Duqu 2.0:

A comparison to Duqu

v1.0 (10/Jun/2015)

Technical Report

by

CrySyS Lab

http://www.crysys.hu/

Budapest, 2015

Authors:

Boldizsár Bencsáth, Gábor Ács-Kurucz, Gábor Molnár, Gábor Vaspöri, Levente Buttyán, Roland Kamarás

Findings in brief

In October 2011, we analyzed a new threat what we named Duqu, and we showed that it has close relationships to the infamous Stuxnet attack.

By courtesy of Kaspersky Lab, in late May 2015 we received samples about a new threat, with the hint that it might be related to the Duqu attacks; however, these new samples are from 2014. We decided to carry out an individual research on the samples with the focus on the connections between the original Duqu attack and the new threat, dubbed “Duqu 2.0”.

After analyzing the samples received, we think, that the adversaries behind Duqu malware are back and active; while they modified their tools to be undetected by old methods, they also strongly reused codes and ideas during their recent attacks. The numerous similarities that we discovered between Duqu and Duqu 2.0 include the following:

- Similar string decryption routines related to Anti-Virus product strings
- Similar methods, magic number, bug and file format related to files encrypted with AES by both threats
- Same non-standard CBC mode AES encryption used by both threats
- Extremely similar logging module with exactly the same magic numbers
- Similar C++-like coding and compiling style

In this report, we present supporting details and analysis for all the similarities listed above.

Table of contents

1. Introduction 4
1.1. Hashes of the analyzed samples 6
2. Similarities and differences 7
2.1. General details 7
2.2. String decryption 9
2.3. AES encryption of the configuration file 14
2.4. Format of the (encrypted) configuration file 24
2.5. Logging functions 26
2.6. Command & Control communication 29
2.7. DLL imports 33
3. Indicators of Compromise 34
3.1. Detection based on communications 34
3.2. Yara rules to identify 36
4. Conclusion 37
5. References 37
6. Contact Information 39

1. Introduction

Stuxnet is probably the most well-known malware of our times. Its fame stems from the facts that it targeted a very specific industrial facility, namely a uranium enrichment plant in Iran, it aimed at physical destruction of uranium centrifuges, and it apparently accomplished its mission successfully. In addition to all these characteristics, IT security experts also appreciate its technical sophistication and the zero-day exploits that it used. Stuxnet was also an alarm to the developed world: it shed light on the capabilities of advanced attackers, and at the same time, on the numerous weaknesses of our computing infrastructure. Putting these two together, people started to feel hopelessly vulnerable.

Yet, unfortunately, Stuxnet is not a unique example for a highly sophisticated targeted threat, but there are numerous other pieces of malware of similar kind, including Duqu, Flame, Regin, *etc*. Among those, Duqu is particularly interesting, not only because *we* discovered it back in 2011, but because our analysis pointed out that - while Duqu's objective is different - it has very strong similarities to Stuxnet in terms of architecture, code, and methods to achieve stealthiness. Today, it is widely believed within the IT security community that Duqu was created by the same attackers who created Stuxnet.

And now we have a new member of the same family! Last month, we received interesting samples from Kaspersky Lab with a hint that they might be related to the Duqu samples of 2011; however, these new samples are from 2014. Our common understanding was that it would be interesting to figure out whether this new threat is indeed related to the old Duqu attack, and we in the CrySyS Lab should try to focus our analysis efforts on answering this question. It is important to emphasize that we did our analysis independently from Kaspersky Lab: we did not read their preliminary report and they did not share any of their findings with us (apart from the samples that we received from them).

The analysis results performed by Kaspersky Lab can be read in the following report:

```
https://securelist.com/blog/research/70504/the-mystery-of-duqu-
2-0-a-sophisticated-cyberespionage-actor-returns/
```

In this report, we present the results of our comparative analysis of the old version of Duqu and the new version, codenamed “Duqu 2.0”. We concentrate on the description of the relevant similarities and differences we have found between the two malware samples.

1.1. Hashes of the analyzed samples

In the table below, one can see the MD5 fingerprints of the two samples we have examined during our initial analysis:

Sample hashes (MD5)	Information
c7c647a14cb1b8bc141b089775130834	main module
3f52ea949f2bd98f1e6ee4ea1320e80d	main module

Table 1 – Hashes (MD5) of the samples we have analyzed

The first module will be referenced in this document with the name "c7c647", and the second with the name "3f52ea" according to the prefix of their MD5 hashes.

2. Similarities and differences

In the following chapter, we will discuss the most conspicuous similarities and differences we have found between the main modules of Duqu and Duqu 2.0.

2.1. General details

Both the two main modules of Duqu 2.0 we have analyzed (”c7c647” and ”3f52ea”) has 6 export functions which can be seen in the following figure:

Name	Address	Ordinal
_1	10008D50	1
_2	10008DF8	2
_3	10008EAD	3
_4	10008F77	4
_5	1000907C	5
_6	10009130	6
DllEntryPoint	10001AE9	

Figure 1 – Structure of the first sample (“3f52ea”) – 6 export functions

Name	Address	Ordinal
_1	10009623	1
_2	100096CB	2
_3	10009780	3
_4	1000984A	4
_5	1000994F	5
_6	10009A03	6
DllEntryPoint	10001AD9	

Figure 2 – Structure of the second sample (“c7c647”) – 6 export functions

The new sample (both versions) is one big executable file that is linked by multiple modules. The original Duqu had a main module that was divided into two sub-modules: an outside layer and an internal part. In one version, the internal part was stored in a specific compressed format, while in another version, which we investigated at a Duqu victim, it was stored in cleartext in a resource data section of the main executable. The Duqu 2.0 version we investigated is different: everything is incorporated in the main executable, but there are still visible marks showing that the malware is linked/compiled from multiple different parts, modules.

2.2. String decryption

Some of the strings in Duqu 2.0 are obfuscated by XOR-based encryption. The actual routine used is printed below:

```
.text:10012F6D                  test    ecx, ecx
.text:10012F6F                  jnz     short loc_10012F77
.text:10012F71                  xor     eax, eax
.text:10012F73                  mov     [edx], ax
.text:10012F76                  retn
.text:10012F77 ; ---------------------------------------------
.text:10012F77
.text:10012F77 loc_10012F77:            ;
.text:10012F77                  mov     eax, [ecx]
.text:10012F79                  push    esi
.text:10012F7A                  push    edi
.text:10012F7B                  mov     edi, 86F186F1h
.text:10012F80                  xor     esi, esi
.text:10012F82                  xor     eax, edi
.text:10012F84                  mov     [edx], eax
.text:10012F86                  cmp     ax, si
.text:10012F89                  jz      short loc_10012FA2
.text:10012F8B                  sub     ecx, edx
.text:10012F8D
.text:10012F8D loc_10012F8D:         ;
.text:10012F8D                  cmp     [edx+2], si
.text:10012F91                  jz      short loc_10012FA2
.text:10012F93                  add     edx, 4
.text:10012F96                  mov     eax, [ecx+edx]
.text:10012F99                  xor     eax, edi
.text:10012F9B                  mov     [edx], eax
.text:10012F9D                  cmp     ax, si
.text:10012FA0                  jnz     short loc_10012F8D
```

Sample 1 – String decryption in Duqu 2.0 (assembly view)

The decompiled version of the above assembly code can be seen in the following sample:

```
unsigned int __fastcall xor_sub_10012F6D(int encrstr, int a2)
{
  unsigned int result; // eax@2
  int v3;              // ecx@4

  if ( encrstr )
  {
    result = *(_DWORD *)encrstr ^ 0x86F186F1;
    *(_DWORD *)a2 = result;

    if ( (_WORD)result )
    {

      v3 = encrstr - a2;

      do
      {
        if ( !*(_WORD *)(a2 + 2) )
          break;

        a2 += 4;
        result = *(_DWORD *)(v3 + a2) ^ 0x86F186F1;
        *(_DWORD *)a2 = result;
      }
      while ( (_WORD)result );

    }
  }
  else
  {
    result = 0;
    *(_WORD *)a2 = 0;
  }

  return result;
}
```

The above string decryptor routine is a simple XOR decoder. It simply XORs consecutive 4-byte blocks of the encrypted string buffer, given by its pointer in the first parameter of the function, with a fixed 4-byte key (“0x86F186F1”). After the decryption of all consecutive 4-byte blocks, the actual cleartext block is stored within the next 4 bytes of the output buffer, pointed by parameter “a2”. The decrypted (cleartext) string is terminated with a “\0” character, and if the decryptor cycle reaches the end of the (cleartext) string, the cleartext string will be pointed by the address stored in output argument “a2”.

