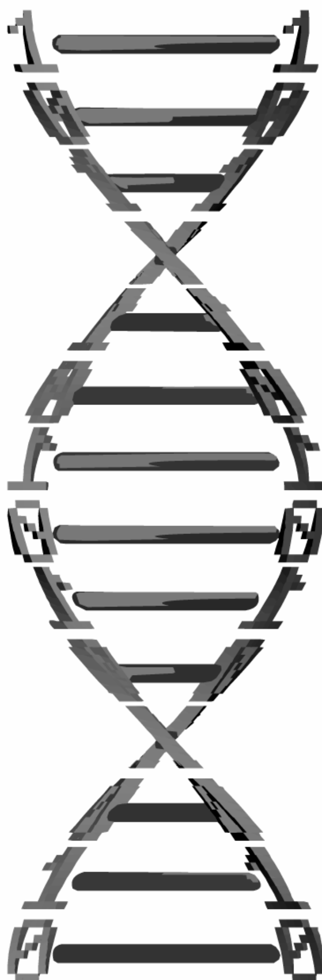
The Evolution of GPCode/Glamour RansomWare

A file-encrypting information stealer

23-July 2007, v1.0



Secure Science Corporation
7770 Regents Rd.
Suite 113-535
San Diego, CA 92122
(877) 570-0455
http://www.securescience.net/

Table of Contents

1 Introduction 3
2 Brothers from the Same Mother 4
3 The Ransom Note and Deep Mystery 7
4 Exploring the New Thread 9
5 Decryptor Development 12
6 Making Your Registry l33t 15
7 References and Tools 16

1 Introduction

This report contains a description of the more obscure, previously undocumented traits belonging to the GPCode/Glamour trojan. The code is a modified version of the Prg/Ntos family which was detailed in depth during our Encrypted Malware Analysis in November 2006. While a majority of the functionality has not changed since then, this recent variant is distinctive enough to warrant additional research. In particular, the trojan is now equipped with the ability to encrypt a victim's files on disk. The motive for adding this feature is clearly monetary, as the victim is advised that the files will remain encrypted unless $300 is turned over to the authors, in exchange for a decryption utility.

This trojan also retains the functionality of hooking API functions to steal information from victims, just like the older ones. As an update, in the 8 months since November, we've recovered stolen data from 51 unique drop sites for use with Intellifound. The 14.5 million records found within these files came from over 152,000 unique victims.

In the forthcoming analysis, we will explore the key points of interest regarding this new feature. We will also present how the encryption algorithm was reverse engineered to build our own decryption program, how users can help protect their file systems in the future, and some interesting tid-bits of information that is only revealed through binary disassembly.

Source code for the decrytor is available here:
http://securescience.net/securescienceblog/ransom-waredecrypted.html

2 Brothers from the Same Mother

Our first notice of the new trojan was received on July 17, 2007 and was accompanied by a brief description of its behavior. After reading this summary, which documented the trojan's interaction with audio.dll, video.dll, and ntos.exe, we were fairly certain that it was just a new variant of Prg/Ntos. However, it would be trivial for a "copycat" malware author to produce a trojan which behaves in a similar manner. To find out for sure, we engaged a binary diff of the Prg/Ntos trojan disassembled during the Encrypted Malware Analysis of 2006 and the new GPCode/Glamour trojan.

The results indicate that these two trojans, found in the wild nearly 6 months apart, originated from the same source tree. This could mean that the original authors are actively modifying the code themselves, or they sold/traded the source code to another group who is now in charge of the modifications. The table below shows the number of functions added, removed, and unchanged between our two samples:

Functions that exist *only* in old	5
Functions that exist *only* in new	105
Functions that exist in both	63 (53 unchanged)

This shows that out of the 168 functions in the new binary, 63 of them exist in Prg/Ntos. Furthermore, 53 of those 63 are direct matches.

As an example, on the following page we show the function which contains the encoding routine for information stolen out of compromised systems' HTTP request buffers. This is the section of code which enabled us to reverse engineer the algorithm and produce a decoder for victim data recovered from blind drop sites around the world. It is important to note that this function is among the 53 matched ones, because this means our decoder will remain effective for all data encoded by the new trojans.

In the diagrams, the three primary nodes are labeled with numbers 1, 2, and 3. Although the positioning is slightly different across the two trojans, they are functionally equivalent.

