WithSecure™ Intelligence Research, April 2024

# KAPEKA
# A novel backdoor spotted in Eastern Europe

by Mohammad Kazem Hassan Nejad

# Contents

# Executive summary

- WithSecure has uncovered a novel backdoor that has been used in attacks against victims in Eastern Europe since at least mid-2022.
- The malware, which we are calling “Kapeka”, is a flexible backdoor with all the necessary functionalities to serve as an early-stage toolkit for its operators, and also to provide long-term access to the victim estate.
- The malware’s victimology, infrequent sightings, and level of stealth and sophistication indicate APT-level activity.
- WithSecure discovered overlaps between Kapeka, GreyEnergy, and Prestige ransomware attacks which are all reportedly linked to a group known as Sandworm. WithSecure assesses it is likely that Kapeka is a new addition to Sandworm’s arsenal. Sandworm is a prolific Russian nation-state threat group operated by the Main Directorate of the General Staff of the Armed Forces of the Russian Federation (GRU). Sandworm is particularly notorious for its destructive attacks against Ukraine in pursuit of Russian interests in the region.
- Kapeka contains a dropper that will drop and launch a backdoor on a victim’s machine and then remove itself. The backdoor will first collect information and fingerprint both the machine and user before sending the details on to the threat actor. This allows tasks to be passed back to the machine or the backdoor’s configuration to be updated. WithSecure do not have insight as to how the Kapeka backdoor is propagated by Sandworm.
- Kapeka’s development and deployment likely follow the ongoing Russia-Ukraine conflict, with Kapeka being likely used in targeted attacks of firms across Central and Eastern Europe since the illegal invasion of Ukraine in 2022.
- It is likely that Kapeka was used in intrusions that led to the deployment of Prestige ransomware in late 2022.
- It is probable that Kapeka is a successor to GreyEnergy, which itself was likely a replacement for BlackEnergy in Sandworm’s arsenal.

# Background

In mid-2023 WithSecure found several artifacts observed in an intrusion set likely linked to Russian APT activity. One of these artifacts was an unknown backdoor/dropper detected in an Estonian logistics company in late 2022.

Upon analysis, we found two additional versions of the dropped backdoor submitted to VirusTotal from Ukraine in mid-2022 and mid-2023, one of which was packaged with a scheduled task file from an infected machine that launched the backdoor. We assessed with moderate confidence that the submitters were victims.

**Based on these sparse data points, several preliminary assessments were made:**

- No previous variants of the backdoor have been observed or publicly reported.
- The backdoor was rarely sighted, hence indicating that it has been used in limited scope attacks since at least mid-2022.
- Based upon victimology, the backdoor was likely used in campaigns specifically targeting victims in Eastern Europe.

Based on the rarity of the backdoor, its characteristics, and sightings in Eastern Europe, we made an initial assessment with low confidence that the backdoor, which we have dubbed “Kapeka” (‘little stork’ in Russian), is likely a bespoke tool used by an advanced persistent actor (APT) possibly of Russian origin in targeted attacks in Eastern Europe. This was later corroborated by Microsoft, who detect this malware as KnuckleTouch , and attribute it to Seashell Blizzard (better known as Sandworm). This is in-line with historical and current (including post 2022 Russian invasion of Ukraine) targeting and activities linked to Sandworm group, who are known to support the wider strategic objectives and changing intelligence requirements of the Russian state.

While examining the possible link between the backdoor and the Sandworm group, WithSecure noted overlaps between Kapeka and GreyEnergy, a toolkit thought to be associated with the Sandworm group. Additionally, we discovered connections between Kapeka, GreyEnergy, and Prestige ransomware attacks that occurred in late 2022.

[1] https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Win64/KnuckleTouch.A!dha

This report provides an in-depth technical analysis of the backdoor and its capabilities, and analyzes the connection between Kapeka and Sandworm group. The purpose of this report is to raise awareness amongst businesses, governments, and the broader security community. WithSecure has engaged governments and select customers with advanced copies of this report. In addition to the report, we are releasing several artifacts developed as a result of our research, including a registry-based & hardcoded configuration extractor, a script to decrypt and emulate the backdoor's network communication, and as might be expected, a list of indicators of compromise, YARA rules, and MITRE ATT&CK mapping.

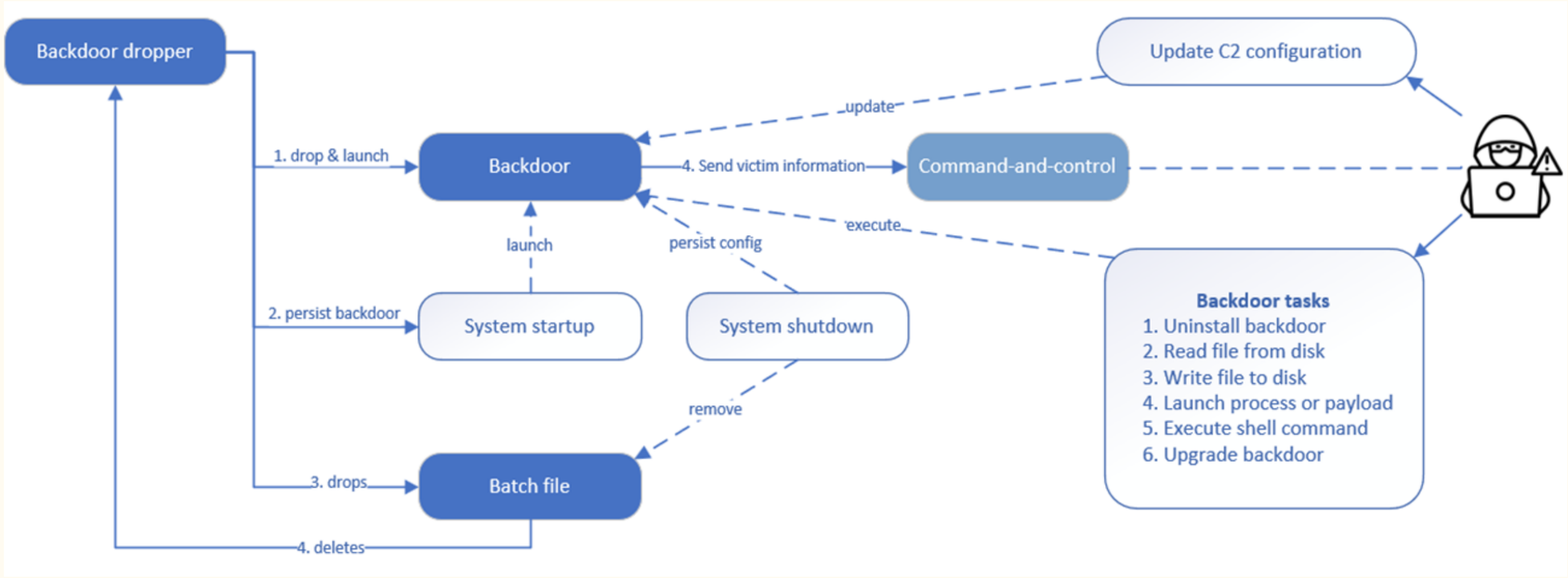



Figure 1. Overview of Kapeka

# Dropper analysis

The Kapeka dropper is a 32-bit Windows executable responsible for dropping, executing, and setting up persistence for the backdoor on a victim's machine as well as removing itself from disk. The backdoor binary, which is embedded within its resource section, is encrypted via AES-256. The dropper's resource section contains both 32-bit and 64-bit version of the backdoor and chooses the appropriate version depending on the victim machine's processor. The dropper utilizes an embedded key to decrypt the binary. However, if the embedded key is not set, then it defaults to using the command line string as the key for decryption. Figure 2 shows code snippet used to extract and decrypt the appropriate backdoor binary from the dropper's resource section.

Depending on the process privileges, the decrypted backdoor binary is dropped as a hidden file under a folder called **Microsoft** in either CSIDL_COMMON_APPDATA (if admin or SYSTEM) or CSIDL_LOCAL_APPDATA (if not). Note: CSIDL_COMMON_APPDATA is typically "C:\ProgramData" and CSIDL_LOCAL_APPDATA is typically "C:\Users\<username>\AppData\Local". The file name is 5-6 characters long and is randomly generated from consonants and vowels (to make it appear like a legitimate word) followed by a ".wll" extension. It is worth noting that the dropper looks for SensApi.dll (a legitimate Windows DLL) under system directory and modifies the file time attributes of the dropped backdoor binary to match the legitimate DLL by using SetFileTime().

```
v4 = (const WCHAR *)&embedded_aes_key;
if ( !embedded_aes_key )
  v4 = (const WCHAR *)cryptKeyHANDLE;
aes_key = v4;
v5 = 1;
fileContent = 0;
memset(&SystemInfo, 0, sizeof(SystemInfo));
GetNativeSystemInfo(&SystemInfo);
cryptKeyHANDLE = 0;
resource_name = (const WCHAR *)((SystemInfo.wProcessorArchitecture != 0) + 3);
ModuleHandle = GetModuleHandleExW(4u, (LPCWSTR)sub_4011F0, (HMODULE *)&cryptKeyHANDLE);
v8 = ModuleHandle ? cryptKeyHANDLE : 0;
ResourceW = FindResourceW((HMODULE)v8, resource_name, (LPCWSTR)10);
v10 = ResourceW;
if ( ResourceW )
{
  Resource = LoadResource((HMODULE)v8, ResourceW);
  if ( Resource )
  {
    phProv = (HCRYPTPROV)LockResource(Resource);
    if ( phProv )
    {
      v12 = SizeofResource((HMODULE)v8, v10);
      v13 = v12;
      if ( v12 )
      {
        v29 = v12;
        ProcessHeap = GetProcessHeap();
        v15 = (BYTE *)HeapAlloc(ProcessHeap, 8u, v29);
        pbData = v15;
        if ( v15 )
        {
          memmove_0(v15, (const void *)phProv, v13);
          fileContent = v13;
          phProv = 0;
          v16 = 0;
          if ( CryptAcquireContextW(&phProv, 0, 0, 0x18u, 0xF0000000) )
          {
            cryptKeyHANDLE = 0;
            if ( initialize_key(phProv, aes_key, (HCRYPTKEY *)&cryptKeyHANDLE) )
              v16 = CryptDecrypt((HCRYPTKEY)cryptKeyHANDLE, 0, 1, 0, pbData, &fileContent);
```

Figure 2. Code snippet to decrypt backdoor file from dropper's resource section

The dropper will then launch the backdoor binary by calling rundll32 and passing the backdoor's first export ordinal (#1) with a "-d" argument. Figure 3 shows an example of the command line used to launch the backdoor.

Command line:

```
"C:\WINDOWS\system32\rundll32.exe" "C:\ProgramData\Microsoft\hocite.wll", #1 -d
```

Figure 3. Example of dropper launching the backdoor

Depending on the process privileges, the dropper then sets persistence for the backdoor either as a scheduled task (if admin or SYSTEM) or autorun registry (if not). For the scheduled task, it creates a scheduled task called "Sens Api" via schtasks command, which is set to run upon system startup as SYSTEM. To establish persistence through the autorun utility, it adds an autorun entry called "Sens Api" under HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run via the 'reg add' command. Both persistence mechanisms are set to launch the binary by calling rundll32 and passing the backdoor's first export ordinal (#1) without any additional argument. Figure 4 shows code snippet used to create the appropriate persistence mechanism.

```
hkcu_string[0] = 0x4B0048;
v1 = 0;
hkcu_string[1] = 0x550043;                    // HKCU
v14 = 0;
wcscpy(String, L"%ws \\\"%ws\\\", #1");
if ( sub_4013C0((char **)&lpString) )
{
  v2 = lstrlenW(lpString);
  v3 = lstrlenW(v15) + v2;
  v4 = 2 * (v3 + lstrlenW(String)) + 2;
  if ( v4 )
  {
    v11 = v4;
    ProcessHeap = GetProcessHeap();
    fileName = (WCHAR *)HeapAlloc(ProcessHeap, 8u, v11);
    if ( fileName )
    {
      wsprintfW(fileName, String, lpString, v15);
      if ( is_user_admin() || is_user_system() )
      {
        v9 = lstrlenW(fileName);
        v8 = add_schtask((BOOL)fileName, 2 * v9 + 2);
      }
      else
      {
        v7 = lstrlenW(fileName);
        v8 = add_autorun((const WCHAR *)hkcu_string, (int)fileName, 2 * v7 + 2);
      }
      v1 = v8;
      HeapFreeWrapper(fileName);
    }
  }
  HeapFreeWrapper((LPVOID)lpString);
}
return v1;
}
```

Figure 4. Code snippet to add persistence

WithSecure identified a Kapeka scheduled task file in-the-wild from an infected machine. The scheduled task was called “OneDrive” instead of “Sens Api” that is created by the dropper binaries analyzed. Furthermore, the backdoor name (wslsrv) in this instance did not follow the same name generation method (using consonants and vowels) found in the dropper binaries analyzed and the command line to launch was slightly different. Figure 5 shows the execution command line seen in this instance versus an example from a scheduled task created by the dropper.

