing four values: MHost, BHost, MX, and TAG. We analyzed the code that parses this string and found that MHost and BHost were used as its main and backup command and control (C2) endpoints. Observe that the MHost and BHost values in Figure 11 are identical, so this sample did not have a backup C2 endpoint.

In dnSpy it is possible to give classes and methods meaningful names just as it is possible to name identifiers in IDA Pro. For example, the method SchemaServerManager.StopCustomer picks the name of a random running process. By right-clicking the StopCustomer identifier and choosing Edit Method, it is possible to change the method name to PickRandomProcessName, as shown in Figure 12.

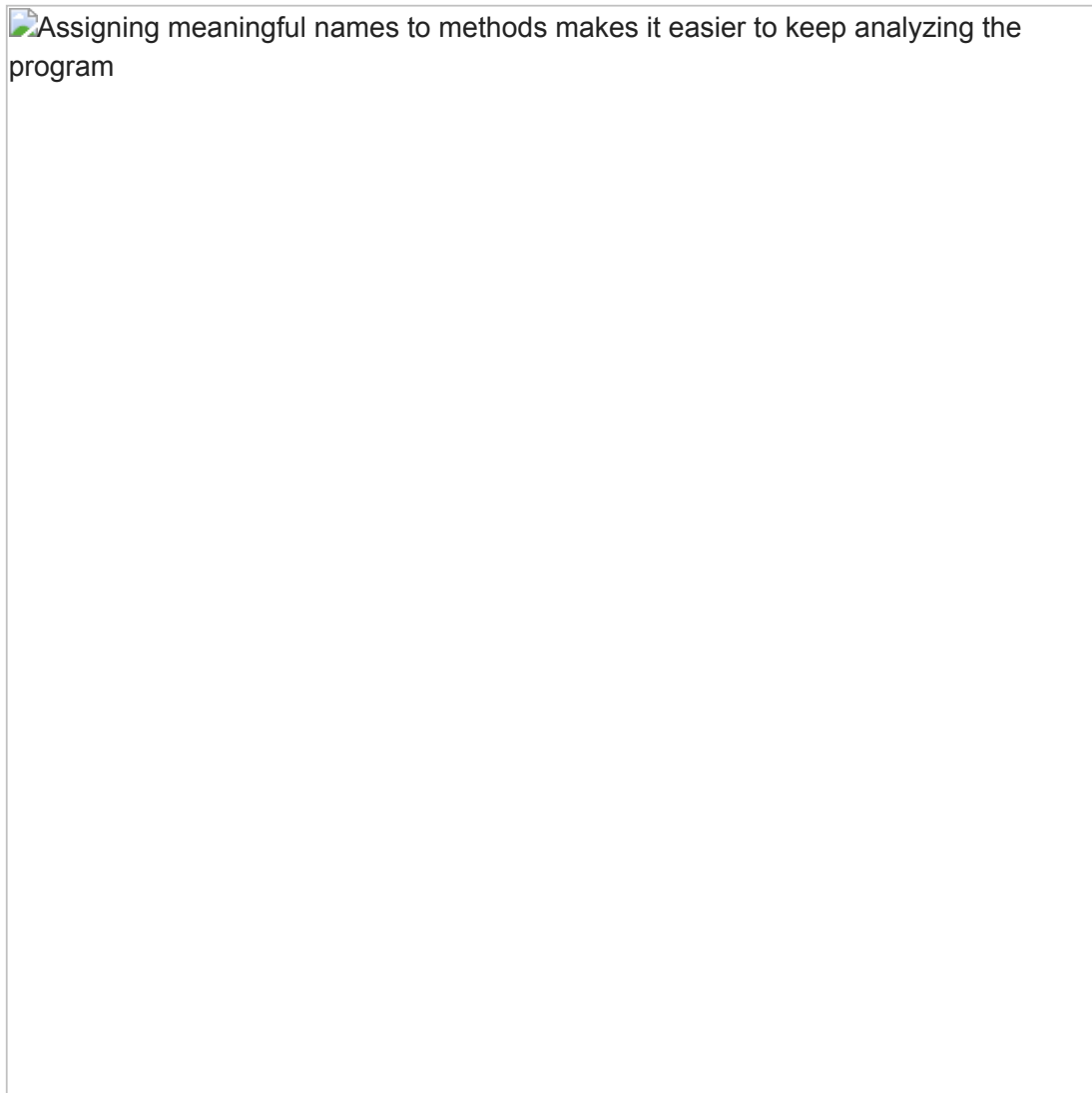


Figure 12:

Assigning meaningful names to methods makes it easier to keep analyzing the program
Continuing to analyze the SchemaServerManager.Main method revealed that the sample persists across reboots. The persistence algorithm can be summarized as follows:

1. The malware picks the name of a random running process, and then copies itself to %APPDATA% and C:\. For example, if svchost.exe is selected, then the malware copies itself to %APPDATA%\svchost.exe and C:\svchost.exe.
2. The malware creates a shortcut %APPDATA%\dotNET.lnk pointing to the copy of the malware under %APPDATA%.

