

★OPERATION★  
**BLOCKBUSTER**

# REMOTE ADMINISTRATION TOOLS & CONTENT STAGING MALWARE REPORT



**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](http://www.novetta.com)

[www.OperationBlockbuster.com](http://www.OperationBlockbuster.com)

# TABLE OF CONTENTS

<b>1. Introduction.....</b>	<b>4</b>
<b>2. Romeo-CoreOne Design Pattern .....</b>	<b>5</b>
<b>3. [RAT] RomeoAlfa .....</b>	<b>12</b>
<b>4. [RAT] RomeoBravo.....</b>	<b>18</b>
<b>5. [RAT] RomeoCharlie .....</b>	<b>20</b>
<b>6. [RAT] RomeoDelta .....</b>	<b>23</b>
<b>7. [RAT] RomeoEcho.....</b>	<b>27</b>
<b>8. [RAT] RomeoFoxtrot.....</b>	<b>29</b>
<b>9. [RAT] RomeoGolf.....</b>	<b>31</b>
<b>10. [RAT] RomeoHotel.....</b>	<b>33</b>
<b>11. [RAT] RomeoMike.....</b>	<b>37</b>
<b>12. [RAT] RomeoNovember.....</b>	<b>39</b>
<b>13. [RAT] RomeoWhiskey (Winsec).....</b>	<b>40</b>
13.1 RomeoWhiskey-One (Base Code) .....	41
13.2 RomeoWhiskey-Two.....	45
<b>14. [Spreader] SierraAlfa .....</b>	<b>50</b>
14.2 SierraAlfa-Two .....	55
<b>15. [Spreader] SierraBravo (Brambul).....</b>	<b>56</b>
15.2 SierraBravo-Two .....	62
<b>16. [Spreader] SierraCharlie .....</b>	<b>63</b>
<b>17. [P2P Staging] SierraJuliatt-MikeOne (Joanap Mk I.) .....</b>	<b>64</b>
17.1.1 Crawler Channel.....	67
17.1.2 RAT Channel.....	68
17.1.3 Sync Channel .....	70
17.2 Client Mode Thread .....	73
17.3 Known SierraJuliatt-MikeOne Command Files .....	78
<b>18. [P2P Staging] SierraJuliatt-MikeTwo (Joanap Mk. II).....</b>	<b>80</b>
<b>19. [Webserver] HotelAlfa .....</b>	<b>81</b>
<b>20. Conclusion.....</b>	<b>83</b>

# 1. Introduction

## THE LAZARUS GROUP

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 RATs and staging malware found within the Lazarus Group's collection.

A Remote Administration Tool (RAT), or Remote Administration Trojan, is a piece of malicious code that gives an attacker control over certain aspects of the infected system. At a minimum, a RAT allows an attacker to execute commands on a victim's machine. A typical RAT also provides functionality to upload and download files from a victim's computer as well. The most common communication mode for a RAT is to act as a client to a remote server. The Lazarus Group employs a variety of RATs that operate in both client mode and server mode. In server mode, a RAT waits for an incoming connection from a C2 client which requires the infected host to have a routable IP address and the ability to listen on a given port.

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.<sup>1</sup> 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, RomeoAlfa specifies a RAT 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.

<sup>1</sup> International Civil Aviation Organization. "Alphabet - Radiotelephony", <http://www.icao.int/Pages/AlphabetRadiotelephony.aspx> Accessed 1 December 2015.

## 2. Romeo-CoreOne Design Pattern

.....

A large portion of the Lazarus Group's RAT collection stems from a common core, Romeo-CoreOne (R-C1); the individual families that use R-C1 need only provide the scaffolding to support the R-C1 code. At a minimum, each family that is built upon R-C1 must provide an interface to their specific communications abstraction and a method by which to activate the R-C1 functionality.

The general flow of execution for families that use R-C1 is as follows:

1. Dynamically load API functions
2. Perform any configuration management tasks that the family may require (e.g., loading the configuration, opening listening ports, establishing persistence)
3. Establish a communication channel with controlling endpoint
4. Pass off the channel to the R-C1 component

There are five known families that are based on, or that incorporate, R-C1 (Figure 2-1): RomeoAlfa, RomeoBravo, RomeoCharlie, RomeoHotel, and RomeoNovember. In addition to the four families having commonality through the use of R-C1, two of the families, RomeoAlfa and RomeoHotel, share the distinctive fake TLS communication scheme and use the Caracachs encryption scheme as their underlying communication encryption. RomeoBravo, RomeoCharlie, and RomeoNovember use DNSCALC-style encoding for communication encryption.

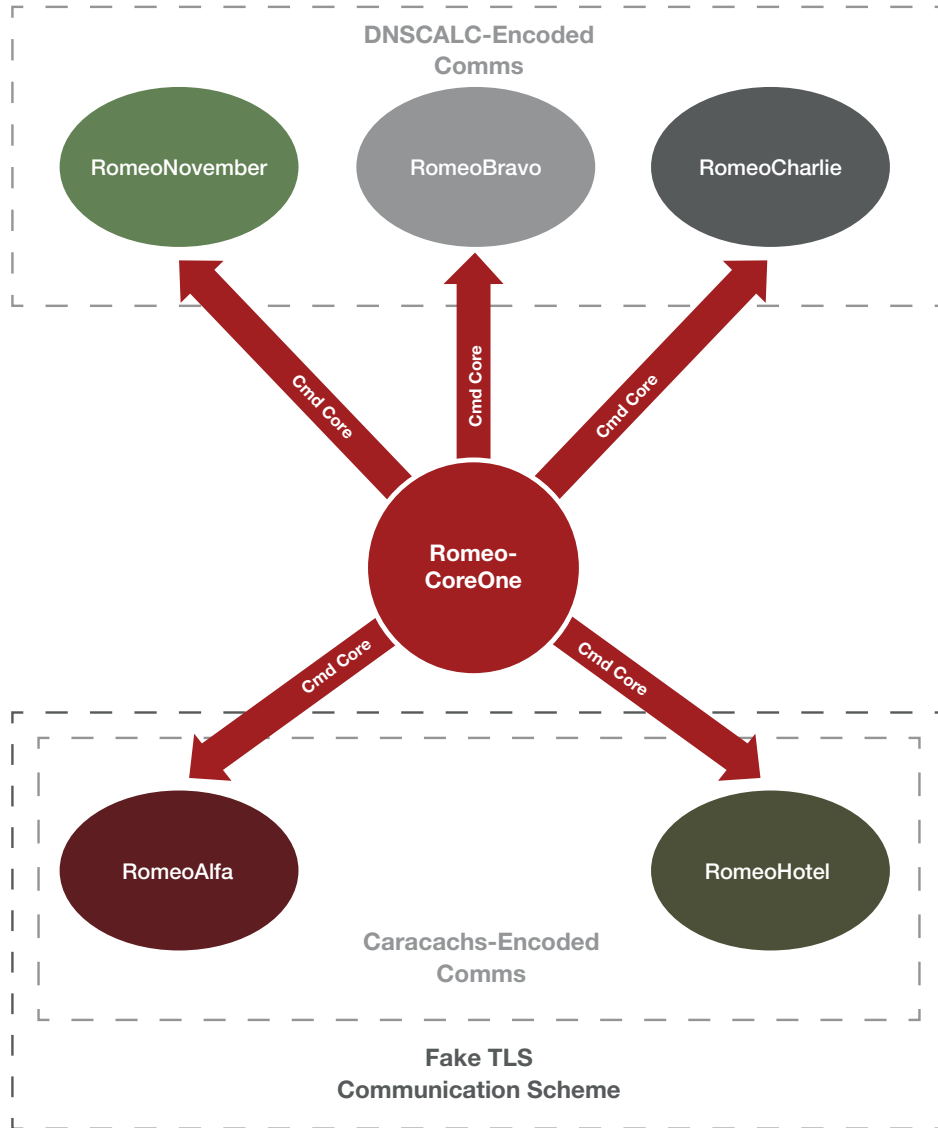


Figure 2-1: Romeo-Core1's Relationship to RomeoAlfa, RomeoBravo, RomeoCharlie, RomeoHotel, and RomeoNovember and Their Communication Underpinnings

There is significant overlap of the development periods between the different families that utilize R-C1, as seen in Figure 2-2. Each family contains a core set of R-C1 commands, and within each family there is largely a consistency among the additional commands that the families support. This indicates that the developer(s) of each family builds off the base of R-C1 but generally speaking do not share additional functionality across family boundaries.

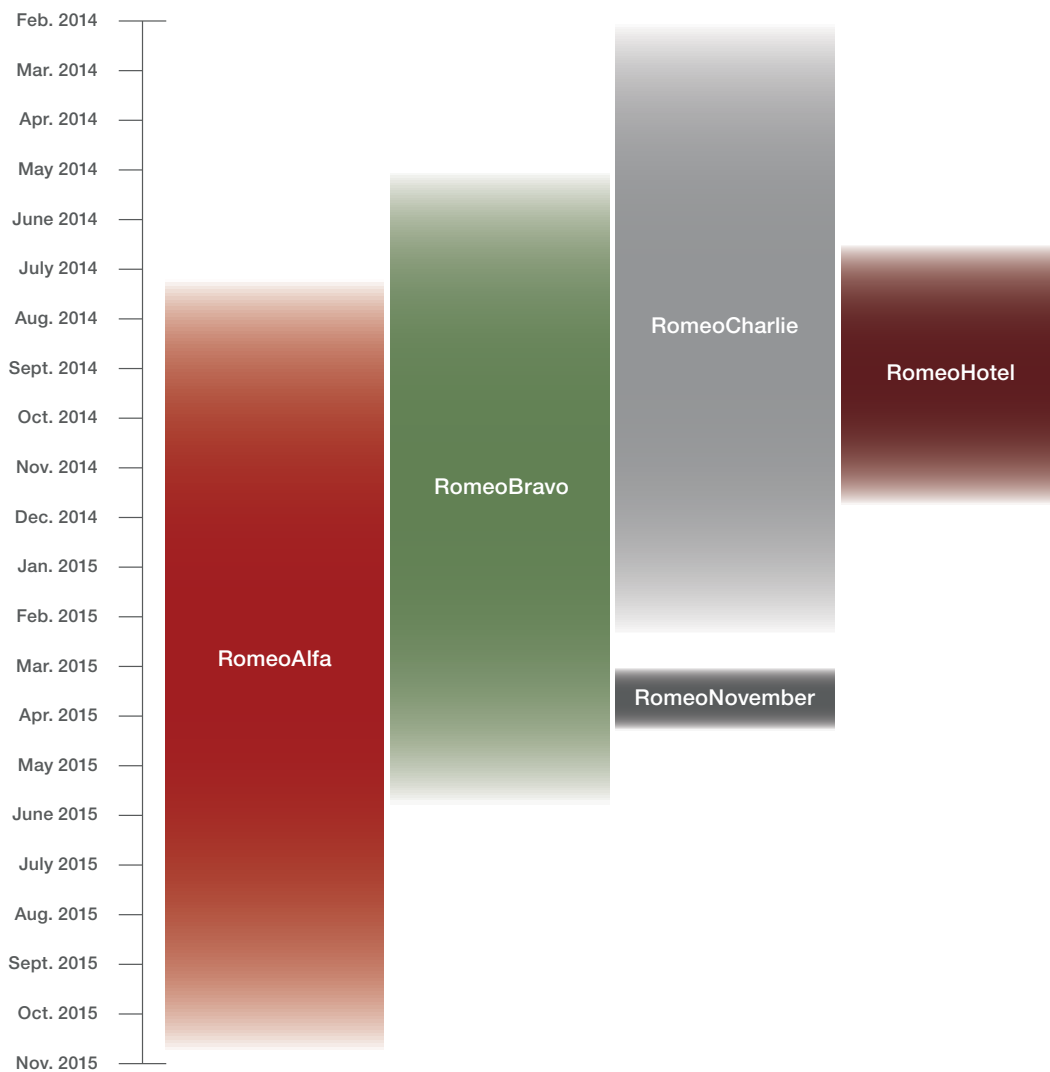


Figure 2-2: Timeline of Observed Romeo-CoreOne Based Families

The bulk of the R-C1's functionality exists within the command processing function, which the developer(s) gave the name **MessageThread**. Within **MessageThread** the following tasks occur:

1. Enter an infinite loop
2. Wait for incoming data from controlling endpoint
3. If the wait period (generally ranging from 1,200 to 36,000 seconds) for incoming data expires, exit the loop and Romeo-CoreOne
4. Read a datagram containing the command identifier and any optional parameters for the requested command
5. Using a case/switch statement, locate and then execute the appropriate command handler function for the requested command
6. Repeat at the top of the loop until a disconnection command occurs or an error with any of the commands occurs, then terminate the loop and exit Romeo-CoreOne

The number of commands offered by R-C1 has fluctuated slightly over time, with a handful of commands coming in and out of the available set. The order of the commands and basic structure of **MessageThread** has remained largely constant, however. Table 2-1 identifies the commands that R-C1 supports and the families that implement those commands. The fact that not all families support all of the R-C1 commands could be the result of new commands coming into existence and older commands being retired over time, or the developer(s) cherry-picking the specific subset of commands required for a particular family's task. Additionally, not all variants of the same family necessarily support the same set of commands. Those families who have differing command sets have a hollow symbol (△) instead of a solid symbol (▲) to indicate the discontinuity between variants.

FOUND IN ONE OR MORE VARIANTS OF					COMMAND	DESCRIPTION
Alfa	Bravo	Charlie	Hotel	November		
▲	▲	▲	▲	▲	Move File	Moves or renames a file.
▲	▲	▲	▲	▲	Directory Statistics	Pulls the number of files and directories under the specified directory along with the total size in bytes of all files.
▲	▲	▲	▲	▲	Enumerate Drives	Returns details about each logical drive including the bytes per sector and number of free sectors.
▲	▲	▲	▲	▲	Enumerate Directory	Returns a directory listing for the specified directory using the specified file mask (e.g. " <b>*.*</b> ")
▲	▲	▲	▲	▲	Write File	Writes a file supplied by the connected endpoint to disk.
▲	▲	▲	▲	▲	Read File	Transfers a local file to the connected endpoint.
△	▲	▲	▲		Upload Directory as Archive	Generates a ZIP archive file (stored in <b>%TEMP%</b> as <b>DQ</b> {random} or <b>QB</b> {random}) of the specified directory and its subdirectories' contents then transfers the file to the connected endpoint. Uses the open source project Zip Utils <sup>1</sup> to perform the archive.
▲	▲	▲	▲	▲	Create Process	Starts a new process using the command line specified by the connected endpoint.
▲	▲	▲	▲	▲	Secure Delete	Securely deletes the file specified by the connected endpoint.
▲	▲	▲	▲	▲	<i>Mimic Timestamp</i>	Duplicates the timestamp of the file specified by the connected endpoint (source) onto the file (target) specified by the connected endpoint.
▲	▲	▲	▲	▲	<i>Execute Shell Command with Output Upload</i>	Execute a command via the command line:  <b>cmd.exe /c {specified command} &gt; {output file} 2&gt;&amp;1</b>  The output file is a file located in the <b>%TEMP%</b> directory having the name <b>PM</b> {random number}. <b>tmp</b> or <b>DM</b> {random number}. <b>tmp</b> . The output file is read up to 60 times and transferred to the connected endpoint before being securely deleted.
▲	▲	▲	▲	▲	<i>Enumerate Processes</i>	Returns a list of all running processes along with details about each process that includes the process's name, PID, parent PID, start time, user's name and domain, and the full path of the process's executable.
▲	▲	▲	▲	▲	<i>Terminate Process</i>	Terminates a process by PID. The PID can be specified as either a DWORD value or an ASCII string representation of the PID number.

1 | Ijw1004. CodeProject. "Zip Utils - clean, elegant, simple, C++/Win32". <http://www.codeproject.com/Articles/7530/Zip-Utils-clean-elegant-simple-C-Win>. 19 Sep 2012



FOUND IN ONE OR MORE VARIANTS OF					COMMAND	DESCRIPTION
Alfa	Bravo	Charlie	Hotel	November		
		▲			<i>Change Listening Port</i>	Changes the configured listening port and saves the information to the configuration before restarting the R-C1 component.
▲	▲	▲	▲	▲	<i>System Information</i>	Returns details about the victim's system, the malware's configuration, specific flags indicating if several ports of interest are open, if the current user is running under Terminal Services, if the session's screen saver is on, and the state of all terminal services sessions.
▲	▲	▲	▲	▲	<i>Change Directory</i>	Changes the current working directory to the directory specified by the connected endpoint.
▲	▲	▲	▲	▲	<i>Port Knock</i>	Returns the status of connection test to the network address/port specified by the connected endpoint.
		▲			<i>Activate Proxy</i>	Activates a relay between the R-C1 instance and the network address/port specified by the connected endpoint.
▲	▲	▲	▲	▲	<i>Send Status Value</i>	Sends a status value (DWORD) to the connected endpoint.
▲	▲	△	▲	▲	<i>Disconnect</i>	Cleanly disconnects the connection between the R-C1 instance and the connected endpoint.
▲	▲		▲	▲	<i>Get Config</i>	Sends a copy of the current running configuration to the connected endpoint.
▲	▲		▲	▲	<i>Set Config</i>	Receives a new running configuration from the connected endpoint.
△			▲		<i>RunAs</i>	Runs a process specified by the connected endpoint as the user specified by the connected endpoint.
▲			▲		<i>NOP</i>	No operation.
△			▲		<i>Suicide</i>	Removes the malware from the victim's system and disconnects from the connected endpoint.

Table 2-1: Romeo-CoreOne's Supported Commands (in Their Identification Order) and the Families that Implement the Commands

The numbering scheme for commands within R-C1 are generally consistent across the families that utilize the code. While the base number for the commands differs by family (and in some cases, by variants of a family), the sequence of commands remains largely unchanged. For example, *Move File* is always the first command (base number + 0), *Directory Statistics* is always the second command (base number + 1), and so on. Table 2-2 identifies the offset for each supported command from the command base number for each family that uses R-C1.

OFFSET FROM BASE COMMAND NUMBER					COMMAND
Alfa	Bravo	Charlie	Hotel	November	
+0x00	+0x00	+0x00	+0x00	+0x00	Move File
+0x01	+0x01	+0x01	+0x01	+0x01	Directory Statistics
+0x02	+0x02	+0x02	+0x02	+0x02	Enumerate Drives
+0x03	+0x03	+0x03	+0x03	+0x03	Enumerate Directory
+0x04	+0x04	+0x04	+0x04	+0x04	Write File
+0x05	+0x05	+0x05	+0x05	+0x05	Read File

OFFSET FROM BASE COMMAND NUMBER					COMMAND
Alfa	Bravo	Charlie	Hotel	November	
+0x06	+0x06	+0x06	+0x06	-	Upload Directory as Archive
+0x08	+0x08	+0x08	+0x08	+0x08	Create Process
+0x09	+0x09	+0x09	+0x09	+0x09	Secure Delete
+0x0A	+0x0A	+0x0A	+0x0A	+0x0A	Mimic Timestamp
+0x0B	+0x0B	+0x0B	+0x0B	+0x0B	Execute Shell Command with Output Upload
+0x0C	+0x0C	+0x0C	+0x0C	+0x0C	Enumerate Processes
+0x0D	+0x0D	+0x0D	+0x0D	+0x0D	Terminate Process
-	-	+0x0E	-	-	Change Listening Port
+0x0E	+0x0E	+0x0F	+0x0E	+0x0E	System Information
+0x0F	+0x0F	+0x10	+0x0F	+0x0F	Change Directory
+0x10	+0x10	+0x11	+0x10	+0x10	Port Knock
-	-	+0x12	-	-	Activate Proxy
+0x12	+0x12	+0x14	+0x12	+0x12	Send Status Value
+0x17	+0x17	+0x19	+0x17	+0x17	Disconnect
+0x18	+0x18	-	+0x18	+0x18	Get Config
+0x19	+0x19	-	+0x19	+0x19	Set Config
+0x1A	-	-	+0x1A	-	RunAs
+0x1B	-	-	+0x1B	-	NOP
+0x1C	-	-	+0x1C	-	Suicide

Table 2-2: Offset from Base Command Number of Commands for Romeo-CoreOne for RomeoAlfa, RomeoBravo, RomeoCharlie, RomeoHotel, and RomeoNovember

RomeoCharlie disrupts the overall consistency of the command-to-offset mapping between the various R-CI-based families due to the introduction of the *Change Listening Port* and *Activate Proxy* commands. Despite the minor disruption of the number scheme, each family exhibits the same offsets between commands (ignoring the RomeoCharlie additions, of course). The consistency of the numbering scheme provides evidence to suggest that the developer(s) behind R-CI customize particular commands between families. The *Suicide* and *RunAs* commands found in RomeoHotel are technically available in RomeoAlfa but have been reduced to little more than *NOP* commands. It is possible to make this claim based on the fact that the numbering scheme is consistent between families, both RomeoAlfa and RomeoHotel support commands at offset 0x1A (*RunAs*) and 0x1C (*Suicide*), and, despite their code being slightly different, they have the same basic structure. Figure 2-3 illustrates how the code structure for RomeoHotel and RomeoAlfa are nearly identical despite RomeoAlfa's functional handicapping.

```

int Cmd_RunCommandAsUser(SOCKET s, LPWSTR lpCommandLine)
{
    unsigned char b[4];
    int result = ReceiveDataFromC2(s, b, 4, 1, 0);
    if ( !result )
    {
        if ( RunCommandAsUser(s, lpCommandLine) )
            result = SendResponseCodeToC2(s, 0x9751, 0);
        else
            result = SendResponseCodeToC2(s, 0x9750, 0);
    }
    return result;
}

int Cmd_RunCommandAsUser(SOCKET s)
{
    unsigned char b[4];
    int result = ReceiveDataFromC2(&g_fConnectToC2, s, b, 4, 1, 0);
    if ( !result )
    {
        if ( returnZer0() )
            result = SendResponseCodeToC2(&g_fConnectToC2, v1, 0x8751, 0);
        else
            result = SendResponseCodeToC2(&g_fConnectToC2, v1, 0x8750, 0);
    }
    return result;
}

```

Figure 2-3: Comparison of the RunAs Command's (offset +0x1A) Function in RomeoHotel (Upper) and RomeoAlfa (Lower)

Table 2-2 somewhat obscures an important trait of the numbering scheme used by R-CI. There are gaps within the number scheme that have significance. Many commands within R-CI return a status code to indicate the success or failure of a command which contributes to at least two of the gaps within the numbering scheme. R-CI uses two numbers within the numbering scheme as the success and failure response codes as indicated in Table 2-3. There is evidence that PapaAlfa acts as a proxy for several R-CI-based families and as a result, at least one of the gaps in the number scheme is the result of a PapaAlfa-based status code. The purpose or reasoning behind the remaining three gaps is currently unclear.

OFFSET FROM BASE COMMAND NUMBER					PURPOSE
Alfa	Bravo	Charlie	Hotel	November	
+0x07	+0x07	+0x07	+0x07	+0x07	Unknown
+0x11	+0x11	+0x13	+0x11	+0x11	Unknown
+0x13	+0x13	+0x15	+0x13	+0x13	Unknown (probably related to relaying)
+0x14	+0x14	+0x16	+0x14	+0x14	Relay Status: Failure
+0x15	+0x15	+0x17	+0x15	+0x15	Command Status: Success
+0x16	+0x16	+0x18	+0x16	+0x16	Command Status: Failure

Table 2-3: Romeo-CoreOne Numbering Gaps and Their Purpose

### 3. [RAT] RomeoAlfa

.....

RomeoAlfa is a client-mode RAT that utilizes the R-CI framework and command set (see Section 2). There are two observed variants, RomeoAlfa-One and RomeoAlfa-Two. Functionally, each variant performs the same basic operations and supports the same R-CI command set. What differentiates the variants are subtle, but important, structural changes as defined in Table 3-1.