A closer look at the above C code reveals that the string decryptor routine actually has two parameters: “encrstr” and “a2”. First, the decryptor function checks if the input buffer (the pointer of the encrypted string) points to a valid memory area (i.e., it does not contain NULL value). After that, the first 4 bytes of the encrypted string buffer is XORed with the key “0x86F186F1” and the result of the XOR operation is stored in variable “result”. The first DWORD (first 4 bytes) of the output buffer a2 is then populated by this resulting value (`*(_DWORD *)a2 = result;`). Therefore, the first 4 bytes of the output buffer will contain the first 4 bytes of the cleartext string.

If the first two bytes (first WORD) of the current value stored in variable “result” contain ‘\0’ characters, the original cleartext string was an empty string and the resulting output buffer will be populated by a zero value, stored on 2 bytes. If the first half of the actual decrypted block (“result” variable) contains something else, the decryptor routine checks the second half of the block (“`if ( !*(_WORD *)(a2 + 2) )`”). If this WORD value is NULL, then decryption will be ended and the output buffer will contain only one Unicode character with two closing ’\0’ bytes.

If the first decrypted block doens’t contain zero character (generally this is the case), then the decryption cycle continues with the next 4-byte encrypted block. The pointer of the output buffer is incremeted by 4 bytes to be able to store the next cleartext block (”`a2 += 4;`”). After that, the following 4-byte block of the ”ciphertext” will be decrypted with the fixed decryption key (“0x86F186F1”). The result is then stored within the next 4 bytes of the output buffer. Now, the output buffer contains 2 blocks of the cleartext string.

The condition of the cycle checks if the decryption reached its end by checking the first half of the current decrypted block. If it did not reached the end, then the cycle continues with the decryption of the next input blocks, as described above. Before the decryption of each 4-byte ”ciphertext” block, the routine also checks the second half of the previous cleartext block to decide whether the decoded string is ended or not.

The original Duqu used a very similar string decryption routine, which we printed in the following figure below. We can see that this routine is an exact copy of the previously discussed routine (variable ”a1” is analogous to ”encrstr” argument). The only difference between the Duqu 2.0 and Duqu string decryptor routines is that the XOR keys differ (in Duqu, the key is”0xB31FB31F”).

We can also see that the decompiled code of Duqu contains the decryptor routine in a more compact manner (within a ”for” loop instead of a ”while”), but the two routines are essentially the same. For example, the two boundary checks in the Duqu 2.0 routine (”`if ( !*(_WORD *)(a2 + 2) )`” and ”`while ( (_WORD)result );`”) are analogous to the boundary check at the end of the ”for” loop in the Duqu routine (”`if ( !(_WORD)v4 || !*(_WORD *)(result + 2) )`”). Similarly, the increment operation within the head of the for loop in the Duqu sample (”`result += 4`”) is analogous to the increment operation ”`a2 += 4;`” in the Duqu 2.0 sample.

```
int __cdecl b31f_decryptor_100020E7(int a1, int a2)
{
  _DWORD *v2;       // edx@1
  int result;       // eax@2
  unsigned int v4; // edi@6

  v2 = (_DWORD *)a1;

  if ( a1 )
  {
    for ( result = a2; ; result += 4 )
    {
      v4 = *v2 ^ 0xB31FB31F;
      *(_DWORD *)result = v4;
```

```
      if ( !(_WORD)v4 || !*(_WORD *)(result + 2) )
        break;

      ++v2;
    }
  }
  else
  {
    result = 0;
    *(_WORD *)a2 = 0;
  }

  return result;
}
```

Sample 3 – String decryptor from original Duqu (from “cmi4432.pnf” file)

2.3. AES encryption of the configuration file

The analyzed main module of Duqu 2.0 and also the old Duqu sample reads configuration information from a special file. This configuration file is encrypted using the AES block cipher in CBC mode with a CTS-like (Ciphertext Stealing) encryption of the last two cleartext blocks. The format of the configuration file will be discussed in details in the next chapter.

Before the encryption of the configuration file, an AES wrapper object is created. This C++ object represents the context (parameters) of the encryption. Therefore, it also stores the initialization vector (IV) of the encryption, the key of the cipher and the data to be encrypted.

The structure of this object's class can be seen in the upper part of the next screenshot:

```
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 aeswrapper      struc ; (sizeof=0x218, mappedto_16)
00000000 iv              dd 4 dup(?)
00000010 aes             aes ?
00000214 data            dd ?                    ; offset
00000218 aeswrapper      ends
00000218
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 aes             struc ; (sizeof=0x204, mappedto_17) ; XREF: aeswrapper/r
00000000 key_schedule    dd 64 dup(?)
00000100 precomputed     dd 64 dup(?)
00000200 iteration_count dd ?
00000204 aes             ends
00000204
```

Figure 3 – Attributes of the AES wrapper class and an AES object

As we can see, the allocated memory area of an instance of the "aeswrapper" structure (class) starts with a 16 bytes (128 bits) IV value (of course, the size of the IV equals the size of an AES input block). It is followed by a 516-byte buffer (or other unused smaller attributes) which can store the encryption key of the AES cipher. Size of this encryption key can be either 128, 192 or 256 bits (16, 24 or 32 bytes). The last 4 bytes of the "aeswrapper" structure contains the pointer to the data to be encrypted.

In addition to the attributes (IV, encryption key, pointer to a data buffer), the “aeswrapper” class also contains methods. The most important methods are the “encrypt” and “initialize” functions. As the name shows, the **`initialize`** method initializes the context (parameters) of the encryption, therefore it sets the IV, key and data members of the “aeswrapper” object. The IV is generated by “hand”, but the key is prepared from an initial key using the **`prepare_key`** function. The **`encrypt`** method encrypts the data in the modified CBC-CTS-like mode. The method uses an AES encryptor function. The **`nth_block`** method of the class gives back a pointer to the n-th block of the data to be encrypted. Finally, the “aeswrapper” class uses the **`last_block`** function to perform the CTS-like encryption mechanism at the end. The function gives back a pointer not to the last partial (smaller than 16 bytes) input block, but to the last 16 bytes of the input data buffer.

The implementation of AES **`prepare_key`** and **`encrypt`** methods are presumably copied from function libraries.

The figure above shows the structures (structures of class instances) which we identified and which are related to the encryption routine and the AES initialization, and the putative attributes of these structures (classes). Using these structures, the disassembled code can be more readable.

There is another structure in addition to the “aeswrapper” class called “aes” on the screenshot above. An instance of this class represents an AES encryptor object. It has probably 3 attributes: **`key_schedule`**, **`precomputed`** and **`iteration_count`**.

In the following table, we can see the AES initialization routine (of the configuration file encryption) of the old Duqu (on the left) and the new Duqu 2.0 sample (on the right) at assembly code level. The decompiled code of the initialization function (for both malware samples) can be seen in figure **Sample 6**. The AES initialization function initializes the mentioned “aeswrapper” object, it sets the data buffer, prepares the encryption key, and finally, generates the IV based on the magic constant.

