★OPERATION★
BLOCKBUSTER

LOADERS, INSTALLERS AND UNINSTALLERS REPORT

NOVETTA

NOVETTA

Novetta is an advanced analytics company that extracts value from the increasing volume, variety and velocity of data. By mastering scale and speed, our advanced analytics software and solutions deliver the actionable insights needed to help our customers detect threat and fraud, protect high value networks, and improve the bottom line.

For innovative solutions for today's most mission-critical, advanced analytics challenges, contact Novetta:

Phone: (571) 282-3000 | www.novetta.com

www.OperationBlockbuster.com

TABLE OF CONTENTS

1. Introduction

This report details some of the technical findings of the Lazarus Group's malware, observed by Novetta during Operation Blockbuster. We recommend reading the initial report prior to the reverse engineering reports for more details on the Operation and the Lazarus Group. This reverse engineering report looks at the installers, loaders, and uninstallers found within the Lazarus Group's collection.

Installers provide an actor with a convenient way to install malware on a victim's system with minimum user intervention. Installing malware on a victim's system usually entails copying the malware binary to the appropriate, desired location, installing a mechanism to ensure that the malware remains active after a system reboot, installing any additional configuration parameters or files, and, of course, the actual activation of the malware on the victim's machine. These steps can be cumbersome and error prone when done manually. An installer eases the burden on the actor by providing a single application that will perform the installation activities consistently across multiple infections.

Loaders are responsible for loading a piece of malware on a victims' system, but not necessary installing the malware. Clearly loaders and installers have similar traits, but the primary distinction is that a loader is usually a support system for another malware component. Loaders are typically responsible for loading a DLL component into memory given that a DLL cannot operate in a standalone mode such as an executable.

On the opposite side of the coin, an uninstaller removes a particular set of files from a victim system. In most cases, the uninstaller not only deletes the malware family's binary, but also removes its services, ancillary files, and registry keys that were installed as part of the installation process.

The Lazarus Group uses a variety of installers and loaders. Generally, there is a one-to-one relationship between installer and the malware family it installs, but there are examples where one installer family has been observed dropping different malware families over time. This report will explore the various installers, uninstallers and loaders Novetta has observed the Lazarus Group using.

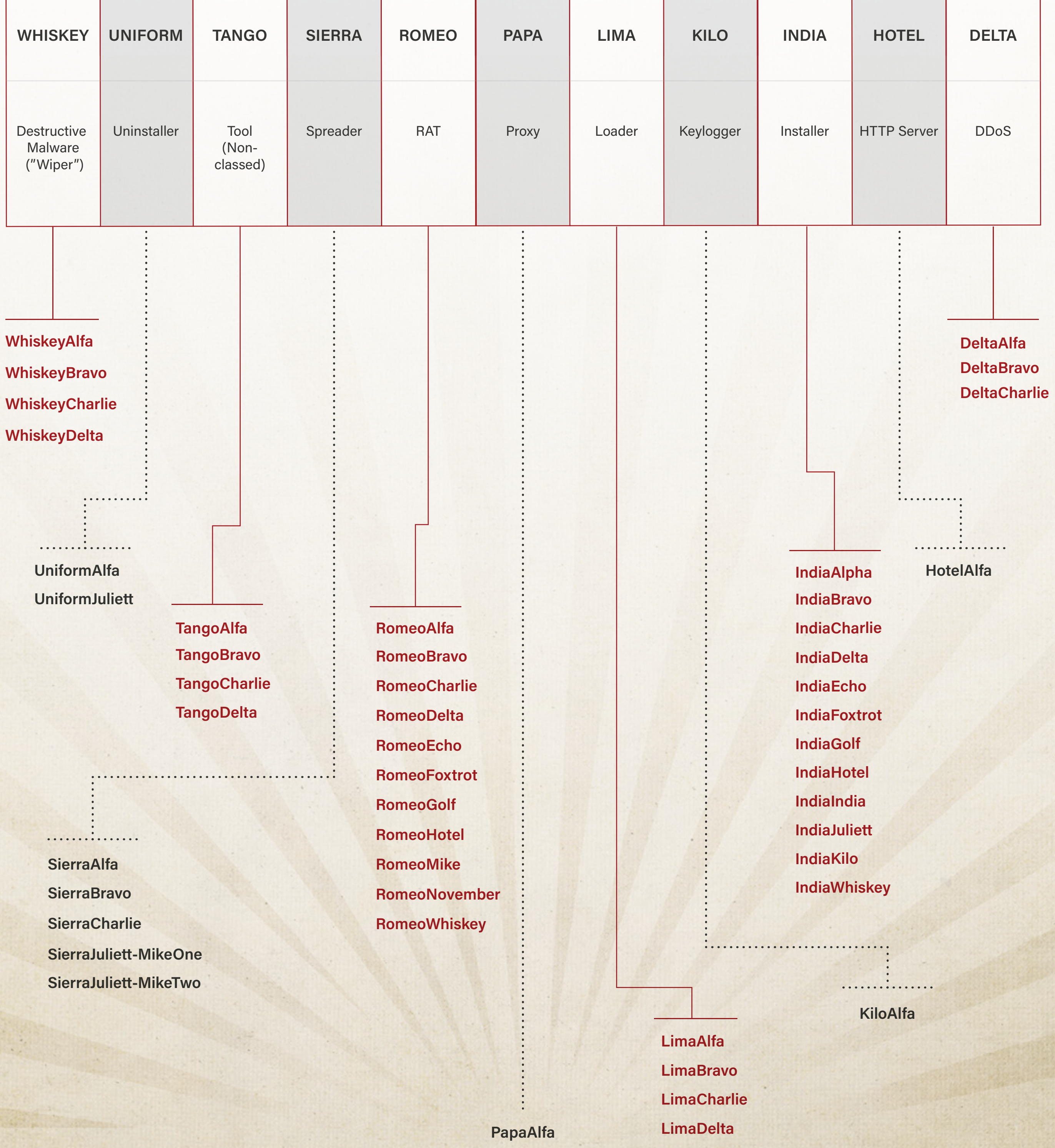
The naming scheme used by Novetta for the malware identified during Operation Blockbuster consists of at least two identifiers which each identifier coming from the International Civil Aviation Organization (ICAO)'s phonetic alphabet, commonly referred to as the NATO phonetic alphabet. The first identifier specifies the general classification of the malware family while the second identifier specifies the specific family within the larger general classification. For example, IndiaAlfa specifies an installer family identified as Alfa.

FIRST LEVEL IDENTIFIER	GENERAL CLASSIFICATION
Delta	DDoS
Hotel	HTTP Server
India	Installer
Lima	Loader
Kilo	Keylogger
Papa	Proxy
Romeo	RAT
Sierra	Spreader
Tango	Tool (Non-classed)
Uniform	Uninstaller
Whiskey	Destructive Malware ("Wiper")

Table 1-1: First Level Identifiers for the Lazarus Group Family Names and their Classification Meanings

There is no temporal component to the second level identifiers given to malware families. While generally the second identifiers are largely sequential (Alfa, Bravo, Charlie, and so on), the identifier does not indicate that one family came before another chronologically. Instead, the second level identifiers were assigned by the order Novetta discovered each particular family.

THE LAZARUS GROUP



2. [Installer] IndiaAlfa

The installer identified as IndiaAlfa has the characteristics of a phishing attack payload. IndiaAlfa mimics the behavior of opening a document file (PDF or Hangul's[1] HWP) while at the same time installing a malicious payload. Samples of the IndiaAlfa family have been observed as early as September 2014 and as recently as May 2015, and identified samples have dropped a variety of Korean-language decoy documents.

There are slight variations between the various IndiaAlfa samples, but at their core each IndiaAlfa sample performs the following tasks:

1. Drop the phishing document (observed phishing documents are either PDF or HWP files)
2. Drop the RomeoAlfa malware as **`%TEMP%\AdobeArm.exe`**
3. Open the phishing document
4. Activate the RomeoAlfa malware
5. Execute the suicide script to remove the IndiaAlfa binary.
6. Terminate

While the basic tasks remain the same over time for IndiaAlfa samples, the underlying methodologies differ. There are two known variants of IndiaAlfa which, for ease of identification, are given the names IndiaAlfa-One and IndiaAlfa-Two. IndiaAlfa-One samples have compile dates within September 2014 and IndiaAlfa-Two samples have observed compile dates from March 2015 onward.

1 Hancom. "Hangul Office 2014". http://www.hancom.com/en/product/product2014vp_01.jsp Accessed 26 October 2015.

2.1 IndiaAlfa-One

IndiaAlfa-One samples drop two binaries in addition to a phishing document on the victim's machine. The first executable is the RomeoAlfa malware and the second is a DLL that IndiaAlfa-One drops as `HwpFilePathCheck.dll` (SHA256: a7622ad260db5e66939e6b571e410df7b6e86cb0b633d155a7e27550f50e9cb6). Internally, the DLL is named `hwp_vs6_com_dll.dll`. `HwpFilePathCheck.dll` is a COM-based scaffold for opening a HWP file. The DLL generates an instance of `HWPFrame.HwpObject.1` in order to interact with the Hangul application for the purpose of opening the phishing document. Once the phishing document has been opened by the DLL's `OpenDocument` exported function, the DLL serves no further purpose. The use of the DLL to open the HWP files is one of the defining characteristics of IndiaAlfa-One samples.

IndiaAlfa-One samples have the earmarks of being the payload of a builder. The phishing document, the RomeoAlfa malware payload, and the `HwpFilePathCheck.dll` binary are appended to the end of the IndiaAlfa-One binary. In order to extract the files (Tasks #1 and #2), the malware reads a stack of 32-bit values that indicate the size of the file image that precedes the value. In other words, each file is appended to the end of the IndiaAlfa-One binary in a {image (variable size), size of image (DWORD)} layer. There are three of these layers within the IndiaAlfa-One samples. In order (from last to first), the layers consist of the phishing document, the RomeoAlfa malware image, and the `HwpFilePathCheck.dll` image. Figure 2-1 illustrates the stacks of appended binaries found within IndiaAlfa-One binaries.

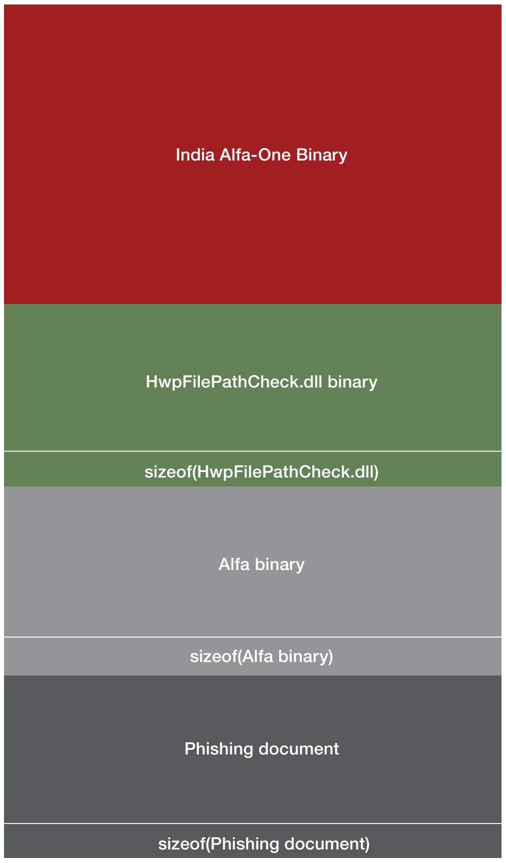

India Alfa-One Binary
HwpFilePathCheck.dll binary
sizeof(HwpFilePathCheck.dll)
Alfa binary
sizeof(Alfa binary)
Phishing document
sizeof(Phishing document)

Figure 2-1: Payload File Stacking within IndiaAlfa-One Samples

In order to perform Task #3, IndiaAlfa-One will load the `HwpFilePathCheck.dll` binary into memory using `LoadLibrary` and then call the exported function `OpenDocument` with the name of the phishing document. The name of the phishing document is hardcoded within IndiaAlfa-One and the observed samples use the Korean code page for the filename string. After activating `OpenDocument`, IndiaAlfa-One will use the `FindWindowA` API function to locate the opened document. If the window of the document is located, **`SetForegroundWindow`** is called to ensure that the phishing document is the topmost window on the victim's display.

The implementation of Task #5, the suicide script, uses a series of string concatenation operations in order to construct the script before writing the script to a batch file, as illustrated below. Many suicide scripts are executed via a call to **`WinExec`** or **`CreateProcess`** API calls, but IndiaAlfa-One uses **`ShellExecuteA`**, which is a notable departure from the majority of Lazarus Group's malware families including IndiaAlfa-One's sibling IndiaAlfa-Two.

```
strcpy(buffer, "@echo off\r\n");
strcpy(buffer, ":Loop\r\ndel /a H \"");
strcat(buffer, szIndiaAlfaBinaryFilename);
strcat(buffer, "\"\r\nif exist \"");
strcat(buffer, szIndiaAlfaBinaryFilename);
strcat(buffer, "\" goto Loop\r\ndel \"");
strcat(buffer, szSuicideScriptFilename);
strcat(buffer, "\"");
WriteFile(fp, buffer, strlen(buffer), &NumberOfBytesWritten, 0);
CloseHandle(fp);
ShellExecuteA(0, "open", szSuicideScriptFilename, 0, 0, 0);
```

Figure 2-2: Snippet of the IndiaAlfa-One Suicide Script Generation Code

There is a unique mistake within the suicide script generation code in IndiaAlfa-One. The first instruction to construct the content of the suicide script is immediately rendered useless by the second instruction. Specifically, the **`strcpy(buffer, "@echo off\r\n")`** instruction has no effect since the next instruction, **`strcpy(buffer, ":Loop\r\ndel /a H \"")`**, will overwrite the results of the first instruction. The developer(s) should have used **`strcat`** instead of **`strcpy`** for the second instruction if the **`@echo off`** command was desired. Ultimately this has no negative effects on the suicide script, but it is an artifact that is also found in other Lazarus Group families using the same code for their suicide script generators, notably WhiskeyAlfa-One and TangoDelta.

2.2 IndiaAlfa-Two

The development of IndiaAlfa-Two shows a change in technique, but only a slight deviation in methodology. The first noticeable difference in IndiaAlfta-Two is the use of dynamic API loading. IndiaAlfa-Two is a simple installer which does not require a significant number of function calls in order to execution its objectives, and the developers only dynamically load 7 functions. The API function names are encoded by XORing each byte in the string with `0xA7`.

While IndiaAlfa-One used a hardcoded filename for the phishing document, IndiaAlfa-Two relies on the name of itself to provide the document's name. For instance, if the IndiaAlfa-Two binary has the filename "abcdef.exe", the dropped phishing document would be created within the same directory as the binary and given the name "abcdef.pdf" (for a PDF file).

IndiaAlfa-Two uses the resource section of its binary to store the RomeoAlfa binary (in `BIN\101`) as well as the phishing document (in `BIN\102`). While IndiaAlfa-Two uses a simple XOR (using `0xA7` as the key) to obfuscate/encode API strings, the authors make no effort to obfuscate the payload components. Both the RomeoAlfa binary and phishing document are stored within the resource section in plaintext.

IndiaAlfa-Two does not use a secondary DLL in order to open the phishing document, instead using a simpler approach with the `ShellExecute` API function, which will perform the default action the operating system has associated with the file's type. Additionally, the use of `ShellExecute` allows the IndiaAlfa-Two binary to serve various file types without additional overhead. For instance, the IndiaAlfa-Two samples can easily switch between PDF and HWP phishing documents without the need to recode the portion of the binary responsible for opening the document.

IndiaAlfa-Two introduces a slight deviation in the tasks that IndiaAlfa-One performs. The first task that IndiaAlfa-Two performs is the removal of the file `%TEMP%\~DF`{random value}. IndiaAlfa-Two will attempt to move itself to the `%TEMP%\~DF`{random value} filename prior to the suicide script's execution. The suicide script itself is generated using a different technique than seen in IndiaAlfa-One. Notably, the script is generated in place on disk as seen in the Figure 2-3, rather than in memory and then copied to disk.

```
void __noreturn Suicide(char *pszFilename)
{
  FILE *fp;
  char szSuicideScriptFilename[256];
  GetTempPathA(0x104u, szSuicideScriptFilename);
  strcat(szSuicideScriptFilename, "PM0D4.bat");
  fp = fopen(szSuicideScriptFilename, "wb");
  fprintf(fp, ":Repeat1\r\n");
  fprintf(fp, "del \"%s\"\r\n", pszFilename);
  fprintf(fp, "if exist \"%s\" goto Repeat1\r\n", pszFilename);
  fprintf(fp, "del \"%s\"\r\n", szSuicideScriptFilename);
  fclose(fp);
  WinExec(szSuicideScriptFilename, 0);
  exit(1);
}
```

Figure 2-3: IndiaAlfa-Two's Suicide Script Generator

While observed samples of IndiaAlfa-Two have only used PDF and HWP phishing documents, the binary indicates that Microsoft Excel spreadsheet files (XLS) may also be used by the Lazarus group in attacks. IndiaAlfa-Two executes the code seen in Figure 2-4 immediately before beginning Task #1.

```
printf("ExePath: %s\nXlsPath: %s\nTmpPath: %s\n", szTrojanFilenamePath,
szCoverFilenamePath, szNewDropperFilenamePath);
```

Figure 2-4: Artifact within IndiaAlfa-Two Indicating the Possibility that IndiaAlfa-Two May Also be used with Excel Decoy Documents

3. [Installer] IndiaBravo

IndiaBravo is a versatile installer that has been observed installing three different malware families: the RomeoBravo RAT, the RomeoCharlie RAT, and the PapaAlfa proxy. IndiaBravo provides a customizable framework that the authors of other malware families can utilize to install their malware on a victim's machine. The IndiaBravo installer performs, at a minimum, four tasks:

1. Dynamically load API functions (including functions that are not used by the installer).
2. Call the **WriteConfig** function to install the configuration for the malware being installed.
3. Call the **ExtractDll** function to install the malware image on the victim's machine.
4. Call **CreateSVC** to install and start the Windows service that is responsible for activating the installed malware.