FEATURE	ROMEOALFA-ONE	ROMEOALFA-TWO
Configuration initialization	Performed in <b>WinMain</b>	Performed in separate function, called by <b>main</b>
Dynamic API loading string obfuscation	Space-Dot	Caracachs
Romeo-CoreOne base command number	0x873B	0x8374
Request Channel	0x6456	0x3594
Configuration data structure size	728 bytes	2784 bytes

Table 3-1: Key Differences between RomeoAlfa-One and RomeoAlfa-Two

The most apparent difference between RomeoAlfa-One and RomeoAlfa-Two is the configuration initialization component of the RomeoAlfa scaffolding code. The initialization of the configuration in both RomeoAlfa variants is functionally identical as seen in Figure 3-1 and Figure 3-2. The difference between the two is the location of the code. While RomeoAlfa-One performs the initialization inline within the **WinMain** function, RomeoAlfa-Two has an entirely separate function that is responsible not only for initializing the configuration parameters but also for dynamically loading the API functions.

```

int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nShowCmd)
{
    wszC2Entry *p = (wszC2Entry *)&config;
    do
    {
        wcsncpy(p->c2, L"0.0.0.0");
        ++p;
    }
    while ( p < &config.field_190 );
    wcsncpy(config.arrwszC2s[0].c2, L"203.131.222.102");
    config.arrdwC2Ports[0] = 443;
    wcsncpy(config.arrwszC2s[5].c2, L"208.105.226.235");
    config.arrdwC2Ports[5] = 443;
    config.dwSleepTimeBetweenReconnects = 60;
    config.field_2D0 = 0;
    config.dwStartupDelayInSeconds = 0;
    config.qwSysInfoPacketsSent = 0i64;
    config.dwConnectionAttemptsBeforeSleep = 5;
    srand(GetTickCount()^ time(0));
    config.initialRandomNumber = rand();
    MoveTrojanToStartupFolder();
    MainLoop(0);
    log((char *)"-----End-----!\n");
    return 0;
}

```

Figure 3-1: RomeoAlfa-One's WinMain with the Configuration Initialization Code Inline

```

int Initialize()
{
    LoadKernel32APIs();
    LoadWs2_32APIs();
    LoadAdvapiAPIs();
    LoadIphlpapiAPIs();
    if ( InitializeWinsock() )
    {
        return 1;
    }
    else
    {
        wszC2Entry *p= &config.arrwszC2s[0];
        do
        {
            wcsncpy(p->c2, L"0.0.0.0");
            ++p;
        }
        while ( p < &config.field_190 );
        wcsncpy(config.arrwszC2s[0].c2, L"91.183.71.18");
        config.arrdwC2Ports[0] = 443;
        wcsncpy(config.arrwszC2s[3].c2, L"160.218.101.125");
        config.arrdwC2Ports[3] = 443;
        wcsncpy(config.arrwszC2s[6].c2, L"37.34.176.14");
        config.arrdwC2Ports[6] = 443;
        config.dwSleepTimeBetweenReconnects = 60;
        config.field_AD0 = 0;
        config.field_ACC = 0;
        config.field_AC0 = 0;
        config.field_AC4 = 0;
        config.dwConnectionAttemptsBeforeSleep = 5;
        srand(GetTickCount() ^ time(0));
        config.field_initialRandomNumber AB8 = rand();
        config.dwStartupDelayInSeconds = 0;
        config.field_ADC = 0;
    }
    return 0;
}

```

Figure 3-2: RomeoAlfa-Two's Configuration Initialization Function

The size of the configuration data structure varies greatly between RomeoAlfa-One and RomeoAlfa-Two. Notably, the configuration data structure of RomeoAlfa-Two matches the configuration data structure of RomeoHotel (see Section 10). In addition, the fields are identical, meaning that RomeoAlfa-Two and RomeoHotel use the same configuration. Yet, while the data structure and fields are identical, RomeoAlfa-Two does not fully utilize all of the configuration data fields. The most probable reason for this behavior is that RomeoAlfa-Two and RomeoHotel use a common code for the scaffolding surrounding R-Cr, but do not implement all of the same features.

A noteworthy fact about the configuration's initialization is the setup of the C2 server addresses. The configuration data structure supports 10 different C2 servers for RomeoAlfa to randomly select when making a connection. The initialization sequence begins by first replacing all 10 entries with **0.0.0.0** to indicate the entry is invalid. Then the initialization code replaces non-sequential entries with the actual C2 server IP addresses. This is interesting for two reasons: the configuration is hardcoded at compile time, meaning any changes to the configuration will not persist after a reboot, and there are likely additional C2 servers that have been commented out in the source code that fill the missing gaps between C2 server IP addresses.

RomeoAlfa maintains its persistence by locating itself within the Start Menu folder (**CSIDL\_STARTUP**) or the All User's Start Menu (**CSIDL\_COMMON\_STARTUP**) of the victim's computer. After initializing the configuration data structure, RomeoAlfa moves its current running executable to the Start Menu directory via a call to **MoveFile**.

RomeoAlfa randomly selects a C2 server from the ten possible server addresses within the configuration data structure and use the selected address to establish a connection. If the connection to the selected C2 server fails or the selected C2 server's address is **0.0.0.0**, a new C2 server is randomly selected and another attempt is made. If 10 unsuccessful attempts are made, RomeoAlfa sleeps for a defined period of time (as specified in the configuration data structure, typically set to 60 seconds) before attempting the random C2 server selection and connection process again.

One of the most distinctive features of RomeoAlfa is the use of the fake TLS communication scheme to encrypt end-to-end communication as well as to obfuscate the nature of the communication. After successfully connecting to a C2 server, RomeoAlfa generates a seemingly legitimate TLS channel, complete with the appropriate handshake protocol packets. However, the malware ultimately uses an encryption method that is not supported by the TLS standard. This encryption method therefore blends in with legitimate TLS traffic while at the same time is immune to SSL/TLS man-in-the-middle proxying that would reveal the plaintext of the communication.

After establishing the fake TLS channel, RomeoAlfa exchanges a 12-byte datagram (through the fake TLS channel) with the C2 server as the first step in a two-part handshake. The datagram contains a channel identifier and the current connection state of the RomeoAlfa instance (Table 3-2). The channel identifier corresponds to the channel identifiers found within PapaAlfa samples. The code for establishing the handshake (Figure 3-3) is the exact inverse of the handshake function found in PapaAlfa. This implies that RomeoAlfa (and, as explained in later sections, RomeoBravo and RomeoHotel) can interface with a PapaAlfa instance, which in turn may be little more than a proxy point for the real C2 server. Regardless of the presence of PapaAlfa in the communication stream, the commands ultimately come from the C2 server.

STATE VALUE	DESCRIPTION
1	Initialized state (only occurs prior to first C2 connection attempt)
2	Sleep state
3	Not used in RomeoAlfa
4	Not used in RomeoAlfa
5	Active state

Table 3-2: RomeoAlfa's State Values

```

int __thiscall EstablishHandshake(DWORD *pfConnectionStatus, SOCKET s, DWORD
dwChannel, DWORD *pdwRequestorIP, DWORD *pdwRequestChannelReply, int dwClientState)
{
    int result;
    struct {
        DWORD dwChannel;
        DWORD dwIPAddress;
        DWORD dwState;
    } outgoing, incoming;

    incoming.dwChannel = 0;
    incoming.dwIPAddress = 0;
    incoming.dwState = 0;
    outgoing.dwIPAddress = 0;
    outgoing.dwChannel = dwChannel;
    outgoing.dwState = dwClientState;
    if ( SendDataToC2(pfConnectionStatus, s, &outgoing, sizeof(outgoing), 1) )
    {
        result = 1;
    }
    else if ( ReceiveDataFromC2(pfConnectionStatus, s, &incoming, sizeof(incoming), 1, 0)
)
    {
        result = 1;
    }
    else
    {
        *pdwRequestorIP = incoming.dwIPAddress;
        *pdwRequestChannelReply = incoming.dwChannelReply;
        result = 0;
    }
    return result;
}

```

Figure 3-3: RomeoAlfa's Handshake Function

The second half of the handshake consists of the C2 server responding with another 12-byte datagram containing RomeoAlfa's external IP address (**incoming.dwIPAddress**) and a response to the requested channel (**incoming.dwChannel**), which is typically one value higher than the requested channel. The conclusion of the handshake indicates a successful communication channel is established between the RomeoAlfa instance and the C2 server.

With the communication channel between RomeoAlfa and the C2 server established, the **MessageThread** of RomeoCoreOne is activated. RomeoAlfa-One fully supports 19 of the R-C1 commands and partially supports three (*RunAs*, *Upload Directory as Archive*, and *Suicide*), while RomeoAlfa-Two fully supports 20 and partially supports two (*RunAs* and *Upload Directory as Archive*). Table 3-3 identifies the commands that the RomeoAlfa variants support along with the command identifiers for each command for each variant.



ROMEOALFA-ONE	ROMEOALFA-TWO	COMMAND
0x873B	0x8374	Move File
0x873C	0x8375	Directory Statistics
0x873D	0x8376	Enumerate Drives
0x873E	0x8377	Enumerate Directory
0x873F	0x8378	Write File
0x8740	0x8379	Read File
0x8741	0x837A	Upload Directory as Archive (Defunct)
0x8743	0x837C	Create Process
0x8744	0x837D	Secure Delete
0x8745	0x837E	Mimic Timestamp
0x8746	0x837F	Execute Shell Command with Output Upload
0x8747	0x8380	Enumerate Processes
0x8748	0x8381	Terminate Process
0x8749	0x8382	System Information
0x874A	0x8383	Change Directory
0x874B	0x8384	Port Knock
0x874D	0x8386	Send Status Value
0x874F	0x8388	(RESPONSE CODE) Relay Status: Failure
0x8750	0x8389	(RESPONSE CODE) Command Status: Success
0x8751	0x838A	(RESPONSE CODE) Command Status: Failure
0x8752	0x838B	Disconnect
0x8753	0x838C	Get Config
0x8754	0x838D	Set Config
0x8755	0x838E	RunAs (Defunct)
0x8756	0x838F	NOP
0x8757	0x8390	Suicide (Defunct in RomeoAlfa-One)

Table 3-3: RomeoAlfa Command Numbers (by Variant) and Their Descriptions

RomeoAlfa deviates from the R-C1 command norms for the *RunAs*, *Upload Directory as Archive*, and, in the case of RomeoAlfa-One variants, *Suicide* commands. The *RunAs* command is reduced to little more than a NOP by virtue of the fact that the core component of the command is replaced with a function that will always return a positive status response, *Command Status: Success*, to the C2 server. Similarly, the *Upload Directory as Archive* command is reduced to a command that generates a temporary file name (starting with **DQ** in RomeoAlfa-One variants, and **QB** in RomeoAlfa-Two variants) in the **%TEMP%** directory and then attempts to upload the file specified to the C2 server, an attempt that will most likely fail. RomeoAlfa-One has disabled (by removal) the portion of the *Suicide* command that uninstalls or deletes the RomeoAlfa malware from the victim's system, leaving only the disconnection portion of the function intact.

## 4. [RAT] RomeoBravo

With a striking resemblance to RomeoAlfa, RomeoBravo is a client-mode RAT based on R-C1 (see Section 2) that operates as either a service DLL or standalone executable. The most notable differences between RomeoAlfa and RomeoBravo are the inclusion of an external configuration file and the use of DNSCALC-style encoding to obscure communication in place of the fake TLS scheme.

RomeoBravo is a service DLL and as such contains the necessary Windows service scaffolding code to operate as a legitimate Windows service. In order to operate as a Windows service, RomeoBravo exports the function **ServiceMain** which contains the functionality to provide callback functions for handling service requests from the operating system. **ServiceMain** also contains dynamic API loading functions for API functions from **kernel32.dll**, **ws2\_32.dll**, **advapi32.dll**, and **iphlpapi.dll** using Space-Dot obfuscation for the API names. **ServiceMain**, upon establishing the necessary Windows service callback and setting the appropriate state for the service, spawns a new thread for the RomeoBravo functionality.

RomeoBravo requires a configuration file in order to operate successfully. Located at **%SYSTEMDRIVE%\tmsconfig.msi**, the configuration file, is a 456-byte DNSCALC-style encoded file that contains up to 10 C2 server addresses and ports along with 5 additional configuration fields. If the file is not found on the victim's machine, the RomeoBravo thread will silently terminate.

After loading the configuration, RomeoBravo randomly selects a C2 server address from the loaded configuration. RomeoBravo attempts to connect to the selected C2 server address if the address is not **0.0.0.0**. If the connection to the C2 server fails or the randomly selected C2 server address is **0.0.0.0**, a new C2 server is randomly selected and another attempt occurs. If 10 unsuccessful attempts to connect to a C2 server occurs, RomeoBravo sleeps for a defined period of time (as specified in the configuration data structure) before attempting the random C2 server selection and connection process again.

RomeoBravo performs the same 12-byte diagram handshake with the C2 server as described in the RomeoAlfa section (see Figure 3-3) in order to select the appropriate channel. The difference between RomeoAlfa's implementation of the channel selection and RomeoBravo's implementation is the exclusion of fake TLS and the inclusion of DNSCALC-style encoding for the network traffic. RomeoBravo can potentially transmit additional states not found in RomeoAlfa (see Table 4-1). As described previously in the RomeoAlfa section, RomeoBravo may interface with PapaAlfa instances instead of the actual C2 server issuing commands, but ultimately commands come from the controlling C2 server.

STATE VALUE	DESCRIPTION
1	Initialized state (only occurs prior to first C2 connection attempt)
2	Sleep state
3	Post-connection activity event state
4	Change in number of active terminal services sessions during post-connection activity state
5	Active state

Table 4-1: RomeoBravo's States

After establishing a communication channel between RomeoBravo and the C2 server, the **MessageThread** of R-CI is activated. RomeoBravo fully supports 20 commands from R-CI's command set. Table 4-2 identifies the commands that the RomeoBravo supports along with the command identifiers for each command.

COMMAND NUMBER	COMMAND
0x523B	<i>Move File</i>
0x523C	<i>Directory Statistics</i>
0x523D	<i>Enumerate Drives</i>
0x523E	<i>Enumerate Directory</i>
0x523F	<i>Write File</i>
0x5240	<i>Read File</i>
0x5241	<i>Upload Directory as Archive</i>
0x5243	<i>Create Process</i>
0x5244	<i>Secure Delete</i>
0x5245	<i>Mimic Timestamp</i>
0x5246	<i>Execute Shell Command with Output Upload</i>
0x5247	<i>Enumerate Processes</i>
0x5248	<i>Terminate Process</i>
0x5249	<i>System Information</i>
0x524A	<i>Change Directory</i>
0x524B	<i>Port Knock</i>
0x524E	<i>Send Status</i>
0x5250	(RESPONSE CODE) <i>Relay Status: Failure</i>
0x5251	(RESPONSE CODE) <i>Command Status: Success</i>
0x5252	(RESPONSE CODE) <i>Command Status: Failure</i>
0x5253	<i>Disconnect</i>
0x5254	<i>Get Config</i>
0x5255	<i>Set Config</i>

Table 4-2: RomeoBravo's Supported Commands and Their Identifiers

The code used by RomeoBravo's implementation of R-CI is nearly identical to that used by RomeoAlfa with the exception of the *Secure Delete* command. RomeoBravo's *Secure Delete* implementation does not attempt to rename the targeted file with a randomly generated name of equal size, as is normally done by the Lazarus Group, but instead generates a new filename with the form **TMP**{random decimal number}.**tmp** and makes a call to **MoveFile** to rename the file. After the file is renamed, the file is then deleted. This is a minor change in form from others that use the secure deletion functionality within the Lazarus Group, but it is a notable change that makes the particular implementation unique.

## 5. [RAT] RomeoCharlie

With observed compile dates going back to February 5, 2014, RomeoCharlie is one of the oldest R-C1-based RATs (see Section 2) in the Lazarus Group's collection. A server-mode RAT, RomeoCharlie uses DNSCALC-style encoding for network communication and RSA encryption for client authentication. There are two observed variants, RomeoCharlie-One and RomeoCharlie-Two. The differences between the two are cosmetic in nature, as Table 5-1 illustrates:

FEATURE	ROMEOCHARLIE-ONE	ROMEOCHARLIE-TWO
Dynamic APIs Loaded	114 Functions from 10 DLLs	103 Functions from 4 DLLs
Start of Execution	<b>ServiceMain</b>	<b>DllMain</b>
Configuration Location	<b>HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security\xc123465- efff-87cc-37abcdef9</b>	<b>HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security\ffcf3465- efff-87cc-37abcdef9</b>
Startup Delay	0 Seconds	3 Seconds
Romeo-CoreOne Commands	19	20

Table 5-1: Key Differences between RomeoCharlie-One and RomeoCharlie-Two

RomeoCharlie-One operates as a service DLL that dictates that the binary must conform to the basic guidelines of a Windows service – specifically, the export of the **ServiceMain** function. **ServiceMain** contains the necessary scaffolding code to ensure the Windows Services subsystem treats the binary as a legitimate service, but the function also contains several dynamic API loading functions for **kernel132.dll**, **ws2\_32.dll**, **advapi32.dll**, **oleaut32.dll**, **iphlpapi.dll**, **urlmon.dll**, **wininet.dll**, **user32.dll**, **shell32.dll**, and **shlwapi.dll**. All said, RomeoCharlie-One loads 114 different API functions into memory, but only 59 (approximately 52%) of these API functions are ever used by the malware. RomeoCharlie-One loads APIs from, but completely ignores, 6 of the 10 DLLs, indicating that the dynamic API loading functions are part of a larger library of code.

RomeoCharlie-Two, on the other hand, does not require the complete Windows Services scaffolding as its sibling and therefore does not export **ServiceMain**, nor does it even contain a **ServiceMain**-type function. Instead, execution of the RomeoCharlie-Two code begins in **DllMain** as a newly generated thread. The RomeoCharlie-Two code begins with a 3 second sleep delay before dynamically loading API functions from **kernel132.dll**, **ws2\_32.dll**, **advapi32.dll**, and **iphlpapi.dll**. Of the 103 API functions that RomeoCharlie-Two loads, the malware uses 73 (approximately 71%) of the functions, a dramatic increase from RomeoCharlie-One.

Both variants of RomeoCharlie begin, after the dynamic API loading, by loading their configuration from the registry into memory. IndiaBravo-RomeoCharlie generates and stores the configuration for RomeoCharlie under the registry branch **HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security**. The key under which the configuration resides varies by variant: RomeoCharlie-One's configuration exists under **xc123465-  
efff-87cc-37abcdef9**, while RomeoCharlie-Two's configuration exists under **ffcf3465-  
efff-87cc-37abcdef9**. Both configurations are the same size (120 bytes) and contain a minimal amount of information: the listening port, the name of the service name under which the RomeoCharlie malware is running, the number of authenticated connections (from clients) that have occurred, and a 64-bit random value (presumably a unique identifier for the infection).

With the configuration of the RomeoCharlie variants loaded into memory, the differences between RomeoCharlie-One and RomeoCharlie-Two cease (save for one exception that will be explained). RomeoCharlie is a server-mode RAT and, as such, must establish a listening port. Before a listening port is established at the Winsock level, RomeoCharlie first opens a hole in the Windows Firewall to allow incoming connections on the desired listening port (as specified in the configuration). The task of opening a firewall port consists of constructing and then issuing the command line seen in Figure 5-1 via **CreateProcess**.

```
cmd.exe /c netsh firewall add portopening TCP <listening port number> "adp"
```

Figure 5-1: RomeoCharlie-One's Firewall Modification Command

The command for opening the firewall is the exact same command (include the “**adp**” rule name) found in TangoAlfa and PapaAlfa. Not only is the command identical, but the method of constructing the command (Figure 5-2) is identical to the method used by both TangoAlfa and PapaAlfa.

```
sprintf(szCommandLine, "%sd.e%sc n%ssh%$rewa%$ ad%$ po%$op%$ing T%$ %d \"%s\"", "cm", "xe", "et", "fi", "ll", "d", "rt", "en", "CP", wPort, "adp")
```

Figure 5-2: Code Snippet from RomeoCharlie-One used for the Construction of the Firewall Modification Command

If RomeoCharlie is unable to execute the firewall command or bind a socket to the desired listening socket, the malware quietly stops executing. If, however, RomeoCharlie is successful in binding a listening socket and opening the firewall for the socket, the malware enters an infinite loop that waits for incoming connections. For each incoming connection, a new thread is spawned for the R-C1 **MessageThread** to handle the client's requests.

Unlike other R-C1 based families, RomeoCharlie integrates the handshake/authentication phase into the **MessageThread** function. Additionally, this handshake/authentication phase is significantly more involved than the other R-C1 based families. The handshake begins by RomeoCharlie sending a 16-byte data structure to the client, consisting of an 8-byte value containing the number of times RomeoCharlie has successfully authenticated clients followed by an 8-byte value containing the randomly generated number generated by IndiaBravo-RomeoCharlie at installation (a value that most likely is the unique identifier for the infection). The communication channel between RomeoCharlie and the client is obfuscated by DNSCALC-style encoding the data stream prior to transmission.

The client must send back a specially crafted 130-byte data blob. The data blob is RSA-encoded using an unknown private key. RomeoCharlie decrypts the data blob using the parameters specified Table 5-2. The data blob, after decryption, contains three fields: the letters a through z (lower case) as a NULL-terminated string, a NULL-terminated string representing the number of times RomeoCharlie has successfully authenticated clients plus one, and a NULL-terminated string representing the 8-byte unique infection identifier of the RomeoCharlie instance. If any of the three values do not match the expected values, the connection between the client and RomeoCharlie is terminated.

PARAMETER	VALUE
d	b076e0580463a202bad74cb9c1b85af3fb4d1be513ccca3ae 8b57d193be77b4ab63802b3216d3a80b00827b693593a76be884f41b491ee1f6136b375add91e2de9b0f5b3849d463fcd 7b9a3b6cd0744caf809f510ee04ab3c714f53422d24f33361f75145b08286d2d7d99704684ed1d25fd5a9dc7b993f8e4d074234fd82d3
n	11

Table 5-2: Components of the RSA Key RomeoCharlie Uses to Decrypt the Client's Authentication Data Blob

After establishing the authenticity of the client, MessageThread begins processing the incoming commands from the client. The final distinction between the RomeoCharlie variants is the number of commands each variant fully supports. RomeoCharlie-One supports 19 commands, while RomeoCharlie-Two supports the same 19 commands plus the *Disconnect* command. Table 5-1 lists the complete command sets for both variants.

ROMEOCHARLIE-ONE	ROMEOCHARLIE-TWO	COMMAND
0x54B7	0x54B7	<i>Move File</i>
0x54B8	0x54B8	<i>Directory Statistics</i>
0x54B9	0x54B9	<i>Enumerate Drives</i>
0x54BA	0x54BA	<i>Enumerate Directory</i>
0x54BB	0x54BB	<i>Write File</i>
0x54BC	0x54BC	<i>Read File</i>
0x54BD	0x54BD	<i>Upload Directory as Archive</i>
0x54BF	0x54BF	<i>Create Process</i>
0x54C0	0x54C0	<i>Secure Delete</i>
0x54C1	0x54C1	<i>Mimic Timestamp</i>
0x54C2	0x54C2	<i>Execute Shell Command with Output Upload</i>
0x54C3	0x54C3	<i>Enumerate Processes</i>
0x54C4	0x54C4	<i>Terminate Process</i>
0x54C5	0x54C5	<i>Change Listening Port</i>
0x54C6	0x54C6	<i>System Information</i>
0x54C7	0x54C7	<i>Change Directory</i>
0x54C8	0x54C8	<i>Port Knock</i>
0x54C9	0x54C9	<i>Proxy</i>
0x54CB	0x54CB	<i>Send Status</i>
0x54CD	0x54CD	(RESPONSE CODE) <i>Relay Status: Failure</i>
0x54CE	0x54CE	(RESPONSE CODE) <i>Command Status: Success</i>
0x54CF	0x54CF	(RESPONSE CODE) <i>Command Status: Failure</i>
	0x54D0	<i>Disconnect</i>

Table 5-3: RomeoCharlie Command Numbers (by Variant) and Their Descriptions

Within the list of supported commands, there are two outliers not found in other R-C1 based families: *Change Listening Port* and *Proxy*. The *Change Listening Port* command receives a new listening port to which the RomeoCharlie sample should bind, resulting in the malware updating and saving the configuration to the registry before terminating the connection to the client. After the client disconnect, RomeoCharlie begins the process of establishing a new listening socket (complete with changing the firewall settings and binding to the specified port) before waiting for new incoming client connections.

The *Proxy* command receives an address from the client and attempts to connect to the computer at the address. If successful, the *Proxy* command spawns two threads. The first thread simply relays any communication originating from the client's socket to the new endpoint, while the second thread relays any communication originating from the new endpoint to the client's socket. There is no encryption or obfuscation applied to the communication going through the relays. The *Proxy* command calls **WaitForMultipleObjects** to return before returning control to **MessageThread**.

## 6. [RAT] RomeoDelta

.....

RomeoDelta is a RAT dropped by IndiaFoxtrot alongside DeltaBravo. In addition to being dropped alongside DeltaBravo, RomeoDelta contains a significant amount of code and artifact overlap with DeltaCharlie, indicating a shared development environment, if not a shared developer(s).

