From BYOVD to a 0-day: Unveiling Advanced Exploits in Cyber Recruiting Scams

decoded.avast.io/luiginocamastra/from-byovd-to-a-0-day-unveiling-advanced-exploits-in-cyber-recruiting-scams/

April 18, 2024

by Luigino CamastraApril 18, 202424 min read

Key Points

- Avast discovered a new campaign targeting specific individuals through fabricated job offers.
- Avast uncovered a full attack chain from infection vector to deploying `“FudModule 2.0”` rootkit with 0-day `Admin -> Kernel` exploit.
- Avast found a previously undocumented `Kaolin` RAT, where it could aside from standard RAT functionality, change the last write timestamp of a selected file and load any received DLL binary from C&C server. We also believe it was loading FudModule along with a 0-day exploit.

Introduction

In the summer of 2023, Avast identified a campaign targeting specific individuals in the Asian region through fabricated job offers. The motivation behind the attack remains uncertain, but judging from the low frequency of attacks, it appears that the attacker had a special interest in individuals with technical backgrounds. This sophistication is evident from previous research where the Lazarus group exploited vulnerable drivers and performed several rootkit techniques to effectively blind security products and achieve better persistence.

In this instance, Lazarus sought to blind security products by exploiting a vulnerability in the default Windows driver, appid.sys (CVE-2024-21338). More information about this vulnerability can be found in a corresponding blog post.

This indicates that Lazarus likely allocated additional resources to develop such attacks. Prior to exploitation, Lazarus deployed the toolset meticulously, employing fileless malware and encrypting the arsenal onto the hard drive, as detailed later in this blog post.

Furthermore, the nature of the attack suggests that the victim was carefully selected and highly targeted, as there likely needed to be some level of rapport established with the victim before executing the initial binary. Deploying such a sophisticated toolset alongside the exploit indicates considerable resourcefulness.

This blog post will present a technical analysis of each module within the entire attack chain. This analysis aims to establish connections between the toolset arsenal used by the Lazarus group and previously published research.

Initial access

The attacker initiates the attack by presenting a fabricated job offer to an unsuspecting individual, utilizing social engineering techniques to establish contact and build rapport. While the specific communication platform remains unknown, previous research by Mandiant and ESET suggests potential delivery vectors may include LinkedIn, WhatsApp, email or other platforms. Subsequently, the attacker attempts to send a malicious ISO file, disguised as VNC tool, which is a part of the interviewing process. The choice of an ISO file is starting to be very attractive for attackers because, from Windows 10, an ISO file could be automatically mounted just by double clicking and the operating system will make the ISO content easily accessible. This may also serve as a potential Mark-of-the-Web (MotW) bypass.

Since the attacker created rapport with the victim, the victim is tricked by the attacker to mount the ISO file, which contains three files: `AmazonVNC.exe`, `version.dll` and `aws.cfg`. This leads the victim to execute `AmazonVNC.exe`.

The `AmazonVNC.exe` executable only pretends to be the Amazon VNC client, instead, it is a legitimate Windows application called `choice.exe` that ordinarily resides in the `System32` folder. This executable is used for sideloading, to load the malicious `version.dll` through the legitimate `choice.exe` application. Sideloading is a popular technique among attackers for evading detection since the malicious DLL is executed in the context of a legitimate application.

When `AmazonVNC.exe` gets executed, it loads `version.dll`. This malicious DLL is using native Windows API functions in an attempt to avoid defensive techniques such as user-mode API hooks. All native API functions are invoked by direct syscalls. The malicious functionality is implemented in one of the exported functions and not in DLL Main. There is no code in `DLLMain` it just returns 1, and in the other exported functions is just Sleep functionality.

After the DLL obtains the correct syscall numbers for the current Windows version, it is ready to spawn an `iexpress.exe` process to host a further malicious payload that resides in the third file, `aws.cfg`. Injection is performed only if the Kaspersky antivirus is installed on the victim's computer, which seems to be done to evade Kaspersky detection. If Kaspersky is not installed, the malware executes the payload by creating a thread in the current process, with no injection. The `aws.cfg` file, which is the next stage payload, is obfuscated by VMProtect, perhaps in an effort to make reverse engineering more difficult. The payload is capable of downloading shellcode from a Command and Control (C&C) server, which we believe is a legitimate hacked website selling marble material for construction. The official website is `https://www[.]henraux.com/`, and the attacker was able to download shellcode from `https://www[.]henraux.com/sitemaps/about/about.asp`

In detailing our findings, we faced challenges extracting a shellcode from the C&C server as the malicious URL was unresponsive.