Lastly, the dropper will drop a hidden batch file into CSIDL_LOCAL_APPDATA and launch it, which will delete the dropper from disk. The file name is 3-4 characters long and is generated with the same name generation algorithm used for the backdoor. If the user is an administrator, the batch file will be set to be removed upon reboot by calling MoveFileExW() and setting dwFlags as MOVEFILE_DELAY_UNTIL_REBOOT and lpNewFileName as NULL. Figure 6 shows the file content of the dropped batch file.

**Kapeka scheduled task called "OneDrive" found in-the-wild**

```
<Exec>
  <Command>cmd</Command>
  <Arguments>/c start C:\Windows\system32\rundll32.exe C:\ProgramData\Microsoft\wslsrv.wll, #1</Arguments>
</Exec>
```

**Kapeka scheduled task called "Sens Api" created by dropper**

```
<Exec>
  <Command>C:\Windows\system32\rundll32.exe</Command>
  <Arguments>"C:\ProgramData\Microsoft\ladoza.wll", #1</Arguments>
</Exec>
```

Figure 5. Kapeka execution command from scheduled task seen in-the-wild called “OneDrive” versus “Sens Api” created by the dropper

```
1 @echo off
2 :label
3 del /q /f "<DROPPER_EXECUTABLE_PATH>"
4 if exist "<DROPPER_EXECUTABLE_PATH" goto label
5
```

Figure 6. File content of dropped batch script

# Backdoor analysis

The Kapeka backdoor is a Windows DLL containing one function which has been exported by ordinal  (rather than by name). The backdoor is written in C++ and compiled (linker 14.16) using Visual Studio 2017 (15.9). The backdoor file masquerades as a Microsoft Word Add-In with its extension (.wll), but in reality it is a DLL file.

The backdoor is meant to be executed with “-d” argument for its initial run, but without it for subsequent runs (which is achieved via the persistence method mentioned in earlier section “Dropper analysis”). The purpose of this flag is explained in subsequent sections.

Like many other backdoors, the backdoor implementation is multi-threaded, utilizing event objects  for data synchronization and signaling across threads.

**In total, the backdoor launches four main threads:**

- First thread: This is the primary thread which performs the initialization and exit routine, as well as C2 polling to receive tasks or an updated C2 configuration.
- Second thread: Monitors for Windows log off events, signaling the primary thread to perform the backdoor’s graceful exit routine upon log off.
- Third thread: Monitors for incoming tasks to be processed. This thread launches subsequent threads to execute each received task.
- Fourth thread: Monitors for completion of tasks to send back the processed task results to the C2.

[2] https://learn.microsoft.com/en-us/cpp/build/exporting-functions-from-a-dll-by-ordinal-rather-than-by-name
[3] https://learn.microsoft.com/en-us/windows/win32/sync/using-event-objects

In terms of data handling, the backdoor utilizes a large principal structure to hold all its subsequent data objects and structures, including thread/mutex/object handles. Furthermore, the backdoor utilizes JSON (implemented using ‘rapidjson’ library) to hold its data (such as C2 configuration and tasks received) internally as well as to send and receive information from its command-and-control server. In total there are 36 unique JSON keys which span over several JSON structures, which have been detailed in later sections. Each JSON key is obfuscated and 6-characters long. The obfuscated field names have not changed between the samples we have analyzed. Figure 7 shows examples of obfuscated JSON field names seen in the backdoor.

For encryption and encoding, the backdoor utilizes three separate methods throughout its execution, namely: AES-256 (CBC mode), XOR, and RSA-2048, with the RSA public key changing between samples.

## Backdoor configuration

The backdoor contains an embedded C2 configuration that is encrypted via AES-256. The configuration consists of a 32-byte key followed by an 8-byte padding and the encrypted configuration data. The configuration is decrypted during the backdoor’s initialization phase. The backdoor also reads any existing configuration that’s persisted in registry during its initialization phase. Depending on whether the backdoor is launched with the ‘-d’ argument and existing configuration in registry, the backdoor chooses which configuration to use. If ‘-d’ argument (which indicates first run) is provided, the backdoor will favor its embedded configuration, otherwise it will read existing configuration from registry, falling back to the embedded configuration if unavailable.

```
.rdata:0000000180023398 aJrczrx:
.rdata:0000000180023398
.rdata:0000000180023398                 text "UTF-16LE", 'jRcZrx',0
.rdata:00000001800233A6                 align 8
.rdata:00000001800233A8 aJxs2hz:
.rdata:00000001800233A8
.rdata:00000001800233A8                 text "UTF-16LE", 'jxs2HZ',0
.rdata:00000001800233B6                 align 8
.rdata:00000001800233B8 aLsml1j:
.rdata:00000001800233B8
.rdata:00000001800233B8                 text "UTF-16LE", 'LSmL1j',0
.rdata:00000001800233C6                 align 8
.rdata:00000001800233C8 aSiskba:
.rdata:00000001800233C8
.rdata:00000001800233C8                 text "UTF-16LE", 'SIsKba',0
.rdata:00000001800233D6                 align 8
.rdata:00000001800233D8 a3qy9vy:
.rdata:00000001800233D8
.rdata:00000001800233D8                 text "UTF-16LE", '3qY9vY',0
.rdata:00000001800233E6                 align 8
.rdata:00000001800233E8 a36d6mo:
.rdata:00000001800233E8
.rdata:00000001800233E8                 text "UTF-16LE", '36d6Mo',0
.rdata:00000001800233F6                 align 8
.rdata:00000001800233F8 aRzynkr:
.rdata:00000001800233F8
.rdata:00000001800233F8                 text "UTF-16LE", 'RzYnkr',0
.rdata:0000000180023406                 align 8
.rdata:0000000180023408 aKbxzsb:
.rdata:0000000180023408
.rdata:0000000180023408                 text "UTF-16LE", 'KBXZSb',0
.rdata:0000000180023416                 align 8
```

Figure 6. File content of dropped batch script

The backdoor persists its configuration via a registry value called “Seed” in “HKU\<SID>\Software\Microsoft\Cryptography\Providers\<GUID>\”. To generate the GUID value, the malware calls GetCurrentHwProfileW() and fetches the szHwProfileGuid field. In earlier versions of the backdoor, the malware would simply use the fetched value as GUID, however in the latest version of the backdoor we have analyzed the malware contains a custom algorithm implementing CRC32 and PRNG (pseudo-random number generator) operations applied to the GUID and a hardcoded value in the binary (described in a later section as “LSmL1j”) to generate a unique GUID. In all versions of the backdoor, the backdoor will default to a hardcoded GUID value (“0CA1BE92-FB73-BB74-5E41-00FDE76B2E8D”) if GetCurrentHwProfileW() fails. The backdoor uses the same algorithm to generate its mutex as “Global\BFE_Notify_Event_<GUID>”, but the fallback value is “{ad584834 - f1b9 - 1587 - 637b - 1e0025582179}” instead.

The persisted configuration is encrypted via AES-256 with a key consisting of 32-bytes of MachineGuid (UTF-16) value from HKLM\SOFTWARE\Microsoft\Cryptography, falling back to a hardcoded 32-byte key “Azbi3l1xIgcRzTsOHopgrwUdJUMWpOFt” if the registry key query fails. An example has been shown in figure 8.

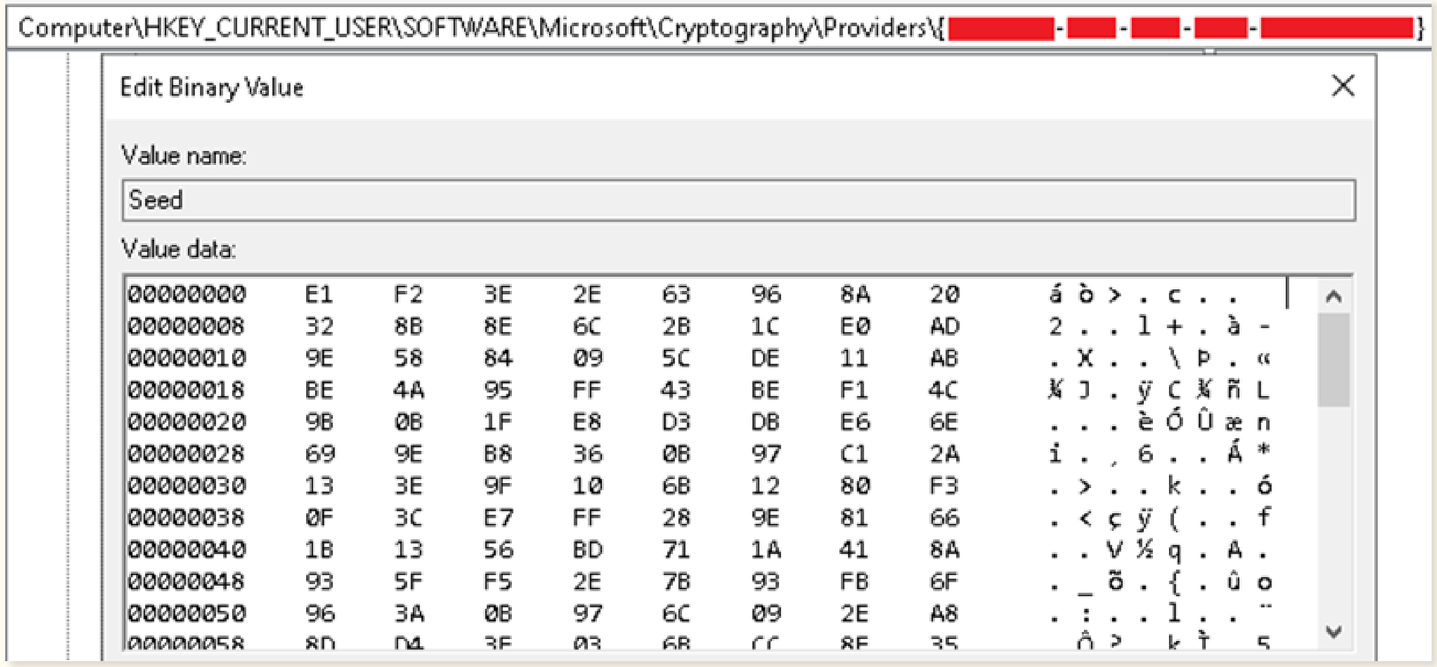



Figure 8. Example of encrypted configuration persisted in registry value “Seed”

Both the embedded and persisted configuration are encoded in JSON format. The C2 configuration JSON structure has been described in figure 9. An example of the C2 configuration is shown in figure 10.

| JSON Key | Value type | Value |
|---|---|---|
| GafpPS | Nested object | Holds the C2 configuration components mentioned below. |
| LsHsAO | Array | C2 Server URLs (required). This is the only mandatory field for the backdoor's embedded configuration. |
| hM4cDc | Integer | C2 polling interval (minutes) – The actual polling interval is randomized each time between the specified amount and next minute. If not present, the default amount is 10 minutes. |
| nLMNzt | Integer | Maximum alive time (days) – The maximum number of days the backdoor will try connecting to the C2 since its initialization or last successful C2 poll before uninstalling itself. If not present, the default amount is 3 days. |
| rggw8m | Nested object | Holds the system time structure  objects mentioned below. The values are generated & updated at runtime by the backdoor using GetSystemTimeAsFileTime(). This essentially keeps track of the backdoor's alive time and last successful C2 poll. This is included in the persisted configuration in registry. |
| bhpaLg | Integer | System time (Low-order part) |
| sEXtXs | Integer | System time (High-order part) |

