LOWKEY: Hunting for the Missing Volume Serial ID

fireeye.com/blog/threat-research/2019/10/lowkey-hunting-for-the-missing-volume-serial-id.html

In August 2019, FireEye released the “Double Dragon” report on our newest graduated threat group: APT41. A China-nexus dual espionage and financially-focused group, APT41 targets industries such as gaming, healthcare, high-tech, higher education, telecommunications, and travel services.

This blog post is about the sophisticated passive backdoor we track as LOWKEY, mentioned in the APT41 report and recently unveiled at the FireEye Cyber Defense Summit. We observed LOWKEY being used in highly targeted attacks, utilizing payloads that run only on specific systems. Additional malware family names are used in the blog post and briefly described. For a complete overview of malware used by APT41 please refer to the Technical Annex section of our APT41 report.

The blog post is split into three parts, which are shown in Figure 1. The first describes how we managed to analyze the encrypted payloads. The second part features position independent loaders we observed in multiple samples, adding an additional layer of obfuscation. The final part is about the actual backdoor we call LOWKEY, which comes in two variants, a passive TCP listener and a passive HTTP listener targeting Internet Information Services (IIS).

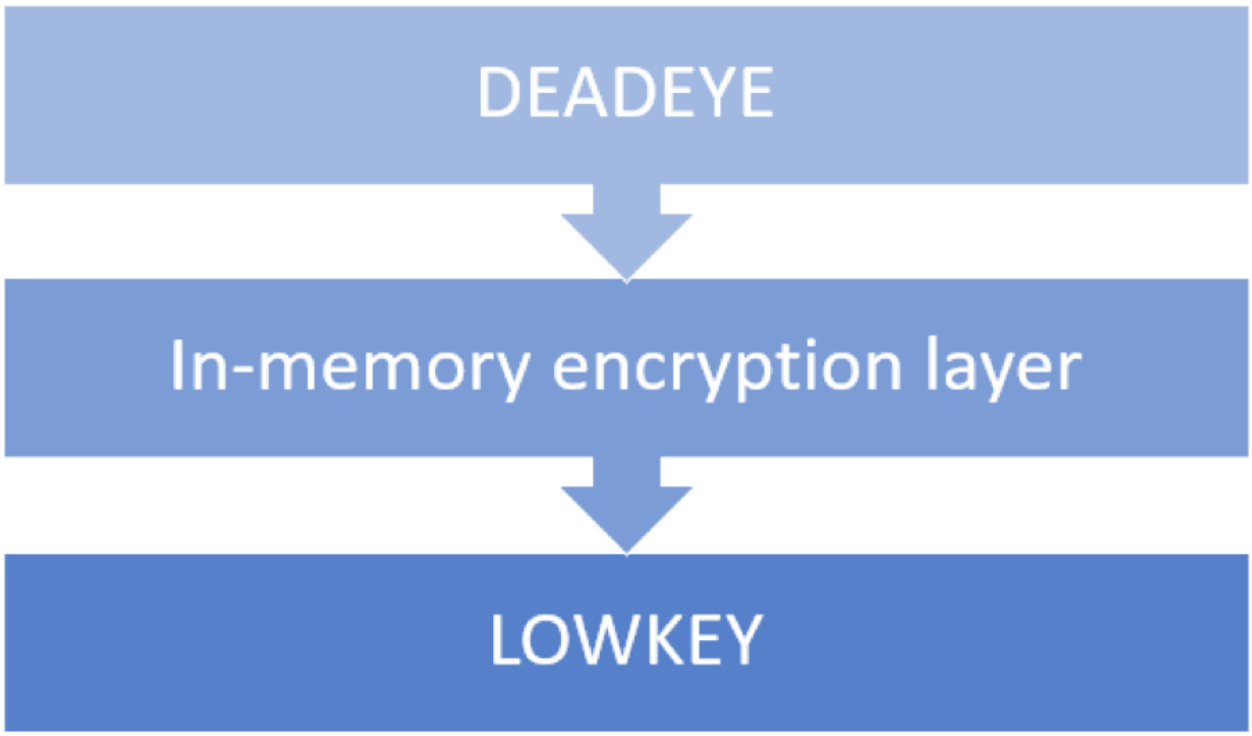

Figure 1: Blog post overview

DEADEYE – RC5

Tracking APT41 activities over the past months, we observed multiple samples that shared two unique features: the use of RC5 encryption which we don't encounter often, and a unique string "f@Ukd!rCto R$.". We track these samples as DEADEYE.

DEADEYE comes in multiple variants:

- DEADEYE.DOWN has the capability to download additional payloads.
- DEADEYE.APPEND has additional payloads appended to it.
- DEADEYE.EXT loads payloads that are already present on the system.

DEADEYE.DOWN

A sample belonging to DEADEYE.DOWN (MD5: **5322816c2567198ad3dfc53d99567d6e**) attempts to download two files on the first execution of the malware.

The first file is downloaded from hxxp://checkin.travelsanignacio[.]com/static/20170730.jpg. The command and control (C2) server response is first RC5 decrypted with the key "wsprintfA" and then RC5 encrypted with a different key and written to disk as <MODULE_NAME>.mui.

The RC5 key is constructed using the volume serial number of the C drive. The volume serial number is a 4-byte value, usually based on the install time of the system. The volume serial number is XORed with the hard-coded constant "f@Ukd!rCto R$." and then converted to hex to derive a key of up to 28 bytes in length. The key length can vary if the XORed value contains an embedded zero byte because the lstrlenA API call is used to determine the length of it. Note that the lstrlenA API call happens before the result is converted to hex. If the index of the byte modulo 4 is zero, the hex conversion is in uppercase. The key derivation is illustrated in Table 1.