3. The malware creates a shortcut named dotNET.lnk in the logged-on user's Startup folder pointing to %APPDATA%\dotNET.lnk.
4. The malware creates a shortcut C:\Sysdll32.lnk pointing to the copy of the malware under C:\.
5. The malware creates a shortcut named Sysdll32.lnk in the logged-on user's Startup folder pointing to C:\Sysdll32.lnk.
6. The malware creates the registry value HKCU\Software\Microsoft\Windows\CurrentVersion\Run\scrss pointing to %APPDATA%\dotNET.lnk.
7. The malware creates the registry value HKCU\Software\Microsoft\Windows\CurrentVersion\Run\Wininit pointing to C:\Sysdll32.lnk.

After its persistence steps, the malware checks for multiple instances of the malware:

1. The malware sleeps for a random interval between 5 and 7 seconds.
2. The malware takes the MD5 hash of the still-base64-encoded configuration string, and creates the mutex whose name is the hexadecimal representation of that hash. For this sample, the malware creates the mutex bc2dc004028c4f0303f5e49984983352. If this fails because another instance is running, the malware exits.

The malware then beacons, which also allows it to determine whether to use the main host (MHost) or backup host (BHost). To do so, the malware constructs a beacon URL based on the MHost URL, makes a request to the beacon URL, and then checks to see if the server responds with the HTTP response body "ok." If the server does not send this response, then the malware unconditionally uses the BHost; this code is shown in Figure 13. Note that since this sample has the same MHost and BHost value (from Figure 11), the malware uses the same C2 endpoint regardless of whether the check succeeds or fails.

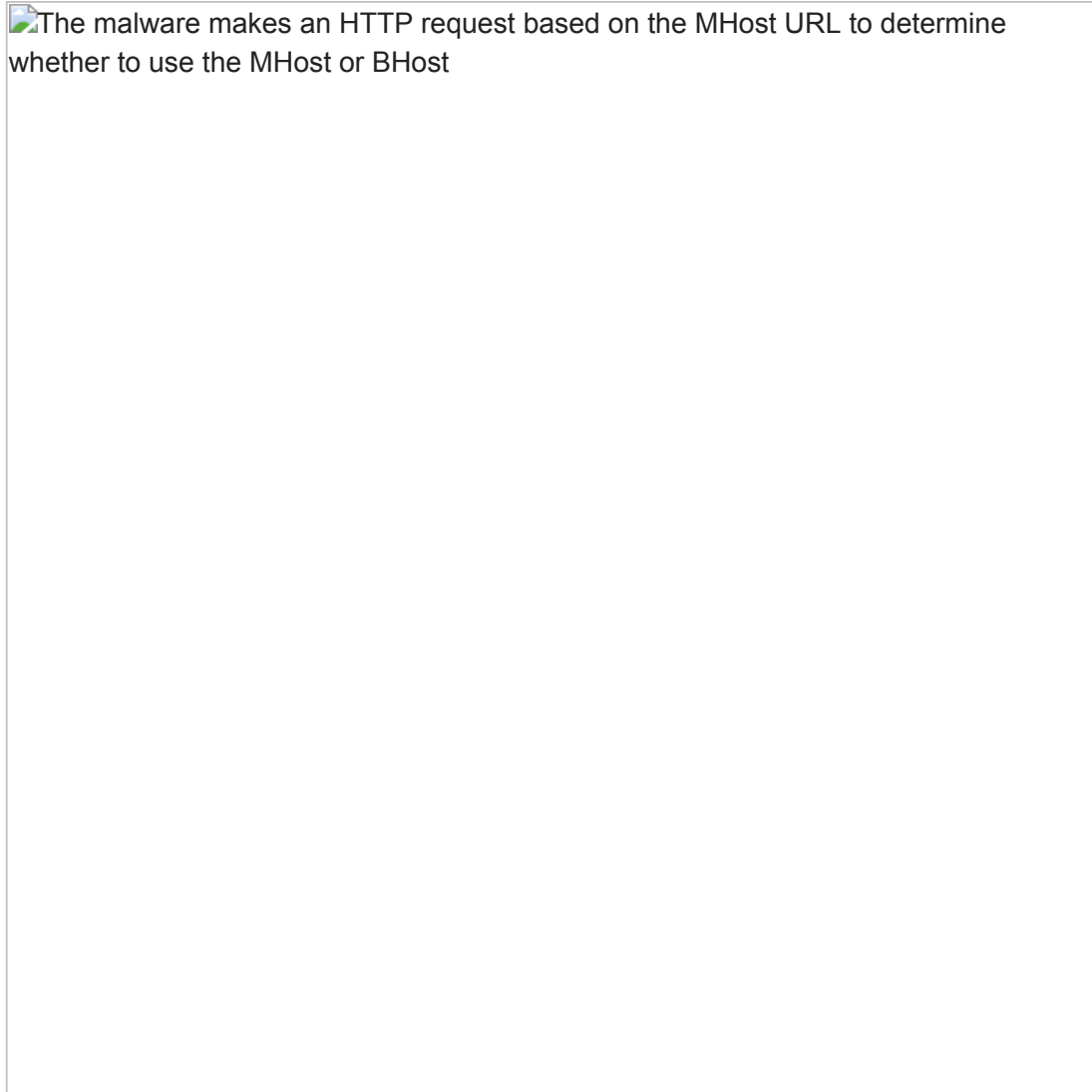


Figure 13: The

malware makes an HTTP request based on the MHost URL to determine whether to use the MHost or BHost

The full algorithm to obtain the beacon URL is as follows:

1. Obtain the MHost URL, i.e.,
`hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w.`
2. Calculate the SHA1 hash of the full MHost URL, i.e.,
`56743785cf97084d3a49a8bf0956f2c744a4a3e0.`
3. Remove the last path component from the MHost URL, and then append the SHA1 hash from above, and `?data=active`. The full beacon URL is therefore
`hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/56743785cf97084d3a49a8bf0956f2c744a4a3e0.php?data=active.`

After beaconing the malware proceeds to send and receive messages with the configured C2.