By analyzing our telemetry, we uncovered potential threats in one of our clients, indicating a significant correlation between the loading of shellcode from the C&C server via an ISO file and the subsequent appearance of the `RollFling`, which is a new undocumented loader that we discovered and will delve into later in this blog post.

Moreover, the delivery method of the ISO file exhibits tactical similarities to those employed by the Lazarus group, a fact previously noted by researchers from Mandiant and ESET.

In addition, a `RollSling` sample was identified on the victim machines, displaying code similarities with the `RollSling` sample discussed in Microsoft's research. Notably, the `RollSling` instance discovered in our client's environment was delivered by the `RollFling` loader, confirming our belief in the connection between the absent shellcode and the initial loader `RollFling`. For visual confirmation, refer to the first screenshot showcasing the SHA of `RollSling` report code from Microsoft, while on the second screenshot is the code derived from our `RollSling` sample.

```
  fix_api();
  pFileName = (char *)&lpFileName;
  if ( lpFileName._Myres >= 0x10 )
    pFileName = lpFileName._Bx._Ptr;
  FirstFile = FindFirstFileExA(pFileName, FindExInfoStandard, &FindFileData, FindExSearchNameMatch, 0LL, 0);
  if ( FirstFile != (HANDLE)-1LL )
  {
    while ( FindFileData.cFileName[0] == '.'
         && (!FindFileData.cFileName[1] || FindFileData.cFileName[1] == '.' && !FindFileData.cFileName[2])
         || (FindFileData.dwFileAttributes & 0x10) != 0
         || load_binary_to_memory_and_execute_StartAction_export_function(FindFileData.cFileName) )
    {
      if ( !FindNextFileA(FirstFile, &FindFileData) )
        goto looking_in_another_path;
    }
    goto exit;
  }
looking_in_another_path:
  if ( !load_binary_to_memory_and_execute_StartAction_export_function(0LL) )
    v1 = 0;
  v3 = v1;
  if ( FirstFile != (HANDLE)-1LL )
exit:
    FindClose(FirstFile);
```

Image illustrates the `RollSling` code identified by Microsoft. SHA: `d9add2bfdfebfa235575687de356f0cefb3e4c55964c4cb8bfdcdc58294eeaca`.

```
  fix_api();
  DLL_folder = (char *)&FullyQualifiedPath_to_folder_where_is_module;
  if ( FullyQualifiedPath_to_folder_where_is_module._Myres >= 0x10 )
    DLL_folder = FullyQualifiedPath_to_folder_where_is_module._Bx._Ptr;
  FirstFile = FindFirstFileExA(DLL_folder, FindExInfoStandard, &FindFileData, FindExSearchNameMatch, 0LL, 0);
  if ( FirstFile == (HANDLE)-1LL )
  {
looking_in_another_path:
    load_binary_to_memory_and_execute_StartAction_export_function(0LL);
    if ( FirstFile == (HANDLE)-1LL )
      goto exit;
  }
  else
  {
    while ( FindFileData.cFileName[0] == '.'
         && (!FindFileData.cFileName[1] || FindFileData.cFileName[1] == '.' && !FindFileData.cFileName[2])
         || (FindFileData.dwFileAttributes & 0x10) != 0
         || load_binary_to_memory_and_execute_StartAction_export_function(FindFileData.cFileName) )
    {
      if ( !FindNextFileA(FirstFile, &FindFileData) )
        goto looking_in_another_path;
    }
  }
  FindClose(FirstFile);
```

Image showcases the `RollSling` code discovered within our targe. SHA: `68ff1087c45a1711c3037dad427733ccb1211634d070b03cb3a3c7e836d210f`.

In the next paragraphs, we are going to explain every component in the execution chain, starting with the initial `RollFling` loader, continuing with the subsequently loaded `RollSling` loader, and then the final `RollMid` loader. Finally, we will analyze the `Kaolin` RAT, which is ultimately loaded by the chain of these three loaders.

Loaders

RollFling

The `RollFling` loader is a malicious DLL that is established as a service, indicating the attacker's initial attempt at achieving persistence by registering as a service. Accompanying this `RollFling` loader are essential files crucial for the consistent execution of the attack chain. Its primary role is to kickstart the execution chain, where all subsequent stages operate exclusively in memory. Unfortunately, we were unable to ascertain whether the DLL file was installed as a service with administrator rights or just with standard user rights.

The loader acquires the System Management BIOS (SMBIOS) table by utilizing the Windows API function `GetSystemFirmwareTable`. Beginning with Windows 10, version 1803, any user mode application can access SMBIOS information. SMBIOS serves as the primary standard for delivering management information through system firmware.

