WINNTI ANALYSIS

As part of Operation SMN, Novetta analyzed recent versions of the Winnti malware. The samples, compiled from mid- to late 2014, exhibited minimal functional changes over the previous generations Kaspersky reported in 2013.[1] What is of note, however, is the increased scrutiny found within the Winnti dropper component that attempts to frustrate analysis of the malware.

Based on multiple active compromises by the Axiom threat group, Novetta was able to capture and analyze new Winnti malware samples. It should be noted that operators of Winnti that were observed by Novetta leveraged existing Axiom-specific malware infections (Hikit) to move laterally and install Winnti in the furtherance of their objectives. It is with high confidence that we assess the operators of Winnti in these monitored environments were not the same actors who originally installed and leveraged Hikit.

This report will focus on three different aspects of the Winnti malware: the start-up sequence of Winnti from the initial infection to steady state, the basics of the Winnti malware, and the command and control (C2) communication protocol.

FROM INSTALLATION TO EXECUTION

The installation process of Winnti by means of the dropper has changed very little since the Winnti version 1.1 (as defined by Kaspersky) droppers of July 2012. The samples Novetta obtained from the active Axiom infection were compiled in mid- to late 2014 and represent what Novetta is referring to as version 3.0 of the Winnti lineage (in order to prevent muddying the versioning scheme Kaspersky has already established).

There are four distinct components within the Winnti malware's installation to execution cycle: a dropper, a service, an engine, and a worker. The installation that Novetta observed of Winnti on a victim's machine requires multiple steps and depends on the dropper, service, and the loader component in order to accomplish the steps. After a successful infection, the activation of Winnti on a victim's machine requires multiple steps as well as coordination between the service, engine, and worker components. The complexity of the installation and activation processes is significant and more involved than typical malware installation and activation procedures. This additional complexity ultimately does not seem to serve a significant purpose other than to perhaps frustrate analysis by defenders.

[1] *Kaspersky. "Winnti: More than just a game". http://kasperskycontenthub.com/wp- content/uploads/sites/43/vlpdfs/winnti-more-than-just-a-game-130410.pdf. 10 April 2014.*

INSTALLATION PHASE 1: DROPPER ACTIVITIES

The dropper is, as the name implies, the component responsible for dropping the Winnti malware on a victim's machine. The dropper performs the following activities (with version- specific annotation):

1. [version 3.0] The dropper verifies the existence of a single parameter on the command line and terminates if the parameter is not found. The dropper later uses this parameter as a decryption key.

2. The dropper loads `CreateProcessA` via a call to `GetProcAddress`.

3. The dropper extracts an embedded data blob, decrypts the data blob, and decompresses the data blob into a heap buffer.

The data blob in step 3 begins with a header structure that describes key attributes of the data blob. The format of the header is as follows:

OFFSET	SIZE	DESCRIPTION
0x0	8	Magic bytes "Tunnel\0\0"
0x8	2	Unknown
0xA	4	Size of the blob after decompression
0xE	4	Size of the blob prior to decompression (current size in memory)

In order to decrypt the data blob, the dropper will iterate over each of the bytes that follows the header (up to the value specified in 0xE offset of the header), XOR the bytes by 0x36, and then perform a nibble swap, ZLib-compressed data immediately follows the header. The dropper will allocate a heap buffer (with a size specified by the value in offset 0xA of the header) and call the ZLib inflate function.

The decompressed data blob contains a second header, and the blob's header consists of the following two entries:

OFFSET	SIZE	DESCRIPTION
0x0	*4*	*Offset of worker component's image (generally 0 and unused)*
0x4	*4*	*Offset of server component's image*

4. The dropper uses the value in offset 0x4 of the decompressed data blob's header to determine the size of the worker component's size and writes that many bytes of the decompressed data blob, starting after the header, to a file within `%TEMP%`. The file has a random name, but the extension is always `.tmp`.

5. [version 3.0] The remaining bytes within the decompressed data blob are decrypted using what appears to be DES encryption and the key the attacker provided via the first argument on the command line.

6. The dropper generates another randomly named file within `%TEMP%` with the file extension of `.tmp` with the remaining bytes within the decompressed data blob. This file becomes the service component.

