Looking Back at LiteDuke – One Night in Norfolk

Published: 2020-05-18 · Archived: 2026-04-05 14:00:12 UTC

Last October (2019), ESET published extensive research regarding additional tooling from the "Dukes" adversary, which analysts have traditionally aligned with [APT29/Cozy Bear] operations. While conducting some unrelated research, I came across a LiteDuke sample and decided to take a deeper dive into the mechanics of its loader and its malware.

The original ESET publication covers the key details at a high level; in addition, this malware is old and has likely been discontinued for several years. This blog post is intended to document some additional lower-level details for operational comparison purposes and general learning.

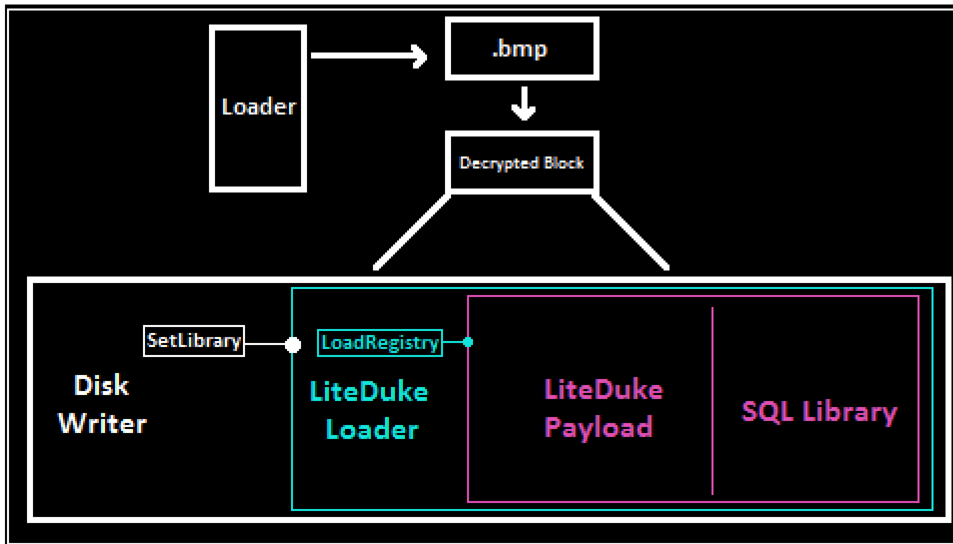
Loading Structure

The loading structure identified by this blog is similar to the one described by ESET (differences may be attributable to a different variant of the malware being examined or more likely just a different analytic perspective). This blog identified a DLL component of the loading chain on VirusTotal with the following properties:

MD5: 79ac8431f484ab50de137544d88d5c83
SHA1: 5c01fcc8c9cf72bd1d5088ee739939353b41b1d6
SHA256: b9670993179beb5c1af574c156c0379e058b2f4a6635a83b98766a3339431274
Compilation Timestamp: 2014-10-02 11:09:36
Export: ShowDialog

This DLL was likely preceded by another executable, script, or command-line invocation, as it requires that the "**ShowDialog**" export be called to deploy the next stage. This DLL allocates a segment of virtual memory and decodes content from a .bmp resource. The decoded memory is a block of *four* executable files:

1) A DLL with an export named **SetLibrary** (that expects #2-4 to be present).
2) A DLL with several exports, including one named **LoadRegistry** (that expects #3-4 to be present).
3) The LiteDuke payload, temporarily decrypted (that expects #4 to be present).
4) A SQLite library that is actually a component of the LiteDuke payload.



LiteDuke Loading Structure

The SetLibrary export from the top component (#1) performs the following actions:

1) Creates a directory at C:\ProgramData\User\
2) Writes a file named NTUSER.DAT here. This file is #2 from the list above, but with the #3/4 blocks encrypted.
3) Calls rundll32.exe [path to NTUSER.DAT],LoadRegistry

This invokes the LiteDuke loader, which decrypts and runs the LiteDuke payload via the LoadRegistry export.

LiteDuke Payload

The LiteDuke payload can be broken down into three core workflows:

1) A setup routine, which establishes persistence, creates the SQL database file, and launches the C2 routines.
2) A "higher-level" C2 block, which performs basic maintenance tasks.
3) A "lower-level" C2 block, which carries out a large number of commands.

