NOVETTA

Hikit Analysis

Basic Description

Hikit consists of at least two generations of malware that provides basic RAT functionality. The first generation of Hikit (referred to as “Gen 1”) operates as a server and requires an externally exposed network interface in order for an attacker to access the victim machine. The second generation of Hikit (referred to as “Gen 2”) uses the more traditional client model and beacons out to an attacker’s C2 server. While the communication models shifted dramatically between Gen 1 and Gen 2, both generations of Hikit retain the same basic RAT function consisting of remote command shell, file management, network proxy and port forwarding.

Both Gen 1 and Gen 2 sub-families of Hikit consist of a main DLL (referred to as “the DLL”) that contains the RAT functionality; a kernel driver (referred to as “the Driver”) with Gen 2 also employs an additional component: a loader executable. The driver component of Gen 1 and Gen 2 are drastically different in their operation and intent. For the Gen 1 sub-family of Hikit, the driver acts as a NDIS (network) driver that is responsible for interfacing the DLL to the network while preventing a direct WinSock interface from occurring. The Gen 1 Driver listens to network traffic arriving at the local network interface and waits for a specific trigger string. The trigger string varies by Driver and DLL sample. The Gen 2 Driver is a simpler system driver that acts as a rootkit to hide processes, registry keys and network connections associated with Gen 2 activity on the victim’s system.

Gen 2 uses a standard client-server malware model meaning that the malware no longer requires a direct Internet-facing network card, no longer uses a network driver for networking, and provides the ability to network multiple Gen 2 samples behind a firewall with greater ease (from the attacker’s perspective). The Gen 2 sub-family, however, no longer employs network stealth provided by the Gen 1 network driver which exposes the C2 server addresses to analysts.

Each of the Hikit generations contains multiple sub-generations as the author(s) of Hikit have evolved their code over time. There is a noticeable steep improvement over the code base of Gen 1 Hikit family during its 2011 development period. The Gen 2 sub-generations share a similar improvement scale between late 2011 and late 2013.

Evolution

The earliest known Hikit sample dates back to 31 March 2011. Known as the Gen 1.0 sub-generation of Hikit Gen 1, the first known sample of Hikit deviated from the later traditional Gen 1 model. The Gen 1.0 sample was a standalone executable whereas subsequent Gen 1 sub-generations use a DLL running as a service. The Gen 1.0 sample is clearly a work-in-progress. The Gen 1.0 sample, while different than subsequent sub-generations, does still rely on the Driver component and for the most part the structure of the code does not differ much going forward into the Gen 1 evolution.

Less than three weeks after Gen 1.0, the author(s) of Hikit move into Gen 1.1. The notable change is that the Hikit model of using a DLL and driver, which has remained until present day, comes into being. The code matures slightly between Gen 1.0 and Gen 1.1 but the functionality does not change. Both Gen 1.0 and Gen 1.1 use plaintext data transmissions.

Development appears to halt on Gen 1 for 4 months between 20 June 2011 and 23 October 2011 based on a lack of available samples found. During this time the development of Hikit appears to change locations. Gen 1 samples have Program Database (PDB) file strings that identify the file path of the Hikit source code. For Gen 1.0 and Gen 1.1 samples, the file path of the Hikit source code is consistently `h:\JmVodServer\hikit`. Starting with Gen 1.2, the file path switches to `e:\SourceCode\hikit_new`. It is at this time that the functionality of Hikit Gen 1 begins to mature.

In Gen 1.2, the communication between the infected machine and the attacker is encrypted using an XOR mask. A more subtle change is the renaming of the "socks5" command to simply "proxy" within the code. The code within Hikit begins to mature but the overall functionality does not expand beyond the original set of commands found in Gen 1.0. The other remarkable change within the Gen 1.1 to Gen 1.2 development is the way in which the session handshake trigger operates. In Gen 1.0 and Gen 1.1, the DLL instructs the Driver to listen for a specific string (typically a HTTP request string) and responds with another string. In Gen 1.2, the Driver has a hardcoded trigger string (a specific HTTP request string) and the DLL instructs the Driver to inspect a specific HTTP header field for a specific hexadecimal value. This moves Gen 1.2 into more of a username/password authentication scheme whereas previous sub-generations could potentially be accessed by accidental HTTP requests. At the same time, the Driver responds with a specific value with in the `Etag` HTTP header field. This places Gen 1.2 into a more stealth position as a random, non-HTTP compliant response from Gen 1.0 and Gen 1.1 samples is more obvious than a legitimate HTTP response with a specially crafted `Etag` header.

Gen 1.2's development cycle appears to exist between 23 October 2011 and 2 November 2011 with several new samples being found on the Internet having legitimate compile times during this time window. There is, however, evidence that the development of Hikit Gen 1 and Gen 2 overlap by several months. The earliest Gen 2 sample known to exist dates to 28 August 2011, two months before the first known Gen 1.2 sample. The last known Gen 1.2 sample, and by extension, the last known Gen 1 sample, dates to 9 April 2012.

The first known Gen 2 sub-generation, Gen 2.0 Alpha, much like Gen 1.0, represents an early development version of the Gen 2 Hikit sub-family. Gen 2.0 Alpha is a stand-alone Windows console executable that can run as a service executable. Gen 2.0 Alpha supports the same commands as Gen 1.2 with an additional command that returns the infection's configuration information.