The decryption of the service component is a new feature within the version 3.0 lineage of Winnti and provides the benefit of preventing defenders from attempting to run the dropper for analysis purposes.

7. [version 3.0] The presence of the `MZ` magic bytes in the first two bytes of the service component are verified to ensure that the file was properly decrypted (and by extension, to verify that the supplied password was correct). If the MZ bytes are not found, the dropper quietly terminates.

8. [version 3.0] The filenames and paths of the service and worker components are appended to the end of the worker component's file in a structure 520 byte array. The irst 260 bytes contain the filename and path of the worker component, and the last 260 bytes contain the filename and path of the service component. The entire 520 byte array is encrypted by XOR'ing each byte with 0x99.

9. The service component's file is scanned for the tag `Coopre`. If this tag is located, the dropper decrypts the configuration data blob attached to the end of the dropper's executable and appends the data to memory immediately followed the `Coopre` magic byte. The dropper will then append the filenames and paths of the service and worker components to the end of the worker component's file in a structure 520 byte array. The first 260 bytes contain the filename and path of the worker component, and the last 260 bytes contain the filename and path of the service component. The entire 520 byte array is encrypted by XOR'ing each byte with 0x99.

10. The dropper instructs the service component to complete the installation by using rundll32 to activate the service component's `Install` [pre-version 3.0] or `DlgProc` [version 3.0] function.

The service component is self-installing when an attacker (or the dropper) activates the `Install` or `DlgProc` export functions. `Install` or `DlgProc` requires the full name and path of the dropper component. The service component can locate the worker component based on the appended filename and path strings located at the end of itself. Once the dropper calls `CreateProcessA` to activate `rundll32`, the dropper's task is complete, and it quietly terminates. At this point, it is up to the service component to continue to install and, eventually, to activate the Winnti malware.

INSTALLATION PHASE 2: SERVICE ACTIVITIES

The service component is at its core an unsophisticated scaffold whose job is to activate the engine component. The distinction between pre-version 3.0 Winnti variants and version 3.0 variants is most evident in the versions' service components. As a result of the larger difference in procedure between the pre-version 3.0 and version 3.0 variants, the discussion on the Installation Phase 2 sequence will focus on only the version 3.0 service component.

The service component has only two functions: activate the installer functionality of the engine component or respond as a service DLL and activate the engine component's malware start-up routines. Both modes of the service component have a common initialization sequence:

1. `DllMain`, upon activation, manually loads the engine binary into memory.

2. The exports from the engine component (`Install`, `DeleteF`, and `Workman`) are loaded into a memory structure.

The engine component exists as a data blob within the service component. The executable image of the engine component contains a stripped-down PE/COFF header that is missing several key attributes, such as the MZ and PE magic bytes as well as the CPU architecture identifier. These missing pieces require the service component to manually load the engine component's executable image into memory. This includes allocating the appropriate memory, loading the sections appropriately, applying location fix-ups, and loading the necessary imports. *The advantage of going to this much effort from an attacker's perspective is that the engine, which is responsible for loading the worker, never touches the disk, thus depriving analysts of a necessary component for analysis when using file-based artifact collection techniques.*

3. The service component determines if the host binary responsible for the service component is either svchost.exe (if run as a service) or rundll32.dll (if loaded pseudo manually or from the dropper).

4. If neither host executable is found, the `Install` function of the engine is activated. Otherwise, the `DllMain` function returns with a success status.

5. Within the `DlgProc` function, the service will verify the non-NULL status of the DeleteF function pointer. If the function pointer is non-NULL, the `DeleteF` function (of the engine) is called.

6. The `DlgProc` function next verifies that the `Install` function pointer is non-NULL and, if the pointer is indeed non-NULL, calls the engine's `Install` function after supplying the path of the service executable.

Once the service component passes control to the engine component's `Install` function, the service component becomes little more than a placeholder. The service component will remain active only so long as the `Install` function is active. The engine component then becomes the active component and will conclude the installation process.

INSTALLATION PHASE 3: ENGINE ACTIVITIES

The engine component exists only in memory and is completely dependent on the service component remaining active in order to have a process space in which to operate. The installation functionality for the version 3.0 Winnti malware variants exists within the `Install` export. When activated, the `Install` export requires a full filename and path back to the service binary.