Duqu “netp” routine	Duqu 2.0 “c7c64” routine
seg000:0002EE95 sub_2EE95 proc near ; CODE XREF: sub_2D0A4+8Cp seg000:0002EE95 ; sub_2EE50+36p seg000:0002EE95 seg000:0002EE95 var_20 = byte ptr -20h seg000:0002EE95 seg000:0002EE95 push ebp seg000:0002EE96 mov ebp, esp seg000:0002EE98 sub esp, 20h seg000:0002EE9B push esi seg000:0002EE9C push edi seg000:0002EE9D mov [ebx+214h], eax seg000:0002EEA3 push 8 seg000:0002EEA5 pop ecx seg000:0002EEA6 lea eax, [ebp+var_20] seg000:0002EEA9 push eax seg000:0002EEAA lea eax, [ebx+10h] seg000:0002EEAD mov esi, 10034600h seg000:0002EEB2 lea edi, [ebp+var_20] seg000:0002EEB5 push eax seg000:0002EEB6 rep movsd seg000:0002EEB8 call AES1_sub_2F9B1 seg000:0002EEBD pop ecx seg000:0002EEBE pop ecx seg000:0002EEBF pop edi seg000:0002EEC0 xor eax, eax seg000:0002EEC2 pop esi seg000:0002EEC3 seg000:0002EEC3 loc_2EEC3: ; CODE XREF: sub_2EE95+3Dj seg000:0002EEC3 mov ecx, eax seg000:0002EEC5 xor ecx, 0DEADBABEh seg000:0002EECB mov [ebx+eax*4], ecx seg000:0002EECE inc eax seg000:0002EECF cmp eax, 4 seg000:0002EED2 jb short loc_2EEC3 seg000:0002EED4 mov eax, ebx seg000:0002EED6 leave seg000:0002EED7 retn seg000:0002EED7 sub_2EE95 endp	.text:1001551D sub_1001551D proc near ; CODE XREF: sub_10007A22+28p .text:1001551D ; sub_10007CB7+121p .text:1001551D .text:1001551D var_20 = byte ptr -20h .text:1001551D arg_0 = dword ptr 8 .text:1001551D arg_4 = dword ptr 0Ch .text:1001551D .text:1001551D push ebp .text:1001551E mov ebp, esp .text:10015520 mov eax, [ebp+arg_0] .text:10015523 lea edx, [ebp+var_20] .text:10015526 sub esp, 20h .text:10015529 push ebx .text:1001552A push esi .text:1001552B mov esi, [ebp+arg_4] .text:1001552E mov ebx, ecx .text:10015530 push edi .text:10015531 push 8 .text:10015533 pop ecx .text:10015534 mov [ebx+214h], eax .text:1001553A lea edi, [ebp+var_20] .text:1001553D rep movsd .text:1001553F push 100h .text:10015544 lea ecx, [ebx+10h] .text:10015547 call AES_1_sub_1001690A .text:1001554C pop ecx .text:1001554D xor ecx, ecx .text:1001554F .text:1001554F loc_1001554F: ; CODE XREF: sub_1001551D+40j .text:1001554F mov eax, ecx .text:10015551 xor eax, 248561EFh ; MAGIC! .text:10015556 mov [ebx+ecx*4], eax .text:10015559 inc ecx .text:1001555A cmp ecx, 4 .text:1001555D jb short loc_1001554F .text:1001555F pop edi .text:10015560 pop esi .text:10015561 mov eax, ebx .text:10015563 pop ebx .text:10015564 mov esp, ebp .text:10015566 pop ebp .text:10015567 retn 0Ch .text:10015567 sub_1001551D endp

Sample 4 – IV generation routine comparison (assembly view) – magic constants

In both cases, the highlighted part of the assembly code corresponds to the highlighted part of the initialization routines in the decompiled versions of the code, which can be seen in figure **Sample 6**. The only difference between the highlighted parts is the values of the magic constants (“0xDEADBABE” vs. “0x248561EF”) which are used for the generation of the 128-bit initialization vectors. The mentioned AES initialization routines (and also the common encryption function) will be discussed later in this section in more details.

We also reverse engineered the encryption routine used by Duqu 2.0, which is illustrated in the following block diagram:

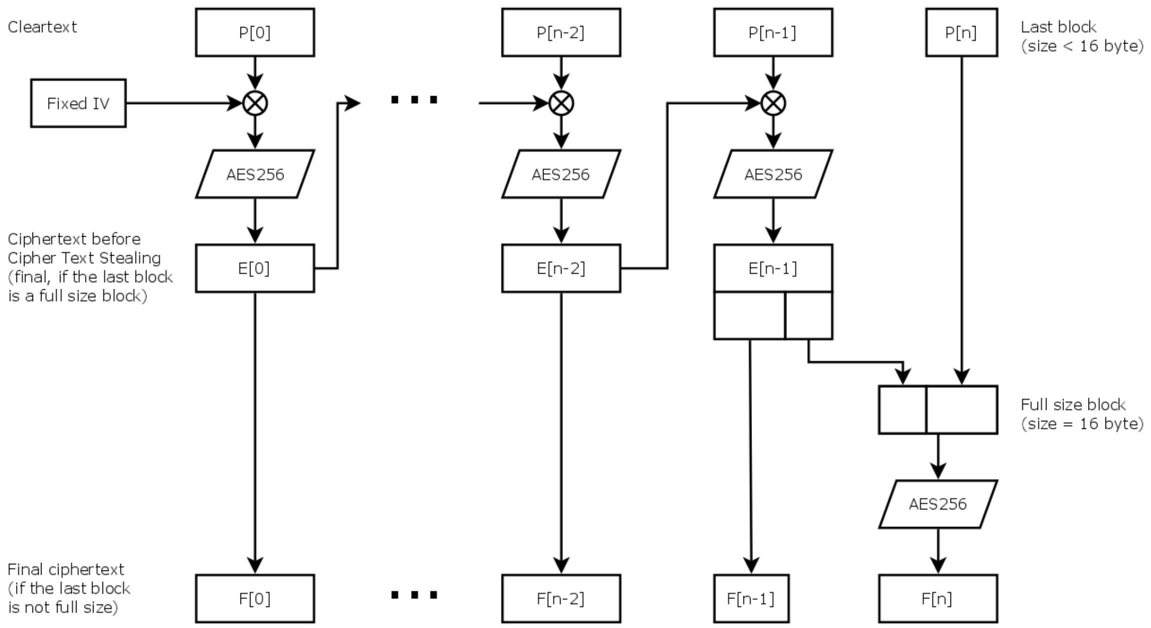


Figure 4 – The applied config file encryption method used by the main module of Duqu 2.0 (and by the old Duqu sample)

With the exception of the last two input blocks, consecutive blocks of the cleartext data are encrypted with the AES encryption algorithm in CBC mode. Accordingly, the first block of the input data (”P[0]”) is XORed with a fixed initialization vector (named as ”Fixed IV” in the figure above). This 128-bit initialization vector (IV) differs between the old Duqu and the new Duqu 2.0 samples. The value of this IV is generated from a magic constant, as it can be seen in the highlighted parts of the previous assembly code. As this magic constant is different in the old and new samples, the generated IV will also be different.

The result of the previously mentioned XOR operation gives the first input block of the AES encryption algorithm (“AES-256” is in use). The number 256 means that the AES algorithm has 256-bit key size. The block size of the AES cipher is constant 128 bits (16 bytes). “E[0]” is the first output of the block cipher, so it will be the first encrypted block (“F[0]”).

Output of the block cipher (“E[0]”) is then XORed with the second input block (“P[1]”), and the resulting block will be encrypted with AES-256. This procedure continues until the encryption of the last but first block of the cleartext data.

If the size of the input data is an integer multiple of the block size of AES (i.e., 128 bits), then the remaining last two blocks of the cleartext are encrypted in the same manner as the previous input blocks. So, in this case, the whole encryption routine matches a simple CBC mode encryption.

However, if the size of the input data is not an exact multiple of the AES block size, the last partial block of the input data needs padding to be completed to a full block. In case of Duqu 2.0, the developers of the malware didn’t use padding in a traditional way. Instead, they use a CTS-like (Ciphertext Stealing) method. The essence of the method used by the encryption routine is that a part of the last but first block of the input data is encrypted twice using AES.

The last but first block (”P[n-1]”) of the cleartext data is XORed with the previous ciphertext block (”E[n-2]”) and encrypted with AES-256 as previously. The result of this operation is the ”E[n-1]” output block. The ”E[n-1]” output block won’t be directly used as the (n-1)st ciphertext block. Instead, the output ”E[n-1]” is splitted into two distinct parts: ”F[n-1]” and another part which is then fed into the AES encryptor again.

The last cleartext partial block ("P[n]") – which has size less than 16 bytes – is completed from its beginning to get a full AES input block. The data used for completing the last partial block is taken from the end of the previous AES output block ("E[n-1]"). The resulting block will be fed into the AES-256 cipher in the last step of the encryption process. The output of the last invocation of the AES cipher will be the last ciphertext block ("F[n]"). The output of the last but first invocation of the AES encryptor ("E[n-1]") is split into two parts, and the first part of size `size_of_the_last_cleartext_block` will be the (n-1)st ciphertext block ("F[n-1]").