The dynamic API loading component of IndiaBravo uses a combination of XOR and Space-Dot string obfuscation. The names of libraries (e.g. **kernel32.dll**, **advapi32.dll**) are obfuscated by using the XOR byte manipulation with the constant **0xA7** while the names of the API functions to load are obfuscated using the Space-Dot method. Of the 102 API functions that IndiaBravo loads during the dynamic API loading process, only 21-31% of the functions are used by the installer, depending on the type of malware being installed.

As previously stated, IndiaBravo is a framework, allowing a malware author to modify the base code to fit a particular malware's installation needs. As such, the **WriteConfig** function is by-and-large dropped depending on the malware, while the functions **ExtractDll** and **CreateSVC** are not. To date, there have been three IndiaBravo variants observed: IndiaBravo-RomeoBravo, as the name implies, installs the RomeoBravo RAT, IndiaBravo-RomeoCharlie installs the RomeoCharlie RAT, and IndiaBravo-PapaAlfa installs the PapaAlfa proxy. The malware image is appended to the end of the IndiaBravo executable followed by a 32-bit value (DWORD) identifying the size of the malware image in exactly the same way that IndiaAlfa stores its dropped components. Both IndiaBravo-RomeoBravo and IndiaBravo-RomeoCharlie store their malware payloads in plaintext, while IndiaBravo-PapaAlfa offsets the value of each byte within its malware payload image by adding **0xD7** to each byte.

Before exploring the intricacies of each variant, it is best to explain the more general functions consistent across variants, **ExtractDll** and **CreateSVC**. The **ExtractDll** function drops the malware on a victim's system. While only IndiaBravo-PapaAlfa has been observed decoding its payload prior to dropping said payload to disk, structural artifacts within IndiaBravo samples suggests that this may the more the norm for the IndiaBravo framework, rather than the exception. Both of the IndiaBravo-Romeo variants contain code within **ExtractDll** that appears to forego manipulating the payload's image (or more precisely, performs the same act as IndiaBravo-PapaAlfa but with **0** as the constant instead of **0xD7**). The code fragment in Figure 3-1 from an IndiaBravo-RomeoBravo sample illustrates the location of where the IndiaBravo framework's **ExtractDll** function would manipulate a payload's image in memory:

```
ptr = (char *)LocalAlloc(0x40u, dwImageSize);
if ( ptr )
{
   memcpy(ptr, &pvImageSource[uBytes - 4] - dwImageSize, dwImageSize);
   for ( i = 0; i < dwImageSize; ++i )
       ptr[i] = ptr[i];
```

Figure 3-1: Neutered Decryption Code Snippet from IndiaBravo's ExtractDLL Function

After extracting the payload's image into memory, **`ExtractDll`** writes the file to disk within the **`%SYSDIR%`** directory using the specified filename (which varies by variant). In order to camouflage the newly dropped malware, **`ExtractDll`** attempts to copy the timestamp for **`calc.exe`** to the newly dropped malware file before the **`ExtractDll`** function returns.

With the dropped malware written to disk thanks to **`ExtractDll`**, **`CreateSVC`** provides the scaffolding for persistence and activation of the dropped malware on the victim's system. IndiaBravo's general approach for establishing a service on a victim's machine is as follows:

1. Create a registry key at **`HKLM\SYSTEM\CurrentControlSet\services\`**<name of new service>
2. Set the **`Description`** and **`Parameter\ServiceDll`** values appropriately
3. Create a new service group under **`HKLM\Software\Microsoft\Windows NT\CurrentVersion\Svchost`** for the new service
4. Call **`CreateServiceW`** to add the new service to the services database
5. Call **`ChangeServiceConfig2W`** to ensure that the service will auto-restart after an error condition
6. Call **`StartServiceW`** to activate the service

The values for the service group, the **`Description`** and **`ServiceDll`** values, and the name of the service are all specific to each of the IndiaBravo variants. Table 3-1 shows each of the **`Description`** fields for the various IndiaBravo variants with the unique elements in bold. What is particularly interesting about the **`Description`** strings is the extremely slight variations between each variant. The similarities between the various **`Description`** values vary so little that the typo of the missing space in "**`network.If`**" is found in all three variants of IndiaBravo. This commonality coupled with the fact that different IndiaBravo variants are being used to deliver different malware payloads adds additional support to the claim that IndiaBravo is an installer framework that is customized for each type of dropped malware.

IndiaBravo-RomeoBravo	**Local WMI Security Service** provides performance library information from Windows Management Instrumentation providers network. If the Service is not running Wmi Security log is not being saved.
IndiaBravo-RomeoCharlie	Provides performance library information from Windows Management Instrumentation **(WMI)** providers to clients on the network. If the Service is not running, Wmi Security log is not being saved.
IndiaBravo-PapaAlfa	**Internet Network Moniter** provides performance library information from Windows Management Instrumentation providers network. If the Service is not running, Wmi Security log is not being saved.

Table 3-1: Description Values for the IndiaBravo Variants

Once IndiaBravo completes its task of dropping the configuration for its malware, dropping the malware itself, and finally creating and activating a service to ensure the malware's persistence, IndiaBravo then terminates. Console applications, such as IndiaBravo, return an exit code upon completion. While the return code for an application is usually set to `0` and subsequently ignored by the calling application (such as `cmd.exe`), the exit code can provide status information about a program to the caller, allowing the caller to determine success or failure of an application's task. This is a common practice for scriptable applications. IndiaBravo actively uses the exit code feature to signal the status of its operation, returning a value of `0` to `4` depending on if an error occurred and, if so, what part of the framework experienced the error. Table 3-2 explains the meaning of each exit code:

EXIT CODE	DESCRIPTION
0	IndiaBravo successfully completed
1	(variant dependent)
2	`WriteConfig` failed
3	`ExtractDll` failed
4	`CreateSVC` failed

Table 3-2: IndiaBravo's Exit Codes

Because each of the IndiaBravo variants customizes the framework to fit the needs of the malware and the developer, the following subsections explore the key variations between each of the IndiaBravo variants.

3.1 IndiaBravo-RomeoBravo Variant

IndiaBravo-RomeoBravo (IBRB) is the installer responsible for dropping the RomeoBravo malware. IBRB begins by dropping the malware binary at **`%SYSDIR%\wmisecsvc.dll`**. Compared to other IndiaBravo variants, the line of demarcation between IBRB's **`ExtractDll`** and **`WriteConfig`** functions is more blurred based on the fact that **`ExtractDll`** also extracts a configuration data blob of 456 bytes from the IBRB binary. The configuration data exists immediately before the RomeoBravo image and is stored in plaintext. **`ExtractDll`** loads the configuration blob into memory.

Once **`ExtractDll`** completes the file and configuration data extraction, **`WriteConfig`** modifies the configuration data blob by assigning a 64-bit, randomly generated value to the **`qwIdentifier`** field of the RomeoBravo configuration. The **`WriteConfig`** encrypts the configuration data using the DNSCALC-style encoding scheme, and the encrypted configuration is then dropped to disk at **`%SYSDIR%\tmscompg.ini`**.

The **`CreateSVC`** function follows the general IndiaBravo method of establishing a new service, but, specifically, the function generates a new service group named **`LocalWmiSecurity`** after establishing the registry entries necessary to define the **`WmiSecSvc`** service.

IBRB monitors the status of each of the installation tasks and, if any of the tasks fails, calls a suicide script (saved in the **`%TEMP%`** directory as **`cvrit000.bat`**) to remove its binary. The suicide script, Figure 3-2, only removes the IBRB binary and will not attempt to clean up any of the successful steps that may have occurred during the installation process.

```
:L1
del "<full path to>wmisecsvc.dll"
if exist "<full path to>wmisecsvc.dll" goto L1
del <full path to>cvrit000.bat
```

Figure 3-2: IndiaBravo-RomeoBravo's Suicide Script

After completing the installation and activation tasks, IBRB then calls the suicide script to remove itself from the victim's machine.

3.2 IndiaBravo-RomeoCharlie Variant

IndiaBravo-RomeoCharlie (IBRC) is a verbose installer for the RomeoCharlie malware. IBRC records each step of the installation process to STDOUT. The IndiaBravo framework uses the Windows GUI subsystem, meaning any data sent to STDOUT is not echoed to the console as would be the case for a program that uses the Windows character subsystem. Instead, the user of IBRC would have to pipe STDOUT to a file. After a successful execution of IBRC, the output from STDOUT looks like Figure 3-3.

```
SvcName = WebOrderService
[SVCNAME] = WebOrderService[1] - WriteConfig...
[2] - Extract Dll...
[3] - CreateSVC...
[4] - Success
```

Figure 3-3: IndiaBravo-RomeoCharlie's Output from STDOUT

The `SvcName` and `[SVCNAME]` fields are identical, but variable. This a byproduct of how IBRC establishes the service. Prior to the call to `WriteConfig`, IBRC constructs the service name, service group, service display name, and the configuration filename by randomly selecting from a group of text strings to construct the values.

FIELD	PATTERN
Service Name	<prefix1 string><name1 string><suffix1 string>
DLL Filename	<prefix1 string><name1 string><suffix1 string>.dll
Config Filename	<prefix1 string><name1 string><suffix1 string>tl.cpl
Service Display Name	<prefix2 string> <name2 string> <suffix2 string>
Service Group Name	<name3 string><prefix1 string>

Table 3-3: IndiaBravo-RomeoCharlie's Service Information Field Patterns

The values for the different prefix, name, and suffix string sets are as follows:

SET	VALUES				
prefix1	Wmi	Dcom	Timer	Win	Remote
	Sys	App	COM	Web	Com
prefix2	WMI	DCOM	Timer	Windows	Remote
	System	Application	COM+	Web Internet	Computer
name1	Sec	Log	hlp	Evt	Order
	Registrys	logon	Mon	Accesses	HW
name2	Security	Log	Helper	Event	Ordering
	Registrys	Logon	Moniter	Accesses	Hardwares
name3	LocalHost	Network	LocalService	LocalCenter	NetWorkRestricted
	LocalNetwork	LocalRestricted	LocalMan	LocalNo	Service
suffix1	Svc	svc	Man	Manager	SS
	host	Psvc	Agent	Service	Client
suffix2	Service	Service	Manager		Scheduler
	HOST	Protocal Service	Agent	Services	Client

Table 3-4: IndiaBravo-RomeoCharlie's Possible Prefix, Names, and Suffix Values

Note that the index of the string used for prefix1 are the same for prefix2. This means that if IBRC randomly selects "Web" as the value for prefix1, prefix2 will have the value "Web Internet." Likewise, the index for name2 will be the same as the index used for name1 and suffix1 and suffix2 will have the same index into their respective string arrays. This ensures that the display name for the service closely corresponds to the name given for the service name and DLL.

IBRC is more interactive than the other IndiaBravo variants and can take an optional command line argument. The argument will be a number that specifies the port number that the RomeoCharlie malware will be configured to listen on. In the absence of such a command line argument, the default port of 488 is selected. To ensure that the port is available as a listening port, IBRC initializes the Winsock subsystem and attempts to connect to the port using the loopback address 127.0.0.1. If the connection is successful, the socket disconnect function `SendErrorAndCloseSocket`, found in many of the Lazarus Group's malware families, is used to close the socket. A successful connection to the port on the loopback interface indicates that the port is unavailable for binding and will result in IBRC aborting the installation process immediately and setting the exit code to 1.

The `WriteConfig` function within IBRC stores the configuration within the victim machine's registry, not as a file. While the process of generating service information does, in fact, generate a configuration file name, a file is never generated by IBRC for the configuration. Instead, the configuration for RomeoCharlie is stored in the key `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security\xc123465-efff-87cc-37abcdef9` as a data blob. The `WriteConfig` function also stores the specified listening port, assigns a randomly generated 64-bit number as the victim machine's unique identifier, and saves the selected name of the service in the configuration's data blob prior to setting the value in the registry.

IBRC is unique among the observed IndiaBravo samples in that it does not use a suicide script to remove itself from a victim's machine after execution. Also unique is the inclusion of the string `RS_SDKInstaller` and `RS_SDKINSTALLER` within the resource section's String branch. This value is not found in the other IndiaBravo variants.

3.3 IndiaBravo-PapaAlfa Variant

IndiaBravo-PapaAlfa (IBPA) is responsible for installing the PapaAlfa proxy malware on a victim's machine. Using the same method as IndiaBravo-RomeoCharlie, IBPA will record the state of each step of the installation process to STDOUT. When captured to file, the output from IBPA after a successful installation appears as seen in Figure 3-4.

```
[2] - Extract Dll...
[1] - WriteConfig...
[3] - CreateSVC...
[4] – Success
```

Figure 3-4: IndiaBravo-PapaAlfa's Output from STDOUT

The most notable attribute of IBPA's output is the reordering of operations. IBPA performs `ExtractDll` prior to `WriteConfig`. The `WriteConfig` function will record the listening port as a 32-bit value within the file stored at `%SYSDIR%\pmsconfig.msi`. The listening port is specified by either supplying an optional command line argument or, if the argument is missing, using the default port 443.

The `ExtractDll` function stores the PapaAlfa binary at `%SYSDIR%\netmonsvc.dll`. The `CreateSVC` function will generate a new service named `NetMonSvc` and assign it to the newly created `LocalMonNetwork` service group before activating the service.

Just as IndiaBravo-RomeoBravo did, IBPA monitors the status of each of the installation tasks and, if any of the tasks fails, removes its binary by calling the suicide script, saved to `%TEMP%` directory as `cvrit000.bat`. The suicide script, seen below and identical to the one used by IndiaBravo-RomeoBravo, only removes the IBPA binary and does not attempt to clean up any of the successful steps that may have occurred during the installation process.

```
:L1
del "<full path to>wmisecsvc.dll"
if exist "<full path to>wmisecsvc.dll" goto L1
del <full path to>cvrit000.bat
```

Figure 3-5: IndiaBravo-PapaAlfa's Suicide Script

After completing the installation and activation tasks, IBPA calls the suicide script to remove itself from the victim's machine before terminating silently.

3.4 Swapping Out Payloads

Evidence suggests that IndiaBravo malware, regardless of the variant, are compiled and assembled independently of the payloads that they carry, as there are several observed samples that have compile dates significantly out of sync with the compile dates of their payloads. Additionally, there exists at least two known samples of IndiaBravo that have the same compile date but carry payloads with dates that are 11 days apart. It is highly likely that the IndiaBravo malware is compiled once and then reused for different payloads by simply using the compiled IndiaBravo binary as the foundation and appending the desired payload. This could easily be done using a simple script.

4. [Installer] IndiaCharlie

With samples observed as far back at late 2013, IndiaCharlie, the installer for RomeoFoxtrot, is an interesting departure from the other India malware families with respect to the cryptography used to obfuscate API names dynamically loaded by the malware. Up until late 2015, when the developer(s) appeared to abandon dynamic API loading, the dynamic API loading functionality used a C++ implementation of AES to decrypt the names of API functions. This particular implementation of AES comes from a Code Project post from 2002.[2] The use of AES encryption is somewhat rare within the collection of families found within the Lazarus Group. That said, the use of AES by the Lazarus Group is not unprecedented, as the same AES library is used by RomeoFoxtrot and SierraJuliett-MikeOne. What makes the use of AES particularly interesting with regards to IndiaCharlie is the use of AES decryption for the deobfuscation of API names during the dynamic API loading phase.

IndiaCharlie can operate as both a standalone installer/loader and as a standalone service executable. There are two variants of IndiaCharlie. Earlier versions, identified as IndiaCharlie-One, do not directly install a service for RomeoFoxtrot; rather, they merely drop and subsequently load the RomeoFoxtrot binary (as `%SYSDIR%\backSched.dll`). In early 2015, the IndiaCharlie malware was observed mutating into a new variant, identified as IndiaCharlie-Two, that not only drops the RomeoFoxtrot binary, but also installs the dropped binary as a service before activating the service. The change in the variants' methodologies corresponds to the changes in the RomeoFoxtrot functionality and the method by which RomeoFoxtrot initializes itself.

Both IndiaCharlie variants have common components and a sequence of events that occur. They begin by determining if the flag `-s` is present on the command line when the executable is activated. If found, the `-s` flag causes IndiaCharlie to activate in standalone mode, a mode that has slightly different meaning depending on the variant in question. If the `-s` flag is missing, IndiaCharlie initializes as a service executable, which consists of establishing a `ServiceMain` callback functions before calling the Windows API function `StartServiceCtrlDispatcherA` to start a new service. When activated as a service, IndiaCharlie establishes the framework of a legitimate Windows service that goes from `SERVICE_START_PENDING` to `SERVICE_RUNNING` and finally to `SERVICE_STOPPED`. During the state changes, the same sequence of functions that are called in standalone mode are called. Ultimately, if IndiaCharlie is run as a standalone executable or a standalone service, the same sequence of installation/loading functions are called, making the use of the Windows service scaffolding largely unnecessary.

The variants of IndiaCharlie share some common code attributes amongst themselves and the larger Lazarus malware collection. For instance, as part of the process that drops the payload malware to the victim's hard drive, the full directory path to the drop site is verified and constructed if necessary using the function described in Section 4.3.4 of the Operation Blockbuster report.[3] Also, whenever the suicide script is necessary, IndiaCharlie uses the same method and text for the suicide script as IndiaGolf (see 6.2.7). The suicide script generation takes the form seen in Figure 4-1.