Volume Serial number of C drive, for example 0xAABBCCDD					
F	^	0xAA	=	0xCC	uppercase
@	^	0xBB	=	0xFB	lowercase
U	^	0xCC	=	0x99	lowercase
k	^	0xDD	=	0xB6	lowercase
d	^	0xAA	=	0xCE	uppercase
!	^	0xBB	=	0x9A	lowercase
r	^	0xCC	=	0xBE	lowercase
C	^	0xDD	=	0x9E	lowercase
t	^	0xAA	=	0xDE	uppercase

o	^	0xBB	=	0xD4	lowercase
(0x20)	^	0xCC	=	0xEC	lowercase
R	^	0xDD	=	0x8F	lowercase
$	^	0xAA	=	0x8E	uppercase
.	^	0xBB	=	0x95	lowercase
Derived key CCfb99b6CE9abe9eDEd4ec8f8E95					

Table 1: Key derivation example

The second file is downloaded from hxxp://checkin.travelsanignacio[.]com/static/20160204.jpg. The C2 response is RC5 decrypted with the key “wsprintfA” and then XORed with 0x74, before it is saved as C:\Windows\System32\wcnapi.mui.

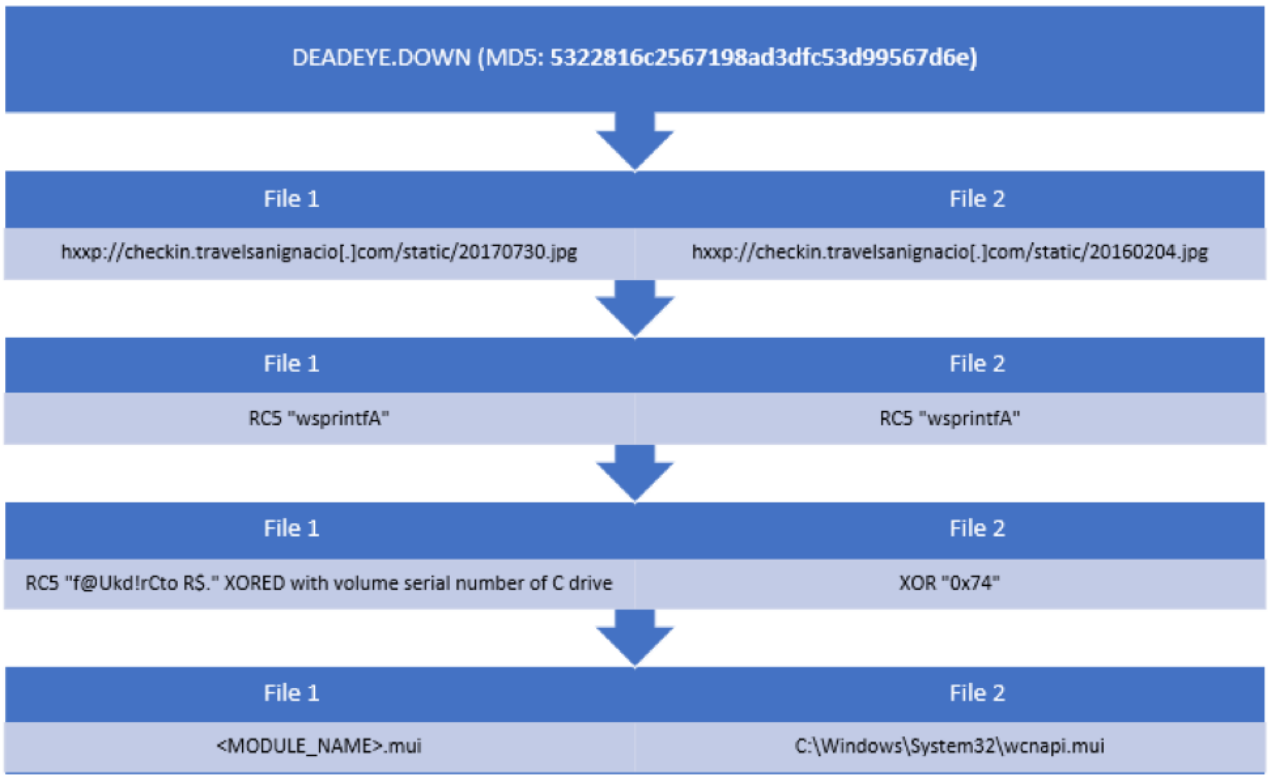

Figure 2: 5322816c2567198ad3dfc53d99567d6e download

The sample then determines its own module name, appends the extension mui to it and attempts to decrypt the file using RC5 encryption. This effectively decrypts the file the malware just downloaded and stored encrypted on the system previously. As the file has been encrypted with a key based on the volume serial number it can only be executed on the system it was downloaded on or a system that has the same volume serial number, which would be a remarkable coincidence.

An example mui file is the MD5 hash **e58d4072c56a5dd3cc5cf768b8f37e5e**. Looking at the encrypted file in a hex editor reveals a high entropy (7.999779/8). RC5 uses Electronic Code Book (ECB) mode by default. ECB means that each code block (64 bit) is encrypted independent from other code blocks. This means the same plaintext block always results in the same cipher text, independent from its position in the binary. The file has 792933 bytes in total but almost no duplicate cipher blocks, which means the data likely has an additional layer of encryption.

Without the correct volume serial number nor any knowledge about the plaintext there is no efficient way to decrypt the payload **e58d4072c56a5dd3cc5cf768b8f37e5e** with just the knowledge of the current sample.

DEADEYE.APPEND

Fortunately searching for the unique string “f@Ukd!rCto R$.“ in combination with artifacts from RC5 reveals additional samples. One of the related samples is DEADEYE.APPEND (MD5: **37e100dd8b2ad8b301b130c2bca3f1ea**), which has been previously analyzed by Kaspersky (https://securelist.com/operation-shadowhammer-a-high-profile-supply-chain-attack/90380/). This sample is different because it is protected with VMProtect and has the obfuscated binary appended to it. The embedded binary starts at offset 3287552 which can be seen in Figure 3 with the differing File Size and PE Size.