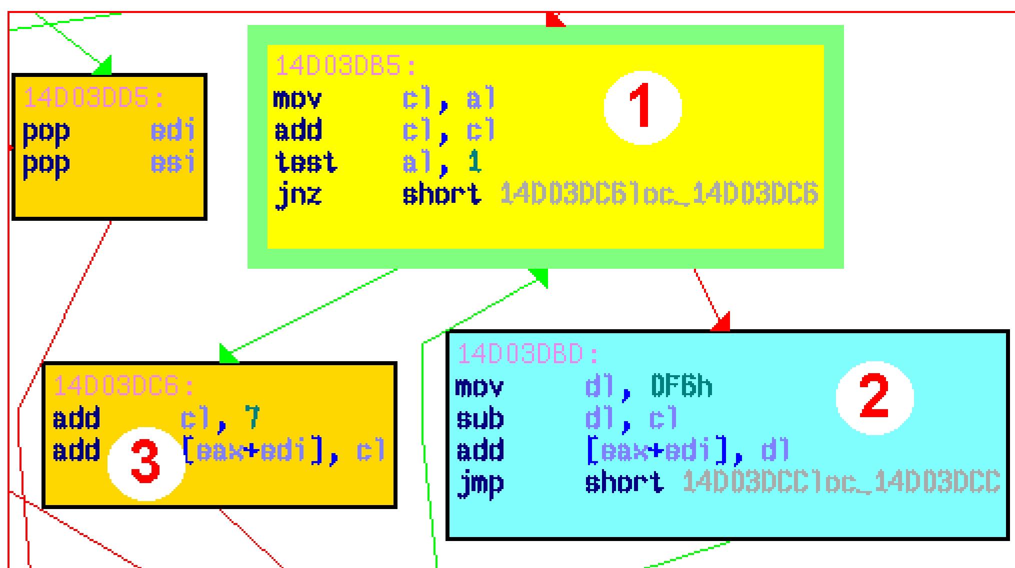


Figure 1: Appearance of encoding algorithm in November 2006.

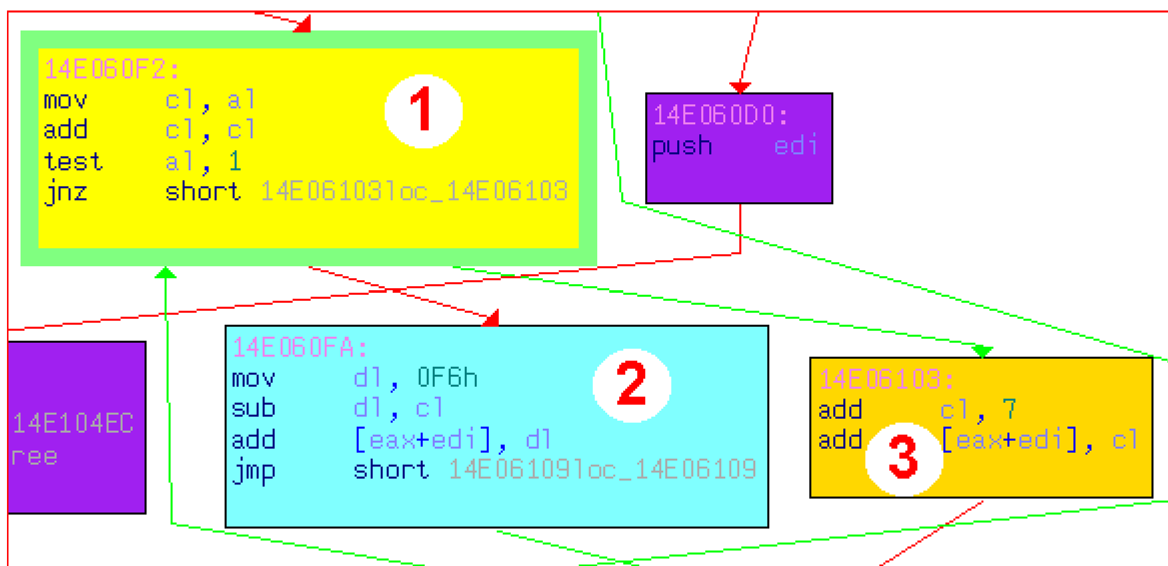


Figure 2: Appearance of encoding algorithm in July 2007.

Through further analysis of the differences, this time focusing on the new functions, we can conclude that the majority are related to either the file encrypting feature or the socket listener. As far back as November 2006, right after disassembly of the original Prg/Ntos sample, it was apparent that changes were already being made to the code. In the final section of the report labeled “New Malware, New Avenues,” we noted that the socket listener had been added to other discovered variants. Therefore, the only bonus feature in this recent sample is the ability to encrypt files.