The old Duqu samples used exactly the same encryption method. The decompiled code of the AES encryptor of Duqu can be seen in the following sample, and one can see that this code implements the method we have just explained and illustrated in the block diagram of Figure 8.

```
void aeswrapper::encrypt(aeswrapper *this)
{
  unsigned __int8 *cursor, *first_block, *prev_encrypted_block,
                  *current_block, *last_block;
  int i, j, offset_to_iv, offset_to_previous_block;

  // First block
  cursor = aeswrapper::nth_block(this, 0);
  offset_to_iv = (char *)this - (char *)cursor;

  i = 16;
  do
  {
    *cursor ^= cursor[offset_to_iv];     // Buffer overflow if data
    ++cursor;                            // is under 16 bytes
    --i;
  }
  while ( i );

  first_block = aeswrapper::nth_block(this, 0);
  AES::encrypt(&this->aes, first_block, first_block);

  // Other full blocks
  j = 1;
  if ((this->data->vtable->length(this->data) & 0xFFFFFFF0) > 0x10)
```

```
  {
    do
    {
      prev_encrypted_block = aeswrapper::nth_block(this, j - 1);
      cursor = aeswrapper::nth_block(this, j);
      offset_to_previous_block = prev_encrypted_block - cursor;

      i = 16;
      do
      {
        *cursor ^= cursor[offset_to_previous_block];
        ++cursor;
        --i;
      }
      while ( i );

      current_block = aeswrapper::nth_block(this, j);
      AES::encrypt(&this->aes, current_block, current_block);
      ++j;
    }
    while ( j < this->data->vtable->length(this->data) >> 4 );
  }

  // Last block
  if ( this->data->vtable->length(this->data) & 0xF )
  {
    last_block = aeswrapper::last_block(this);
    AES::encrypt(&this->aes, last_block, last_block); // Buffer underwrite
                                                      // if data is under 16
                                                      // bytes

  }
}
```

Sample 5 – Main file encryption routine (same in the new and old sample) with implementation bugs – highlighted (red comments)

The next table compares the AES initialization routines of the old Duqu sample (upper part of the table) and the main module of Duqu 2.0 (lower part of the table).

First, the initialization routine copies the pointer of the input data buffer into the "data" member of the "aeswrapper" object. The routine takes this pointer as its second parameter. The first parameter is the pointer (reference) of the object instance, since in C++, the first (hidden) parameter of a (non-static) class method is always the pointer of the object, or in other words, the "this" pointer. In case of Duqu 2.0, the routine has a third parameter, the pointer to the buffer containing the key.

After that, the content of the "key" buffer (which is a global buffer in the first case) is copied into the local "key_" buffer in both cases. Then the **`prepare_key`** method of the AES object prepares the final encryption key based on this key, and feeds it into the "aeswrapper" object. Invocation of the **`prepare_key`** method can also be seen in the assembly view (see Sample 4.), the method is referred by the name **`AES1_sub_2F9B1`** in case of Duqu and **`AES_1_sub_1001690A`** in case of Duqu 2.0. In the Duqu 2.0 case, the function has one more parameter, as this can also be seen in the assembly view, and the length of the AES key is chosen as 256 bits.

Finally, the remaining part of the code initializes the IV member of "aeswrapper" object. Every byte of the IV is generated by XORing the index of the actual byte with a magic constant ("0xDEADBABE" and "0x248561EF", respectively, in the two cases). Byte index starts from zero.

```
aeswrapper *aeswrapper::initialize(aeswrapper *this, buffer *data)
{
  unsigned int i;
  char key_[32];

  this->data = data;
  // Key is a constant global variable with fixed value
  qmemcpy(key_, key, sizeof(key_));
  // AES::prepare_key assumes that the key is always 256 bits
  AES::prepare_key(&this->aes, key_);

  i = 0;
  do
  {
```

```
    this->iv[i] = i ^ 0xDEADBABE;  // Magic value
    ++i;
  }
  while ( i < 4 );

  return this;
}
```

```
aeswrapper *aeswrapper::initialize(aeswrapper *this, buffer *data,
                                   char *key
                                   )
{
  unsigned int i;
  char key_[32];

  this->data = data;
  // Key is an argument
  qmemcpy(key_, key, sizeof(key_));
  // AES::prepare_key takes a key_length argument, supports 128, 192, 256
  AES::prepare_key(&this->aes, key_, 256);

  i = 0;
  do
  {
    this->iv[i] = i ^ 0x248561EF;  // Magic value
    ++i;
  }
  while ( i < 4 );

  return this;
}
```

Sample 6 – Old Duqu and new Duqu 2.0 encryption initialization routine with differences – highlighted (red comments)

As we can see, there are only three small differences between the routines: the magic constants used by the IV generation, the fact that in Duqu the key is a constant global variable with fixed value while in Duqu 2.0 it is an argument of the initialization function, and finally, the possible length of the key.

In case of Duqu, the **`prepare_key`** function assumes that the key is always 256 bits, while in case of Duqu 2.0, the **`prepare_key`** function takes the key length as an argument. Key length can be 128, 192 or 256 bits.

2.4. Format of the (encrypted) configuration file

Under the encryption layer (which is identical in the new and old samples as described in the previous section), the configuration file format of the new Duqu 2.0 samples is very similar to the old Duqu config file format. For an overview, see Figure 5 below.

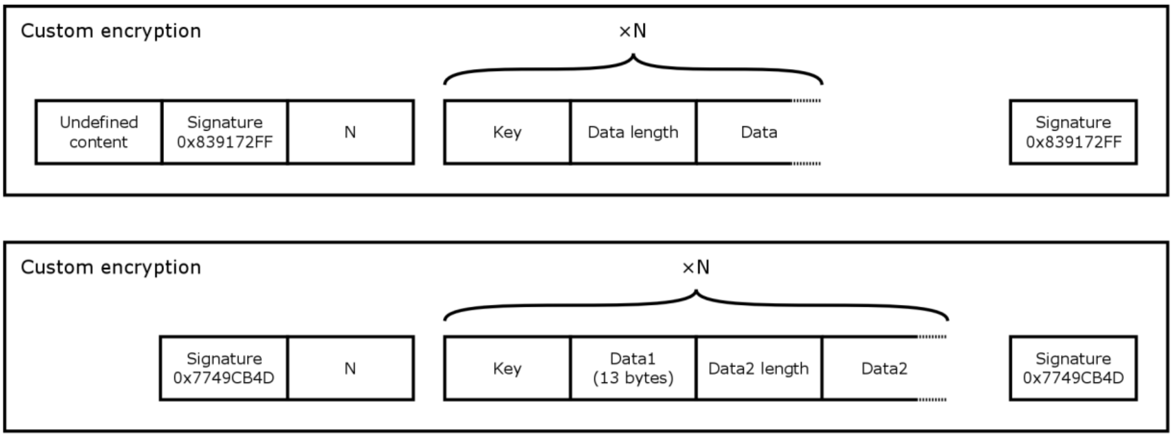


Figure 5 – File format found in Duqu (first diagram) and Duqu 2.0 (second diagram)
(Rectangles always denote little endian 4 byte integers if not stated otherwise)

The format is designed to hold *key-value pairs*. The keys are always 4-byte long, and the values can be of arbitrary size. We believe that the keys are timestamps and the values are configuration entries, although the file format could hold any other similarly structured information (e.g. configurations).

The old file format begins with 4 bytes whose value is undefined. In the serialization process, it is read from an uninitialized buffer, and it is ignored in the deserialization process. The new file format does not have such a beginning byte sequence.

The main part of the file format is surrounded by 4 signature bytes at the beginning and at the end. The byte sequence in the old Duqu file format is 0x839172FF, and in the new Duqu 2.0 version, it is 0x7749CB4D.

In both cases, the next integer indicates the number of entries, followed by the entries themselves.

Each entry begins with a 4-byte key, and then the value. In the new format, the value always begins with 13 bytes (that can be logically divided into four 4 byte integers and a 1 byte value: 4+4+1+4+4), but in the old format, this is missing. Furthermore, the value contains a variable size part in both formats. This is a length prefixed buffer that can hold arbitrary data.

In essence, the only difference between the Duqu and the Duqu 2.0 config file formats is the presence of the undefined 4 bytes at the beginning of the file in the old version, and the presence of the 13 additional value bytes in the new version.

2.5. Logging functions

We've identified a characteristic logging function that is present in both Duqu and Duqu 2.0, and is used extensively in the networking (mainly HTTP handling) part of the code. The logging function itself is identical, and the data structure used for storing log entries is very similar.