Setup Routine

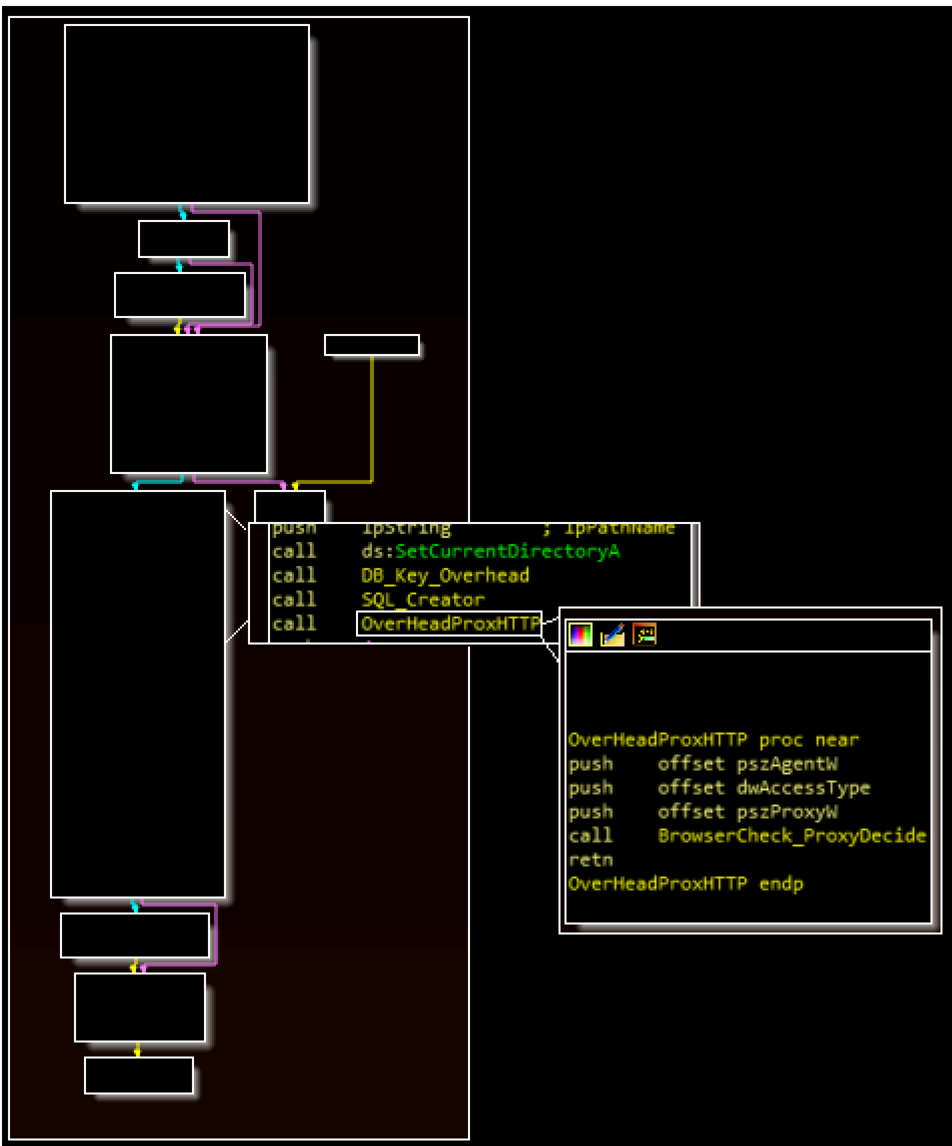
As part of its setup, the malware creates a mutex named "NtUserRegistryService" and terminates the process if the mutex already exists. The malware then calls three functions. The first of these creates the encryption key for a local database, the second creates this database (written to the same ProgramData\User folder with the name NTUSER.DAT.BAK), and the third assigns a User Agent used for HTTP communication.

The SQL database contains three tables: config, modules, and objects. The config table contains information for the C2 routine and can be updated (as this post will later show). The modules field appears to allow for additional code execution (although, as ESET noted, none have been publicly documented). The purpose of the objects field is less clear; a handful of functions in the binary suggest it could be used to hold files, although the phrase "object" itself has a distinct meaning in object databases.