Operating as a service DLL, RomeoDelta performs the necessary scaffolding required by Windows Services in order to appear as a legitimate service before spawning a thread that contains the core functionality of RomeoDelta. The core functionality begins by dynamically loading the necessary API functions (the names of which are obfuscated using Space-Dot encoding). RomeoDelta then determines its exclusivity on the victim's system by detecting the presence (or absence) of `Global\WindowsUpdateTracing{number}.{number}`, where the two {number} values change as the malware has evolved over time. The first {number} is typically 0 with the second {number} incrementing over time as if to indicate a versioning system. The same type of mutex name is also found in IndiaFoxtrot.

The configuration file for RomeoDelta typically exists within a file named `msxml13.xml` (or `msxml15.xml` in later variants of RomeoDelta). Encrypted using RC4, the configuration file is loaded and decrypted by RomeoDelta with the most commonly observed passwords for the configuration file being `BAISEO%$2fas9vQsfvx%$`, `C!@I#%VJSIEOTQWPVz034vuA`, and `GetFileAttributesW`.

The configuration file contains three C2 server addresses (and their listening port numbers). After loading the configuration, RomeoDelta attempts to contact one of the C2 server addresses (in the order present in the configuration). The connection process begins with RomeoDelta resolving the domain name of the C2 server. If the address resolves, the IP address is XOR'd with `0x1AB9C2D8` in order to reveal the real address of the C2 server. RomeoDelta attempts to contact the C2 server and, if successful, begins the handshake process. The handshake procedure is identical to that used by DeltaCharlie, consisting of a 16-byte authentication sequence involving the use of DNSCALC-style encoding. The following sequence occurs during the handshake between RomeoDelta and the C2 server:

1. RomeoDelta generates a random 16-byte buffer
2. RomeoDelta encodes the bytes using DNSCALC-style encoding
3. RomeoDelta sends the encoded buffer to the C2 server
4. The C2 server replies with the original 16-bytes (before the DNSCALC-style encoding)
5. RomeoDelta verifies the original 16-byte buffer and sends back a packet with the identifier of `0x611` to indicate a successful handshake.

The handshake that RomeoDelta (as well as DeltaCharlie) employs has a striking similarity to the handshake procedure of SierraJuliett-MikeOne.

After the handshake, RomeoDelta transmits a portion of the configuration to the C2 server. If the configuration of RomeoDelta does not contain a non-zero ID number, the malware transmits details of the victim's system to the C2 server and the C2 server responds with an ID number.

For later variants of RomeoDelta, the malware will locate the directory containing the keylog files of KiloAlfa and proceed to decrypt the files. Once decrypted, RomeoDelta will generate a ZIP file with the contents of the keylog files, encrypt the file using RC4, and transmit the file to the C2 server.

Commands from the C2 server come in the form of command files. RomeoDelta downloads data from the C2 server and saves the data to a temporary file beginning with **FXSAPI** in the **%TEMP%** directory. The structure of the command file is identical to the structure of the DeltaCharlie command files, where bytes 4 through 132 contain a RSA encrypted data blob that once decrypted with a public key (Figure 6-1) reveals the MD5 value of the data that follows the 132<sup>nd</sup> byte. The RSA functionality that RomeoDelta employs is the exact same code found in SierraJuliett-MikeOne. Only the public keys differ between the two families.

```
23805C8DB86385D315F2D4E43072EF0B432333834A2058DD5AFD0637D3681D5B79463AB2BA15ECE38BEB680B
64C884F15AC2D8FDF4CB463634B4EB2725398C7AC51DA787526C5FDA235DA913C0C7E04B1A405BFEEA4F63568
E5B25B3D2636F4D50996BD1D2390EFDFFEF636BB901D9C1C7128033CF0FE951AEBD303F3967527FD6
```

Figure 6-1: RomeoDelta’s Public Key

The data that follows the encrypted MD5 value is encrypted with RC4 using the password **BAISEO%\$2fas9vQsfvx%\$**. After decryption, the decrypted data is loaded into a buffer and the command file is deleted. An acknowledgment of the decryption is sent to the C2 server by RomeoDelta before the command within the command file is handled.

RomeoDelta supports a limited set of commands as identified in Table 6-1.

COMMAND ID	DESCRIPTION
0x58692AB8	Uploads victim’s system information to C2 server
0x58692AB9	Open a listening port on the victim’s machine, sends the port number to the C2 server, waits up to 10 seconds for the C2 server to successfully connect to the port. C2 server responds with the status of the connect-back test.
0x58692ABA	Writes the embedded file within the command file to the victim’s hard drive (in the <b>%TEMP%</b> directory with a name starting with <b>gbl_</b> ) and executes the file via a call to <b>CreateProcess</b> .
0x58692ABB	Opens a listening port (using the same technique as command 0x58692AB9) and performs a connect-back test. If the resulting response code from the C2 server is <b>0x611</b> , either writes the file contained within the original command file or downloads a file from the URL specified in the command file. After the file is on the victim’s hard drive (in the <b>%TEMP%</b> directory with a name starting with <b>gbl_</b> ), executes the file via a call to <b>CreateProcess</b> .
0x58692ABC	Updates the 3 <sup>rd</sup> DWORD in the configuration with a value specified in the command file.
0x58692ABD	Updates the C2 server addresses within the configuration file via the values specified in the command file.
0x58692ABE	Archives (in ZIP format) the specified directory, encrypts the ZIP file (saved as <b>MSI{random}.LOG</b> ), and uploads the file to the C2 server. This command is not available in earlier versions of RomeoDelta.



COMMAND ID	DESCRIPTION
0x58692ABF	<p>For each user on the victim's machine, copies the following directories and files into a ZIP archives with filenames beginning with <b>A000</b>.</p> <ul style="list-style-type: none"> <li>▪ User's "Recent" folder</li> <li>▪ User's "Favorites" folder</li> <li>▪ User's Vandyke Software's SecureCRT <b>config</b> folder<sup>2</sup></li> <li>▪ User's FileZilla folder<sup>3</sup></li> <li>▪ User's ESTsoft's ALFTP folder<sup>4</sup></li> <li>▪ User's NetSarang folder<sup>5</sup></li> <li>▪ User's mRemoteNG folder<sup>6</sup></li> <li>▪ User's Default.rdp file</li> <li>▪ User's Desktop.lst file</li> <li>▪ User's Documents.lst file</li> </ul> <p>Generate a directory listing for each user's Desktop and My Documents directories in separate text files with names beginning with <b>A000</b>. The following commands are executed and their output saved to the files beginning with <b>A000</b>:</p> <ul style="list-style-type: none"> <li>• <b>ipconfig -all</b></li> <li>• <b>net view</b></li> <li>• <b>net view /domain</b></li> <li>• <b>netstat -ano</b></li> <li>• <b>tasklist /svc</b></li> <li>• <b>query user</b></li> </ul> <p>Once all of the data is archived into a ZIP file, encrypts the archive using RC4 and transmit to the C2 server.</p> <p>The command is also capable of looking for common Windows user profile directories in English, Italian, Spanish and Portuguese.</p>
0x58692AC0	<p>For the service specified in the command file, renames the existing service DLL to <b>wmdrmsdk.dat</b>, and writes the file embedded within the command file to the victim's hard drive using the name of the service's DLL. This command is not available in earlier versions of RomeoDelta.</p>

Table 6-1: RomeoDelta's Supported Command

<sup>2</sup> VanDyke Software. "VanDyke Software Products" <https://www.vandyke.com/products/index.html> Accessed 5 February 2016

<sup>3</sup> FileZilla. "FileZilla - The Free FTP Solution". <https://filezilla-project.org/> Accessed 5 February 2016

<sup>4</sup> ESTsoft. "ALTools" <http://www.estsoft.com/altools> Accessed 5 February 2016

<sup>5</sup> Netsarang Computer. <https://www.netsarang.com/> Accessed 5 February 2016

<sup>6</sup> mRemoteNG. "mRemoteNG" <http://www.mremoteng.org/> Accessed 5 February 2016

Command 0x58692AB9, a NAT check, attempts to bind to the first available port from a list of ordered port numbers (Figure 6-2). As part of the test, RomeoDelta opens a hole in the victim's host-based firewall by issuing the command seen in Figure 6-3. After the C2 server connects, the listening port is shutdown and the firewall rule is removed.

```
443, 110, 53, 80, 995, 25, 8080, 1816, 465, 1521, 3306, 1433, 3128, 109, 161, 444, 1080, 520,
700, 1293, 1337, 2710, 3100, 3305, 3689, 11371, 1024, 1035, 1900, 2004, 2053, 1098, 3098,
4343, 3024, 1058
```

Figure 6-2: RomeoDelta's Ordered Port List

```
cmd.exe /c netsh advfirewall firewall add rule name="Windows Media Player Network
Sharing" dir=in action=allow Protocol=TCP localport={PORT NUMBER}
```

Figure 6-3: RomeoDelta's Firewall Modification Command

After each command is received and executed, RomeoDelta sleeps for a period of time specified in the configuration file. The interval between commands are expressed in minutes, not seconds, indicating that the attackers using RomeoDelta are not likely issuing real-time commands, but rather strategic, predefined commands.

If a C2 server address fails to result in a connection between RomeoDelta and the C2 server, RomeoDelta sleeps for one hour before attempting the next C2 server address in the configuration. Again, this indicates that RomeoDelta is a tool that is not for rapid, or even real-time, interaction, but rather slower, stealthier command interchange.

Later versions of RomeoDelta introduce commands that pertain to long-term recon collection (specifically 0x58692ABF and 0x58692AC0). Coupling the slow command execution and C2 server interactions with the long-term collection functions, RomeoDelta is designed to be a data collection tool for longer running operations instead of rapid fire remote administration.

## 7. [RAT] RomeoEcho

.....

A stark departure from the design pattern found in Romeo-CoreOne-base families, RomeoEcho is a RAT that uses a more interactive command shell format for command identification. The RomeoEcho samples begin within the **DllMain** function by attempting to create a mutex named “**\_\_mutex\_set\_cookie\_\_**”. If RomeoEcho is unable to create the mutex due to the error condition **ERROR\_ALREADY\_EXISTS**, indicating that the named mutex already exists, the DLL framework will not activate the RomeoEcho operational core.

Whether activated by **DllMain** or **WinMain** (for the standalone executable sample), the operational core of RomeoEcho is the same. Upon activation, the core binds to a listening port on the victim’s computer. The port number varies per samples with known port numbers being 1984, 4558, 2550, 3080, and 3579.

When an incoming connection occurs, the first task the core performs is a handshake to establish the authenticity of the client making the connection. The handshake protocol consists of the core sending a constant (**0x18D1F71F**) to the client in plaintext as a 4-byte (DWORD) data buffer. The client must respond with another constant (**0xC46FF197**) over the same socket, also in plaintext, otherwise RomeoEcho terminates the connection with the client. The client then transmits another 4-byte value to the core. This 4-byte value is the communications key for any further communication between the client and the RomeoEcho instance.

Communication between the client and RomeoEcho, after the handshake, is encrypted using a bitwise roll, an XOR, and either an addition or a subtract operation (depending on the direction of the encryption). The communications key dictates the particulars of the transformation, as Figure 7-1 illustrates. The core of the transformation is an XOR/SUB or XOR/ADD in the same vein as the DNSCALC-style encoding.

```
void DecryptBuffer(char *pvData, int dwLength)
{
    for (int j = 0; j < dwLength; ++j )
    {
        pvData[j] = (cryptoKey[1] ^ __ROL__(pvData[j], cryptoKey[2])) - cryptoKey[0];
    }
}

void EncryptBuffer(char *pvData, int dwLength)
{
    for ( int j = 0; j < dwLength; ++j )
    {
        pvData[j] = __ROR1__( cryptoKey[1] ^ (cryptoKey[0] + pvData[j]), cryptoKey[2]);
    }
}
```

Figure 7-1: RomeoEcho’s Communication Decryption/Encryption Functions

With the communications key established and the handshake complete, RomeoEcho’s core activates the communications loop function. RomeoEcho is single-threaded and does not spawn a new thread for incoming connections. As a result, only one client can access a RomeoEcho-infected node at a time.

The communications loop begins each loop by receiving a 4-byte (DWORD) value from the client specifying the size of the next packet that the client is to send. The client then sends the specified number of bytes to the RomeoEcho instance. Effectively, RomeoEcho is using a datagram format found in many of the Lazarus Group's families but at a much more simplified level. The datagram received by RomeoEcho specifies the particular command to execute as an ASCII string with the command and arguments delimited by a pipe (|) characters. At its most basic, the format for incoming commands is:

```
{command name}|{optional arguments 1}|{optional argument 2}|{and so on}
```

RomeoEcho supports seven commands, identified in Table 7-1.

COMMAND IDENTIFIER	ARGUMENT COUNT	DESCRIPTION
<code>_del</code>	1	Securely deletes the file specified
<code>_dir</code>	1	Returns the list of files and their attributes (flags, size, timestamps) in the specified directory.
<code>_exe</code>	1	Executes the specified command line via <code>WinExec</code>
<code>_get</code>	2	Transfers the specified file (first argument) from the victim's machine to the client. The second argument is unused.
<code>_got</code>	2	Transfers the specified file (first argument) from the victim's machine to the client and then securely deletes the file on the victim's computer. The second argument is unused.
<code>_quit</code>	0	Terminates the session between the client and RomeoEcho.
<code>_put</code>	2	Transfers a file from the client to the victim's machine and saves the file at the specified location (second argument). The first argument is unused.

Table 7-1: RomeoEcho's Supported Commands

The Windows operating system provides a variety of APIs for interfacing with the file system. The two most notable APIs are the Windows-native APIs (such as `CreateFile`, `ReadFile`, and `WriteFile`) and the POSIX APIs (such as `_open`, `_read`, and `_write`). RomeoEcho uses the POSIX API when dealing with a victim's file system for the `_get`, `_got`, and `_put` commands. However, the `_del` command uses the Windows API primarily for the destruction of files. It is somewhat unusual for a developer to switch to a different API so abruptly, possibly suggesting that more than one developer was responsible for the RomeoEcho source code.

The `_del` function deviates from the normal secure delete functionality found in many of the Lazarus Group's families. The `_del` command begins by determining the size of the file to destroy and then generating a heap of equal size. The heap is then zeroed and written to the target file. The file is read into memory, in its entirety, and compared to the zeroed buffer to ensure the write was successful. The heap buffer is set to all `0xFF` values and written to the target, and the write is verified. The buffer is filled with cryptographically strong random data by calling `CryptGenRandom` from the Microsoft cryptographic API and written to the target file, and the write is, again, verified. Finally, the heap buffer is again zeroed and written to the disk with one final verification that the write was successful. The deletion of the file consists of calling the POSIX function `_chsize` to set the file's size to 0 bytes, then calling `DeviceIoControl` with `FSCTL_DELETE_OBJECT_ID` to attempt to unlink the file from the file system, followed by calls to `MoveFileExA` and `DeleteFileA`. The secure deletion functionality of RomeoEcho is thorough and exhibits coding styles contrary to the level of sophistication found in other parts of the RomeoEcho code base.

## 8. [RAT] RomeoFoxtrot

Operating as a server mode RAT, RomeoFoxtrot uses a simple handshake to establish a connection and variant-dependent encryption to transfer data making the malware significantly less sophisticated from a network perspective than other members of the Romeo class. Despite the lack of network sophistication, RomeoFoxtrot provides a large number of commands to handle aspects of file management, process management, network proxying, and victim computer information enumeration.

There are two known variants of RomeoFoxtrot: RomeoFoxtrot-One and RomeoFoxtrot-Two. The RomeoFoxtrot family has been observed as the payload of the IndiaCharlie variants, with IndiaCharlie-One observed dropping RomeoFoxtrot-One and IndiaCharlie-Two observed dropping RomeoFoxtrot-Two. Functionally, the two variants are very similar with only two distinctions. The primary distinction is the inclusion of a configuration file for RomeoFoxtrot-Two that specifies the listening port, while RomeoFoxtrot-One uses a hardcoded value. The second is a renumbering of command identifiers. Given the similarities, the remainder of this section will simply refer to them equally as RomeoFoxtrot unless a particular detail is specific to one variant over the other.

Upon activation, RomeoFoxtrot generates a listening port on the victim's system and spawns a new handler thread for all incoming connection, thereby allowing RomeoFoxtrot to handle multiple clients at once. When a new client connects, a handshake procedure begins with the client sending a NULL-terminated string of **POST HTTP REQUEST?** and RomeoFoxtrot responding with **RESPONSE 200 OK!!!**. If the handshake fails, the connection and the handler thread terminate.

After the handshake, RomeoFoxtrot enters a continual loop: waiting for a command from the client, executing the command, and returning to a waiting state for more commands. The communication between the client and RomeoFoxtrot-Two is encrypted with RC4. The first 128 bytes of a datagram between RomeoFoxtrot-Two and the client contain a 128-byte RC4 key if and only if the size of the datagram is larger than 128 bytes. Conversely, all communication between RomeoFoxtrot-One and the client is either in plaintext (for older versions of the variant) or encoded with a simply XOR 0x81.

ROMEOfOXTROT-ONE COMMAND ID	ROMEOfOXTROT-TWO COMMAND ID	COMMAND DESCRIPTION
0x2010	0x2000A	Echoes the request packet back to the client with the size field set to 512 bytes.
0x2020	0x30002	Uploads the specified file from the victim's machine to the client.
	0x3000B	Uploads the specified directory's files from the victim's machine to the client.
0x2030	0x30001	Writes the specified file at the specified location from the client to the victim's machine
0x2040	0x30005	Recursively deletes directory and its descendants
	0x3000A	Recursively secure deletes directory and its descendants
0x2050	0x30006	Move or renames a file
0x2060	0x30007	Mimics the timestamp of one file unto another file as specified by the client
0x2080	0x30008	Creates the specified directory on the victim's machine
	0x30009	Port knock on the specified endpoint.
0x2090		Executes the specified command while piping the output of the command to a file on the victim's machine. The file containing the output is uploaded to the client and deleted from the victim's machine
0x20C0, 0x20E0, 0x2180	0x20006, 0x2000B, 0x30004	Enumerates the processes on the victim's machine
0x20F0	0x20005	Returns information about the network interface cards (NICs) of the victim's machine (up to 16 NICs) as an array of <b>IP_ADAPTER_INFO</b> structures.
0x2100	0x20002	Returns the victim's computer name to the client
0x2110		Returns the output of the API function <b>GetLocaleInfoA</b> to the client.
	0x20003	Returns the output of the API function <b>GetLocaleInfoW</b> to the client.
0x2120	0x20007	Returns the output of the API function <b>GetVersionExA</b> to the client.
0x2130		Sends the DWORD <b>0x1000000</b> to the client.
	0x20008	Sends the DWORD <b>0x2000001</b> to the client.
0x2150 0x2170	0x40001	Activates an interactive command shell on the victim's computer with input and output piped to the client's endpoint.
0x2160	0x20009	Attempts to bind to the following ports on the victim's machine in order to determine if the port is active or not:  3389, 443, 80, 53, 110, 8080, 1433, , 3306, 1521  A list of the ports that are in use is sent to the client or, if none of the desired ports are in use, returns " <b>There aren't open ports.</b> "
0x2310	0x20004	For each attached drive on the victim's system, returns the volume's name and its free space to the client.
0x2320	0x2000C	Returns a file listing for the specified directory
	0x2000D	Changes the listening port of RomeoDelta and restarts the malware.
0x2360	0x40002	Client supplies a list of endpoint that RomeoFoxtrot uses to attempt a connection using the client protocol for establishing a connection to a RomeoFoxtrot node. This effectively allows for the establishment of a proxy through one RomeoFoxtrot node to another.

Table 8-1: RomeoFoxtrot Supported Commands and Their Identifiers

## 9. [RAT] RomeoGolf

Observed as the payload of IndiaEcho and loaded by LimaBravo, RomeoGolf is a RAT with some similarity to the families derived from Romeo-CoreOne (see Section 2) and as RomeoFoxtrot-Two (see Section 8) in terms of the commands supported. Structurally, however, RomeoGolf is a family unto its own.

While capable of operating as a service DLL, the core functionality of RomeoGolf is activated out of the **DllMain**, not the **ServiceMain**, export. Upon activation, RomeoGolf spawns a new thread that attempts to open a handle to the RomeoGolf binary and then calls the **LockFile** API function with that handle in order to prevent disruption of the RomeoGolf executable. The thread then goes into an infinite sleep.

RomeoGolf is heavily object oriented and written in C++. With the exception of the file locking thread just mentioned, all of RomeoGolf's functionality is contained within a single class. Within the class are two additional classes: a class for network communication to and from the C2 server and a class containing the list of queued commands.

RomeoGolf's class contains a primary member function responsible for its operation (a function Novetta identifies as the **Execute** function). When the **Execute** function is called, it enters an infinite loop where the following tasks occur:

1. The configuration file (**crkdf32.inf**) is loaded into memory. Failure to load the configuration results in a one-minute delay before another attempt is made. RomeoGolf does not continue until the configuration file is loaded successfully, and instead continues the cycle of attempted loads and sleeps.
2. The unique identifier for the victim's machine is checked within the configuration and, if found to be **0**, is generated by using 4 calls of the **rand()** function to generate the necessary 64-bits of the identifier.
3. A flag within the configuration is checked to determine if all the files and drives of the victim's machine have been enumerated, and, if not, a new thread is generated (along with a signaling event) to perform the task.
4. A flag within the configuration is checked to determine if the victim's Terminal Service's has been enumerated for active users, and, if not, a new thread is generated (along with a signaling event) to perform the task. The enumeration lasts until at least one user logs in through either Terminal Services or the local console.
5. The victim's system information is gathered including the machine's name, the victim's username, the operating system, location information, the processor type, available memory size, and the IP addresses associated with the machine.
6. A C2 server address is randomly selected from the configuration and an attempt to connect to the C2 server is made. Up to 10 attempts to contact a C2 server are made before the operation fails. Between each attempt to contact a C2 server, RomeoGolf sleeps for 1 minute.
7. If a successful connection is made to a C2 server, a new thread is generated that sends a heartbeat signal to the C2 server every 1 minute. If after 10 attempts no connections are successful, RomeoGolf sleeps for a preconfigured period of time (measured in minutes) before looping back to Task #1.
8. RomeoGolf receives commands from the C2 server and executes the appropriate handler for each command.
9. After the command processing concludes, the connection to the C2 server is terminated, and any queued commands are purged.
10. The last contact time within the configuration is updated, indicating the last time the malware contacted a C2 server, and the configuration is saved to disk.
11. Loops back to Task #1.

Task #3 involves enumerating all of the files on the victim's machine and retaining this information in the folder **%TEMP%\Z802056**. The enumeration task has similarities to the same task in RomeoHotel (see Section 10): not only are the files recursively enumerated, but also, if at least 5 GB of hard drive space is available on the victim's system hard drive (typically **C:**), copies of enumerated files on removable devices (excluding CD/DVD drives) and file shares are retained.

RomeoGolf utilizes fake TLS to encode the communication between itself and its C2 server. Unlike other Romeos, RomeoGolf does not use an additional handshake protocols over the fake TLS handshake. The C2 server can send a burst of commands to RomeoGolf. RomeoGolf reads command datagrams from the C2 server and places each command in a linked-list of queued commands. The linked-list supports up to 10 queued commands. After the C2 server completes the transfer of commands, RomeoGolf begins executing each command in order.

The command table (Table 9-1) has a striking similarity in both functionality and order to that of the R-C1-derived families and RomeoFoxtrot-Two. Each command returns its result, if any, to the C2 server and uses a special command ID value to indicate the success or failure of the command in the same way as Romeo-CoreOne-derived families.

COMMAND ID	DESCRIPTION
0x10001000	Returns information about the attached drives on the victim's machine
0x10001001	Enumerates the specific directory.
0x10001002	Copies file (locally)
0x10001003	Deletes the specified file
0x10001004	Securely deletes the specified file
0x10001005	Downloads a file from the C2 server.
0x10001006	Uploads file
0x10001007	Executes the supplied command line
0x10001008	Mimics the file timestamp of the specified file unto another specified file.
0x10001009	(missing)
0x1000100A	Port knock.
0x1000100B	Executes a command, pipes the output of the command to a file and uploads the file to the C2 server.
0x1000100C	Disconnects from C2 server
0x1000100D	Enumerates running processes.
0x1000100E	Terminates a process.
0x1000100F	(missing)
0x10001010	Gets the statistics (directory count, total size, timestamps) of the specified directory.
0x10001011	Uploads the current configuration to the C2 server.
0x10001012	Updates the current configuration from the configuration received.
0x10001013	ZIPs the contents of the specified directory (and its descendent directories) and uploads the file to the C2 server.
0x10001014	Changes the startup delay time to match that of the current sleep delay between C2 connections.
0x10001015	(missing)
0x10001016	[STATUS CODE] Command successful
0x10001017	[STATUS CODE] Command unsuccessful

Table 9-1: RomeoGolf's Supported Commands and Their Identifiers



# 10. [RAT] RomeoHotel

While capable of running as a loaded DLL, observed RomeoHotel samples rely on LimaCharlie to load. RomeoHotel is a client-mode RAT that utilizes R-C1 (Section 2) for its core command processing code. RomeoHotel is notable for having both 32-bit and 64-bit samples, though there are functionally no differences between the variants other than their supported architectures.