The Duqu version of the data structure has embedded function pointers, while the Duqu 2.0 version uses a virtual function table like structure. The main difference from a C++ virtual function table is that the pointer to the table is the last field of the associated structure instead of the first field (see Figure 6).

In general, change in the coding style can be seen all over the code. While Duqu uses object oriented style that is similar, but not identical to what C++ compilers do, Duqu 2.0 moved mainly to "real" C++, but there are still deviations from the standard C++ style (like the previously function table).

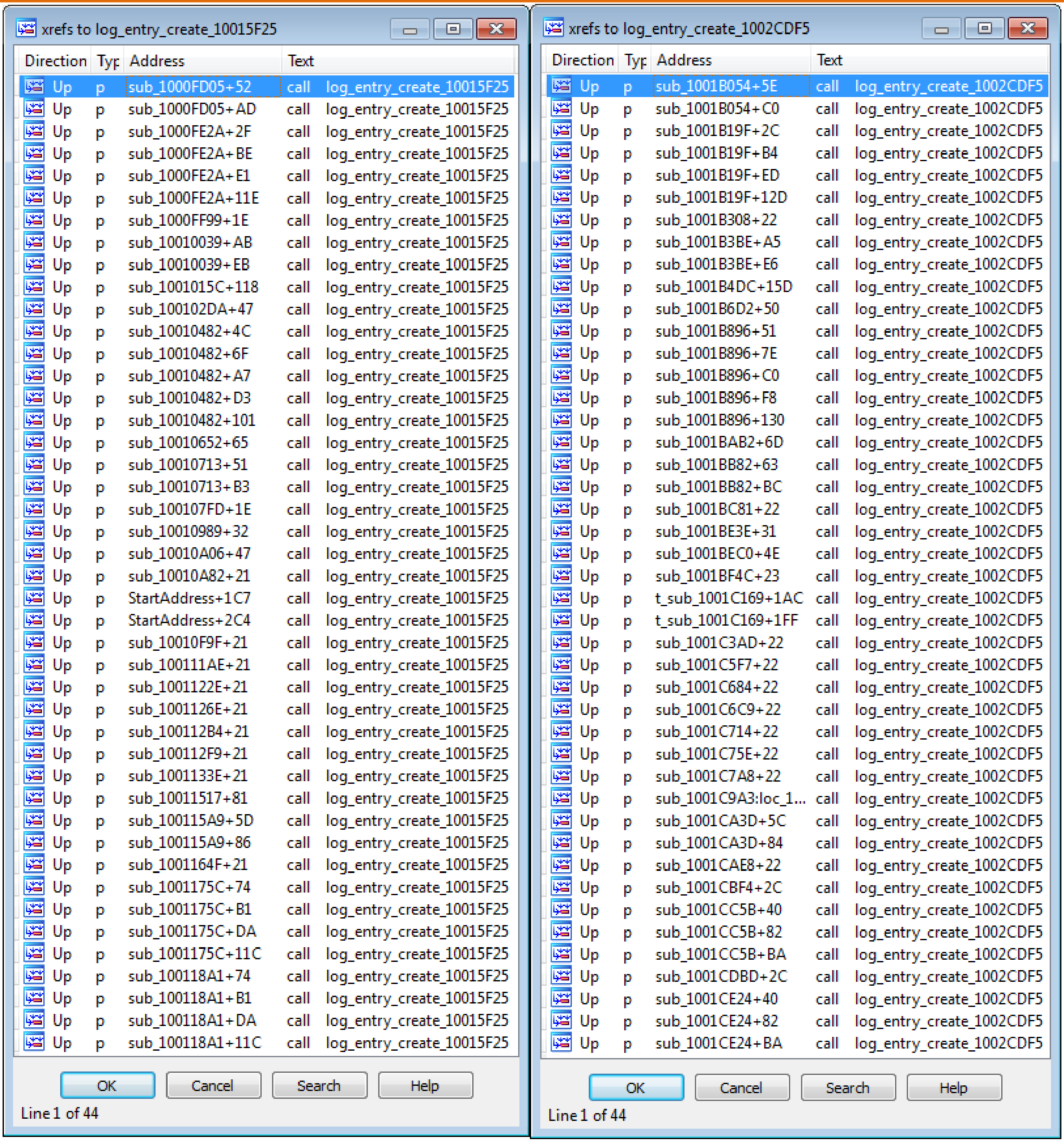
```
00000000 log_entry       struc ; (sizeof=0x82C,
00000000 field_0         dd ?
00000004 field_4         dd ?
00000008 field_8         dd ?
0000000C field_C         dd ?
00000010 field_10        dd ?
00000014 field_14        db ?
00000015 field_15        db ?
00000016 message         dw 1025 dup(?)
00000818 timestamp       dd ?
0000081C field_81C       dd ?
00000820 fn_1            dd ?
00000824 fn_2            dd ?
00000828 destructor      dd ?
0000082C log_entry       ends
0000082C
```

```
00000000 log_entry       struc ; (sizeof=0x824,
00000000 field_0         dd ?
00000004 field_4         dd ?
00000008 field_8         dd ?
0000000C field_C         dd ?
00000010 field_10        dd ?
00000014 field_14        db ?
00000015                 db ? ; undefined
00000016 message         dw 1025 dup(?)
00000818 timestamp       dd ?
0000081C field_81C       dd ?
00000820 vtable          dd ?
00000824 log_entry       ends
00000824
00000000 ; ------------------------------------
00000000
00000000 log_entry_vtable struc ; (sizeof=0xC,
00000000
00000000 fn_1            dd ?
00000004 fn_2            dd ?
00000008 destructor      dd ?
0000000C log_entry_vtable ends
0000000C
```

Figure 6 – Log entry structure and the associated virtual function table in Duqu and Duqu 2.0

Both the Duqu and Duqu 2.0 avoids storing the messages logged through this function. In both codebase, a “handle_log_entry” function is called after creating the log entry structure, but this function throws the object away (frees the memory) and does not print or save it. The authors probably used C/C++ macros to avoid detailed logging in release builds, but in this case we still see the logging function invocation. In this case, the macro was probably placed in the function that should have printed the log message (handle_log_entry), and since this is a virtual function, the compiler could not optimize out the function invocations directly.

The logging function is called equal times in the Duqu and the Duqu 2.0 samples, and the invocation is always very similar (see Figure 7). The arguments are usually not strings describing the event directly, but 4 byte magic numbers. The logging function is invoked equal times, and the magic numbers are almost always identical in Duqu and Duqu 2.0.



xrefs to log_entry_create_10015F25

Direction	Typ	Address	Text
Up	p	sub_1000FD05+52	call log_entry_create_10015F25
Up	p	sub_1000FD05+AD	call log_entry_create_10015F25
Up	p	sub_1000FE2A+2F	call log_entry_create_10015F25
Up	p	sub_1000FE2A+BE	call log_entry_create_10015F25
Up	p	sub_1000FE2A+E1	call log_entry_create_10015F25
Up	p	sub_1000FE2A+11E	call log_entry_create_10015F25
Up	p	sub_1000FF99+1E	call log_entry_create_10015F25
Up	p	sub_10010039+AB	call log_entry_create_10015F25
Up	p	sub_10010039+EB	call log_entry_create_10015F25
Up	p	sub_1001015C+118	call log_entry_create_10015F25
Up	p	sub_100102DA+47	call log_entry_create_10015F25
Up	p	sub_10010482+4C	call log_entry_create_10015F25
Up	p	sub_10010482+6F	call log_entry_create_10015F25
Up	p	sub_10010482+A7	call log_entry_create_10015F25
Up	p	sub_10010482+D3	call log_entry_create_10015F25
Up	p	sub_10010482+101	call log_entry_create_10015F25
Up	p	sub_10010652+65	call log_entry_create_10015F25
Up	p	sub_10010713+51	call log_entry_create_10015F25
Up	p	sub_10010713+B3	call log_entry_create_10015F25
Up	p	sub_100107FD+1E	call log_entry_create_10015F25
Up	p	sub_10010989+32	call log_entry_create_10015F25
Up	p	sub_10010A06+47	call log_entry_create_10015F25
Up	p	sub_10010A82+21	call log_entry_create_10015F25
Up	p	StartAddress+1C7	call log_entry_create_10015F25
Up	p	StartAddress+2C4	call log_entry_create_10015F25
Up	p	sub_10010F9F+21	call log_entry_create_10015F25
Up	p	sub_100111AE+21	call log_entry_create_10015F25
Up	p	sub_1001122E+21	call log_entry_create_10015F25
Up	p	sub_1001126E+21	call log_entry_create_10015F25
Up	p	sub_100112B4+21	call log_entry_create_10015F25
Up	p	sub_100112F9+21	call log_entry_create_10015F25
Up	p	sub_1001133E+21	call log_entry_create_10015F25
Up	p	sub_10011517+81	call log_entry_create_10015F25
Up	p	sub_100115A9+5D	call log_entry_create_10015F25
Up	p	sub_100115A9+86	call log_entry_create_10015F25
Up	p	sub_1001164F+21	call log_entry_create_10015F25
Up	p	sub_1001175C+74	call log_entry_create_10015F25
Up	p	sub_1001175C+B1	call log_entry_create_10015F25
Up	p	sub_1001175C+DA	call log_entry_create_10015F25
Up	p	sub_1001175C+11C	call log_entry_create_10015F25
Up	p	sub_100118A1+74	call log_entry_create_10015F25
Up	p	sub_100118A1+B1	call log_entry_create_10015F25
Up	p	sub_100118A1+DA	call log_entry_create_10015F25
Up	p	sub_100118A1+11C	call log_entry_create_10015F25