By calling the `GetSystemFirmwareTable` (see Figure 1.) function, `SMBIOSTableData` is retrieved, and that `SMBIOSTableData` is used as a key for decrypting the encrypted `RollSling` loader by using the XOR operation. Without the correct `SMBIOSTableData`, which is a 32-byte-long key, the `RollSling` decryption process would be ineffective so the execution of the malware would not proceed to the next stage. This suggests a highly targeted attack aimed at a specific individual.

This suggests that prior to the attacker establishing persistence by registering the `RollFling` loader as a service, they had to gather information about the `SMBIOS` table and transmit it to the C&C server. Subsequently, the C&C server could then reply with another stage. This additional stage, called `RollSling`, is stored in the same folder as `RollFling` but with the `".nls"` extension.

After successful `XOR` decryption of `RollSling`, `RollFling` is now ready to load decrypted `RollSling` into memory and continue with the execution of `RollSling`.

```
if ( GetModuleHandleExA(6u, ServiceMain, &phModule) )
{
  if ( GetModuleFileNameA(phModule, Filename, 0x104u) )
  {
    strcpy_s(nlsFile, 260uLL, Filename);
    strcat_s(nlsFile, 260uLL, ".nls");
    memset(v27, 0, 0x104uLL);
    hFile = CreateFileA(nlsFile, 0x80000000, 1u, 0LL, OPEN_EXISTING, 0x80u, 0LL);
    hFileNls = hFile;
    if ( hFile != -1LL )
    {
      GetFileSizeEx(hFile, &FileSize);
      mem = LocalAlloc(0x40u, FileSize.QuadPart);
      v3 = mem;
      if ( mem )
      {
        ReadFile(hFileNls, mem, FileSize.LowPart, &NumberOfBytesRead, 0LL);
        CloseHandle(hFileNls);
        numberOfBytesWrittentoTheBUffer = GetSystemFirmwareTable('RSMB', 0, 0LL, 0);
        FirmwareTable = LocalAlloc(0x40u, numberOfBytesWrittentoTheBUffer);
        GetSystemFirmwareTable('RSMB', 0, FirmwareTable, numberOfBytesWrittentoTheBUffer);
        SMBIOSTableData = FirmwareTable->SMBIOSTableData;
```

Figure 1: Obtaining SMBIOS firmware table provider

RollSling

The `RollSling` loader, initiated by `RollFling`, is executed in memory. This choice may help the attacker evade detection by security software. The primary function of `RollSling` is to locate a binary blob situated in the same folder as `RollSling` (or in the Package Cache folder). If the binary blob is not situated in the same folder as the `RollSling`, then the loader will look in the Package Cache folder. This binary blob holds various stages and configuration data essential for the malicious functionality. This binary blob must have been uploaded to the victim machine by some previous stage in the infection chain.

The reasoning behind binary blob holding multiple files and configuration values is twofold. Firstly, it is more efficient to hold all the information in a single file and, secondly, most of the binary blob can be encrypted, which may add another layer of evasion meaning lowering the chance of detection.

`Rollsling` is scanning the current folder, where it is looking for a specific binary blob. To determine which binary blob in the current folder is the right one, it first reads 4 bytes to determine the size of the data to read. Once the data is read, the bytes from the binary blob are reversed and saved in a temporary variable, afterwards, it goes through several conditions checks like the MZ header check. If the MZ header check is done, subsequently it looks for the `“StartAction”` export function from the extracted binary. If all conditions are met, then it will load the next stage `RollMid` in memory. The attackers in this case didn’t use any specific file name for a binary blob or any specific extension, to be able to easily find the binary blob in the folder. Instead, they have determined the right binary blob through several conditions, that binary blob had to meet. This is also one of the defensive evasion techniques for attackers to make it harder for defenders to find the binary blob in the infected machine.

This stage represents the next stage in the execution chain, which is the third loader called `RollMid` which is also executed in the computer’s memory.

Before the execution of the `RollMid` loader, the malware creates two folders, named in the following way:

- %driveLetter%:\\ProgramData\\Package Cache\\[0-9A-Z]{8}-DF09-AA86-YI78-[0-9A-Z]{12}\\
- %driveLetter%:\\ProgramData\\Package Cache\\ [0-9A-Z]{8}-09C7-886E-II7F-[0-9A-Z]{12}\\

These folders serve as destinations for moving the binary blob, now renamed with a newly generated name and a `".cab"` extension. `RollSling` loader will store the binary blob in the first created folder, and it will store a new temporary file, whose usage will be mentioned later, in the second created folder.