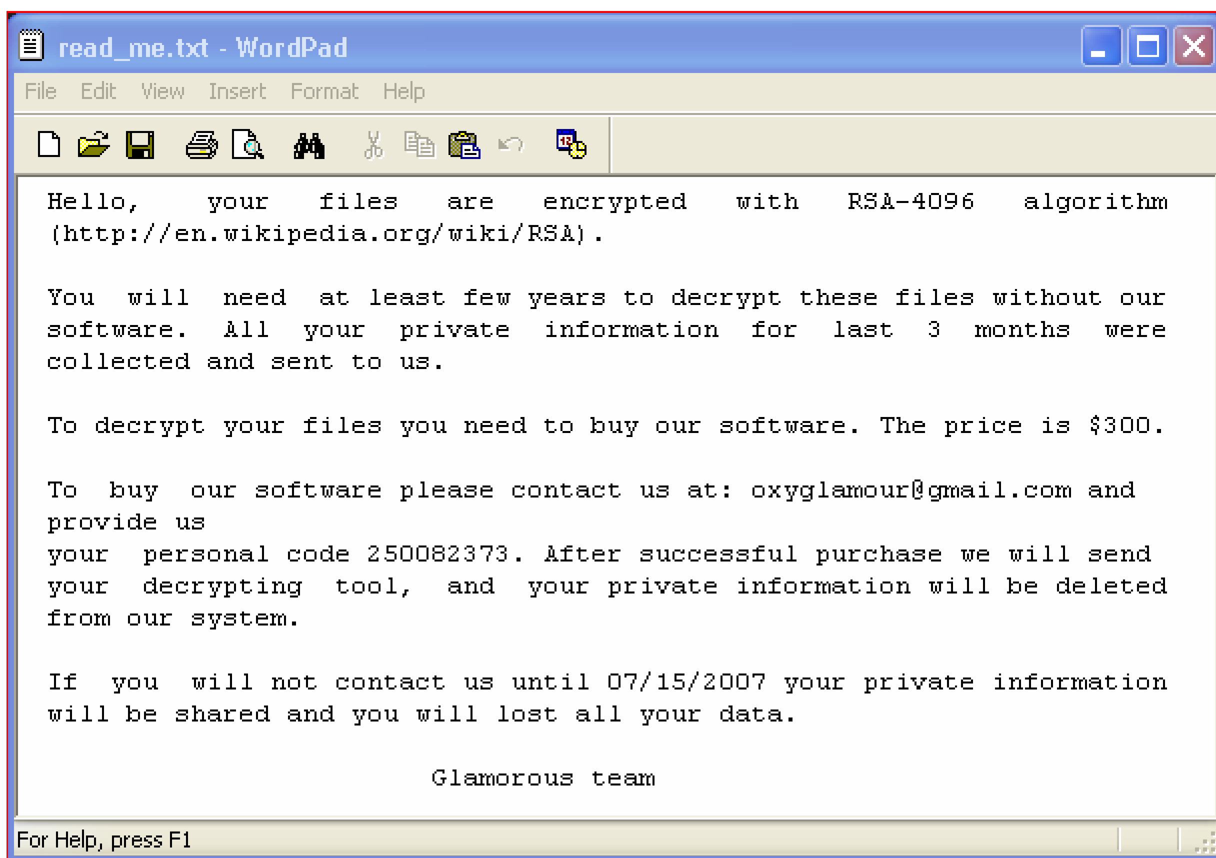
In the screen shot below, we show a brief list of the function addresses and names associated with the new feature. These are not the same names as the original source code; rather we named them according to their observed behavior after analyzing the function's composition.

Function EA	Nodes	Links	Children	Function Name
14e04c71	6	9	4	_check_extension
14e04aaa	23	43	18	_encrypt_file
14e04ecf	6	9	4	_enum_files
14e04f3b	9	14	18	_file_encrypt_thread
14e0c6ce	1	0	2	_formatpipename
14e0cd07	1	0	2	_getpipe
14e04753	4	5	0	_init_key_buffer
14e0480b	5	7	6	_install_reg_win32
14e048a0	9	14	8	_install_reg_wincode
14e059a0	10	16	3	_net_comm
14e0566a	6	7	1	_net_recv
14e05637	6	7	1	_net_send
14e05b0e	6	9	10	_private_comm_thread
14e05084	10	17	12	_upload_data_thread
14e0497d	10	16	12	_write_readme_txt
14e04cd2	4	6	10	do_main_file_work
14e0dfd4	1	0	0	j__WSAStartup
14e0dfbc	1	0	0	j__accept
14e0dfc8	1	0	0	j__bind
14e0df9e	1	0	0	j__closesocket_0
14e0dfaa	1	0	0	j__connect
14e0dfb6	1	0	0	j__gethostbyname

Figure 3: Added functions in the July 2007 sample.

