Investigating the LuminosityLink Remote Access Trojan Configuration

By Josh Grunzweig

Published: 2016-07-08 · Archived: 2026-04-05 12:44:36 UTC

In recent weeks, I've spent time investigating the LuminosityLink Remote Access Trojan's (RAT) embedded configuration. For those unaware, LuminosityLink is a malware family costing $40 that purports to be a system administration utility. However, when executed, the malware leverages a very aggressive keylogger, as well as a number of other malicious features that allow an attacker to gain full control over a victim machine.

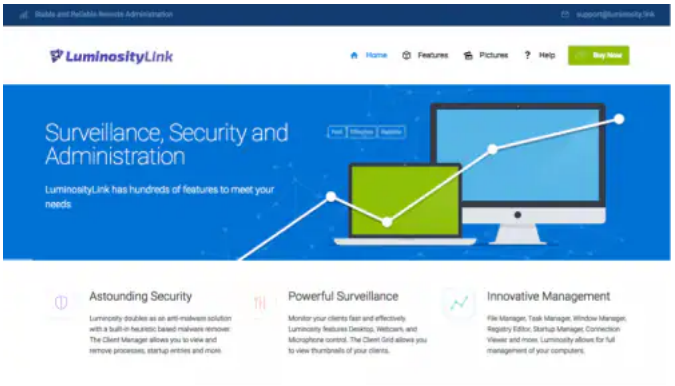

Figure 1 LuminosityLink website

At the request of a coworker, I was asked to extract the configuration of a LuminosityLink sample, and while I could have simply executed the malware in a sandboxed environment and pulled the configuration from memory, I chose to see if I could perform the same action against the static binary.

This led to me understanding how the configuration is encrypted within the binary, as well as how to parse that configuration. I've created a script to perform this action against an unaltered LuminosityLink malware sample, which I will be sharing within this blog post. Furthermore, I looked at the roughly 18,000 LuminosityLink samples Palo Alto Networks has collected over time, and using this script, I was able to extract the configurations of 14,700 samples. (This data can be found in the Appendix section of this post.)

Overview of LuminosityLink

Originally surfacing in May 2015, LuminosityLink's popularity has been on the rise, as shown in the following chart. To date, Palo Alto Networks has tracked approximately 50,000 attempted infections of LuminosityLink against our customers.

Figure 2 AutoFocus graph of LuminosityLink sessions over time.

LuminosityLink currently sells for $40 and can be purchased directly from its author. This package allows attackers to host a LuminosityLink server as well as generate customized binaries, which are obfuscated with ConfuserEx 0.4.0. ConfuserEx is an open-source project that obfuscates the underlying .NET code, making it much more difficult for reverse engineers that decompile it. This is important to note for later when we discuss determining how to reverse-engineer the encryption process.

As mentioned previously, a number of configuration options are included, as we can see in the following screenshot.

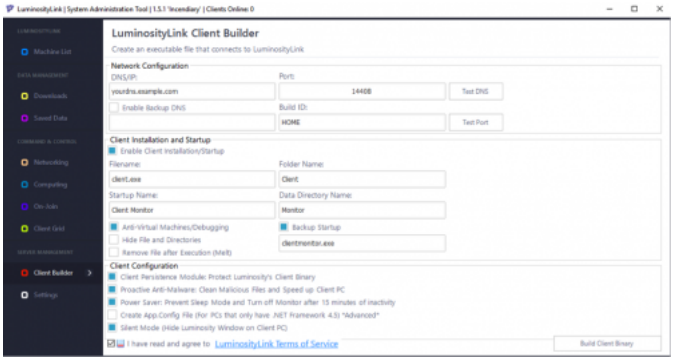

Figure 3 Client configuration options in LuminosityLink

Once executed, attackers are given a wealth of options, including keylogging, remote desktop, password stealing, and interacting with a shell on the device.

Reverse-Engineering the Configuration

The first step in parsing out the configuration of LuminosityLink is to extract it statically. I initially opened up a clean LuminosityLink sample using a program named dnSpy to search for clues as to where the configuration might be stored. (As a quick aside, I highly recommend dnSpy, as it not only does a great job of decompiling the provided .NET binary, but it also comes equipped with a built-in debugger, which is instrumental in tackling problems such as the one we are facing.)

When initially opening up a sample binary, I didn't expect much as I knew the sample was obfuscated using ConfuserEx. However, looking at the resources of the sample, I saw some strings that looked promising.

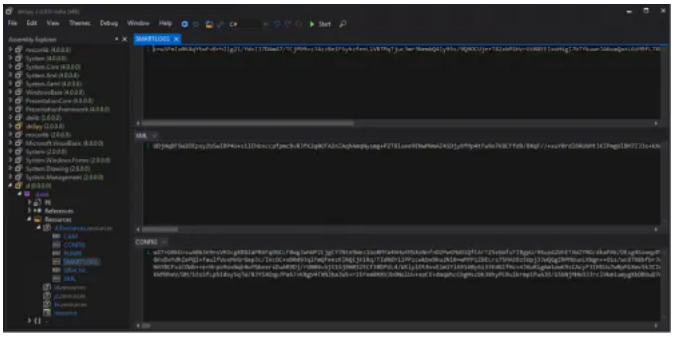

Figure 4 Embedded resource strings in LuminosityLink

As we can see, the malware contains a number of resources that in turn contain what appears to be Base64-encoded data. The "SMARTLOGS", "XML", and "CONFIG" resources all contain a wealth of data, which, at this point, is still unknown. Unfortunately, decoding these strings results in garbage, which likely means some other form of encryption is being used underneath.