RomeoHotel acts as a hybrid of RomeoAlfa-Two (Section 3) and RomeoCharlie-Two (Section 5). Structurally, RomeoHotel is nearly identical to RomeoCharlie-Two: RomeoHotel retains its configuration in the registry (specifically, the registry entry `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Security\zc62a465-fff-87cc-47cdcdefa`), uses a startup delay, and operates as a thread spawned out of `DllMain`.

Functionally, RomeoHotel is nearly identical to RomeoAlfa: RomeoHotel supports the same commands within the R-C1 portion of itself, has an identical configuration data structure in both size and field meaning to that of RomeoAlfa-Two, and uses fake TLS for communication. RomeoHotel supports the same command set found in RomeoAlfa-Two, but supports all commands fully, while RomeoAlfa-Two does not fully support all of its commands. For example, RomeoHotel supports the `RunAs` command fully while the RomeoAlfa variants will accept the command but will perform not action as a result of the instruction from the C2 server.

The numbering of commands between RomeoAlfa-One and RomeoHotel are exactly one-to-one with a 0x1000 offset, e.g. `Move File` in RomeoAlfa-One is 0x873B and 0x973B in RomeoHotel. It is unclear if the offset between the two is indicative of a generational shift, a coincidence of versioning, or a conscious decision by the developer(s). But the fact remains that given the similarities between the RomeoAlfa and RomeoHotel families, the one-to-one (with offset) of commands is eye catching.

Table 10-1 lists RomeoHotel's supported commands and their command identifiers.

COMMAND IDENTIFIER	COMMAND
0x973B	<i>Move File</i>
0x973C	<i>Directory Statistics</i>
0x973D	<i>Enumerate Drives</i>
0x973E	<i>Enumerate Directory</i>
0x973F	<i>Write File</i>
0x9740	<i>Read File</i>
0x9741	<i>Upload Directory as Archive (Defunct)</i>
0x9743	<i>Create Process</i>
0x9744	<i>Secure Delete</i>
0x9745	<i>Mimic Timestamp</i>
0x9746	<i>Execute Shell Command with Output Upload</i>
0x9747	<i>Enumerate Processes</i>

COMMAND IDENTIFIER	COMMAND
0x9748	<i>Terminate Process</i>
0x9749	<i>System Information</i>
0x974A	<i>Change Directory</i>
0x974B	<i>Port Knock</i>
0x974D	<i>Send Status Value</i>
0x974F	(RESPONSE CODE) <i>Relay Status: Failure</i>
0x9750	(RESPONSE CODE) <i>Command Status: Success</i>
0x9751	(RESPONSE CODE) <i>Command Status: Failure</i>
0x9752	<i>Disconnect</i>
0x9753	<i>Get Config</i>
0x9754	<i>Set Config</i>
0x9755	<i>RunAs</i>
0x9756	<i>NOP</i>
0x9757	<i>Suicide</i>

Table 10-1: RomeoHotel's Supported Commands and Their Identifiers

Despite the similarities between RomeoHotel and other Romeo-CoreOne-based families, there are key differences that make RomeoHotel unique in the Lazarus Group's collective. After loading the configuration from the registry and prior to the startup sleep delay, the malware performs three tasks specific to RomeoHotel:

1. Stop a service specified by the configuration and unload its DLL from memory by calling **FreeLibrary**.
2. Run two commands specified by the configuration via **CreateProcess** calls.
3. Spawn a new thread that loads two DLLs specified by the configuration after a 20 second sleep delay.

A viable configuration was not observed in any identified samples, making it difficult to determine the intention of the three tasks.

The sleep delay, hardcoded in RomeoCharlie-Two, is a configurable item specified by the configuration data structure. RomeoHotel enforces a maximum startup delay of 180 minutes (3 hours) by reducing any startup delay value to 180 minutes. RomeoHotel also introduces a delay between successive C2 server connection attempts. Prior to attempting a connection to a C2 server, RomeoHotel sleeps for 10 seconds.

RomeoAlfa, RomeoBravo (see Section 4), RomeoCharlie, and RomeoNovember (see Section 12) each call a function that introduces a sleep delay between endpoint connections. Evidence suggests that each of the families is using a neutered version of the function found in RomeoHotel immediately after exiting the **MessageThread** function (Figure 10-1).

```

void __cdecl PostConnectionActivities(int dwMaxDrives)
{
    wchar_t wszRootPath[6] = L"::\\";
    DWORD dwPrevDriveMask = GetLogicalDrives();
    WTS_SESSION_INFO *pSessionInfos = 0;
    HMODULE hWtsapi32 = LoadLibraryA("Wtsapi32.dll");
    pfnWTSEnumerateSessionsA = GetProcAddress(hWtsapi32,
                                                "WTSEnumerateSessionsA");

    if ( pfnWTSEnumerateSessionsA )
        pfnWTSEnumerateSessionsA(0, 0, 1, &pSessionInfos, &dwSessionCnt);
    for ( i = 0; i < dwMaxDrives; ++i )
    {
        DWORD dwDrivesMask = GetLogicalDrives();
        if ( dwDrivesMask > dwPrevDriveMask )
        {
            DWORD dwDriveMaskDiff = dwDrivesMask - dwPrevDriveMask;
            g_connectionState = 3;
            DWORD dwDriveIndex = 2;
            while ( !((dwDriveMaskDiff >> dwDriveIndex) & 1) )
            {
                if ( ++dwDriveIndex >= 26 )
                    goto LABEL_10;
            }
            wszRootPath[0] = dwDriveIndex + 'A';
        LABEL_10:
            wprintf(L"USB Volumn = %s\r\n", wszRootPath);
            dwPostConnectionActivityMode = config.dwPostConnectionActivityMode;
            if ( config.dwPostConnectionActivityMode == 3
                || config.dwPostConnectionActivityMode == 5
                || config.dwPostConnectionActivityMode == 6 )
            {
                CreateThread(0, 0, DuplicateDirectoriesThread, wszRootPath, 0, 0);
                Sleep(1000);
                dwPostConnectionActivityMode = config.dwPostConnectionActivityMode;
            }
            if ( dwPostConnectionActivityMode == 2
                || dwPostConnectionActivityMode == 4
                || dwPostConnectionActivityMode == 6 )
            {
                GetDrivesFileListAndSaveToFile(wszRootPath);
                dwPostConnectionActivityMode = config.dwPostConnectionActivityMode;
            }
            if ( dwPostConnectionActivityMode == 1
                || dwPostConnectionActivityMode == 4
                || dwPostConnectionActivityMode == 5 )
                break;
        }
        dwPrevDriveMask = dwDrivesMask;
        if ( pfnWTSEnumerateSessionsA )
        {
            pfnWTSEnumerateSessionsA(0, 0, 1, &pSessionInfos, &pdwActiveSessions);
            if ( pdwActiveSessions > dwSessionCnt
                && config.fAbortPostConnectionActivityOnNewTSSessions == 1 )
            {
                g_connectionState = 4;
                return;
            }
            dwSessionCnt = pdwActiveSessions;
        }
        Sleep(60000);
    }
}

```

Figure 10-1: RomeoHotel's Post MessageThread/Disconnect Activity Function

The function in Figure 10-1 enumerates the logical drives on the victim's system and potentially performs up to three tasks based on the value set within the configuration data structure for each logical drive.

dwPostConnection ActivityModeValue	TASK PERFORMED
3, 5, or 6	If at least 5 GB of space is available on the system drive (e.g. C:), copy the contents of the logical drive to %TEMP%\PAS02034\{serial number of logical drive}\temp.
2, 4, or 6	Process a full list of all files on the logical drive in a text file stored at %TEMP%\PAS02034\{serial number of logical drive}\{serial number of logical drive in hexadecimal}.dat
1, 4, or 5	Stop processing additional logical drives

Table 10-2: Tasks RomeoHotel's Post MessageThread Activity Function May Perform on Each Logical Drive

At first glance, the behavior of the **PostConnectionActivities** function seems unusual, but it is, in actuality, a very clever method of storing the contents of removable drives such as USB drives, CD/DVD drives, and network shares that may accessible on the victim's computer for short and infrequent periods of time. What is curious, however, is why this functionality was excluded from other R-Cr based families.

## 11. [RAT] RomeoMike

.....

A component of the reported Ten Days of Rain attacks, RomeoMike is a RAT with a very limited set of capabilities yet exhibits a great deal of functional and procedural similarity to SierraJuliatt (see Section 17) and DeltaCharlie with regards to the way commands are processed through signed command files. RomeoMike is a service DLL that, after establishing the scaffolding code to appear as a legitimate Windows service, begins by calling **DialogBoxParam**. The inclusion of the **DialogBoxParam** call is unusual due to the fact that the **DialogFunc** callback passed to **DialogBoxParam** simple returns **0** meaning that a modal dialogue, or any dialogue for that matter, will never appear.

RomeoMike performs a dynamic API loading operation for **iphlpapi.dll**, **kernel32.dll**, **ws2\_32.dll**, **psapi.dll**, and **wtsapi32.dll**. AES encryption is used for the obfuscation of the API names. After loading the various API functions, the configuration file located at **%SYSTEMROOT%\faultrep.dat** is loaded into memory. The configuration file contains information related to the unique identifier for the particular instance of RomeoMike, the identifier of the last command file executed (which will be explained shortly), a list of up to 10 C2 servers, and various other management artifacts.

A new thread is generated by RomeoMike to handle the bulk of its functionality in order to free up the service's thread and thus avoid appearing as a hung service. The new thread spawns another thread for the actual functionality of RomeoMike. This new thread contains an infinite loop that performs the following tasks:

1. Determine if the last contact to the C2 was more than 3,6 hours ago, if not wait 20 minutes.
2. Attempt to connect to a C2 server (using the list of C2 servers in sequence). If a connection cannot be made, sleep for 20 minutes before trying the list again until a connection is successful.
3. Send a DWORD (**0x45196327**) to the C2 server, read a DWORD from the C2 server and verify that the value sent and the value received are the same. A failure at this stage results in a C2 retry after 20 minutes.
4. Send an initial WORD (**0x3000**) to the C2 server.
5. Send the (in order) the 64-bit identifier of the infection, the first DWORD of the configuration, and the DWORD of the ID of the last command executed to the C2 server.
6. Receive a series of files from the C2 server, storing each file in the **%SYSTEMROOT%\111** directory with a random file timestamp. For each file successfully received, send a WORD (**0x0001**) to the C2 server. Repeat until the server responds with WORD **0xFFFF**.
7. Sleep for one second
8. Disconnect from the C2 server.
9. For each file in the **%SYSTEMROOT%\111** directory, process the command file and execute the appropriate command handler.
10. Update the last contact time and save the configuration to disk.
11. A sleep of 20 minutes occurs.
12. Repeat at step 1.

Step #9 requires RomeoMike to load each file within the **%SYSTEMROOT%\111** directory into memory in chunks. The first chunk read consists of 4 bytes. The first four bytes indicate the number of commands that the command file contains. Following the first 4 bytes, a 128-byte chunk is read into memory. Encrypted with a private RSA key, the 128-byte chunk contains another header (the command header) which RomeoMike must decrypt with a public key (Figure 11-1).

```
5BEFBF9CE323994CA723C71EF91F3E6E1A233D56B8DA897B5E01D3AE3BB02E6552D66F9E7F9993DC35048811E7
651E26CF16C5151E742093C30E865E4F2056738374A830FE47E14E4655AE58FA1B1C65D03DD61B19ED8294948D4
87C75CC146D73A346-6DC190B313845FB2F303,253302E5E43273504B32F6B6B421EB66E249F1
```

Figure 11-1: RomeoMike's Public Key

Each command header is 24 bytes in size and contains the structure identified in Table 11-1. The command header defines the type of command, the size of the data that follows the command header, and the MD5 hash of the data (prior to decryption). The data that follows the command header is encrypted with RC4 requiring RomeoMike to decrypt the data prior to use.

OFFSET	SIZE	DESCRIPTION
0	2 Bytes (WORD)	Command Type
2	16 Bytes	MD5 hash of the data the follows the command header
18	2 Bytes	Unused
20	4 Bytes (DWORD)	Size of the data the follows the command header

Table 11-1: RomeoMike's Command Header Structure

It is the responsibility of each command handler to verify the MD5 hash of the data within its section of the command. If the MD5 hash fails, the command is considered invalid. RomeoMike supports three command types, all of which relate to the execution of an additional binary. Table 11-2 lists the commands RomeoMike supports.

COMMAND TYPE IDENTIFIER	DESCRIPTION
0x1001	Writes file to disk and executes the file via <b>CreateProcess</b> .
0x1002	Loads an unknown module into memory and calls one of its exports, then write the file embedded in the command file to disk and executes the file via <b>CreateProcess</b> . As indicated in McAfee's report, <sup>7</sup> the name of the module and export are unknown as the decryption of the AES encrypted string results in unusable output. What is known about the module's name is that it is up to 11 characters long plus a NULL-byte and what is known about the export's name is that it is up to 19 characters long plus a NULL-byte.
0x1003	Executes the supplied command line string via <b>CreateProcess</b> .

Table 11-2: RomeoMike's Support Commands

For the first command header within a command file, an additional field exists. Prior to the first field in the command header, a DWORD exists that provides the identification number of the command file. In much the same way SierraJuliett-MikeOne and DeltaCharlie do not execute commands that have a lower command number than the last command they execute, RomeoMike ignores any command file with a lower command identifier number.

<sup>7</sup> 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>

## 12. [RAT] RomeoNovember

.....

RomeoNovember is a client-mode RAT that has a strong structural and familial relationship to both RomeoAlfa (see Section 3) and RomeoBravo (see Section 4). Romeo-CoreOne-based, structurally RomeoNovember is most like RomeoAlfa, as it operates as a standalone executable, constructs its configuration data structure from hardcoded values, and leverages the same scaffolding for supporting R-C1.

Functionally, however, RomeoNovember is closer to RomeoBravo than RomeoAlfa. Like RomeoBravo, RomeoNovember uses DNSCALC-style encoding to obfuscate network communication instead of RomeoAlfa's reliance on fake TLS. The similarity to RomeoBravo also extends to the use of the same base command number (0x523B) and channel ID (0x3456). The commands within R-C1 supported by RomeoNovember are the nearly the same as those supported by RomeoBravo, to the extent that RomeoNovember and RomeoBravo both implement the *Secure Delete* command with the same code. Only the *Upload Directory as Archive* command is missing from RomeoNovember.

RomeoNovember's hybrid nature may indicate an active development period for the developer(s). Known samples of RomeoNovember exist between March 13, 2015 and April 13, 2015, a period bookended by active RomeoAlfa and RomeoBravo development. The first observed RomeoNovember sample (SHA256: 6dab43a75647c20ac46c6f1cc65607dd4d7bb104e234b4f74f301e772e36ab9b) has a compile time that is 1 hour, 13 minutes after the RomeoAlfa sample (SHA256: f46d277baf0bb8d63805ff51367d34a9cbdd7a0a1394ab384fber1d98c8fc4b8) that marks the beginning of the RomeoNovember life span.

## 13. [RAT] RomeoWhiskey (Winsec)

In terms of sophistication and functionality, RomeoWhiskey is a mid-tier RAT. At its core, RomeoWhiskey provides the basic functionality one would expect in a RAT: file transfer commands, program execution, basic intelligence gathering, etcetera. Observed as early as May 2011, RomeoWhiskey, also known as Winsec, is one of the older family members used by the Lazarus Group, and, over the course of its lifetime, it has undergone at least one major revision. The first variant of RomeoWhiskey, RomeoWhiskey-One, has been observed with compile dates from May 2011 to late January/early February 2012, while RomeoWhiskey-Two, the second variant, was compiled from late February 2012 until at least March 2014.

RomeoWhiskey uses numerical constants to identify specific commands, reflective of the way RATs like the Romeo-CoreOne-based families (see Section 2) identify commands by unique numerical constants. There are two sets of constants for identifying commands within the RomeoWhiskey samples that do not necessarily align with variant boundaries: *command base 0x7D50* and *command base 0x1E10*. The different command identifier constants are interesting because they could lead to multiple conclusions:

1. There were two developers (or development teams) working on RomeoWhiskey during its lifetime
2. There were two users (or teams of users) of RomeoWhiskey
3. A combination of 1 and 2

In addition, the authentication function uses a set of constants to calculate the appropriate authentication response values. There are three known sets of these cryptographic constants; two of the constants (identified as constants **0xA230** and **0x3230**) align to *command base 0x1E10*, while the remaining cryptographic constant (**0xF3C0**) is only found in samples belonging to the *command base 0x7D50* set. There is no overlap between the cryptographic constants sets and command base sets, meaning that at the very least there are two distinct sets of RomeoWhiskey.

From a development perspective, the *command base 0x7D50* set exists only in RomeoWhiskey-One variants while there is some overlap between the two RomeoWhiskey variants for *command base 0x1E10*. The overlap for *command base 0x1E10* occurs in mid-May/early June 2011 when both base command sets are found within the RomeoWhiskey-One variant sample set. Once the RomeoWhiskey-One variant gives way to RomeoWhiskey-Two variant, the *command base 0x7D50* ceases producing new binaries as there are no observed samples of RomeoWhiskey-Two using the *command base 0x7D50* constants. The supported commands are largely similar, both for samples with *command base 0x7D50* and those with *command base 0x1E10*, but there are cases where one command base supports different functions than the other. The commands that *command base 0x1E10* supports can be found in both variants of RomeoWhiskey.

Both observed variants of RomeoWhiskey operate as service DLL images. Beyond their basic form factor being the same, the RomeoWhiskey-One variant is, for all intents and purposes, the base code for the RomeoWhiskey variants. RomeoWhiskey-Two expands on that core functionality by adding more advanced authentication as well as the concept of channels. Therefore, it is easiest to understand the inner workings of the RomeoWhiskey variants by first understanding the base code that is RomeoWhiskey-One before looking at the enhancements introduced by RomeoWhiskey-Two.



## 13.1 RomeoWhiskey-One (Base Code)

.....

Based on the premise that RomeoWhiskey-One is the base code for the entirety of the RomeoWhiskey samples, this section will refer to RomeoWhiskey-One simply as RomeoWhiskey when addressing information that relates to both observed variants. When a piece of information is specific to only the RomeoWhiskey-One variants, this section will use the full name of RomeoWhiskey-One.

RomeoWhiskey exports two functions in addition to the **DllMain: ServiceMain** and **SecuritySetting**. The **ServiceMain** function provides the scaffolding necessary for a legitimate Windows service before calling **CreateThread** to activate the function housing the core functionality of RomeoWhiskey. The **SecuritySetting** function is simply a wrapper to the function housing the core functionality of RomeoWhiskey. While both **ServiceMain** and **SecuritySetting** ultimately activate the core functionality of RomeoWhiskey, the distinction between the two is that **ServiceMain** causes the core functionality to activate asynchronously to the calling process, while **SecuritySetting** does not return control to the calling process until the core functionality returns control.

The startup sequence for RomeoWhiskey is as follows:

1. Dynamically load API functions
2. Attempt to load the configuration file
3. **bind** to a listening port
4. Save the configuration
5. Open a hole in the Windows firewall
6. **listen** for incoming connections
7. Perform the authentication handshake and spawn a handler thread if successful
8. Repeat steps 6-8 indefinitely.

The functions that dynamically load API functions can be found in other Lazarus Group malware families with the function responsible for loading the **kernel32.dll** API functions being identical to the same function used in IndiaWhiskey. The **kernel32.dll** API loader function also generates the names of the RomeoWhiskey service, its service display name, its service description, and, most importantly for RomeoWhiskey, the name of the configuration file. The reuse of this code clearly ties the developer(s) of the IndiaWhiskey family to the developer(s) of the RomeoWhiskey family.

RomeoWhiskey attempts to load the configuration file from **%SYSTEMROOT%\dayipmr.tbl** (for *command base 0x7D50* samples) or **%SYSTEMROOT%\ansi.nls** (for *command base 0x1E10* samples). The 240-byte file, if found, is read into memory and decoded using DNSCALC-style. If the 101<sup>st</sup> byte is set to 1 then RomeoWhiskey will read the 4-byte value starting at the 217<sup>th</sup> byte as a DWORD containing the listening port for the RomeoWhiskey instance.

The act of establishing the listening port is somewhat more involved than merely calling **bind** on a new socket. Situated within a loop that can iterate 30 times, the process of binding a listening port consists of first attempting to bind to the port specified in the configuration file, if it exists. Failing to bind on the specified port results in up to 4 additional bind attempts using the ports 173, 155, 129, and 192 (in that order). If RomeoWhiskey still hasn't found a suitable port, the remaining iterations calculate a new potential listening port that is 1046 ports higher than the previous attempt (starting at an initial offset of 4). With a suitable listening port found, the configuration file is replaced with the new port value set, and the configuration file's timestamp is set to match that of **kernel32.dll**'s timestamp.

RomeoWhiskey requires that the victim's system not block inbound access to the malware's listening port. In order to modify the victim's Windows firewall and set an exception for inbound connections destined for the RomeoWhiskey port, RomeoWhiskey will add a new registry key under **HKLM\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\GloballyOpenPorts\List**. RomeoWhiskey names the new key {port number}:**TCP** and sets its value to {port number}:**TCP:\*:Enabled:Internet Connection Sharing(ICS)**.

For incoming connections, RomeoWhiskey begins by authenticating the client through a handshaking protocol. The handshake of RomeoWhiskey, Figure 13-1, starts by taking the current tick count of the victim's computer, performing a series of bitwise transformations, and sending the result to the client as a challenge value. The client has up to 30 seconds to respond with a 4-byte response. The response value is then decomposed using another series of bitwise transformations to ensure a particular result. If any of the steps of the handshake fails, the handshake function returns a positive result, and the connection to the client terminates by means of a socket disconnection function call.

```
int AuthenticationHandshake(SOCKET clientSkt)
{
    unsigned __int16 wChallengeSeed = GetTickCount();
    unsigned __int32 dwChallengeResponse = 0;

    if ( SendData(clientSkt,
                 (wChallengeSeed << 16) | (wChallengeSeed >> 52) ^ 0xF3C0),
        4, 1 )
        return 1;

    if ( WaitForSocketRecv(clientSkt, 30) )
        return 1;

    if ( RecvData(clientSkt, dwChallengeResponse, 4, 1) )
        return 1;

    return (((dwChallengeResponse >> 16) & 0xFFFF) >> 52) ^ 0xF3C0 !=
(dwChallengeResponse & 0xFFFF);
}
```

Figure 13-1: RomeoWhiskey-One's Authentication Handshake Function for *Command base 0x7D50* Samples

The **SendData** and **RecvData** functions are common network data transmission functions found in a variety of families within the Lazarus Group's collection. These particular instances use DNSCALC-style encoding to obfuscate the data as it traverses the network.

After the authentication handshake completes, RomeoWhiskey spawns a new thread to handle the incoming requests from the client. Incoming requests consist of the client sending a modified version of a datagram specifying the command identifier value (WORD), followed by the a 2-byte (WORD) value for the size of the payload, and then an optional payload value of up to 260 bytes. RomeoWhiskey uses the command identifier value to locate and execute the appropriate command handler. Table 13-1 lists the commands that RomeoWhiskey-One supports.