2 George Anescu. "A C++ Implementation of the Rijndael Encryption/Decryption method" http://www.codeproject.com/Articles/1380/A-C-Implementation-of-the-Rijndael-Encryption-Decr 8 Nov. 2002

3 http://www.operationblockbuster.com/wp-content/uploads/2016/02/Operation-Blockbuster-Report.pdf

```
fp = fopen("d.bat", "w+");
if ( fp)
{
  fprintf(fp, ":R\nIF NOT EXIST %s GOTO E\ndel /a %s\nGOTO R\n:E\ndel /a d.bat", "%1",
"%1");
  fclose(fp);
  sprintf(cmdLine, "d.bat \"%s\"", &Filename);
  WinExec(cmdLine, 0);
```

Figure 4-1: Snippet of IndiaCharlie's Suicide Script Generator

The suicide script itself is somewhat unique in its approach. What makes this suicide script somewhat unique is that the name of the target file is not hardcoded into the batch file, but rather taken from the first argument of the command line executing the batch file. This technique is only used in a small number of the Lazarus Group's families.

4.1 IndiaCharlie-One

IndiaCharlie-One has compile dates ranging from at least late 2013 to early 2014 and is responsible for the delivery of the RomeoFoxtrot malware variant identified as RomeoFoxtrot-One. IndiaCharlie-One binaries perform the following steps as part of their payload loading:

1. Ensure the path to `%SYSDIR%` exists, create the hierarchy of folders if necessary
2. Copy the RomeoFoxtrot-One binary from a memory buffer within IndiaCharlie-One to the victim's computer at `%SYSDIR%\backSched.dll`
3. Set the `backSched.dll`'s file timestamp to match that of `mspaint.exe`'s.
4. Load `backSched.dll` by calling `LoadLibrary`
5. [If IndiaCharlie-One is activated as a service] Drop and execute the suicide script

IndiaCharlie-One is better classified as a loader than an installer given that the malware simply drops the RomeoFoxtrot-One malware image on the victim's machine, and it is up to the RomeoFoxtrot-One malware to perform the actual persistence operations. Given RomeoFoxtrot-One's functionality to install itself on a victim's machine as a service, the burden of establishing persistence is effectively lifted from IndiaCharlie-One. Instead, IndiaCharlie-One need only ensure that the `DllMain` of the RomeoFoxtrot-One malware is activated. To perform this task, a call to `LoadLibray` is made in order to the load the RomeoFoxtrot-One malware into memory and thereby activate the `DllMain` function.

While IndiaCharlie variants rely on AES to obfuscate the API function names, the embedded RomeoFoxtrot malware is actually stored within the IndiaCharlie-One binary in cleartext. Furthermore, the RomeoFoxtrot-One malware exists as a static buffer within IndiaCharlie-One's `.data` section, meaning that swapping out the malware may require a recompile of the IndiaCharlie-One binary. This links IndiaCharlie-One to the specific RomeoFoxtrot-One malware it is delivering.

When IndiaCharlie-One is activated as a service (which has the static identifier of `WMPNetworkSvcUpdate`), instead of standalone mode, a suicide script is generated and executed. This indicates that in service mode, the IndiaCharlie-One binary is expendable and should be removed after execution. The generation and execution of the suicide script occurs whether or not the loading of the RomeoFoxtrot-One malware is successful.

4.2 IndiaCharlie-Two

IndiaCharlie-Two is the installer for the RomeoFoxtrot variant identified as RomeoFoxtrot-Two and exhibits more variation over the course of its known life span than IndiaCharlie-One. The RomeoFoxtrot-Two malware lacks the necessary functionality to install itself as an active Windows service on the victim's computer, therefore it is the responsibility of IndiaCharlie-Two to not only drop and activate the RomeoFoxtrot-Two payload it contains, but to also establish its persistence. In both standalone mode and in service mode, IndiaCharlie-Two performs the following tasks in order to drop, install, and activate the RomeoFoxtrot-Two payload on a victim's machine:

1. Dynamically generate the service name, service display name, service DLL filename, and the service description
2. Ensure the path to location generated for the RomeoFoxtrot-Two binary (typically **`%SYSDIR%`**) exists, create the hierarchy of folders if necessary
3. Copy the RomeoFoxtrot-Two binary from a memory buffer within IndiaCharlie-Two to the victim's computer within the **`%SYSDIR%`** directory under the name generated in Task #1
4. Set the RomeoFoxtrot-Two's file timestamp to match that of **`mspaint.exe`**'s.
5. Install the Windows service using the generated parameters from Task #1
6. Activate the newly installed service
7. Open a hole within the victim's Windows firewall for the listening port of the newly activate RomeoFoxtrot-Two malware.
8. [If IndiaCharlie-Two was activated as a service] Drop and execute the suicide script

The timeline for IndiaCharlie-Two is not as easily determined as IndiaCharlie-One's. Two known samples of IndiaCharlie-Two have compile times of July 13, 2009 23:15. The associated RomeoFoxtrot-Two payloads within these two samples both have timestamps of July 14, 2009 01:05. While this may suggest that the malware date back to 2009, a closer inspection of the payload malware's export table reveals the more probably compile dates of January 13, 2015 07:41 and December 11, 2014 03:45. It is not uncommon for malware developers to alter the compile timestamp in the PE header of a file to obfuscate when the malware was originally developed, but more often than not when this is done to a DLL with exports, developers forget to modify the timestamp associated with the export table and thus leave behind the true compilation timestamp. Give the intimate relationship between IndiaCharlie-Two and its RomeoFoxtrot-Two payload, it is more likely the earliest observed IndiaCharlie-Two binaries date from mid-December 2014. This assertion is further supported by the first submitted dates on VirusTotal for the binaries with the July, 2009 compile dates showing the samples being submitted within hours of each other on January 13, 2015.[4][5]

IndiaCharlie-Two goes out of its way to attempt to blend into the existing Windows services. Task #1 is responsible for constructing the necessary information for the new service that will provide persistence for the RomeoFoxtrot-Two payload. In order to better mimic the victim's currently installed services, Task #1 begins by using the Windows API function **`EnumServicesStatusA`** to pull a list of all currently installed services. From this list of services, one service entry is randomly selected to become the template for RomeoFoxtrot-Two's service. IndiaCharlie-Two will take the service name, service display name, and service description values from the template service and modify them as identified in Table 4-1.

4 VirusTotal. "Antivirus Scan for 8fe806470914f9cdaaaa8448aa6317547c618efd65d15947767753fc88bc73d9 at 2015-01-13 06:39:09 UTC" https://www.virustotal.com/en/file/8fe806470914f9cdaaaa8448aa6317547c618efd65d15947767753fc88bc73d9/analysis/ 13 January 2015.

5 VirusTotal. "Antivirus Scan for 2b731d82b76f6d50a9d3fd72ac16e6fbb76779b57b114044bb61cd6e422f0cd0 at 2015-01-13 07:58:44 UTC" https://www.virustotal.com/en/file/2b731d82b76f6d50a9d3fd72ac16e6fbb76779b57b114044bb61cd6e422f0cd0/analysis/ 13 January 2015.

SERVICE DETAIL	ALTERATIONS
Service Name	Change template's name to all lowercase If contains "**`svc`**", replace "**`svc`**" with "**`mgr`**". Otherwise append "**`svc`**"
Service Display Name	From the template's display name, make the following substitutions: "**`Services`**" to "**`Provider`**" "**`Service`**" to "**`Manager`**" Find the last space and overwrite last word with "**`Service`**" If there are no spaces, append "**`Service`**"
Service Description	Concatenate the new service's display name with "is an essential element in Windows System configuration and management." Append the template's description

Table 4-1: IndiaCharlie-Two's Service Detail Modification Steps

If for any reason IndiaCharlie-Two is unable to access any of the services, and therefore unable to access the name, display name, and description of an installed Windows service on the victim's machine, IndiaCharlie-Two falls back to a set of default values seen in Table 4-2.

SERVICE DETAIL	VALUE
Service Name	fsbrmgr
Service Display Name	File System Backup and Restore Manager
Service Description	File System Backup and Restore Manager is used by Windows Backup to perform backup and recovery operations. If this service is stopped by a user, it may cause the currently running backup or recovery operation to fail. Disabling this service may disable backup and recovery operations using Windows Backup on this computer.

Table 4-2: IndiaCharlie-Two's Failback Service Details

Sometime between April 2015 and late-August 2015, the default values for IndiaCharlie-Two changed to the value seen in Table 4-3.

SERVICE DETAIL	VALUE
Service Name	drvins
Service Display Name	Driver Install Manager
Service Description	Driver Install Manager is used by Windows Dirver Install to perform backup and recovery operations. If this service is stopped by a user, it may cause the currently running backup or recovery operation to fail. Disabling this service may disable backup and recovery operations using Windows Backup on this computer.

Table 4-3: IndiaCharlie-Two's Post-April 2015 Fallback Service Details

The name for the RomeoFoxtrot-Two binary is the combination of the generated service name and the extension **`.dll`**. The location of the RomeoFoxtrot-Two malware is generated by calling the API function **`GetSystemDirectoryA`**, but, if that call fails, IndiaCharlie-Two defaults to **`C:\windows\system32`**.

IndiaCharlie-Two uses the exact same function, sans the **LoadLibrary** call, that IndiaCharlie-One uses for extracting the RomeoFoxtrot-Two binary from its **.data** section and writing the file to disk. Just as IndiaCharlie-One does not have an encrypted payload, IndiaCharlie-Two's RomeoFoxtrot-Two payload is in cleartext with the exception of two known samples that use simple XOR encoding. These two samples are believed to date from mid-Dec 2014 and are the same samples that have the likely false July 2009 compile dates discussed above.

IndiaCharlie-Two installs the RomeoFoxtrot-Two malware as a service on the victim's machine (Task #5) as an svchost-dependent service. IndiaCharlie-Two begins Task #5 by creating a new registry key under **HKLM\SYSTEM\CurrentControlSet\Services\** with the generated name of the new service. The **Parameters\ServiceDll** value is set to point to the RomeoFoxtrot-Two binary on the victim's machine. A new service group is added to the registry under **HKLM\Software\Microsoft\Windows NT\CurrentVersion\SvcHost**. The generated service name serves as both the new service group's name and its only member. The newly generated service is added to the Windows service database by calling **CreateServiceA** which also adds the description of the service. Lastly, **ChangeServiceConfig2A** is called twice, once to add the description of the service and again to establish the behavior of the service in the event of an error (which is to simply restart the service). Finally, IndiaCharlie-Two activates the service by calling **StartServiceA** (Task #6).

RomeoFoxtrot-Two is a server-mode RAT, and so it requires a listening port on the victim's machine. In order to allow externally originating network connections, the Windows firewall must be modified by IndiaCharlie-Two to allow incoming connections. IndiaCharlie-Two attempts to poke a hole in the victim's Windows firewall using two methods at the same time (Figure 4-2).

```
cmd.exe /c netsh advfirewall firewall add rule name="<service's name>" dir=in action=allow
service="<service's name>" enable=yes
cmd.exe /c netsh firewall addallowedprogram "c:\windows\system32\svchost.exe -k <service's
name>" "<service name>" enable
```

Figure 4-2: Commands used by IndiaCharlie-Two to Open Ports on the Victim's Firewall

The first **netsh** command allows the service to bind to any port (that is available) and the Windows firewall then allows incoming connections to the bound port. This command is specific to Windows versions newer than Windows XP. The second **netsh** command appears to attempt to perform the same task to grant service access through the firewall of Windows XP computers, but a typo in the command ("**addallowedprogram**" instead of "**add allowedprogram**") prevents the command from working.

When IndiaCharlie-Two is activated as a service or as a standalone executable, a suicide script is generated and executed at the conclusion of the installation process. As with IndiaCharlie-One, the IndiaCharlie-Two binary is expendable and should be removed after execution based on the use of the suicide script. And, as with IndiaCharlie-One, the generation and execution of the suicide script occurs whether or not the loading of the RomeoFoxtrot-Two malware is successful.

An interesting shift in design occurred between April 2015 and late August 2015 with regards to how information leaking strings are addressed. Prior to this, the values for the default service details as well as the strings used to construct the firewall disabling commands are in cleartext in IndiaCharlie-Two samples. But at the same time that the default service details change, the developers begin encrypting the strings for the default service details as well as the component that make up the firewall deactivation commands. The developers encrypt these important strings by applying a 64-byte XOR buffer to the strings as seen in Figure 4-3.

```
unsigned char decoderMatrix[64] =  {64, 66, 35, 69, 121, 115, 78, 41, 39, 101, 51, 52, 39,
52, 55, 103, 65, 69, 101, 89, 111, 112, 106, 121, 94, 101, 98, 64, 89, 61, 66, 55, 69, 39,
105, 80, 33, 106, 84, 121, 33, 35, 101, 108, 51, 87, 112, 78, 42, 64, 72, 57, 113, 110, 35,
68, 62, 93, 48, 103, 55, 101, 51, 108};
for ( i = 0; i < dwLength; ++i )
  pszString[i] ^= decoderMatrix[i % 64];
```

Figure 4-3: IndiaCharlie-Two's String Decoder Code Snippet

The use of such a simple, but effective, encoding scheme is an interesting departure from the use of AES. As a matter of fact, the developers completely abandon AES in three known IndiaCharlie-Two samples along with the dynamic API loading functionality. IndiaCharlie-Two variants use only a small subset of the API functions that are dynamically loaded, and, even then, they are rarely used in place of their imported twins (e.g. `CreateFileA` from the import table is used instead of the dynamically loaded `CreateFileA`). The two known samples with incorrect compile dates and the sample that changes the default service detail values all forego the use of dynamic API loading and, as a result, AES encryption. It is unclear why these particular samples depart from the form of the other IndiaCharlie-Two variants.

5. [Installer] IndiaDelta

Consisting of a single known member, IndiaDelta is a simplistic installer responsible for dropping then activating the LimaAlfa loader (see Section 14) as well as dropping the WhiskeyCharlie wiper and its command file. While the general functionality the malware provides is simple (drop files and activate the next stage), the steps are somewhat convoluted.

The operating mode of IndiaDelta is determined by the number of command line arguments that are supplied to the executable when it is activated. If there are less than two command line arguments, the program quietly terminates. If there are only two command arguments, IndiaDelta enters the *relocation and re-initialization* mode. If there are only four arguments, IndiaDelta enters the *installation and activation* mode. Any other combination of command line arguments results in IndiaDelta terminating quietly.

Both of IndiaDelta's modes begin by dynamically loading API functions. Unlike the majority of other code within the Lazarus Group's collection, the dynamic API loading functionality is done without obfuscation of the API names. Rather, IndiaDelta performs a series of `GetProcAddress` calls with very little fanfare.

The *relocation and re-initialization* mode is responsible for, as the mode's description suggests, relocating the IndiaDelta binary to the victim's `%TEMP%` directory and re-activating itself from within that location. The task of relocation consists of locating the `%TEMP%` directory on the victim's machine, deleting any file within that directory that contains the same filename as the currently active IndiaDelta binary, and then copying the original IndiaDelta binary to the victim's `%TEMP%` directory. The re-initialization task generates a new set of command line arguments with the following format:

```
/install <name of the original IndiaDelta binary> <original command line arguments>
```

The new instance of IndiaDelta is activated by passing the new IndiaDelta's filename and the new command line to `CreateProcess`. The original IndiaDelta instance then terminates with the exit code set to `1`.

The *installation and activation* mode is the workhorse mode of the malware. The mode begins by verifying that the first command line argument is equal to `/install` as set by the *relocation and re-initialization* mode. IndiaDelta's main thread continues as a Message Loop[6] to support the unnecessary Windows GUI subsystem that IndiaDelta operates under while the actual installation and activation tasks are spun off into their own thread. The first task this mode performs is to set the current working directory to the directory that contained the original IndiaDelta binary (the location of IndiaDelta at the time of the *relocation and re-initialization* mode). Next, IndiaDelta begins the process of extracting its embedded files to disk.

IndiaDelta stores its embed as a sequence of {header, file data} blocks appended to the end of its own binary image. The last 8 bytes of the IndiaDelta binary contains two 32-bit values, each XOR'd with the 32-byte value `0x1234567` (in little endian). The first 32-bit value defines the number of embedded blocks contained within the installer, and the second (or last) 32-bit value defines the offset to the start of the embedded blocks.

Each embedded block contains a header that consists of 20 bytes that define the filename of the embedded file followed by a 32-bit value that defines the size of the embedded file image (but not necessarily the final size of the embed). The

6 Microsoft. "About Messages and Message Queues" https://msdn.microsoft.com/en-us/library/windows/desktop/ms644927(v=vs.85).aspx Accessed 8 November 2015

embedded file's image is BZip-compressed, which requires IndiaDelta to decompress each file upon extraction. Figure 5-1 illustrates the layout of IndiaDelta's embed stacking scheme.

The extraction process for each embedded file begins by reading in the header, determining the number of bytes to extract from the IndiaDelta image, and then saving those bytes (following the header) to the **`%TEMP%`** directory with a filename that begins with **`zz`** followed by some random digits as generated by a call to **`GetTempFileNameA`**. Once the file has been written to disk, the BZip decompression function is called to inflate the binary and save the newly expanded file to the name specified within the embedded file's header. The **`zz`** file is then securely deleted using a secure deletion function common to the Lazarus Group.