I continued to investigate the underlying code, which, while obfuscated, still provides a high-level idea of what various classes are doing. Using imported namespaces, API calls and certain un-obfuscated strings, we're able to get clues as to what is going on within the program. Specifically, we see the fd() class using the namespace of 'System.Security.Cryptography', which certainly merits investigation as we suspected crypto being used against the earlier referenced resource strings.

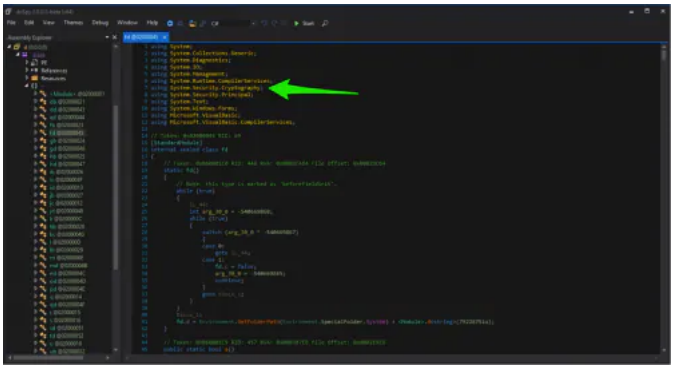

Figure 5 Cryptography namespace being used by LuminosityLink

As we further investigate this class, we see references to the following classes and functions:

- MD5CryptoServiceProvider
- ComputeHash
- FromBase64String
- RijndaelManaged

At this point, I turned to my debugger in an attempt to see how these strings were handled. I set breakpoints on various calls previously mentioned. Specifically, the breakpoint on the RijndaelManaged class yields excellent results.

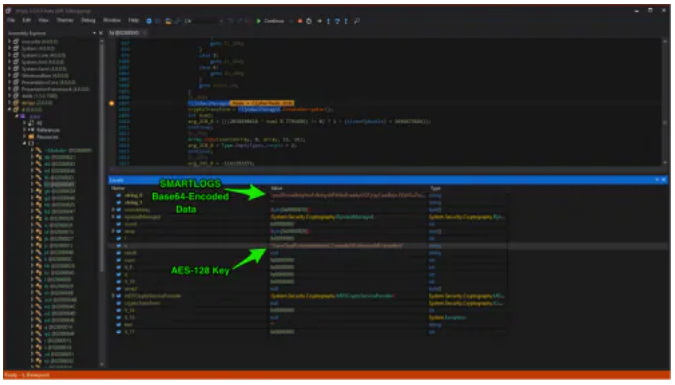

Figure 6 Successful recovery of AES-128 key

We're able to not only verify that AES-128 encryption is used, but also verify that the "SMARTLOGS" resource string is being used. We also are able to identify the string being used as the key, which in this particular example is "\\ecnOnuR\\noisreVtnerruC\\swodniW\\tfosorciM\\erawtfoS". Further investigation reveals that this string is hashed using the MD5 algorithm. The first 15 bytes of this hash is concatenated with the entire 16 bytes of the hash, followed by a null byte. We can replicate this decryption process in Python as such:

```
def decrypt_data(data, key_string):

decoded_data = base64.b64decode(data)

m = hashlib.md5()

m.update(key_string)

md5 = m.digest()

key = md5[0:15]+md5+"\x00"

mode = AES.MODE_ECB

iv = '\x00' * 16

e = AES.new(key, mode, iv)

return e.decrypt(decoded_data)
```

We can further verify that this is correct using the above code. In the following example, the data variable has been set with the base64-encoded string found within the "SMARTLOGS" resource, and the key_string variable has been set to the reversed registry key previously mentioned.

```
>>> print decrypt_data(data, key_string)

172.16.1.100|11111|backup.dns.com|filename.exe|startup_name|folder_name|data_directory_name|backup_startup.exe|df5e0e1c667c399588b014cb9
```

Using trial and error, we're able to successfully map each variable witnessed in the above configuration. The last variable proves to be quite interesting, as each character except for '1' maps to a particular configuration option. The following mapping was determined:

- *i : Enable Client Installation/Startup*
- *d : Client Persistence Module: Protect Luminosity's Client Binary*
- *s : Silent Mode (Hide Luminosity Window on Client PC)*

- *a : Proactive Anti-Malware: Clean Malicious Files and Speed up Client PC*
- *n : Power Saver: Prevent Sleep Mode and Turn off Monitor after 15 minutes of inactivity*
- *m : Remove File after Execution (Melt)*
- *v : Anti-Virtual Machines/Debugging*
- *h : Hide File and Directories*
- *b : Backup Startup*

