# Waterbear Returns, Uses API Hooking to Evade Security

trendmicro.com/en_us/research/19/l/waterbear-is-back-uses-api-hooking-to-evade-security-product-detection.html

December 11, 2019


Waterbear, which has been around for several years, is a campaign that uses modular malware capable of including additional functions remotely. It is associated with the cyberespionage group BlackTech, which mainly targets technology companies and government agencies in East Asia (specifically Taiwan, and in some instances, Japan and Hong Kong) and is responsible for some infamous campaigns such as PLEAD and Shrouded Crossbow. In previous campaigns, we've seen Waterbear primarily being used for lateral movement, decrypting and triggering payloads with its loader component. In most cases, the payloads are backdoors that are able to receive and load additional modules. However, in one of its recent campaigns, we've discovered a piece of Waterbear payload with a brand-new purpose: hiding its network behaviors from a specific security product by API hooking techniques. In our analysis, we have discovered that the security vendor is APAC-based, which is consistent with BlackTech's targeted countries.

Knowing which specific APIs to hook might mean that the attackers are familiar with how certain security products gather information on their clients' endpoints and networks. And since the API hooking shellcode adopts a generic approach, a similar code snippet might be used to target other products in the future and make Waterbear harder to detect.

## A closer look at Waterbear

Waterbear employs a modular approach to its malware. It utilizes a DLL loader to decrypt and execute an RC4-encrypted payload. Typically, the payload is the first-stage backdoor which receives and loads other executables from external attackers. These first-stage backdoors can be divided into two types: First, to connect to a command-and-control (C&C) server, and second, to listen in on a specific port. Sometimes, the hardcoded file paths of the encrypted payloads are not under Windows native directories (e.g., under security products or third-party libraries' directories), which may indicate that the attackers might have prior knowledge of their targets' environments. It is also possible that the attackers use Waterbear as a secondary payload to help maintain presence after gaining some levels of access to the targets' systems. The evidence is that Waterbear frequently uses internal IPs as its own C&C servers (for instance, b9f3a3b9452a396c3ba0ce4a644dd2b7f494905e820e7b1c6dca2fdcce069361 uses an internal IP address of 10[.]0[.]0[.]211 as its C&C server).