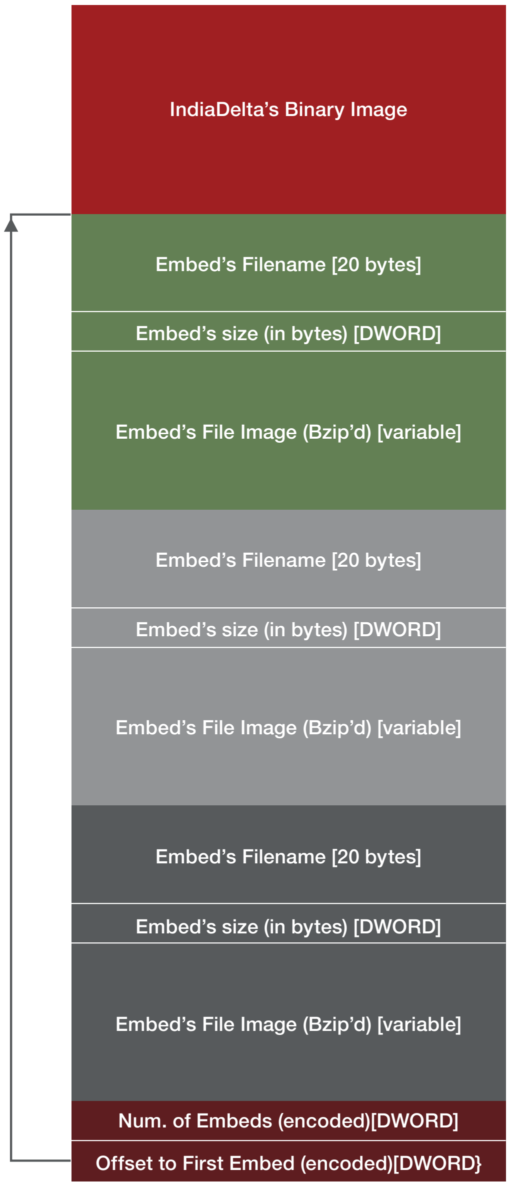

IndiaDelta's Binary Image
Embed's Filename [20 bytes]
Embed's size (in bytes) [DWORD]
Embed's File Image (Bzip'd) [variable]
Embed's Filename [20 bytes]
Embed's size (in bytes) [DWORD]
Embed's File Image (Bzip'd) [variable]
Embed's Filename [20 bytes]
Embed's size (in bytes) [DWORD]
Embed's File Image (Bzip'd) [variable]
Num. of Embeds (encoded)[DWORD]
Offset to First Embed (encoded)[DWORD}

Figure 5-1: IndiaDelta's Embed Stacking Scheme

Once all of the embedded files are extracted and decompressed, IndiaDelta enters an infinite loop that only terminates after the secure delete function is able to successfully remove the original IndiaDelta's file from the victim's computer. Effectively, this is an overly complicated suicide script that doesn't use a batch file. This is also a key characteristic of IndiaDelta and the malware it drops, as each subsequent stage attempts to delete the previous stage.

At the completion of the installation and activation mode, IndiaDelta attempts to move the first extract file (the LimaAlfa binary) to the same name as the original IndiaDelta. LimaAlfa is then executed by a call to `CreateProcess` with the following command line arguments:

```
<name of current IndiaDelta binary> <original first argument> <original second argument>
```

IndiaDelta does not use or care about the contents of the first and second arguments when IndiaDelta is originally called, beyond the fact that arguments are present. Throughout both the *relocation and re-initiation mode* and the *installation and activation mode*, the original command line arguments are merely carried forward to the next mode and ultimately the next stage (LimaAlfa).

6. [Installer] IndiaEcho

A classic example of open-source code usage, the installer IndiaEcho borrows heavily from the project "Pure WIN32 Self-Extract EXE Builder" by the CodeProject user CT Chang,[7] specifically the function `ExtractBinaryFile` from the SetupEx sub-project. While SetupEx uses the Windows GUI interface to provide feedback to the user about the progress of installation activities, IndiaEcho provides absolutely no feedback to the victim, given that the task is the installation of malware. The developer(s) of IndiaEcho leverage the file extraction technique of SetupEx while foregoing the GUI components.

There are two known variants of IndiaEcho in the wild: IndiaEcho-One and IndiaEcho-Two. IndiaEcho-One is the full-featured base model, while IndiaEcho-Two greatly simplifies the installation operation. Both variants drop files from their respective payload sections in the same manner, but IndiaEcho-One also installs a Windows service before generating and executing a suicide script. In addition to their operational differences, the variants also drop different members of the Lazarus Group's malware: IndiaEcho-One drops LimaBravo (see Section 15) and RomeoGolf along with a configuration file, while IndiaEcho-Two drops IndiaBravo-RomeoBravo (see Section 3.1).

7 CT Chang. "Pure WIN32 Self-Extract EXE Builder". http://www.codeproject.com/Articles/7053/Pure-WIN-Self-Extract-EXE-Builder 2004 May 16

6.1 IndiaEcho-One

IndiaEcho-One loads API functions from **kernel32.dll**, **urlmon.dll**, **userenv.dll**, **advapi32.dll**, **ws2_32.dll**, and **wininet.dll**. If the entirety of the 53 API functions that IndiaEcho-One attempts to load do not successfully load, IndiaEcho-One generates and executes a suicide script without performing any further operations. Of the 53 API functions that IndiaEcho-One loads, only 7 of the APIs are actively used by the malware; instead, the bulk of the API function calls made by IndiaEcho-One during its operation come from the same API functions loaded as part of the import table load operation when the binary is initially loaded into memory by Windows.

IndiaEcho-One requires two temporary files for the process of extracting the payload components from itself. The developer(s) of IndiaEcho-One used the source code verbatim from SetupEx's **ExtractBinaryFile** to generate these two filenames and ensure that there are no conflicting files with the same name within the victim's **%TEMP%** directory (Figure 6-1).

```
GetTempPath(sizeof(szWinTmpPath), szWinTmpPath);
sprintf(szTmpBinFile1, "%s%08X", szWinTmpPath, GetTickCount());
sprintf(szTmpBinFile2, "%s%08X", szWinTmpPath, (DWORD)(GetTickCount()*12345)/8725);
DeleteFile(szTmpBinFile1);
DeleteFile(szTmpBinFile2);
```

Figure 6-1: SetupEx's Temporary Filename Generation Code Snippet from ExtractBinaryFile

IndiaEcho-One stores its payload within the **SETUP\2000** resource entry of its own binary. The contents of the resource are written to the **szTmpBinFile1** specified path and then read into memory for parsing. With the contents of the payload file now in memory, the file from which they originated (specified by **szTmpBinFile1**) is deleted. The first 268 bytes of the file specify the number of individual drop files that exist within the payload along with a string. The SetupEx project specifies the format for the first 268 bytes as seen in Figure 6-2 (from the SetupEx source code).

```
typedef struct tagSETUPINFO
{
    DWORD      dwFileCount;
    char       szAutoExecFile[MAX _ PATH];
} SETUPINFO, FAR * LPSETUPINFO;
```

Figure 6-2: SetupEx's SETUPINFO Structure Definition

The **szAutoExecFile** string, while populated, has no purpose in IndiaEcho-One. **dwFileCount** specifies the number of individual drop files that exist within the payload memory structure. There are **dwFileCount** number of structures named **EXTRACTFILEINFO** that define the size, name, and timestamps of the individual files that follow the **SETUPINFO** structure. The **EXTRACTFILEINFO** structure is defined as seen in Figure 6-3 (from the **SetupEx** source code).

```
typedef struct tagEXTRACTFILEINFO
{
    // Running index value of the current file out of the
    // total distributed file count.
    DWORD           dwIndex;
    // Original file created time.
    FILETIME        CreateTime;
    // Original file last read/written time.
    FILETIME        LastAcessTime;
    // Original file last written time.
    FILETIME        LastWriteTime;
    // Specifies the high-order DWORD value
    // of the file size, in bytes.
    DWORD           dwFileSizeHigh;
    // Specifies the low-order DWORD
    // value of the file size, in bytes.
    DWORD           dwFileSizeLow;
    // A null-terminated string that
    // is the name of the original file.
    char            szBinFileName[MAX _ PATH];
} EXTRACTFILEINFO, FAR * LPEXTRACTFILEINFO;
```

Figure 6-3: SetupEx's EXTRACTFILEINFO Structure Definition

IndiaEcho-One uses the **EXTRACTFILEINFO** structures to extract the individual drop files that follow the last **EXTRACTFILEINFO** structure. The extraction process for each file begins by generating a filename for the extracted file based on the **szBinFileName** field and placing the file in the **%TEMP%** directory. IndiaEcho-One modifies the first two bytes of the file being extracted if they match **JE** so that the first bytes become **MZ**. The extracted file is written to the victim's hard drive, and its timestamp is set according to the **CreateTime**, **LastAcessTime**, and **LastWriteTime** fields from the file's **EXTRACTFILEINFO** structure.

After extracting and dropping the files within its payload, IndiaEcho-One installs a service to support the extracted LimaBravo binary. IndiaEcho-One has an entire service installation subsystem responsible for installing the dropped LimaBravo binary as a Windows system service. From the onset, IndiaEcho-One establishes the necessary information to establish a service, with known samples using the values in Table 6-1 to describe the service to install.

Service Name	Mwsagent
Service Display Name	Mobile Communication Agent Service
Service Description	Allows mobile service to commnicate
Service DLL	Mwsagent.dll

Table 6-1: Known Service Description Field Values for IndiaEcho-One

Before the service can be installed, the service DLL must be prepared and relocated. IndiaEcho-One moves the file from its dropped location in **%TEMP%** to the **%SYSDIR%** directory and mimic the timestamp of **mspaint.exe**. Next, it installs and activates the service that ensures LimaBravo's persistence on the victim's machine. This task is relatively straightforward:

1. Create a new service group under **HKLM\Software\Microsoft\Windows NT\CurrentVersion\Svchost** for the new service named **Mwsagent**
2. Call **CreateServiceA** to add the new service to the service's database
3. Call **ChangeServiceConfig2A** to set the service's description
4. Create a registry key at **HKLM\SYSTEM\CurrentControlSet\services\Mwsagent**
5. Set the **Parameter\ServiceDll** value to point to the dropped LimaBravo binary
6. Call **StartServiceA** to activate the service

Next, IndiaEcho-One attempts to purge itself from the victim's machine by generating and executing a suicide script. The suicide script (Figure 6-4) is dropped as **%TEMP%\msgl.bat**.

```
:Repeat
del "<IndiaEcho-One binary>"
if exist "<IndiaEcho-One binary>" goto Repeat
del msgl.bat
```

Figure 6-4: IndiaEcho-One's Suicide Script

6.2 IndiaEcho-Two

The sample set of IndiaEcho-Two is limited to a single observed sample from September 2014. The sample mimics a screensaver installer called "`sunny-leone-II-screensaver Setup File`" by dropping both the legitimate `install-sunny-leone-II-screensaver.exe` executable and an IndiaBravo-RomeoBravo sample.

IndiaEcho-Two, from a structural perspective, has some interesting artifacts not found in the IndiaEcho-One variants. The resource section of binary may contain references to the language used for various text strings found in dialogues and file information data. These language identifiers are typically set to match the language of the compiling computer or the computer upon which the source code was originally developed. IndiaEcho-Two contains three different language identifiers: the icons within the resource section identify themselves as Dutch, the file information section uses the English language, and an unused dialogue box defined within the resource section specifies Korean as its language. The language associated with the icons is explainable based on the fact that they are lifted from the Inno Setup installer which, itself, has the language for the icon specified as Dutch. The English language of the file information section is most likely a product of the fact that the file information was lifted directly from another installer. This only leaves the Korean language for the dialogue without an immediately recognizable source outside of the original development machine's default language possibly being Korean.

As IndiaEcho-Two uses the same dynamic API loading function as IndiaEcho-One, IndiaEcho-Two loads the same 53 API functions. Similarly, if the entirety of the 53 API functions that IndiaEcho-Two attempts to load do not successfully load, it silently terminates without performing any further operations, as IndiaEcho-Two does not use a suicide script upon termination. Despite this insistence on dynamically loading API functions, IndiaEcho-Two does not use a single one of the loaded API functions, instead using the same API functions loaded as part of the import table loading operation.

IndiaEcho-Two uses the same file extraction routine found in IndiaEcho-One, with the exception that upon successful extraction, regardless of its type, each file to the victim's hard drive is passed to `CreateProcess` in an attempt to execute the file. Since the observed IndiaEcho-Two sample only drops the screensaver installer and the IndiaBravo-RomeoBravo installer, both of which are standalone executables, simply calling `CreateProcess` for each of the files is by and large one of the simplest methods for activating both components. Calling `CreateProcess` is a gross deviation from the SetupEx source code and therefore is a feature, like the `JE/MZ` substitution, unique to the IndiaEcho (specifically IndiaEcho-Two) code base.

Since both dropped files are standalone, executable installers, the need for the service installation component found in IndiaEcho-One is no more; as such, IndiaEcho-One's code for the service installation subsystem has been removed by the developers. Also, given that IndiaEcho-Two is masquerading as a screensaver installer, the use of a suicide script to delete the source file would turn out to be more suspicious than leaving the file in place, so the suicide script component of IndiaEcho-One is not found in IndiaEcho-Two. The removal of the service installation subsystem, the suicide script generation and execution, and the wholesale execution of the dropped files as they are dropped significantly streamlines the code and functionality of IndiaEcho-Two compared to IndiaEcho-One. This makes the inclusion of unnecessary artifacts revealing the probable native language of the developer somewhat baffling.

7. [Installer] IndiaFoxtrot

IndiaFoxtrot is an installer that drops, installs, and activates the combination of DeltaBravo and RomeoDelta. Activation of IndiaFoxtrot can include 1 or 4 additional command line arguments. If present, the first argument on the command must be `-i` otherwise IndiaFoxtrot will simply terminate after executing its suicide script.

Before IndiaFoxtrot begins the installation process, it first verifies that the named mutex `Global\WindowsUpdateTracing0.5` — an indication of an existing RomeoDelta infection — does not exist on the victim's system. After loading only two Windows DLLs (`kernel32.dll` and `advapi32.dll`) via a dynamic API loading scheme that relies on Space-Dot obfuscation, IndiaFoxtrot loads an encrypted data blob from its own resource section into memory, where it applies a RC4 transformation to decrypt the data blob into the structure payload data.

The format of the structured payload data is the same as the format for IndiaWhiskey's payload data (see Section 13), with the exception of the first four bytes. In order to determine if the RC4 transformation resulted in correct decryption, the first four bytes of the data must equal `ZBI` followed by a NULL (`0`) character. For each file found within the structured payload data, IndiaFoxtrot writes the file to the `%SYSDIR%` directory. Any DLL found within the payload data is given a randomized filename, while other file types are given the name described in the file's payload header.

After dropping the files within the payload data, IndiaFoxtrot installs the RomeoDelta DLL as a service. IndiaFoxtrot will randomly construct the name of the service and the service's description in the same manner as the installer IndiaGolf (see Section 8) and, to a lesser degree, IndiaBravo-RomeoBravo (see Section 3.1). IndiaFoxtrot uses randomly selected indices into a set of hardcoded strings to construct the service's name, display name, and description. The three sets of hardcoded strings consist of a set of actions (Figure 7-1), a set of targets (Figure 7-2), and a set of descriptions (Figure 7-3).

Enables	Allows	Provides
Supports	Monitor	

Figure 7-1: IndiaFoxtrot's Service Action Words

Audio Service	Basic CAPI SubSystem	Account Manager	Internet Security
Power Manager	Software Licence	Cryptograpic API	System Event Manager
Dialup Networking	Display Manager	Remote Access Protocol	ClamAV Updater
Configuration	Network FileShare	BitLocker Encryption	Secure FileSystem
Windows MediaPlayer			

Figure 7-2: IndiaFoxtrot's Service Targets

If this service is stopped, system cannot be well performed
This service cannot be stopped.
If this service is disabled, any services that explicitly depend on it will fail to start.

Figure 7-3: IndiaFoxtrot's Service Description Framing Strings

The service name is constructed by selecting one of the target strings. The description is constructed by using the format

```
<action> <target>.<description>
```

where the <target> is the same as the value used for the service's name. The description strings are identical to the description strings found in IndiaGolf.

After generating the service information, a new service is constructed by generating new registry entries to make the RomeoDelta binary a svchost-based service DLL. With the registry and operating system now configured to treat the RomeoDelta binary as a service DLL, IndiaFoxtrot activates the service by calling **`StartService`** API function.

If the file drop or service activation steps fail, IndiaFoxtrot attempts to securely delete any artifacts it may have produced before calling the suicide script and terminating.

If IndiaFoxtrot is activated with only the **`-i`** command line argument (or activated without any arguments), IndiaFoxtrot activates the DeltaBravo binary by calling **`CreateProcess`** after appending **`-reg`** to the command line supplied to the DeltaBravo binary. Otherwise, if IndiaFoxtort is activated with four additional arguments (including the **`-i`** argument), DeltaBravo is activated with the following command line arguments:

```
0 <IndiaFoxtrot's 2nd argument> <IndiaFoxtrot's 3rd argument> <IndiaFoxtrot's 4th argument>
```

Once IndiaFoxtrot has completed its task, it generates a suicide script named **`AUTOEXEO.bat`**. The suicide script is executed via a call to **`CreateProcessA`**, and IndiaFoxtrot silently terminates.

8. [Installer] IndiaGolf

Observed as part of the 10 Days of Rain campaign,[8] IndiaGolf is a DLL-based installer responsible for the installation of two to four pieces of malware and their respective configuration files. There are three observed variants of IndiaGolf; these variants differ primarily only in their payloads and exported activation function names (identified in Table 8-1), though there are some minor functional changes that will be addressed later in this section. All three observed variants drop RomeoMike and DeltaAlfa.