Using this knowledge, we can parse the above configuration, which yields the following results.

```
1   Domain/IP:           172.16.1.100
2   Port:                11111
3   Backup DNS:          backup.dns.com
4   Filename:            filename.exe
5   Startup Name:        startup_name
6   Folder Name:         folder_name
7   Data Directory Name:  data_directory_name
8   Backup Startup Exe:   backup_startup.exe
9   Mutex:               df5e0e1c667c399588b014cb9b4ae7b33c9c0b4cac9d4169f39197e41e9a43a4
10  Build ID:            BuildID
11  Settings:
12   [X] Enable Client Installation/Startup
13   [X] Client Persistence Module: Protect Luminosity's Client Binary
14   [X] Silent Mode (Hide Luminosity Window on Client PC)
15   [ ] Proactive Anti-Malware: Clean Malicious Files and Speed up Client PC
16   [X] Power Saver: Prevent Sleep Mode and Turn off Monitor after 15 minutes of inactivity
17   [ ] Remove File after Execution (Melt)
18   [ ] Anti-Virtual Machines/Debugging
19   [ ] Hide File and Directories
20   [X] Backup Startup
```

Parsing LuminosityLink Configurations at Scale

Using this knowledge, we created a script to parse the configuration of a given sample. The script searches for strings that appear to be base64-encoded with a length greater than 50 and takes a brute-force approach. While not elegant, it does the job quite successfully. The script can be downloaded in the Appendix section of this blog post.

Going through Palo Alto Networks repository of samples, we found roughly 18,000 files tagged as LuminosityLink. For these 18,000 samples, we applied our static configuration extraction and parsing script and successfully retrieved about 4,500 configurations. The remaining samples were packed beyond the built-in ConfuserEx obfuscation routine, and as such, the raw configuration strings were not present.

These samples were run through a local instance of the open-source Cuckoo Sandbox, where the process dumps were extracted. The same script was applied to these process dumps, where we were able to obtain an additional 10,200 configurations, leaving us with a total of 14,700 parsed LuminosityLink configurations.

As the samples were processed, further keys were discovered to be used by the author. The following additional three strings were used to generate the keys to LuminosityLink samples.

- This conf'ig contains nothing useful. Quit acting as if you're cool by decrypting it.
- Resources.SMARTLOGS
- Specify a Password

It appears as though the author of LuminosityLink is not without a sense of humor. Additionally, as we parsed older samples, it was discovered that the configuration made a change sometime between February and June of this year. Fewer options were available in the configuration of older samples. The provided script accounts for these differences and various keys used.

Using the aggregated data from the 14,700 configurations, the following high-level statistics were pulled:

Figure 7 Prevalence of enabled settings in LuminosityLink configurations

Top C2 TLDs/IP Addresses

- [3308] ddns[.]net
- [2537] duckdns[.]org
- [904] no-ip[.]biz
- [670] chickenkiller[.]com
- [378] no-ip[.]org
- [377] mooo[.]com
- [242] fishdns[.]com
- [174] no-ip[.]info
- [165] ignorelist[.]com
- [157] freedns[.]su

Top Build IDs

- [4829] HOME
- [83] Client
- [82] crtx1
- [71] M4CHINATION
- [65] CSGO
- [65] NEW
- [59] Slave
- [47] CAPO
- [44] Youtube
- [42] PROJECT.D

Top Executable Names

- [1973] sysmon.exe
- [1831] client.exe
- [1254] helper.exe
- [1207] repair.exe
- [1087] winlogon.exe
- [509] svchost.exe
- [315] Luminosity.exe
- [83] WinCOMHost.exe
- [82] chrome.exe
- [61] windowsbootapp.exe

Top Ports

- [1866] 6318
- [1055] 1400
- [493] 1604
- [412] 1337
- [214] 3175
- [182] 22028
- [162] 9045
- [119] 2122

- [115] 100
- [113] 9999

The parsed configuration data, provided in the CSV file format, is being freely provided to the security community in the hope that protections will be created against this threat.

Conclusion

LuminosityLink, while marketed as a systems administration utility, is a formidable keylogger and backdoor used by a large number of criminals. To date, Palo Alto Networks has witnessed over 50,000 attempted infections of LuminosityLink, encompassing 18,000 unique samples. The malware is cheap and readily available to the public, making this a dangerous threat to both organizations and individuals alike.

By reverse-engineering LuminosityLink samples, we were able to statically extract and parse the embedded configuration, which in turn provides valuable information about with what hosts and ports the malware is configured to communicate. It also provides information regarding configured protections configured within the executable, as well as installation information.

Palo Alto Networks customers are protected from this threat in the following ways:

An AutoFocus tag can be used to track this malware family

- All LuminosityLink samples are appropriately marked as malicious in WildFire
- All identified domains are flagged as malicious
- Network traffic is appropriately identified and blocked as threat ID #14460 (LumonosityLinkRAT.Gen Command And Control Traffic)

Appendix

An extraction and parsing script for LuminosityLink samples can be found here.

A script to be used to parse plain configuration strings can be found here.

A CSV file containing all of the configuration data extracted from Palo Alto Networks sample set can be found here.