3 The Ransom Note and Deep Mystery

Through dynamic analysis (executing the trojan) we can determine that a large number of files on the target system's disk are unreadable following the full infection. In each directory on disk, where at least one file was encrypted, the trojan leaves a ransom note named read_me.txt. The contents of this note explains that a strong, asymmetric algorithm (4096-bit RSA) was used to encrypt the files and that upon receiving $300 at an email drop box, the authors will provide a decryption program to restore victims' files. The full ransom message is displayed below.



Hello, your files are encrypted with RSA-4096 algorithm (http://en.wikipedia.org/wiki/RSA).

You will need at least few years to decrypt these files without our software. All your private information for last 3 months were collected and sent to us.

To decrypt your files you need to buy our software. The price is $300.

To buy our software please contact us at: oxyglamour@gmail.com and provide us
your personal code 250082373. After successful purchase we will send your decrypting tool, and your private information will be deleted from our system.

If you will not contact us until 07/15/2007 your private information will be shared and you will lost all your data.

Glamorous team

Figure 4: The read_me.txt random message.

It is unacceptable for innocent users to have to pay a price to restore their files, especially when the price is $300 and the recipients are the attackers. Therefore, we engaged a reverse engineering project to develop our own decryptor, which will be free of charge to victims, including the relevant source code for educational purposes of other malware researchers.

In the early stages of research, it was quickly apparent that the files were not really encrypted with 4096-bit RSA. The encrypted files are all exactly 7 bytes larger than the originals, they all contain a common header of "GLAMOUR" (that is the extra 7 bytes), and most of them actually contain readable text at the very end. For example, the following images show the beginning and end appearance of an encrypted

file, viewed with a hex editor. At offsets 0x0 – 0x7, the “GLAMOUR” header is shown. At offsets 0x314 – 0x31B (also 7 bytes), we can see the plain text “/* path” at the very end.

```
00000007  47 4C 41 4D 4F 55 52 E8  E7 B3 7D EE 3D 02 C0 BF  GLAMOURèç³}î= À¿
00000010  2D CA 0B D6 E7 DD FB 58  ED 54 59 2F 05 BA F2 F8  -Ê ÖçÝûXíTY/ ºòø
00000020  C1 C5 0A A5 72 66 7E A3  FB E1 00 36 A9 DE 81 73  ÁÅ ¥rf~£ûá 6©Þ s
00000030  F7 38 D9 15 C4 52 69 5F  2D BD 65 AA 25 A6 55 FE  ÷8Ù ÄRi_-½eª%¦Uþ
00000040  E7 61 5D BD F7 9D 2B 30  99 84 09 3A CF 54 88 B3  ça]½÷ +0■■ :ÏT■³
00000050  7D CD F8 94 46 48 94 AE  AC AA 88 54 3A 99 DA 0A  }Íø■FH■®¬ª■T:■Ú
```

Figure 5: The common “GLAMOUR” header of encrypted files.

```
00000300  C5 A8 DD 2E 5C 1E 2D E0  A6 E1 F4 E5 02 86 8E 4E  Å¨Ý.\ -à¦áôå ■■N
0000031B  73 57 59 04 2F 2A 20 70  61 74 68                 sWY /* path
00000320
00000330
00000340
```

Figure 6: Visible plain-text at the end of many encrypted files.

If these files were encrypted with RSA, there would not be such characteristics in the output files. So before we even dive into the binary for clues of the real algorithm, we know it isn’t consistent with the authors’ claim. This is a bit bewildering, because implementing real 4096-bit RSA is simple and would have made it extremely difficult, if not impossible, to produce a working decryptor without paying $300.

4 Exploring the New Thread

In all variants of this malware, each major feature has its own thread. The file encrypting feature is no exception. Below is a flow chart of the fairly simplistic thread, with key points of interest marked individually. On the following page, we describe each key point in greater detail.