Database Structure | Browse Data | Edit Pragmas | Execute SQL

Name	Type	Schema
Tables (4)		
config		CREATE TABLE config (uid INTEGER PRIMARY KEY AUTOINCREMENT, agent_id CHAR(128), remote_host CHAR(256), re
uid	INTEGER	"uid" INTEGER PRIMARY KEY AUTOINCREMENT
agent_id	CHAR(128)	"agent_id" CHAR(128)
remote_host	CHAR(256)	"remote_host" CHAR(256)
remote_port	integer	"remote_port" integer
remote_path	char(1024)	"remote_path" char(1024)
update_interval	integer	"update_interval" integer
server_key	CHAR(32)	"server_key" CHAR(32)
rcv_header	CHAR(64)	"rcv_header" CHAR(64)
modules		CREATE TABLE modules (uid INTEGER PRIMARY KEY, version char(255), code blob, config blob, type char(10), md5sum
uid	INTEGER	"uid" INTEGER
version	char(255)	"version" char(255)
code	blob	"code" blob
config	blob	"config" blob
type	char(10)	"type" char(10)
md5sum	char(32)	"md5sum" char(32)
autorun	INTEGER	"autorun" INTEGER
objects		CREATE TABLE objects (uid INTEGER PRIMARY KEY AUTOINCREMENT, name CHAR(255), body blob, type char(10), md
uid	INTEGER	"uid" INTEGER PRIMARY KEY AUTOINCREMENT
name	CHAR(255)	"name" CHAR(255)
body	blob	"body" blob
type	char(10)	"type" char(10)
md5sum	char(32)	"md5sum" char(32)