COMMAND NUMBER		DESCRIPTION
<i>Command base 0x7D50</i>	<i>Command base 0x1E10</i>	
0x7D50		Returns victim's MAC address and computer name.
	0x1E11	Gets System Information. See <b>VictimInfoPacket</b> definition for details. This reports the malware as version "1.5"
0x7D51	0x1E12	Returns the drive type (from <b>GetDriveTypeA</b> ) for each logical drive on the victim's computer.
0x7D52	0x1E13	Enumerates the files in the specified directory.
0x7D53	0x1E14	Enumerates the processes currently active on the victim's computer. For each process, includes the process's executable name and path, PID, the parent PID, and the timestamp of when the process was initially executed.
0x7D54	0x1E15	Terminates a process by its PID, specified as either an ASCII string or a DWORD.
0x7D55	0x1E18	Executes a new process using a supplied command line string.
0x7D56	0x1E19	Deletes a specified file.
0x7D57	0x1E20	Matches the timestamp of a specified file to the timestamp of <b>kernel32.dll</b> .
0x7D58	0x1E21	Executes the supplied command line via the command shell ( <b>cmd.exe</b> ). The output of <b>STDOUT</b> (and, optionally, <b>STDERR</b> ) are captured to a file in the <b>%TEMP%</b> directory. The file is read up to 60 times and transmitted to the client providing a pseudo-live stream of the output from the executed command.
0x7D59	0x1E16	Disconnects. Returns a success response before terminating the connection to the client.
0x7D5A	0x1E22	Changes the current working directory to the directory specified.
0x7D5B	0x1E23	Uploads the contents of the specified file to the client.
0x7D5C	0x1E24	Downloads a file from the client and saves the file at the location and name specified.
0x7D5D	0x1E25	Returns the number of used and free bytes on the specified logical drive, as well as the drive's name, serial number, and file system type.
0x7D5E	0x1E26	Returns the creation, last accessed, and last write timestamps of the specified file along with the file size.
0x7D5F		Returns the <b>OSVERSIONINFOA</b> data structure for the victim's computer.
0x7D60	0x1E17	Establishes a relay or proxy between the RomeoWhiskey-One instance and a specified endpoint. The relay uses the same authentication handshake function to authenticate the endpoint prior to activating the relay threads.
0x7D61	0x1E27	[Response code] Success
0x7D62	0x1E28	[Response code] Failure
0x7D63	0x1E29	NOP. Returns invalid command status to the client
0x7D64	0x1E30	[Response code] Invalid command requested.

Table 13-1: RomeoWhiskey-One's Support Commands and their Command Identifier Values

Much the same way Romeo-CoreOne uses specific numerical values to indicate success or failure of an operation, RomeoWhiskey uses **0x76D1** (for *command base 0x7D50*) or **0x1E27** (for *command base 0x1E10*) to indicate a successful operation and **0x7D62** (for *command base 0x7D50*) or **0x1E28** (for *command base 0x1E10*) for a failed operation.

Largely, the commands supported by *command base 0x7D50* and *command base 0x1E10* are identical, with the exception of command types 0x7D50, 0x1E11 and 0x7D5F. The *command base 0x1E10* essentially combines commands 0x7D50 and 0x7D5F, from *command base 0x7D50*, into the command identified by 0x1E11. Command 0x1E11 generates a large data structure (Table 13-2) with identifiable information about the victim's computer.

OFFSET	SIZE	DESCRIPTION
0	32 Bytes	Computer name
32	128 Bytes	Processor name (from registry key <b>HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0\ProcessorNameString</b> )
160	156 Bytes (OSVERSIONINFOEXA structure)	OS version information from <b>GetVersionExA</b>
316	30 Bytes	First character to <b>r</b> if the service <b>TermService</b> is running. For each terminal session on the victim's machine, either the character <b>s</b> is appended, if the session currently has an active screen saver (e.g. is idle), or <b>e</b> if the session has an active explorer.exe instance running.
346	16 Bytes (FILETIME)	Victim's login time
354	6 bytes	MAC address of first NIC
360	4 Bytes	Believed to indicate the version of WhiskeyRomeo (set to "1.5")
364	2 Bytes (WORD)	Total number of MBs on all logical hard drives ( <b>DRIVE_FIXED</b> )
366	2 Bytes (WORD)	Total number of free MBs on all logical hard drives ( <b>DRIVE_FIXED</b> )
368	4 Bytes (DWORD)	Interesting open ports bitmask. Bit 0 = port 3389, Bit 1 = port 80, Bit 2 = port 445, Bit 3 = 3306, Bit 4 = 1433
372	5 Bytes	Unused
377	2 Bytes (WORD)	Unused, explicitly set to 0
379	1 Byte	Unused, explicitly set to 0

Table 13-2: RomeoWhiskey Command Base 0x1E10 based VictimInfoPacket Data Structure

RomeoWhiskey continues to receive commands from the client until a command results in an error or the disconnect command is received from the client. RomeoWhiskey disconnects from the client by using the Lazarus Group's standard socket disconnection function.

## 13.2 RomeoWhiskey-Two

.....

RomeoWhiskey-Two expands on the base code established by RomeoWhiskey-One. However, observed samples use only the *command base oxiEro* supported commands. The most noticeable differences between RomeoWhiskey-Two and RomeoWhiskey-One are the introduction of the concept of channels and the more advanced authentication that utilizes asymmetric encryption.

RomeoWhiskey-Two adheres to the basic model outlined in the previous section, but introduces a few deviations to the startup sequence:

1. Determine exclusivity on the infected system
2. Dynamically load API functions
3. Attempt to load the configuration file
4. **bind** to a listening port
5. Save the configuration
6. Open a hole in the Windows firewall
7. Open a hole in the perimeter firewall using SSDT
8. **listen** for incoming connections
9. Spawn a handler thread
10. Repeat steps 8-10 indefinitely.

In order to determine if only one instance of RomeoWhiskey-Two is active on a victim's system at any given time, the malware uses the somewhat common technique of creating a mutex with a specific name before calling **WaitForSingleObject** and determining the return code of the call. If the **WaitForSingleObject** call returns any result outside of **WAIT\_TIMEOUT**, then RomeoWhiskey-Two assumes that only one instance of the malware is running on the victim's machine. The name of the mutex depends on whether or not the victim's processor supports the **CPUID** command to retrieve processor info and feature bits. If the processor does not support the **CPUID** command for retrieving the processor's info and feature bits, the mutex is given the name **Microsoft**. Otherwise, the mutex is the hexadecimal representation of the 32-bit info bits and the 32-bit feature bits concatenated together.

RomeoWhiskey-Two stores the configuration in the file **%WINDIR%\tlvc.nls**. The format of the configuration file is identical to that of RomeoWhiskey-One and is encoded using the DNSCALC-style encoding scheme.

The task of determining a listening port is greatly simplified compared to RomeoWhiskey-One's method. The attempt to bind to a listening port is limited to four attempts, first using the port defined in the configuration and, if unsuccessful, then using three predefined port values (**547**, **133**, and **117**).

In order to allow inbound connections to the listening port, RomeoWhiskey-Two, like its predecessor, must open a hole in the Windows firewall. Instead of modifying the victim's registry, RomeoWhiskey-Two will use the **netsh** command to adjust the firewall's settings. The specific command that RomeoWhiskey-Two uses is Windows version specific as identified in Table 13-3.

OS VERISON	COMMAND LINE
XP or older	<code>cmd.exe /c netsh firewall addportopening protocol=tcp port={listening port} name=CoreNetworkingHTTPS</code>
Vista or newer	<code>cmd.exe /c netsh advfirewall firewall addrule name=CoreNetworkingHTTPS dir=in action=allow Protocol=TCP localport={listening port}</code>

Table 13-3: RomeoWhiskey-Two's Firewall Modification Command Lines

RomeoWhiskey-Two's developer(s) were not intent on modifying only the host-level firewall, as they actively try to manipulate the perimeter firewall as well. By using the Simple Service Discovery Protocol (SSDP), RomeoWhiskey-Two attempts to determine the next hop between the infected machine and the Internet. If the local router responds with the URL for its UPnP interface, RomeoWhiskey-Two issues a series of Universal Plug and Play (UPnP) commands in order to map an external port on the firewall to the RomeoWhiskey-Two's listening port and name the new mapping "**DHCP Client**". The net effect, if successful, allows RomeoWhiskey-Two to tunnel a hole from the exterior of the firewall (the Internet-facing interface) through to the RomeoWhiskey-Two instance. SSDP and UPnP configuration of firewalls is not common in larger networks, but rather is more likely to be found in a SOHO environment. This would indicate that either the developer(s) of RomeoWhiskey-Two were unaware of the limitation of SSDP and UPnP in a corporate environment, or that they were targeting smaller infrastructures.

Whenever a new connection from a client occurs, RomeoWhiskey-Two immediately spawns a new thread to handle the incoming requests. The generation of the new thread occurs prior to the authentication handshake, unlike RomeoWhiskey-One. The authentication handshake, meanwhile, is more evolved than the previously described function seen in Figure 13-1. RomeoWhiskey-Two generates a 16-byte buffer of random bytes before modifying the 3<sup>rd</sup> byte through the 7<sup>th</sup> byte with the seed value and the 10<sup>th</sup> through 13<sup>th</sup> bytes with the challenge value as Figure 13-2 illustrates. The client has 10 seconds to respond with another 16-byte buffer of which contains random data, a seed value and a challenge value in the same format. While the system, at face value, seems somewhat imposing, in actuality it is worthless for any form of cryptographic authentication. Both the request from the RomeoWhiskey-Two instance to the client and the client's response contain all of the components necessary to generate a valid response. The challenge value is derived from the seed value by performing several bitwise XORs and shifts, and an addition as Figure 13-3 describes. Therefore, by supplying both the seed and the challenge values, there is practically no value to the handshake other than to prove both sides know the algorithm. Furthermore, a simple packet replay by the client would satisfy the handshake's conditions.

OFFSET	+0	+1	+2	+3	+4	+5	+6	+7
0	Random	Random	Random	Seed value (DWORD)				Random
8	Random	Challenge value (DWORD)				Random	Random	Random

Figure 13-2: RomeoWhiskey-Two's Authentication Handshake Data Blob's Format

```

int Handshake(SOCKET s)
{
    struct
    {
        char r1[3];
        unsigned __int32 seed;
        char r2[2];
        unsigned __int32 challenge;
        char r3[3]
    } pkt;

    srand(GetTickCount());
    seed = rand() * rand();
    unsigned char *p = (unsigned char*)&pkt;
    i = 0;
    do
        p[i] = rand() % 256;
    while ( i < 16 );

    pkt.seed = seed;
    pkt.challenge = (((seed ^ 0x1A1E1C40u) >> 1) + 0x2E3E56E0) ^ 0xAF313230;

    if ( SendData(s, &pkt, sizeof(pkt), 1) )
        return 1;
    if ( WaitForSocketRecv(s, 10) )
        return 1;
    if ( RecvData(s, &pkt, sizeof(pkt), 1) )
        return 1;

    return (((pkt.seed ^ 0x1A1E1C40u) >> 1) + 0x2E3E56E0) ^ 0xAF313230 != pkt.challenge;
}

```

Figure 13-3:RomeoWhiskey-Two's Authentication Handshake Function

If the handshake fails, RomeoWhiskey-Two shuts down the connection with the client using the socket disconnection function found in many of the Lazarus Group's families. If the handshake is successful, however, RomeoWhiskey-Two expects the client to transmit two bytes (WORD). The WORD that the client sends specifies the particular channel the client is requesting. RomeoWhiskey-Two supports two channels: RAT (identified by **0xC8C8**) and Proxy (identified by **0x5A5A**). Each channel dictates a different form of traffic and supported commands. For instance, the RAT channel passes data in the previously mentioned datagram form, while the Proxy channel uses a different format entirely. This use of channels essentially allows RomeoWhiskey-Two to operate as two loosely coupled malware families at once.

If the client activates the RAT channel, RomeoWhiskey-Two sends a datagram of type **0x1E11** to the client before entering into another authentication phase. The RAT channel authentication phase is independent of the handshake authentication process prior to entering the RAT channel and is based on the model of the handshake authentication from RomeoWhiskey-One. The RAT channel authentication, while similar to that of Figure 13-1, now involves asymmetric encryption using RSA public and private keys, specifically the public key found in SierraJuliatt-MikeOne (see Section 17). After generating the challenge value through a series of bitwise shifts, XOR, and addition, RomeoWhiskey-Two encrypts the 4-byte (DWORD) value using the same RSA transform function found in other Lazarus Group families, again most notably SierraJuliatt-MikeOne. After transmitting the encrypted value to the client, RomeoWhiskey-Two waits up to 10 seconds for the client to respond. The client responds with a 4-byte (DWORD) value that must match the original challenge value for the authentication to succeed. The implications of this is that only a client who possesses the private key to decrypt the challenge value can access the RAT channel and its functionality.

After the authentication phase is complete, RomeoWhiskey-Two's RAT channel settles into the same pattern of receiving a datagram and dispatching the appropriate handle for the command found within the datagram. RomeoWhiskey-Two

supports all of the same commands as *command base 0x1E10* in RomeoWhiskey-One with the exception that the command for establishing a proxy/relay between the malware and an endpoint (**0x1E17**) has been replaced by the Proxy channel. RomeoWhiskey-Two's RAT channel introduced a new command (**0x1E10**) and a new response code (**0x1E31**) as seen in Table 13-4.

COMMAND NUMBER	DESCRIPTION
0x1E10	"Knock" on remote host port - Test if a connection is possible as the socket is immediately closed on success. Basically this is a port ping. Returns a success status if the port responds, otherwise returns a failure status.
0x1E31	[Response code] RAT channel active acknowledgment
0x1E32	[Response Code]

Table 13-4: RomeoWhiskey-Two's Additional Supported RAT Channel Commands and their Command Identifier Values

The Proxy channel in RomeoWhiskey-Two allows for the construction of a ghost network on top of existing infrastructure by linking RomeoWhiskey-Two infections to form virtual point-to-point sessions. After entering the Proxy channel, the client transmits a 112-byte structure (Table 13-5) that defines the hops (up to 10) used by the virtual connection in much the same way IP packets can specify Loose Source Routing or Strict Source Routing. The RomeoWhiskey-Two instance parses the Proxy channel command record sent by the client and determines the next hop, then makes a connection to the endpoint. The command record generally specifies the route as a predefined sequence of RomeoWhiskey-Two nodes to use, but, if the operations flag (offset 104) is set to **0xC2672253**, then the array of hops is used to randomly select the next node for the connection.

OFFSET	SIZE	DESCRIPTION
0	80 Bytes [array of endpoints]	Array of 10 records consisting of an IP (DWORD), a port (WORD) and a 2-byte unused field. Each record specifies a possible next hop in the virtual circuit.
80	4 Bytes (DWORD)	Next hop index in the array of hop records (offset 0)
84	4 Bytes (DWORD)	Maximum hops
88	4 Bytes (DWORD)	Final endpoint's IP address (as DWORD)
92	2 Bytes (WORD)	Final endpoint's port (as WORD)
94	2 Bytes	Unused
96	4 Bytes (DWORD)	Endpoint IP address (as DWORD)
100	2 Bytes (WORD)	Endpoint port (as WORD)
102	2 Bytes	Unused
104	4 Bytes (DWORD)	Operations flags
108	4 Bytes (DWORD)	Connection timeout

Table 13-5: RomeoWhiskey-Two's Proxy Channel Command Record Data Structure

Once the next hop has been determined by RomeoWhiskey-Two, a connection to the hop is made and the authentication handshake is performed. Earlier in this section it was pointed out that the algorithm used by the authentication handshake ultimately serves no value in terms of authentication. When viewed in the context of the Proxy channel, the method for the handshake begins to make a bit more sense, since the same function for performing the handshake is used here to establish a connection between one RomeoWhiskey-Two instance and another RomeoWhiskey-Two instance. Furthermore, taking into consideration that the handshake is no longer providing any real form of authentication, the use of asymmetric cryptography for the RAT channel's authentication makes significantly more sense to the overall design of the malware.



Depending on the position of the node receiving the command record within the overall virtual connection, the RomeoWhiskey-Two node requests either the RAT channel or the Proxy channel. The determination of which channel to use is made based on the value of the final endpoint IP address (offset 88). If the virtual connection has reached the final destination, then the RAT channel is opened, otherwise the Proxy channel is requested. With the appropriate channel established, RomeoWhiskey-Two activates two threads to handle the relaying of data from node to node, with each node responsible for data in only one direction. It is worth noting that the relay threads transmit data without employing the encoding scheme found elsewhere within the RomeoWhiskey-Two communications.

## 14. [Spreader] SierraAlfa

.....

A self-install service-based executable, SierraAlfa begins a chain of infection that ultimately leads to the potential devastation of an entire network of computers. SierraAlfa is responsible for the distribution and activation of WhiskeyAlfa on a victim's network. The observed samples of SierraAlfa were clearly built specifically for the SPE attacks as they contain infrastructure and account information specific to SPE's networks.

Two variants have been observed: SierraAlfa-One and SierraAlfa-Two. SierraAlfa-One is the base model, while SierraAlfa-Two provides additional features to ensure the propagation of the malicious payload within.

### 14.1 SierraAlfa Base (SierraAlfa-One)

.....

SierraAlfa-One is the base upon which the other SierraAlfa variant is derived. There have been only two observed SierraAlfa samples (one for SierraAlfa-One and one for SierraAlfa-Two) in the wild. Given the very specific targeting and nature of their functionality, it is highly probable the SierraAlfa family is a one-off series. Given that SierraAlfa-One is the base model, this section will refer to SierraAlfa-One simply as SierraAlfa. Unless otherwise noted in the following section, the activities present in SierraAlfa-One are the same as SierraAlfa-Two in both design and execution.

SierraAlfa, when activated, determines which, if any, command line arguments are present. If no command line arguments are present, SierraAlfa relaunches itself using `CreateProcess` after adding `-i` to its command line. The `-i` argument causes SierraAlfa to install itself as a service. SierraAlfa also supports the `-k` argument, which causes SierraAlfa to operate as a standalone service. Any other command line arguments will result in a window appearing as seen in Figure 14-1 (without the "About" dialog box which is shown to give a full perspective of the application). The window, and the resources within the SierraAlfa binary that produce the window, reveal that the developer(s) used the Visual C++ v6 "Hello World" template to create the basic application framework. Knowing that the title of the application, in this case `Hello`, is the same as the project name by default for the template, the original SierraAlfa project had the simple name of `Hello`.

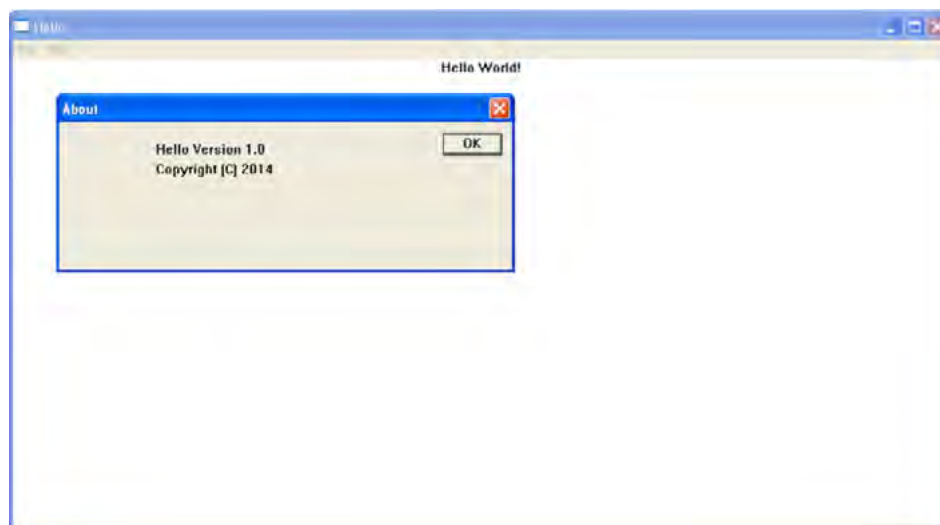


Figure 14-1: SierraAlfa's Window When Supplied an Invalid Command Line Argument

When activated with the **-i** command line argument in order to induce the service installation mode, SierraAlfa follows a very simple series of steps to install the **WinsSchMgmt** service on the victim's computer:

1. Call **CreateServiceW** to add a new service (**WinsSchMgmt**) to the victim's computer with the description "**Windows Schedule Management Service**" and set the command line argument for the service's binary to **-k**.
2. Call **ChangeServiceConfig2A** to, again, set the description to "**Windows Schedule Management Service**".
3. Call **ChangeServiceConfig2A** to set the on-failure retry modes.
4. Open a handle to the new service and call **StartServiceW** to activate the service.
5. Terminate silently.

SierraAlfa does not attempt to move its binary prior to installing itself as a service on the victim's computer. The inclusion of the **-k** argument for the service ensures that upon activation by the Windows Services system, SierraAlfa activates its service handler. The service handler, aside from the normal Windows services scaffolding, calls the function of SierraAlfa that kicks off the propagation functionality of the malware.

The propagation functionality begins by dropping a file on the victim's system and loading another into memory. The payload files of SierraAlfa are appended to the end of the SierraAlfa binary in a stacked fashion, preceded by a table of contents data structure. The location for the start of the table of contents and the stacked files that follow is determined by the last 4 bytes (DWORD) of the SierraAlfa binary. The 4 bytes at the end of the binary define the distance from the beginning of the binary to the beginning of the table of contents. The table of contents specifies both the actual size and the compressed size for each of the payload components that SierraAlfa drops or loads. Figure 14-2 visualizes the format of the payload's organization. The payload contains two files: a WhiskeyAlfa executable and a text file containing a list of target servers. The WhiskeyAlfa executable is always compressed with Zlib, while the target list may be either compressed using Zlib or simply stored. If the decompressed size and the size of the target list are not equal, SierraAlfa assumes the list is compressed and attempts to decompress the list in memory. Otherwise, the contents of the list are XOR'd with **0x67** to reveal the original content.

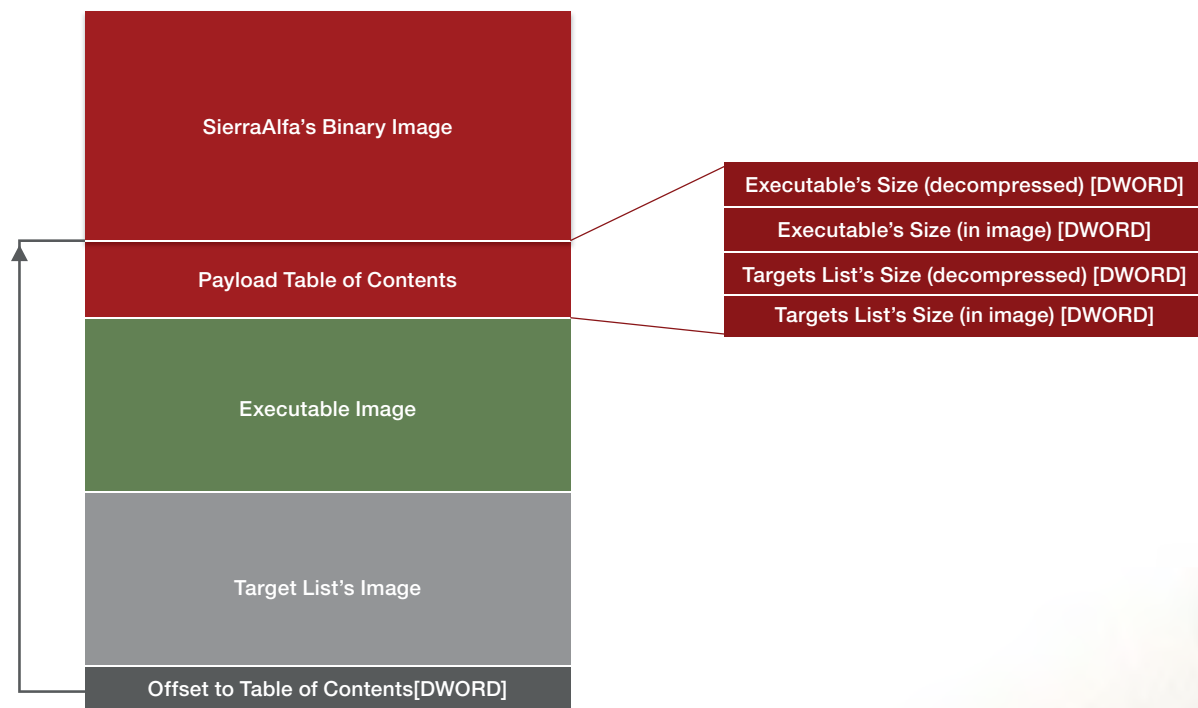


Figure 14-2: SierraAlfa's Stacked Payload Layout

SierraAlfa saves the WhiskeyAlfa binary to disk as **igfxtrayex.exe** in the same directory as the running SierraAlfa image and then immediately executes the binary by calling **CreateProcess**. The target list is parsed to extract target information and compromised accounts to use against the targets. The target list consists of two sections: a section specifying the username and passwords for compromised accounts and a section containing target servers to infect. SierraAlfa parses the target information as a list of compromised accounts until it reaches a line containing only a hyphen, at which point it assumes the rest of the file contains the target hosts list.