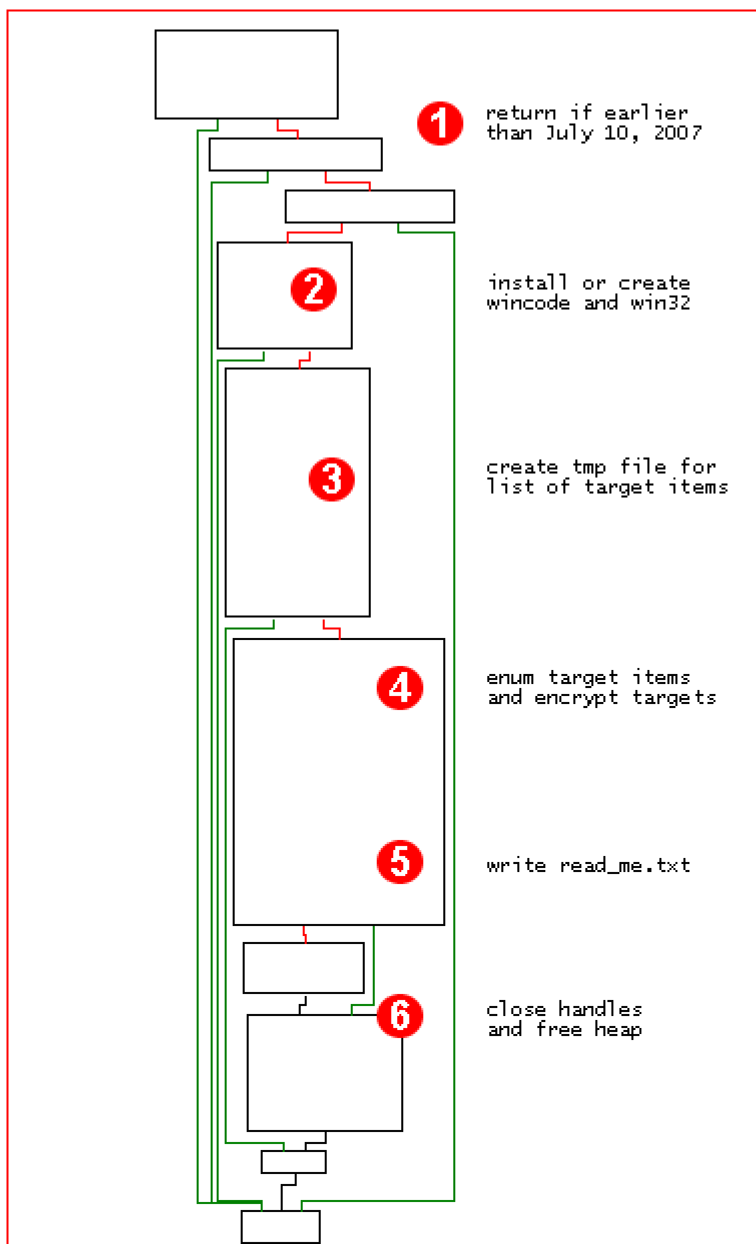


Figure 7: Labeled WinGraph-view of the new file encryption thread.

At point 1, after calling GetSystemTime(), the trojan's file encrypting thread returns if the date is earlier than July 10th. Based on the fact that we got our hands on a copy of this trojan on July 17th, means it wasn't around in the wild too long unnoticed. It also indicates that there is a pattern of development in malware schemes just as there are in any other software vendor's schedules. For example, the compile date of the binary is July 5th; and it is hard-coded to be functional shortly after July 10th but not before. This time was also probably taken to do some testing and to secure a reliable method of distribution. We can also note that the payment deadline in read_me.txt is July 15th, so there was a heavily planned event that occurred and flourished rather quickly. The authors expected to get the most scores within the first 5 days of wild time. This is a rather reasonable timeline given that the email drop boxes are hard-coded into the binary (they must know that it would get reported and shut down before too long).

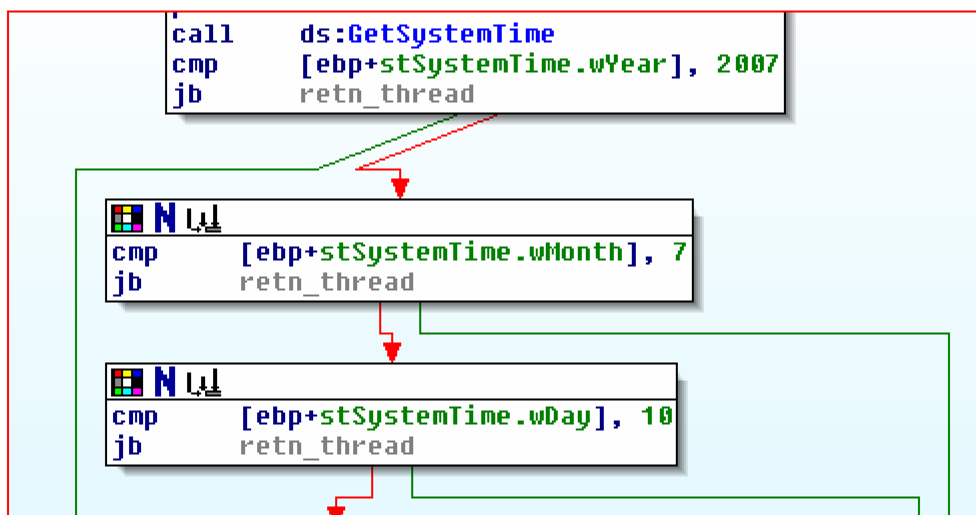


Figure 8: Time-based decision tree for halting the thread.