1. The `Install` function checks if the process responsible for calling the `Install` function is explorer.exe. If so, the function uses a variant of the sysprep User Account Control (UAC) bypass.[2]

2. The current privilege level of the process is verified to be suitable for installing additional binaries. The test consists of determining if the operating system is newer than Windows XP and if the process has administrative privileges. If these conditions are met, the dropper is called again from explorer.exe.

The authors went to great pains to ensure that the installation process had a suitable chance of performing its function both undetected (by using a UAC bypass) and with suitable privileges. The use of the sysprep UAC bypass and the verification of process privileges indicate that the authors were at least aware of the changing security environment in later versions of Windows and were adapting to ensure the successful propagation of their tools on a victim's machine.

[2] *K. Kleissner. "UAC Bypass for Windows 7 RTM, SP1/ Windows 8 DP, CP all 32-bit for admin with default UAC settings". http://download.pureftpd.org/pub/misc/UAC.cpp. Accessed 1 December 2014*

3. The configuration information implanted at the end of the service component is extracted and decrypted.

The configuration information for the infection typically exists at the very end of the service configuration. The last DWORD of the file contains an offset from the end of the file to the beginning of the configuration. The structure of the configuration is as follows:

```
struct ScheduleEntry
{
   __int16 wDayOfWeek;
   __int16 wStartTime;
   __int16 wEndTime;
};


struct Config
{
   char szC2[100];
   char szCampaignID1[32];
   char szCampaignID2[32];
   char szCampaignIDNumber[24];
   DWORD unknown0;
   DWORD unknown1;
   DWORD unknown2;
   DWORD dwCommMode;
   DWORD dwProxyType;
   char szProxyServer[32];
   char szProxyUser[32];
   char szProxyPassword[32];
   ScheduleEntry arrActivePeriods[7];
   DWORD iReconnectTime;
};
```

The configuration is encrypted using a rolling XOR with the first byte of the XOR being 0x99. The XOR increments one byte at a time until a wraparound occurs at 0xFF.

4. The files `%SYSDIR%\wmtsk.dll` and `%SYSDIR%\wmm2.dat` are checked to ensure that they do not exist and that they are not directories. If either condition is not met, an INI file named `%SYSDIR%\otfkyt.dat` is generated with the `UPDATE` section containing `File`={path to service component DLL} and the process terminates.

5. A global event named `Global\ 5164FDA21542C0EB638BA110F9F3ADAF` establishes a poor man's mutex, indicating that the installation process is currently ongoing. To give other processes time to read the event check, a 12-second delay is introduced before continuing the installation process.

6. The service component is copied from the `%TEMP%` directory to `%SYSDIR%\wmtsk.dll`, and the worker component is copied from the `%TEMP%` directory to `%SYSDIR%\wmm2.dat`.

7. The timestamps of both `wmtsk.dll` and `wmm2.dat` are set to match that of cmd.exe.

8. The original `%TEMP%` instances of the worker and service DLLs are deleted.

9. The configuration is appended to the wmtsk.dll file using the same rolling XOR (starting with the value 0x99).

The bulk of the installation of Winnti is now complete. The dropped files are now in the correct location and ready for activation. The final steps of the installation establish the persistence model for the malware and begin the activation of the malware on the victim's system.

10. A new service entry is manually added to the registry to ensure the malware will activate upon reboot of the victim's machine. The new service is named `wind0ws`, with a display name of `automaticallyssl` and the description of `Monitoring of hardwares and automatically updates the device drivers`.

11. `WinExec` is used to call the command `net start wind0ws.`

12. The installation completes by returning control to the service component, which terminates quietly.

The service that the engine component installs runs under the svchost.exe executable as a `netsvc`. The engine will directly modify the registry after using the `CreateServiceA` function to establish the basics of the new service. The use of direct registry modifications to entries under the `LOCAL MACHINE` (`HKLM`) hive requires elevated privileges, which may explain why the authors went to such great pains to ensure the installation process occurred in a suitable process space.

ACTIVATION PHASE 1: SERVICE ACTIVITIES

Activation of the Winnti malware begins whenever the service component is activated. Typically, activation is the result of the registered service (e.g., `wind0ws`) being activated by either a reboot of the victim's machine or by the net start command being issued during the final phase of the installation process. In either case, the `DllMain` function is called prior to the ServiceMainEx function of the service component being called by Windows, thus kicking off the activation of the Winnti version
3.0 malware.