On 9 February 2012 the first known sample for Gen 2.0 Beta is compiled by the developer(s) of Hikit. Also a stand-alone console executable like Gen 2.0 Alpha, the Gen 2.0 Beta code changes internally without providing significant functionality improvements with the exception of now the executable uses a device driver to hide network, file, and registry artifacts related to its operation. Both Gen 2.0 Alpha and Gen 2.0 Beta still retain PDB file path information within their binaries. During the development phase of Gen 2.0 Alpha, development of the Gen 2.0 Alpha variants changes locations. First version of the Gen 2.0 Alpha malware, from 28 August 2011, has the PDB path located in `H:\JmVodServer\Matrix_new2` whereas the file path for later Gen 2.0 Alpha and Gen 2.0 Beta binaries has the PDB path in `E:\SourceCode\Matrix_new` which suggests that the source code for both Gen 1 and Gen 2 existed on the same machine and moved at roughly the same time. This may indicate either a single developer or a team (or set of teams) with shared resources.

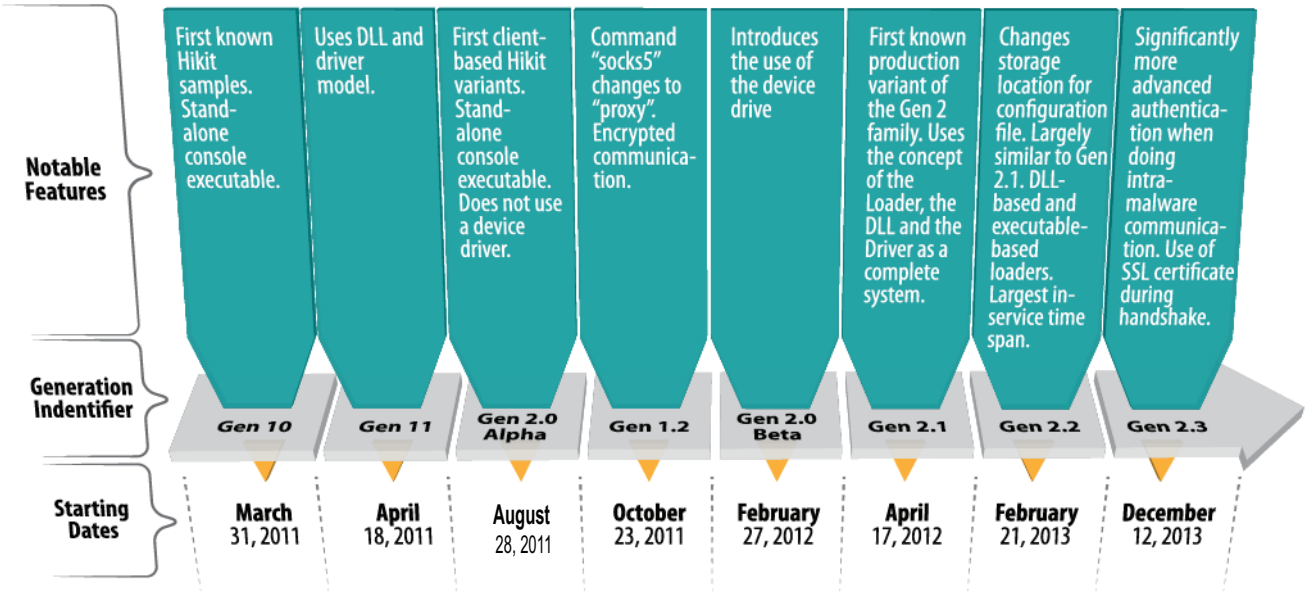
The first known Gen 2.1 binary has a compile date of 17 April 2012. Gen 2.1 represents the first Gen 2 sub-generation to use an executable-based loader, DLL and driver model, a model that all subsequent Gen 2 sub-generations employ. The functionality of the Gen 2.1 sub-generation is the same as the

previous generations with no new commands being introduced. Gen 2.1 is the first sub-generation in the Gen 2 sub-family to introduce 64-bit binaries.

The Gen 2.2 sub-generation appears to have begun on 20 July 2012. Gen 2.2 is notable for altering where the configuration information of the RAT is stored and using both DLL-based and executable-based loaders. Also notable is the fact that the sub-generation spans a significant amount of time with intermittent periods of development. The bulk of the Gen 2.2 samples that have the tell-tale marks of being the product of a builder have a compile date of 26 July 2013, a full year after the first known Gen 2.2 sample. Between 21 July 2012 and 20 February 2013, there are no known Gen 2.2 binaries. The two 20 July 2012 samples have different compile times indicating they were not the product of a builder but rather unique compilations. Between 21 February 2013 and 27 February 2013, there are 4 unique compilation dates for the DLL component with 7 unique, known Gen 2.2 DLLs. The bulk of Gen 2.2 samples have a compile date of 26 July 2013. There are approximately 25 known Gen 2.2 DLLs with the 26 July 2013 compile date. The Gen 2.2 sub-generation appears to exist through at least 19 September 2013.