OK Cancel Search Help

Line 1 of 44

xrefs to log_entry_create_1002CDF5

Direction	Typ	Address	Text
Up	p	sub_1001B054+5E	call log_entry_create_1002CDF5
Up	p	sub_1001B054+C0	call log_entry_create_1002CDF5
Up	p	sub_1001B19F+2C	call log_entry_create_1002CDF5
Up	p	sub_1001B19F+B4	call log_entry_create_1002CDF5
Up	p	sub_1001B19F+ED	call log_entry_create_1002CDF5
Up	p	sub_1001B19F+12D	call log_entry_create_1002CDF5
Up	p	sub_1001B308+22	call log_entry_create_1002CDF5
Up	p	sub_1001B3BE+A5	call log_entry_create_1002CDF5
Up	p	sub_1001B3BE+E6	call log_entry_create_1002CDF5
Up	p	sub_1001B4DC+15D	call log_entry_create_1002CDF5
Up	p	sub_1001B6D2+50	call log_entry_create_1002CDF5
Up	p	sub_1001B896+51	call log_entry_create_1002CDF5
Up	p	sub_1001B896+7E	call log_entry_create_1002CDF5
Up	p	sub_1001B896+C0	call log_entry_create_1002CDF5
Up	p	sub_1001B896+F8	call log_entry_create_1002CDF5
Up	p	sub_1001B896+130	call log_entry_create_1002CDF5
Up	p	sub_1001BAB2+6D	call log_entry_create_1002CDF5
Up	p	sub_1001BB82+63	call log_entry_create_1002CDF5
Up	p	sub_1001BB82+BC	call log_entry_create_1002CDF5
Up	p	sub_1001BC81+22	call log_entry_create_1002CDF5
Up	p	sub_1001BE3E+31	call log_entry_create_1002CDF5
Up	p	sub_1001BEC0+4E	call log_entry_create_1002CDF5
Up	p	sub_1001BF4C+23	call log_entry_create_1002CDF5
Up	p	t_sub_1001C169+1AC	call log_entry_create_1002CDF5
Up	p	t_sub_1001C169+1FF	call log_entry_create_1002CDF5
Up	p	sub_1001C3AD+22	call log_entry_create_1002CDF5
Up	p	sub_1001C5F7+22	call log_entry_create_1002CDF5
Up	p	sub_1001C684+22	call log_entry_create_1002CDF5
Up	p	sub_1001C6C9+22	call log_entry_create_1002CDF5
Up	p	sub_1001C714+22	call log_entry_create_1002CDF5
Up	p	sub_1001C75E+22	call log_entry_create_1002CDF5
Up	p	sub_1001C7A8+22	call log_entry_create_1002CDF5
Up	p	sub_1001C9A3:loc_1...	call log_entry_create_1002CDF5
Up	p	sub_1001CA3D+5C	call log_entry_create_1002CDF5
Up	p	sub_1001CA3D+84	call log_entry_create_1002CDF5
Up	p	sub_1001CAE8+22	call log_entry_create_1002CDF5
Up	p	sub_1001CBF4+2C	call log_entry_create_1002CDF5
Up	p	sub_1001CC5B+40	call log_entry_create_1002CDF5
Up	p	sub_1001CC5B+82	call log_entry_create_1002CDF5
Up	p	sub_1001CC5B+BA	call log_entry_create_1002CDF5
Up	p	sub_1001CDBD+2C	call log_entry_create_1002CDF5
Up	p	sub_1001CE24+40	call log_entry_create_1002CDF5
Up	p	sub_1001CE24+82	call log_entry_create_1002CDF5
Up	p	sub_1001CE24+BA	call log_entry_create_1002CDF5

OK Cancel Search Help

Line 1 of 44

```
logger2_vtable = logger2->vtable;
log_entry = log_entry_create_1002CDF5(0x56E7817A, 0xF7ADDCA2, 0x7B421F59, 0, 0, 4);
logger2_vtable->handle_log_entry(thread->logger2, log_entry);
```

Figure 7 – References to the logger function in Duqu and Duqu 2.0, and one of the invocations

2.6. Command & Control communication

The network communication methods used by Duqu 2.0 are described in the following list.

SocketServer1:

In export function **`nr1`**, if in the config the "startSockServer" parameter is set, it will start a server accordingly

SocketServer2:

Binds between ports 17000 and 17100, can be configured to be client or server

GifServer:

With Custom HTTP Server implementation, possibly based on **SocketServer2**

PipeComm:

PIPE or IPC communication, customizable network communication

HttpClient:

WinHTTP-based, simple client, uses "COUNTRY=" in cookie parameters, (standard HTTP client)

Table 2 – Network communication methods used by Duqu 2.0

Duqu has used a very unique user agent string when communicating over HTTP:

```
Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.9)
```

In contrast, Duqu 2.0 chooses user agent string randomly from a large set of often used values listed in Sample 7.

The following list shows the browser agent strings found in Duqu 2.0:

```
Mozilla/5.0 (Windows NT 5.1) AppleWebKit/535.6 (KHTML, like Gecko) Chrome/16.0.897.0 Safari/535.6

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; chromeframe/11.0.696.57)

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0; Trident/5.0; chromeframe/11.0.696.57)

Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; InfoPath.1; SV1; .NET CLR 3.8.36217; WOW64; en-US)

Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 1.0.3705; .NET CLR 1.1.4322)

Mozilla/5.0 (Windows NT 6.2; WOW64; rv:15.0) Gecko/20120910144328 Firefox/15.0.2

Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)

Mozilla/5.0 (Windows NT 6.1; rv:6.0) Gecko/20110814 Firefox/6.0

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 2.0.50727; Media Center PC 6.0)

Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.2; Trident/4.0; Media Center PC 4.0; SLCC1; .NET CLR 3.0.04320)

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; FunWebProducts)

Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.15 (KHTML, like Gecko) Chrome/24.0.1295.0 Safari/537.15
```