At point 2, the WinCode and Win32 values are created and added to the registry, or they are retrieved from the registry if they already exist. The WinCode entry is of type DWORD and is generated by mangling the result of GetTickCount() with some proprietary operations. The WinCode is functionally equivalent to a symmetric key and it is required for successful decryption of the files. The Win32 entry is also a DWORD and is generated dynamically, so that it can later be used in a switch() statement to determine which gmail.com drop box it should write in the read_me.txt file.

```
switch(win32) {
      case 1: // use glamourpalace@gmail.com
            break;
      case 2: // use oxyglamour@gmail.com
            break;
      case 3: // use tristanniglam@gmail.com
            break;
      case 4: // use kiloglamour@gmail.com
            break;
      default: // use "" (empty)
            break;
};
```

At point 3, the GetTempPathW() and GetTempFileNameW() functions are used and the result is where the thread will temporarily write its list of potential targets. This is essentially a list of files that it plans to encrypt.

At point 4, two sub functions are called to enumerate the list of potential targets and to engage the encryption routine (or back-off if the file is already encrypted). When searching for targets, it uses GetLogicalDrives() to return a list of drive letters such as A, C, and D. Then it calls GetDriveType() for each letter and skips the entry if it is a CDROM or has no root directory. Otherwise, the recursive search continues, for any files on the drive that have extensions types matching any of the 232 extensions that are hard-coded in the binary's seg001 section.

In the second sub function, the list items are sent to the encryption routine. This is described in greater detail in the next section.

At point 5, the read_me.txt file is written to disk. Its content is comprised of a hard-coded string in the seg000 section; using a %s format specifier for the email address and %d for the WinCode ("personal code"). These are the only two variables in the message.

5 Decryptor Development

We built our decryptor by reverse engineering the encryption algorithm, as preserved in the disassembly, and then by writing the reverse of the encryptor. The encryptor begins by reading the first 7 bytes of the each target item and checking if it matches “GLAMOUR” (not NULL terminated). This is the universal “I’m already encrypted, leave me alone” sign. The code then enters a loop, processing up to 10 MB worth of data in 64K chunks.

The encryption starts with the initialization of a 256-byte key buffer, using a hard-coded source of entropy. This is done with two for() loops that always fill the buffer according to the following format (even across different infected machines):

```
for(int i=0; i<sizeof(key); i++)
      key[i] = i + 24;

for(int i=0; i<sizeof(key); i++)
{
      unsigned char a = key[i];
      ga += (a + entropy[i % 24]);
      int b = (int) ga;
      unsigned char c = a;
      a = key[b];
      key[i] = a;
      key[b] = gb = c;
}
```

After the first for() loop, we’ll have a key buffer that looks like this (break points mark the top and bottom of the loop):

```
14E04756   33C0            XOR EAX,EAX
14E04758   B9 B8FEE014     MOV ECX,14E0FEB8
14E0475D   BE 00010000     MOV ESI,100
14E04762   8AD0            MOV DL,AL
14E04764   80C2 18         ADD DL,18
14E04767   40              INC EAX
14E04768   8811            MOV BYTE PTR DS:[ECX],DL
14E0476A   41              INC ECX
14E0476B   66:3BC6         CMP AX,SI
14E0476E  ^7C F2           JL SHORT 14E04762
14E04770   33C9            XOR ECX,ECX
14E04772   8BC1            MOV EAX,ECX
```

```
Address   Hex dump
14E0FEB8  18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
14E0FEC8  28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
14E0FED8  38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47
14E0FEE8  48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57
14E0FEF8  58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67
14E0FF08  68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77
14E0FF18  78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87
14E0FF28  88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97
14E0FF38  98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7
14E0FF48  A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7
14E0FF58  B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7
14E0FF68  C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7
14E0FF78  D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7
14E0FF88  E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7
14E0FF98  F8 F9 FA FB FC FD FE FF 00 01 02 03 04 05 06 07
14E0FFA8  08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
```

Figure 9: The 256-byte key buffer after initialization with first for() loop.