Messages and Capabilities

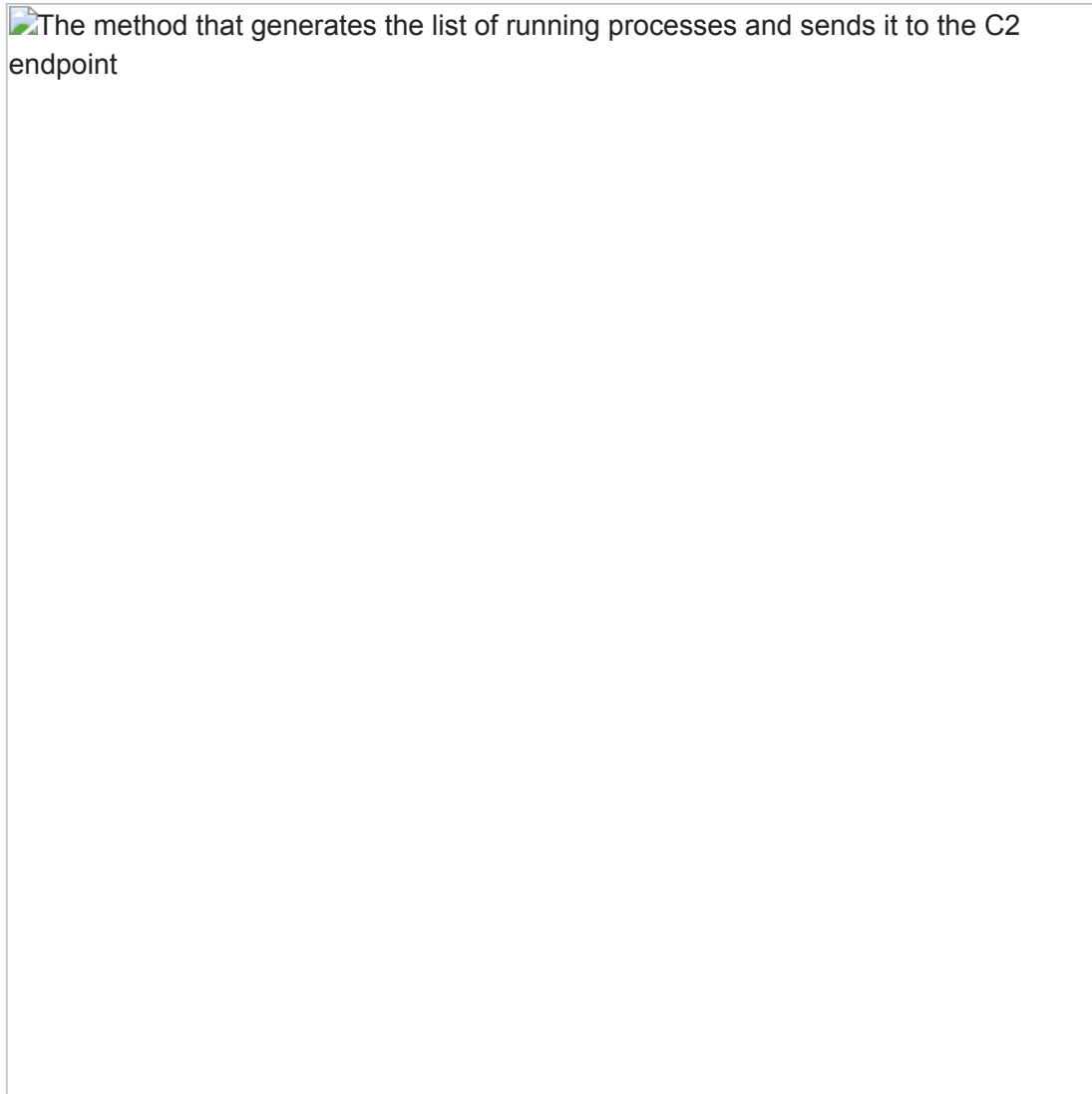
After performing static analysis of dfsds.exe to determine how it selects the C2 endpoint and confirming the C2 endpoint URL, we shifted to dynamic analysis in order to collect sample C2 traffic and make it easier to understand the code that generates and accepts C2 messages. Luckily for our analysis, the malware continues to generate requests to the C2 endpoint even if the server does not send a valid response. To listen for and intercept requests to the C2 endpoint (domalo[.]online) without allowing the malware Internet access, we used [FLARE's FakeNet-NG tool](#). Figure 14 shows some of the C2 requests that the malware made being captured by FakeNet-NG.



Figure 14:

FakeNet-NG can capture the malware's HTTP requests to the C2 endpoint

By comparing the messages generated by the malware and captured in FakeNet-NG with the malware's decompiled code, we determined its message format and types. Observe that the last HTTP request visible in Figure 14 contains a list of running processes. By tracing through the decompiled code, we found that the method `SchemaServerManager.ObserverWatcher.NewMerchant` generated this message. We renamed this method to `taskThread` and assigned meaningful names to the other methods it calls; the resulting code for this method appears in Figure 15.