## The typical Waterbear infection chain

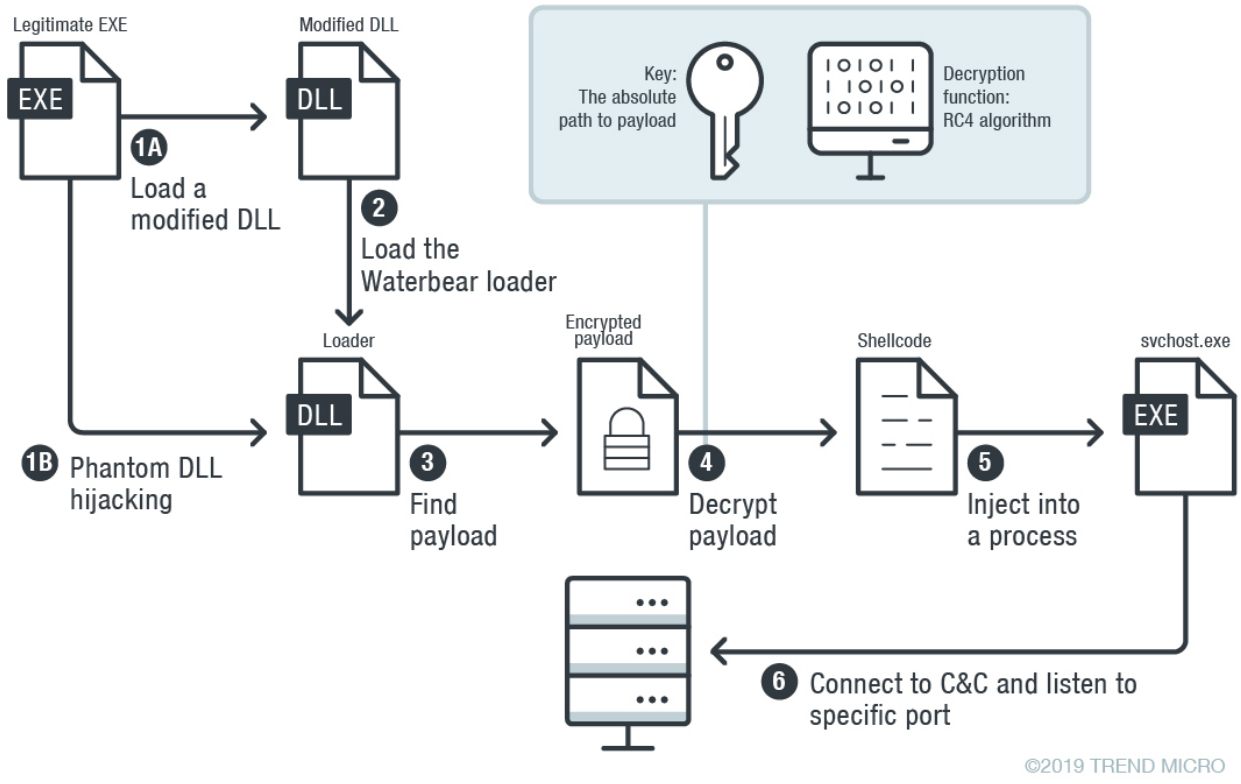



Figure 1. A typical Waterbear infection chain

A Waterbear infection starts from a malicious DLL loader, as shown in Figure 1. We have seen two techniques of DLL loader triggering. One is modifying a legitimate server application to import and load the malicious DLL loader, while the second technique is performing phantom DLL hijacking and DLL side loading. Some Windows services try to load external DLLs with hardcoded DLL names or paths during boot-up. However, if the DLL is a legacy DLL (i.e., one that is no longer supported by Windows) or a third-party DLL (i.e., one that is not part of the original Windows system DLLs), attackers can give their malicious DLL a hardcoded DLL name and place it under one of the searched directories during the DLL loading process. After the malicious DLL is loaded, it will gain the same permission level as the service that loads it.

During our recent Waterbear investigation, we discovered that the DLL loader loaded two payloads. The payloads performed functionalities we have never seen in other Waterbear campaigns. The first payload injects code into a specific security product to hide the campaign's backdoor. The second payload is a typical Waterbear first-stage backdoor, which we will attempt to dissect first based on a specific case we observed during our analysis.

## Waterbear's first-stage backdoor

We saw a Waterbear loader named **"ociw32.dll"** inside one of the folders in the **%PATH%** environmental variable. This DLL name is hardcoded inside **"mtxoci.dll"** which is loaded by the MSDTC service during boot-up. "**mtxoci.dll"** first tries to query the registry key **"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSDTC\MTxOCI"** to see if the value **"OracleOciLib"** exists. If so, it retrieves the data inside it and loads the corresponding library. If the value doesn't exist, "**mtxoci.dll"** tries to load **"ociw32.dll"** instead. During our investigation, we noticed that the value **"OracleOciLib"** was deleted from the

victim's machine, as shown in Figure 2. Hence, the malicious loader **"ociw32.dll"** was loaded and successfully executed on the host.

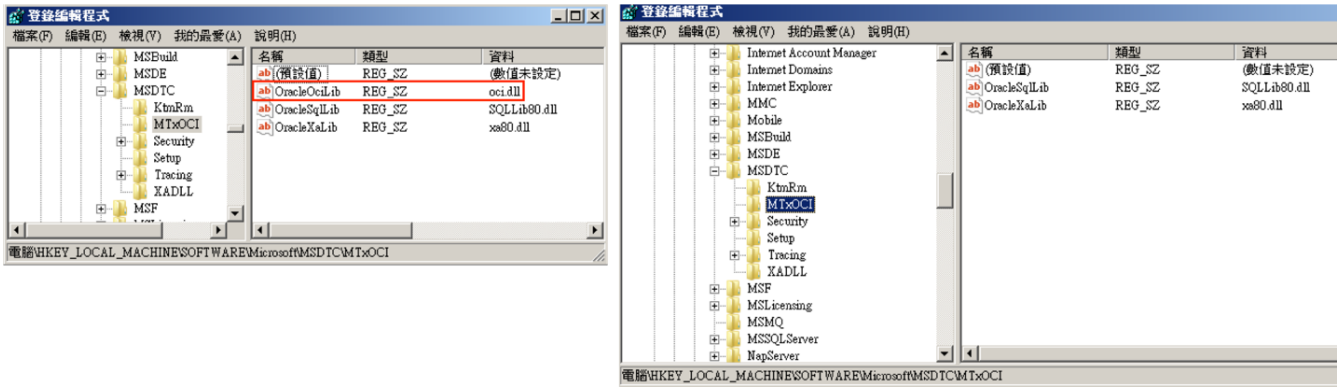



Figure 2. The deleted value "OracleOciLib" on the victim's host

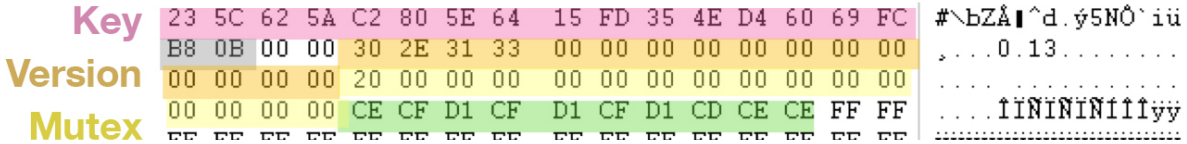
*Note: The image on the left shows how the DLL on a normal machine normally looks. The image on the right showcases how the DLL on a victim's machine appears. Because there is no* **"OracleOciLib"** *value, it loads the hardcoded DLL* **"ociw32.dll"** *instead, which triggers the malicious Waterbear DLL loader.*