Figure 9. C2 configuration JSON structure

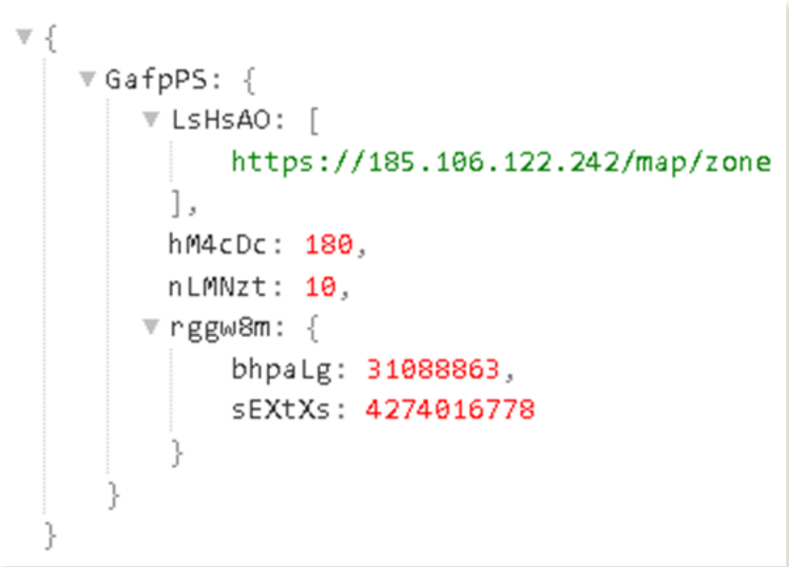



Figure 10. Example of C2 configuration

[4] https://learn.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-filetime

## Initial fingerprinting

During its initialization phase, the backdoor collects information about the victim's machine and user through a set of WinAPI calls and registry queries. This information is stored internally within a defined structure, which is later converted into a JSON format. The backdoor forwards this JSON blob in its first and subsequent communication with the threat actor's command-and-control server.

Figure 12 shows a complete list of information gathered from the victim's machine, collection method, and JSON key mapping. An example of JSON holding fingerprinted information is shown in figure 11.


Figure 11. Example of JSON holding collected information

| JSON Key | Value type | Value |
|---|---|---|
| KBXZSb | Username | GetUserNameW()<br>Overwritten by NetUserGetInfo() -> USER_INFO_1. usri1_name |
| Cwiq4j | User privileges | NetUserGetInfo() -> USER_INFO_1.usri1_priv |
| KKGCUr | Token elevation type | GetTokenInformation() -> TokenElevationType |
| arqSO1 | Computer name | NetWkstaGetInfo() -> WKSTA_INFO_100.wki100_computername |
| pHsy0J | Domain name | NetWkstaGetInfo() -> WKSTA_INFO_100.wki100_langroup |
| ozYekP | OS Major Version | NetWkstaGetInfo() -> WKSTA_INFO_100.wki100_ver_major |
| 8ORGRb | OS Minor Version | NetWkstaGetInfo() -> WKSTA_INFO_100.wki100_ver_minor |
| b0HqGu | ProductName | HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductName |
| xsRMVc | Processor Architecture | GetNativeSystemInfo() -> SYSTEM_INFO. wProcessorArchitecture |
| q200c6 | CSDVersion | HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\CSDVersion |
| RAJ5MJ | ProductId | HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductId |
| 7N4QJp | RegisteredOwner | HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\RegisteredOwner |
| tczMsk | RegisteredOrganization | HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\RegisteredOrganization |
| GQKkuo | UnknownFlag | GetVersionExW(&OSVERSIONINFOEXW)<br>This flag is set as 1 if the API call is successful. The exact reason is unknown, but it is likely an OS check. |
| Wqk8xK | Windows Server 2003 R2 Build Number | GetSystemMetrics(89) -> SM_SERVERR2<br>This can serve as a check whether the OS is Windows Server 2003 R2. |
| eEM2N9 | System Locale - Language | GetLocaleInfoW() -> LOCALE_SISO3166CTRYNAME |
| NPvllV | System Locale - Country | GetLocaleInfoW() -> LOCALE_SISO639LANGNAME |

Figure 12. Information collected, method, and JSON mapping.

## Network communication

The backdoor uses WinHttp 5.1 COM interface (winhttpcom.dll) to implement its network communication component. The backdoor communicates with its C2 to poll for tasks and to send back fingerprinted information and task results. The backdoor utilizes JSON to send and receive information from its C2.

Two separate threads are responsible for network communication, one to send fingerprinted information and poll for tasks, and another to send completed tasks results back to the C2. Both threads implement the same request/response handler. The request JSON structure has been described in figure 13. An example of C2 request JSON is shown figure 14.

| JSON Key | Value type | Value |
|---|---|---|
| jxs2HZ | Integer | Integer value distinguishing between the C2 poller (value 0) and response handler (value 1) thread. |
| LSmL1j | String | 16-byte hexadecimal string that's hardcoded inside the binary. The exact purpose of this string is unknown. However, it is most likely a form of campaign/build identifier. For instance, to distinguish between private RSA keys to use for decryption. |
| SIsKba | Nested object | This holds the fingerprinted information that has been mentioned in section "Initial fingerprinting". |
| jRcZrx | Nested object | This holds the output for each of the executed backdoor tasks. This is only populated when task results are available to be sent to the C2. This structure is described further in section "Backdoor tasks". |

Figure 13. C2 request JSON structure

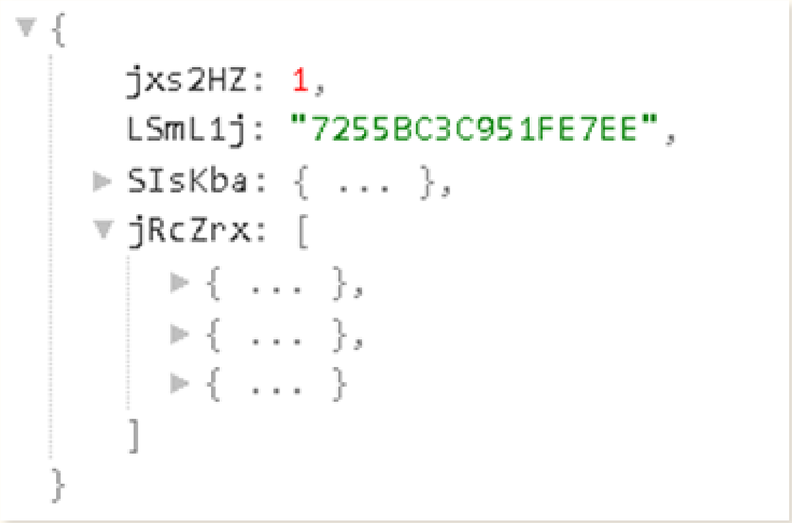



Figure 14. Example of C2 request JSON

The backdoor uses a custom structure to format the data (figure 13) it sends to its C2. Figure 15 shows an annotated example of the C2 request structure.

**The custom structure is generated via the following steps:**

- Generate a random 32-byte AES key that’s used to encrypt the JSON data to be sent.
- Encrypt the JSON data using the randomly generated AES key.
- Store encrypted data size formatted using htonl()
- Encrypt the 32-byte AES key with the RSA-2048 public key that’s embedded in the binary.
- Store encrypted key size formatted using htonl()
- Generate an arbitrary amount of random data.
- Arrange the data as:
  <SIZEOFENCRYPTEDKEY><ENCRYPTEDKEY><SIZEOFENCRYPTEDJSONDATA><ENCRYPTEDJSONDATA><APPENDEDRANDOMDATA>
- Generate a 4-byte XOR key.
- XOR the data structure using the generated XOR key.
- Prepend the XOR key to the data structure.
- Send the final data structure to the C2.

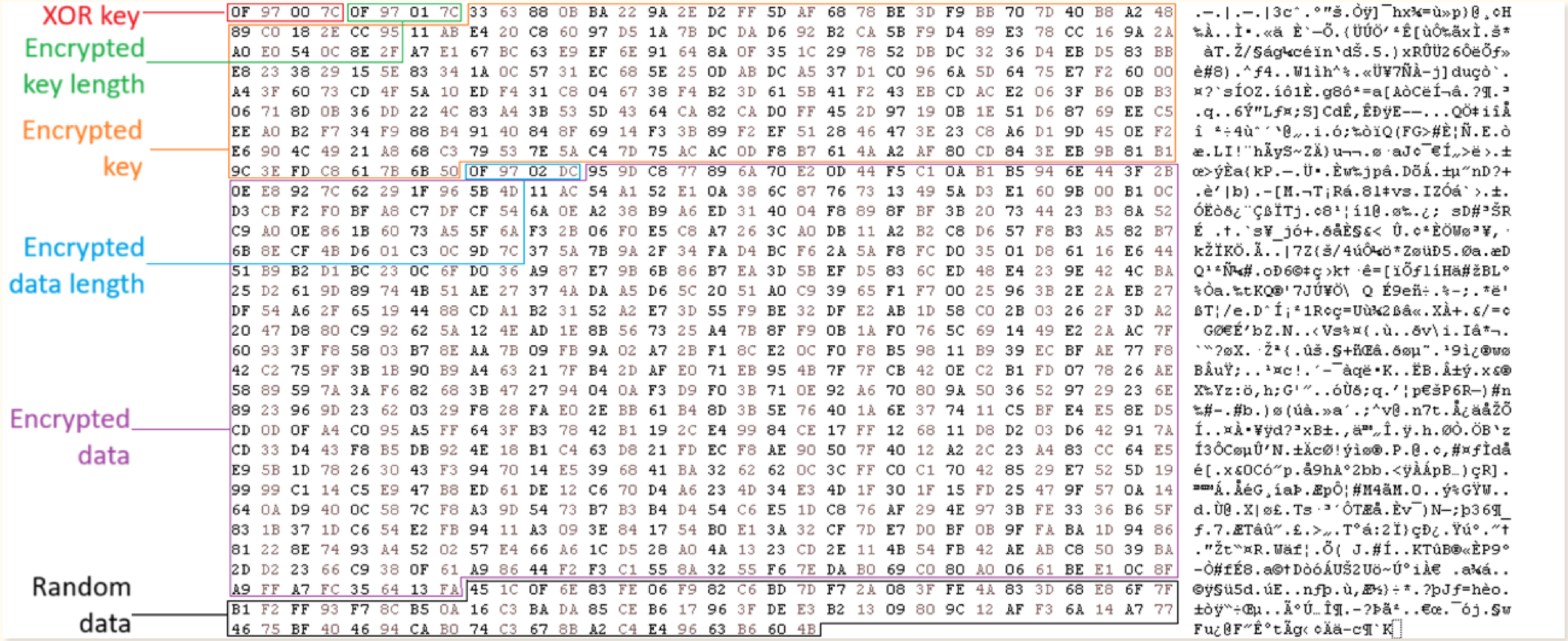



Figure 15. Example of annotated C2 request structure

The backdoor will re-use the same randomly generated 32-byte AES key to decrypt the response it receives. The backdoor can receive two types of responses, one to update its configuration and another to execute tasks, both of which have been described in later sections.

Moreover, the backdoor checks for internet proxy settings using WinHttpGetIEProxyConfigForCurrentUser() during its initialization phase and C2 polling. If a proxy setting exists, the backdoor will use the specified proxy server for its C2 communication. This functionality was only observed in the latest version of the backdoor analyzed.