After the second for() loop, we'll have a key buffer that looks like this (break points mark the top and bottom of the loop):

```
14E04770   33C9             XOR ECX,ECX
14E04772   8BC1             MOV EAX,ECX
14E04774   99               CDQ
14E04775   6A 18            PUSH 18
14E04777   5F               POP EDI
14E04778   F7FF             IDIV EDI
14E0477A   8A99 B8FEE014    MOV BL,BYTE PTR DS:[ECX+14E0FEB8]
14E04780   8A82 80F5E014    MOV AL,BYTE PTR DS:[EDX+14E0F580]
14E04786   02C3             ADD AL,BL
14E04788   0005 A0FEE014    ADD BYTE PTR DS:[14E0FEA0],AL
14E0478E   0FB605 A0FEE014  MOVZX EAX,BYTE PTR DS:[14E0FEA0]
14E04795   8AD3             MOV DL,BL
14E04797   8D80 B8FEE014    LEA EAX,DWORD PTR DS:[EAX+14E0FEB8]
14E0479D   8A18             MOV BL,BYTE PTR DS:[EAX]
14E0479F   8899 B8FEE014    MOV BYTE PTR DS:[ECX+14E0FEB8],BL
14E047A5   41               INC ECX
14E047A6   4E               DEC ESI
14E047A7   8815 A1FEE014    MOV BYTE PTR DS:[14E0FEA1],DL
14E047AD   8810             MOV BYTE PTR DS:[EAX],DL
14E047AF  ^75 C1            JNZ SHORT 14E04772
14E047B1   5F               POP EDI
14E047B2   5E               POP ESI

Address   Hex dump                                          ASCII
14E0FEB8  C3 7D 19 20 DC FE F2 93 70 D0 8B 8A 77 92 2D 4F
14E0FEC8  AD 6D 89 E8 D3 5C A0 41 05 6A 0B A5 98 2C C7 C1
14E0FED8  59 72 35 64 1F 12 83 F7 1E 6C A8 86 5B E1 F0 EA
14E0FEE8  16 AE 5D 42 A3 A4 62 61 E5 3D 47 14 7B C8 7C CE
14E0FEF8  E3 25 DF 1C 08 9D E2 D9 46 6E 22 60 1A 99 B8 E6
14E0FF08  5F AA 6F 80 DE E4 88 1D A6 7A 02 2E DB FA 39 79
14E0FF18  D4 07 4E 76 BB 40 01 0E A9 66 51 0C F5 95 A1 D5
14E0FF28  33 13 F4 96 78 FC 2F 24 28 E7 3E 81 5E 0D 65 53
14E0FF38  D7 B3 B1 C4 57 3F 30 A2 F1 B0 44 6B 04 52 0A E9
14E0FF48  58 8D F9 A7 FF 48 4B F8 B9 94 26 AC CA B2 DA BA
14E0FF58  AB 32 F6 8F C5 BE E0 06 91 00 EE 21 3B 18 8C 50
14E0FF68  15 D8 C6 CD 36 FB 68 B5 ED CB 9A 17 84 B4 BD F3
14E0FF78  C0 9F 09 C2 BF 97 38 BC 27 0F 03 2A D6 63 29 74
14E0FF88  55 10 EF 31 37 EB B7 49 9C B6 11 FD 1B AF 75 56
14E0FF98  4C 45 EC 9E 69 82 3C 54 85 C9 3A 71 87 CC 4D 9B
14E0FFA8  5A 73 DD D1 7E 23 4A CF 43 2B 34 67 8E 7F 90 D2
```

Figure 10: The 256-byte key buffer after initialization with second for() loop.

This initialization is always done before any calls to encryption or decryption functions. It is used as a read/write lookup table to further manipulate the 24 hard-coded entropy bytes, using each byte from the 4 byte WinCode value (multiple times each) as an encryption key and XOR key. This is where things begin to change between different infected machines (because the WinCode value will be different). The two involved functions are shown below, however their corresponding code in the C source code (available from the Introduction) is condensed into one.

```
seg000:14E047F2 ; ¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦
seg000:14E047F2
seg000:14E047F2 _encrypt        proc near
seg000:14E047F2
seg000:14E047F2 arg_0           = dword ptr  8
seg000:14E047F2
seg000:14E047F2                 push    ebx
seg000:14E047F3                 mov     bl, ds:ga
seg000:14E047F9                 push    [esp+arg_0]
seg000:14E047FD                 xor     bl, byte ptr [esp+4+arg_0]
seg000:14E04801                 call    _encrypt_sub
seg000:14E04806                 pop     ecx
seg000:14E04807                 mov     al, bl
```

```
seg000:14E04809                  pop     ebx
seg000:14E0480A                  retn
seg000:14E0480A _encrypt         endp

seg000:14E047B5 ; ¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦
seg000:14E047B5
seg000:14E047B5 _encrypt_sub     proc near
seg000:14E047B5
seg000:14E047B5 arg_0            = byte ptr  4
seg000:14E047B5
seg000:14E047B5                  movzx   eax, [esp+arg_0]
seg000:14E047BA                  mov     dl, ds:key[eax]
seg000:14E047C0                  mov     cl, ds:ga
seg000:14E047C6                  add     cl, dl
seg000:14E047C8                  movzx   ecx, cl
seg000:14E047CB                  lea     ecx, key[ecx]
seg000:14E047D1                  push    ebx
seg000:14E047D2                  mov     bl, [ecx]
seg000:14E047D4                  mov     ds:key[eax], bl
seg000:14E047DA                  mov     [ecx], dl
seg000:14E047DC                  mov     cl, dl
seg000:14E047DE                  add     cl, ds:key[eax]
seg000:14E047E4                  mov     ds:gb, dl
seg000:14E047EA                  mov     ds:ga, cl
seg000:14E047F0                  pop     ebx
seg000:14E047F1                  retn
seg000:14E047F1 _encrypt_sub     endp
```