The structure for the account information is as follows:

```
domain\username|password
```

The structure for the target list is:

```
hostname|IP1|IP2
```

The *hostname* field specifies the Windows name of the computer, *IP1* field specifies the first IP address of the computer, and the optional *IP2* field specifies a second IP address for the server. The fact that the developer(s) allow for multiple addressing options for a target illustrates that the developer(s) took into consideration that an infected host may be on different network segments than the target computer; as such, different routing and addressing methods may be necessary in order to compromise a host. The use of multiple addressing options for a target shows both a sufficient level of reconnaissance within the victim's network and a desire by the developer(s) to ensure as many compromised targets as possible with the least amount of attacker intervention.

With the WhiskeyAlfa executable dropped and executed, and the targeting information loaded, the task of distributing itself across the victim's infrastructure begins. SierraAlfa maintains two lists: a list of targets (loaded by the previous step) and a list of hosts that it has already infected, or at the very least attempted to infect. Until the size of both lists becomes equal, indicating that all of the targeted hosts have been attacked, SierraAlfa attempts to infect up to 10 hosts at a time, with each attack occurring in its own thread. When SierraAlfa reaches a thread saturation level (10 threads), SierraAlfa enters a tight loop that simply sleeps rooms before checking to see if the current number of threads has fallen below 10. This thread management system is effective but rudimentary, indicating that the developer(s) were not well versed in multithreaded, asynchronous programming on Windows.

The selection of which host to attack at any given time is completely random. To avoid the duplication of effort that would stem from attacking a host that has already been attacked, the randomly selected hosts' entries are checked to ensure they have not already been used. If the host has already been attacked by the instance of SierraAlfa, a new host is randomly selected until a host is found that has not yet been attacked. This behavior would suggest that the longer SierraAlfa runs and attacks hosts, the longer the selection process will take. However, this methodology does allow for a better statistical coverage model when viewed as a larger set of infected hosts all using the same targeting information that exists in the same order. The randomization of which hosts to attack at any given point means that the probability of two hosts attacking the same sequence of targets in the same order is relatively low for a large enough data set. To that point, the target list within SierraAlfa-One contains over 20,000 target machines.

For each attack thread that SierraAlfa generates, the thread attempts to resolve the host name of the target, specified in the target information's *hostname* field. A determination is made as to whether or not the host is accessible and has an available Windows share port open by attempting to connect to either port 445 or 139. If the thread is able to access either port, the thread randomly selects one of the username and password entries before performing the following tasks:

1. Open a channel, via SMB, to the target host via a call to **WNetAddConnection2A**.
2. If unable to access the **admin\$** share on the target host, create a service named **RasMgrp**{random number} on the victim's machine with the command line:

```
cmd.exe /q /c net share shared$=%SystemRoot% /GRANT:everyone,FULL
```

3. Randomly select the destination filename from one of the following names:
  - **recdiscm32.exe**
  - **taskhosts64.exe**
  - **taskchg16.exe**
  - **rdpshellelex32.exe**
  - **mobsync1m64.exe**
  - **comon32.exe**
  - **diskpartmg16.exe**
  - **dpnsvr16.exe**
  - **expandmn32.exe**
  - **hwrcompsvc64.exe**
4. Copy the SierraAlfa binary to the **admin\$\syswow64** or **shared\$\syswow64** (if unable to access the **admin\$** share) directory as the randomly selected filename (from #3)
5. Copy the SierraAlfa binary to the **admin\$\system32** or **shared\$\system32** (if unable to access the **admin\$** share) directory as the randomly selected filename (from #3) and verify the copy was successful
6. Set the timestamp of the newly copied executable to match the target host's **calc.exe**'s timestamp.
7. If #2 resulted in a new service, replace the service's command line with

```
cmd.exe /q /c net share shared$ /delete
```

and restart the service

8. Install and activate a new service named **RasSecurity** (note the misspelling of "Security") with the command line set to the name of the newly installed binary in the **system32** directory.
9. If the service **RasSecurity** fails to activate, using the Windows Management Instrumentation Command-line application **wmic**, attempt to start the SierraAlfa binary on the target system by calling **CreateProcess** with the command line

```
cmd.exe /c wmic.exe /node:"{name of target host}" /user:"{username}"
/password:"{password}" PROCESS CALL CREATE "{randomly selected name}" > {current thread ID}_
{current tick count}
```

and parse the {current thread ID}\_{current tick count} file to determine if the string **ProcessId** was found, indicating the process started successfully.

If any of the thread's tasks fail, the thread disconnects from the target host by calling **NetCancelConnection2A** before attempting the tasks again using first the value of the *IP1* field as the target host; if that is unsuccessful, the value of the *IP2* field is used, if present.

If the infection of the target host is successful, SierraAlfa sends a reporting data packet to one the hardcoded C2 servers chosen at random. Table 14-1 defines the structure of the reporting data packet. If the transmission to a particular C2 server is unsuccessful, a new C2 server is chosen from the list of hard coded C2 servers, and the transmission it attempted again. SierraAlfa attempts to report the status of an infection up to three times before abandoning this endeavor.

OFFSET	SIZE	DESCRIPTION
0	2 Byte (WORD)	Size of the data that follows (40)
2	4 Bytes (DWORD)	IP address of the infected host (as 32-bit value)
6	32 Bytes	Hostname of the infected host
38	4 Bytes (DWORD)	Successful infection status (set to 1)

Table 14-1: The Structure of the SierraAlf Reporting Data Packet

After the transmission of the report data (or if the infection was unsuccessful, after the disconnect from the infected host), the thread decrements the global counter of active threads, thereby freeing up a new slot to allow a new infection thread to commence.

Once all attack operations have concluded, the SierraAlfa service remains in a running state but becomes idle.

## 14.2 SierraAlfa-Two

.....

SierraAlfa-Two has a compile time roughly two hours after the compile time of SierraAlfa-One. During the two hours between the variants, the developer(s) made several modifications:

- The status of an infection attempt is logged to **net\_ver.dat**
- The command line argument **-k** results in SierraAlfa being run under the context of each user currently logged in to victim machine through terminal services
- The propagation functionality is activated by supplying **-s** on the command line

SierraAlfa-One contains the basic framework for recording infection status information, but it is not until SierraAlfa-Two that the framework is fully completed by the developer(s) and utilized. At the conclusion of an attack thread, SierraAlfa-Two generates a string using the following form:

```
{Target Hostname}|{Target's IP Address}|{Infection Status}
```

The {Infection Status} field indicates whether an infection was successful (set to 1) or unsuccessful (set to 2). The targeting information within SierraAlfa-Two is significantly less involved than SierraAlfa-One. The target list contains only 58 hosts, many of which do not have a host name but only an IP address, and all of which are routable on the Internet. From this information, it would appear that SierraAlfa-Two was targeting network-facing, or firewall-exposed, targets only.

There is also the fact that the service that SierraAlfa installs does not immediately begin the propagation tasks, but rather targets the currently logged-in users of the victim host, another behavior change. The **-k** mode begins by calling **WTSEnumerateSessionW** to list the active terminal services sessions and, for each session found, the token of the user is obtained and given to **CreateProcessAsUser** in order to run SierraAlfa-Two as the logged-in user. This method does not necessarily increase the likelihood of a successful infection, given that the infection mechanism uses a finite set of preconfigured username/password combinations that have no relationship to the user under which the new SierraAlfa instances are running. It is therefore unclear why the developer(s) chose to add this method.

## 15. [Spreader] SierraBravo (Brambul)

SierraBravo, commonly known as Brambul, is a spreader that uses insecure user accounts to propagate its malware, and itself, across a both intra-connected and interconnected networks. SierraBravo operates as a standard executable, a service executable or, with some observed samples, as a service DLL.

SierraBravo has had several variants over the course of its developmental lifespan. The primary focus of the SierraBravo code base is the propagation of malware through unsecured or insecure network shares. Over time, the developer(s) of SierraBravo have added additional functions such as the ability to report to a C2 server the status of propagation. Despite a general cohesive task of propagating malware and a common functionality for performing this task, it is necessary to split it into two variants, SierraBravo-One and SierraBravo-Two, due to some functional and structural differences that are outside the scope of normal malware evolution and refinement. SierraBravo-One represents the variants that operate as either a standalone or service executable, while SierraBravo-Two contains the samples that operate as a service DLL.

Despite their structural differences, there are some commonalities between the variants:

- They rely on dynamic API loading with some, but not all, API function names encrypted using a variant of **DecryptPassword** from an open source malware known as rBot<sup>8</sup>
- The IP ranges 10.0.0.0/8, 12.0.0.0/8, 127.0.0.0/8, 192.0.0.0/8, 198.0.0.0/8, 216.0.0.0/8 are excluded from any attack (SierraBravo-One also excluded 8.0.0.0/8)

### 15.1 SierraBravo-One

The first operation SierraBravo-One performs is the verification of its exclusivity on the victim's system. By checking for the presences of the named mutex **Global\\FwtSqmSession106829323\_S-1-5-19**, SierraBravo-One can determine if it is the only instance of itself running on the victim's machine. If a copy of the malware is already active, SierraBravo-One generates and executes a suicide script in order to remove the extraneous copy of itself.

SierraBravo-One determines the number of command line arguments present at the time of activation. If at least one command line argument is given and the first argument is **-i**, SierraBravo-One enters an installation mode. Installation mode begins by verifying that there are 5 arguments on the command line. The five command line arguments SierraBravo-One expects are as follows:

```
-i <max. number of attack threads> <timeout> <primary C2 server address> <prime C2 server port>
```

SierraBravo-One uses the command line arguments to construct a base configuration file which it saves to **%WINDIR%\KB25468.dat** and encrypts using RC4. A new directory is generated at **%WINDIR%\system**, and the SierraBravo-One binary copies itself to the directory under the name **svchost.exe**. Using the Windows Service API functions, SierraBravo generates a new service named **Windows Filter Driver** with the newly installed

<sup>8</sup> "rbot6.6.rar crypt.cpp" [http://read.pudn.com/downloads110/sourcecode/hack/scanner/454581/rBot\\_041504/crypt.cpp\\_\\_\\_htm](http://read.pudn.com/downloads110/sourcecode/hack/scanner/454581/rBot_041504/crypt.cpp___htm) 14 April 2004.



**svchost.exe** binary being the target of the service. The **Windows Filter Driver** service is activated using the **StartService** API function before a suicide script is generated (as **msvcrt.bat**) and executed to remove the original SierraBravo-One binary from the victim's system.

When SierraBravo-One is activated as a standard executable with no command line arguments, the binary effectively activates as a service executable. When activated as a service executable, SierraBravo-One uses the **StartServiceCtrlDispatcher** API function to establish the service framework for the binary. After activating the necessary service framework to establish itself as a legitimate service on the victim's computer, SierraBravo-One transfers control to its core functionality.

The core functionality of SierraBravo-One is contained within two C++ classes and requires a minimal amount of scaffolding code in order to activate. Specifically, the dynamic loading of API functions, the loading of the configuration from **%WINDIR%\KB25468.dat** into memory, and the activation of the Windows's WinSock API. The two SierraBravo-One classes divide the tasks of running the attacks and managing the attacks. Novetta has given the class responsible for running the attacks the identifier **CSmbSpreader** and given the class responsible for managing the attacks the name **CBrambulManager**.

The instantiation of the **CSmbSpreader** class object consists establishing a list of targets and configuration settings. The file **%WINDIR%\KB25879.dat** contains a list of targets queued from previous executions of SierraBravo-One and is augmented at startup by a list of local network IP addresses. The instantiation of the **CBrambulManager** class object is significantly less involved and only includes the establishment of a watchdog event (which will be explained shortly).

After instantiating the two primary classes of SierraBravo-One, the malware activates **CBrambulManager** first by generating a new thread and calling the primary method within the class. The main member of **CSmbSpreader** is similarly activated within its own thread. At this point, the main thread of SierraBravo-One enters an infinite sleep.

The main member of **CBrambulManager** begins by generating a watchdog thread. The watchdog thread, as the name implies, periodically (every 5 minutes, approximately) sets the watchdog event. This process repeat until SierraBravo-One terminates. The main member of **CBrambulManager** uses the watchdog as an indicator of when it should begin the following set of tasks:

1. Connect to the configured C2 server
2. Transmit the current attack log (located at **%SYSTEMDRIVE%\perfw06.dat**) to the C2 server
3. Generate new, random target IPs if instructed by the C2 server
4. Receive additional targets from the C2 server
5. [Optionally] Send a heartbeat to the C2 server.

After performing the set of tasks, the main member of **CBrambulManager** resets the watchdog event and waits for the next watchdog event to occur.

What is important to note about the tasks of the main member of **CBrambulManager** is that SierraBravo-One does not only generate its own random targets, but it can also receive explicitly stated targets from a C2 server. If the C2 server provides additional targets, SierraBravo-One does not send a heartbeat. Instead, the main member of **CBrambulManager** sleeps for 3 minutes before waiting for the next watchdog event to occur.

Communication with the C2 server is encrypted using a simple XOR **0x37**. The handshake upon connect with the C2 server is a simple DWORD value exchange to verify that the encoding is symmetric between hosts. Such a simplistic encryption allows for easy decryption of communication between the C2 server and the infected host.

If a heartbeat that SierraBravo-One sends results in a failure to receive from the C2 server or if the server replies with a **0** (after decryption), the main function of **CBrambulManager** begins the process of uninstalling and terminating SierraBravo-One. The configuration file, the target list file, and the attack log files are deleted, and the service under

which SierraBravo-One exists is terminated and deleted. To complete the removal of SierraBravo-One, the malware generates and executes a suicide script before calling **ExitProcess**.

The main member of the **CSmbSpreader** class consists of an infinite loop that performs the following tasks until the SierraBravo-One process terminates:

1. Wait until the number of attack threads is less than the maximum allowed thread count
2. Convert the target IP from a binary number into a quad-dot string (e.g. **1.1.1.1**)
3. Generate a new attack thread for the target
4. After 255 attack threads, save the current state of the targeting queue
5. If the attack queue is empty, set the watchdog event
6. Sleep for 1 second

The attack thread that SierraBravo-One generates for each new target begins by attempting to connect to the target via Windows share (SMB). If successful, the domain and Windows OS type (e.g. Windows XP, Windows Vista, etc.) are obtained. The attack thread then attempts to bruteforce the Windows share by attempting a combination of the generated username and password combinations. The generating of usernames centers around permutations of the username **Administrator** in two different languages (English and Spanish) combined with variations of the target's reported domain name. The passwords that SierraBravo-One uses are generated from a list of 185 weak passwords, of which 11 are used as templates. If the password string contains **%u**, SierraBravo-One generates a new password where the **%u** substring has been replaced with generated username. Table 15-1 lists the hardcoded passwords found within SierraBravo-One.

%u	007	baseball	Admin789&*(
%u1234	qazwsx	1313	Password
%u!	root	!@#	lpassword
%u12	!@#\$\$%^&*	88888	password123
%u!@	00000	shadow	password.
%u2014	12	win	999999
%u1004	888	winxp	7777777
%u!	1212	sunshine	Admin
%u#1	dell	gateway	PASSWORD
%u123	abc	harley	adobe123
%u123\$%^	manager	internet	princess
112233	88888888	temp123	azerty
123456	q1w2e3r4	xp	qwer1234
123	1q2w3e	2007	qwer1234!~
admin	54321	admin!@#	admin@123
1234	password1	asdfg	1qaz@wsx
1	aaa	!@#\$\$%^	!!!!
password	home	2003	himself.51
P@ssw0rd	qazwsxedc	trustnol	elmismo.51
P@ssw0rd1	2010	golf	Elmismo.51
p@ssw0rd	pass	!@#\$\$%^&	ElMismo.51
0	computer	2112	
12345	4321	default	
1111qq	qwert	fish	
112233	test123	god	
QWER1234	121212	!	
0000	secret	2005	
12345678	iloveyou	6969	
123456789	asdf	!@#\$	
000000	aa	blank	
a	welcome	foobar	
123123	master	owner	
1111	compaq	passwd	
111	temp	test1	
111111	oracle	xxxx	
guest	1234qwer	password!	
admin123	abcd	passw0rd	
qwerty	q1w2e3	passw0rd!	
000	xxx	p@ssw0rd!	
654321	2008	1234567890	
1234567	7777	*1234	
abc123	cisco	1q2w3e4	
321	asdf123	1qaz2wsx	
11	asdfgh	!@#123	
11111111	q1w2e3r4t5	admin!@#456	
1q2w3e4r	zxcv	Admin!@#456	
server	00	admin123\$%^	
888888	control	Admin123\$%^	
11111	123abc	rootroot	
123qwe	2009	87654321	
love	backup	2014	
super	qwer	win2012	
8888	q1w2e3r4t5y6	admin123!@#	
test	win2003	Admin123!@#	
letmein	2002	admin789&*(	

Table 15-1: SierraBravo-One Password List (note that the entries in bold are template passwords)

With a list of username and passwords ready for the specific target, the attack thread attempts to bruteforce the target. SierraBravo-One attempts to pull a list of valid usernames from the target host by calling the **NetUserEnum** API function. If successful, the bruteforcing process uses the names returned from **NetUserEnum** instead of the variations of **Administrator**. Regardless of which usernames list is used, the bruteforce process does not use the Windows SMB API functions but rather performs the SMB interaction manually through SMB packet crafting. For each username and password combination, SierraBravo-One attempts to login to the target through the root Windows share (e.g. \\1.1.1.1) until a successful combination is found or the set of combinations is exhausted.

If successful in finding a vulnerable account on the targeted machine, SierraBravo-One attempts to install itself and its payload malware (typically, the payload malware is the instance of SierraJuliatt-MikeOne running on the same machine as the SierraBravo-One infection). The procedure for infecting a vulnerable machine is as follows:

1. Connect to the root share on the targeted host using the username and password obtained from the bruteforcing process by using the Windows SMB API.
2. Create a service on the targeted host as **RPCEvent**{random decimal number} with one of the following command lines in order to establish a new share named **\$adnim** on the targeted machine. Note the misspelling of the new share. The developer(s) specifically mislabeled the share to be close to the legit **admin\$** share. The use of the **GRANT** parameter occurs on targets who report their operating system as any value other than **Windows 5.0**, **Windows 5.1**, or **Windows 2002** (e.g., non-Windows NT based computers).

```
cmd.exe /q /c net share adnim$=%SystemRoot% /GRANT:{username},FULL
```

Figure 15-1: SierraBravo-One Share Command for Victims Using Windows 2000 or later

```
cmd.exe /q /c net share adnim$=%SystemRoot%
```

Figure 15-2: SierraBravo-One's Share Command for Victims running Non-Windows NT Computers

3. Copy the payload malware to targeted host's **%SYSDIR%** directory via the new **adnim\$** share.
4. Reads the local SierraJuliatt-MikeOne's *seed list* file (**mssscardprv.ax**) to obtain the first IP and port in the list.
5. Create a new service called **HelpEvent**{same random decimal number as step #2} with the command line seen in Figure 15-1 in order to activate the SierraJuliatt-MikeOne payload on the targeted machine.

```
cmd.exe /c {Binary Name} {IP of Seed Node} {Port of Seed Node}
```

Figure 15-3: SierraBravo-One's SierraJuliatt-MikeOne Activation Command

6. The **adnim\$** share is deleted by issuing the command seen in Figure 15-4 under the same **HelpEvent**{...} service.

```
cmd.exe /q /c net share adnim$ /delete
```

Figure 15-4: SierraBravo-One's Command to Delete the Previously Established Share

After the installation (or attempted installation) of the SierraJuliatt-MikeOne malware on the target machine, SierraBravo-One attempts to determine if the Windows Terminal Services port (3389) is accessible on the target. In order to make the accessibility determination, SierraBravo-One merely attempts to connect to the port. If the connection is successful (without any data transfer), SierraBravo-One considers the port available.

SierraBravo-One retains a log of all successful attacks. Each log entry contains the fields illustrated in Table 15-2.

OFFSET	SIZE	DESCRIPTION
0	4 Bytes (DWORD)	IP address of target
4	1 Byte	Host attributes bitmask: bit 0 – infected successfully bit 4 – has Terminal Services port open
5	1 Byte (Boolean)	Target successfully infected
6	50 Bytes (NULL-terminated string)	Username used to access the target
56	50 Bytes (NULL-terminated string)	Password used to access the target
106	50 Bytes (NULL-terminated string)	Domain of the target
156	100 Bytes (NULL-terminated string)	Windows OS version reported by SMB

Table 15-2: SierraBravo-One's Attack Log Entry Structure

Before saving the log entry to the log file, SierraBravo-One encrypts each log entry with RC4. The encryption key used for the RC4 encryption is 118 bytes long and changes every 10 minutes. The generation of the RC4 keys involves the creation of a random 118-byte buffer and then the encryption of the 118-byte buffer using the same RSA public key found in SierraJuliett-MikeOne. The encrypted RC4 key is then saved to the attack log file followed immediately by the encrypted log entry.

After performing an attack against a target, the resources of the attack thread are released and the global indicator of number of attack threads in use is decremented.

## 15.2 SierraBravo-Two

SierraBravo-Two operates as service DLL with the primary entry point for the malware existing within the **ServiceMain** function. After establishing the scaffolding for a legitimate Windows service, SierraBravo-Two spawns a new thread to contain the core SierraBravo-Two functionality. At inception, the core of SierraBravo-Two attempts to verify if the named mutex **PlatFormSDK2.1** exists on the victim's system indicating at another instance of SierraBravo-Two is already active. If another instance is active, the malware terminates to avoid collisions.

Like SierraBravo-One, SierraBravo-Two is heavily object oriented with the bulk of its functionality contained within a single class object. Upon instantiation, the SierraBravo-Two class requires the maximum number of parallel attack threads allowed, the filename of the malware to spread, and the number of seconds to wait for a response from a targeted machine before aborting the attack. This information is used by the class as the basis for the configuration of the SierraBravo-Two's operation. During the class initialization process, the list of queued attack targets is loaded into memory from the file **%SYSTEMDRIVE%\KB25879.dat** as well as an additional set of targets based on the local network's IP range (as determined by the configuration of the victim's network cards).

SierraBravo-Two performs the same attack against as a target as SierraBravo-One with the following modifications:

- The initial username list consists of Administrator in English, Spanish and French
- The SierraJuliett-MikeOne configuration/seed list file is also copied to the target
- The activation of SierraJuliett-MikeOne on the target machine does not include the {IP of Seed Node} or {Port of Seed Node} command line parameters
- A record of the compromise is not recorded locally, but an email of the event occurs instead
- After every 255 attacks, the current state of the attack queue is saved to disk.
- After the entirety of the attack queue is complete, the maximum of attack threads are generated with each thread marked for randomly generating IP addresses to attack and then attacking said targets. Note that, as indicated previously, some network ranges are excluded.

The most distinctive difference between SierraBravo-One and SierraBravo-Two is the use of email for alerting the operators to newly infected targets. For each target infected, SierraBravo-Two generates a thread to handle the generation and transmission of an email message via SMTP. The email addresses used by SierraBravo-Two for both the **To:** and **From:** fields change periodically, but the structure of the email is largely consistent. The message contains no body but the subject line provides all of the necessary information for an attacker. The subject is structured as follows:

```
<ip in dotquad>|<domain>|<os version>|<username>|<password>|<response code from the infection attempt>|<flags>
```

The **<flags>** field, as in SierraBravo-One, provides details about the status of the infection (bit 0) and if Windows Terminal Services are accessible via port 3389 (bit 4).

## 16. [Spreader] SierraCharlie

SierraCharlie is a spreader that appears to target RDP as its vector for propagation. Novetta has not spent a significant amount of time investigating the SierraCharlie family before publication, but the following characteristics of the malware family are known:

1. The random IP generation code found in both SierraJuliett-MikeOne and SierraBravo can be found within SierraCharlie
2. SierraCharlie, structurally speaking, is heavily object oriented (C++)
3. The suicide script within SierraCharlie is consistent with other Lazarus Group malware families
4. The propagation mechanism appears to focus on RDP
5. At least one sample identifies the malware's program name as "RDPBForce"
6. At least two samples have two distinct version information entries with in the resource section with one entry in English and the other in Korean.