The last known Gen 2 sub-generation, Gen 2.3, began on 12 December 2013. Gen 2.3 is notable for its use of a legitimate SSL certificate as part of the handshake between the infected machine and the attacker's C2. The DLL will send a legitimate SSL certificate as a means to disrupt heuristic IDS sensors that look for encrypted traffic. Another interesting aspect of the Gen 2.3 sub-generation is that there is no longer a marker to designate the beginning of the embedded configuration. Gen 2.1 and Gen 2.2 uses a specific string to indicate the beginning of the embedded configuration presumably in order to allow the builder to locate the configuration space when constructing a new configuration for the binaries. Gen 2.3, however, uses a specific location instead, requiring the builder to calculate the specific location using the PE/COFF header of the binary. Also, while Gen 2.1 and Gen 2.2 retain the configuration within the DLL component, Gen 2.3 stores the configuration within the loader component. This allows the attackers to configure the loader without having to update the DLL.

The evolution of Hikit is a long and drawn out series of small, incremental development changes. The important take away from the evolution of Hikit is that the developers for Gen 1 appear to have changed in late 2011 and development of Gen 2 has a several month overlap with the development and usage of Gen 1. The following table provides a quick reference to the generational (and sub-generational) designations of Hikit.



Timeline Outlined in Appendix A: HiKit Versions

The Driver

The Driver component for Hikit varies based on the specific Hikit sub-family (Gen 1 or Gen 2). As such it is necessary to describe each in the context of its specific sub-family.

Gen 1 Driver

The Driver component of Gen 1 Hikit variants provides the interface between the victim's network interface card (NIC) and the DLL. The Driver is a NDIS (network) driver that integrates into the victim's network stack. The Driver intercepts any and all network communication that traverses the Windows network stack and potentially removes the data from the network stack under very specific conditions. When the Driver removes data from the network stack, the Driver stores the removed data in local buffers for the Gen 1 DLL to query against. The purpose of this behavior is to allow the DLL to interact with the network without utilizes the WinSock API which could potentially reveal the presence of Hikit.

In order to interact with the Driver, the DLL uses the function `IoDeviceControl` to send commands to the Driver. The Driver registers itself at both `\Device\w7fw` and `\DosDevices\w7fw` thereby allowing the DLL to access the Driver by performing a `CreateFile` request to `\\.\w7fw` or `\\.\Globals\`w7fw in order to obtain a handle to the Driver. The Driver's interface supports the following OIDs:

OID	Function
0x12C828	No-op
0x12C82C	Retrieves bytes from the recv queue.
0x12C830	Add bytes to the xmit queue.
0x12C838	Set key value (the trigger value)
0x12C840	Change mode for current process's channel to 2
0x12C844	Activates channel
0x12C848	Shuts down a channel by flushing all queued packets/data to the network with ACK|FIN set in the flags
0x12C84C	Returns the current mode for a given channel
0x12C850	Get the Driver's version

The Driver will remove data from the network stack only if a new channel is being established. A new channel occurs when the Driver detects a trigger string. The trigger string is typically a short form HTTP request with the following trigger strings found in the wild:

Generation(s)	Trigger String	Authentication Value	Response Value
Gen 1.0, 1.1	GET /password HTTP/1.1\r\n\r\n		.welcome.
Gen 1.2	GET / HTTP/1.1\r\n	75BCD15	HTTP/1.1 200 OK Pragma: no-cache Content-Type: text/html ETag: "{other digits}75BCD15{other digits}:{3 hex digits}" Connection: Keep-Alive
Gen 1.2	POST / HTTP/1.1\r\n	75BCD15	HTTP/1.1 200 OK Pragma: no-cache Content-Type: text/html ETag: "{other digits}75BCD15{other digits}:{3 hex digits}" Connection: Keep-Alive

Up to and including Gen 1.1 Drivers required the DLL to specify the trigger string in addition to the authentication value whereas Gen 1.2 Drivers had the trigger strings hardcoded.

In Gen 1.2, whenever the Driver detects a trigger string, the Driver inspects the rest of the data received for the authentication value. If the token follows the trigger string (there is no specific limitation on how far from the trigger string the password token must be), then the Driver generates a new channel that the DLL will use as the conduit between the DLL and the client.

The Driver appears to be based off the NDIS example source code PassThru. More specifically, the author(s) of the Driver appear to have used the modified version of the PassThru example, PassThruEx, by James Antognini and Thomas Devine from a 2003 blog post[1].

Gen 2 Driver

The Gen 2 sub-family, beginning with Gen 2.0 Beta, employs a Windows device driver ("the Driver") to hide aspects of the DLL's functionality from normal system processes. The Driver is a relatively straightforward piece of software. It does not attempt to obfuscate its functionality from static analysis and it hooks a minimum number of kernel API functions in order to hide different pieces of information. The Driver is based primarily on the open source Agony rootkit[2] and it has evidence of some portions of the code coming directly from a Chinese blog[3].