1. The `DllMain` function, upon activation, manually loads the engine binary into memory.

2. The exports from the engine component (`Install`, `DeleteF`, and `Workman`) are loaded into a memory structure.

3. The service component determines if the host binary responsible for the service component is either svchost.exe (if run as a service) or rundll32.dll (if loaded pseudo-manually or from the dropper).

4. If neither host executable is found, the `Install` function of the engine is activated. Otherwise, the `DllMain` function returns with a success status.

With the initialization of the service component's DLL complete, Windows calls the `ServiceMainEx` function to activate the service. The `ServiceMainEx` function, like the `DlgProc` function, is extremely lightweight in its functionality.

5. The `ServiceMainEx` function registers a service handler function to respond to service status control requests from Windows.

6. The service is set to the Running state.

7. An unnamed event is created.

8. The engine's `Workmain` function is called with both the path to the host executable (the service DLL) and the name of the service supplied as parameters.

9. A sleep of 3 seconds occurs before the function waits indefinitely for the unnamed event to become set.

The `ServiceMainEx` function does little more than establish a basic scaffold for activating the `Workmain` function of the engine component. The service remains active, thus providing a process space for the engine, until the unnamed event is set. The unnamed event becomes set only after the service receives the `SERVICE_STOP` signal from Windows.

After the service component calls the `Workmain` function of the engine component, the engine component picks up the baton to complete the next phase of the activation sequence.

ACTIVATION PHASE 2: ENGINE ACTIVITIES

The engine component's `Workmain` function, much like the service's `ServiceMainEx` function, provides a scaffolding for the next phase of the activation. In this case, that next phase is dependent on the worker component.

1. The `Workmain` function determines if an active activation thread exists within the process. If so, the `Workmain` function simply returns to avoid activating two or more concurrent instances of Winnti under the same process space.

2. The configuration is extracted from the service component based on the filename and path supplied to the `Workmain` function from the `ServiceMainEx` function. If the configuration extraction fails, the filename and path of the service DLL is gleaned from the `ServiceDLL` registry value for the service, and the configuration is extracted from that file.

3. The path to the worker component (e.g., `%SYSDIR%\wmm2.dat`) is extracted from the service component's file.

4. A new thread responsible for the activation of the worker and engine components is generated.

5. The `Workmain` function returns.

`Workmain` is a very simple function with a singular purpose: collect the data needed to locate the necessary components for activating Winnti on the victim's system. With the necessary information found, a new thread is generated that allows the service component to decouple from the worker component's functionality. Had this not occurred, the service component would not be able to respond to Windows status requests, and the service would have appeared to be hung, causing Windows to terminate the service.

The activation thread generated within the `Workmain` function loads the worker component, activates the worker component, and provides a thread independent of the service's thread under which to execute.

6. The path to the worker component's file is verified to exist. If the worker component's file does not exist, the activation process terminates immediately.

7. The worker component is loaded into memory.

8. The worker component's `work_start` function is called.

The worker component's executable image is encrypted. As part of the loading process, the engine must XOR each byte of the worker component's file with the value 0x36 and perform a nibble swap. The worker component's executable image suffers from the same malformed PE/COFF header that the engine component's image exhibits. As a result of the malformed PE/COFF header and the encrypted file image, the engine must manually load the worker component's image into memory in exactly the same manner that the service component loaded the engine component manually into memory.

The `work_start` function is the true beginning of the Winnti malware. The `work_start` function performs the various Remote Administration Tool (RAT) initialization functions of Winnti before activating the communication subsystem of Winnti. The result of calling the `work_start` function is the completed activation of Winnti and placing the system in a steady-state mode of C2 server requests and response actions. Once the `work_start` function initializes the Winnti malware, a new thread is generated to house the Winnti RAT functionality, allowing the `work_start` function to return control back to the activation thread within the engine component.

9. Upon completion of the `work_start` function, the activation thread sleeps for 30 seconds before entering an infinite loop.

10. The loop begins by verifying that the global event established during the installation process (Phase 3, step 5) does not exist. If the event exists, the loop is broken, the event is set, the Winnti malware shuts down by means of a net stop command, and the service and worker component files are deleted.