SQL table and fields set up by the binary

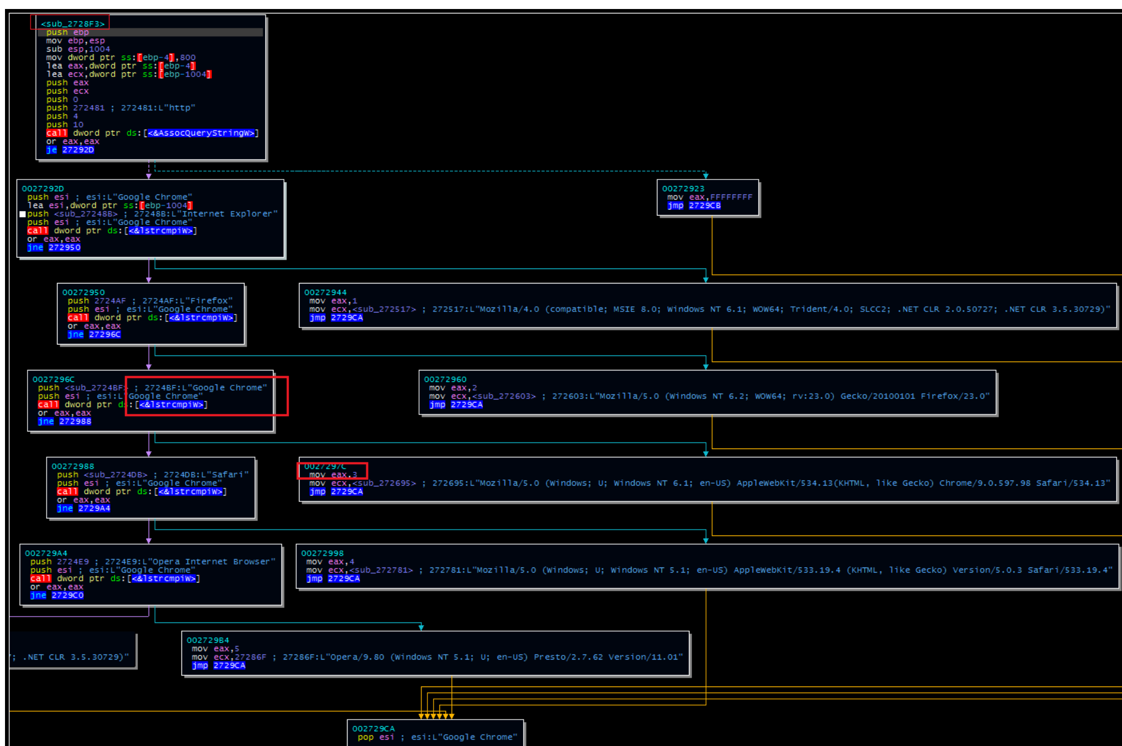


Segment of setup workflow leading to the browser query

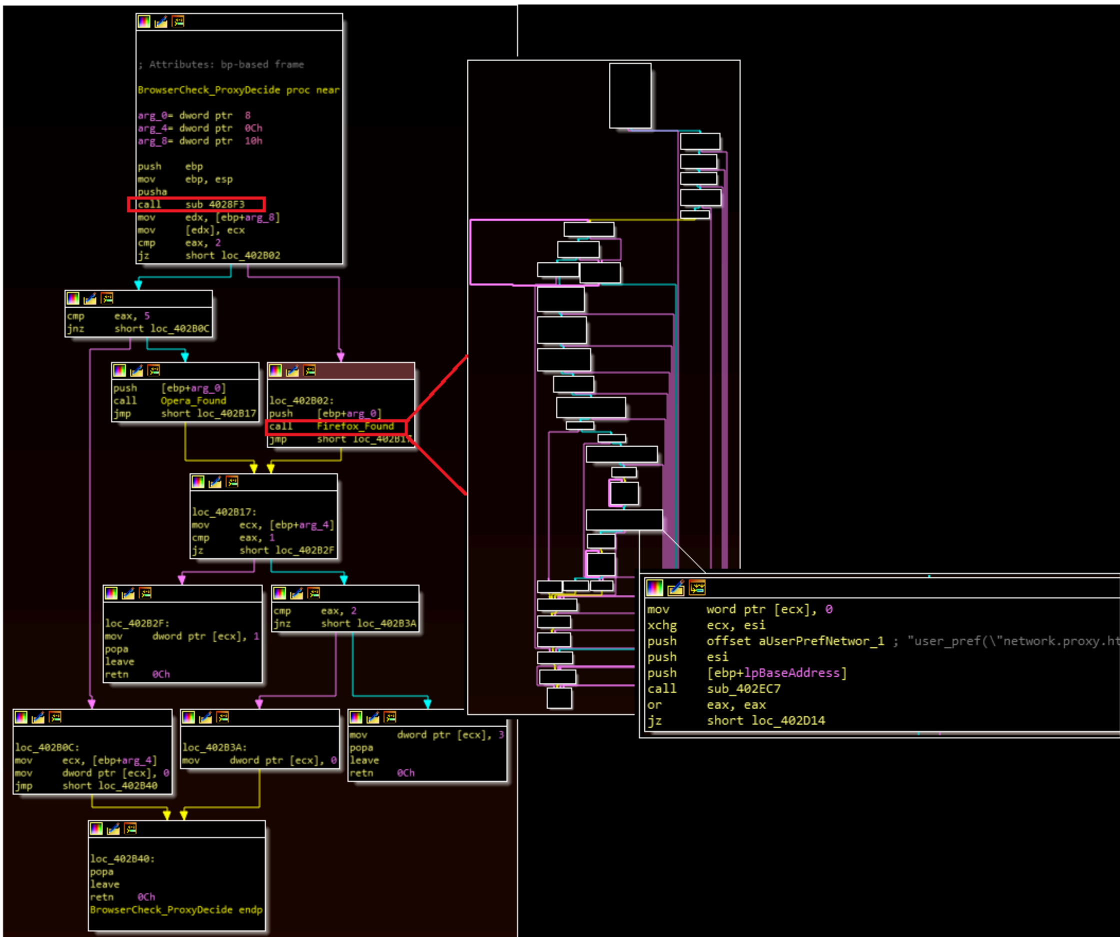
To select a User Agent, the malware uses the AssocStringQuery API to search the root registry for the default program associated with the “http” extension. In practical terms, this returns the default browser, which is checked against a hardcoded list. The malware assigns a User Agent corresponding to the default browser and also moves a value into the EAX register based on which browser was chosen.

If this check sets the EAX value to either 2 or 5 (Firefox and Opera respectively), the malware will *also* attempt to retrieve and use proxy configuration information stored in the browser settings.

The first image below shows the browser/User Agent association function. The second image below shows the parent function that calls both this User Agent function and the proxy function. This second image corresponds with the “BowserCheck_ProxyDecide” label in the setup routine image above.



User Agent selection (right click and open in a new tab to expand)



Parent function for user agent and proxy selections

Finally, the malware takes two additional setup steps.

First, the malware creates a persistence mechanism. As ESET previously noted, this takes the form of a .lnk file. In this case, the file is named NTUSER.DAT.lnk and placed inside of the same ProgramData\User\ directory as the database and loader. The .lnk file calls rundll32.exe, passing the .DAT file and the LoadRegistry DLL, forcing the application to run at login.