The Driver expose an IOCTL interface that supports the following OIDs:

OID	Function
0x22C000	Add driver (system module) to hide.
0x22C004	Reveal all hidden items.
0x22C008	Add IP:Port endpoint to hide.
0x22E000	No-Op
0x22FFD0	Remove PID from hidden list.
0x22FFD4	Add PID to list of PIDs to hide.
0x22FFD8	Add service to list of services to hide.
0x22FFE0	Add local port to list of ports to hide.
0x22FFE4	Currently unused. Evidence suggest this was previously a port hiding function, but it is no longer functional.
0x22FFE8	Currently unused. It is unclear the purpose of this function. It takes a string as its argument.
0x22FFEC	Add directory to list of directories to hide.
0x22FFF0	Add registry key to list of registry keys to hide.
0x22FFF4	Add registry key value to list of registry values to hide.
0x22FFFC	Purge all hooks and hidden items ("unhook")

The Driver is capable of hiding processes (by PID, not name), system modules, services, network connections, listening ports, directories (and by extension, files), as well as registry keys and values. In order to hide these items, the Driver hooks various Windows Kernel API calls. The following table maps the items the Driver can hide to the API function that the Driver hooks:

Item	API Function Hooked
Process ID (PID)	`ZwDeviceIoControlFile`
Registry Key	`ZwEnumerateKey`
Registry Value	`ZwEnumerateValueKey`
Directory	`QueryDirectoryFile`

[1] James Antognini and Thomas F. Divine. "Extending the Microsoft PassThru NDIS Intermediate Driver—Parts: Two IP Address Blocking NDIS IM Drivers". December 15, 2003

[2] pudn. "Agony Rootkit code, the stability and can be useful Driver Develop". http://en.pudn.com/downloads74/sourcecode/windows/vxd/detail265112_en.html. 8 April 2007.

[3] CardMagic. "[Reserved] NSI Module Hook: Hiding Port Under Windows Vista". http://forum.eviloctal.com/archiver/tid-29604.html. 8 July 2007.

Item	API Function Hooked
Local Listening Port	`ZwDeviceIoControlFile`
Remote Endpoint	`ZwDeviceIoControlFile`
Loaded Drivers	`ZwQuerySystemInformation`

In order to hide services, the Driver will access the memory of the services.exe process, locate the linked list of services and remove the service entry that the Driver wishes to hide. This is a surprisingly invasive method to obfuscate a process.

Upon activation, the Driver will expose its interface by calling `IoCreateDevice` with the name `\Device\agony` (for Gen 2.0 Beta samples), `\Device\HTTPS` (for Gen 2.1 samples), `\Device\advcachemgr` (for Gen 2.2 samples) or `\Device\diskdump` (for Gen 2.3 samples). The Driver also creates a symbolic link to the device using the same name but under the `\DosDevices\` tree.

For reasons unknown, the authors of the Driver used code from a Chinese blog that details how to hide network connections on Windows Vista and later decided to keep the example IP address within the code.

Functionality and Commands

The Hikit family has supported roughly the same set of commands since the first known samples of Gen 1.0. Gen 2.0 introduced a single command to provide insight into an infected machine's Hikit configuration (something that is not necessary for Gen 1 variants since they are server-based). The RAT supports the following commands:

Command	Introduced	Description
shell	Gen 1.0	Establishes a remote command shell on the victim machine
file	Gen 1.0	File managerment
connect	Gen 1.0	Establishes a tunnel connection (e.g. port forwarding) through another Hikit sample
socks5	Gen 1.0	Establishes a forwarding proxy (*retired in Gen 1.2*)
proxy	Gen 1.2	Establishes a forwarding proxy
information	Gen 2.0 Alpha	Returns the configuration for the Hikit infection
exit	Gen 1.0	Terminates a channel

Command: `shell`

The `shell` command activates a remote shell on the victim's computer. The remote shell function uses the standard pipe redirection method for interfacing a network application (in this case, the DLL) to a hidden command shell.

Command: `file`

The `file` command provides an attacker with a variety of disk access options such as listing directories, changing the current directory, and uploading and downloading files.

Command: `connect`

The `connect` function provides the functionality to allow one Hikit DLL to interface with another DLL of a similar version. The use of this functionality can best be illustrated by considering the fact that the Gen 1 Driver requires an exposed network interface in order for an external attacker to access the Gen 1's RAT function. This would prohibit lateral movement within a victim's network as the bulk of any organization's network infrastructure is not directly exposed to the Internet. By using the connect command, an attacker can instruct the externally exposed Gen 1 DLL to route traffic to a Gen 1 DLL that is behind the firewall, effectively making the externally exposed Gen 1 DLL a local router for Hikit traffic.

Command: `proxy` (Gen 1.2 and later), `socks5` (Gen 1.0 and 1.1)

The `proxy` (or `socks5`) command allows an attacker to utilize a Hikit-infected machine as a proxy.

Command: `information`

Gen 2 samples rely on a configuration in order to know where the C2 server exists along with other operational aspects such as the name of its service and operational times. This information is important for the attacker to have access to in order to determine if any aspect of the configuration is out of date (thus requiring a new variant of the Gen 2 binary to be configured and deployed). The `information` command returns to the attacker the complete configuration and current state of the Gen 2 malware.

Command: `exit`

As the name implies, the `exit` command causes the DLL to discontinue the current connection.

Hikit Core Analysis

With the Gen 1 sub-family using a server model and the Gen 2 sub-family using the client model, understanding how each of the DLL components of the sub-families works is best done, as with the Driver above, in the context of the specific sub-family.