37e100dd8b2ad8b301b130c2bca3f

Property	Value
File Name	C:\Users[redacted]Desktop\37e100dd8b2ad8b301b130c2bca3f1ea
File Type	Portable Executable 64
File Info	No match found.
File Size	3.23 MB (3383519 bytes)
PE Size	3.14 MB (3287552 bytes)
MD5	37E100DD8B2AD8B301B130C2BCA3F1EA
SHA-1	32466D8D232D7B1801F456FE336615E6FA5E6FFB

Property	Value
CompanyName	Microsoft Corporation
FileDescription	Remote Desktop Services
FileVersion	10.0.14393.0 (rs1_release.160715-1616)
InternalName	TSMSISrv.DLL
LegalCopyright	© Microsoft Corporation. All rights reserved.
OriginalFilename	TSMSISrv.DLL
ProductName	Microsoft® Windows® Operating System

Figure 3: A look at the PE header reveals a larger file size than PE size

The encrypted payload has a slightly lower entropy of 7.990713 out of 8. Looking at the embedded binary in a hex editor reveals multiple occurrences of the byte pattern 51 36 94 A4 26 5B 0F 19, as seen in Figure 4. As this pattern occurs multiple times in a row in the middle of the encrypted data and ECB mode is being used, an educated guess is that the plaintext is supposed to be 00 00 00 00 00 00 00 00.

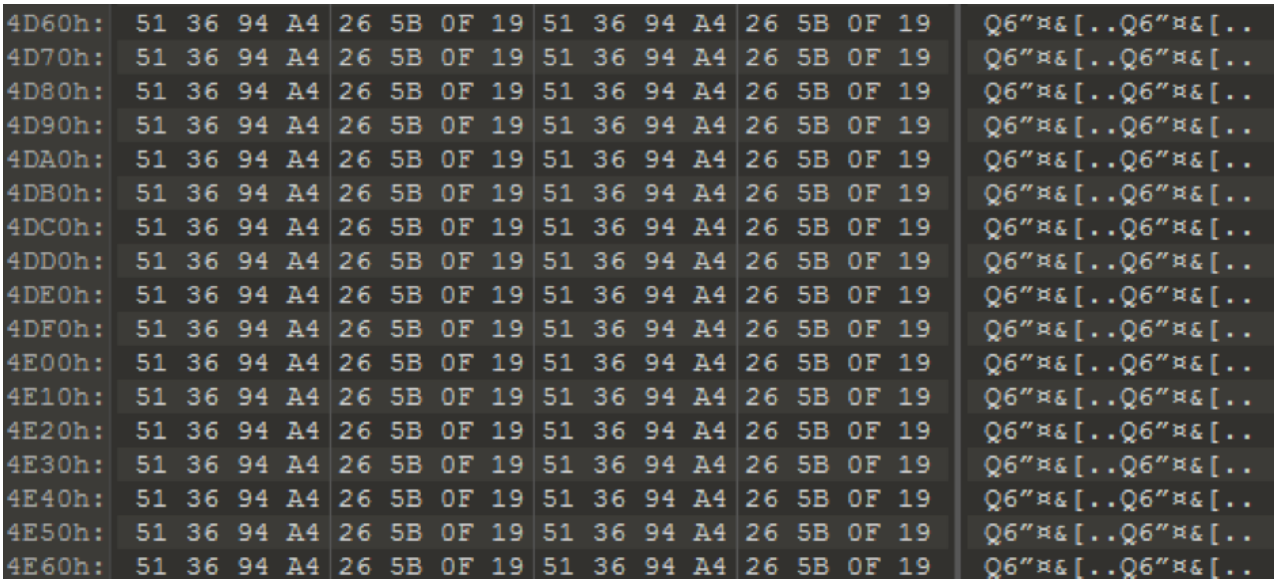

Figure 4: Repeating byte pattern in 37e100dd8b2ad8b301b130c2bca3f1ea

RC5 Brute Forcer

With this knowledge we decided to take a reference implementation of RC5 and add a main function that accounts for the key derivation algorithm used by the malware samples (see Figure 5). Brute forcing is possible as the key is derived from a single DWORD; even though the final key length might be 28 bytes, there are only 4294967296 possible keys. The code shown in Figure 5 generates all possible volume serial numbers, derives the key from them and tries to decrypt 51 36 94 A4 26 5B 0F 19 to 00 00 00 00 00 00 00 00. Running the RC5 brute forcer for a couple of minutes shows the correct volume serial number for the sample, which is 0xc25cff4c.

Note if you want to run the brute forcer yourself
The number of DWORDs of the key in the reference implementation we used is represented by the global c, and we had to change it to 7 to match the malware's key length of 28 bytes. There were some issues with the conversion because in the malware a zero byte within the generated key ultimately leads to a shorter key length. The implementation we used uses a hard-coded key length (c), so we generated multiple executables with c = 6, c = 5, c = 4… as these usually only ran for a couple of minutes to cover the entire key space. All the samples mentioned in the Appendix 1 could be solved with c = 7 or c = 6.

```
int main(int argc, char* argv[])
{
  char seed[4] = { 0x4c,0xff,0x5c,0xc2 };
  WORD dat[2], dec[2];
  *dat = 0xA4943651;//encrypted bytes offset 0x327760 in 37e100dd8b2ad8b301b130c2bca3f1ea
  *(dat + 1) = 0x190F5B26;
  char uppercase[] = "0123456789ABCDEF";
  char lowercase[] = "0123456789abcdef";
  char temp;
  char key2[] = "f@Ukd!rCto R$.";
  unsigned char key[28] = {};
  for (int i0 = 0; i0 < 256; i0++) {
    for (int i1 = 0; i1 < 256; i1++) {
      for (int i2 = 0; i2 < 256; i2++) {
        for (int i3 = 0; i3 < 256; i3++) {
          seed[0] = i0;
          seed[1] = i1;
          seed[2] = i2;
          seed[3] = i3;
          for (int i = 0; i < sizeof(key2); i++) {
            temp = key2[i] ^ seed[i % 4];
            if (temp == 0) {
              while (i < sizeof(key2)-1) {
                key[2 * i + 1] = 0;
                key[2 * i] = 0;
                i += 1;
              }
              break;
            }
            if (i % 4) {
              key[2 * i + 1] = lowercase[temp & 0xF];
              key[2 * i] = lowercase[(temp >> 4) & 0xF];
            }
            else {
              key[2 * i + 1] = uppercase[temp & 0xF];
              key[2 * i] = uppercase[(temp >> 4) & 0xF];
            }
          }
          RC5_SETUP(key);
          RC5_DECRYPT(dat, dec);
          if (dec[0] == 0 and dec[1] == 0)
            printf("found volume serial: 0x%02x%02x%02x%02x\n", i3, i2, i1, i0);
        }
      }
    }
  }
  return 0;
}
```

Figure 5: Main function RC5 brute forcer

The decrypted payload belongs to the malware family POISONPLUG (MD5: **84b69b5d14ba5c5c9258370c9505438f**). POISONPLUG is a highly obfuscated modular backdoor with plug-in capabilities. The malware is capable of registry or service persistence, self-removal, plug-in execution, and network connection forwarding. POISONPLUG has been observed using social platforms to host encoded command and control commands.