Lastly, while the backdoor makes use of WinHttp 5.1 COM interface, we identified unused code snippets implementing XML HTTP 6.0 COM interface. Figure 16 shows a list of COM interfaces implemented by the backdoor, including XML HTTP 6.0 as well as WinHttp 5.1.

| Name | Function | Address | GUID | Object type | Module |
|---|---|---|---|---|---|
| IUnknown | sub_180006F70 | 0x180006f89 | 00000000-0000-0000-c000-000000000046 | interface | N/A |
| IDispatch | sub_180006F70 | 0x180006fb3 | 00020400-0000-0000-c000-000000000046 | interface | N/A |
| IXMLHTTPRequest | sub_1800070D0 | 0x18000721b | ed8c108d-4349-11d2-91a4-00c04f7969e8 | interface | N/A |
| XML HTTP 6.0 | sub_1800070D0 | 0x180007229 | 88d96a0a-f192-11d4-a65f-0040963251e5 | class | C:\Windows\System32\msxml6.dll |
| IWinHttpRequest | sub_1800076B0 | 0x18000782d | 016fe2ec-b2c8-45f8-b23b-39e53a75396b | interface | N/A |
| WinHttpRequest Component version 5.1 | sub_1800076B0 | 0x180007844 | 2087c2f4-2cef-4953-a8ab-66779b670495 | class | %SystemRoot%\system32\winhttpcom.dll |
| IUnknown | sub_1800076B0 | 0x180007933 | 00000000-0000-0000-c000-000000000046 | interface | N/A |
| IConnectionPointContainer | sub_1800076B0 | 0x18000794d | b196b284-bab4-101a-b69c-00aa00341d07 | interface | N/A |
| IWinHttpRequestEvents | sub_1800076B0 | 0x18000796b | f97f4e15-b787-4212-80d1-d380cbbf982e | interface | N/A |
| IWinHttpRequest | sub_180007FB0 | 0x1800080a8 | 016fe2ec-b2c8-45f8-b23b-39e53a75396b | interface | N/A |
| WinHttpRequest Component version 5.1 | sub_180007FB0 | 0x1800080be | 2087c2f4-2cef-4953-a8ab-66779b670495 | class | %SystemRoot%\system32\winhttpcom.dll |
| IConnectionPointContainer | sub_180007FB0 | 0x18000812c | b196b284-bab4-101a-b69c-00aa00341d07 | interface | N/A |
| IWinHttpRequestEvents | sub_180007FB0 | 0x180008158 | f97f4e15-b787-4212-80d1-d380cbbf982e | interface | N/A |
| IUnknown | sub_180008520 | 0x180008549 | 00000000-0000-0000-c000-000000000046 | interface | N/A |

Figure 16. COM interfaces implemented in backdoor, highlighting WinHttp 5.1 COM interface.

## Update C2 configuration

The backdoor can update its C2 configuration by receiving a new configuration as a JSON response (with key “GafpPS”) from its command-and-control server during polling. If the received configuration differs from the existing configuration, the backdoor will update its configuration on-the-fly as well as persist the latest C2 configuration by updating the registry value (“Seed”) that holds its configuration.

## Backdoor tasks

The backdoor can execute tasks on the victim’s machine by receiving a list of tasks as a JSON response (with key “Td7opP”) from its command-and-control server during polling, spawning a separate thread to execute each task.

Figure 17 shows the C2 response JSON structure that houses the tasks and data associated to each task. An example of C2 response JSON with received tasks is shown in figure 18.

| JSON Key | Value type | Value |
|---|---|---|
| Td7opP | Array | This holds a list of backdoor tasks to be executed on the victim's machine. Each task holds the key/value pairs mentioned below. |
| CwbJ4E | Integer | Command ID to execute (zero-based number). See figure 19 for full list of supported command IDs. |
| XVXLNm | String | **First argument**<br>Used mainly for file name/command line to read, write, launch. |
| INlB5x | Nested object | **Second argument**<br>Used for payload purposes, such as upgrading backdoor or writing a file to disk. This holds a key/value pair with the key being the filename and the value being the file content that's base64-encoded. |
| J8yWIG | String | Identifier string that can be passed to distinguish between executed commands, this is logged in the output sent back to the command-and-control as "3qY9vY". |

Figure 17. C2 response JSON structure


Figure 18. Example of C2 response JSON with received tasks

The backdoor supports all basic functionalities that allow it to operate as a flexible backdoor in the victim's estate. Figure 19 shows the list of commands supported by the backdoor.

| Command ID | Command | Required parameters |
|---|---|---|
| 0 | NotImplemented | - |
| 1 | Uninstall backdoor | - |
| 2 | Read file from disk | XVXLNm – File path to read |
| 3 | Write to file on disk | XVXLNm – File path to write<br>INlB5x – File content to write |
| 4 | Launch process or payload | XVXLNm – Command line to process & launch<br>INlB5x (optional) – Custom payload |
| 5 | Execute shell command | XVXLNm – Shell command to launch |
| 6 | Upgrade backdoor | - |
| Other | Return "unknown\n" | - |

Figure 19. Supported commands

Once the tasks are completed, the results are sent back to the command-and-control server in JSON format (under key "jRcZrx"). Figure 20 shows the JSON structure that houses the task results sent back to the command-and-control server. An example of C2 response containing task results is shown in figure 21.

| JSON Key | Value type | Value |
|---|---|---|
| jRcZrx | Array | This holds a list of results for executed tasks. Each result holds the key/value pairs mentioned below. |
| 3qY9vY | String | Identifier string that was passed in the command input as "J8yWIG" |
| 36d6Mo | String | Logged message during task execution |
| RzYnkr | Nested object | This holds a key/value pair that's used during read file task, with the key being the filename and the value being the file content that's base64-encoded. |

Figure 20. Completed task results JSON structure

```
{
  jxs2HZ: 1,
  LSmL1j: "7255BC3C951FE7EE",
▶ SIsKba: { ... },
▼ jRcZrx: [
  ▼ {
      3qY9vY: "Write to file",
      36d6Mo: "C:/Windows/secrets.txt EMPTY
      ",
    ▶ RzYnkr: { ... }
    },
  ▼ {
      3qY9vY: "Read from file",
      36d6Mo: "C:/Windows/secrets.txt OK
      ",
    ▼ RzYnkr: {
        C:/Windows/secrets.txt: "QXJlbid0IHlvdSBhIHNtYXJ0IGZlbGxhPw=="
      }
    },
  ▼ {
      3qY9vY: "Execute command",
      36d6Mo: "PID : li
      ---------------
      desktop-[redacted]\shawarma

      ---------------
      ExitCode : li
      ",
    ▶ RzYnkr: { ... }
    }
  ]
}
```


Figure 21. Example of C2 response containing task results

## Uninstall backdoor

This functionality removes all backdoor artifacts from the victim's machine by launching several shell commands combined via ampersands through the functionality "Launch process or payload" mentioned in a later section. Figure 22 shows an example of C2 response to uninstall the backdoor.


```
{
  Td7opP: [
    {
      J8yWIG: "Uninstall backdoor",
      CwbJ4E: 1,
      XVXLNm: "",
      INlB5X: ""
    }
  ]
}
```


Figure 22. Example of C2 response to uninstall the backdoor

**The set of commands include:**

- Sleeping for 10 seconds via ping command, allowing adequate time for the process to terminate before file deletion.
- Deleting persistence set by the dropper by issuing ‘schtasks delete’ or ‘reg delete’ command depending on process elevation. All analyzed backdoor samples deleted the same scheduled task/registry value called “Sens Api”.
- Deleting the backdoor executable via erase command with /f /q and /a:h flags.

The task thread will then set the main event object that’s used for synchronization across the process to a signaled state, which causes the backdoor to run its graceful exit routine. However, before doing so it sets a global flag that causes the exit routine to delete the registry key that persists the backdoor’s configuration on the victim’s machine. Figure 23 shows the usage of SHDeleteKeyW() by Kapeka to delete its persisted configuration.

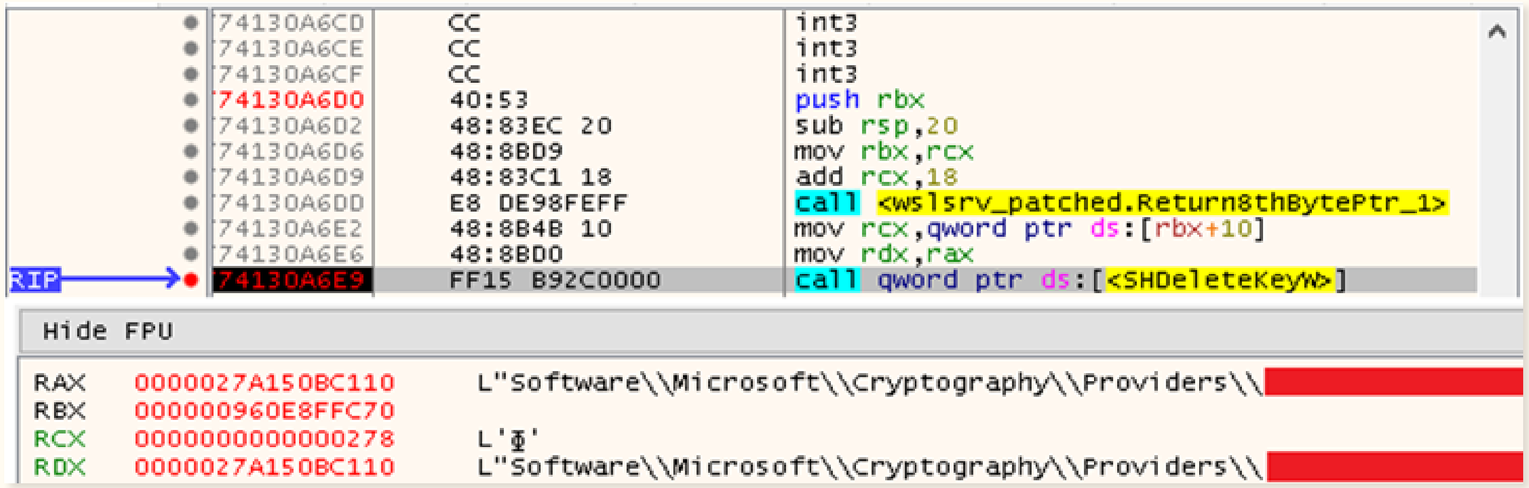
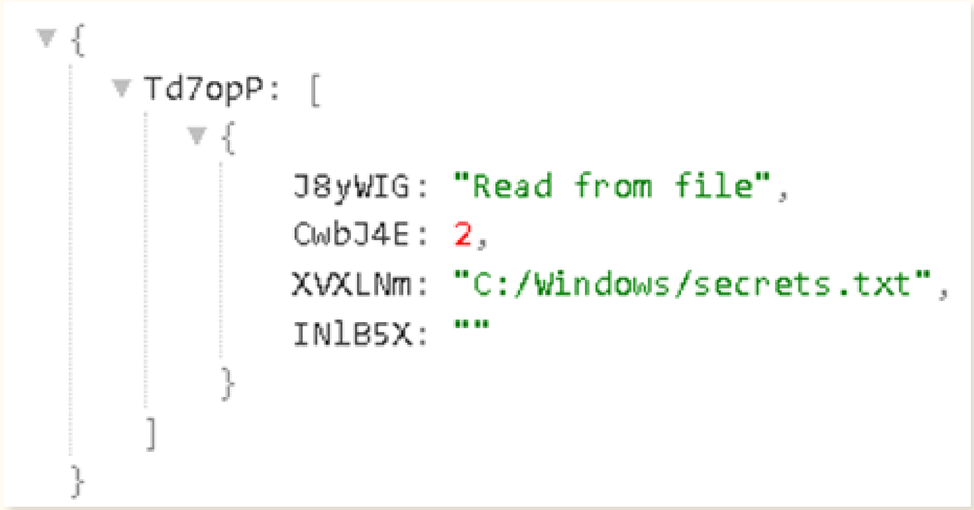



Figure 23. Kapeka removing its persisted configuration

## Read file from disk

This functionality reads any file that's below 50 MB from disk and sends the output back to the C2, essentially enabling data collection from the victim's machine. The file to be read is specified under the first argument ("XVXLNm"). Figure 24 shows an example of C2 response to read a file from the victim's machine.



```
{
  Td7opP: [
    {
      J8yWIG: "Read from file",
      CwbJ4E: 2,
      XVXLNm: "C:/Windows/secrets.txt",
      INlB5X: ""
    }
  ]
}
```