INDIAGOLF VARIANT	PAYLOADS	ACTIVATION EXPORT
IndiaGolf-One	RomeoMike, DeltaAlfa, WhiskeyBravo	`StartInstall`
IndiaGolf-Two	RomeoMile, DeltaAlfa, WhiskeyBravo, TangoBravo	`_gdimain`
IndiaGolf-Three	RomeoMike, DeltaAlfa	`UICreate`

Table 8-1: IndiaGolf Variants, Their Payloads and Their Activation Exports

Activation of IndiaGolf requires a loader that loads the IndiaGolf binary into memory and calls the appropriate export, referred to as the "activation function." Evidence suggests that LimaDelta (see Section 17) is responsible for these tasks. The connection between LimaDelta and IndiaGolf is assumed based on the following characteristics:

1. The compilation dates for LimaDelta and IndiaGolf samples are within hours or, in some cases, minutes of each other.
2. The LimaDelta samples that have close compilation times to IndiaGolf ones load a binary and call an export with the same name as the activation export found in the IndiaGolf samples.
3. The size of the data structure passed from the LimaDelta members from characteristic #2 matches the expected data structure for the corresponding IndiaGolf binary.

While the particular payload components that the various IndiaGolf variants install may change from variant to variant, outside of RomeoMike and DeltaAlfa, a basic design pattern exists for all of the variants. For any payload component that requires a Windows service to exist for the payload to operate, IndiaGolf performs the following tasks:

1. Establish the necessary configuration parameters for the malware being dropped
2. Write the necessary configuration file(s) to the victim's hard drive using the names specific to the particular piece of malware being deployed
3. Decompress the payload image using a Zlib library
4. Generate the parameters that define the service for the payload malware
5. Install and activate the service for the payload malware

The activation function takes a single parameter: a memory pointer containing the configuration necessary for MikeRomeo. Once called, the activation function passes control to an installation function, which uses the memory pointer not only as the configuration for the RomeoMike component to be installed, but as an indicator for if the RomeoMike component should be dropped at all. In order for IndiaGolf to drop, install, and activate RomeoMike, the memory pointer must not be NULL.

8 Ten Days of Rain: Expert analysis of distributed denial-of-service attacks targeting South Korea." McAfee. 2011. http://www.mcafee.com/us/resources/white-papers/wp-10-days-of-rain.pdf

Understanding the installation and activation process for RomeoMike is necessary in order to understand how additional payload components are installed and activated. The installation of RomeoMike begins (Task #1) by first establishing a 64-bit identification number for the victim's machine. The identification number will come from one of two places: if `GetAdaptersInfo` returns a valid result, the identification number is calculated as the 6-byte MAC address of the first NIC plus 4 bytes from `rand` or, failing the API call, the concatenation of four rand calls. The configuration file is stored as `%SYSDIR%\faultrep.dat`. It should be noted that the name of the configuration file is stored in plaintext within IndiaGolf despite the same string being encrypted with AES within RomeoMike. Before the configuration is dropped to disk (Task #2), the directory hierarchy verification and generation function found in many of the Lazarus Group's malware samples is used to ensure that the path exists (although, given it is `%SYSDIR%`, it should clearly already exist). Once the file is written to disk, a randomly generated timestamp is applied to the file.

The RomeoMike binary is stored within the `.data` segment of IndiaGolf as a Zlib compressed byte array. To extract the binary (Task #3), a memory buffer is allocated on the heap and passed to a Zlib decompression routine along with the expected size of the file after decompression, the compressed byte array, and the size of the compressed byte array. If successfully decompressed, the RomeoMike image is housed within the previously allocated heap memory buffer.

RomeoMike requires that it run as a service DLL. IndiaGolf accommodates this requirement by first establishing the basic information for RomeoMike's new service (Task #4). The generation of the service name, the service's display name, the service description, and the name of the RomeoMike DLL is done using a set of randomly generated indices into an array of hardcoded values in much the same way that IndiaBravo-RomeoBravo (see Section 3.1) generates service information. The service information generator contains three different arrays—one array of actions (Figure 8-1), one array of service targets (Figure 8-2), and one array of description framing strings (Figure 8-3).

Provides	Manages	Retrieves	Monitors
Enables	Configures	Adds	Manages
Detects	Supports	Maintains	Creates
Allows	Transmits	Creates	Contains

Figure 8-1: IndiaGolf's Service Action Words

Storage protect	Application information	Wmi adapter	Efficent game controler
Removal stoarage	Com library	Mpeg adapter	Comfortable game controler
Account information	Distributed Link Tracking Client	Vga adapter	Comfortable Desktop Controler
Remote user	Cryptograpic file	Wireless protection	Beautiful Background Color
Remote account	Network account	DNS resolve	Beautiful Windows background
System information	Logon information	Security Account	DNS Security controler
Security inforamion	Security catalog	Security Inernet Bank	Media security container

Figure 8-2: IndiaGolf's Service Targets

If this service is stopped, system cannot be well performed
This service cannot be stopped.
If this service is disabled, any services that explicitly depend on it will fail to start.

Figure 8-3: IndiaGolf's Service Description Framing Strings

To generate the service information, one entry from each of the three arrays is selected at random and applied to the patterns identified in Table 8-2. While the methodology around generating seemingly random but realistic service information is largely sound, the high number of spelling errors in the service target strings makes the services much more obvious, even to a novice system administrator. It is unclear why the developer(s) of IndiaGolf did not bother to spell check their work.

FIELD	PATTERN
Service Display Name	{Service Target String} Service
Service Description	{Action Word} {Service Target String}. {Framing String}
Service Name	{4 random characters from service description}svc
Service DLL Filename	%SYSDIR%\{Service Name}.dll

Table 8-2: IndiaGolf's Service Information Generation Patterns

After generating the service information, the RomeoMike image is written to disk and given a randomized timestamp. The service is then established (Task #5) by performing the following step:

1. Create a new service group under `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Svchost` for the new service
2. Call `CreateServiceA` to add the new service to the services database
3. Call `ChangeServiceConfig2A` to set the service's description
4. Create a registry key at `HKLM\SYSTEM\CurrentControlSet\services\`<name of new service>
5. Set `Parameter\ServiceDll` value appropriately
6. Call `StartServiceA` to activate the service

Following the installation and activation of RomeoMike, IndiaGolf will repeat the tasks again to drop, install, and activate DeltaAlfa. The only different between the installation and activation of DeltaAlfa in comparison to RomeoMike is the effort required to install the two necessary configuration files for DeltaAlfa (Task #1): an attack activation configuration and an attack targeting configuration. IndiaGolf supplies both of the configuration files. The attack targeting configuration is dropped (in its encrypted form) from a memory buffer within IndiaGolf's `.data` section while the attack activation configuration is potentially modified prior to being dropped. If the current time is later than the hardcoded DeltaAlfa activation time, then the activation is adjusted to one day after the installation date. In other words, if the activation time specified within IndiaGolf were March 6, 2011 at 9:30 AM, and IndiaGolf were executed on October 8, 2015 at noon, IndiaGolf would supply DeltaAlfa with an attack activation configuration specifying October 9, 2015 at noon. For each of the DeltaAlfa configuration files that IndiaGolf drops, IndiaGolf generates a new randomized timestamp to assign to the files.

Depending on the particular files that an IndiaGolf variant may drop, the tasks for dropping the configuration, decompressing the binary to drop, and installing another service with newly generated service details is repeated for each file that is dropped. Once all of the files that the IndiaGolf variant is tasked by the developer(s) to drop have been successfully installed and activated, IndiaGolf's activation function will return and thus terminate the activity associated with IndiaGolf.

8.1 IndiaGolf-One

The activation function (which is exported as `StartInstall`) of IndiaGolf-One contains additional functionality besides simply calling the installation function. The first operation that IndiaGolf-One performs is the modification of the victim's `%SYSDIR%\drivers\etc\hosts` file in exactly the same way that TangoBravo performs the task. In addition, the functions within IndiaGolf-One responsible for the modification of the hosts file are the same as those found in TangoBravo, indicating that TangoBravo may simply be an offshoot of IndiaGolf-One. The targeted domains found in IndiaGolf-One are a subset of the targeted domains found in TangoBravo:

- explicitupdate.alyac.co.kr
- gms.ahnlab.com
- ko-kr.albn.altools.com
- ko-kr.alupdatealyac.altools.com
- su.ahnlab.com
- su3.ahnlab.com
- update.ahnlab.com
- ahnlab.nefficient.co.kr

IndiaGolf-One drops three different malware families in the following order: RomeoMike, DeltaAlfa, and WhiskeyBravo-One. The task of dropping RomeoMike and DeltaAlfa occurs in the manner described previously. Likewise, the task of dropping, installing, and activating WhiskeyBravo-One follows the same pattern as the RomeoMike and DeltaAlfa installation and activation. IndiaGolf generates the configuration of WhiskeyBravo-One by setting the installation time field (offset 0) to the current system time on the victim's machine in `VARIANTTIME` form and the activation delay (offset 8) to 7 days.

IndiaGolf-One generates and executes a suicide script after its tasks are completed. The suicide script generation uses `fprintf` to write the entire script to disk as `d.bat` in the current working directory of IndiaGolf-One before calling `CreateProcess` to execute the script. The following code fragment illustrates the method of suicide script generation and type of suicide script generated for IndiaGolf-One:

```
fp = fopen("d.bat", "w+");
if ( fp )
{
  fprintf(fp, ":R\nIF NOT EXIST %s GOTO E\ndel /a %s\nGOTO R\n:E\ndel /a d.bat", "%1",
"%1");
  fclose(fp);
  sprintf(cmdLine, "d.bat \"%s\"", szIndiaGolfFilename);
  CreateProcessA(0, cmdLine, 0, 0, 0, 0x8000000u, 0, 0, &StartupInfo,
&ProcessInformation);
}
```

Figure 8-4: IndiaGolf-One's Suicide Script Generation and Execution Code

8.2 IndiaGolf-Two

IndiaGolf-Two is nearly identical to IndiaGolf-One with the exceptions that IndiaGolf-Two replaces the **`%SYSDIR%\drivers\etc\hosts`** file modification function with a dropped TangoBravo executable and the activation function is exported as **`_gdimain`**. The TangoBravo executable is stored in the **`.data`** section of IndiaGolf and written to disk as **`%SYSDIR%\rtdrvupr.exe`** with a randomly generate timestamp. IndiaGolf-Two activates the dropped TangoBravo by calling **`WinExec`**.

8.3 IndiaGolf-Three

IndiaGolf-Three significantly streamlines the number of operations it performs. The activation function, exported as **`UICreate`**, does not drop TangoBravo nor WhiskeyBravo. Instead, IndiaGolf-Three only drops the RomeoMike and DeltaAlfa payloads as described previously. IndiaGolf-Three forgoes the generation and execution of a suicide script. Instead, once the installation task completes, IndiaGolf-Three merely returns control to the caller of the activation function.

9. [Installer] IndiaHotel

Observed as part of the SierraJuliett-MikeOne pushed commands, IndiaHotel is an installer that primarily installs IndiaIndia binaries and, in one known sample, also installs RomeoWhiskey. IndiaHotel uses a file storage format similar to that of IndiaDelta (see Section 5), which involves appending the file(s) to drop at the end of the IndiaHotel binary.

Upon activation, IndiaHotel, which is a standalone executable, begins by locating the end of its own binary and reading the last four bytes as a DWORD value. The value indicates the number of files that IndiaHotel contains. For each file, the extraction process consists of the following operations:

1. Read the previous four bytes as a DWORD to determine the size of the string containing the drop file's filename
2. Read the filename into memory
3. Read the four bytes immediately before the filename as a DWORD to determine the size of the drop file
4. Read the four bytes immediately before the file size field as a DWORD to determine the location of the start of the drop file. The DWORD expresses the number of bytes from the beginning of the file.
5. Read the drop file into memory and decompress the file with Zlib.

The IndiaHotel sample that is responsible for dropping RomeoWhiskey also generates a new service to support the dropped malware. This behavior consists of generating a new service name, service display name, and service description based on artifacts from existing services on the victim's computer. IndiaHotel then generates a set of registry entries in order to install the RomeoWhiskey as a svchost-based service on the victim's computer.

Activation of IndiaHotel's payload varies depending on the particular malware family that the installer drops. For the observed sample installing RomeoWhiskey, the malware activates the newly installed service via the Windows Services API. For variants installing IndiaIndia, the DLL of the malware is loaded into memory and executed via a call to **`LoadLibrary`**.

After dropping and activating the payload, IndiaHotel generates and executes a suicide script named either **`tsc.bat`** (for the RomeoWhiskey installer) or **`dvi.bat`** (for the IndiaIndia installer).

10. [Installer] IndiaIndia

Observed as the payload of IndiaHotel, IndiaIndia is an installer that uses the same method of encapsulating payload files as IndiaHotel, but it also boasts the ability to report to a C2 information about the infected computer. Structured as a DLL, IndiaIndia begins executing its core functionality as soon as the loading binary (e.g. IndiaHotel) loads the DLL and activates the `DllMain` function.

IndiaIndia performs the following set of tasks before terminating:

1. Dynamically load API functions
2. Generate a name for the configuration file
3. Generate information about the victim's system and report the information to the C2 server
4. Drop the configuration file (using the name specified in Task #2)
5. Install a service for the malicious binary being dropped
6. Drop the malicious binary (IndiaKilo)
7. Tie the newly installed service to svchost as a subservice
8. Activate the new service
9. Generate and execute a suicide script to clean up the IndiaIndia artifacts

Novetta observed IndiaHotel dropping a file with the name `dvpi.dna` alongside IndiaIndia. The file is the configuration, in plaintext, for both IndiaIndia and IndiaKilo. The data within the configuration is eventually saved to the victim's machine (Task 4) in an encrypted form (using a currently unidentified encryption scheme) and saved to a filename that is specific to the victim's machine. Task 2 generates a 6-character filename with a 3-character extension. The filename and its extension consist solely of lowercase letters that are randomly selected using the `rand` function. Prior to the generation of the filename, however, IndiaIndia calls `srand` to seed the pseudorandom number generator (PRNG) with the value of the primary hard drive's serial number. The effect of calling of `srand` with a repeatable seed value is that the output of `rand` becomes predictable and repeatable itself. IndiaKilo uses the same method for generating its configuration file's filename, thereby locking an infection resulting from IndiaIndia to a specific machine or, more precisely, a specific hard drive.

Using one of the 10 C2 server addresses found within the configuration file, IndiaIndia reports its activation to the C2 server. IndiaIndia also reports details regarding the victim's system along with a full list of the processes that are running on a victim's system, the filename of the executables responsible for each running process, and the title of the window for each running process. IndiaIndia gathers the following specific pieces of information about a victim's system:

- Computer name
- Infected user's username
- Windows OS version information (in a `OSVERSIONINFOEXA` structure)
- CPU information
- Locale information
- First network interface card's (NIC's) configuration including IP address, gateways, subnet mask, DHCP information and if WINS is available.

- Serial number and file system type of the primary hard drive
- Victim's local time
- If any of the following applications are present on the victim's machine based on registry keys within **HKCU** and **HKLM:** SecureCRT, Terminal Services, RealVNC, TightVNC, UltraVNC, Radmin, mRemote, TeamViewer, FileZilla, pcAnyware, Remote Desktop.

The list of processes (along with their related information) and the information that IndiaIndia gathers about a victim's system are saved together in a single file located in the **%TEMP%** directory before being compressed with Zlib, encrypted, and uploaded to one of the 10 C2 servers.

Communication between IndiaIndia and one of its configured C2 server begins by IndiaIndia randomly selecting from the list of 10 possible C2 servers. If a connection to the first selected C2 server fails, IndiaIndia waits 2 minutes before attempting to try to connect to another randomly selected C2 server. After a successful connection to a C2 server, IndiaIndia performs a handshake to establish a communication channel between itself and the C2 server. The handshake consists of generating a random buffer, calculating the CRC32 value of the buffer, further encrypting the buffer using a series of XOR operations, and then transmitting the size of the buffer, the buffer itself (prior to encryption), and the CRC32 value of the unencrypted buffer. If the C2 server responses with the CRC32 value of the encrypted buffer, the handshake is successful. The unencrypted buffer's contents become the encryption key for all future communication between the C2 server and IndiaIndia.

After the transmission of the victim's system information file to the C2 server, the C2 server responds with a 64-bit value representing the unique identifier for the victim's machine. The uploaded file is deleted using a secure deletion function that simply overwrites the contents with random data prior to calling the **DeleteFile** API. The unique identifier is converted to a string by means of the **ui64toa** function and used as the name for a named mutex. The named mutex is used by IndiaIndia, and later IndiaKilo, to determine exclusivity on the victim's machine.

Task #5 establishes a service for the binary that IndiaIndia ultimately drops by generating a service name, display name, and description by combing through the list of existing services and randomly picking components of the service description. With the service information generated, Task 5 calls **CreateService** to generate a new service with the path to the service's binary set to **%SystemRoot%\system32\svchost.exe -k** <name of service>.

As mentioned previously, IndiaIndia uses the exact same method and format for embedding binaries within itself. Novetta only observed IndiaIndia dropped IndiaKilo binaries, but the framework is generic enough to allow IndiaIndia to drop any binary that operates as a svchost-based service. Task 6 is responsible for dropping the embedded payload binary to the victim's machine.

While IndiaIndia has already installed a new service for the dropped binary (Task 5), it is still necessary to establish registry keys to support the service and tie it together with svchost.exe. Task 7 performs this operation by generating a new service entry under **HKLM\SYSTEM\CurrentControlSet\Services\**<name of service>. The registry entry contains the path to the dropped binary (from Task 6) and little more else.