The attacker utilizes the `"Package Cache"` folder, a common repository for software installation files, to better hide its malicious files in a folder full of legitimate files. In this approach, the attacker also leverages the `".cab"` extension, which is the usual extension for the files located in the `Package Cache` folder. By employing this method, the attacker is trying to effectively avoid detection by relocating essential files to a trusted folder.

In the end, the `RollSling` loader calls an exported function called `"StartAction"`. This function is called with specific arguments, including information about the actual path of the `RollFling` loader, the path where the binary blob resides, and the path of a temporary file to be created by the `RollMid` loader.

```
  fix_api();
  DLL_folder = (char *)&FullyQualifiedPath_to_folder_where_is_module;
  if ( FullyQualifiedPath_to_folder_where_is_module._Myres >= 0x10 )
    DLL_folder = FullyQualifiedPath_to_folder_where_is_module._Bx._Ptr;
  FirstFile = FindFirstFileExA(DLL_folder, FindExInfoStandard, &FindFileData, FindExSearchNameMatch, 0LL, 0);
  if ( FirstFile == (HANDLE)-1LL )
  {
looking_in_another_path:
    load_binary_to_memory_and_execute_StartAction_export_function(0LL);
    if ( FirstFile == (HANDLE)-1LL )
      goto exit;
  }
  else
  {
    while ( FindFileData.cFileName[0] == '.'
         && (!FindFileData.cFileName[1] || FindFileData.cFileName[1] == '.' && !FindFileData.cFileName[2])
         || (FindFileData.dwFileAttributes & 0x10) != 0
         || load_binary_to_memory_and_execute_StartAction_export_function(FindFileData.cFileName) )
    {
      if ( !FindNextFileA(FirstFile, &FindFileData) )
        goto looking_in_another_path;
    }
  }
  FindClose(FirstFile);
```

Figure 2: Looking for a binary blob in the same folder as the `RollFling` loader

RollMid

The responsibility of the `RollMid` loader lies in loading key components of the attack and configuration data from the binary blob, while also establishing communication with a C&C server.

The binary blob, containing essential components and configuration data, serves as a critical element in the proper execution of the attack chain. Unfortunately, our attempts to obtain this binary blob were unsuccessful, leading to gaps in our full understanding of the attack. However, we were able to retrieve the `RollMid` loader and certain binaries stored in memory.

Within the binary blob, the `RollMid` loader is a fundamental component located at the beginning (see Figure 3). The first 4 bytes in the binary blob describe the size of the `RollMid` loader. There are two more binaries stored in the binary blob after the `RollMid`

loader as well as configuration data, which is located at the very end of the binary blob. These two other binaries and configuration data are additionally subject to compression and AES encryption, adding layers of security to the stored information.

As depicted, the first four bytes enclosed in the initial yellow box describe the size of the `RollMid` loader. This specific information is also important for parsing, enabling the transition to the subsequent section within the binary blob.

Located after the `RollMid` loader, there are two 4-byte values, distinguished by yellow and green colors. The former corresponds to the size of `FIRST_ENCRYPTED_DLL` section, while the latter (green box) signifies the size of `SECOND_ENCRYPTED_DLL` section. Notably, the second 4-byte value in the green box serves a dual purpose, not only describing a size but also at the same time constituting a part of the 16-byte AES key for decrypting the `FIRST_ENCRYPTED_DLL` section. Thanks to the provided information on the sizes of each encrypted DLL embedded in the binary blob, we are now equipped to access the configuration data section placed at the end of the binary blob.