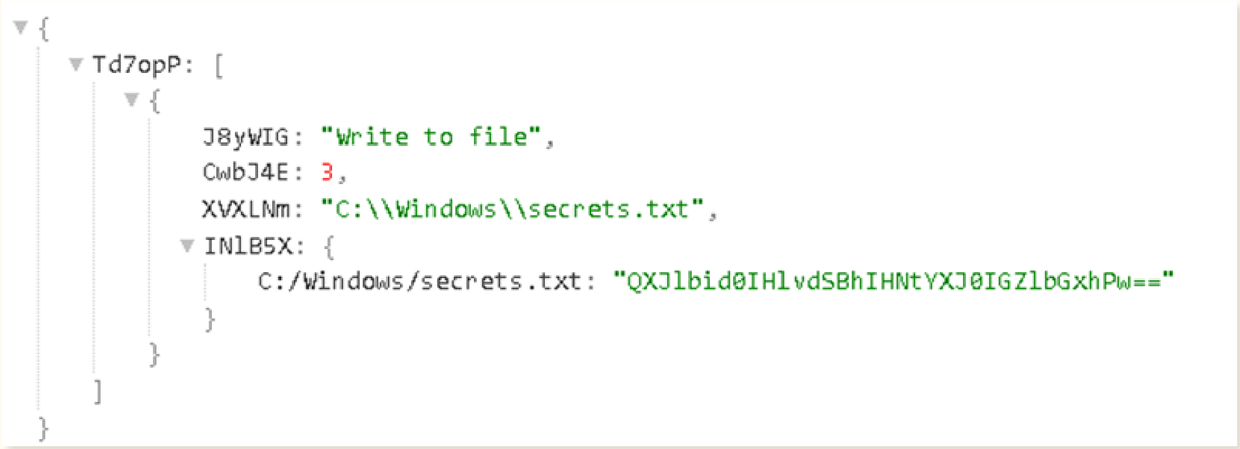

Figure 24. Example of C2 response to read from file

If the operation succeeds, it logs a success message "<filename> OK\n" and sends back the file content as key/value pair underneath a key called "RzYnkr", with the key being the filename, and value being the base64-encoded file content.

In case of an error, the error message is fetched via GetLastError() and logged in an error message "ERROR <LastError>". If the file size is above 50 MB, the file size is logged in an error message as "ERROR: File too large. [<size> > 50 MB]".

## Write file to disk

This functionality writes any file content passed (under the second argument, "INlB5x") into the desired file path (under the first argument, "XVXLNm") on the victim's machine. Figure 25 shows an example of C2 response to write a file onto the victim's machine.



```
{
  Td7opP: [
    {
      J8yWIG: "Write to file",
      CwbJ4E: 3,
      XVXLNm: "C:\\Windows\\secrets.txt",
      INlB5X: {
        C:/Windows/secrets.txt: "QXJlbid0IHlvdSBhIHNtYXJ0IGZlbGxhPw=="
      }
    }
  ]
}
```


Figure 25. Example of C2 response to write to file

The threat actor can pass “-f” parameter alongside the file path in the first argument to forcefully create the respective file path, for instance, if the file directory does not already exist.

If the file path does already exist, the malware negates the READ_ONLY file attribute of the file before writing the file content to ensure a successful file operation. If the file write operation is successful, then a success message is logged as “<filename> OK \n” otherwise an error message is logged as “<filename> FAIL\n”. If the passed content is empty, then an error message “<filename> EMPTY\n” is returned.

## Launch process or payload

This functionality launches a new process as a specified command line (under the first argument, “XVXLNm”), essentially allowing any arbitrary executable on disk to be executed. The first argument is parsed as command line arguments (white-space delimited tokens) to extract the executable file to be launched and any arguments passed.

**Additionally, the first argument can contain multiple additional parameters that alters the backdoor’s behavior, namely:**

- Waiting for the child process in 100 millisecond intervals. This is specified via “-w” argument and it can take a parameter to specify the number of minutes to wait. For instance, “-w=1” would cause the backdoor process to wait for 1 minute, unless the launched child process exits sooner.
- Log output and error messages from launched child process (and all its subsequent subprocesses). This is specified via “-o” argument. To achieve this, the standard input, error, and output are redirected via anonymous pipes.
- File path to write custom payload into. This is specified via “-f” and its functionality is explained further below.
- An unused parameter “-bc”. The functionality of this parameter is unknown.

These additional parameters are not passed into the child process command line that’s launched. Figure 26 shows an example of C2 response to launch a process.


Figure 26. Example of C2 response to launch process

This functionality also supports execution of custom payloads. To do so, the payload must be passed through the second argument (“INlB5x“).

**The backdoor will write the payload to disk before execution and there are two ways the backdoor will determine the file path to write the payload into:**

- If “-f” parameter was specified in the first argument, it will parse the specified file path passed (white-delimited parameter following -f).
- If “-f” parameter was not specified and/or file path was not provided, then it will generate a temporary file name with a “00” prefix (via GetTempFileNameW()) under temporary folder (via GetTempPathW()).

This functionality essentially makes the backdoor modular by allowing additional modules to be dropped and executed. Figure 27 shows an example of C2 response to launch a custom payload on the victim’s machine.

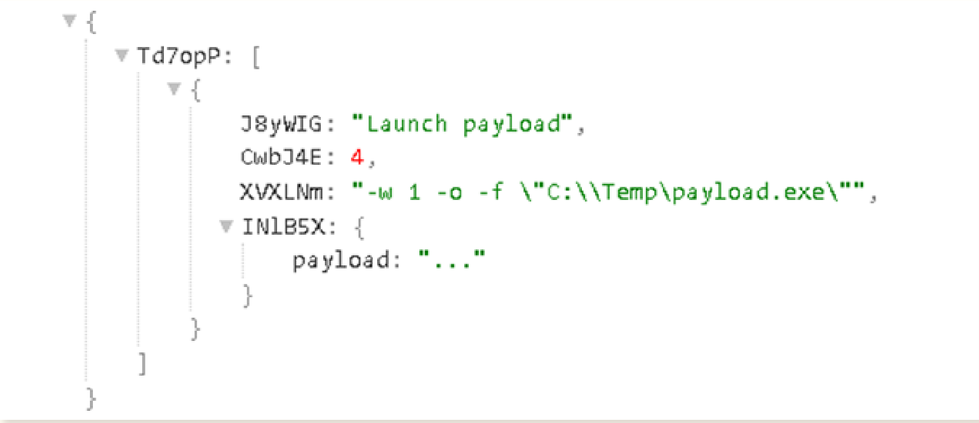



Figure 27. Example of C2 response to launch custom payload

This functionality also supports execution of custom payloads. To do so, the payload must be passed through the second argument (“INlB5x“).

**The backdoor will write the payload to disk before execution and there are two ways the backdoor will determine the file path to write the payload into:**

- If “-f” parameter was specified in the first argument, it will parse the specified file path passed (white-delimited parameter following -f).
- If “-f” parameter was not specified and/or file path was not provided, then it will generate a temporary file name with a “00” prefix (via GetTempFileNameW()) under temporary folder (via GetTempPathW()).

This functionality essentially makes the backdoor modular by allowing additional modules to be dropped and executed. Figure 27 shows an example of C2 response to launch a custom payload on the victim’s machine.

To launch the process, the backdoor will combine the specified executable file and list of arguments into a string and call CreateProcessW() passing the string as a command line. If the process was launched successfully, then a success message is logged as “PID : <PID>\n”. If the wait flag was set, the backdoor will wait for the specified amount of time or until the child process exits. If the output flag was set, it will log the output/error received from the child process(es) as “----------\n<OUTPUT/ERROR>”.

If the time out is reached and the child process is still running, it will forcefully terminate the child process and all its subsequent child processes and log “\n----------\nTerminateProcess\n”, otherwise if the child process had already exited, it will log the exit code as “\n----------\nExitCode : <exitcode>\n”.

**Additionally, there are five error messages that the backdoor will log within this functionality, namely:**

- If the payload can't be written to disk, it will log "1: <filename> <LastError>\n"
- If the standard output pipe can't be created, it will log "2: 0"
- If the standard input pipe can't be created, it will log "3: 0"
- If the standard error pipe can't be created, it will log "4: 0"
- If process creation fails, it will log "5: 0".

## Execute shell command

This functionality executes any shell command specified under the first argument ("XVXLNm") by using the functionality "Launch process or payload" mentioned in an earlier section and passing "-w" and "-o" parameters to wait and log the received process output. Figure 28 shows an example of C2 response to execute a shell command on the victim's machine.

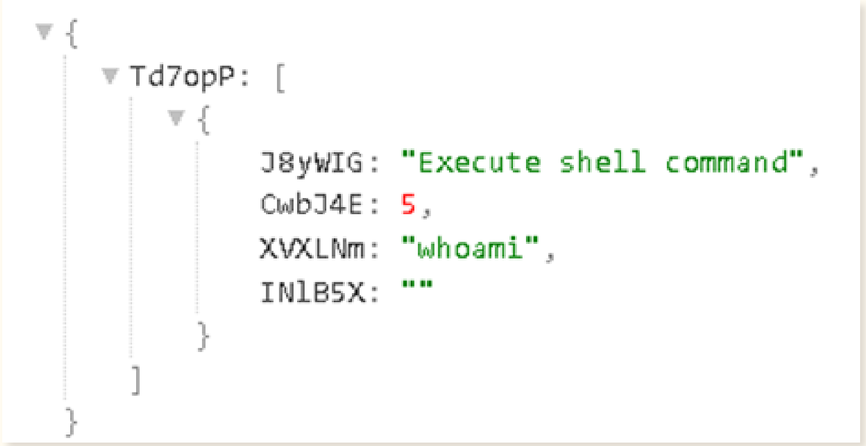



Figure 28. Example of C2 response to execute shell command

## Upgrade backdoor

This functionality allows the backdoor to upgrade itself by passing a newer version under the second argument ("INlB5x"). Figure 29 shows an example of C2 response to upgrade the backdoor.


Figure 29. Example of C2 response to upgrade backdoor

The backdoor will rename the existing backdoor binary by adding ".old" extension using MoveFileExW() function. It will drop the new backdoor binary on disk using the existing backdoor's file path. It will then re-use the file attributes and file time attributes of the old backdoor on the newly created backdoor binary.

It will then launch the new backdoor binary in the same fashion as the dropper would, that is by calling rundll32 and passing the backdoor's first export ordinal (#1) with a "-d" argument, essentially launching the upgraded binary with the initial run flag.

If the backdoor binary is launched successfully, it will log a success message "PID : <NewProcessId>\n".

**Otherwise, there are three error messages that the backdoor can log, namely:**

- If the second argument is empty (i.e. no file content passed), it will log "1\n".
- If the old backdoor binary could not be moved, it will log "2: <LastError>\n"
- If the new binary could not be created, it will log "3: <LastError>\n"

The task thread will then set the main event object that's used for synchronization across the process to a signaled state in a similar fashion explained under section "Uninstall backdoor".