11. The presence of the `%SYSDIR%\otfkty.dat` file is checked, and if the file does not exist, control returns to the top of the loop (step 10).

12. The `%SYSDIR%\otfkty.dat` file is read as an INI file, the filename specified by the `File` variable under the `UPDATE` section is read, and the `otfkty.dat` file is deleted from disk if the file specified by the File variable exists and is not a directory.

13. If the worker has a `work_end` export, the `work_end` function is called.

14. The configuration of `%SYSDIR%\wmtsk.dll` is loaded into memory.

15. If `%SYSDIR%\sysprep\cryptbase.dll` exists, the file is deleted.

16. The worker component's file is deleted from disk.

17. The file specified by `otfkty.dat`'s `File` variable is copied to the filename of the worker component's file.

18. The timestamp of the new worker component file is set to that of cmd.exe's timestamp.

19. The file specified by `otfkty.dat`'s `File` variable is deleted.is deleted.

20. After sleeping for 3 seconds, the new worker component's image is loaded into memory, and the new worker component's `work_start` function is executed.

21. Control returns to the start of the loop (step 10).

The authors of version 3.0 of Winnti use the engine's scaffolding to allow for on-the-fly worker component updating without a need to restart the service. The infinite loop listens for the indicator that the engine's `Install` function is performing an installation (with an existing Winnti installation, this effectively becomes an update). As part of the installation process by the engine component's `Install` function, the presence of an existing service and worker component's files is verified (Installation Phase 3, step 4), resulting in the generation of the `%SYSDIR%\otfkty.dat` file. The presence of the `otfkty.dat` file informs the engine's activation thread that a new worker component is available and should be loaded. As a result, the engine cleanly shuts down the existing worker component by calling its `work_end` function, deletes the old worker, and replaces the worker component's file before loading and executing the new worker component. *The beauty of this approach is that not only does it allow hot patching of the malware but it does so without restarting the service, which could indicate an erroneous system event to an astute systems administrator.*

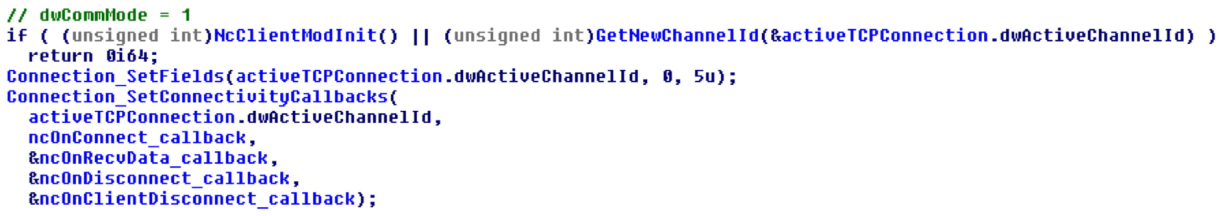
THE BASICS OF WINNTI'S WORKER

The Winnti worker component is an exercise in over engineering. As with the other components within the Winnti system, such as the service and the engine, the worker component is a scaffold for additional functionality. Unlike the service and engine components, the scaffolding provided by the worker component is substantial and complex, but at its core, the worker component has two primary functions: communication interface and plugin management.

The communication subsystem of the worker module supports three communication protocols, but the framework is developed in such a way that adding additional protocols requires little more than adding a different communication mode module to the source code at compile time. The communication subsystem in the samples analyzed by Novetta includes three modes: custom TCP protocol (used when `Config.dwCommMode` is set to 1), encapsulation within HTTP (used when `Config.dwCommMode` is set to 3), and encapsulation within HTTPS (used when `Config.dwCommMode` is set to 2). To further expand the reach of the HTTP and HTTPS modes, the HTTP and HTTPS modes can utilize a proxy local (or potentially external) to the victim's computer. In order to support a variety of different communication protocols and methods, the communication subsystem relies heavily on callback functions. For instance, when a communication module (be it the custom TCP protocol, HTTP, HTTPS, or some other type) initializes, it supplies a series of callback functions to the communication subsystem. The callback functions provide hooks to the communication subsystem for handing off important communication events.

Figure 1 illustrates the worker component using the `Connection:SetConnectivityCallbacks` function to register callback functions that handle the following series of events.