With the payload (IndiaKilo) dropped and the service configured, the remaining task for IndiaIndia prior to clean up is the activation of the payload binary on the victim's computer. IndiaIndia calls **StartService** to activate the newly installed service and thereby activate the next step of the infection.

The final act of IndiaIndia is the generation and execution of a suicide script named **dvi.bat**. IndiaIndia generates the suicide script using a single **wsprintfA** call with the format string seen in Figure 10-1. The suicide script and its method of generation is identical to that found employed by IndiaHotel.

```
@echo off\r\n:L1\r\ndel /a \"%s\"\r\nif exist \"%s\" goto L1\r\ndel /a \"%s\"\r\n
```

Figure 10-1: wsprintfA Format String used to Generate IndiaIndia's Suicide Script

11. [Installer] IndiaJuliett

IndiaJuliett is a simplistic installer observed dropping and installing SierraJuliett-MarkOne, SierraJuliett-MarkOne along with SierraBravo, and SierraJuliett-MarkTwo. IndiaJuliett's structure is reminiscent of the structure of IndiaWhiskey-Three (see Section 13.3), in that the **`WinMain`** contains only a single function call to **`DialogBoxParamA`**. The core of IndiaJuliett's functionality exists within the dialogue callback function that **`DialogBoxParamA`** invokes.

The malware installed by IndiaJuliett resides in the resource section of the IndiaJuliett binary. Depending on the particular sample, the embedded malware may be encrypted using RC4. In order to properly decrypt the embedded files, IndiaJuliett decrypts the files in blocks of 5120 or 10240 bytes (depending on the sample). The S-Box of the RC4 decryption system is reset using the password **`cdefghijkl`** before decrypting each block.

Prior to installing a service for the SierraJuliett-based malware or the SierraBravo malware, IndiaJuliett will attempt to stop and delete the existing services related to the malware families (**`SCardPrv`** or **`wuaserv`** for the SierraJuliett malware and **`Wmmvsvc`** for the SierraBravo malware). While other India families more commonly attempt to directly modify the victim's registry to install a new service, IndiaJuliett utilizes the Windows Services API function **`CreateServiceA`** to create a new service for its payload malware before calling **`StartService`** to activate the malware.

The configuration data, with special attention on the initial peer seed list data, of the SierraJuliett malware families is hardcoded into the IndiaJuliett binary. This requires the developer(s) of IndiaJuliett to recompile the malware each time a new seed list is required.

Some variants of IndiaJuliett attempts to disable the victim's firewall prior to the installation of the malware payload(s). IndiaJuliett does this by setting the registry key **`HKEY_ LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\EnableFirewall`** to **`0`**.

After the installation process concludes, IndiaJuliett will generate and execute a suicide script saved as **`d.bat`**.

12. [Installer] IndiaKilo

Dropped by IndiaIndia, IndiaKilo is a repeatable installer that requires a C2 server to download new binaries at regular intervals. The categorization of IndiaKilo as an installer (an India) and not a RAT (a Romeo) is somewhat problematic. IndiaKilo could conceivably be considered an extremely lightweight RAT due solely to the fact that the C2 determines whether or not IndiaKilo is to download and execute a file - this differs from the usual case of a downloader/installer, where the determination is an inherent certainty. However, given that the only real operation that IndiaKilo performs is the downloading and executing of a specified file from the C2 server, the classification of IndiaKilo as an India is the best fit.

IndiaKilo operates as service DLL, and, after establishing the necessary environment to behave as a legitimate service, spawns a new thread for the execution of its core functionality. After initializing the Windows Winsock subsystem, IndiaKilo enters an infinite loop where the following tasks play out over and over again:

1. Determine the machine-specific filename for the configuration file and load the configuration into memory
2. Verify if only one instance of IndiaKilo is active
3. Verify that a preconfigured number of minutes has elapsed since the last communication with a C2 server occurred
4. Connect to one of the C2 servers found within the configuration
5. Generate information about the currently active processes on the victim's machine and save the information to a temporary file
6. Encrypt the process information file and upload the encrypted file to the C2 server
7. Delete the process information file
8. Receive a status command from the C2 server
9. If the status indicates a new set of files are available, download and execute the files
10. Reply to the C2 server with status of download and execution operation
11. Update the last C2 server contact time and save the current configuration to disk
12. Loop to Task 1

IndiaKilo uses the same configuration file as IndiaIndia which also requires IndiaKilo to generate the same configuration name in Task 1. The configuration file's filename is used not only for the name of the configuration file but also for the basis of the name of the named mutex that determines exclusivity on the victim's machine (Task 2). If either the configuration fails to load or another named mutex with the configuration file's filename exists, IndiaKilo terminates its core functionality thread, leaving only the shell of the service running.

The configuration file (Table 12-1) contains a wait time that determines how long IndiaKilo waits between successful C2 server contacts. IndiaKilo sleeps in one second intervals until the time since the last C2 server contact and the current time exceeds the wait time.

OFFSET	SIZE	DESCRIPTION
0	8 bytes	Victim Identifier
8	4 bytes (DWORD)	unused by IndiaKilo (used by IndiaIndia)
12	4 bytes (DWORD)	unused by IndiaKilo (used by IndiaIndia)
16	4 bytes (DWORD)	Wait time between C2 server contacts (in minutes)
20	4 bytes (DWORD)	Last C2 server contact time (in `time()` form)
24	80 bytes	Array of 10 C2 server entries with each entry having the following format:

Offset	Size	Description
0	4 bytes (DWORD)	IP address
4	2 bytes (WORD)	Port
6	2 bytes	not used

Table 12-1: Configuration File Structure of IndiaIndia and IndiaKilo

IndiaKilo uses the same C2 selection and connection procedure and protocol as IndiaIndia for Task #4. The code between IndiaKilo and IndiaIndia for the C2 server connection task is identical. The generation of process information, however, differs slightly between IndiaKilo and IndiaIndia. A notable difference is the inclusion of debugger and VM detection: IndiaKilo calls **`IsDebuggerPresent`** to determine if a debugger is attached to its process and attempts to open a handle to the events **`VMwareUserManagerEvent`** and **`VBoxHookNotifyEvent`** to determine if the malware is running in a VM. The detection of a VM environment or a debugger does not directly alter the flow of execution inside IndiaKilo, but the information is stored within the process list file the malware generates (Task 5) and then encrypts before sending to the C2 server (Task 6).

After deleting the process information file (Task 7), IndiaKilo receives a 4-byte status response from the C2 server indicating if additional data is to follow (Task #8). If the status value is set to 4, then the C2 server transmits two files, which IndiaKilo saves in the **`%TEMP%`** directory with filenames beginning with **`TS`**. After downloading and decrypting the two files to the victim's hard drive, IndiaKilo activates the first binary with the following command line (Task 9):

```
<first file received> <second file received> <unique identifier for victim's infection>
```

After executing the command, IndiaKilo responds to the C2 server with a 4-byte value (DWORD) of 4 for a successful launch of the command or 3 for a failure (Task #10). In the event that the files failed to execute, the two downloaded files are securely deleted by IndiaGolf by overwriting their contents with random bytes before deleting them from the victim's hard drive.

The connection between the C2 server and IndiaKilo terminates after the status update is transmitted and the last C2 server contact time is updated in the configuration in memory. The configuration in memory is then encrypted and saved to disk (Task #11).

13. [Installer] IndiaWhiskey

The principal installer for the RomeoWhiskey malware, IndiaWhiskey is a simple installer that, over the course of its lifespan, has used three different scaffolding codes to execute the same core installation process, resulting in three distinguishable variants (IndiaWhiskey-One, IndiaWhiskey-Two, and IndiaWhiskey-Three). It is unclear the developer(s)'s motivation for the scaffolding code changes for IndiaWhiskey, but regardless the core set of tasks remains the same:

1. Dynamically load API functions
2. Establish service information (filename, name, and display name)
3. Drop the payload file(s) to the victim's hard drive
4. Install the Windows service that loads the RomeoWhiskey malware
5. Activate the Windows service
6. Generate and execute a suicide script
7. Terminate

IndiaWhiskey uses the Lazarus Group's standard Space-Dot encoding scheme to obfuscate the names of the API functions that it dynamically loads. In the same subroutine that loads the `kernel32.dll` API functions, IndiaWhiskey establishes the name, display name, DLL filename, and configuration filename. The generation of service and file names do not use Space-Dot encoding but rather are constructed using stack strings (the method by which a string is generated one character at a time in a function's stack) in order to prevent the strings from being visible. The default values for the generated names is provided in Table 13-1.

ITEM	DEFAULT VALUE
Service Name	Windows Security
Service Display Name	Windows Security
Service DLL's Filename	winsec.dll, rdmgr.dll or winauc.dll
Configuration File's Filename	dayipmr.tbl

Table 13-1: IndiaWhiskey's Default Service and File Names

IndiaWhiskey stores its payload files within the resource section of its binary. Stored in either the resource section `RT_BITMAP\1033` (IndiaWhiskey-One and IndiaWhiskey-Three) or `IMG\159` (IndiaWhiskey-Two), the payload data blob is encrypted using the DNSCALC-style encoding. The payload data blob is a collection of one or more files, each with their own header providing the desired filename for the file. Before parsing the payload data blob, IndiaWhiskey first verifies that the decryption of the data blob was successful by determining if the decrypted data blob's first four bytes are `BMZA` (IndiaWhiskey-One, IndiaWhiskey-Three) or the first two bytes are `BM` (IndiaWhiskey-Two). If the magic value does not match the expected value, the installer will skip to Task 6.

There is one notable deviation from the above described payload file storage method. A single version of IndiaWhiskey-One embeds its payload data blob within its `.data` segment, not within a resource. The `BMZA` magic bytes are not present in this variant, and instead `XADD` is used to determine if the DNSCALC-style encoding was successfully removed.

With the integrity of the payload data blob verified, IndiaWhiskey begins the process of extracting the various payload files. Each file within the payload data blob begins with a header that defines the length of the filename for the file as a 16-bit value, the filename, and a 32-bit value representing the size of the file itself. Immediately following the file's information header is the file image itself. Figure 13-1 provides a visual illustration of how IndiaWhiskey stores files within its payload data blob. The payload data blob does not include any information that determines the number of files that it includes; instead IndiaWhiskey continues to extract files until it reaches the end of the payload data blob. This approach could allow a bad resource to result in erroneous output.

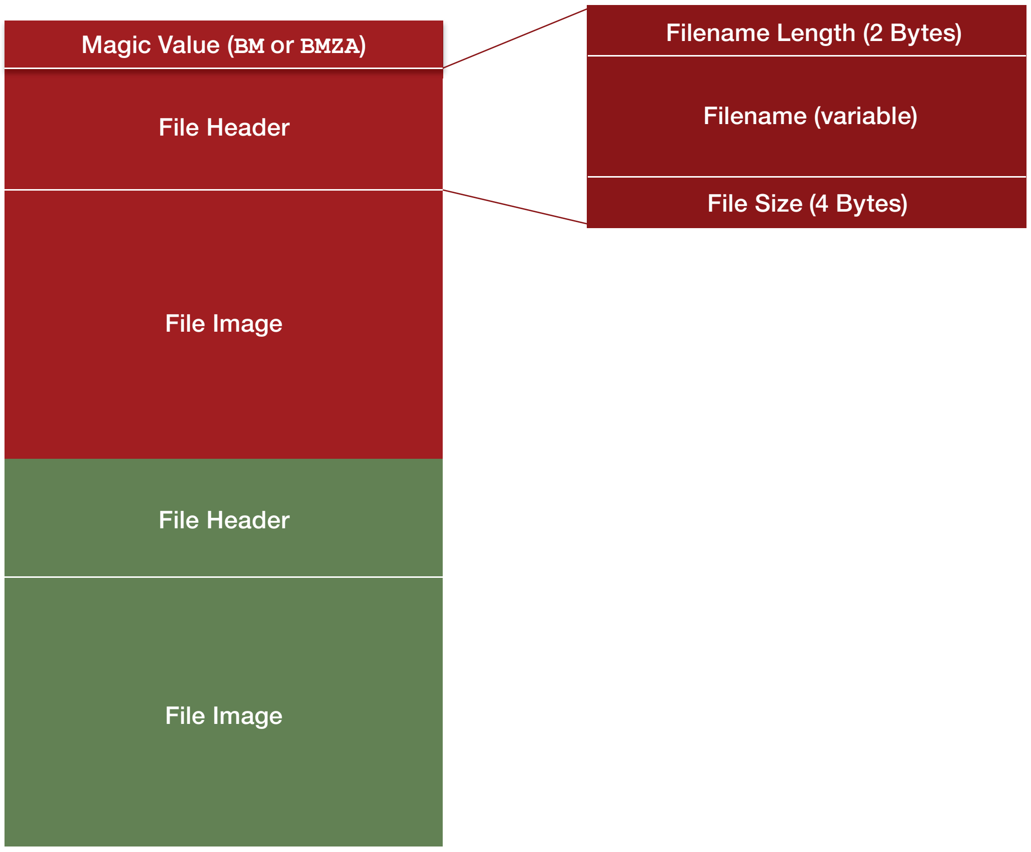


Figure 13-1: IndiaWhiskey's Payload File Storage Format

IndiaWhiskey stores each file in the same directory as `kernel32.dll`– the `%SYSDIR%` directory. For each file that IndiaWhiskey extracts to disk, the installer first determines if a file with the same name exists, and, if so, it deletes the file to make way for the extracted file. After writing the extracted file to disk, IndiaWhiskey alters the timestamp of the extracted file to match the timestamp of `kernel32.dll` in order to avoid drawing attention to the new addition.

With the RomeoWhiskey file dropped to disk, IndiaWhiskey must install and activate the service that ensures the malware's persistence, which is a relatively straightforward task:

1. Call `CreateServiceA` to add the new service to the service's database
2. Call `ChangeServiceConfig2A` to set the service's description
3. Create a registry key at `HKLM\SYSTEM\CurrentControlSet\services\`<name of new service>
4. Set the `Parameter\ServiceDll` value appropriately to point to the dropped RomeoWhiskey binary
5. Create a new service group under `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Svchost` for the new service
6. Call `StartServiceA` to activate the service

The description of the new service is variant-dependent, and, interestingly, the observed samples each have a misspelling as shown in Table 13-2 in bold.

VARIANT	SERVICE DESCRIPTION VALUE
IndiaWhiskey-One	Allows reporting for windows services and **applictions** running in standard environments. If this service is disabled, any services that explicitly depend on it will fail to start.
IndiaWhiskey-Two	Track system status such as Windows logon, network, and power events. **Notifiy** COM+ Event System subscribers of these events.
IndiaWhiskey-Three	Allows reporting for windows services and **applictions** running in standard environments. If this service is disabled, any services that explicitly depend on it will fail to start.

Table 13-2: IndiaWhiskey Variants' Service Description Values with Misspelled Highlighted in Bold

IndiaWhiskey concludes its life by generating and then executing a suicide script, saved as **`msvcrt.bat`**. An interesting artifact is the **`msvcrt.bat`** string: IndiaWhiskey-One and IndiaWhiskey-Three variants contain the string within the binary's **`.data`** section, while IndiaWhiskey-Two constructs the string using stack strings. This is yet another example of the subtle differences between variants. Although IndiaWhiskey-Two appears to exist between IndiaWhiskey-One and IndiaWhiskey-Three, chronologically speaking, these changes may indicate that at least two different developers generated the IndiaWhiskey variants. As mentioned previously, while it is the scaffolding code that truly differentiates the IndiaWhiskey variants, these subtle code and text changes may point to the existence of more than one active user of the IndiaWhiskey code base.

13.1 IndiaWhiskey-One

Of the IndiaWhiskey variants, IndiaWhiskey-One is by far the most simplistic when tracing the execution of the installer. Given that it contains no unnecessary code, and in fact contains no scaffolding code at all, it is arguable that IndiaWhiskey-One is the baseline for the IndiaWhiskey family. The `WinMain` function for IndiaWhiskey-One, Figure 13-2, is reused in all IndiaWhiskey variants nearly as-is.

```
int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nShowCmd)
{
  LoadKernel32APIAndGenerateNames();
  LoadAdvapi32();
  LoadShlwapiAPIs();
  if ( !DropFile() )
  {
    InstallService();
    ActivateService();
  }
  Suicide();
  return 0;
}
```

Figure 13-2: IndiaWhiskey-One's WinMain Function

13.2 IndiaWhiskey-Two

The outlier of the IndiaWhiskey variants, IndiaWhiskey-Two expands on the basic structure of IndiaWhiskey-One by adding an unnecessary chunk of code that displays two message boxes in case **LoadShlwapiAPIs** generates a Windows error. It is unclear the intention of this additional code, as the first message box informs the user that no error has occurred, followed immediately by a message indicating that some error has occurred. Interestingly, the "Error Code" value is set to 0, a condition that defines the variable to always be false. The additional code is superfluous at best.

```
int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nShowCmd)
{
  DWORD lastError;
  char String[260];
  int v7;
  LoadKernel32APIAndGenerateNames();
  LoadAdvapiAPIs();
  LoadShlwapiAPIs();
  lastError = GetLastError();
  memset(String, 0, 0x104);
  if ( lastError && (v7 = lstrlenA(String)) != 0 )
  {
    lstrcpyA(String, "No error occured.\nClick 'OK' to continue.");
    MessageBoxA(0, String, "Error", 0);
    sprintf(String, "Unknown error occured.\nError Code : %d", v7);
    MessageBoxA(0, String, "Error", 0);
  }
  else
  {
    if ( !DropMalware() )
    {
      InstallService();
      ActivateService();
    }
    Suicide();
  }
  return 0;
}
```

Figure 13-3: IndiaWhiskey-Two's WinMain Function

13.3 IndiaWhiskey-Three

IndiaWhiskey-Three is by far the biggest shift from the baseline established by IndiaWhiskey-One. IndiaWhiskey-Three adds a bit of complexity by moving a large portion of the core functionality into a Windows dialog box function. The **`WinMain`**, Figure 13-4, still calls the functions to perform the dynamic API loading and the service and file name generation, but it then calls **`DialogBoxParamA`** to generate a dialogue box.

```
int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nShowCmd)
{
  LoadKernel32APIAndGenerateNames();
  LoadAdvapi32();
  LoadShlwapiAPIs();
  DialogBoxParamA(hInstance, "TEST", 0, DialogFunc, 0);
  return 0;
}
```

Figure 13-4: IndiaWhiskey-Three's WinMain Function

The DialogFunc function, Figure 13-5, is called when Windows begins to generate the dialogue box. It is at this point that the rest of the IndiaWhiskey functionality continues.

```
BOOL DialogFunc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  if ( msg == WM_PAINT )
  {
    if ( !fActive )
    {
      ShowWindow(hWnd, 0);
      fActive ^= 1u;
    }
  }
  else if ( msg == WM_INITDIALOG )
  {
    ShowWindow(hWnd, 0);
    if ( !DropFile() )
    {
      InstallService();
      ActivateService();
    }
    Suicide();
    EndDialog(hWnd, 0);
    PostQuitMessage(0);
  }
  return 0;
}
```

Figure 13-5: IndiaWhiskey-Three's Dialogue Callback Function

14. [Loader] LimaAlfa

There are times when a piece of malware requires a support executable in order to execute. LimaAlfa is such a piece of malware. LimaAlfa is responsible for the loading, removal of, and activation of the WhiskeyCharlie wiper malware and requires three command line arguments in order to function:

```
<file to delete> <configuration file key> <delay before commands, in seconds>
```

If there are not exactly three command line arguments, LimaAlfa quietly terminates.