We confirmed the findings from Kaspersky and additionally found a second command and control URL hxxps://steamcommunity[.]com/id/oswal053, as mentioned in our APT 41 report.

Taking everything into account that we learned from DEADEYE.APPEND (MD5: **37e100dd8b2ad8b301b130c2bca3f1ea**), we decided to take another look at the encrypted mui file (**e58d4072c56a5dd3cc5cf768b8f37e5e**). Attempts to brute force the first bytes to match with the ones of the decrypted POISONPLUG payload did not yield any results.

Fortunately, we found additional samples that use the same encryption scheme. In one of the samples the malware authors included two checks to validate the decrypted payload. The expected plaintext at the specified offsets for DEADEYE.APPEND (MD5: **7f05d410dc0d1b0e7a3fcc6cdda7a2ff**) is shown in Table 2.

Offset	Expected byte after decryption
0	0x48
1	0x8B
0x3C0	0x48
0x3C1	0x83

Table 2: Byte comparisons after decrypting in DEADEYE.APPEND (MD5: **7f05d410dc0d1b0e7a3fcc6cdda7a2ff**)

Applying these constraints to our brute forcer and trying to decrypt mui file (**e58d4072c56a5dd3cc5cf768b8f37e5e**) once moreresulted in a low number of successful hits which we could then manually check. The correct volume serial number for the encrypted mui is 0x243e2562. Analysis determined the decrypted file is XMRig miner. This also explains why the dropper downloads two files. The first, <MODULE_NAME>.mui is the crypto miner, and the second C:\Windows\System32\wcnapi.mui, is the configuration. The decrypted mui contains another layer of obfuscation and is eventually executed with the command x -c wcnapi.mui. An explanation on how the command was obtained and the additional layer of obfuscation is given in the next part of the blog post.

For a list of samples with the corresponding volume serial numbers, please refer to Appendix 1.

Additional RC4 Layer

An additional RC4 layer has been identified in droppers used by APT41, which we internally track as DEADEYE. The layer has been previously detailed in a blog post by ESET. We wanted to provide some additional information on this, as it was used in some of the samples we managed to brute force.

The additional layer is position independent shellcode containing a reflective DLL loader. The loader decrypts an RC4 encrypted payload and loads it in memory. The code itself is a straight forward loader with the exception of some interesting artifacts identified during analysis.

As mentioned in the blog post by ESET, the encrypted payload is prepended with a header. It contains the RC4 encryption key and two fields of variable length, which have previously been identified as file names. These two fields are encrypted with the same RC4 encryption key that is also used to decrypt the payload. The header is shown in Table 3.

Header bytes	**Meaning**
0 – 15	RC4 key XOR encoded with 0x37
16 – 19	Size of loader stub before the header
20 – 23	RC4 key size
24 – 27	Command ASCII size (CAS)
28 – 31	Command UNICODE size (CUS)
32 – 35	Size of encrypted payload
36 – 39	Launch type
40 – (40 + CAS)	Command ASCII
(40 + CAS) – (40 + CAS + CUS)	Command UNICODE
(40 + CAS + CUS) – (40 + CAS + CUS + size of encrypted payload)	Encrypted payload

Table 3: RC4 header overview

Looking at the payloads hidden behind the RC5 layer, we observed, that these fields are not limited to file names, instead they can also contain commands used by the reflective loader. If no command is specified, the default parameter is the file name of the loaded payload. In some instances, this revealed the full file path and file name in the development environment. Table 4 shows some paths and file names. This is also how we found the command (x -c wcnapi.mui) used to launch the decrypted mui file from the first part of the blog post.

MD5 hash	**Arguments found in the RC4 layer**
7f05d410dc0d1b0e7a3fcc6cd-da7a2ff	E:\code\PortReuse\3389-share\DeviceIOContrl-Hook\v1.3-53\Inner-Loader\x64\Release\Inner-Loader.dll
7f05d410dc0d1b0e7a3fcc6cd-da7a2ff	E:\code\PortReuse\3389-share\DeviceIOContrl-Hook\v1.3-53\NetAgent\x64\Release\NetAgent.exe

7f05d410dc0d1b0e7a3fcc6cd-da7a2ff	E:\code\PortReuse\3389-share\DeviceIOContrl-Hook\v1.3-53\SK3.x\x64\Release\SK3.x.exe
7f05d410dc0d1b0e7a3fcc6cd-da7a2ff	UserFunction.dll
7f05d410dc0d1b0e7a3fcc6cd-da7a2ff	ProcTran.dll
c11d-d805de683822bf4922aecb9bfef5	E:\code\PortReuse\iis-share\2.5\IIS_Share\x64\Release\IIS_Share.dll
c11d-d805de683822bf4922aecb9bfef5	UserFunction.dll
c11d-d805de683822bf4922aecb9bfef5	ProcTran.dll

Table 4: Decrypted paths and file names

LOWKEY

The final part of the blog post describes the capabilities of the passive backdoor LOWKEY (MD5: **8aab5e2834feb68bb645e0bad4fa10bd**) decrypted from DEADEYE.APPEND (MD5: **7f05d410dc0d1b0e7a3fcc6cdda7a2ff**). LOWKEY is a passive backdoor that supports commands for a reverse shell, uploading and downloading files, listing and killing processes and file management. We have identified two variants of the LOWKEY backdoor.