The method that generates the list of running processes and sends it to the C2 endpoint

Figure 15: The

method that generates the list of running processes and sends it to the C2 endpoint

By analyzing the code further, we identified the components of the URLs that the malware used to send data to the C2 endpoint, and how they are constructed.

Beacons

The first type of URL is a beacon, sent only once when the malware starts up. For this sample, the beacon URL was always

`hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhenn7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/<hash>.php?data=active`, where `<hash>` is the SHA1 hash of the MHost URL, as described earlier.

GET requests, format 1

When the malware needs to send data to or receive data from the C2, it sends a message. The first type of message, which we denote as “format 1,” is a GET request to URLs of the form

`hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhenn7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w/<hash>.php? type=__ds_setdata&__ds_setdata_user=<user_hash>&__ds_setdata_ext=<message_hash>&__ds_setdata_data=<message>`, where:

- *<hash>* is MD5(SHA1(MHost)), which for this sample, is 212bad81b4208a2b412dfca05f1d9fa7.
- *<user_hash>* is a unique identifier for the machine on which the malware is running. It is always calculated as SHA1(OS_version + machine_name + user_name) as provided by the .NET System.Environment class.
- *<message_hash>* identifies what kind of message the malware is sending to the C2 endpoint. The *<message_hash>* is calculated as MD5(*<message_type>* + *<user_hash>*), where *<message_type>* is a short keyword identifying the type of message, and *<user_hash>* is as calculated above.
 - Values for *<message_type>* exist for each command that the malware supports; for possible values, see the “msgs” variable in the code sample shown in Figure 19.
 - Observe that this makes it difficult to observe the message type visually from log traffic, or to write a static network signature for the message type, since it varies for every machine due to the inclusion of the *<user_hash>*.
 - One type of message uses the value u instead of a hash for *<message_hash>*.
- *<message>* is the message data, which is not obscured in any way.

The other type of ordinary message is a getdata message. These are GET requests to URLs of the form `hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjcic3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w/<hash>.php? type=__ds_getdata&__ds_getdata_user=<user_hash>&__ds_getdata_ext=<message_hash>&__ds_getdata_key=<key>`, where:

- *<hash>* and *<user_hash>* are calculated as described above for getdata messages.
- *<message_hash>* is also calculated as described above for getdata messages, but describes the type of message the malware is expecting to receive in the server’s response.
- *<key>* is MD5(*<user_hash>*).

The server is expected to respond to a getdata message with an appropriate response for the type of message specified by *<message_hash>*.

GET requests, format 2

A few types of messages from the malware to the C2 use a different format, which we denote as “format 2.” These messages are GET requests of the form `hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjcic3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w/<user_hash>.<message_hash>`, where:

- *<user_hash>* is calculated as described above for getdata messages.
- *<message_hash>* is also calculated as described above for getdata messages, but describes the type of message the malware is expecting to receive in the server’s response. *<message_hash>* may also be the string comm.

Table 1 shows possible *<message_types>* that may be incorporated into *<message_hash>* as part of format 2 messages to instruct the server which type of response is desired. In contrast to format 1 messages, format 2 messages are only used for a handful of *<message_type>* values.

***<message_type>* Response desired**

s_comm	The server sends a non-empty response if a screenshot request is pending
m_comm	The server sends a non-empty response if a microphone request is pending
RDK	The server responds directly with keystrokes to replay
comm	The server responds directly with other types of tasking

Table 1: Message types when the malware uses a special message to request tasking from the server

POST requests

When the malware needs to upload large files, it makes a POST request. These POST requests are sent to `hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gji3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w/<hash>.php`, with the following parameters in the POST data:

- *name* is `<user_hash> + "." + <message_type>`, where `<user_hash>` is calculated as described above and `<message_type>` is the type of data being uploaded.
- *upload* is a file with the data being sent to the server.

Table 2 shows possible `<message_type>` values along with the type of file being uploaded.

<code><message_type></code>	Type of File
jpg	Screenshot
zipstealerlog	Cookie stealer log
wav	Microphone recording
file	Uploaded file
bmp	Webcam image
RD.jpg	Remote control screenshot

Table 2: Message types when files are uploaded to the server

Capabilities

By analyzing the code that handles the responses to the comm message (format 2), it was possible for us to inventory the malware's capabilities. Table 3 shows the keywords used in responses along with the description of each capability.

Keyword	Description
shell	Execute a shell command
deleteall	Recursively delete all files from C:, D:, F:, and G:
closecd	Close the CD-ROM drive door
setwallpaper	Change the background wallpaper
ddos	Send TCP and UDP packets to a given host or IP address
logoff	Log off the current user
keyboardrecorder	Replay keystrokes as if the user had typed them
fm_newfolder	Create a new folder
fm_rename	Rename or move a file
desktopHide	Hide desktop icons
keyloggerstart	Start logging keystrokes
exec_cs_code	Compile and execute C# code
msgbox	Open a Windows MessageBox
fm_upload	Transfer a file from the C2 to the client
rdp	Re-spawn the malware running as an administrator
fm_zip	Build a ZIP file from a directory tree and transfer it from the client to the C2
webcam	Take a webcam picture
fm_unzip	Unzip a ZIP file to a given path on the client
keyloggerstop	Stop logging keystrokes