It is worth noting that this functionality was only observed in the latest version of the analyzed backdoor. This version also spawns a thread upon launch to delete the old version of the backdoor (with the .old extension), given that "-d" argument was passed into it (which is typically the case during the backdoor's first execution). To achieve this, the backdoor tries several methods, firstly removing the file's READ_ONLY attribute (if this attribute exists). It then attempts to delete the file using DeleteFileW and if that fails, it retries 45 more times within a loop that contains a 1 second sleep between retries. As a final resort, the file will be set to be removed upon reboot by calling MoveFileExW()and setting dwFlags as MOVEFILE_DELAY_UNTIL_REBOOT and lpNewFileName as NULL, a method seen in the Kapeka dropper as well (described in section "Dropper analysis"). Figure 30 shows code snippet of file deletion method used by the backdoor to remove older version of itself.

```
bool __fastcall RemoveFileWrapper(__int64 filePath, unsigned int secondsToSleep)
{
  const WCHAR *v4; // rax
  DWORD FileAttributesW; // ebx
  const WCHAR *v6; // rax
  unsigned int v7; // ebx
  const WCHAR *v8; // rax
  const WCHAR *v10; // rax
  const WCHAR *v12; // rax

  v4 = (const WCHAR *)Return8thBytePtr_1(filePath);
  FileAttributesW = GetFileAttributesW(v4);
  if ( FileAttributesW != -1 && (FileAttributesW & 1) != 0 )
  {
    v6 = (const WCHAR *)Return8thBytePtr_1(filePath);
    SetFileAttributesW(v6, FileAttributesW & 0xFFFFFFFE);
  }
  v7 = secondsToSleep / 1000;
  v8 = (const WCHAR *)Return8thBytePtr_1(filePath);
  if ( !DeleteFileW(v8) )
  {
    do
    {
      if ( !v7-- )
        break;
      Sleep(1000u);
      v10 = (const WCHAR *)Return8thBytePtr_1(filePath);
    }
    while ( !DeleteFileW(v10) );
  }
  if ( v7 == -1 )
  {
    v12 = (const WCHAR *)Return8thBytePtr_1(filePath);
    return MoveFileExW(v12, 0i64, 4u);
  }
  else
  {
    return 1;
  }
}
```

Figure 30. Code snippet used to remove old backdoor

This functionality could potentially allow the threat actor to first infect victims with a skeleton version of the backdoor in order to fingerprint them and only drop a more complete version of the backdoor if the victim is deemed an appropriate target.

### Other behavior

The dropper and the backdoor implement stackstrings to obfuscate some of the strings used in the malware. Figure 31 shows examples of stackstrings seen in the backdoor.

FLOSS STACK STRINGS (24)

| Function | Function Offset | Frame Offset | String |
|---|---|---|---|
| 0x180006054 | 0x415d300f | 0x50 | text/xml |
| 0x180006054 | 0x415d300f | 0x38 | Content-Type |
| 0x1800065ac | 0x415c500f | 0x80 | Content-Type |
| 0x1800065ac | 0x415c500f | 0x60 | application/octet-stream |
| 0x1800089bc | 0x415c300f | 0x1b0 | Global |
| 0x1800089bc | 0x415c300f | 0x1a0 | BFE_Notify_Event_ |
| 0x1800089bc | 0x415c300f | 0x170 | {ad584834 - f1b9 - 1587 - 637b - 1e0025582179} |
| 0x180009224 | 0x180008364 | 0x248 | erase /f /q  /a:h |
| 0x180009224 | 0x180008364 | 0x220 | C:\Windows\System32\cmd.exe |
| 0x180009224 | 0x180008364 | 0x1e8 | ping localhost -n 10 -w 1000 |
| 0x180009224 | 0x180008364 | 0x1a0 | schtasks /delete /tn \"Sens Api\" /f |
| 0x180009224 | 0x180008364 | 0x150 | reg delete HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run /v \"Sens Api\" /f |
| 0x18000a4b0 | 0x180003698 | 0x48 | %SYSTEMROOT%\system32\cmd.exe |
| 0x18001041c | 0x18000366c | 0x240 | SOFTWARE\Microsoft\Windows NT\CurrentVersion |
| 0x18001041c | 0x18000366c | 0x1b0 | ProductName |
| 0x18001041c | 0x18000366c | 0x1c8 | CSDVersion |
| 0x18001041c | 0x18000366c | 0x1e0 | ProductId |
| 0x18001041c | 0x18000366c | 0x198 | RegisteredOwner |
| 0x18001041c | 0x18000366c | 0x178 | RegisteredOrganization |
| 0x18001454c | 0x415c300f | 0x1e0 | Seed |
| 0x18001454c | 0x415c300f | 0x1d0 | Software\Microsoft\Cryptography\Providers |
| 0x18001454c | 0x415c300f | 0x170 | {0CA1BE92-FB73-BB74-5E41-00FDE76B2E8D} |
| 0x180014770 | 0x18000366c | 0xc0 | MachineGuid |
| 0x180014770 | 0x18000366c | 0xa0 | Software\Microsoft\Cryptography |

Figure 31. Stackstrings in the backdoor

[5] https://learn.microsoft.com/en-us/windows/win32/shutdown/logging-off

Before initialization, the backdoor sleeps for an arbitrary amount of time using WaitForSingleObjectEx() and waitable timer.

The backdoor monitors for log off events by monitoring for WM_QUERYENDSESSION messages through a Window procedure callback that's created in a separate thread. Figure 32 shows the implemented callback function. If this message is received, the thread will set the main event object that's used for synchronization across the process to a signaled state, causing the backdoor to run its exit routine. The only noteworthy function of the exit routine is its ability to persist the backdoor's current state (C2 configuration, tasks, and task results) into the registry value ("Seed"), which houses the backdoor's configuration on the victim machine. This retains the latest state of the backdoor so that it can be re-processed once the machine is restarted, and the backdoor is re-launched. It is worth noting that the backdoor also sets its process shutdown parameters as SHUTDOWN_NORETRY during its initialization phase, ensuring that it does not become a blocking process during a log off in order to remain stealthy.

```
LRESULT __fastcall WindowProcCallback(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
  LONG_PTR WindowLongPtrW; // rax
  void *main_event_handle; // rcx

  if ( Msg != 17 )                                  // WM_QUERYENDSESSION
    return DefWindowProcW(hWnd, Msg, wParam, lParam);
  WindowLongPtrW = GetWindowLongPtrW(hWnd, 0xFFFFFFEB);
  if ( lParam )
  {
    if ( (lParam & 0x80000000) != 0 )               // ENDSESSION_LOGOFF
    {
      main_event_handle = *(void **)(WindowLongPtrW + 8);
      goto signal_exit;
    }
    return DefWindowProcW(hWnd, Msg, wParam, lParam);
  }
  main_event_handle = *(void **)WindowLongPtrW;
signal_exit:
  if ( main_event_handle )
    SetEvent(main_event_handle);
  return 1i64;
}
```

Figure 32. Implemented callback function to monitor log off events

# Sandworm attribution analysis

To determine the origin and goal of Kapeka, we examined the possible link established between Kapeka and Sandworm group. Based on publicly available reporting, the closest toolkit WithSecure found that shared similarities with Kapeka was GreyEnergy. In this section, we will highlight some of the similarities and lay several propositions to encourage further research. Information regarding GreyEnergy referenced throughout this section are based on reports from ESET , Trellix (FireEye) , and Nozomi Networks .

GreyEnergy is a modular backdoor thought to be part of Sandworm's arsenal, with GreyEnergy itself being regarded as a likely successor to the BlackEnergy toolkit that the threat group was initially known for utilizing in their early attacks. At a high-level, GreyEnergy consists of a dropper component that is responsible for dropping and executing the GreyEnergy backdoor, as well as setting up the backdoor's persistence and removing itself from disk. Two versions of the GreyEnergy toolkit have been identified, the main GreyEnergy backdoor and a lighter version known as GreyEnergy "mini".

**There are some conceptual overlaps between Kapeka and GreyEnergy, namely:**

- Both toolkits consist of a dropper component that has the main backdoor embedded within. The dropper component is responsible for dropping & setting up the backdoor's persistence, then removing itself from disk. However, the GreyEnergy dropper is packed, while the Kapeka dropper is not.
- The GreyEnergy mini and Kapeka backdoors are DLL files with a masqueraded extension to make them appear legitimate, with GreyEnergy mini using ".db" and Kapeka using ".wll". Both backdoors are also dropped into a folder named "Microsoft" in the file directory with the parent directory commonly being C:\ProgramData.
- Both backdoor DLLs are exported and called by the first ordinal (#1) via rundll32. This is an uncommon yet not unique method of exporting DLLs.
- Both droppers look for a legitimate Windows DLL on disk and set the dropped backdoor's file time to the same as that DLL. GreyEnergy also modifies the file description of the backdoor, while Kapeka doesn't.
- GreyEnergy and Kapeka use a similar custom algorithm to structure data that's sent to their C2. Both generate a unique AES-256 key per communication to encrypt the data that's to be sent. The AES key is then encrypted via an embedded RSA-2048 key. In each case the encrypted key and its length as well as the encrypted data and its length are structured in a similar format, though there are some subtle differences. Kapeka XOR encodes the data and appends random data, while GreyEnergy encodes the data via base64. Figure 33 shows a comparison between the two custom structures.
- The GreyEnergy dropper with service DLL persistence looks for an appropriate Windows service to mimic and names the dropped backdoor DLL with a randomly generated four-character name followed by either 'srv' or 'svc'. One Kapeka backdoor sample we found in-the-wild that was bundled with a scheduled task from an infected machine (97e0e161d673925e42cdf04763e7eaa53035338b) was called 'wslsrv.dll'. This naming convention did not follow the algorithm found in the droppers we analyzed. Furthermore, the scheduled task name was 'OneDrive' (instead of Sens Api) and the command line was slightly different. It is plausible that a different dropper which shares some high-level similarities with the GreyEnergy dropper may have been used.
- The dropper component in GreyEnergy checks and creates mutex based on the GUID value fetched via GetCurrentHwProfileA, as does the backdoor component in Kapeka. Utilizing GetCurrentHwProfileA() to generate a mutex value is not a common technique in other threats we have observed.

- Some configuration components of Kapeka also match GreyEnergy. Both utilize a configuration structure that holds a defined maximum alive time and a pair of fields that hold the high and low order part of system time. The maximum alive time defines the maximum number of days with no successful C2 connection before the backdoors will remove themselves. The system time pair is generated & updated at runtime by the backdoor and used to keep track of the backdoor's alive time and last successful C2 poll. This specific implementation is not a common technique in other threats we have observed. Moreover, both backdoors also contain a 16-byte hexadecimal string that is likely some form of identifier. Figure 34 compares the hexadecimal strings found in GreyEnergy and Kapeka. Furthermore, figure 35 shows the mentioned similarity between the configuration components of Kapeka and GreyEnergy.
- Kapeka utilizes obfuscated names in its configuration to make analysis more difficult, as do some versions of GreyEnergy. Figure 36 shows a comparison of obfuscated field names seen in GreyEnergy as well as Kapeka's configuration.