Unfortunately, we aren't done yet (but close!). After all of this, the initialization routine is called again, executing both for() loops a second time, however in this case, the entropy has been modified with WinCode. Now is when bytes at offset [7-end] from the file to encrypt is sent through the same _encrypt function, and we're done encrypting.

The "GLAMOUR" identification string is written to the beginning of the file, followed by the (now encrypted) bytes from offset [7-end], followed by the unmodified bytes at offset [0-6]. This is why, in The Ransom Note and Deep Mystery, we made the observation that the final several bytes of encrypted files (provided the source was plain text) is readable at the very end – because the first 7 bytes is exempt from encryption.

6 Making Your Registry l33t

Users can protect themselves from these versions by creating a simple registry entry. If the WinCode is set to 31337, then the trojan will abort the file encrypting thread. This will have no effect on the other threads in the event that the trojan executes, but it will keep files from being encrypted. At point 2 in Exploring the New Thread, after querying the registry for (or creating) the WinCode, it is compared with 31337.

```
seg000:14E04F70                 call    _install_reg_wincode
seg000:14E04F75                 call    _install_reg_win32
seg000:14E04F7A                 push    4
seg000:14E04F7C                 lea     eax, [ebp+wincode]
seg000:14E04F7F                 push    offset wincode_buff
seg000:14E04F84                 push    eax
seg000:14E04F85                 call    _format_wincode
seg000:14E04F8A                 add     esp, 0Ch
seg000:14E04F8D                 cmp     [ebp+wincode], 31337
seg000:14E04F94                 jz      retn_thread
```

The entry should be a REG_DWORD named WinCode in the HKLM\SOFTWARE\Microsft\Windows NT\CurrentVersion location, as shown below:

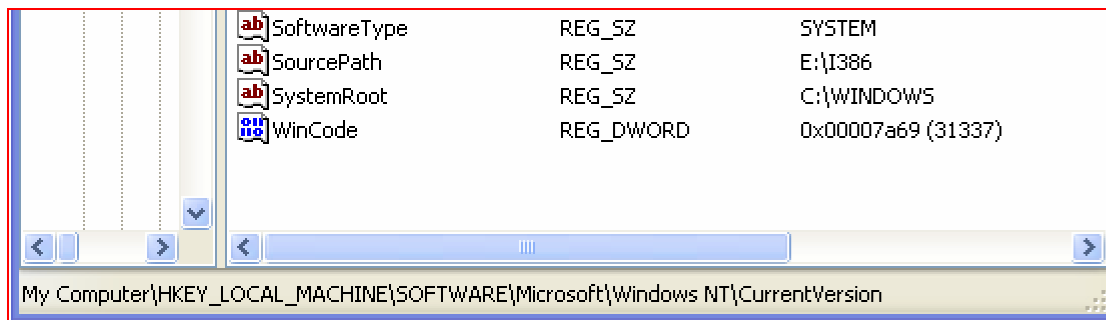


Figure 11: The encryption-halting magic WinCode.

7 References and Tools

Trend Micro – an early analysis and detection summary.
http://www.trendmicro.com/vinfo/grayware/ve_graywareDetails.asp?GNAME=TSPY_KOLLAH.F

Prevx – multiple blog entries and a set of decryptor programs.
http://www.prevx.com/blog.asp?ID=52 (ID=51, ID=31)

Blackmailer: the story of Gpcode – how Kaspersky cracked an older ransomware.
http://www.viruslist.com/en/analysis?pubid=189678219

[Tools] SABRE BinDiff – binary diffing utility, http://www.sabre-security.com/products/bindiff.html
[Tools] IDA Pro – disassembler, http://www.datarescue.com
[Tools] OllyDbg – debugger, http://www.ollydbg.de
[Tools] FlexHex – hex editor, http://www.flexhex.com
[Tools] Snippy – screen acquisition, http://www.bhelpuri.net/Snippy