fm_drives	Enumerate drive letters
cookiestealer	Transfer cookies and browser/FileZilla saved credentials to the C2
fm_delete	Recursively delete a given directory
dismon	Hide desktop icons and taskbar
fm_uploadu	Transfer a file from the C2 to the client
taskstart	Start a process
cleardesktop	Rotate screen
lcmd	Run shell command and send standard output back to C2
taskbarShow	Show taskbar
clipboard	Set clipboard contents
cookiestealer_file	Save cookies and credentials to a local file
newuserpass	Create a new local user account
beep	Beep for set frequency and duration
speak	Use speech synthesizer to speak text
openchat	Open chat window
taskbarHide	Hide the taskbar
RDStart	Start remote control over user's desktop
closechat	Close chat window
RDStop	Stop remote control over user's desktop
fm_opendir	List directory contents

uninstall	Remove the malware from the client
taskkill	Kill a process
forkbomb	Endlessly spawn instances of cmd.exe
fm_get	Transfer a file from the client to the C2
desktopShow	Show desktop icons
Clipboardget	Transfer clipboard contents to C2
playaudiourl	Play a sound file
opencd	Open the CD-ROM drive door
shutdown	Shut down the machine
restart	Restart the machine
browseurl	Open a web URL in the default browser

Table 3: Capabilities of DCRat

Proof-of-Concept Dark Crystal RAT Server

After gathering information from Dark Crystal RAT about its capabilities and C2 message format, another way to illustrate the capabilities and test our understanding of the messages was to write a proof-of-concept server. Here is a code snippet that we wrote containing a [barebones DCRat server written in Python](#). Unlike a real RAT server, this one does not have a user interface to allow the attacker to pick and launch commands. Instead, it has a pre-scripted command list that it sends to the RAT.

When the server starts up, it uses the Python BaseHTTPServer to begin listening for incoming web requests (lines 166-174). Incoming POST requests are assumed to hold a file that the RAT is uploading to the server; this server assumes all file uploads are screenshots and saves them to “screen.png” (lines 140-155). For GET requests, the server must distinguish between beacons, ordinary messages, and special messages (lines 123-138). For ordinary messages, `__ds_setdata` messages are simply printed to standard output, while the only `__ds_getdata` message type supported is `s_comm` (screenshot communications), to which the server responds with the desired screenshot dimensions (lines 63-84). For messages of type `comm`, the server sends four types of commands in sequence: first, it hides the desktop icons; then, it causes the string “Hello this is tech support” to be spoken; next, it displays a message box asking for a password; finally, it launches the Windows Calculator (lines 86-121).

Figure 16 shows the results when Dark Crystal RAT is run on a system that has been configured to redirect all traffic to domalof[.]online to the proof-of-concept server we wrote.



Figure 16: The

results when a Dark Crystal RAT instance communicates with the proof-of-concept server

Other Work and Reconnaissance

After reverse engineering Dark Crystal RAT, we continued reconnaissance to see what additional information we could find. One limitation to our analysis was that we did not wish to allow the sample to communicate with the real C2, so we kept it isolated from the Internet. To learn more about Dark Crystal RAT we tried two approaches: the first was to browse the Dark Crystal RAT website ([files.dcrat\[.\]ru](http://files.dcrat[.]ru)) using Tor, and the other was to take a look at YouTube videos of others' experiments with the "real" Dark Crystal RAT server.

Dark Crystal RAT Website

We found that Dark Crystal RAT has a website at [files.dcrat\[.\]ru](http://files.dcrat[.]ru), shown in Figure 17. Observe that there are options to download the RAT itself, as well as a few plugins; the DCLIB extension is consistent with the plugin loading code we found in the RAT.