1. When a new connection occurs (`ncOnConnect_callback`)
2. When data is received over the network connection (`ncOnRecvData_callback`)
3. When the connection terminates by the socket closing (`ncOnDisconnect_callback`)
4. When the other end of the connection disconnects, but has not closed, the connection (`ncOnClientDisconnect_callback`)



```
// dwCommMode = 1
if ( (unsigned int)NcClientModInit() || (unsigned int)GetNewChannelId(&activeTCPConnection.dwActiveChannelId) )
  return 0i64;
Connection_SetFields(activeTCPConnection.dwActiveChannelId, 0, 5u);
Connection_SetConnectivityCallbacks(
  activeTCPConnection.dwActiveChannelId,
  ncOnConnect_callback,
  &ncOnRecvData_callback,
  &ncOnDisconnect_callback,
  &ncOnClientDisconnect_callback);
```

Figure 1: Setting Callbacks for the Custom TCP Protocol Communication Mode

By using callbacks, it is relatively easy for the authors of the Winnti malware to add new communication protocols without making significant changes to the source code. There is, however, the question as to why the authors chose to use a callback scheme for this purpose instead of a more modern object-oriented approach, such as using derived classes in C++.

The callbacks within the communication subsystem cloud an important aspect of the nature of the communication within Winnti: the communication subsystem is largely asynchronous. The communication subsystem allocates a thread solely for listening to incoming data, determining to which channel the data belongs, queuing the data in a series of network queue structures, and alerting the communication subsystem that the `ncOnRecvData_callback` (or equivalent) callback should be called to address the incoming data. This allows the sending of data from the communication subsystem to decouple the receiving of data from the communication subsystem thereby providing asynchronous data streams. The fact that the data streams are decoupled does introduce some complexity, as it is up to the higher layers of the data stream to reassemble the data in the appropriate form for whatever task to which the data applies. Evidence suggests, however, that despite the fact that sending and receiving data is asynchronous within the communication subsystem, in practice the data follows a standard request-and-reply model in which the Winnti malware makes a request over the network and then waits for a reply before repeating the sequence.

By itself, the worker component does very little. It does not have any built-in RAT functionality such as file management, remote command shell interaction, network monitoring, or other features common to malware that falls within the RAT category. Similar to the way that PoisonIvy provides only a framework and requires at-runtime modules to perform RAT functions, Winnti must load a plugin for each desired RAT (or class of RAT) feature. These modules, which internally the authors refer to as "Plus" modules, are basic plugins that the worker component is responsible for maintaining.

It would be a poor design for the malware to request a download of code for each RAT function that an attacker wishes to use. The amount of extraneous data would be excessive and would make the malware's traffic more prone to detection, as plugins are usually a minimum of several tens of kilobytes each. The authors of Winnti compensate for this by caching plugin modules in memory and possibly on disk. Whenever a new module is loaded into the victim's machine by virtue of a download from the C2 server, the plugin is stored, loaded into memory, and registered with the plugin subsystem, which allows the communication subsystem to pass requests to the plugins from that point on. Optionally, as part of the integration of the plugin into the Winnti malware, the attacker can request that a copy of the plugin be stored within the `%PROGRAMFILE%\Microsoft Shared\MSInfo\en-US\` directory, which will allow the worker component to load the plugin automatically whenever the Winnti malware restarts; however, storing a plugin is not mandatory. It is entirely possible that the attacker may specify that the plugin should exist in memory only as long as the malware is active. This prevents disk based forensics from detecting the plugins and limits the availability of data for analysis to determine what code may have executed on a victim's machine.

Plugins are architecture dependent, but the authors of Winnti make no special effort to ensure that only 64-bit plugins run on 64-bit variants of Winnti or that only 32-bit plugins run on 32-bit variants. Plugins are DLLs with their PE/COFF headers manipulated (like the engine and worker components) to make them unloadable by standard Windows Application Programming Interface (API) functions, therefore requiring that the plugin manager manually load the plugins. A plugin information header precedes the modified PE/COFF header. The plugin information header (`PluginEntry`) , as seen below, contains information defining attributes about the plugin, such as its architecture (64 or 32-bit), the size of the plugin's image, the entry point function, the version of the plugin, and, most importantly, the identification number of the plugin.

```
struct PluginEntry
{
	DWORD dwPluginID;
	DWORD Version;
	DWORD ArchitectureType;
	DWORD unknown;
	DWORD dwPluginSize;
	DWORD dwEntryFunctionNameHash;
	DWORD fLoaded;
	void *pPluginImage;
	int (__stdcall *pfnEntryPoint)
	(void *incomingData, int (__cdecl

**pfnNetDataSend)(PacketHeader *));
};
```