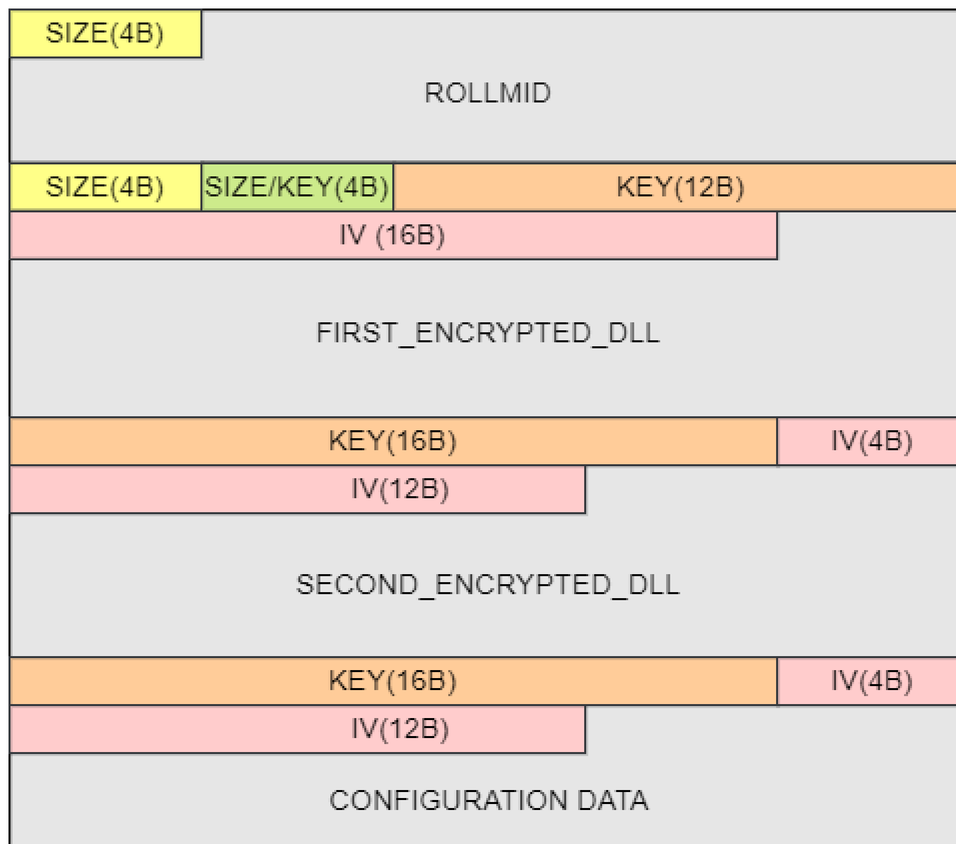

Figure 3: Structure of the Binary blob

The `RollMid` loader requires the `FIRST_DLL_BINARY` for proper communication with the C&C server. However, before loading `FIRST_DLL_BINARY`, the `RollMid` loader must first decrypt the `FIRST_ENCRYPTED_DLL` section.

The decryption process applies the AES algorithm, beginning with the parsing of the decryption key alongside an initialization vector to use for AES decryption. Subsequently, a decompression algorithm is applied to further extract the decrypted content. Following this, the decrypted `FIRST_DLL_BINARY` is loaded into memory, and the `DllMain` function is invoked to initialize the networking library.

Unfortunately, as we were unable to obtain the binary blob, we didn't get a chance to reverse engineer the `FIRST_DLL_BINARY`. This presents a limitation in our understanding, as the precise implementation details for the imported functions in the `RollMid` loader remain unknown. These imported functions include the following:

- `SendDataFromUrl`
- `GetImageFromUrl`
- `GetHtmlFromUrl`
- `curl_global_cleanup`
- `curl_global_init`

After reviewing the exported functions by their names, it becomes apparent that these functions are likely tasked with facilitating communication with the C&C server. `FIRST_DLL_BINARY` also exports other functions beyond these five, some of which will be mentioned later in this blog.

The names of these five imported functions imply that `FIRST_DLL_BINARY` is built upon the curl library (as can be seen by the names `curl_global_cleanup` and `curl_global_init`). In order to establish communication with the C&C servers, the `RollMid` loader employs the imported functions, utilizing HTTP requests as its preferred method of communication.

The rationale behind opting for the curl library for sending HTTP requests may stem from various factors. One notable reason could be the efficiency gained by the attacker, who can save time and resources by leveraging the HTTP communication protocol. Additionally, the ease of use and seamless integration of the curl library into the code further support its selection.

Prior to initiating communication with the C&C server, the malware is required to generate a dictionary filled with random words, as illustrated in Figure 4 below. Given the extensive size of the dictionary (which contains approximately hundreds of elements), we have included only a partial screenshot for reference purposes. The subsequent sections of this blog will delve into a comprehensive exploration of the role and application of this dictionary in the overall functionality of malware.

```
v3->dictionary1->dic[0] = "atyp";
v3->dictionary1->dic[1] = "dt19";
v3->dictionary1->dic[2] = "loop";
v3->dictionary1->dic[3] = "keep";
v3->dictionary1->dic[4] = "KP00";
v3->dictionary1->dic[5] = "eisp";
v3->dictionary1->dic[6] = "ssid";
v3->dictionary1->dic[7] = "User";
v3->dictionary1->dic[8] = "type";
v3->dictionary1->dic[9] = "site";
v3->dictionary1->dic[10] = "dish";
v3->dictionary1->dic[11] = "apps";
v3->dictionary1->dic[12] = "warn";
v3->dictionary1->dic[13] = "tone";
v3->dictionary1->dic[14] = "dogs";
v3->dictionary1->dic[15] = "cats";
v3->dictionary1->dic[16] = "size";
v3->dictionary1->dic[17] = "deep";
v3->dictionary1->dic[18] = "moon";
v3->dictionary1->dic[19] = "a109";
v3->dictionary1->dic[20] = "gi93";
v3->dictionary1->dic[21] = "imex";
v3->dictionary1->dic[22] = "imeh";
v3->dictionary1->dic[23] = "imea";
v3->dictionary1->dic[24] = "cook";
v3->dictionary1->dic[25] = "otp0";
v3->dictionary1->dic[26] = "otp1";
v3->dictionary1->dic[27] = "ages";
v3->dictionary1->dic[28] = "sole";
```

```
v3->dictionary1->dic[29] = "poem";
v3->dictionary1->dic[30] = "gets";
```