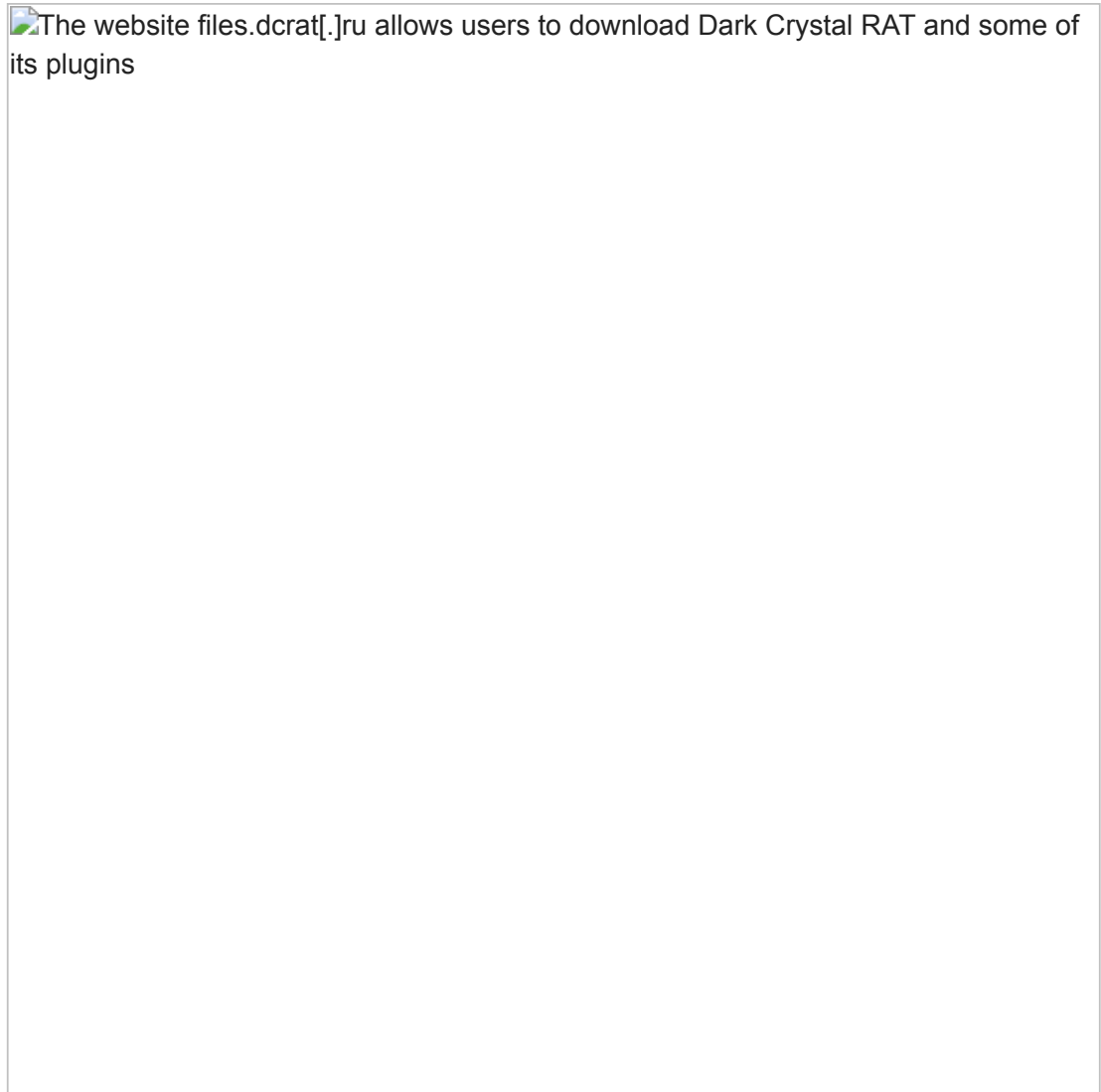


Figure 17: The

website files.dcrat[.]ru allows users to download Dark Crystal RAT and some of its plugins. Figure 18 shows some additional plugins, including plugins with the ability to resist running in a virtual machine, disable Windows Defender, and disable webcam lights on certain models. No plugins were bundled with the sample we studied.