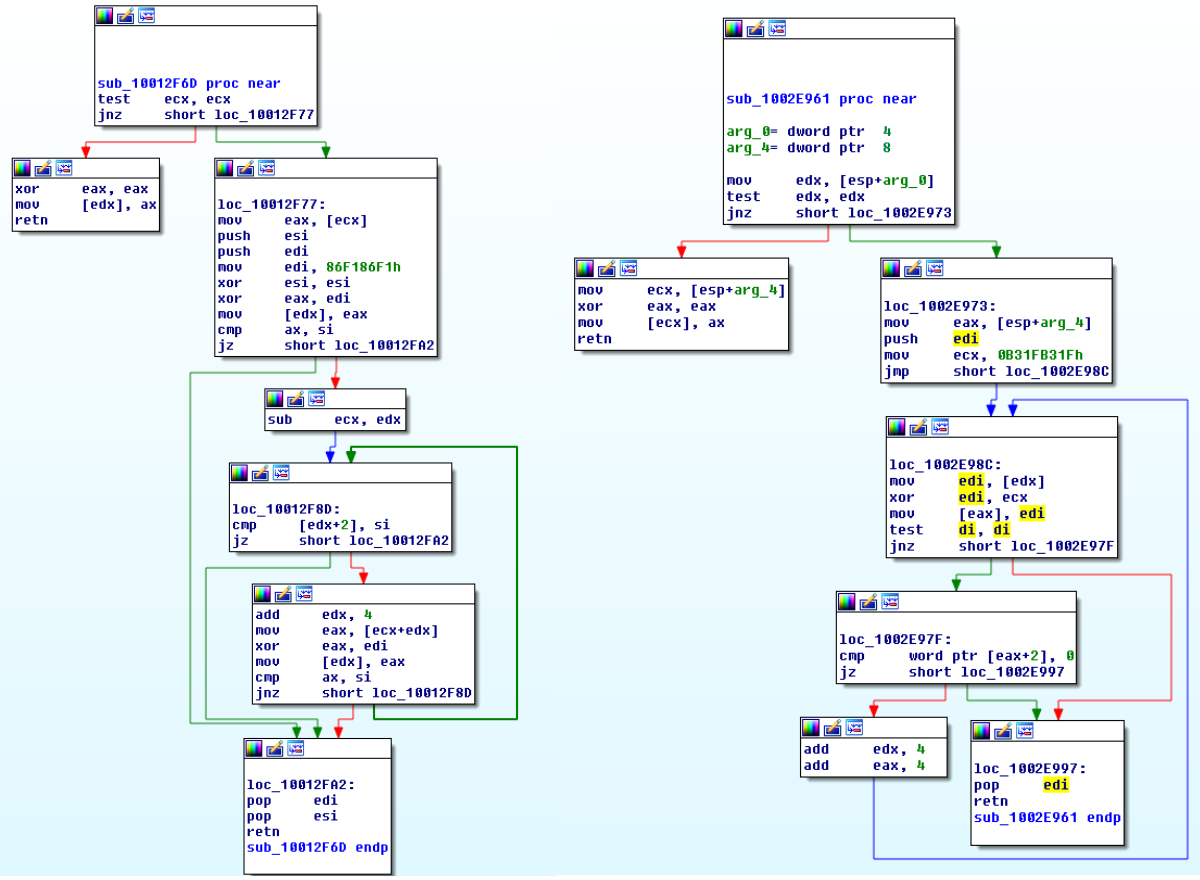
```
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/5.0)

Mozilla/5.0 (Windows NT 6.1; rv:12.0) Gecko/20120403211507 Firefox/12.0

Mozilla/5.0 (Windows NT 6.2) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.94 Safari/537.4

Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:5.0) Gecko/20110619 Firefox/5.0

Mozilla/5.0 (Windows; U; MSIE 7.0; Windows NT 6.0; en-US)

Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)

Mozilla/5.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4325)

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/4.0; GTB7.4; InfoPath.1; SV1; .NET CLR 2.8.52393; WOW64; en-US)

Mozilla/5.0 (Windows NT 6.1) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.77 Safari/535.7ad-imcjapan-syosyaman-xkgi3lqg03!wgz

Mozilla/4.0 (compatible; MSIE 7.0b; Windows NT 5.1; FDM; .NET CLR 1.1.4322)

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; InfoPath.2; .NET CLR 1.1.4322; .NET4.0C; Tablet PC 2.0)

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; GTB6.5; QQDownload 534; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ; SLCC2; .NET CLR 2.0.50727; Media Center PC 6.0; .NET CLR 3.5.30729; .NET CLR 3.0.30729)

Mozilla/4.0 (compatible; MSIE 7.0b; Windows NT 5.1; .NET CLR 1.1.4322)

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0) chromeframe/10.0.648.205

Mozilla/5.0 (Windows NT 6.1; rv:15.0) Gecko/20120716 Firefox/15.0a2

Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.66 Safari/535.11

Mozilla/5.0 (Windows NT 6.0; WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.56 Safari/535.11

Mozilla/5.0 (Windows NT 6.2) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.26 Safari/537.11
```

```
Mozilla/5.0 (Windows NT 6.1; U; ru; rv:5.0.1.6) Gecko/20110501 Firefox/5.0.1 Firefox/5.0.1

Mozilla/5.0 (Windows NT 6.1.1; rv:5.0) Gecko/20100101 Firefox/5.0

Mozilla/5.0 (compatible; MSIE 7.0; Windows NT 5.2; WOW64; .NET CLR 2.0.50727)

Mozilla/5.0 (Windows NT 6.1; WOW64; rv:6.0a2) Gecko/20110612 Firefox/6.0a2

Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0

Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.16) Gecko/20120427 Firefox/15.0a1

Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 1.1.4322; .NET CLR 2.0.50727)

Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.17 Safari/537.11

Mozilla/5.0 (Windows NT 5.1; rv:6.0) Gecko/20100101 Firefox/6.0 FirePHP/0.6

Mozilla/4.0 (MSIE 6.0; Windows NT 5.1)

Mozilla/5.0 (Windows NT 6.2; Win64; x64; rv:16.0.1) Gecko/20121011 Firefox/16.0.1

Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:5.0) Gecko/20100101 Firefox/5.0

Mozilla/5.0 (Windows NT 6.0; WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.66 Safari/535.11

Mozilla/5.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR 2.0.50727; Media Center PC 5.0; c .NET CLR 3.0.04506; .NET CLR 3.5.30707; InfoPath.1; el-GR)

Mozilla/5.0 (Windows NT 6.1; U;WOW64; de;rv:11.0) Gecko Firefox/11.0

Mozilla/3.0 (Windows NT 6.1; rv:2.0.1) Gecko/20100101 Firefox/5.0.1

Mozilla/5.0 (Windows; U; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727)

Mozilla/5.0 (Windows NT 6.1; de;rv:12.0) Gecko/20120403211507 Firefox/12.0
```

Sample 7 –48 Browser agent strings in Duqu 2.0

2.7. DLL imports

Duqu 2.0 uses more than one method to import functions from DLLs. One of the methods utilizes a hash method to represent function names as 4 byte integers. It iterates through all importable function and finds the one whose function name hash matches the given hash. This hash function uses a magic number. A very similar import method and hash function is used in Duqu and Duqu 2.0 although the magic numbers are different: 0x86F186F1 and 0xB31FB31F. Note that even the inner structure of the magic numbers are similar (2x2 bytes).



Sample 8 – Hash function used for imports in Duqu and Duqu 2.0

3.Indicators of Compromise

3.1. Detection based on communications

The malware can transmit information through HTTP traffic. It is most likely that one or more infected computers can be proxy points towards the attacker, meaning that other infected computers communicate with these proxies. These proxies can act as HTTP or HTTPS servers. For HTTPS, a self signed certificate is created by the malware itself. (Most likely by contacting gpl3.selfsigned.org). The Common Name (CN) field seems to be "*" in the created certificate. During data transfer, the malware uses **`<5 random numbers>.gif`** for file name and a 843-byte GIF file + additional random bytes. The transmissions may be protected by AES.

One possible way to detect such transmission (if cleartext traffic is somehow available) to detect the actual 843-byte GIF file. For the known two samples, this GIF portion was identical.