The first is a TCP variant that listens on port 53, and the second is an HTTP variant that listens on TCP port 80. The HTTP variant intercepts URL requests matching the UrlPrefix http://+:80/requested.html. The + in the given UrlPrefix means that it will match any host name. It has been briefly mentioned by Kaspersky as “unknown backdoor”.

Both variants are loaded by the reflective loader described in the previous part of the blog post. This means we were able to extract the original file names. They contain meaningful names and provide a first hint on how the backdoor operates.

HTTP variant (MD5: c11dd805de683822bf4922aecb9bfef5)
E:\code\PortReuse\iis-share\2.5\IIS_Share\x64\Release\IIS_Share.dll

TCP variant (MD5: 7f05d410dc0d1b0e7a3fcc6cdda7a2ff)
E:\code\PortReuse\3389-share\DeviceIOContrl-Hook\v1.3-53\SK3.x\x64\Release\SK3.x.exe

The interesting parts are shown in Figure 6. PortReuse describes the general idea behind the backdoor, to operate on a well-known port. The paths also contain version numbers 2.5 and v1.3-53. IIS_Share is used for the HTTP variant and describes the targeted application, DeviceIOContrl-Hook is used for the TCP variant.

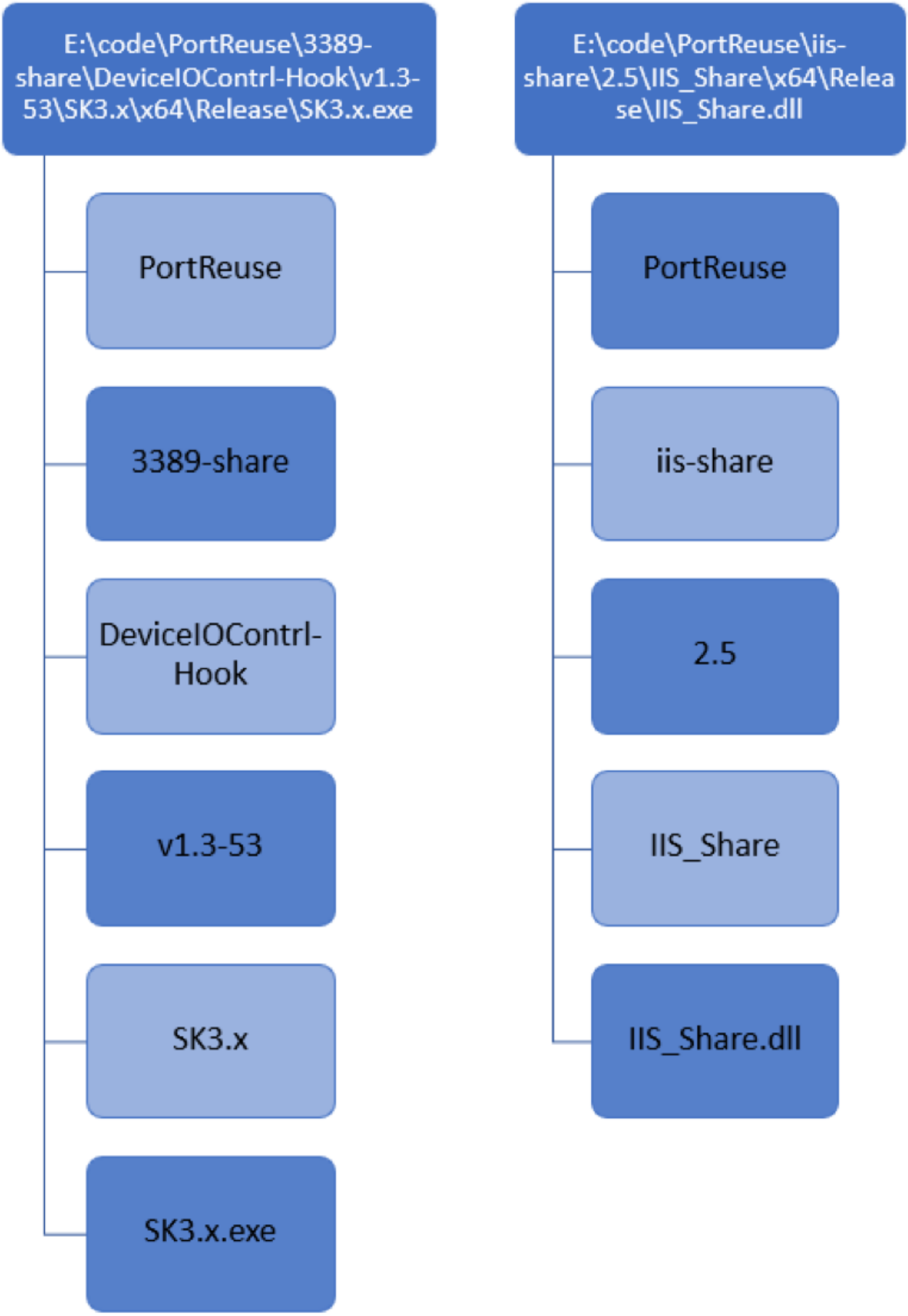

Figure 6: Overview important parts of executable path

Both LOWKEY variants are functionally identical with one exception. The TCP variant relies on a second component, a user mode rootkit that is capable of intercepting incoming TCP connections. The internal name used by the developers for that component is E:\code\PortReuse\3389-share\DeviceIOContrl-Hook\v1.3-53\NetAgent\x64\Release\NetAgent.exe.