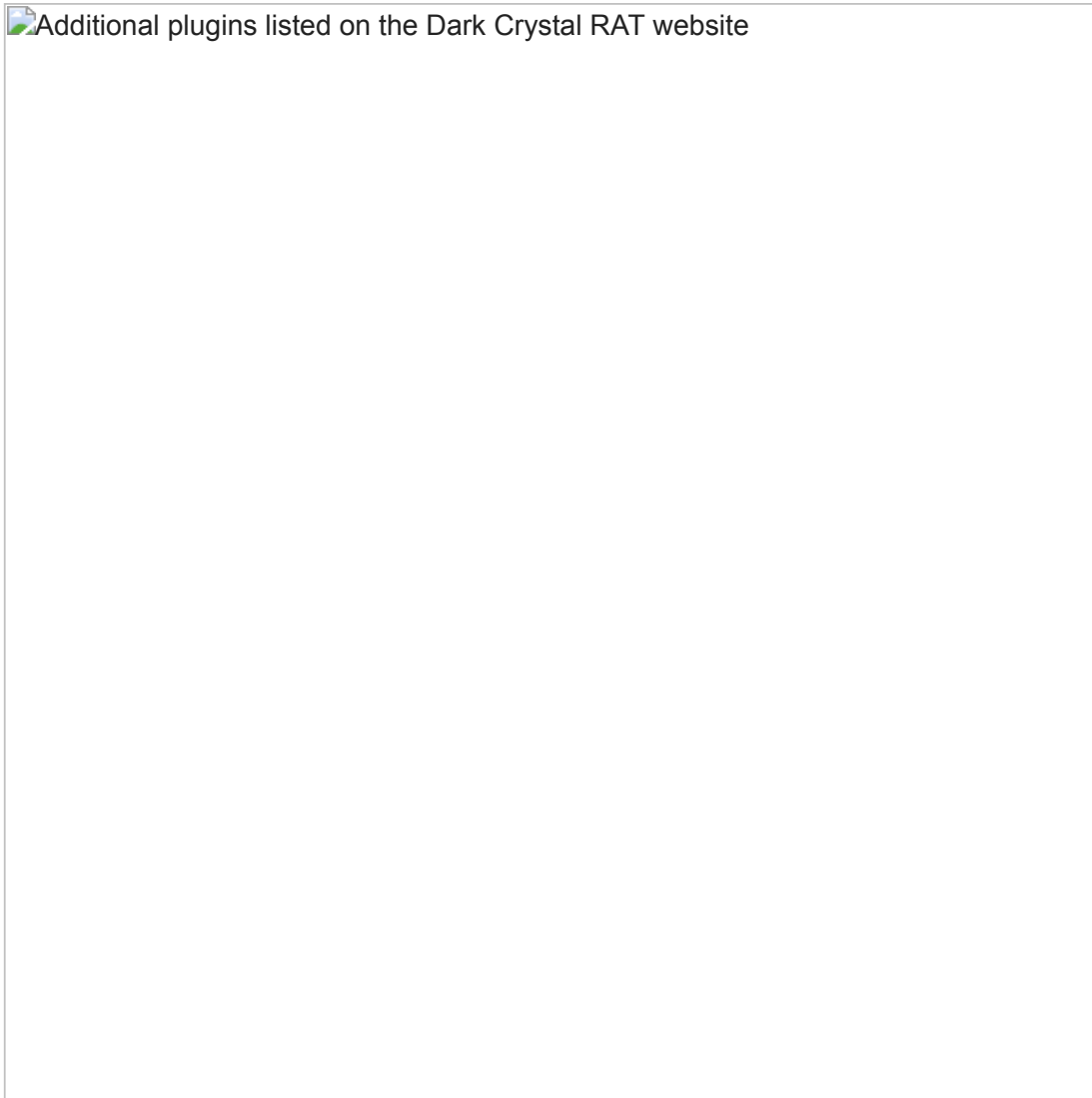



Figure 18:

Additional plugins listed on the Dark Crystal RAT website

Figure 19 lists software downloads on the RAT page. We took some time to look at these files; here are some interesting things we discovered:

- The DCRat listed on the website is actually a “builder” that packages a build of the RAT and a configuration for the attacker to deploy. This is consistent with the name DCRatBuild.exe shown back in Figure 4. In our brief testing of the builder, we found that it had a licensing check. We did not pursue bypassing it once we found public YouTube videos of the DCRat builder in operation, as we show later.
- The DarkCrystalServer is not self-contained, rather, it is just a PHP file that allows the user to supply a username and password, which causes it to download and install the server software. Due to the need to supply credentials and communicate back with dcrat[.]ru (Figure 20), we did not pursue further analysis of DarkCrystalServer.

The RAT page lists software for the RAT, the server, an API, and plugin development

RAT page lists software for the RAT, the server, an API, and plugin development

Figure 19: The

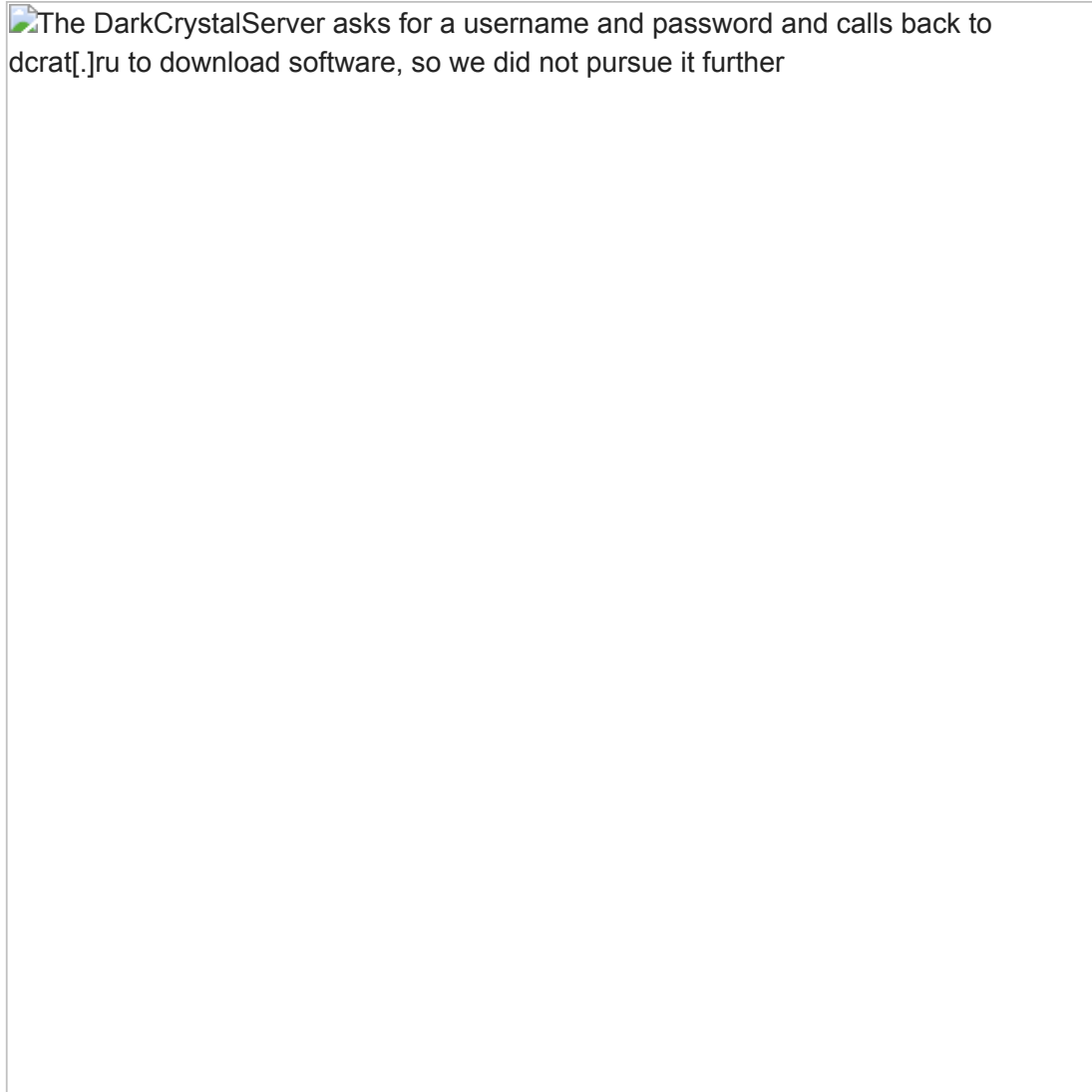


Figure 20: The

DarkCrystalServer asks for a username and password and calls back to dcrat[.]ru to download software, so we did not pursue it further

YouTube Videos

As part of confirming our findings about Dark Crystal RAT capabilities that we obtained through reverse engineering, we found some YouTube demonstrations of the DCRat builder and server.