## 17. [P2P Staging] SierraJuliett-MikeOne (Joanap Mk I.)

Commonly known as Joanap, SierraJuliett-MikeOne is a peer-to-peer (P2P) malware family that gives the Lazarus Group the ability to rapidly establish a common program base across all infected machines as well as provide remote administration functionality on each individual infection. SierraJuliett-MikeOne (SJM1) is the older sibling of SierraJuliett-MikeTwo (SJM2) (see Section 18). While both SJM1 and SJM2 perform essentially the same function and follow roughly the same communication protocols, the two do not constitute variants of one another in the sense that the term variant has been established in this report. SJM1 and SJM2 have clearly different code bases, indicating they were most likely developed independently of one another but based on a common design specification. It is the clear distinction between the two code bases that necessitates they fall into different families.

The samples within the SJM1 family are largely homogenous with each consecutive sample (based on compile time) having an average similarity 98.7% to its neighboring sample. Installed typically by IndiaJuliett, SJM1 operates as a svchost-dependent service DLL. The **ServiceMain** export does little more than provide the necessary scaffolding for SJM1 to appear to be an active service. The core of SJM1 is activated when the DLL is loaded by svchost and calls **DllMain**. **DllMain** spins the core of SJM1 off into its own thread. Some later samples of SJM1 place an intermediate piece of code between the **DllMain** and the core by having the newly generated thread call a function to call **DialogBoxParamA** with the **lpDialogFunc** ultimately calling the core of SJM1. It is unclear why this small variation was introduced as it seems to serve no real purpose.

The core of SJM1 consists of three functions as Figure 17-1 illustrates. The first function initializes the SJM1 system. The initialization function begins by performing dynamic API loading. The dynamic API loading functions within SJM1 use an AES implementation to decrypt the names of the API functions to load via GetProcAddress. The AES implementation, CRijndael, is a direct lift from a CodeProject project by George Anescu that he published in November, 2002.<sup>9</sup> The same dynamic API loading functions for the API functions from **kernel32.dll**, **psapi.dll**, and **ws2\_32.dll** used by SJM1 can be found in RomeoFoxtrot (see Section 8) albeit with different AES keys. A strange feature of these dynamic API loading functions, in both SJM1 and RomeoFoxtrot, is that all but the **ws2\_32.dll** functions use the same AES key, but for some reason the Winsock API loading code uses a different key.

```
void SierraRomeoMikeOneCore()
{
    if ( Initialize() )
    {
        if ( StartIncomingClientsHandler(0) )
            StartP2PClientThread(0);
    }
}
```

Figure 17-1: SierraJuliett-MikeOne's Core

<sup>9</sup> George Anescu. CodeProject. "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.



The **Initialize** function loads the current configuration file into memory from **%SYSDIR%\mssscardprv.ax**. The configuration file contains a 1346-byte structure that contains the basic configuration information for SJMI as well as the *seed list* of known peer nodes. Table 17-1 details the structure of the configuration data. The node type field (offset 0) is initially set to **0x1000101** during the initialization phase of SJMI.

OFFSET	SIZE	FIELD DESCRIPTION
0	4 Bytes (DWORD)	Node type
4	4 Bytes (DWORD)	IP address of the first NIC of the victim's system
8	16 Bytes (SYSTEMTIME structure)	Internal timestamp of the configuration data
24	6 Bytes	MAC Address of the first NIC of the victim's system
30	2 Bytes (WORD)	Checksum/hash of CPU's ID value
32	4 Bytes (DWORD)	Last command ID number
36	2 Bytes (WORD)	Listening port number
38	130 Bytes	Actor's remarks/Campaign ID
168	720 Bytes (30 NodeInfo structure array)	List of seed/known peer nodes (see Table 17-2 for details of the structure)
888	260 Bytes	Unknown string
1148	64 Bytes	Unknown string
1212	64 Bytes	Unknown string
1276	64 Bytes	Unknown string
1340	4 Bytes (DWORD)	Zero if the SJMI node is known to be behind a NAT and non-routable from the Internet
1344	2 Bytes (WORD)	Counter indicating the number of times the node has connected to peers

Table 17-1: Configuration Data Structure of SierraJuliatt-MikeOne

If the listening port field (offset 36) is set to 0, SJMI will attempt to determine a valid listening port on the victim's host. The process for determining a listening consist of the following steps:

1. Using a list of preferred listening ports, attempt to bind the port
2. If the bind is successful, close the socket and return the port number, otherwise try the next preferred listening port until the list is exhausted or a viable port is found
3. If the preferred listening port list is exhausted, attempt to find an available port, using the same method in steps 1 and 2 above, for all ports between 1024 and 2047, inclusive.

SJMI has a list of 26 preferred listening ports. The list begins with more commonly found ports such as HTTPS, POP3, DNS, and HTTP and tappers off to more obscure ports. The preferred listening port list is, in order of preference:

443, 110, 53, 80, 995, 25, 8080, 1816, 465, 1521, 3306, 1433, 3128, 109, 161, 444, 1080, 520, 700, 1293, 1337, 2710, 3100, 3305, 3689, 11371

With SJMI initialized, the malware calls the **StartIncomingClientsHandler** function to establish the server side component of the P2P bot. The **StartIncomingClientsHandler** function begins by establishing a listening socket

on the configured listening port before spawning a thread to handle incoming connections from peer nodes. Section 17.1 explains the operations of the server mode thread.

After **StartIncomingClientsHandler** returns, SJMI's core calls the function **StartP2PClientThread** to activate the client mode thread. After activating the client mode thread, **StartP2PClientThread** returns control to the core which in turn returns control to the loading application, in this case svchost. At this point, the SJMI malware is running in two asynchronous threads so the service scaffolding of SJMI is allowed to run by the Windows Services system.

## 17.1 Server Mode Thread

The service mode thread is an infinite loop that waits for incoming connections on the listening port and spawns a new thread to handle any connection. When a new peer node connects to another SJMI node, an authentication phase begins. The authentication between two SJMI nodes begins by the connecting node (the client, in this case) transmitting a 4-byte (DWORD) value to the receiving node (the server). The value that the client sends to the server indicates the general class of node the client is: standard node (0x1000) or a super-node (0x1000000). The authentication changes depending on the type of node connecting.

A super-node transmits another 4-byte (DWORD) value that specifies the size of the next transmission. The next transmission contains a buffer of data that has been encrypted by the super-node's private key. SJMI decrypts the data using a hardcoded public key found within the SJMI's **.data** section and the **RSATransform** function found in other Lazarus Group families. A standard node, on the other hand, will simply transmit a 16-byte buffer of random bytes to the server node. Both a super-node and a standard node will perform the initial data transmissions in cleartext.

Regardless of the type of node attempting to authentication, the server node echoes back to the client the data the client sent to the server, with the exception that, in the case of a super-node, the data is now decrypted using the RSA transform. When the server sends the data to the client, the data is encrypted using RC4. For each buffer that SJMI sends (in both server mode and client mode), the data is encrypted with RC4 using the key in Figure 17-2.

```
0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0, 0x1A, 0xFF, 0xEE, 0x48
```

Figure 17-2: SierraJuliett-MikeOne's RC4 Key

The Sbox within the RC4 implementation is reset after each buffer meaning that data boundaries are critical to avoid corrupting the data stream. It also means that particularly short bursts of data are going to retain discernable patterns as they traverse the network.

The client will then transmit a 30-byte string to the server (over the encrypted channel) if the server's response was correct. The server compares the 30-byte string with the hardcoded string **https://www.google.com/index.h** and if the two strings are identical, the authentication completes successfully. A misstep in any of the authentication steps will result in the authentication failing and the channel being closed by the server.

While the client can identify as a super-node, there is no advantage to do so as SJMI do not grant additional access or privileges to any node that authenticates as a super-node. The authentication process, from the perspective of the node in the server role, is purely a binary output: successful or unsuccessful. The authentication process does, however, ensure that both sides of the conversation have the same communication key for the RC4 encryption and that both sides understand the basic protocol for communication. After the authentication sequence completes, all communications between the two nodes uses the RC4 encryption for data that traverses the network between them.

Follow the authentication phase, the server mode expects the client to send a 2-byte (WORD) value that specifies the particular channel the client wishes to access. SJMI supports three different channels: crawler (**0x2000**), RAT (**0x4000**), sync (**0x8000**).

## 17.1.1 Crawler Channel

The crawler channel allows one node to quickly determine the node list of another node. By accessing the peer list of a node, it is a simple process to enumerate all of the connected nodes of the SJM1 network that have Internet-facing, routable interfaces.

The client node transmits a 6-byte data structure to the server node. The data structure consists of a 4-byte (DWORD) value specifying if the client node knows it is not behind a NAT device and thus Internet accessible followed by a 2-byte (WORD) value the defines the client's own listening port number. If the value specifying if the client is accessible from the Internet is **0**, the server will attempt to connect to the client on the specified listening port. This allows the server to inform the client if it is behind a non-routable NAT device. The routability test begins by attempting to connect to the client node on the specified listening port and then performing the authentication phase if the connection is successful. If both of those events occur, then the client node is considerable routable, otherwise the node is considered inaccessible. The status of the test is transmitted to the client over the original channel (which the client initiated) in the form of a 4-byte (DWORD) value of either **1** (routable) or **0** (non-routable). If the routability test fails, it is repeated two more times. Given that the routability test waits up to 30 seconds per test for a connection to the client node to succeed or timeout, a full minute and a half may expire from the time the crawler channel is activated and the routability test completes.

SJM1 maintains three lists of **NodeInfo** entries: a list of *seed nodes* (from the configuration file), a list of *known nodes* that the SJM1 has either connected to or received from another node who connected to, and a list of *client nodes* that connected to the SJM1 node. The *seed nodes* list is limited to 30 **NodeInfo** entries, the *known nodes* list has a limit of 50 **NodeInfo** entries, and the list of *client nodes* that have connected to the SJM1 node has a maximum of 100 **NodeInfo** entries.

A **NodeInfo** structure contains information about a particular node as Table 17-2 illustrates. The most important fields within the structure are the IP address of the node, the port upon which the node listens for incoming connections and the timestamp of the last time a node successfully connected to the node. The timestamp is recorded in the **VARIANTTIME** format which is essentially a floating point number (a **double**) that defines the number of days (the integer value) and partial days (the decimal value) since December 30, 1899 at midnight.<sup>10</sup> A value of 2.5 for instance, represents January 1, 1900 at noon.

OFFSET	SIZE	FIELD DESCRIPTION
0	4 Bytes (DWORD)	IP address of node
4	2 Bytes (WORD)	Listening port of node
6	2 Bytes	Unused
8	4 Bytes (DWORD)	Tracking field 1
12	4 Bytes (DWORD)	Tracking field 2
16	8 Bytes ( <b>VARIANTTIME</b> )	Timestamp of the last time the node was contacted

Table 17-2: SierraJuliett-MikeOne's *NodeInfo* Data Structure

At the completion of the routability test, the server node transmits its current time, in the **VARIANTTIME** format to the client followed by the full array of 50 *known nodes*'s **NodeInfo** entries. In order to prevent loops where a node connects back to itself, prior to transmitting the *known nodes* entries the server node removes any **NodeInfo** entry for the client node and replace the removed **NodeInfo** with an empty (all zeros) entry.

<sup>10</sup> Microsoft. "SystemTimeToVariantTime function" [https://msdn.microsoft.com/en-us/library/windows/desktop/ms221646\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms221646(v=vs.85).aspx) Accessed 7 December 2015.

After the server transmits the *known nodes* entries, the server again attempts to connect back to the client and authentication in order to determine if the client has a routable interface. If the server is successful in the connect back, the node is added to the list of *known nodes* if it does not already exist or, if the client is already in the *known nodes* list, its last contact time (field 16 of **NodeInfo**) is updated to reflect the current time. The process of adding or updating the client node's entry is repeated for the *seed nodes* list.

The server shuts down the channel and disconnect from the socket before terminating the thread handling the client's connection. Visually, the crawler channel's communication is illustrated by Figure 17-3. In the illustration the gray arrows represent the client initiated socket and the direction of communication for each step while the red arrows represent the server initiated connection back to the client during the routability test.

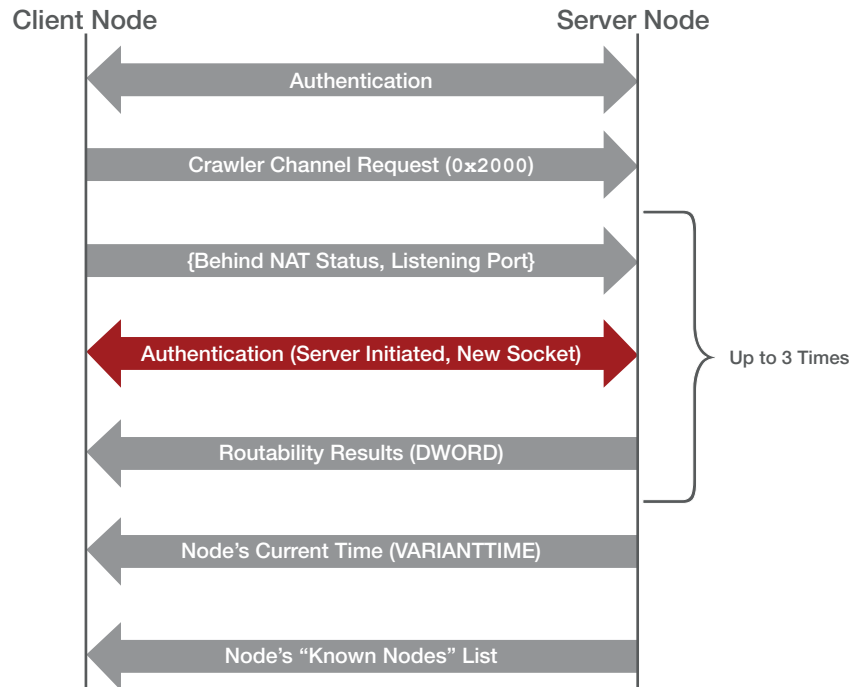


Figure 17-3: Crawler Channel Communication Sequence

### 17.1.2 RAT Channel

The RAT channel provides, as the name would imply, RAT capability to the SJM<sub>1</sub> family. Unauthenticated outside of the initial handshake between two nodes, the RAT channel support 21 different commands ranging from file management to data exfiltration to process management to node management.

Once a client requests the RAT channel, the client must send a datagram specifying the command and arguments (if any) for the command. Datagrams, in the context of SJM<sub>1</sub>, are variable sized data structures that specify a data type (or command type) followed by optional unstructured data specific to the data type specified. The datagram structure (Table 17-3) dictates that at a minimum a datagram is 6 bytes in size on the network.

OFFSET	SIZE	FIELD DESCRIPTION
0	4 Bytes (DWORD)	Size of the data transmission to follow
4	2 Bytes (WORD)	Data type
6	Variable	Optional payload data

Table 17-3: SierraJuliatt-MikeOne's Datagram Structure

The transmission of a datagram is always a two-step process. SJMI transmits the 4-byte size value first then transmits the remainder of the datagram. This is important to understand because, as mentioned previously, the RC4 encryption is reset after each transmission meaning that same sized datagrams, but potentially with different data types and data payloads, will always send the same 4-byte initial transmission.

The RAT channel uses the data type field (offset 4) of the datagram as the command type the client is requesting. If the command type is a recognized value, the appropriate command function is called and, if required, the payload data is passed to the function. Table 17-4 lists the supported command types and their descriptions.

COMMAND NUMBER	DESCRIPTION
0x4001	Echo. Receives another datagram from client then returns the same datagram to the client with the size field set to 512 bytes before sleeping for 1 second.
0x4002	Retrieves the client list and <i>known nodes</i> list.
0x4003	Sends Client Information. Sends the node's <b>ClientInfo</b> and <b>ClientInfoEx</b> data in individual datagrams.
0x4004	Attempts to connect to the specified endpoint. The first 4-bytes (DWORD) of the payload data specifies the IP address of the end point with the next 2-bytes (WORD) specifying the port number. If successful, return a datagram with the data type (offset 4) set to 1, otherwise the data type is set to 0 indicating the end point was unreachable.
0x4005	Uploads a local file to the client. The payload data contains the full name and path of the file to transfer to the client.
0x4006	Downloads a file from the client node. The payload data contains the destination filename and path starting at offset 3. Offset 1 of the payload data, if set, indicates if the file should be deleted if the download fails. The timestamp of the downloaded file is set to random date with the year set to 2 years prior to the current year.
0x4007	NOP
0x4008	Downloads a file from the client, executes the file, then deletes it. The filename is randomly generated as <b>rundll</b> {random number}. <b>exe</b> and given a random dates set two years prior to the current year.
0x4009	NOP
0x400A	Starts a process. The payload data contains the full command line to execute.
0x400B	Set the actor's remarks/campaign ID (offset 28) within the configuration data structure. The payload data contains the value for the field.
0x400C	Deletes a file or directory (if the name specified is a directory). The payload data contains a string specifying the full name and path to the file or directory to delete.
0x400D	Move (or rename) file. The datagram's payload data for the command contains two null terminated strings with the first specifying the source file's name and full path and the second string the new name and full path of the file.
0x400E	Creates a directory. The datagram's payload data contains a string specifying the full pathname of the new directory
0x400F	Terminates process by name. The datagram's payload data contains a string specifying the name of the process to terminate.
0x4010	Resets the last command ID to 0
0x4011	Deletes all command files under the <b>%SYSDIR%\1008</b> directory.
0x4012	Replaces the first <b>NodeInfo</b> entry in the seed node list. The payload data contains first 16-bytes of the new <b>NodeInfo</b> structure.
0x4013	Sets three configuration values (offsets 888, 1148, and 1212). The payload data of the datagram consists of three NULL-terminated strings, one for each value.
0x4014	Variation on command 0x4002.
0x4015	Pushes a command to all neighboring nodes.

Table 17-4: RAT Commands and Their Command Type Values Supported by SierraJuliett-MikeOne

After the RAT channel processes the requested command, SJMI terminates the connection between the client node and itself. Therefore, each time an attacker wishes to issue a command against a SJMI node, the attacker must reconnect to the node.

The inclusion of the RAT channel is somewhat unusual for several reasons. First, the channel is unauthenticated while, as presented in the sync channel discussion, there is a heavy use of asymmetric encryption to ensure command files are legitimate and only producible by the Lazarus Group attackers. Secondly, the construction of the RAT channel appears almost as if it were an afterthought of the developer(s) given that it requires constant reconnections to a node to issue multiple commands. Lastly, the RAT channel is single node focused, while the rest of the functionality of SJMI is geared toward a hive or collective.

### 17.1.3 Sync Channel

The crawler channel and sync channel share the same handler function with the crawler channel being the basis for the sync channel's operations. More to the point, the entirety of the crawler channel's events occurs as a precursor to the sync channel specific operations. Therefore, immediately following the transfer of the *known nodes* list to the client from the server and the updating of the appropriate **NodeInfo** entries in the *known nodes* and *seed nodes* lists, the sync channel adds to or updates the client to the *client nodes* list.

The client sends two datagrams containing information about the client node to the server. The first datagram contains a **ClientInfo** data structure (see Table 17-8) describing aspects of the client node related to its basic properties such as IP address and listening port as well as its type. The second datagram contains a **ClientInfoEx** data structure (see Table 17-9) that describes hardware and operating system level aspects of the node such as the OS version information, CPU details, and if particular ports of interest are open, among other details. If the size of the two datagram payloads matches their respective data structure sizes, the data structures are joined into a single data structure and stored within the list of *client nodes*. If, however, the size of the **ClientInfo** data structure is incorrect, the **ClientInfoEx** transfer is aborted by the server node.

The client node and the server node now begin the process of the client node synchronizing its command files. Command files are specifically formatted data files that SJMI uses to transfer commands from node to node. The structure of a command file (Table 17-5) consists of a RSA encrypted header, the command's data and its associated header in a cleartext header, and an optional encrypted data blob containing the information necessary to verify the integrity of the entire command file.

OFFSET	SIZE	FIELD DESCRIPTION
0	128 Bytes	Encrypted Command File Header (see Table 17-6)
128	Variable but at least 4 Bytes (m)	Data header (see Table 17-7)
m+4	4 Bytes (DWORD)	Size of data field
m+132	Variable (n)	Data
m+n+132	128 Bytes	(Optional) encrypted verification data

Table 17-5: SierraJuliatt-MikeOne's Command File Structure

The first 128 bytes of a command file are encrypted using a private RSA key that presumably only belongs to the Lazarus Group, as the key has not been found disclosed publicly. Underneath the RSA encryption lies the command file header (see Table 17-6) which species the type of command, an identifier for the command, the data/parameters of the command and the size of that data along with its header, and, optionally, the necessary information to verify the integrity of the entire command file along with the command's data and the data's associated header. The command ID field (offset 4) is particularly important as it allows SJMI nodes to quickly determine if a particular command file has previously been executed based on a running counter of the last command ID executed (offset 32 of the node's configuration data structure).

OFFSET	SIZE	FIELD DESCRIPTION
0	4 Bytes (DWORD)	Magic value ( <b>0xB4F4</b> )
4	4 Bytes (DWORD)	Command ID
8	2 Bytes (WORD)	Command type
10	2 Bytes (WORD)	Verify file flag
12	4 Bytes (DWORD)	Data header size

Table 17-6: SierraJuliatt-MikeOne's Command File Header

Interestingly, the data header, the size of the data portion, and the data portions of the command file are in cleartext. The data header contains three byte fields followed by an optional NULL-terminated string. The data that follows the data header contains the command specific data and varies in structure based on the command type.

OFFSET	SIZE	FIELD DESCRIPTION
0	1 Byte	Unknown field, seemingly unused
1	1 Byte	Activate command for node types less than <b>0x1000000</b>
2	1 Byte	Activate command for node types greater than or equal to <b>0x1000000</b>
3	Variable (NULL-terminated string)	Parameter string for command

Table 17-7: SierraJuliatt-MikeOne's Command File's Data Header

If the verify file flag (offset 10 of the command file header) is non-zero, the last 128 bytes of the command file contains another RSA private key encrypted data blob containing the MD5 hash of all of the bytes in the command up to, but not including, the last 128 bytes, which the client node can use to verify the integrity of the command file upon reception.

To perform the synchronization of command files from the server node to the client node, the client node transmits a 4-byte (DWORD) value to the server node representing the last command ID that the client executed. The server node responds with a 4-byte (DWORD) value containing its last command ID executed. The server node then enters a loop punctuated by calls to **FindFirstFile/FindNextFile** in order to enumerate all of the server's stored command files from the **%SYSTEMROOT%\1008** directory.

For each command file that the server node finds, the server node decrypts the command file's header, verifies that the decryption was successful by ensuring that offset 0 of the command file header is equal to **0xB4F4**, and then compares the command ID (offset 4) against the client's last command ID. For any file that successfully decodes and has a value greater than the last command ID reported by the client node, the server node sends a 4-byte value (DWORD) of **0x00000010** to the client to indicate that a command file is inbound. The server then sends the command file to the client using a sequence of datagrams. The protocol for sending a file to or from a node is as follows:

1. Transmit the entirety of the file in sequence of 4KB datagrams with the data type field set to **0x1111**
2. Conclude the transfer by sending a datagram with the data type field set to **0xFFFF**.

Once all necessary command files, if any, have been sent to the client, the server terminates the synchronization operation by transmitting 4-bytes (DWORD) of **0x00000002** to the client, and the client acknowledges the synchronizations conclusion by replying with 4-bytes (DWORD) of **0x00000001**. The server node terminates the communication channel with the client.

Visually, the sequence of events that make up the sync channel operations as viewed from the network perspective is illustrated in Figure 17-4.

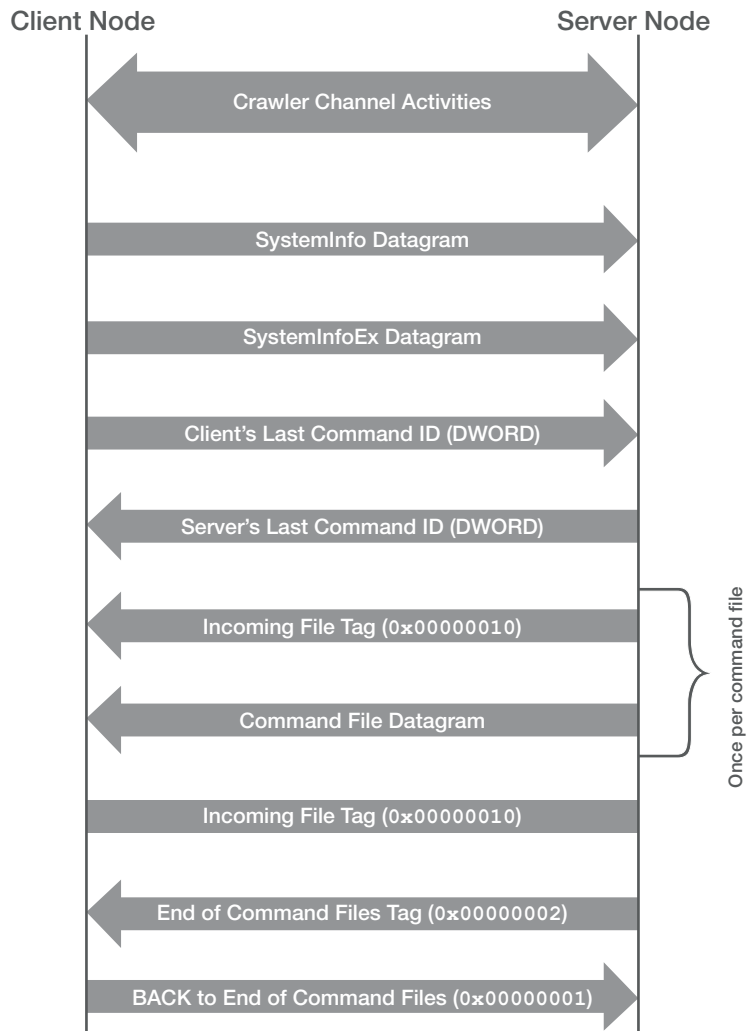


Figure 17-4: Sync Channel Communication Activities



## 17.2 Client Mode Thread

.....