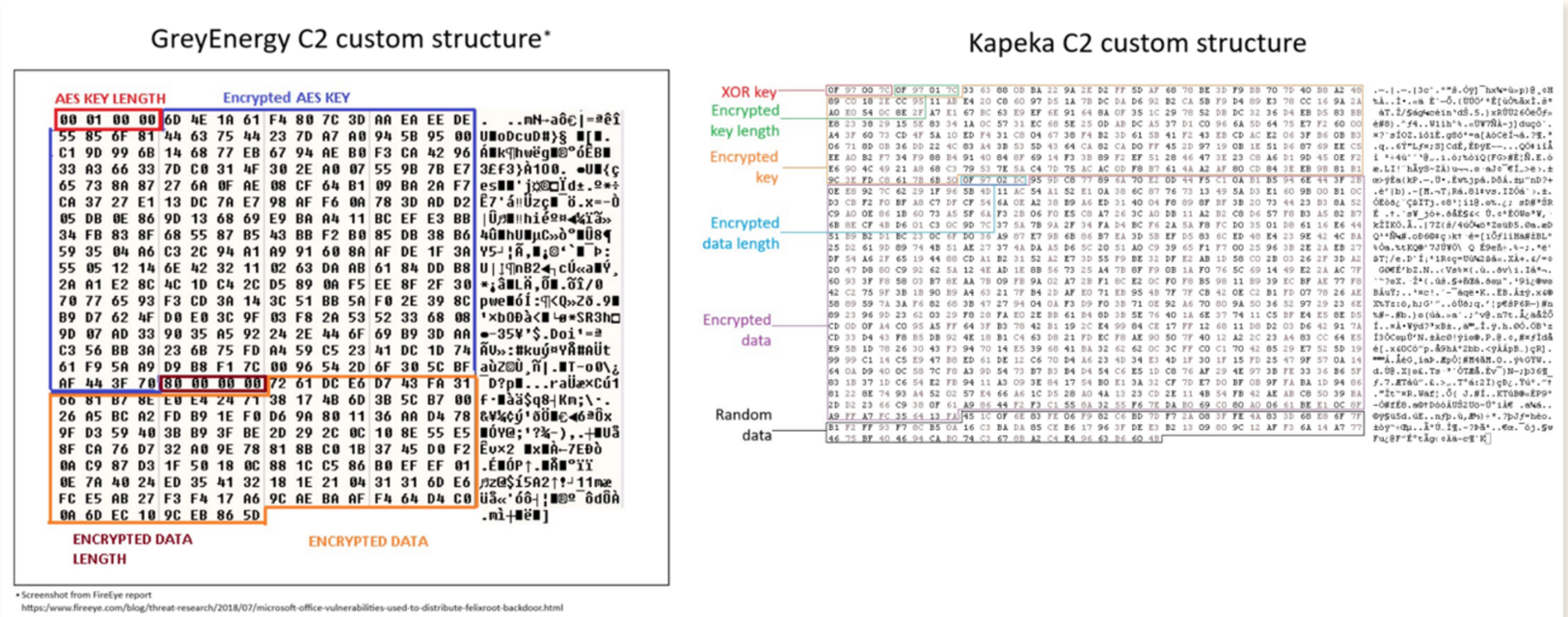



Figure 33. Comparison between GreyEnergy and Kapeka's C2 custom structure

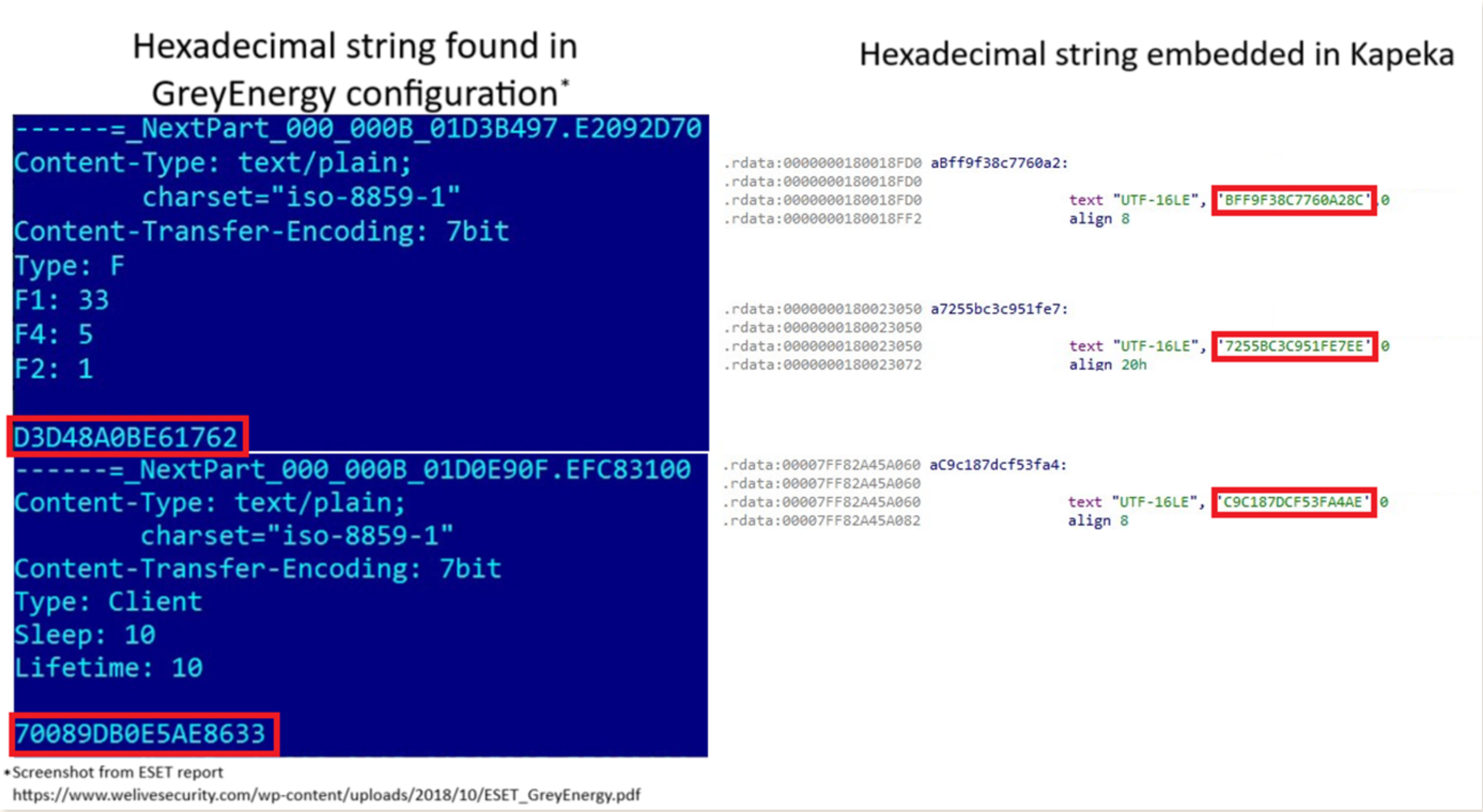



Figure 34. Example of hexadecimal strings found in GreyEnergy and Kapeka samples

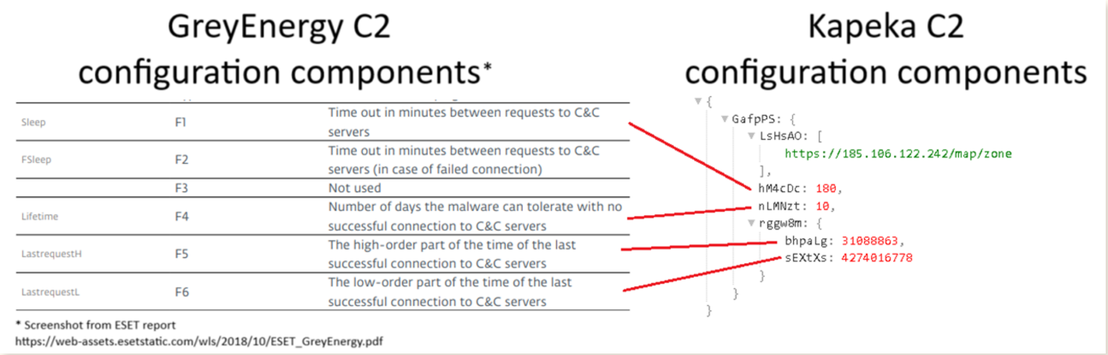



Figure 35. Similarities between GreyEnergy and Kapeka's C2 configuration

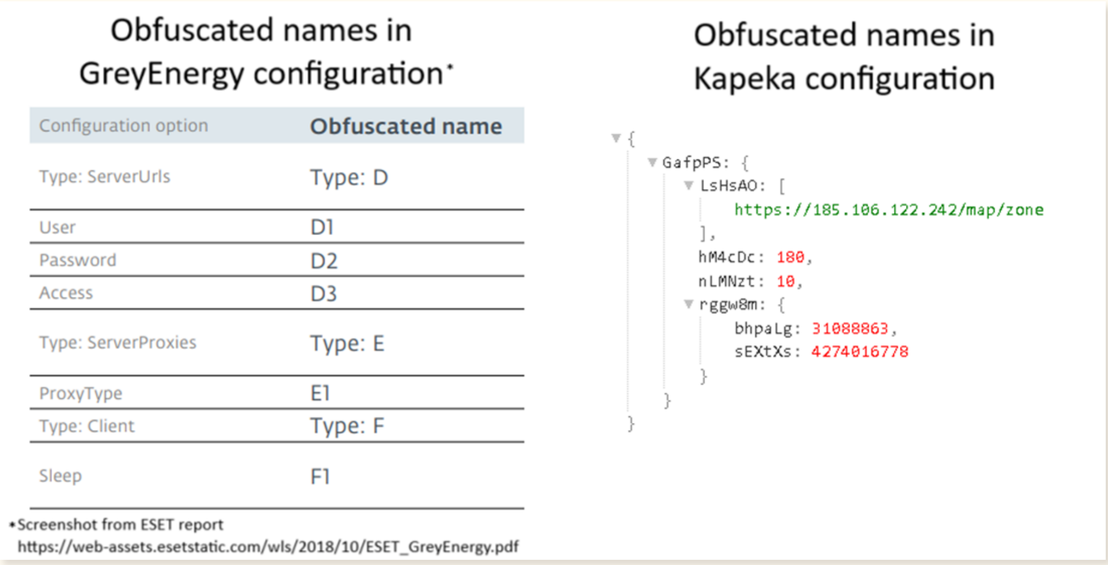



Figure 36. Example of obfuscated field names found in GreyEnergy and Kapeka's configuration

**While there are similarities between the two, there are also differences such as, but not limited to:**

- The backdoor commands and their implementations are vastly different.
- Kapeka persists its C2 configuration via registry, while GreyEnergy does so via a file on-disk.
- GreyEnergy utilizes WMI to fingerprint the victim, while Kapeka utilizes Windows API and registry.
- For persistence, GreyEnergy mini utilizes a shortcut file via Startup folder, GreyEnergy utilizes Windows service via ServiceDLL registry, while Kapeka utilizes either autorun registry or scheduled task.

Beyond functional similarity between the two toolkits, we examined other indicators relating to Kapeka, GreyEnergy, and Sandworm.

While we did not observe any post-compromise activity following the detection of Kapeka in our upstream due to limited telemetry, Kapeka has been reportedly used in destructive attacks including ransomware campaigns. We correlated publicly reported incidents temporally that were ransomware-related and attributed to Sandworm group within the same time frame, and we observed some overlaps with attacks leading to the deployment of Prestige ransomware.

It had been reported that Prestige ransomware was used by Sandworm in destructive attacks against transportation and logistics companies in Ukraine and Poland in October 2022, with an increase in precursor activity in September 2022 . The victim organization in which we observed Kapeka was also a logistics company in Eastern Europe, the backdoor was spotted in late September 2022, and the other Kapeka samples found in-the-wild were observed in Ukraine. Separately, the geographical targeting of Prestige ransomware and GreyEnergy overlap as well, as both were reportedly used in Ukraine and Poland. GreyEnergy has also been observed as a precursor in destructive attacks.

[9] https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Win64/KnuckleTouch.A!dha
[10] https://www.microsoft.com/en-us/security/blog/2022/10/14/new-prestige-ransomware-impacts-organizations-in-ukraine-and-poland/
[11] https://blogs.microsoft.com/on-the-issues/2022/12/03/preparing-russian-cyber-offensive-ukraine/