Figure 4: Filling the main dictionary

To establish communication with the C&C server, as illustrated in Figure 5, the malware must obtain the initial C&C addresses from the `CONFIGURATION_DATA` section. Upon decrypting these addresses, the malware initiates communication with the first layer of the C&C server through the `GetHtmlFromUrl` function, presumably using an HTTP GET request. The server responds with an HTML file containing the address of the second C&C server layer. Subsequently, the malware engages in communication with the second layer, employing the imported `GetImageFromUrl` function. The function name implies this performs a GET request to retrieve an image.

In this scenario, the attackers employ steganography to conceal crucial data for use in the next execution phase. Regrettably, we were unable to ascertain the nature of the important data concealed within the image received from the second layer of the C&C server.

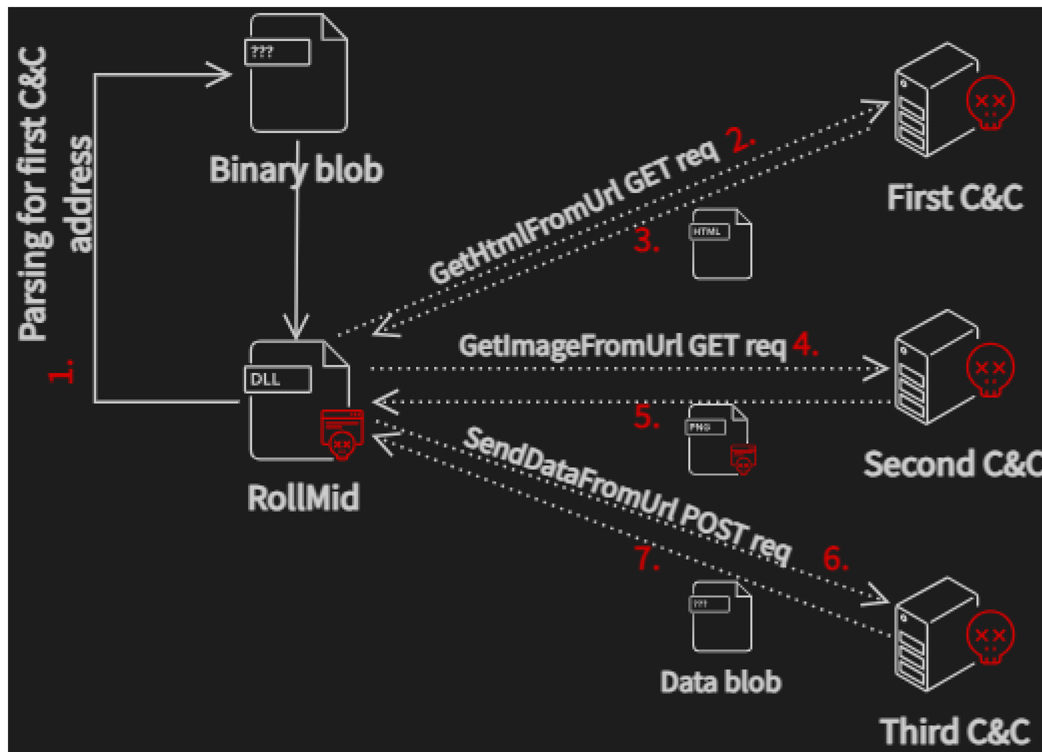

Figure 5: Communication with C&C servers

We are aware that the concealed data within the image serves as a parameter for a function responsible for transmitting data to the third C&C server. Through our analysis, we have determined that the acquired data from the image corresponds to another address of the third C&C server. Communication with the third C&C server is initiated with a POST request.