The application *does* have handling for administrative privileges that includes a SID check for default SIDs and the %systemprofile% ProfileImage path, but the difference in outcome from this routine was not immediately obvious during debugging.

Second, the malware queries the SQL database for any "autorun" plugins. Presumably, these are plugins transmitted by the C2 that are then executed when the malware first loads, although this is unconfirmed due the previously mentioned absence of such data.

Command and Control

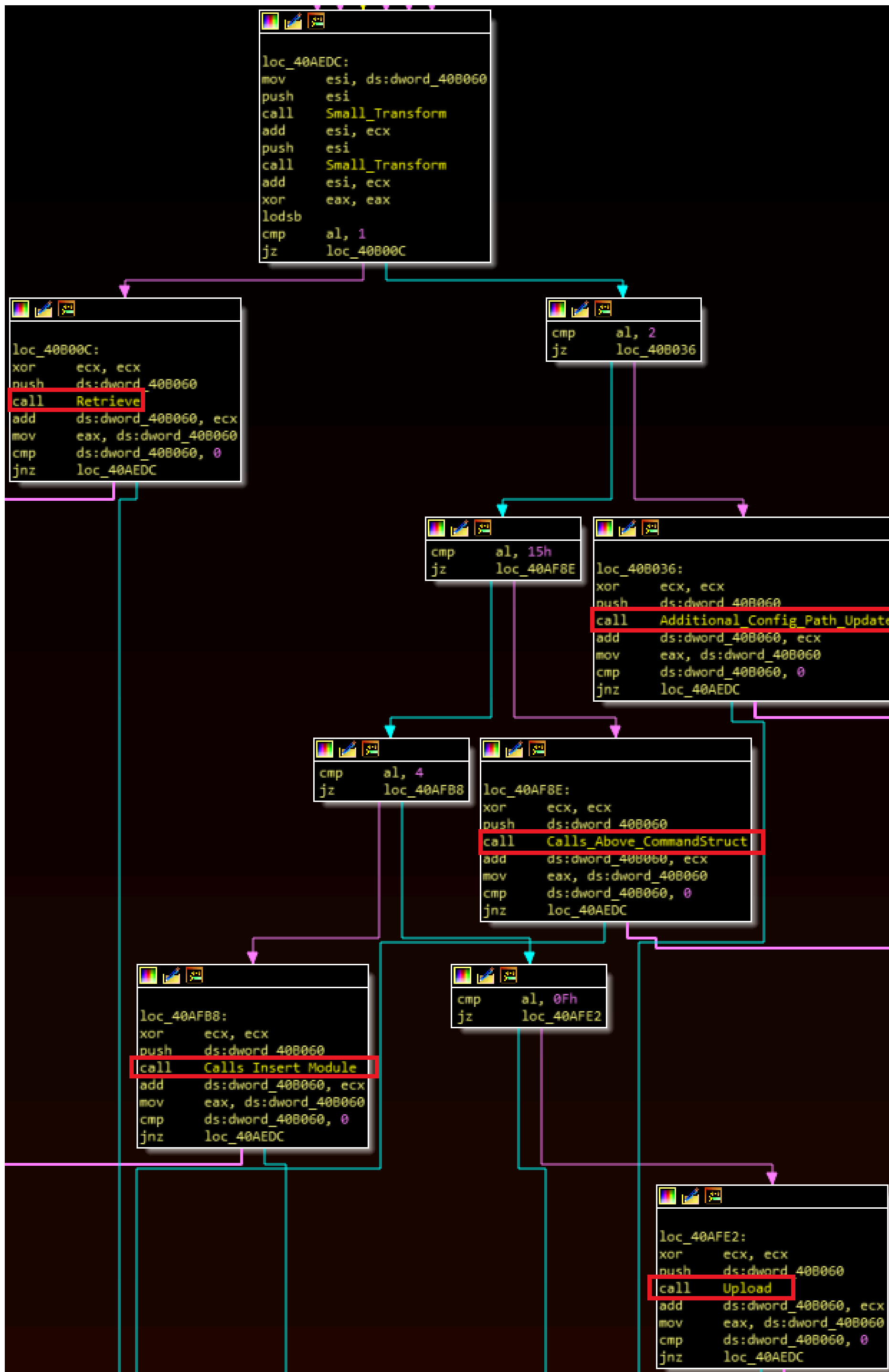
The malware has an initial C2 check-in that serves an unclear purpose; however, the primary two C2 workflows are described below.