Gen 1 Analysis

As noted previously in this report, the Gen 1 sub-family has several sub-generations but overall the functionality of the Gen 1 sub-family has remained constant. With the exception of Gen 1.0, the functionality of Gen 1 comes from the DLL component (Gen 1.0 uses a stand-alone executable to achieve the same results). The DLL operates as a service, requiring an attacker is install the DLL as a service at some point prior to activation. The DLL contains only two exports: `DllEntryPoint` and `DllRegisterServer`. Ultimately, both exports generate a new thread of the same function ("`mainThread`"). The difference between the two exports is that `DllRegisterServer` can take an optional command line argument of the letter "u" which will instruct the main thread to uninstall the Gen 1 system from a victim's computer. If the uninstall argument exists, `mainThread` will simply remove the Driver from the victim's machine and terminate. The authors of Gen 1 used freely available source code found online for their removal function.[4]

[4] PCAUSA. "Programmatically Installing NDIS Protocol Drivers" http://www.ndis.com/ndis-general/ndisinstall/programinstall.htm. 2 December 2013.

When the DLL activates, either by a call to `DllEntryPoint` or by calling `DllRegisterServer` without the u parameter, `mainThread` begins by verifying the version of the Driver installed on the victim's machine. This requires sending OID 0x12C850 to the Driver and comparing the resulting 32-bit value with the required driver version. If the version is incorrect (i.e. it doesn't match the specified version), the DLL installs the version of the Driver found within the DLL's resource section (under the `BIN` resource tree).

With the Driver version verified (or forcibly corrected by installing the appropriate Driver), the DLL will instruct the Driver to use a specified string (for Gen 1.0 and Gen 1.1 samples) or a DWORD (for Gen 1.2 samples) as the acknowledgment value to send to a connecting client who requests the appropriate URL. The DLL again checks the version of the Driver and, in some versions of the DLL, will print a message indicating the version of the Driver installed and report the "Transate version" (the word translate is misspelled within the binary). It appears that the Driver and the communication protocol version do not necessarily have to match exactly, allowing the possibility that the Driver and the DLL could be compiled at separate times. If the Driver version is less than the "Transate" version (indicating that the Driver is a version too old to support the necessary communication protocols), the DLL will, in some version of the DLL, print out a line to the screen indicating the `DRIVER_MIN_VERSION` required along with the current Driver version. Following this, the DLL will then attempt to install the correct version of the Driver prior to terminating. It is unclear why this code exists given that the DLL will check the Driver version and correct the Driver if necessary prior to reaching the portion of the code that reports the `DRIVER_MIN_VERSION`. It is possible that the second Driver version check is a last ditch effort to ensure the correct Driver is installed.

The DLL enters an infinite loop where the DLL waits for the Driver to report a new channel exists. A channel represents an established connection between the Driver and an external party that has provided the proper initial request and, for Gen 1.2 variants, provided the proper authentication value. When the Driver establishes a new channel, the DLL generates a runtime data structure before generating a new thread ("`HikitThreadFunc`") which will service any request coming from the new channel. This allows the DLL to service multiple channels at one time.

The `HikitThreadFunc` function is, at its core, a simple wait and respond loop. The function begins by transmitting a Hikit command prompt to the client (`Hikit>`) before settling into an infinite loop of

- Read data from channel (wait until data is available)
- [For Gen 1.2] Decrypt the packet header
- Verify the packet header to ensure the communication version is correct and the payload data size is non-zero
- Read the remainder of the packet (e.g. the payload portion)
- If the packet type field (dwPacketType) is zero, send the payload section to the command processor.
- Send the Hikit prompt

The communication scheme between the DLL and the client consists of a 20 to 24 byte header (for Gen 1.0 and Gen 1.1) or a 28 byte header (for Gen 1.2) followed by an optional payload. The format of the Gen 1.0 and Gen 1.1 header is as follows:

```
struct PacketHeader
{
      char magic[5];
      char zeros[3];
      DWORD dwHikitVersion;
      DWORD dwCmdType;
      DWORD dwLocale;      // omitted in some Gen 1.0 variants
      DWORD dwPayloadSize;
};
```

While the Gen 1.2 header is:

```
struct PacketHeader
{
      DWORD key;
      DWORD dwHikitVersion;
      DWORD dwPacketType;
      DWORD dwLocale;
      DWORD dwCodePage;
      DWORD dwPayloadSize;
};
```

For Gen 1.0 and Gen 1.1 samples, the magic field contains the string "`.. ..`" (two dots followed by a space then two more dots). Whereas the key field in Gen 1.2 samples contains a 32-bit value that represents the XOR key for the remainder of the PacketHeader and any additional payload data. The XOR scheme works on 32-bit chunks of data where each 32-bit chunk of data is XOR'd against the key value.

Version checking is important in all Gen 1 variants. The `dwHikitVersion` field allows the client and the DLL to ensure that they have a compatible communication scheme in place prior to executing commands.

Gen 1 samples have a particular interest in the victim's locale language preferences. While it is typical for most RATs that provide remote shells to simply pass data unfiltered from client to server and server to client without regard to code pages, Gen 1 samples take special care to record the code page and locale information in each and every packet header that traverses the divide between client and server and server and client. This could indicate that the authors of Gen 1 understood from an early stage in the development of Gen 1 that they would be attacking computer systems with different locales and code pages.