Malware authors strategically employ multiple C&C servers as part of their operational tactics to achieve specific objectives. In this case, the primary goal is to obtain an additional data blob from the third C&C server, as depicted in Figure 5, specifically in step 7. Furthermore, the use of different C&C servers and diverse communication pathways adds an additional layer of complexity for security tools attempting to monitor such activities. This complexity makes tracking and identifying malicious activities more challenging, as compared to scenarios where a single C&C server is employed.

The malware then constructs a URL, by creating the query string with GET parameters (name/value pairs). The parameter name consists of a randomly selected word from the previously created dictionary and the value is generated as a random string of two characters. The format is as follows:

"%addressOfThirdC&C%?%RandomWordFromDictonary%=%RandomString%"

The URL generation involves the selection of words from a generated dictionary, as opposed to entirely random strings. This intended choice aims to enhance the appearance and legitimacy of the URL. The words, carefully curated from the dictionary, contribute to the appearance of a clean and organized URL, resembling those commonly associated with authentic applications. The terms such as `"atype"`, `"User"`," or `"type"` are not arbitrary but rather thoughtfully chosen words from the created dictionary. By utilizing real words, the intention is to create a semblance of authenticity, making the HTTP `POST` payload appear more structured and in line with typical application interactions.

Before dispatching the `POST` request to the third layer of the C&C server, the request is populated with additional key-value tuples separated by standard delimiters "?" and "=" between the key and value. In this scenario, it includes:

%RandomWordFromDictonary %=%sleep_state_in_minutes%?
%size_of_configuration_data%

The data received from the third C&C server is parsed. The parsed data may contain an integer, describing sleep interval, or a data blob. This data blob is encoded using the base64 algorithm. After decoding the data blob, where the first 4 bytes indicate the size of the first part of the data blob, the remainder represents the second part of the data blob.

The first part of the data blob is appended to the `SECOND_ENCRYPTED_DLL` as an overlay, obtained from the binary blob. After successfully decrypting and decompressing `SECOND_ENCRYPTED_DLL`, the process involves preparing the `SECOND_ENCRYPTED_DLL`, which is a Remote Access Trojan (RAT) component to be loaded into memory and executed with the specific parameters.

The underlying motivation behind this maneuver remains shrouded in uncertainty. It appears that the attacker, by choosing this method, sought to inject a degree of sophistication or complexity into the process. However, from our perspective, this approach seems to border on overkill. We believe that a simpler method could have sufficed for passing the data blob to the `Kaolin` RAT.

The second part of the data blob, once decrypted and decompressed, is handed over to the `Kaolin` RAT component, while the `Kaolin` RAT is executed in memory. Notably, the decryption key and initialization vector for decrypting the second part of the data blob reside within its initial 32 bytes.

Kaolin RAT

A pivotal phase in orchestrating the attack involves the utilization of a Remote Access Trojan (RAT). As mentioned earlier, this `Kaolin` RAT is executed in memory and configured with specific parameters for proper functionality. It stands as a fully equipped tool, including file compression capabilities.

However, in our investigation, the `Kaolin` RAT does not mark the conclusion of the attack. In the previous blog post, we already introduced another significant component – the FudModule rootkit. Thanks to our robust telemetry, we can confidently assert that this rootkit was loaded by the aforementioned `Kaolin` RAT, showcasing its capabilities to seamlessly integrate and deploy `FudModule`. This layered progression underscores the complexity and sophistication of the overall attack strategy.

One of the important steps is establishing secure communication with the RAT's C&C server, encrypted using the AES encryption algorithm. Despite the unavailability of the binary containing the communication functionalities (the RAT also relies on functions imported from `FIRST_DLL_BINARY` for networking), our understanding is informed by other components in the attack chain, allowing us to make certain assumptions about the communication method.

The `Kaolin` RAT is loaded with six arguments, among which a key one is the base address of the network module DLL binary, previously also used in the `RollMid` loader. Another argument includes the configuration data from the second part of the received data blob.

For proper execution, the `Kaolin` RAT needs to parse this configuration data, which includes parameters such as:

- Duration of the sleep interval.
- A flag indicating whether to collect information about available disk drives.
- A flag indicating whether to retrieve a list of active sessions on the remote desktop.
- Addresses of additional C&C servers.

In addition, the `Kaolin` RAT must load specific functions from `FIRST_DLL_BINARY`, namely:

- `SendDataFromURL`
- `ZipFolder`
- `UnzipStr`
- `curl_global_cleanup`
- `curl_global_init`

Although the exact method by which the `Kaolin` RAT sends gathered information to the C&C server is not precisely known, the presence of exported functions like `"curl_global_cleanup"` and `"curl_global_init"` suggests that the sending process involves again API calls from the curl library.