“Higher” Workflow

This workflow is built around a series of “if-else” statements in which the ‘al’ register is compared to a hardcoded hex value:

0x01 – Retrieve an entry from the configuration table in the SQL database OR retrieve the operating system version.
0x02 – Update the configuration via SQL statement.
0x15 – Call “lower” command structure.
0x04 – Insert new data into the “modules” table via SQL statement.
0x0F – Insert new data into the “objects” table via SQL statement.

As previously mentioned, the purpose of the objects table (and thus the 0x0F) command isn’t entirely clear given the possible double meaning of the phrase “objects.” This routine includes the phrase “upload,” which for some malware can mean “upload to the victim device.” The phrase “moved to crypto container” is also present and is preceded by the “SaveToCC” export that inserts data into the objects table. These factors (and traditional C2 routines discussed below) point towards file transmission, but this remains hypothetical.



“Higher” C2 Workflow

“Lower” Workflow

When the malware enters the “lower” command structure (based on the 0x15 check above), it can execute one of a large number of possible commands. The initial intent of this research was to identify as many commands as possible. So far, these include:

– Set current working directory.

– Retrieve data from the “body” field of the “objects” table and write it to the disk.

– Copy a file from one location to another location.

– Read data from a file.
– Delete a file.
– Save data to the SQL database.
– Create a pipe, create process, open mutex (“ZWQSI_Mutex”) and open file mapping (“ZWQSI_PIDs”).
– Unknown, possibly terminates process with “ZWQSI_Mutex.”
– Retrieve current directory.
– Read data from a named pipe.
– Update a module’s “autorun” configuration value.
– Read the body of an item in the “object” table.
– Read the configuration of an item in the “module” table.
– Delete a file using the SHFileOperation API, with flags set for Silent, No Confirmation, and No Error.
– Move a file.
– Create a directory.
– Terminate a process (specified by PID).
– Terminate a process (via CreateToolhelp32Snapshot and Process32First/Next).
– Set working directory to user’s profile (“C:\Users\[User]\”).
– Set working directory to the Temp path.
– Retrieve drive and disk space information.
– List installed programs by enumerating the “DisplayName” values in the “\CurrentVersion\Uninstall” key.
– Retrieve HDD, OS, ProcessorNameString, SystemBiosVersion, net adapter (IP, MAC, DHCP, Description).
– Get values that comprise database key (BIOS, username, computername, processor).
– Retrieve entries in modules table.
– Delete object from SQL database.
– Delete module from SQL database.
– Retrieve entries in objects table.
– Return an error message if an unsupported command is issued.

Workflow for command that enumerates DisplayName values in Uninstall registry key

The number of features are extensive. Most notably, several of these commands perform the same or similar functions. For example, there are multiple supported operations for deleting a file that use different APIs. Several commands change the working directory. There are also several commands for interacting with the SQL database.

While such actions could have been streamlined, one possible explanation for these redundancies is that the authors of the tool intended to minimize user errors during an intrusion that could be caused by issuing invalid SQL commands or invalid command-line entries. From a development standpoint, however, this can become cumbersome: every possible operator action would require a new entry into the malware.

One additional note includes the “ZWQSI_Mutex” mutex. This blog has not yet identified a process associated with this specific value. It may refer to an additional executable or plugin used by the attacker; however, given that this malware has likely not been in use for five years, that data may never be recovered. Google searches for ZWQSI suggest it may be used as a common abbreviation for the “ZwQuerySystemInformation” API, and perhaps this implies the attackers tried to hide this mutex by naming it after that.

Concluding Thoughts

The LiteDuke malware is extensive, featuring a multi-stage loading process, a SQL table that holds configuration information and plugins, and a wide range of commands that allowed the operators to interact with this database and carry out common backdoor actions on the device. While this workflow has several advantages – most notably, unique but predictable database encryption keys as well as plugin support – it also requires the implementation of several redundant commands. This blog suspects that the attackers likely abandoned this in favor of their more flexible tools.

Additional hashes:

“LoadRegistry” loader: f00297add1ea84ace5677d710cd68b51 (Compiled 2014-10-02 11:09:35)

Unpacked LiteDuke payload: 8267a73a144e0f0d9ccf487958050354 (Uploaded to VT by this blog after analysis)

Source: https://norfolkinfosec.com/looking-back-at-liteduke/