As identified in the section describing IndiaDelta (see Section 5), LimaAlfa is dropped and activated by IndiaDelta, and the first command line argument passed to LimaAlfa is the filename of the IndiaDelta binary. The first task that LimaAlfa perform is the secure erasure of the IndiaDelta function by means of a secure deletion function. LimaAlfa then enters an infinite loop that only breaks with the removal of the IndiaDelta binary. For each iteration through the loop, the secure file deletion function is called against the IndiaDelta binary, and the check condition is handled by calling **`GetFileAttributesA`** to determine if the IndiaDelta binary exists. In the same way that IndiaDelta would attempt to delete the previous stage of its loading process, LimaAlfa is continuing this trend by using a somewhat overly engineered approach to avoid using a suicide script.

For reasons that are not immediately clear, the next task LimaAlfa performs is to call **`LoadLibrary`** for **`user32.dll`**, **`advapi32.dll`**, **`shell32.dll`**, and **`shlwapi.dll`**. However, LimaAlfa does not attempt to load any API functions from these DLLs, making the decision to load them into the process's memory space somewhat baffling.

LimaAlfa expects the presence of a file named **`mcu.dll`**, as the loader attempts to open the file by means of a **`fopen`** call. The **`mcu.dll`** binary, a WhiskeyCharlie wiper binary, is one of the files that IndiaDelta drops. Instead of loading **`mcu.dll`** directly into the process's memory space by calling **`LoadLibrary`**, LimaAlfa's developers use the POSIX file APIs to open the file (**`fopen`**), determine the size of the file (**`ftell`**), and read the file into an allocated buffer (**`fread`**). After the contents of **`mcu.dll`** are loaded, LimaAlfa attempts to perform a secure delete of the file. This explains why the developer(s) of LimaAlfa went to such trouble to load the file into memory instead of calling **`LoadLibrary:`** an active executable, even a DLL, cannot be deleted until it terminates. By loading the file into memory, LimaAlfa can manually load the DLL's image and delete the source material without waiting.

The next task that LimaAlfa performs is the act of manually loading the **`mcu.dll`** image into its own process memory space. This task requires LimaAlfa to allocate the correct amount of memory space, manually parse the PE header of the image to properly fill the memory space, parse the import table and load the appropriate API functions necessary, fix offsets, and call the **`DllMain`** of the loaded image.

With the **`mcu.dll`** image loaded into LimaAlfa's process space, the next task is to locate **`mcu.dll`**'s exported function **`Register`**. **`Register`** is the entry point of the WhiskeyCharlie malware and is explained in greater detail in the section dedicated to that particular malware family in Novetta's technical report on the Lazarus Group's destructive malware.[9] LimaAlfa calls **`Register`** and passes the function three parameters: the name of the command file (hardcoded to **`mcu.inf`**), the string to decrypt the command file (as specified by the second command line argument) and the string

9 http://www.operationblockbuster.com/wp-content/uploads/2016/02/Operation-Blockbuster-Destructive-Malware-Report.pdf

presenting the number of seconds to wait between each command (as specified by the third command line argument). At this point, LimaAlfa is acting more like an incubator as it waits for **`Register`** to return after completing its tasks.

Once **`Register`** returns, LimaAlfa generates and executes a suicide script. The generation of the suicide script is extremely close in method and construction as IndiaAlfa's suicide script generation function. The name of the batch file that contains the suicide script is generated as **`zz`** followed by the output of **`rand`** and is located within the **`%TEMP%`** directory. Construction of the script itself is performed as seen in the code snippet in Figure 14-1.

```
fp = CreateFileA(&szSuicideScriptFilename, 0x40000000u, 0, 0, 2u, 0x80u, 0);
if ( fp != (HANDLE)-1 )
{
    lstrcpyA(buffer, ":x\r\ndel \"");
    lstrcatA(buffer, &szLimaAlfaFilename);
    lstrcatA(buffer, "\"\r\nif exist \"");
    lstrcatA(buffer, &szLimaAlfaFilename);
    lstrcatA(buffer, "\" goto x\r\ndel \"");
    lstrcatA(buffer, &szSuicideScriptFilename);
    lstrcatA(buffer, "\"");
    WriteFile(fp, buffer, lstrlenA(buffer), &NumberOfBytesWritten, 0);
    CloseHandle(fp);
    ShellExecuteA(0, "open", &szSuicideScriptFilename, 0, 0, 0);
}
```

Figure 14-1: LimaAlfa's Suicide Script Generator Code Snippet

The execution of the suicide script occurs via a **`ShellExecute`** function call, as with IndiaAlfa. It is more than likely that either the same author was responsible for the suicide script function in both IndiaAlfa and LimaAlfa or that the code was shared.

15. [Loader] LimaBravo

Novetta observed IndiaEcho dropping LimaBravo as part of a two-stage loading system that ultimately loads RomeoGolf. LimaBravo is a service DLL that manually loads a Windows executable image (a DLL named `perfc010.dat`) into memory, then executes the entry point function of the loaded image. Performing the same function as calling `LoadLibrary`, LimaBravo manually handles the tasks of memory allocation, section loading, import loading, and offset fix-ups before locating the entry point of the image and transferring control to that function.

LimaBravo generates a separate thread for the loading and eventual execution of the target image. This ensures that the code necessary to support the service framework does not hang and at the same time provides a semi-autonomous execution environment for the new image. Prior to loading the target time, LimaBravo ensures that a mutex with the name `{17121AB3-079E-4622-9315-44C0364C6123}` does not exist on the victim's system, ensuring that only one instance of LimaBravo is active at a time on the system.

The code responsible for the image loading is from the open-source code MemoryModule by Joachim Bauch.[10]

10 Joachim Bauch. "MemoryModule: Library to load a DLL from memory" https://github.com/fancycode/MemoryModule Accessed 5 February 2016.

16. [Loader] LimaCharlie

LimaCharlie is a DLL-based loader and is both the loader and dropper for the RomeoHotel malware family. LimaCharlie's operation consists of four tasks:

1. Dynamically load API functions
2. Verify RomeoHotel's configuration and drop RomeoHotel DLL to disk
3. Load and activate the RomeoHotel DLL
4. Remove the RomeoHotel DLL

LimaCharlie appears to be the second stage of a longer malware chain, as the second task it performs is the verification of the RomeoHotel's configuration and temporary installation of the RomeoHotel DLL on the victim's hard drive. This, along with the fact that LimaCharlie operates as a DLL, indicates that LimaCharlie requires an installer prior to its activation. It is currently not known which malware family is responsible for dropping and installing LimaCharlie.

The verification of the RomeoHotel configuration data's registry entry is somewhat spotty in its approach. The verification consists of decrypting the string containing the registry branch, using Caracachs, and then attempting to open the registry key at the location **`HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security`** by calling **`RegOpenKey`**. If the **`RegOpenKey`** function returns an error, LimaCharlie terminates. However, the configuration data is stored under a subkey for RomeoHotel (**`HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security\zc62a465-efff-87cc-47cdcdefa`**) that LimaCharlie attempts to read into memory via **`RegQueryValueEx`**. LimaCharlie makes no attempt to determine if the **`RegQueryValueEx`** call fails, thereby providing no real verification that the configuration data truly exists. If the configuration data is available, LimaCharlie uses the data as the RomeoHotel configuration and produces a series of calls to a **`nullsub`** function (a function that does nothing other than to return to the caller) with various fields from the configuration (Figure 16-1).

```
RegQueryValueExA(phkResult, lpValueName, 0, 0, &Data, &NumberOfBytesRead);
RegCloseKey(phkResult);
GetSystemDirectoryW(FileName, 0x104);
wcscat(FileName, L"\\");
wcscat(FileName, Data.wszServiceDLL);
nullsub_1(&Data);
nullsub_1(&Data.arrwszC2s[9]);
nullsub_1(Data.wszServiceName);
nullsub_1(FileName);
memset(&buffer[0], 0, 100);
swprintf(
  buffer,
  L"\r\nDefualt Sleep = %d, read Size = %d, %d \r\n",
  Data.dwSleepTimeBetweenReconnects,
  NumberOfBytesRead,
  0xAE0);    // 0xAE0 is the expected size of the RomeoHotel configuration
nullsub_1(buffer);
```

Figure 16-1: LimaCharlie's Use of the RomeoHotel Configuration Data during the Verification Task

LimaCharlie's dropping of the RomeoHotel binary is rather straightforward and similar to other installers' within the Lazarus Group's collection of malware. LimaCharlie first determines its own file size and then reads a DWORD from the last 4 bytes of itself. This value indicates the offset to the start of the RomeoHotel image within the LimaCharlie binary. The RomeoHotel image begins with a DWORD specifying the size of the RomeoHotel binary contained within the image data. LimaCharlie then allocates memory on the heap of the size specified before reading the RomeoHotel binary into memory. The RomeoHotel binary is encrypted using Caracachs, which LimaCharlie removes before saving the binary to disk as **`%TEMP%\DQ`**{random number}.

The loading and activation of the dropped RomeoHotel binary is based on a CodeProject project by the user pasztorpisti.[11] Dating back to 2000 but updated several times between 2012 and 2014, the LoadDll project contains a framework for loading a DLL into memory without using the Windows API function **`LoadLibrary`**. LimaCharlie uses the core code of the project to, as the description suggests, load the RomeoHotel binary into memory and call the DLL's **`DllMain`** function to activate the RomeoHotel malware.

After loading the RomeoHotel binary into memory, the RomeoHotel binary is deleted from the victim's hard drive via the standard Windows API function **`DeleteFile`**. It is this action that justifies the use of the LoadDll code to load RomeoHotel. Had the developer(s) of RomeoHotel used **`LoadLibary`** to load the DLL into memory, the call to **`DeleteFile`** would have failed, as the Windows operating system would have a lock on the DLL. But by manually loading the DLL into memory and then activating the DLL, Windows does not have such a lock, allowing the RomeoHotel DLL to be deleted and leaving little trace of the actual, malicious binary running on the victim's system.

11 pasztorpisti. CodeProject. "Loading Win32/64 DLLs "manually" without LoadLibrary()" http://www.codeproject.com/Tips/430684/Loading-Win-DLLs-manually-without-LoadLibrary. 8 Mar 2014

17. [Loader] LimaDelta

A bit of an outlier in the Lazarus Group's collection of malware, LimaDelta blurs its categorical lines by operating as both a downloader and loader. Known LimaDelta samples were compiled between September 2010 and March 2011 and most likely only operated during that time period, making it problematic to find intelligence on the files they downloaded. Despite the temporal distance between LimaDelta's usage in the wild and the analysis of the malware family for this report, there is evidence to suggest that LimaDelta downloaded and then loaded IndiaGolf (see Section 8). This section works under the assumption that the downloaded binary is, indeed, IndiaGolf.

LimaDelta, when viewed by its primary function of downloading and loading a file into running memory, is an example of over-engineering a solution for a relatively simple problem. When viewed at a macro level, LimaDelta performs the following tasks:

1. Connect to a C2 server and authenticate
2. Send information about the victim's machine and exchange LimaDelta configuration between the C2 server and the malware
3. Receive a list of download servers and a data blob containing the data to send to the installation function
4. Connect to one of the download servers and authenticate
5. Send information about the victim's machine and receive 16 bytes from the download server
6. Receive file from the download server and save it to the victim's hard drive as a randomly generated named DLL
7. Load the DLL into memory and decrypt a portion of the data blob
8. Call the installation function with the decoded portion of the data blob
9. Report status of DLL load to the C2 server and receive response from the C2 server
10. Save the configuration to disk and delete additional files
11. Generate and execute suicide script.

There are three variants of LimaDelta that differ primarily in their verification method for the files they download. LimaDelta-One blindly assumes the file it downloads is non-corrupt. LimaDelta-Two verifies the integrity of the downloaded file via a CRC32 checksum. LimaDelta-Three uses MD5 to verify the integrity of the file it downloads. There are subtle differences between each of the various variants that will be discussed below, but the average similarity between the variants (using BinDiff as the similarity engine) is roughly 94.5%, indicating very little change across the life span of LimaDelta.

LimaDelta can support up to 10 C2 servers via hardcoded IP and port number structures. From this list of C2 servers, LimaDelta randomly selects an entry and attempts to connect to the C2 server. If a server does not respond within 15 seconds, the C2 server entry is removed from the list, another C2 server entry is selected, and the process repeats until either a successful connection occurs or the list empties.

Once connected to a C2 server, LimaDelta must authenticate with the C2 server. This authentication process (the last half of Task #2) is over-engineered when compared to other families within the Lazarus Group. Authentication between LimaDelta and the C2 server consists of constructing a variable-sized data blob that has a minimum of 70 bytes and an upper limit of 120 bytes, transmitting the data blob to the C2 server, receiving a 5-byte response from the C2 server, and using four of the received bytes as the communications key for any further communication. While the high-level events just described for the authentication process may not seem overly complex or cumbersome, it is the construction of the variable-sized data blob that makes the authentication task appear overly engineered. The data blob, detailed in Table 17-1, contains a large number of constant values and randomly generated data.

OFFSET	SIZE (BYTES)	DESCRIPTION
0	11	Constant values: `0x16`, `0x03`, `0x01`, `0x00`, `0x41`, `0x01`, `0x00`, `0x00`, `0x3D`, `0x03`, `0x01`
11	4	Current time (from `time` function) in big-endian order
15	4	Randomly generated bytes
19	4	Data from offset 15 XOR'd with the string `0x4C`, `0x86`, `0x25`, `0xB7`
23	1	Total size of the authentication data blob
24	1	Constant value: `0xC0`
25	18	Randomly generated bytes
43	27	Constant values: `0x00`, `0x00`, `0x16`, `0x00`, `0x04`, `0x00`, `0x05`, `0x00`, `0x0A`, `0x00`, `0x09`, `0x00`, `0x64`, `0x00`, `0x62`, `0x00`, `0x03`, `0x00`, `0x06`, `0x00`, `0x13`, `0x00`, `0x12`, `0x00`, `0x63`, `0x01`, `0x00`, `0x00`
70	Variable (up to 50 bytes)	If the C2 server's listening port is not 443, this field contains up to 50 bytes of randomly generated bytes. The number of bytes is randomly selected between 0 and 50.

Table 17-1: LimaDelta's Authentication Request Data Structure

After generating and transmitting the authentication data blob, LimaDelta receives a 5-byte data burst containing a status code and an array of 4-bytes that constitutes the basis for the communication key going forward. Table 17-2 illustrates the structure of the response from the C2 server.

OFFSET	SIZE (BYTES)	DESCRIPTION
0	1	Status code: 0 indicates a failed authentication, 1 indicates a successful authentication
1	4	Communications key base array

Table 17-2: LimaDelta's Authentication Response Data Structure

The authentication scheme ultimately defines the encryption key that the LimaDelta binary and the C2 server use for further communications. The communication key base array (offset 1) sent from the server makes up one half of the communication key that LimaDelta uses for encrypting traffic between itself and the C2 server. The full key consists of taking the communications key base array and XOR'ing each byte against its respective byte within the 4 random bytes (offset 15) LimaDelta sent as part of the authentication data. Figure 17-1 illustrates the code responsible for generating the communication key.

```
if ( response.bStatus )
{
  memcpy(commKey, response.commKey, 4);
  j = 0;
  do
  {
     commKey[j] ^= random4ByteBuffer[j];
     ++j;
  }
  while ( j < 4 );
  dwDataKeyIndex = 0;
  result = 1;
}
```

Figure 17-1: LimaDelta Generating the Communications Encryption Key

With the connection between the LimaDelta and the C2 server established and the communication encryption initialized, LimaDelta sends another packet of information to the C2 server, this time containing information about the victim's computer (Table 17-3). At this point, all communication between LimaDelta and the C2 server uses the common network data transmission and receiving functions to support an encrypted communication channel. LimaDelta uses the code fragment in Figure 17-2 to encode and decode communication between LimaDelta and the C2 server.