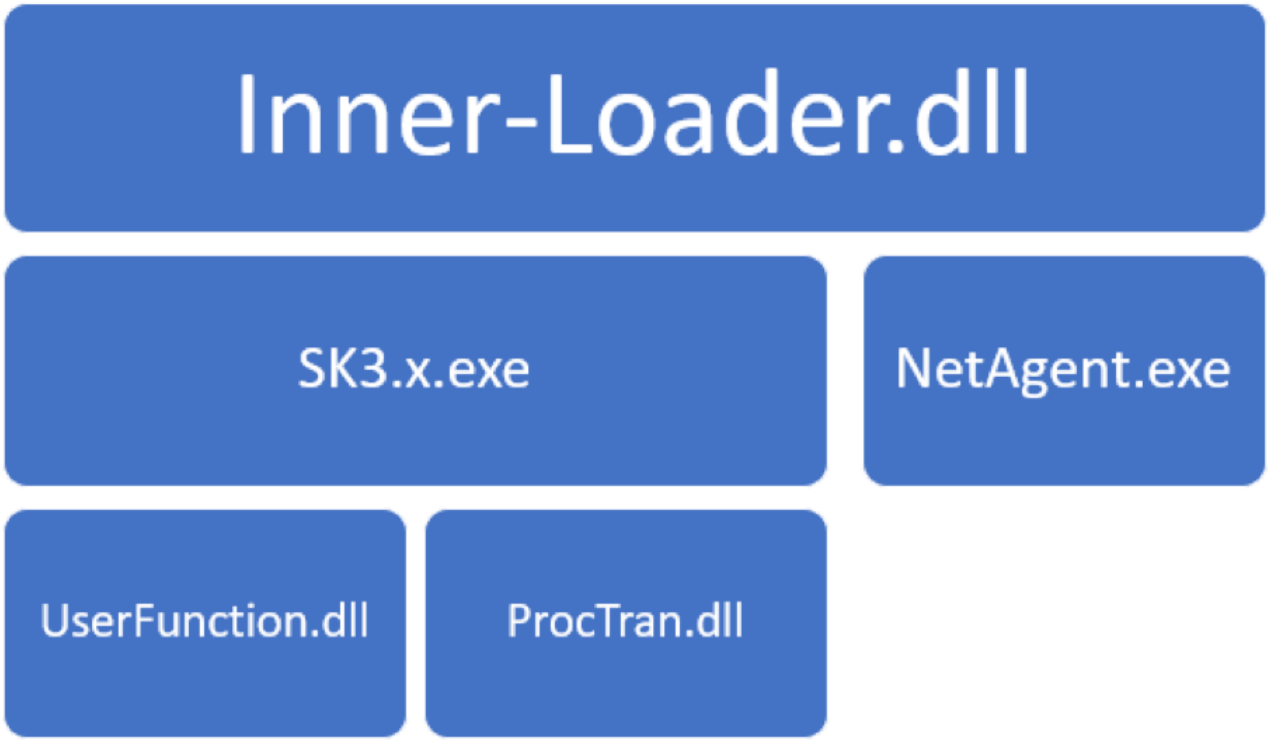

Figure 7: LOWKEY components

Inner-Loader.dll

Inner-Loader.dll is a watch guard for the LOWKEY backdoor. It leverages the GetExtendedTcpTable API to retrieve all processes with an open TCP connection. If a process is listening on TCP port 53 it injects NetAgent.exe into the process. This is done in a loop with a 10 second delay. The loader exits the loop when NetAgent.exe has been successfully injected. After the injection it will create a new thread for the LOWKEY backdoor (SK3.x.exe).

The watch guard enters an endless loop that executes every 20 minutes and ensures that the NetAgent.exe and the LOWKEY backdoor are still active. If this is not the case it will relaunch the backdoor or reinject the NetAgent.exe.

NetAgent.exe

NetAgent.exe is a user mode rootkit that provides covert communication with the LOWKEY backdoor component. It forwards incoming packets, after receiving the byte sequence FF FF 01 00 00 01 00 00 00 00 00 00, to the named pipe \\.\pipe\Microsoft Ole Object {30000-7100-12985-00001-00001}.

The component works by hooking the NtDeviceIoControlFile API. It does that by allocating a suspiciously large region of memory, which is used as a global hook table. The table consists of 0x668A0 bytes and has read, write and execute permissions set.

Each hook table entry consists of 3 pointers. The first points to memory containing the original 11 bytes of each hooked function, the second entry contains a pointer to the remaining original instructions and the third pointer points to the function hook. The malware will only hook one function in this manner and therefore allocates an unnecessary large amount of memory. The malware was designed to hook up to 10000 functions this way.

The hook function begins by iterating the global hook table and compares the pointer to the hook function to itself. This is done to find the original instructions for the installed hook, in this case NtDeviceIoControlFile. The malware then executes the saved instructions which results in a regular NtDeviceIoControlFile API call. Afterwards the IoControlCode is compared to 0x12017 (AFD_RECV).

If the IoControlCode does not match, the original API call results are returned.

If they match the malware compares the first 12 bytes of the incoming data. As it is effectively a TCP packet, it is parsing the TCP header to get the data of the packet. The first 12 bytes of the data section are compared against the hard-coded byte pattern: FF FF 01 00 00 01 00 00 00 00 00 00.

If they match it expects to receive additional data, which seems to be unused, and then responds with a 16 byte header 00 00 00 00 00 91 03 00 00 00 00 00 80 1F 00 00, which seems to be hard-coded and to indicate that following packets will be forwarded to the named pipe \\.\pipe\Microsoft Ole Object {30000-7100-12985-00001-00001}. The backdoor component (SK3.x.exe) receives and sends data to the named pipe. The hook function will forward all received data from the named pipe back to the socket, effectively allowing a covert communication between the named pipe and the socket.

SK3.x.exe

SK3.x.exe is the actual backdoor component. It supports writing and reading files, modification of file properties, an interactive command shell, TCP relay functionality and listing running processes. The backdoor opens a named pipe \\.\pipe\Microsoft Ole Object {30000-7100-12985-00001-00001} for communication.

Data received by the backdoor is encrypted with RC4 using the key “CreateThread“ and then XORed with 0x77. All data sent by the backdoor uses the same encryption in reverse order (first XOR with 0x77, then RC4 encrypted with the key “CreateThread“). Encrypted data is preceded by a 16-byte header which is unencrypted containing an identifier and the size of the following encrypted packet.