Gen 2 Analysis

The Gen 2 sub-family, like Gen 1.2, uses a DLL for the core of its RAT functionality. In order for the DLL to load, Gen 2 (starting with Gen 2.1) uses a loader application (referred to simply as "the Loader"). The Loader comes in the form of a standard executable image or a DLL image. Despite the different models, both variants of the Loader load the embedded DLL in the exact same way. The only difference between the executable and DLL versions of the Loader comes in how they handle the initialization of the embedded DLL.

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason
{
  ImageLoaderData *v4; // esi@6

  if ( fdwReason )
  {
    if ( fdwReason == 1 )
    {
      hInstDll = hinstDLL;
      if ( !pEmbeddedImage )
      {
        pEmbeddedImage = LoadEmbeddedImage(102u);
        SetModuleHandle(hinstDLL);
        Activate_StartServer();
        return 1;
      }
    }
  }
  else if ( pEmbeddedImage )
  {
    hInstDll = hinstDLL;
    Activate_StopServer();
    v4 = pEmbeddedImage;
    if ( pEmbeddedImage )
    {
      ImageLoaderData::UnloadDll(pEmbeddedImage);
      operator delete(v4);
    }
  }
  return 1;
}
```

```
int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPre
{
  ImageLoaderData *v4; // esi@4

  hInstDll = hInstance;
  if ( !pEmbeddedImage )
    pEmbeddedImage = LoadEmbeddedImage(102);
  SetModuleHandle(hInstance);
  if ( pEmbeddedImage )
  {
    Activate_MatrixMain();
    v4 = pEmbeddedImage;
    if ( pEmbeddedImage )
    {
      ImageLoaderData::UnloadDll(pEmbeddedImage);
      operator delete(v4);
    }
  }
  return 0;
}
```

Figure 1: DLL (left) and executable (right) Loader startup routines

Figure 1 provides a side by side comparison of the startup routines for the executable and DLL Loaders. Both versions of the Loader begin by loading the embedded DLL from the Loader's resources (item 102 under the Group Icons resource tree), decrypting and decompressing the image into memory, then manually loading the DLL into memory using a custom loading routine. The function `LoadEmbeddedImage`, as seen in part in Figure 2, is responsible for this operation.

```
v28 = 0;
v1 = FindResourceW(hInstDll, (unsigned __int16)dwItemID, RT_GROUP_ICON);
if ( v1
  && (v2 = LoadResource(hInstDll, (HRSRC)v1)) != 0
  && (v3 = (BYTE *)LockResource(v2)) != 0
  && (v4 = LookupIconIdFromDirectory(v3, 1), v5 = FindResourceW(hInstDll, v4, 3), (v6 = (HRSRC)v5) != 0)
  && (v7 = (unsigned __int8 *)LoadResource(hInstDll, (HRSRC)v5), (resourceImage = v7) != 0)
  && LockResource(v7)
  && (resourceSize = SizeofResource(hInstDll, v6),
      searchKeyLength = strlen((const char *)DecodeString(&zzzzzzzzz)) + 1,
      searchKey = DecodeString(&zzzzzzzzz),
      targetOffset = memfind(resourceImage, resourceSize, searchKey, searchKeyLength - 1),
      targetOffset >= 0)
  && (targetStartOffset = strlen((const char *)DecodeString(&zzzzzzzzz)) + targetOffset,
      resourceSize - targetStartOffset >= 0x10)
  && (imageSize = *(_DWORD *)&resourceImage[targetStartOffset],
      pImageHeader = (ImageHeader *)&resourceImage[targetStartOffset],
      imageSize == resourceSize - (targetStartOffset + 15))
  && (v16 = pImageHeader->dwImageSizeDecompressed, v16 >= 0)
  && v16 <= 0xA00000 )
{
  v17 = imageSize;
  v18 = GetProcessHeap();
  v19 = HeapAlloc(v18, 8u, v17);
  if ( v19 )
  {
    qmemcpy(v19, &pImageHeader->startOfEncodedImageData, pImageHeader->dwImageEncodedSize);
    decodeBuffer(v19, pImageHeader->dwImageEncodedSize, pImageHeader->EncodingKey);
    v20 = pImageHeader->dwImageSizeDecompressed;
    v21 = GetProcessHeap();
    pDllImage = (unsigned __int8 *)HeapAlloc(v21, 8u, v20);
    if ( pDllImage )
    {
      dwItemID = pImageHeader->dwImageSizeDecompressed;
      lzo_decompress((unsigned __int8 *)v19, pImageHeader->dwImageEncodedSize, pDllImage, &dwItemID, 0);
      if ( dwItemID == pImageHeader->dwImageSizeDecompressed )
      {
        v23 = (ImageLoaderData *)operator new(0x18u);
        pImageData = v23 ? ImageLoaderData::ctor(v23) : 0;
        v28 = pImageData;
        if ( !ImageLoaderData::LoadDll(pImageData, pDllImage, dwItemID) )
        {
          if ( pImageData )
          {
            ImageLoaderData::UnloadDll(pImageData);
            operator delete(pImageData);
          }
          v28 = 0;
        }
      }
      v25 = GetProcessHeap();
      HeapFree(v25, 0, pDllImage);
    }
    v26 = GetProcessHeap();
```

Figure 2: LoadEmbeddedImage function snippet

The Loader obfuscates many strings by using a simple XOR encoding scheme. Decryption of encoded strings consists of taking the first value of the string as the XOR key, XOR'ing all subsequent bytes until the result of the XOR returns 0. The decoding of the encoded strings is handled by the DecodeString function.

The Loader stores the embedded DLL within a Group Icon resource within a legitimate icon image. In order to locate the embedded DLL, `LoadEmbeddedImage` will use the `DecodeString` function to decrypt the delimiter string (which is typically `zzzzzzzzzz` or `yyyyyyyyyy`) and then search the icon's resource memory for the delimiter string. Once located, `LoadEmbeddedImage` will use the first 12 bytes immediately after the string as the information structure about the embedded DLL. The structure (seen below) defines the size of the embedded DLL within the icon's resource memory, the size of the

DLL after it is decompressed and a 4-byte XOR key that `LoadEmbeddedImage` must use to decode the embedded DLL prior to decompression.