SIZE IN BYTES (DATA TYPE)	DESCRIPTION
2 (WORD)	Size of data blob (minus this field)
Variable (NULL-terminated string)	Name of LimaDelta's executable (as returned by GetModuleFileNameA)
Variable (NULL-terminated string)	Full path to LimaDelta's executable
Variable (NULL-terminated string)	Computer's name
Variable (NULL-terminated string)	User's name
10	Randomly generated 10-byte buffer
8 (VARIANTTIME)	Time and date that LimaDelta's executable was written to disk
8 (VARIANTTIME)	Time and date that LimaDelta was executed
8 (VARIANTTIME)	Current time and date on victim's system
4 [DWORD]	Non-zero if SysInternal tools Process Monitor, Registry Monitor or File Monitor are active
4 [DWORD]	Current value from GetTickCount
Variable	Optional payload data blob

Table 17-3: LimaDelta's Victim's System Information *Data Blob Structure*

```
i = dwDataKeyIndex;
do
{
  *p++ ^= commKey[i];
  i = dwDataKeyIndex++ + 1;
  if ( dwDataKeyIndex == 4 )
  {
    i = 0;
    dwDataKeyIndex = 0;
  }
  --dwBytesRemaining;
}
while ( dwBytesRemaining );
```

Figure 17-2: LimaDelta's Communication Encryption/Decryption Scheme's Code

It is interesting that the developer(s) of LimaDelta paid attention to the SysInternal tools while there is no other indication within the LimaDelta code base that tools such as debuggers are of any concern. The method by which LimaDelta determines if the SysInternal tools are present and active on the victim's computer is the identification of window class names from the tools. For the tool Process Monitor, LimaDelta looks for the window class `PROCMON_WINDOW_CLASS` while for Registry Monitor and File Monitor—two tools that have been out of active circulation since September 19, 2009 due to their integration into Process Monitor—LimaDelta looks for the window class `18467-41`. Conceivably, the inclusion of an indicator that these tools were active at the time of the infection could change the behavior of the C2 server's responses, but without an active C2 server to test against, this is only a theory.

Task 2 consists of LimaDelta receiving the name for a file to store on the victim's computer, uploading the contents of the file to the C2 server, receiving a data blob, and saving the data to the same file on the victim's disk. It is not immediately clear the purpose of the file that LimaDelta and the C2 server exchange, but there is a limitation that the file cannot exceed 100 bytes. The content of the file is not used by LimaDelta, and without a filename it is not possible to speculate as to what malware family it may relate to.

The C2 server is not the ultimate end point from which LimaDelta receives the binary to load. Instead, the next task (Task 3) consists of LimaDelta receiving a DWORD containing the number of download server records that the C2 server is going to send, followed by the download server records themselves. LimaDelta accepts a maximum of 10 download server records. The records are 6 bytes in length, with the first four bytes being the IP address of the download server and the last two bytes being the port the download server listens to. Following the download of server records, the C2 server sends another 4-byte value identifying the number of bytes that the C2 server is going to transmit. The C2 server then transmits up to 1KB of data which becomes the basis for the configuration data that LimaDelta passes to the installation function in Task 8. The connection to the C2 server is terminated upon completion of the data blob transfer.

LimaDelta uses the same function to randomly select from the provided download server records and perform the same form of authentication as was done when the malware contacted the C2 server (Task 4). Following a successful authentication to a download server, LimaDelta sends another system information packet to the download server as it did with the C2 server (Task 5), with the exception that the optional payload section now contains the first 14 bytes of the data blob to be passed to the installation function.

After LimaDelta sends the system information data (along with the additional 14-byte payload), the download server will respond with a 16-byte array. The contents of the 16-byte array are disregarded by all variants except LimaDelta-One, which ultimately passes the array to the installation function. LimaDelta must generate a filename before the file can be loaded into memory. The name that LimaDelta generates is variant-specific, but it is always sent as a `.dll`.

In order to transfer the DLL (Task 6), LimaDelta receives a 4-byte value, a DWORD, indicating the size of the file from the download server. LimaDelta receives the file in 12288 byte transmissions from the download server. To confirm the file was received, LimaDelta sends at least 128 bytes but no more than 384 bytes of the downloaded chuck, starting four bytes into the chuck, back to the download server. The number of bytes sent is randomly selected for each confirmation. The confirmation is preceded by the first four bytes (DWORD) indicating the number of bytes that LimaDelta is sending to the download server as confirmation. With LimaDelta having saved the file to disk, the file is loaded into memory via a call to **`LoadLibraryA`**, and the installation function is obtained by calling **`GetProcAddress`** with the variant-appropriate export name.

With the exception of the LimaDelta-One variant, the data blob to be used as the installation function's parameter is decrypted using the simple algorithm in Figure 17-3. From the algorithm, it is possible to infer a few details of the data structure received in Task 3. For starters, the data contains 10 records, IP and port records, starting at offset 22. The value for the IP address is encoded with the static XOR key **`0x58713A6D`**, while the port number is not. Secondly, the encoded IP and port records appear to be 16 bytes in size, meaning LimaDelta expects exactly 10 such records to be present in the data blob; therefore, at a minimum, the data blob is 182 bytes once you factor in the initial 22 byte offset.

```
unsigned char *pParamDataDecoded = pRomeoMikeConfigData;
unsigned char *pParamDataEncoded = &pDataBlob[22];
i = 10;
do
{
  v13 = *( _ DWORD *)(pParamDataEncoded - 4);
  *( _ DWORD *)pParamDataDecoded = *(DWORD*) pParamDataDecoded ^ 0x58713A6D;
  *(( _ DWORD *)pParamDataDecoded + 1) = *( _ WORD *)(pParamDataEncoded);
  pParamDataDecoded += 8;
  pParamDataEncoded += 16;
  --i;
}
while ( i );
```

Figure 17-3: LimaDelta's Code for Transforming the Downloaded Data Blob into the Installation Function's Data Blob

With the installation function now available in memory and the data structure necessary for the installation function ready, LimaDelta calls the installation function and waits for the function to return. Once the installation function returns, LimaDelta sends the value of **`1`** (as a 4-byte DWORD) to the download server and reads a 4-byte response that it promptly ignores before shutting down the connection to the download server.

With the usefulness of the DLL now over, LimaDelta removes the binary from memory by calling **`FreeLibraryA`** before sleeping for 300ms. The DLL is removed from the victim's system by first overwriting the file with 4096-byte buffers containing the same 16-bit value. For each 4096-byte buffer used, a new 16-bit value is randomly generated. LimaDelta deletes the destroyed file.

LimaDelta saves the contents of the data blob obtained from the C2 server during Task 2. After saving the data blob to disk, the actions that occur before termination vary by variant. Variants LimaDelta-One and LimaDelta-Two delete entries from the victim's URL cache of Internet Explorer, while LimaDelta-Three deletes two **`.ini`** files. The final step performed by all variants of LimaDelta prior to termination is the generation and execution of a suicide script. Using **`sprintf`** to generate the script in a single pass and **`WriteFile`** to save the script to disk, LimaDelta calls **`CreateProcessA`** to execute the script as Figure 17-4 illustrates. The filename of the script is variant-dependent.

```
hFile = CreateFileA _ 0(szScriptName, 0x40000000u, 1u, 0, 2u, 0x80u, 0);
if ( hFile != (HANDLE)-1 )
{
  sprintf(
    szScript,
    "@echo off\r\n:R\r\ndel /a \"%s\"\r\nif exist \"%s\" goto R\r\ndel /a \"%s\"\r\n",
    szLimaDeltaFilename,
    szLimaDeltaFilename,
    szScriptName);
  WriteFile(hFile, szScript, strlen(szScript), &NumberOfBytesWritten, 0);
  CloseHandle(hFile);
  StartupInfo.cb = 0;
  memset(&StartupInfo, 0, 0x40u);
  StartupInfo.dwFlags = 1;
  StartupInfo.wShowWindow = 0;
  CreateProcessA(0, szScriptName, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
}
```

Figure 17-4: LimaDelta's Suicide Script Generator

17.1 LimaDelta-One

LimaDelta-One is the oldest observed variant within the LimaDelta family, dating back to September 2010. The most notable difference from other LimaDelta variants is the fact that, instead of calling a single installation function, LimaDelta-One accesses two functions: `DialingConnect` and `DialingClose`. LimaDelta-One calls `DialingClose` first and passes two arguments to the function, the first being the data blob from Task 3 starting at offset 14 and the second being the 16 bytes returned from the download server during Task 5. LimaDelta-One then calls `DialingConnect` with a pointer to an uninitialized 132-byte buffer. After the function returns, the first DWORD of the buffer is overwritten with return value from `DialingClose` and then transmitted to the download server during Task 9.

Prior to the generation and execution of the suicide script, LimaDelta-One calls `DeleteUrlCacheEntryA` to remove `hxxp://sub.sharebox.co.kr/SBUpdate.exe` from the cache files on the victim's machine. If any of the tasks between Tasks 1 and 9 fail, LimaDelta-One also deletes the file `ShareBoxC.dat` that exists in the same path as the LimaDelta-One binary. The combination of the cache file deletion and `dat` file deletion may point to the possibility that `sub.sharebox.co.kr` was compromised and that LimaDelta-One was being installed as `SBUpdate.exe`.

When LimaDelta-One generates the suicide script, the script is saved to the `%TEMP%` directory using a randomly generated name. The name for the suicide script consists of four randomly generated digits between 0 and 9 followed by `.bat`. It is interesting that the developer(s) would choice to generate each digit individually when it would be just as effective to use a `sprintf("%d.bat", rand())` statement to achieve the same result.

17.2 LimaDelta-Two

LimaDelta-Two introduces the idea of file integrity checks. After receiving the file from the download server, the last 4-bytes are read from the file as a DWORD. A call to **`SetFilePointer`** and **`SetEndOfFile`** is used to remove the last 4 bytes from the file. The DWORD represents the CRC32 checksum for the downloaded file. LimaDelta-Two calculates the CRC32 checksum for the downloaded binary and compares it to the expected checksum value. If the values match, LimaDelta-Two considers the file valid, otherwise LimaDelta-Two indicates a failed download.

LimaDelta-Two has been observed using two different names for the installation function: **`StartInstall`** and **`_gdimain`**. Despite their different names, both installation functions take the same parameters as described previously for Task 8.

Prior to the generation and execution of the suicide script, LimaDelta-Two calls **`DeleteUrlCacheEntryA`** to remove the following cache entries (and their associated files on disk):

- hxxp://sub.sharebox.co.kr/SBUpdate.exe
- hxxp://webfile.bobofile.co.kr/app/bobofile/setup/setup_bobofile.exe
- hxxp://webfile.filecity.co.kr/app/filecity/setup/setup_filecity.exe

As with LimaDelta-One, if any of the tasks between Task 1 and 9 fail, LimaDelta-Two deletes the file **`ShareBoxC.dat`** that exists in the same path as the LimaDelta-Two binary. Just as the combination of specific URL cache entry and specific file deletion of LimaDelta-One indicates that LimaDelta-One was hosted on a compromised site, LimaDelta-Two's URL cache deletion may indicate that three Korean websites were compromised and hosting LimaDelta-Two.

LimaDelta-Two uses the same suicide script generation function as LimaDelta-One. The reuse of the generator means that the same sloppy random name generator that produces a four character **`.bat`** file filename is used in LimaDelta-Two, and that the suicide script will exist within the **`%TEMP%`** directory.

17.3 LimaDelta-Three

LimaDelta-Three enhances the file integrity check by replacing the CRC32 checksum with a MD5 hash. Like LimaDelta-Two, after receiving the file from the download server, LimaDelta-Three reads the last 16-bytes from the file and calls to `SetFilePointer` and `SetEndOfFile` to remove the last 16 bytes from the file. The 16 bytes represent the MD5 hash of the downloaded file. LimaDelta-Three calculates the MD5 hash of the downloaded binary and compares it to the expected hash value. If the values match, LimaDelta-Three considers the file valid, otherwise the download is considered invalid.

LimaDelta-Three uses a different name for the installation function: `UICreate`.

LimaDelta-Three does not remove any URL cache files from the victim's system prior to calling the suicide script. The malware does, however, delete `_ver.ini` and `ver.ini` from the current working directory if any task between Task 1 and Task 9 fails. Neither the purpose of these files nor their origin are unknown.

The suicide script generator in LimaDelta-Three produces the same suicide script as the other variants, and in the same manner. But, unlike the previous LimaDelta variants described, LimaDelta-Three uses a static filename, saving the generated suicide script to `%TEMP%\vlt.bat`.

LimaDelta-Three has an interesting artifact within its strings. Using a dynamic API loader like all of the observed LimaDelta variants, LimaDelta-Three has what appears to be a gross editing error within its source code. The dynamic API loading function calls `GetProcAddress` for the specified API names after doing a bulk decryption of all API names. For the `WSACleanup` API function, the authors supply the following to the `GetProcAddress` function:

```
WSACleanupSTR_READFILE1 ReadFile
```

The error would appear to indicate that the names of API functions are declared using `#define` preprocessor statements and that somehow a serious copy and paste or find and replace operation error has occurred but went undetected at compile time. While the error has no operational impact on the malware, it does crack a door open to allow analysts to peak at the construction of the malware at the source code level.

18. [Uninstaller] UniformAlfa

Tools and objectives can change during the course of an operation, making the rapid removal of an existing attack tool necessary. UniformAlfa is an example of a tool designed for the purpose of removing other tools. When activated, it removes the service and file artifacts that are associated with RomeoBravo.

UniformAlfa begins by entering an infinite loop that calls the API function `ControlService` in order to stop the service named `WmiSecSvc`. The loop is only broken when the status of the `WmiSecSvc` returns `SERVICE_CONTROL_ STOP`, indicating that the service has successfully stopped. At this point the service can be safely removed from the Windows operating system by a call to the `DeleteService` function.

Next, the files `pmsconfig.msi`, `pmslog.msi`, `wmisecsvc.dll`, and `tmscompg.msi` are deleted with a call to `DeleteFile`. Unlike other families within the Lazarus Group's collection, the deletion of the four files does not include a more secure form of deletion.

With the RomeoBravo service stopped and deleted and the RomeoBravo executable and configuration files deleted, UniformAlfa concludes by generating and executing a suicide script which it writes to disk as `001pdm.bat`.

19. [Uninstaller] UniformJuliett

The SierraJuliett-MikeOne malware instructs its members to download UnformJuliett, the streamlined uninstaller for the malware. Using the same dialog-based installer design pattern seen in other Lazarus Group malware families—most notably, IndiaWhiskey-Three (see Section 13.3)—but using the template name of `MAIN`, UniformJuliett performs the following tasks:

1. Attempt to open a handle to the `Rpcss` service, terminate if unsuccessful.
2. Open a handle to the `wauserv` service
3. Stop the `wauserv` service
4. Wait 1 second and then delete the `wauserv` service
5. Delete the SierraJuliett-MikeOne binary at `%SYSDIR%\wauserv.dll`
6. Generate and execute suicide script

Task 1, at first glance, seems unusual, as the `Rpcss` service is a legitimate service that SierraJuliett-MikeOne has shown no interaction with in the observed samples. The purpose of Task #1 appears to be determining if UniformJuliett has sufficient administrative privileges to interact with the Windows Services Manager through the Windows API. Since `Rpcss` is a required system service, being unable to access the service via the Windows API means that the application does not have privileges. Of course, this raises the question as to why UniformJuliett does not use the SierraJuliett-MikeOne service for this test. The answer would appear to be that the test could be less reliable, as it is possible that the service for SierraJuliett-MikeOne may have been removed by an administrator or antivirus product or the installation of the service may have been faulty. Since the test of necessary privileges is a prerequisite of the subsequent tasks, ensuring the reliability of the test is paramount.

The removal of the service is a straightforward set of calls to the Windows API functions that allow interaction with the Windows services database. The deletion of the SierraJuliett-MikeOne binary, Task 5, has a peculiar code artifact, however. After constructing the filename and path information to the binary, UniformJuliett uses the directory hierarchy verification and generation function found in many of the Lazarus Groups families. This is wholly unnecessary, since the SierraJuliett-MikeOne binary is stored within the `%SYSDIR%` directory, which is always available, else the Windows operating system would fail to load.

The suicide script that UniformJuliett generates is identical to the script found in IndiaCharlie. The generation of the suicide script is the same in both, but the execution is different. IndiaCharlie uses the `WinExec` API function while UniformJuliett uses the `CreateProcess` API function. Functionally, both of the binaries perform the same task. But this does indicate that sometime in the two years that occurred between UniformJuliett was compiled and the first observed IndiaCharlie binary was compiled, the developer(s) decided to modify their code for some unknown reason.

20. Conclusion

While the capabilities for the installers, loaders, and uninstallers in this report are relatively straight forward and single-focused, analysis of these malware families provide further insight into the capabilities of the Lazarus Group. The malware families within this report also show the differing level of refinement observed across Lazarus Group malware: ranging from simplistic to convoluted to more advanced, but all ultimately effective. Analysis of these malware families showcases some of common Lazarus Group strategies of code reuse and the customization of open-source code libraries to alleviate some software development burdens. The use of decoy documents also reveals some of the potential targets of the group's malicious activity, specifically the use of decoy documents in spear phishing attacks observed targeting South Korean government and aerospace organizations.

Regardless of their sophistication or refinement, the malware families within the Lazarus Group's India and Lima classes perform at a reasonable level for their designed purpose: the introduction and persistence of malware from the Lazarus Group on a victim's infrastructure.

NOVETTA

McLean, Virginia – Headquarters
7921 Jones Branch Drive
5th Floor
McLean, VA 22102
Phone: (571) 282-3000
www.novetta.com

www.OperationBlockbuster.com