An example header looks as follows:
00 00 00 00 00 FD 00 00 10 00 00 00 00 00 00 00

Bytes	Meaning
00 00 00 00 00	unknown

FD 00	Bytes 5 and 6 are the command identifier, for a list of all supported identifiers check Table 6 and Table 7.
00	unknown
10 00 00 00	Size of the encrypted packet that is send after the header
00 00 00 00	unknown

Table 5: Subcomponents of header

The backdoor supports the commands listed in tables Table 6 and Table 7. Most commands expect a string at the beginning which likely describes the command and is for the convenience of the operators, but this string isn't actively used by the malware and could be anything. For example, KILL <PID> could also be A <PID>. Some of the commands rely on two payloads (UserFunction.dll and ProcTran.dll), that are embedded in the backdoor and are either injected into another process or launch another process.

UserFunction.dll

Userfunction.dll starts a hidden cmd.exe process, creates the named pipe \\.\pipe\Microsoft Ole Object {30000-7100-12985-00000-00000} and forwards all received data from the pipe to the standard input of the cmd.exe process. All output from the process is redirected back to the named pipe. This allows interaction with the shell over the named pipe.

ProcTran.dll

The component opens a TCP connection to the provided host and port, creates the named pipe \\.\pipe\Microsoft Ole Object {30000-7100-12985-00000-00001} and forwards all received data from the pipe to the opened TCP connection. All received packets on the connection are forwarded to the named pipe. This allows interaction with the TCP connection over the named pipe.

Identifier	**Arguments**	**Description**
0xC8	<**cmd**> <arg1> <arg2>	Provides a simple shell, that supports the following commands, dir, copy, move, del, systeminfo and cd. These match the functionality of standard commands from a shell. This is the only case where the <cmd> is actually used.
0xC9	<cmd> <arg1>	The argument is interpreted as a process id (PID). The backdoor injects UserFunction.dll into the process, which is an interactive shell that forwards all input and output data to Microsoft Ole Object {30000-7100-12985-00000-00000}. The backdoor will then forward incoming data to the named pipe allowing for communication with the opened shell. If no PID is provided, the `cmd.exe` is launched as child process of the backdoor process with input and output redirected to the named pipe Microsoft Ole Object {30000-7100-12985-00001-00001}

0xCA	<cmd> <arg1> <arg2>	Writes data to a file. The first argument is the <file_name>, the second argument is an offset into the file
0xCB	<cmd> <arg1> <arg2> <arg3>	Reads data from a file. The first argument is the <file_name>, the second argument is an offset into the file, the third argument is optional, and the exact purpose is unknown
0xFA	-	Lists running processes, including the process name, PID, process owner and the executable path
0xFB	<cmd> <arg1>	Kills the process with the provided process id (PID)
0xFC	<cmd> <arg1> <arg2>	Copies the files CreationTime, LastAccessTime and LastWriteTime from the second argument and applies them to the first argument. Both arguments are expected to be full file paths. The order of the arguments is a bit unusual, as one would usually apply the access times from the second argument to the third
0xFD	-	List running processes with additional details like the SessonId and the CommandLine by executing the WMI query SELECT Name,ProcessId,SessionId,CommandLine,ExecutablePath FROM Win32_Process
0xFE	-	Ping command, the malware responds with the following byte sequence 00 00 00 00 00 65 00 00 00 00 00 00 06 00 00 00. Experiments with the backdoor revealed that the identifier 0x65 seems to indicate a successful operation, whereas 0x66 indicates an error.

Table 6: C2 commands

The commands listed in Table 7 are used to provide functionality of a TCP traffic relay. This allows operators to communicate through the backdoor with another host via TCP. This could be used for lateral movement in a network. For example, one instance of the backdoor could be used as a jump host and additional hosts in the target network could be reached via the TCP traffic relay. Note that the commands 0xD2, 0xD3 and 0xD6 listed in Table 7 can be used in the main backdoor thread, without having to use the ProcTran.dll.

Identifier	**Arguments**	**Description**
0x105	<cmd> <arg1>	The argument is interpreted as a process id (PID). The backdoor injects ProcTran.dll into the process, which is a TCP traffic relay component that forwards all input and output data to Microsoft Ole Object {30000-7100-12985-00000-00001}. The commands 0xD2, 0xD3 and 0xD6 can then be used with the component.
0xD2	<arg1> <arg2>	Opens a connection to the provided host and port, the first argument is the host, the second the port. On success a header with the identifier set to 0xD4 is returned (00 00 00 00 00 D4 00 00 00 00 00 00 00 00 00 00). This effectively establishes a TCP traffic relay allowing operators to communicate with another system through the backdoored machine.

0xD3	<arg1>	Receives and sends data over the connection opened by the 0xD2 command. Received data is first RC4 decrypted with the key “CreateThread“ and then single-byte XOR decoded with 0x77. Data sent back is directly relayed without any additional encryption.
0xD6	-	Closes the socket connection that had been established by the 0xD2 command
0xCF	-	Closes the named pipe Microsoft Ole Object {30000-7100-12985-00000-00001} that is used to communicate with the injected ProcTran.dll. This seems to terminate the thread in the targeted process by the 0x105 command

Table 7: C2 commands TCP relay

Summary

The TCP LOWKEY variant passively listens for the byte sequence FF FF 01 00 00 01 00 00 00 00 00 00 on TCP port 53 to be activated. The backdoor then uses up to three named pipes for communication. One pipe is used for the main communication of the backdoor, the other ones are used on demand for the embedded payloads.

- \\.\pipe\Microsoft Ole Object {30000-7100-12985-00001-00001} main communication pipe
- \\.\pipe\Microsoft Ole Object {30000-7100-12985-00000-00001} named pipe used for interaction with the TCP relay module ProcTran.dll
- \\.\pipe\Microsoft Ole Object {30000-7100-12985-00000-00000} named pipe used for the interactive shell module UserFunction.dll

Figure 8 summarizes how the LOWKEY components interact with each other.