For establishing communication, the `Kaolin` RAT begins by sending a `POST` request to the C&C server. In this first `POST` request, the malware constructs a URL containing the address of the C&C server. This URL generation algorithm is very similar to the one used in the `RollMid` loader. To the C&C address, the `Kaolin` RAT appends a randomly chosen word from the previously created dictionary (the same one as in the `RollMid` loader) along with a randomly generated string. The format of the URL is as follows:

`"%addressOfC&Cserver%?%RandomWordFromDictonary%=%RandomString%"`

The malware further populates the content of the `POST` request, utilizing the default `"application/x-www-form-urlencoded"` content type. The content of the `POST` request is subject to AES encryption and subsequently encoded with base64.

Within the encrypted content, which is appended to the key-value tuples (see the form below), the following data is included *`(EncryptedContent)`*:

- Installation path of the `RollFling` loader and path to the binary blob
- Data from the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\Iconservice
- `Kaolin` RAT process ID
- Product name and build number of the operating system.
- Addresses of C&C servers.
- Computer name
- Current directory

In the `POST` request with the encrypted content, the malware appends information about the generated key and initialization vector necessary for decrypting data on the backend. This is achieved by creating key-value tuples, separated by “&” and “=” between the key and value. In this case, it takes the following form:

%RandomWordFromDictonary%=%TEMP_DATA%&%RandomWordFromDictonary%=%IV%%KEY%&%RandomWordFromDictonary%=%EncryptedContent%&%RandomWordFromDictonary%=%EncryptedHostNameAndIPAddr%

Upon successfully establishing communication with the C&C server, the `Kaolin` RAT becomes prepared to receive commands. The received data is encrypted with the aforementioned generated key and initialization vector and requires decryption and parsing to execute a specific command within the RAT.

When the command is processed the `Kaolin` RAT relays back the results to the C&C server, encrypted with the same AES key and IV. This encrypted message may include an error message, collected information, and the outcome of the executed function.

The `Kaolin` RAT has the capability to execute a variety of commands, including:

- Updating the duration of the sleep interval.
- Listing files in a folder and gathering information about available disks.
- Updating, modifying, or deleting files.
- Changing a file's last write timestamp.
- Listing currently active processes and their associated modules.
- Creating or terminating processes.
- Executing commands using the command line.
- Updating or retrieving the internal configuration.
- Uploading a file to the C&C server.
- Connecting to the arbitrary host.
- Compressing files.
- Downloading a DLL file from C&C server and loading it in memory, potentially executing one of the following exported functions:
    - `_DoMyFunc`
    - `_DoMyFunc2`
    - `_DoMyThread (executes a thread)`
    - `_DoMyCommandWork`
- Setting the current directory.

Conclusion

Our investigation has revealed that the Lazarus group targeted individuals through fabricated job offers and employed a sophisticated toolset to achieve better persistence while bypassing security products. Thanks to our robust telemetry, we were able to uncover almost the entire attack chain, thoroughly analyzing each stage. The Lazarus group's level of technical sophistication was surprising and their approach to engaging with victims was equally troubling. It is evident that they invested significant resources in developing such a complex attack chain. What is certain is that Lazarus had to innovate continuously and

allocate enormous resources to research various aspects of Windows mitigations and security products. Their ability to adapt and evolve poses a significant challenge to cybersecurity efforts.

Indicators of Compromise (IoCs)

ISO
b8a4c1792ce2ec15611932437a4a1a7e43b7c3783870afebf6eae043bcfade30

RollFling
a3fe80540363ee2f1216ec3d01209d7c517f6e749004c91901494fb94852332b

NLS files
01ca7070bbe4bfa6254886f8599d6ce9537bafcbab6663f1f41bfc43f2ee370e
7248d66dea78a73b9b80b528d7e9f53bae7a77bad974ededeeb16c33b14b9c56

RollSling
e68ff1087c45a1711c3037dad427733ccb1211634d070b03cb3a3c7e836d210f
f47f78b5eef672e8e1bd0f26fb4aa699dec113d6225e2fcbd57129d6dada7def

RollMid
9a4bc647c09775ed633c134643d18a0be8f37c21afa3c0f8adf41e038695643e

Kaolin RAT
a75399f9492a8d2683d4406fa3e1320e84010b3affdff0b8f2444ac33ce3e690

Tagged asAPT, Lazarus, Recruiting scams
Share:XFacebook