The identification number of the plugin (`dwPluginID`) is the value that allows the communication subsystem to direct incoming requests to the appropriate plugin. The plugin manager itself supports only the following three commands from the communication subsystem:

COMMAND ID	DESCRIPTION
0x12	*Unknown purpose*
0x14	*Send a list of plugins currently registered to the C2*
0x15	*Add a new plugin to the active Winnti malware with an option to save the plugin to disk*

If a command coming from the communication subsystem does not match one of the plugin manager's commands, the `RemoteLib:CallPlusList` function is called to redirect the data packet to the appropriate plugin or return an error to the C2 server.

COMMUNICATION SCHEME

Regardless of the communication model currently active for a Winnti instance, the underlying communication remains constant. Each datagram that originates from or is destined for the C2 server has the following predefined header structure:

```
struct PacketHeader
{
	DWORD dwTickCount; WORD cmd;
	DWORD unknown;
	DWORD dwPayloadSize;
	DWORD dwStreamID;
};
```

The format of the data that follows the `PacketHeader` is `cmd` dependent. The `cmd` field allows the communication subsystem to route the request to the appropriate plugin by using the plugin manager to match the `cmd` value with the `dwPluginId` value. The worker component allows a maximum datagram size of 261120 bytes. To accommodate data streams larger than the maximum datagram size, the stream can be chunked.

The `dwStreamID` value is used to reassemble the streams by appending datagrams with the same `dwStreamID` together. The `dwPayloadSize` field defines the number of bytes within the datagram. The protocol that the worker component uses to transmit the `PacketHeader` and the optional payload of the datagrams can and will add additional complexity to the network traffic. The custom TCP protocol and the HTTP and HTTPS communication modes each deliver the datagrams differently. The HTTP and HTTPS communication modes will generate POST requests to the C2 server (typically to `/index.htm`) with the datagram (compressed using Zlib) as the body of the POST.

The custom TCP protocol uses a combination of encryption and compression to transfer the datagrams. The custom TCP protocol uses a stacked approach to transforming the data. First, the datagram, which makes up the payload of the custom TCP protocol, is typically compressed with LZMA. The compressed payload is appended to the following header specific to the custom TCP protocol:

```
struct TCPProtocolHeader
{
        DWORD magic;
        DWORD flags;
        DWORD dwXORKey;
        QWORD crc64;
        DWORD dwCompressedSize;
        DWORD dwPacketSize;
};
```

The `magic` value is `0xACED1984`. The `flags` value will specify if the datagram is compressed or not. The `dwXORKey` value is initialized to zero. The `crc64` value for the datagram (prior to compression) is stored in `crc64`. The size of the compressed payload is recorded in `dwCompressedSize` while the original size of the datagram is stored in `dwPacketSize`.

The final transformation prior to transmission for the custom TCP protocol involves encrypting the entire packet. A 32-bit value is generated (by calling `GetTickCount`) and used as the DWORD XOR key. Each DWORD within the packet is then XOR'd with the key. Given that the `dwXORKey` field of the `TCPProtocolHeader` was initialized to zero and exists on a DWORD boundary, the XOR key is recorded within the `dwXORKey` field. A successful decryption is determined by XOR'ing the `magic` and `dwXORKey` fields to produce the `0xACED1984` value.