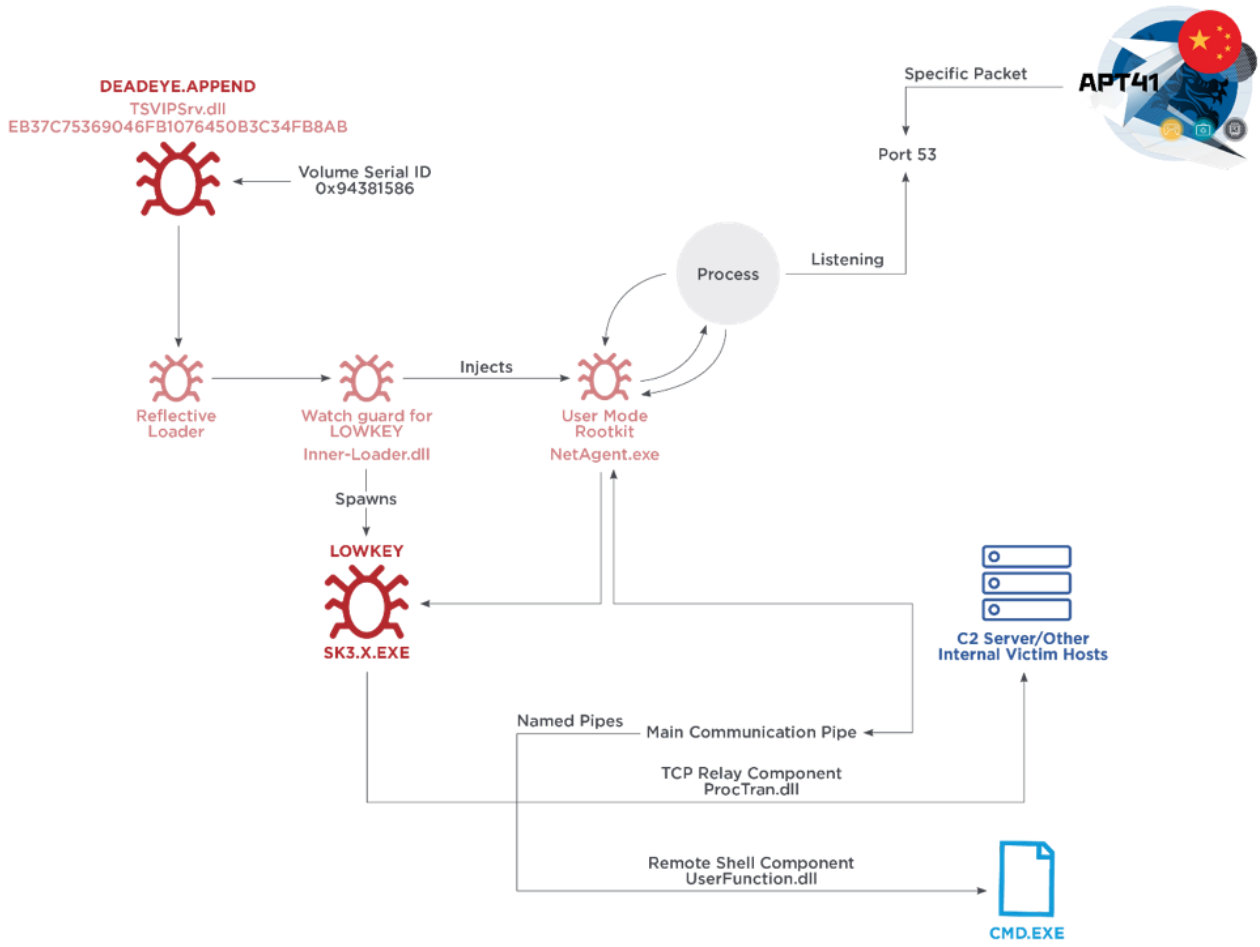

Figure 8: LOWKEY passive backdoor overview

Appendix

MD5 HASH	Correct Volume Serial	Dropper Family	Final Payload Family
2b9244c526e2c2b6d40e79a8c3ed-b93c	0xde82ce06	DEADEYE.AP-PEND	POISONPLUG
04be89f-f5d217796bc68678d2508a0d7	0x56a80cc8	DEADEYE.AP-PEND	POISONPLUG
092ae9ce61f6575344c424967b-d79437	0x58b5ef5c	DEADEYE.AP-PEND	LOWKEY.HTTP
37e100dd8b2ad8b301b130c2b-ca3f1ea	0xc25cff4c	DEADEYE.AP-PEND	POISONPLUG
39fe65a46c03b930c-cf0d552ed3c17b1	0x24773b24	DEADEYE.AP-PEND	POISONPLUG

5322816c2567198ad3d-fc53d99567d6e	-	DEADEYE.DOWN	-
557ff68798c71652d-b8a85596a4bab72	0x4cebb6e9	DEADEYE.AP-PEND	POISONPLUG
64e09cf2894d6e5ac50207edf-f787ed7	0x64fd8753	DEADEYE.AP-PEND	POISONPLUG
650a3dce1380f9194361e0c7be9ff-b97	0xeaa61f82	DEADEYE.AP-PEND	POISONPLUG
7dc6bbc202e039d-d989e1e2a93d2ec2d	0xa8c5a006	DEADEYE.AP-PEND	LOWKEY
7f05d410dc0d1b0e7a3fcc6cdda7a2ff	0x9438158b	DEADEYE.AP-PEND	LOWKEY
904bbe5ac0d53e74a6cefb14eb-d58c0b	0xde82ce06	DEADEYE.AP-PEND	POISONPLUG
c11dd805de683822bf4922aecb9bfe-f5	0xcab011e1	DEADEYE.AP-PEND	LOWKEY.HTTP
d49c186b1bfd7c9233e5815c2572e-b98	0x4a23bd79	DEADEYE.AP-PEND	LOWKEY
e58d4072c56a5dd3c-c5cf768b8f37e5e	0x243e2562	None - encrypted data	XMRIG
eb37c75369046fb1076450b3c34f-b8ab	0x00e5a39e	DEADEYE.AP-PEND	LOWKEY
ee5b707249c562d-c916b125e32950c8d	0xdecb3d5d	DEADEYE.AP-PEND	POISONPLUG
ff8d92dfbcda572ef97c142017eec658	0xde82ce06	DEADEYE.AP-PEND	POISONPLUG
ffd0f34739c1568797891b9961111464	0xde82ce06	DEADEYE.AP-PEND	POISONPLUG

Appendix 1: List of samples with RC5 encrypted payloads