```
struct ImageHeader
{
   DWORD dwImageEncodedSize;
   DWORD dwImageSizeDecompressed;
   DWORD EncodingKey;
};
```

`LoadEmbeddedImage` copies the compressed embedded DLL into a newly allocated heap buffer and then calls the function `decodeBuffer` (using the `EncodingKey` value) to decrypt the embedded DLL. Another heap buffer is allocated with a size equal to the value of `dwImageSizeDecompressed`. The decompression buffer along with the now decoded compressed buffer are given to `lzo_decompress` which decompresses the compressed image using the LZO1X algorithm[5].

With the embedded DLL now decompressed into a heap buffer, `LoadEmbeddedImage` calls `ImageLoaderData::LoadDll` to manually load the DLL into memory. `ImageLoaderData::LoadDll` interprets the PE/COFF header of the DLL image, loads the image into the appropriate memory configuration, performs the necessary relocation operations, and calls the `DllMain` (DLL's entry point) function.

After loading the embedded DLL image into memory, the Loader will either call the DLL's `StartServer` or `MatrixMain` function depending on the type of Loader. The standalone Loaders use the `MatrixMain` function while the DLL Loaders will call the `StartServer` function. The Loaders, upon unloading, will call the StopServer function in order to shut down the embedded DLL.

The Gen 2 DLL exposes five exported functions (besides the `DllEntryPoint`/`DllMain`).

Export Name	Description
DllRegisterServer	If the Gen 2 RAT is running, waits for the RAT to shut down before returning.
MatrixMain	Activates the Gen 2 RAT (called from a stand-alone Loader)
SetModuleHandle	The given parameter becomes the new module handle for the RAT.
StartServer	Activates the Gen 2 RAT (called from a DLL Loader)
StopServer	Stops the Gen 2 RAT (called from a DLL Loader)

`MatrixMain` and `StartServer` both ultimately generate a new thread (using the POSIX API function `beginthreadex` instead of the more common `CreateThread`) that contains the main loop of the Gen 2 RAT functionality. `MatrixMain`, however, has added functionality. The prototype for MatrixMain is as follows:

```
int MatrixMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR
                    lpCmdLine, int nShowCmd))
```

[5] Markus F.X.J. Oberhumer, "LZO real-time data compression library" http://www.oberhumer.com/opensource/lzo/. 29 June 2014.

where Arguments parameter can be:

Arguments string	Purpose
`test {DWORD identifier (IP?)} {listening Port} [C2 address] [C2 port]`	Overrides the current configuration with the given settings. The C2 arguments are optional.
`i`	Installs trojan service
`u`	Uninstalls the trojan service
`s`	Sets the SHOW flag for the service to instruct the Driver to reveal the service.
`h`	Sets the HIDE flag for the service to instruct the Driver to hide the service.
`q`	Sets the STOP flag for the RAT.

If the `i` parameter is given, the DLL will install itself as a service on the victim's machine. The DLL will create a new service (e.g. "Network DDE Service") and assign itself as the executable for the service.

The DLL's RAT functionality provides basic features such as network port forwarding (proxying), file transfer, and remote command shell. The RAT functionality provides an attacker with the ability to establish a phantom network within a victim's infrastructure by having individual instances of Gen 2 DLL listen for incoming connections on local ports (presumably, NAT'd ports) and accept commands from the inbound connection. This allows an attacker to establish several Gen2 infections within a victim's infrastructure and if outbound connectivity is prohibited for any of the infected machines, the attacker can route commands to the pseudo-isolated infections through accessible infected machines providing a high level of persistence to the malware. Each Gen 2 infection can support up to 10 listening ports.

The communication between the Gen 2 malware and the C2 (or other Gen 2 malware, in the case of the internal routing functionality) is encrypted using a simple DWORD XOR scheme. Each communication burst (either between the malware and the C2 or the malware and neighboring malware) begins with a 24-byte header identical to the header found in Gen 1.2. Immediately following the header is the type-specific (as indicated by the `dwPayloadType` field) payload data. Note that the `dwXORKey` value is NOT encoded with the XOR value, but rather is the value that is used for encoding the header and payload.

Each DLL includes a hardcoded, default configuration. At the time that the RAT functionality activates, the DLL will drop the current configuration to disk. If the configuration file already exists, then the RAT will use the file version of the configuration over the default configuration. The configuration data structure (seen below) doubles as a current state record for some aspects of the communication subsystem of the DLL. When stored on disk, the configuration is preceded by a GUID value (16 bytes) that represents the unique identifier for the specific infection. The configuration is XOR encoded using the first 4 bytes (as a DWORD) of the GUID.