The client mode thread, as one would expect, is largely the reciprocal of the server mode thread. Effectively an endless loop of constantly connection to neighboring nodes, the client mode thread is responsible for synchronizing command files from the larger SJMI botnet along with ensuring that the commands are bring properly executed.

The top of the endless loop begins with the client mode thread attempting to connect to a peer node. The client mode thread randomly selects a node from the *seed nodes* list and attempts to connect to the selected peer. If the connection is unsuccessful, a node from the *known nodes* list is randomly selected, and a connection is attempted. In the event that the connection is also unsuccessful, the client mode thread sleeps for one minute before repeating the process again.

When the client mode thread (in this context, making the SJMI node the client node) attempts to locate a remote peer, it does so by connecting to the node to determine the validity of the IP and port information, as it immediately terminates the connection if the connection is successful. Only after a node is validated as having an open port at the specified port number does the client node attempt to establish a lasting connection to the peer node. Once a connection has been established, the client initiates the authentication handshake. If the handshake fails, the connection is terminated, and control returns to the top of the endless loop in order to find a new peer node.

After the connection and authentication phase concludes, the client mode thread selects the appropriate channel based on its node type. If the node type, specified in the configuration file at offset 0, is greater than or equal to **0x1000000**, then the crawl channel (**0x2000**) is selected, and the node's node type is set to **0x1000101**. If the node's node type is less than **0x1000000**, the client node requests the sync channel.

The node type is adjusted periodically throughout the life span of a SJMI node. Upon activation, the **Initialize** function specifies that the node's node type is **0x1000101**. As will be described later in this section, if during the routability testing the peer node indicates that the client node is Internet-accessible, the node type is upgraded to **0x101**. While other values for the node type have been observed in the wild (most likely due to researchers attempting to distinguish themselves from legitimate SJMI nodes), the SJMI code supports only two node types:

- **0x101** for Internet-accessible (routable) nodes
- **0x1000101** for NAT'd (non-routable) nodes

Since sync channel is built upon crawl channel, regardless of the node's node type, the client node enters into the routability test as described previously in Section 17.1.1. From the client node's perspective, the routability test begins with the client node transmitting a 6-byte data structure to the server node containing the 4-byte (DWORD) value and a 2-byte (WORD) value. The 4-byte value specifies if the client node knows it is behind a NAT device and thus non-routable, and the 2-byte value defines the client node's listening port number. If the client node has already identified it is not NAT'd (offset 1340 of the configuration file is non-zero), the routability test phase concludes. Otherwise, the client node waits for the peer node to send a 4-byte (DWORD) value indicating if the client node's listening port is Internet-accessible or not. The response from the peer is recorded directly into the configuration file (at offset 1340).

In the event that the peer node indicates that the client node's listening port is inaccessible, the client node terminates its server mode thread by shutting down its listening port, randomly selects a new listening port, and starts a new server mode thread. With a new server mode thread and listening port, the client node performs the routability test again. The process of shutting down the server mode thread and generating a new listening port can occur twice before the client node concedes that it is inaccessible.

The results of the routability test may result in the node changing its node type. If a node is non-routable, its node type is set to **0x1000101**, otherwise the node's node type is adjusted to **0x101**. After the routability test, the client node resets its routability status value (offset 1340 in the configuration file) back to **0**, thereby forcing the routability test to commence each and every time it connects to a new peer node.

The client node receives the peer's current time as an 8-byte value (a **double**) followed immediately by the peer's 50 *known nodes* list. The client node then calculates the difference between its local time and the server's in order to determine the bias that must be applied when determining the age of any of the nodes the peer transmitted. Since the peer did successfully provide information, the peer is deemed a viable peer node to add to the client node's *seed list* if it was not there already by replacing the oldest node within the list. If the peer node is already within the client node's *seed list*, the last contact time is updated, thereby refreshing the node.

The client node scans the 50 nodes received from the peer node to determine which, if any, are newer than the client node's own *seed list* nodes. Each node has its timestamp recalculated by applying the bias value before determining if the oldest node within the client's *seed list* is newer than the received node. Should it turn out that the node is newer, the *seed list* node is replaced. The process repeats for each of the 50 nodes. It is therefore entirely possible for a client node to have all but one of its *seed list* entries replaced in full by a peer's node list if the client node had lost contact with the SJMI botnet for a long enough period of time.

If the client node opens the sync channel instead of the crawl channel, the client node constructs and then transmits to the peer node via datagrams both the **ClientInfo** and **ClientInfoEx** data structures. The **ClientInfo** data structure, as defined in Table 17-8, is the product of the client mode thread merging components of the configuration data structure into the **ClientInfo** form.

OFFSET	SIZE	FIELD DESCRIPTION
0	4 Bytes (DWORD)	IP address of node
4	2 Bytes (WORD)	Listening port of node
6	2 Bytes	Unused
8	4 Bytes (DWORD)	Node's tracking field 1
12	4 Bytes (DWORD)	Node's tracking field 2
16	4 Bytes (DWORD)	Node's type
20	4 Bytes	Unused
24	6 Bytes	MAC address
30	2 Bytes	Unused
32	16 Bytes (SYSTEMTIME structure)	Node's activation time
48	128 Bytes	Node's actor remarks/campaign ID

Table 17-8: SierraJuliatt-MikeOne's ClientInfo Data Structure

The **ClientInfoEx** data structure, Table 17-9, requires more processing to produce than its smaller sibling. The client mode thread leverages both Windows API functions such as **GetLocaleInfo**, **GetVersionEx**, **GetComputerName**, and **GetDiskFreeSpaceEx** as well as processor level instructions such as **cpuid** in order to construct the **ClientInfoEx** information. The *interesting ports* field (offset 280) is the product of determining if specific ports are listening for connections on the local machine, but not as a result of the SJMI node listening on a given port. For example, if port 80 responds to a **connect** request, the *interesting ports* bitmask is set to indicate the port is listening. However, if the SJMI node is configured to listen on port 80, the bitmask field is not set.

OFFSET	SIZE	FIELD DESCRIPTION
0	64 Bytes	Computer name
64	6 Bytes	MAC address
70	2 Bytes	Unused
72	64 Bytes	CPU brand (reported by CPUID)
136	16 Bytes	CPU's vendor ID (reported by CPUID)
152	4 Bytes (DWORD)	CPU's model and stepping (reported by CPUID)
156	4 Bytes (DWORD)	CPU's features bitmask (reported by CPUID)
160	4 Bytes (DWORD)	CPU's signature
164	4 Bytes (DWORD)	Number of processors
168	4 Bytes (DWORD)	CPU's type
172	4 Bytes (DWORD)	CPU's clock speed
176	64 Bytes	Computer's locale string
240	4 Bytes (DWORD)	OS's major version
244	4 Bytes (DWORD)	OS's minor version
248	4 Bytes (DWORD)	OS's build number
252	4 Bytes (DWORD)	OS's platform ID
256	16 Bytes (SYSTEMTIME structure)	Computer's uptime
272	8 Bytes (QWORD)	Total number of free bytes on the computer's <b>%WINDIR%</b> hard drive
280	1 Byte	Bitmask of <i>interesting ports</i> in use 0x01 – Port 80 0x02 – Port 3389 0x04 – Port 443
281	1 Byte	Unused
282	2 Bytes	Number of users on the computer in an idle state
284	2 Bytes	Number of logged in users on the computer
286	32 Bytes (16 WORDs)	Array of terminal server sessions' connection states.
318	2 Bytes	Unused

Table 17-9: SierraJuliett-MikeOne's ClientInfoEx Data Structure

The synchronization process begins by the client node sending its last command ID to the peer node and then receiving the peer node's last command ID. If the peer node has a last command ID that is larger than the client node's, the client node expects to receive an unknown number of command files from the peer as described previously in Section 17.1.3. Immediately prior to beginning the command file transfer, the client mode thread changes the last command ID in the node's configuration to match the value specified by the peer node.

For each command file the client node receives, the client mode thread saves the command file to the **%SYSDIR%\1008** directory with a filename taking the pattern **reg{4 digit value}** before parsing the contents of the file. While the order of the command IDs received by a node is indeterminate, the naming scheming has a definite order with each received file being stored with a file name one digit higher than the previous file. The timestamp of command files, when saved to disk, is set to a random date within 1 to 4 years from the current year.

The client mode thread begins the process of parsing a command file immediately after the file is saved to the node's hard drive. The encrypted command header (offset 0) is read into memory and decrypted using the public RSA key hard coded within the SJMI binary. To verify successful decryption, the first field of the decrypted header (offset 0) is compared to **0xB4F4**. A memory buffer of the size specified by the data header size value (offset 12) is allocated and the contents of the data header are read into memory. SJMI has the capability to selectively run commands based on the node type value. Offsets 1 and 2 within the data header specify the node types that will execute the given command within the command file. If the data header specifies that the node's node type is not to execute the command, the processing of the command file concludes, and the next file, if any, is loaded and parsed.

The data portion of the command file, if the command type is not **0x4006**, is saved to disk within the **%SYSDIR%\1008** directory with a name of the form **rundll{4 digit value}.exe**. If the verify file flag (field 10 of the command header) is set, the client thread reads all but the last 128 bytes of the command file into memory, performs a MD5 hash of the content, decrypts the last 128 bytes of the command file to reveal the expected MD5 hash, and then compares the hash values. If they do not match, the command file is considered invalid, and parsing of the file terminates. The client mode thread then moves to the next command file. The fact that invalid command files are not removed from the command file directory means that invalid command files propagate throughout the SJMI botnet, potentially leaving a considerable amount of noise. This behavior was observed when the SJMI botnet was enumerated in June 2015 by Novetta.

The client mode thread supports only a subset of the commands found within the RAT channel, but the command numbers are identical to the list found in Table 17-4. The client mode supports the commands in Table 17-10.

COMMAND NUMBER	DESCRIPTION
0x4006	Downloads a file from the client node.
0x4007	NOP
0x4008	Executes the file within the data field, then deletes it.
0x4009	NOP
0x400A	Starts a process.
0x400B	Set the actor's remarks/campaign ID (offset 28) within the configuration data structure.
0x400C	Deletes a file or directory (if the name specified is a directory).
0x400D	Moves (or renames) file.
0x400E	Creates a directory.
0x400F	Terminates process by name.
0x4010	Resets the last command ID to 0
0x4011	Deletes all command files under the <b>%SYSDIR%\1008</b> directory.
0x4012	Replaces the first <b>NodeInfo</b> entry in the seed node list.
0x4013	Sets three configuration values (offsets 888, 1148, and 1212).

Table 17-10: Client Mode Thread Supported Command Types

After the peer node transfers the last of the command files to the client node, the peer node sends a 4-byte (DWORD) value of **0x00000002** to the client node indicating the completion of the transfer. The client node only reads the last byte (0x02) to determine if the transfer is complete but returns a 4-byte (DWORD) value of **0x00000001** as an acknowledgement of the completion of the transfer before disconnecting from the peer node.

There is no verification that a command's execution is successful. Coupled with the fact that the node's last command ID is updated prior to receiving even the first command file from the peer node, this can lead to schisms within the SJMI

botnet. Take for example a new node joining the SJMI botnet. A new node has a last command ID of 0. If the first peer that the new node connects with were to have a last command ID of 65000, then the new node would immediately change its last command ID to 65000 to match. If the transfer of command files is error-prone due to network instability and some or all of the command files fail to transfer correctly, the new node would still retain the last command ID value of 65000. Going forward, when a node attempts to synchronize command files with the new node, the new node would report 65000 as its last command ID but would provide an incomplete set of command files to the requesting node. This error could therefore propagate unchecked throughout large sections of the SJMI botnet.

Regardless of the validity of command file transfer, once the client mode thread has disconnected from a peer, the thread enters a sleep period. The duration of the sleep varies depending on the number of times the client node has repeated the endless loop. The first 30 cycles through the loop will result in a sleep period of 30 minutes per cycle while any cycle after the initial 30 will cause a 2-hour sleep.

## 17.3 Known SierraJuliect-MikeOne Command Files

.....

In mid-2015, the SJM1 botnet was enumerated by Novetta to determine the current state of the command file distribution. The SJM1 botnet appears to be fractured and in disrepair. A large number of nodes have incomplete command file sets, extremely old (greater than 90 days) **NodeInfo** entries, or multiple corrupt command files. No single node appeared to have a complete set of command files. It was possible, however, to reconstruct the majority of the command file set by enumerating all command files from all nodes and identify unique, valid command files. All command files contained commands to execute a file contained within their data section. Table 17-11 maps the command ID to the embedded executable found within the command file's data field. Table 17-12 maps the command ID to the compile date of and type of executable found within the data section.

COMMAND ID	SHA256 HASH OF COMMAND FILE'S DATA SECTION
2	9b03695ca0945995ec6e2bc31662c08b0f499998dcbcd51701bf03add19f1000
10	e8d1d9d6bb13a06fc893323a05063c868ba237b8729c120271384382eb60ed41
12	e8d1d9d6bb13a06fc893323a05063c868ba237b8729c120271384382eb60ed41
200	2e20410ce8369572beee811f1898f6bc5c6782083aa1cc8e6dacc07b3fd392c9
210	3ee8fa11b85ec7a3e1f3cf3cee2553f795c56610091e373d4a7df344a66ae35d
300	7c55af4675cf0a3d173cb4e1b9282425c6e00b6ccfad1a1bcb0fddf29631461e
1000	7c55af4675cf0a3d173cb4e1b9282425c6e00b6ccfad1a1bcb0fddf29631461e
1010	7c55af4675cf0a3d173cb4e1b9282425c6e00b6ccfad1a1bcb0fddf29631461e
1050	231af2bfa36b6b0d2e892fbb967062eb0b421ee4f7126709c51adb564d0c5a2
1100	a64cb2496fb1ef1adf9b5473e664dc1d124634233dd76b4d8fb5aa8d970742b5
1200	191e14e54cae4b33c077065b782a7161f0fd807a550a98fd1dac2db2b622c94c
1205	f340bb3c2d175e027351319573ddc451b632defe9dc47bbc30eabf62f749fb46
3000	f340bb3c2d175e027351319573ddc451b632defe9dc47bbc30eabf62f749fb46
3500	1fd96cc95ec3f48e97cfd08bb15d4dd30c11a5b582776dfa15f1a2e2b4ed94e
3501	1200c02da0d6505a841f140f6d1947f1ae43a13664ec65b356b273c75f42713b
8000	81c87a5a67963eab5193d342781e6b65604f7af74dd5cf7da960d20074da06b5
8050	2d8e052bb93839dfef77b45be4418f64eeae35a7470a3c20827bae914dc1c7e4
10000	6ce54331e126fd18c94e854a5e7fe3650a125cc83604f1a27a28f383e5193c07
10000	c1820cc86b5cca32d9b09a191a9461552f1f4477d427270e7440bd9d03737a64
10000	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
10001	1dfe016ae106feb6112fd689faeaa1d61c19a911493a4201fb510551364f7247
10010	5ccfb9ba9aa0f05d2dd4006afd7769f2e186dd321b521617a469936de89aa9a7
10010	1b78ffb5e6a6e3a98baf433d1932d8b3e4907acb1fd27501f799cb2966c1395e
10011	1b78ffb5e6a6e3a98baf433d1932d8b3e4907acb1fd27501f799cb2966c1395e
30000	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
30001	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
50000	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
50001	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
60000	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
60001	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf
62001	d88d27eb6cbc7da8d8c61f42756153f386c7edae7a45b77d7368fbbf060eddf

Table 17-11: Command IDs and the SHA256 of Their Dropped Files

COMMAND ID	TYPE	COMPILATION DATE	NOTES
2	IndiaWhiskey	7/29/2011 6:29	
10	IndiaJuliect	7/26/2011 1:08	Installs SierraJuliect-MikeOne
12	IndiaJuliect	7/26/2011 1:08	Installs SierraJuliect-MikeOne
200	IndiaJuliect	8/23/2011 3:13	Installs IndiaJuliect and SierraBravo
210	IndiaJuliect	9/14/2011 5:54	Installs IndiaJuliect
300	IndiaJuliect	11/30/2011 1:55	Installs SierraJuliecta-MikeTwo
1000	IndiaJuliect	11/30/2011 1:55	Installs SierraJuliecta-MikeTwo
1010	IndiaJuliect	11/30/2011 1:55	Installs SierraJuliecta-MikeTwo
1050	IndiaJuliect	11/30/2011 16:34	Installs SierraJuliecta-MikeTwo
1100	IndiaJuliect	11/30/2011 17:06	Installs SierraJuliecta-MikeTwo
1200	IndiaJuliect	12/1/2011 12:24	Installs SierraJuliecta-MikeTwo
1205	UniformJuliect	12/4/2011 3:48	
3000	UniformJuliect	12/4/2011 3:48	
3500	IndiaJuliect	12/5/2011 10:42	Installs SierraJuliecta-MikeTwo
3501	IndiaJuliect	12/5/2011 12:18	Installs SierraJuliecta-MikeTwo
8000	IndiaJuliect	1/5/2012 4:02	Installs SierraJuliecta-MikeTwo
8050	TangoCharlie	1/8/2012 1:01	
10000	IndiaHotel	12/4/2012 7:30	Multiple valid hashes for the same command ID
10000	IndiaHotel	4/3/2013 11:26	Multiple valid hashes for the same command ID
10000	IndiaHotel	12/4/2012 7:30	Multiple valid hashes for the same command ID
10001	IndiaHotel	4/3/2013 11:26	
10010	IndiaHotel	3/29/2012 15:23	Multiple valid hashes for the same ID
10010	IndiaHotel	4/3/2012 0:29	Multiple valid hashes for the same ID
10011	IndiaHotel	4/3/2012 0:29	
30000	IndiaHotel	12/4/2012 7:30	
30001	IndiaHotel	12/4/2012 7:30	
50000	IndiaHotel	12/4/2012 7:30	
50001	IndiaHotel	12/4/2012 7:30	
60000	IndiaHotel	12/4/2012 7:30	
60001	IndiaHotel	12/4/2012 7:30	
62001	IndiaHotel	12/4/2012 7:30	

Table 17-12: Command File Payload Types and Their Compile Dates

Table 17-12 identifies some interesting irregularities within the command file set. For instance, there are three valid 10000 command ID numbers and two valid 10010 command ID numbers. This would indicate that the attackers utilizing the SJM1 botnet introduced multiple files with the same command ID which would result in potential inconsistency in the commands executed across the botnet. Command IDs 30000 and higher all distribute the same executable (an IndiaHotel installer). It is unclear why the attackers would continually redistribute the same executable.

The command file set does reveal a definite shift from SJM1 to SJM2. This is evidenced first by the fact that the IndiaJuliect files deployed across the botnet switch to the distribution of SJM2 instead of SJM1, and also by the fact that on two different occasions the attackers drop and execute UniformJuliect binaries.

## 18. [P2P Staging] SierraJuliect-MikeTwo (Joanap Mk. II)

SierraJuliect-MikeTwo (SJM2) has an incredibly similar structure as SierraJuliect-MikeOne (SJM1) but has no code (at the binary level) overlap. SJM2 is a complete rewrite of the concept seen with SJM1. Novetta observed the SJM2 malware as the payload of several IndiaJuliect samples that were introduced by SJM1 during its operational run. Installed as a svchost-dependent service, SJM2's binary is a DLL with the common **ServiceMain** function as its only export.

One of the most notable difference between SJM2 and SJM1 is the location of the configuration and *seed list* information. SJM2 stores its persistent data within the victim's registry at two different locations: **HKLM\SOFTWARE\Microsoft\DbgJITDebugLaunchSetting\00000000** for the configuration data and **HKLM\SOFTWARE\Microsoft\DbgManagedDebugger\00000000** for the peer list. The second most notable difference between the SierraJuliect families is the coding structure. Structurally, SJM2 differs from SJM1 by its heavy use of C++ instead of C. The bulk of the functionality of SJM2 is encapsulated in a set of C++ classes. While on the one hand the use of C++ classes provides clear delineation between the malware's various features, the use of C++ requires additional overhead for the developer. The implementation of SJM2 through C++ class objects suggests the malware was written by a developer with a more academic approach to coding when compared to SJM1, which has a style suggesting a more task-centric developer.

The communication protocol of SJM2 is incompatible with the protocol of SJM1. The incompatibility between the two indicates that SJM2 is not an evolutionary enhancement of SJM1 but a separate entity that must maintain its own network.



## 19. [Webserver] HotelAlfa

.....

HotelAlfa is a stripped down HTTP server that hosted the Guardians of Peace (GOP) hackers' webpage announcing their demands against SPE as well as the locations of the data that the GOP attackers stole. Consisting of only 4 functions, HotelAlfa is an extremely simple piece of code and is clearly created for a limited purpose.

Upon activation, HotelAlfa attempts to bind a listening socket to port 80 on the victim's machine. If port 80 is unavailable, HotelAlfa attempts to shutdown services (via a call to the API function **StopService**) in order to free up port 80 before attempting another bind operation. HotelAlfa attempts to stop the following services:

- **W3SVC** – IIS service
- **WMServer** – Windows Media Service
- **SSIS** – SQL Server Integration Service
- **SSRS** – SQL Server Reporting Service
- **MSDEPSVC** – Web Deployment Agent Service

For each incoming connection, HotelAlfa spins off a new thread to handle the request. The thread reads up to 4096 bytes from the client and scans the response for specific keywords. The request from the client does not necessarily need to conform or comply with the HTTP request standard. Instead, the request merely must contain the appropriate file extension otherwise the default HTML page is returned. HotelAlfa responds to **.wav** and **.jpg** file extensions with the appropriate file.

HotelAlfa only supplies three files to the client: an HTML page, a WAV sound file, and a JPG image. These files are stored within the HotelAlfa binary's resource section under the **RC\_DATA** branch. Each file is encoded with XOR **0x63**, requiring HotelAlfa to decode each file prior to transmitting the data back to the requesting client. When HotelAlfa sends a response back to the client, the response does conform to the HTTP 1.1 standard.

Table 19-1 describes each of the three files that HotelAlfa returns to the requesting client.


RESOURCE NAME	FILE DETAILS
RSRC_HTML	HTML code for the #GOP webpage. Contains links to a warning to SPE along with URLs to leaked SPE data.
RSRC_JPG	<p>Background image for the #GOP webpage, seen here:</p> 
RSRC_WAV	WAV sound file of gun shots that plays on the #GOP webpage in a loop.

Table 19-1: The Locations within the Resource Section of HotelAlfa and the Description of the Various File the Malware Serves to Users

## 20. Conclusion

The Lazarus Group employs a variety of RATs and staging malware to conduct cyber operations, many of which contain significant code overlap that points to at least a shared development environment. The development of these families also emphasizes the resources and organization of the Lazarus Group. The SierraJuliatt families, for instance, provides a common operating environment that effectively allows operators of any technical skill to access victim networks. Additionally, the Romeo-CoreOne-based families essentially acts as a modular design platforms and further simplifies the process for developing custom, targeted, and effective RATs.

While some members within the Romeo and Sierra groups may not implement sound authentication strategies, shift their design focus in abrupt and unusual manners, and fail to understand the pitfalls of distributed command networks, on the whole the families within the Lazarus Group's collection of RATs and staging malware perform their tasks with surprising effectiveness. As the maturity of the code base increases, so too does the effectiveness and design integrity of the malware families employed by the Lazarus Group.



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

[www.OperationBlockbuster.com](http://www.OperationBlockbuster.com)