CODE REUSE

The authors of Winnti are clearly proponents of the open-source software movement, as large chunks of the worker binary consist of open-source software packages. The authors statically linked in the OpenSSL library (version 0.9.8x),[3] the LZMA compression library,[4] the nedalloc memory allocation library,[5] and the Zlib library (version 1.2.7).[6] As for the part of the code that generates the unique identifier for the victim's computer, the authors of the worker component lifted the code DISKID32,[7] which is an open-source utility for reading the manufacturer data from a hard drive. The DISKID32 package is a surprisingly obscure piece of software from a company in Texas that writes an industrial process simulator that has "over 1,000 active users."[8]

[3] ftp://ftp.openssl.org/source/old/0.9.x/openssl-0.9.8x.tar.gz. 10 May 2012.
[4] http://sourceforge.net/projects/sevenzip/files/LZMA%20SDK/lzma922.tar.bz2/download. 18 April 2010. 5 Niall Douglas. "nedalloc Homepage". http://www.nedprod.com/programs/portable/nedmalloc/. 5 December 2014.
[6] Jean-loup Gailly and Mark Adler. "zlib". http://zlib.net/. 26 March 2014.
[7] WinSim, Inc. "DISKID32 (FREEWARE)". https://www.winsim.com/diskid32/diskid32.html. Accessed 1 December 2014.
[8] WinSim, Inc. "About WinSim". https://www.winsim.com/index.html. Accessed 1 December 2014.

The basis for the worker component is a library called RemoteLib. Some earlier variants of version 3.0 of Winnti contain debugging output that references the RemoteLib library. RemoteLib does not appear to be a publically available piece of software but does appear to be actively maintained, as evident by the gradual refinements seen between the 32-bit and 64-bit variants of version 3.0 of Winnti.

The compilation of the full OpenSSL library into the worker component solely for implementing HTTPS results in a significant size increase. Approximately 60 percent of the worker component is made up of OpenSSL functions when only a small fraction (less than 1 percent) of the OpenSSL library is actually used by the worker component.

DETECTION

Detecting Winnti via standard IDS signatures or network traffic inspection is not a straightforward process whenever the malware is configured to use HTTPS or the custom TCP protocol due to the use of encryption. However, more advanced network based behavioral analytic capabilities as well as host-based indicators do exist that can alert a security team or systems administrator to the presence of Winnti.

The version 3.0 variants of Winnti attempt to install themselves as a service with the following characteristics:

SERVICE NAME	`WIND0WS`
SERVICE DISPLAY NAME	`automaticallyssl`
SERVICE DESCRIPTION	`Monitoring of hardwares and automatically updates the device drivers`

From a file system perspective, it is possible to identify Winnti infections by looking for the following filenames:

- `%SYSDIR%\otfkty.dat`
- `%SYSDIR%\wmtsk.dll`
- `%SYSDIR%\wmm2.dat`

Given that Winnti will alter the time stamp of files to match that of the victim's cmd.exe file, looking for files with the **exact** same time as the victim's cmd.exe may identify other foreign files on the victim's system that warrant inspection and possible isolation.

Novetta established the following YARA signatures for detecting the various components of version 3.0 of the Winnti malware; administrators are advised to use these signatures to help detect and remediate active version 3.0 Winnti infections.

```
rule Winnti_Dropper
{
       meta:
              copyright = “Novetta Solutions”
              author = “Novetta Advanced Research Group”
       strings:
              $runner = “%s\\rundll32.exe \”%s\”, DlgProc %s”
              $inflate = “Copyright 1995-2005 Mark Adler”

       condition:
              $runner and $inflate

}

rule Winnti_service
{
       meta:
              copyright = “Novetta Solutions”
              author = “Novetta Advanced Research Group”

       strings:
              $newmem = “new memory failed!”
              $value = “can not find value %d\n”
              $onevalue = “find one value %d\n”
              $nofile = “Can not open the file (error %d)”

       condition:
              3 of ($newmem, $value, $onevalue, $nofile)

}

rule Winnti_engine {
       meta:
              copyright = “Novetta Solutions”
              author = “Novetta Advanced Research Group”

       strings:
              $api1 = “SHCreateItemFromParsingName” $datfile = “otfkty.dat”
              $workstart = “work_start”
              $workend = “work_end”

       condition:
              ($api1 or $datfile) and ($workstart and $workend)
}

rule Winnti_worker
{
       meta:
              copyright = “Novetta Solutions”
              author = “Novetta Advanced Research Group”

       strings:
              $pango = “pango-basic-win32.dll”
              $tango = “tango.dll”
              $dat = “%s\\%d%d.dat”
              $cryptobase = “%s\\sysprep\\cryptbase.dll”

       condition:
              $pango and $tango and $dat and $cryptobase

}
```