The YouTube user *LIKAR* has a [YouTube demonstration of Dark Crystal RAT](#). The author demonstrates use of the Dark Crystal RAT software on a server with two active RAT instances. During the video, the author browses through the various screens in the software. This made it easy to envision how a cyber threat would use the RAT, and to confirm our suspicions of how it works.

Figure 21 shows a capture from the video at [3:27](#). Note that the Dark Crystal RAT builder software refers to the DCRatBuild package as a “server” rather than a client. Nonetheless, observe that one of the options was a type of Java, or C# (Beta). By watching this YouTube video and doing some additional background research, we discovered that Dark Crystal RAT has existed for some time in a Java version. The C# version is relatively new. This explained why we could not find much detailed prior reporting about it.


 A YouTube demonstration revealed that Dark Crystal RAT previously existed in a Java version, and the C# version we analyzed is in beta

Figure 21: A YouTube demonstration

revealed that Dark Crystal RAT previously existed in a Java version, and the C# version we analyzed is in beta

Figure 22 shows another capture from the video at 6:28. The functionality displayed on the screen lines up nicely with the “msgbox”, “browseurl”, “clipboard”, “speak”, “opencd”, “closecd”, and other capabilities we discovered and enumerated in Table 6.


 A YouTube demonstration confirmed many of the Dark Crystal RAT capabilities we found in reverse engineering



Figure 22: A YouTube demonstration

confirmed many of the Dark Crystal RAT capabilities we found in reverse engineering

Conclusion

In this post we walked through our analysis of the sample that the threat intel team provided to us and all its components. Through our initial triage, we found that its “dfsds.exe” component is Dark Crystal RAT. We found that Dark Crystal RAT was a .NET executable, and reverse engineered it. We extracted the malware’s configuration, and through dynamic analysis discovered the syntax of its C2 communications. We implemented a small proof-of-concept server to test the correct format of commands that can be sent to the malware, and how to interpret its uploaded screenshots. Finally, we took a second look at how actual threat actors would download and use Dark Crystal RAT.

To conclude, indicators of compromise for this version of Dark Crystal RAT (MD5: 047af34af65efd5c6ee38eb7ad100a01) are given in Table 4.

Indicators of Compromise

Dark Crystal RAT (dfsds.exe)

Handle artifacts

Mutex name bc2dc004028c4f0303f5e49984983352

Registry artifacts

Registry value HKCU\Software\Microsoft\Windows\CurrentVersion\Run\scrss

Registry value	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\Wininit
File system artifacts	
File	C:\Sysdll32.Ink
File	%APPDATA%\dotNET.Ink
File	Start Menu\Programs\Startup\Sysdll32.Ink
File	Start Menu\Programs\Startup\dotNET.Ink
File	%APPDATA%\<random process name>.exe
File	C:\<random process name>.exe
Network artifacts	
HTTP request	hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/212bad81b4208a2b412dfca05f1d9fa7.php?data=active
HTTP request	hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w212bad81b4208a2b412dfca05f1d9fa7.php? type=__ds_getdata&__ds_getdata_user=<user_hash>&__ds_getdata_ext=<message_hash>&__ds_getdata_key=<key>
HTTP request	hxxp://domalo[.]online/ksezblxlvou3kcmbq8l7hf3f4cy5xgeo4udla91dueu3qa54/46kqbjvyklunp1z56txzkhen7gjci3cyx8ggkptx25i74mo6myqpx9klvv3/akcii239myzon0xwjlxqnn3b34w/<user_hash>.<message_hash>
TCP connection	domalo[.]online:80
TCP connection	ipinfo[.]ip
DNS lookup	domalo[.]online
DNS lookup	ipinfo[.]ip

Strings

Static string	DCRatBuild
---------------	------------

Table 4: IoCs for this instance of DCRat

FireEye Product Support for Dark Crystal RAT

Table 5 describes how FireEye products react to the initial sample (MD5: b478d340a787b85e086cc951d0696cb1) and its Dark Crystal RAT payload, or in the case of Mandiant Security Validation, allow a stakeholder to validate their own capability to detect Dark Crystal RAT.

FireEye Product	Support for Dark Crystal RAT
FireEye Network Security (NX)	Backdoor.Plurox detection
FireEye Email Security (EX & ETP)	Backdoor.MSIL.DarkCrystal, Backdoor.Plurox, Malware.Binary.exe, Trojan.Vasal.FEC3, Win.Ransomware.Cerber-6267996-1, fe_ml_heuristic detections
FireEye Endpoint Security (HX)	Trojan.GenericKD.32546165, Backdoor.MSIL.DarkCrystal detections
FireEye Malware Analysis (AX)	Backdoor.Plurox.FEC2 detection
FireEye Detection on Demand (DoD)	Backdoor.Plurox.FEC2, FireEye.Malware detections
Mandiant Security Validation	Built-in Action coming soon

Table 5: Support in FireEye products to detect Dark Crystal RAT or validate detection capability