The actual image in hex dump is the following:

```
00000000  47 49 46 38 39 61 0b 00  0b 00 70 00 00 21 f9 04  |GIF89a....p..!..|
00000010  01 00 00 fc 00 2c 00 00  00 00 0b 00 0b 00 87 00  |.....,..........|
00000020  00 00 00 00 33 00 00 66  00 00 99 00 00 cc 00 00  |....3..f........|
00000030  ff 00 2b 00 00 2b 33 00  2b 66 00 2b 99 00 2b cc  |..+..+3.+f.+..+.|
00000040  00 2b ff 00 55 00 00 55  33 00 55 66 00 55 99 00  |.+..U..U3.Uf.U..|
00000050  55 cc 00 55 ff 00 80 00  00 80 33 00 80 66 00 80  |U..U......3..f..|
00000060  99 00 80 cc 00 80 ff 00  aa 00 00 aa 33 00 aa 66  |............3..f|
00000070  00 aa 99 00 aa cc 00 aa  ff 00 d5 00 00 d5 33 00  |..............3.|
00000080  d5 66 00 d5 99 00 d5 cc  00 d5 ff 00 ff 00 00 ff  |.f..............|
00000090  33 00 ff 66 00 ff 99 00  ff cc 00 ff ff 33 00 00  |3..f.........3..|
000000a0  33 00 33 33 00 66 33 00  99 33 00 cc 33 00 ff 33  |3.33.f3..3..3..3|
000000b0  2b 00 33 2b 33 33 2b 66  33 2b 99 33 2b cc 33 2b  |+.3+33+f3+.3+.3+|
000000c0  ff 33 55 00 33 55 33 33  55 66 33 55 99 33 55 cc  |.3U.3U33Uf3U.3U.|
000000d0  33 55 ff 33 80 00 33 80  33 33 80 66 33 80 99 33  |3U.3..3.33.f3..3|
000000e0  80 cc 33 80 ff 33 aa 00  33 aa 33 33 aa 66 33 aa  |..3..3..3.33.f3.|
000000f0  99 33 aa cc 33 aa ff 33  d5 00 33 d5 33 33 d5 66  |.3..3..3..3.33.f|
00000100  33 d5 99 33 d5 cc 33 d5  ff 33 ff 00 33 ff 33 33  |3..3..3..3..3.33|
00000110  ff 66 33 ff 99 33 ff cc  33 ff ff 66 00 00 66 00  |.f3..3..3..f..f.|
00000120  33 66 00 66 66 00 99 66  00 cc 66 00 ff 66 2b 00  |3f.ff..f..f..f+.|
00000130  66 2b 33 66 2b 66 66 2b  99 66 2b cc 66 2b ff 66  |f+3f+ff+.f+.f+.f|
00000140  55 00 66 55 33 66 55 66  66 55 99 66 55 cc 66 55  |U.fU3fUffU.fU.fU|
00000150  ff 66 80 00 66 80 33 66  80 66 66 80 99 66 80 cc  |.f..f.3f.ff..f..|
00000160  66 80 ff 66 aa 00 66 aa  33 66 aa 66 66 aa 99 66  |f..f..f.3f.ff..f|
00000170  aa cc 66 aa ff 66 d5 00  66 d5 33 66 d5 66 66 d5  |..f..f..f.3f.ff.|
00000180  99 66 d5 cc 66 d5 ff 66  ff 00 66 ff 33 66 ff 66  |.f..f..f..f.3f.f|
```

```
00000190  66 ff 99 66 ff cc 66 ff  ff 99 00 00 99 00 33 99  |f..f..f.......3.|
000001a0  00 66 99 00 99 99 00 cc  99 00 ff 99 2b 00 99 2b  |.f..........+..+|
000001b0  33 99 2b 66 99 2b 99 99  2b cc 99 2b ff 99 55 00  |3.+f.+..+..+..U.|
000001c0  99 55 33 99 55 66 99 55  99 99 55 cc 99 55 ff 99  |.U3.Uf.U..U..U..|
000001d0  80 00 99 80 33 99 80 66  99 80 99 99 80 cc 99 80  |....3..f........|
000001e0  ff 99 aa 00 99 aa 33 99  aa 66 99 aa 99 99 aa cc  |......3..f......|
000001f0  99 aa ff 99 d5 00 99 d5  33 99 d5 66 99 d5 99 99  |........3..f....|
00000200  d5 cc 99 d5 ff 99 ff 00  99 ff 33 99 ff 66 99 ff  |..........3..f..|
00000210  99 99 ff cc 99 ff ff cc  00 00 cc 00 33 cc 00 66  |............3..f|
00000220  cc 00 99 cc 00 cc cc 00  ff cc 2b 00 cc 2b 33 cc  |..........+..+3.|
00000230  2b 66 cc 2b 99 cc 2b cc  cc 2b ff cc 55 00 cc 55  |+f.+..+..+..U..U|
00000240  33 cc 55 66 cc 55 99 cc  55 cc cc 55 ff cc 80 00  |3.Uf.U..U..U....|
00000250  cc 80 33 cc 80 66 cc 80  99 cc 80 cc cc 80 ff cc  |..3..f..........|
00000260  aa 00 cc aa 33 cc aa 66  cc aa 99 cc aa cc cc aa  |....3..f........|
00000270  ff cc d5 00 cc d5 33 cc  d5 66 cc d5 99 cc d5 cc  |......3..f......|
00000280  cc d5 ff cc ff 00 cc ff  33 cc ff 66 cc ff 99 cc  |........3..f....|
00000290  ff cc cc ff ff ff 00 00  ff 00 33 ff 00 66 ff 00  |..........3..f..|
000002a0  99 ff 00 cc ff 00 ff ff  2b 00 ff 2b 33 ff 2b 66  |........+..+3.+f|
000002b0  ff 2b 99 ff 2b cc ff 2b  ff ff 55 00 ff 55 33 ff  |.+..+..+..U..U3.|
000002c0  55 66 ff 55 99 ff 55 cc  ff 55 ff ff 80 00 ff 80  |Uf.U..U..U......|
000002d0  33 ff 80 66 ff 80 99 ff  80 cc ff 80 ff ff aa 00  |3..f............|
000002e0  ff aa 33 ff aa 66 ff aa  99 ff aa cc ff aa ff ff  |..3..f..........|
000002f0  d5 00 ff d5 33 ff d5 66  ff d5 99 ff d5 cc ff d5  |....3..f........|
00000300  ff ff ff 00 ff ff 33 ff  ff 66 ff ff 99 ff ff cc  |......3..f......|
00000310  ff ff ff 00 00 00 00 00  00 00 00 00 00 00 00 08  |................|
00000320  28 00 ed 09 1c 48 50 20  3c 7b 07 13 22 5c 68 70  |(....HP <{.."\hp|
00000330  e0 41 87 0d 1f 2a 64 d8  b0 e2 c4 8b 10 09 4a 8c  |.A...*d.......J.|
00000340  c8 10 63 c5 8f 1b 37 06  04 00 3b                 |..c...7...;|
0000034b
```

Sample 9 – Hexdump of the actual GIF image

The image itself is a small picture, basic color is yellow and there are some orange dots in it:

Sample 10 – The actual GIF image

3.2. Yara rules to identify

For the main binary of the malware, we propose the following rules for detection:

```
rule duqu2
{

strings:
$a = { 0F B6 C8 8B C1 0F AF C9 83 E0 ?? C1 E0 ?? 05 ?? ?? ?? ?? 0F
AF D8 8B ?? ?? ?? 33 D9 }
$b = { 0F 84 ?? ?? ?? ?? 0F B7 06 B9 ?? ?? ?? ?? 33 C1 3D ?? ?? ??
?? 0F 85 ?? ?? ?? ?? 8B }

condition:
any of them

}
```

Sample 11 – Yara rules for detection of Duqu 2.0

4. Conclusion

We've made an initial analysis to prove our claims that there is a strong connection between Duqu and Duqu 2.0 malwares. Our main goal was to highlight the most striking similarities and differences between the samples. Similarities shows that the developers of Duqu 2.0 have reused the code basis of the old Duqu specimens and the differences found in the binaries indicates that the developers of Duqu have modified their tools to avoid detections.

5. References

[CrySySDuqu]

CrySyS, Duqu: A Stuxnet-like malware found in the wild, v0.93 (14/Oct/2011)
http://www.crysys.hu/publications/files/bencsathPBF11duqu.pdf

[SymantecDuqu]

Symantec, W32.Duqu: The precursor to the next Stuxnet, Version 1.4 (November 23, 2011)
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet.pdf

[KasperskyDuqu]

Kaspersky Lab, Duqu: Steal Everything, Kaspersky Lab's investigation - "The Mystery of Duqu" in blogs
http://www.kaspersky.com/about/press/major_malware_outbreaks/duqu

[SymantecDossier]

Symantec, W32.Stuxnet Dossier, Version 1.4 (February 2011) http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[KasperskyDuqu2.0]

Kaspersky Lab, The Duqu 2.0: Technical Details, Version: 1.9.8 (2.June.2015) https://securelist.com/blog/research/70504/the-mystery-of-duqu-2-0-a-sophisticated-cyberespionage-actor-returns/

6. Contact Information

Questions and comments are welcome. The corresponding author is

Dr. Boldizsár Bencsáth
bencsath@crysys.hu

Laboratory of Cryptography and System Security
CrySyS – http://www.crysys.hu/
Budapest University of Technology and Economics
Department of Telecommunications
1117 Magyar Tudósok Krt. 2.
Budapest, Hungary

GPG BENCSATH Boldizsar <boldi@crysys.hu>
Key ID 0x64CF6EFB
Fingerprint 286C A586 6311 36B3 2F94 B905 AFB7 C688 64CF 6EFB