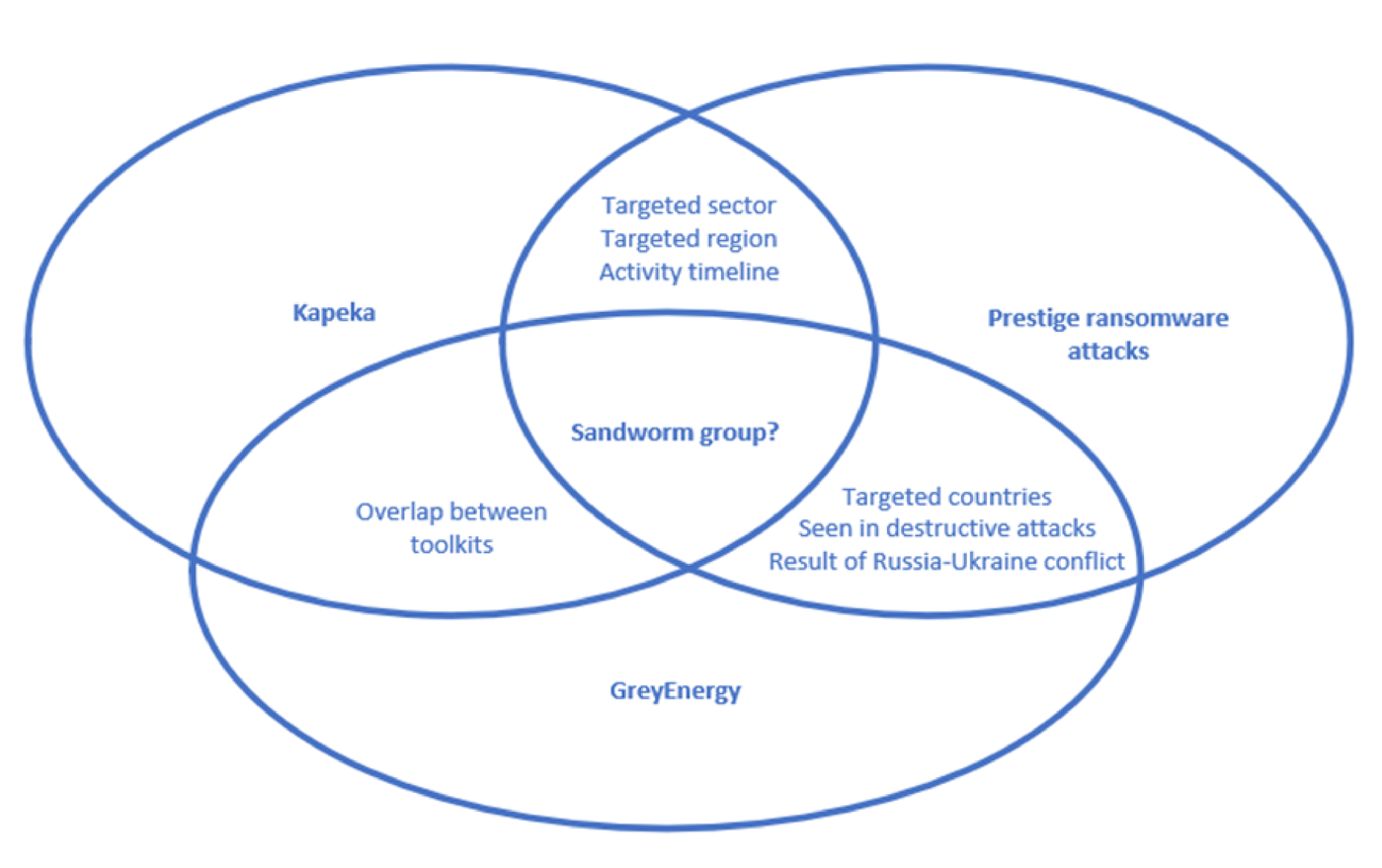



Figure 37. Overlaps between Kapeka, GreyEnergy, Prestige ransomware attacks.

Figure 37 summarizes our findings related to Kapeka, GreyEnergy, and Prestige ransomware attacks that are all reportedly linked to Sandworm group. We do not believe these findings are substantial enough to form a conclusive assessment or attribution.

**However we believe several non-competing hypotheses can be proposed that lay the foundation for further research:**

- Kapeka is part of Sandworm's latest arsenal, serving as a flexible backdoor likely used as part of wider espionage campaigns to support intelligence collection that can also lead to sabotage operations at later stages, including ransomware attacks.
- Kapeka was likely used in intrusions that led to the deployment of Prestige ransomware in late 2022.
- The toolkit is developed and employed as part of the ongoing Russia-Ukraine conflict, with targets mostly in Eastern and Central Europe.
- Kapeka is a successor to GreyEnergy backdoor, as GreyEnergy is considered a successor to BlackEnergy. The lowered sophistication observed from BlackEnergy to GreyEnergy can be witnessed from GreyEnergy to Kapeka as well.

# Conclusion

Kapeka is a previously unreported backdoor that has been sporadically spotted in Eastern Europe since at least mid-2022. It is a flexible backdoor with all the necessary functionalities to serve as an early-stage toolkit for its operators, and also to provide long-term access to the victim estate.

The backdoor's victimology, infrequent sightings, and level of stealth and sophistication indicate APT-level activity, highly likely of Russian origin. However, due to sparsity of data at the time of writing the infection vector, the threat actor, and the actor's 'actions on objectives' cannot be conclusively stated. Nevertheless, we examined multiple data points that strongly suggests a link between Kapeka and Sandworm.

Sandworm is a prolific Russian nation-state threat group notorious for their destructive attacks against Ukraine in pursuit of Russian interests. Based on overlaps in functionality we have noted between GreyEnergy (a toolkit thought to be part of Sandworm's arsenal) and Kapeka, as well as the latest events publicly attributed to Sandworm since the 2022 Russian invasion of Ukraine, we hypothesize Kapeka is a new addition to Sandworm's arsenal. It was likely used in intrusions that led to the deployment of Prestige ransomware in late 2022. It is probable that Kapeka is a replacement for GreyEnergy, which itself was likely a replacement for BlackEnergy in Sandworm's arsenal. Kapeka's development and deployment likely follows the ongoing Russia-Ukraine conflict, with Kapeka being likely used in targeted attacks across Central and Eastern Europe ever since the illegal invasion of Ukraine in 2022.

WithSecure last observed Kapeka in May 2023. It is uncommon for threat groups, especially nation-state, to cease operations or dispose tooling altogether, particularly before they are publicly documented. Therefore, Kapeka's infrequent sightings can be a testament for its meticulous usage by an advanced persistent actor (APT) in operations that span over years, such as the Russia-Ukraine conflict. It remains to be seen whether the developers and operators of Kapeka will evolve with newer versions of the tool or develop and use a new toolkit with threads of similarity to Kapeka (such as conceptual overlaps or code re-use) like those found between Kapeka and GreyEnergy, as well as GreyEnergy and BlackEnergy. Regardless of Kapeka's origin and objectives, the threat of the backdoor as documented in this report remains the same.

While the backdoor and its dropper contain capabilities to remove all traces of compromise, WithSecure has identified several infection artifacts and developed several scripts to aid with analysis and detection, which can be found in the appendix section of this report.

# Appendices

## MITRE ATT&CK Mapping

| Tactic | Technique | Description |
|---|---|---|
| Execution | Command and Scripting Interpreter: Windows Command Shell | Kapeka uses batch script files and Windows shell commands for various purposes. |
| | Inter-Process Communication: Component Object Model | Kapeka uses WinHttp 5.1 COM interface to implement its network communication. |
| Persistence | Scheduled Task/Job: Scheduled Task | Kapeka creates a scheduled task called “Sens Api” or “OneDrive” for persistence. |
| | Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder | Kapeka creates an autorun registry called “Sens Api” for persistence. |
| Defense evasion | Masquerading: Masquerade File Type | Kapeka masquerades its backdoor file as a Microsoft Word Add-In with its extension (.wll), but in reality it is a DLL file |
| | Obfuscated Files or Information | Kapeka obfuscates some of its plaintext strings as stackstrings. The embedded backdoor and its configuration are also AES-256 encrypted. |
| | Obfuscated Files or Information: Embedded Payloads | The dropper embeds the main backdoor binary in its resource section. |
| | Hide Artifacts: Hidden Files and Directories | The dropper drops the main backdoor and removal batch script as hidden files on the victim’s machine. |
| | Indicator Removal: File Deletion | The dropper will remove itself upon execution and the main backdoor can remove itself as well. |
| | Indicator Removal: Clear Persistence | The backdoor can remove its own persistence. |
| | Modify Registry | The backdoor persists its configuration via registry. |
| | System Binary Proxy Execution: Rundll32 | Kapeka utilizes rundll32 to execute its main backdoor. |
| | Data Obfuscation: Junk Data | Kapeka adds junk data to the data it sends to its C2. |
| | Virtualization/Sandbox Evasion: Time Based Evasion | The backdoor sleeps for an arbitrary amount of time using WaitForSingleObjectEx() and waitable timer before initialization. |

| Tactic | Technique | Description |
|---|---|---|
| **Discovery** | System Time Discovery | The backdoor keeps track of its last successful connection to its C2 by using system time. |
| | System Owner/User Discovery | The backdoor collects information about the user and organization through a set of WinAPI calls and registry queries. |
| | System Information Discovery | The backdoor collections various information about the system through a set of WinAPI calls and registry queries. |
| | System Language Discovery | The backdoor queries language and country by using GetLocaleInfoW() API call. |
| | Query Registry | The backdoor steals information about the victim and the system via registry queries. |
| **Command and Control** | Ingress Tool Transfer | The backdoor can receive and execute additional payloads. |
| | Exfiltration Over C2 Channel | The backdoor can exfiltrate fingerprinted information as well as local files from the victim's machine over to its C2. |
| | Encrypted Channel: Asymmetric Cryptography | The backdoor uses RSA-2048 encryption as part of its custom algorithm to encrypt data sent to its C2. |
| | Encrypted Channel: Symmetric Cryptography | The backdoor uses AES-256 and XOR operations as part of its custom algorithm to encrypt data sent to its C2. |
| | Proxy: Internal Proxy | The backdoor detects internet proxy settings via WinHttpGetIEProxyConfigForCurrentUser() and uses them if available. |

## Scripts

**WithSecure has developed several scripts to aid with the analysis and detection of Kapeka, namely:**

- A script to decrypt and emulate Kapeka's network communication. This has been implemented as a custom HTTP handler for fakenet [https://github.com/mandiant/flare-fakenet-ng].
- A script to extract Kapeka's configuration from either registry or embedded within the backdoor binary.
- A script to extract and decrypt the backdoor binary from the dropper's resource section.

These can be found in WithSecure Lab's GitHub [https://github.com/WithSecureLabs/iocs/tree/master/Kapeka].

# Detection opportunities

## WithSecure Elements

WithSecure™ Elements Endpoint Protection detects multiple stages of the attack lifecycle.

**Our products currently offer the following detections against the threat:**

- Backdoor:W64/Kapeka.*
- Trojan:BAT/Naida.*
- Trojan-Dropper:W32/Klavdia.*

## YARA rules

YARA rules can be found in WithSecure Lab's GitHub [https://github.com/WithSecureLabs/iocs/tree/master/Kapeka/].

## Indicators of compromise (IOCs)

Indicators of compromise can be found in WithSecure Lab's GitHub [https://github.com/WithSecureLabs/iocs/tree/master/Kapeka/].

| Type | Value | Note | Seen in | Seen on |
|---|---|---|---|---|
| Filename | crdss.exe | Backdoor dropper file name | Ukraine | June 2022 |
| Filename | %SYSTEM%\win32log.exe | Backdoor dropper file name | Estonia | September 2022 |
| SHA1 | 80fb042b4a563efe058a71a647ea949148a56c7c | Backdoor dropper hash | Ukraine | June 2022 |
| SHA1 | 5d9c189160423b2e6a079bec8638b7e187aebd37 | Backdoor dropper hash | Estonia | September 2022 |
| SHA1 | 6c3441b5a4d3d39e9695d176b0e83a2c55fe5b4e | Backdoor hash | Estonia | September 2022 |
| SHA1 | 97e0e161d673925e42cdf04763e7eaa53035338b | Backdoor hash | Ukraine | May 2023 |
| SHA1 | 9bbde40cab30916b42e59208fbcc09affef525c1 | Backdoor hash | Ukraine | June 2022 |
| URL | https[:]//103[.]78[.]122[.]94/help/healthcheck | Backdoor C2 address | - | - |
| URL | https[:]//88[.]80[.]148[.]65/news/article | Backdoor C2 address | - | - |
| URL | https[:]//185[.]181[.]229[.]102/home/info | Backdoor C2 address | - | - |
| URL | https[:]//185[.]38[.]150[.]8/star/key | Backdoor C2 address | - | - |

# Who We Are

WithSecure™, formerly F-Secure Business, is cyber security's reliable partner. IT service providers, MSSPs and businesses – along with the largest financial institutions, manufacturers, and thousands of the world's most advanced communications and technology providers – trust us for outcome-based cyber security that protects and enables their operations. Our AI-driven protection secures endpoints and cloud collaboration, and our intelligent detection and response are powered by experts who identify business risks by proactively hunting for threats and confronting live attacks. Our consultants partner with enterprises and tech challengers to build resilience through evidence-based security advice. With more than 30 years of experience in building technology that meets business objectives, we've built our portfolio to grow with our partners through flexible commercial models.

WithSecure™ Corporation was founded in 1988, and is listed on NASDAQ OMX Helsinki Ltd.

WITH secure | Intelligence