```
struct Config
{
   WCHAR wszComment[32];
   C2ConfigInfo arrC2s[2];
   ListeningPortConfig ListeningPorts[10];
   int dwStartTime;
   int dwEndTime;
   __int16 Flags;
   SYSTEMTIME sleepUntil;
   __int16 unused_align2_2;
   int fRunHidden;
};

struct C2ConfigInfo
{
   WCHAR wszAddress[32];
   __int16 wPort;
   __int16 unused_align2;
   int fValidC2;
};

struct ListeningPortConfig
{
   unsigned __int16 wPort;
   unsigned __int16 unused_align2;
   int fReady;
   SOCKET hSocket;
   HANDLE hEvent;
   HANDLE hListenerThread;
};
```

In order to provide some level of stealth, the RAT will install a rootkit on 32-bit versions of Windows. The DLL contains a device driver image embedded within an encoded buffer which the RAT functionality code will extract to the `%TEMP%` directory (after XOR'ing the buffer with `0x76`). To activate the rootkit, the RAT functionality code creates a service with the driver in the `%TEMP%` directory as the executable for the service. The RAT functionality code then activates the service and opens a handle to device driver's interface (e.g. `\Globals\HTTPS`). With the handle open to the rootkit driver, the RAT functionality code deletes the service in order to reduce the visible footprint of the new driver. To further reduce the footprint of the driver, the RAT functionality code also uses the cloaking functionality of the rootkit to hide the DLL's PID, any references to the GUID `{4AE26357-79A3-466D-A6D9-FC38BFB67DEA}`, the DLL's service names (e.g. "NetDDESrv" and "Network DDE Service") and the service entry as well. Additionally, the code also attempts to hide a service named "Hitx".

Support Software

In addition to the main Hikit malware, there are at least two examples of support programs that belong to the Hikit family. Samples b04de6c417b6f8836e3f2d8822be2e68f4f9722b and 7c4da9deff3e5c7611b9e1bd67d0e74aa7d2d0f6 are examples of Gen 1.0 and Gen 1.2 operator consoles. The console is a text based application that takes a Gen 1.0 or Gen 1.2 infection's IP address and proceeds to connect and authenticate with the infected server. Once connected, the operator has the basic Hikit functionalities available to them via commands such as `file` and `shell`.

Detection

Detecting Hikit variants on disk and in memory is possible using the following YARA signature developed by Symantec:

```
rule hikit
{
      strings:
            $hikit_pdb1 = /(H|h)ikit_/
            $hikit_pdb2 = "hikit\\"
            $hikit_str3 = "hikit>" wide

            $driver = "w7fw.sys" wide
            $device = "\\Device\\w7fw" wide
            $global = "Global\\%s__HIDE__" wide nocase
            $backdr = "backdoor closed" wide
            $hidden = "*****Hidden:" wide

      condition:
            (1 of ($hikit_pdb1,$hikit_pdb2,$hikit_str3)) and ($driver or
$device or $global or $backdr or $hidden)
}

rule hikit2
{
      strings:
       $magic1 = {8C 24 24 43 2B 2B 22 13 13 13 00}
       $magic2 = {8A 25 25 42 28 28 20 1C 1C 1C 15 15 15 0E 0E 0E 05 05 05
00}
      condition:
       $magic1 and $magic2
}

rule hidkit
{
      strings:
       $a = "---HIDE"
       $b = "hide---port = %d"
      condition:
       uint16(0)==0x5A4D and uint32(uint32(0x3c))==0x00004550 and $a and $b
}
```

Detecting nominal Gen 1.2 and later network activity is problematic given the nature of the communication structure. The encrypted nature of the nominal Gen 1.2 and later network traffic makes a signature difficult. Snort signature 30948 may detect some Hikit based network traffic for only Gen 1.0 and Gen 1.1.

From a system objects perspective, Gen 2 samples produce up to three named events. The event names change per infection, but have a common format. The following three strings represent the known mutex strings for Gen 2 samples:

```
Global\%s__SHOW__
Global\%s__HIDE__
Global\%s__STOP__
```

where the %s format variable is replaced with a UUID value string specific to the infected machine.

Appendix A: HiKit Versions

Generation Identifier	Starting Date	Notable Features
Gen 1.0	31 March 2011	First known Hikit samples. Stand-alone console executable.
Gen 1.1	18 April 2011	Uses DLL and driver model.
Gen 2.0 Alpha	28 August 2011	First client-based Hikit variants. Stand-alone console executable. Does not use a device driver. Encrypted communication.
Gen 1.2	23 October 2011	Command “socks5” changes to “proxy”. Encrypted communication.
Gen 2.0 Beta	27 February 2012	Introduces the use of the device driver.
Gen 2.1	17 April, 2012	First known production variant of the Gen 2 family. Uses the concept of the Loader, the DLL and the Driver as a complete system.
Gen 2.2	21 February 2013	Changes storage location for configuration file. Largely similar to Gen 2.1. DLL-based and executable-based loaders. Largest in-service time span.
Gen 2.3	12 December 2013	Significantly more advanced authentication when doing intra-malware communication. Use of SSL certificate during handshake.