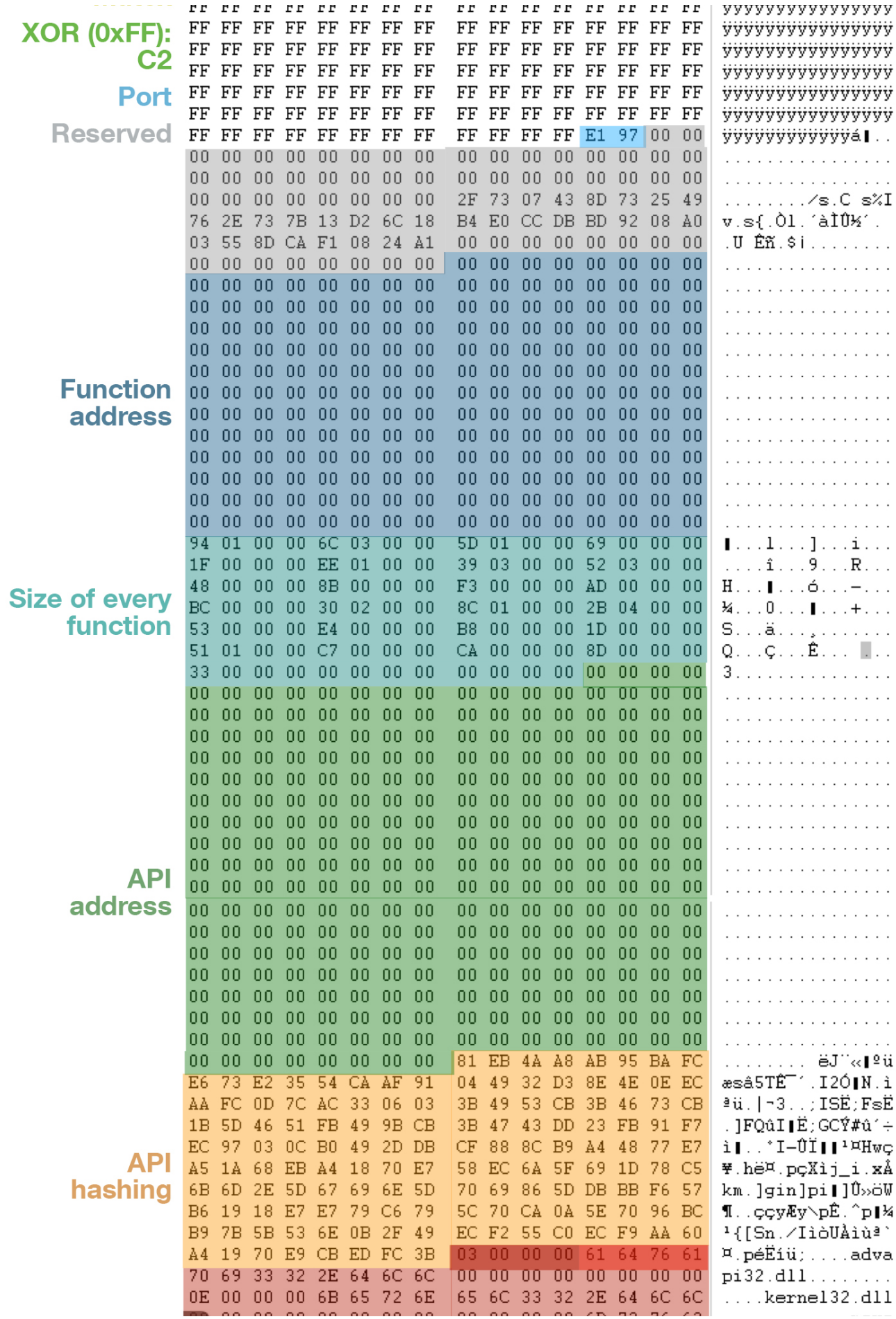
After the Waterbear DLL loader is executed, it searches for a hardcoded path and tries to decrypt the corresponding payload, which is a piece of encrypted shellcode. The decryption algorithm is RC4, which takes the hardcoded path to form the decryption key. If the decrypted payload is valid, it picks a specific existing Windows Service — LanmanServer, which is run by *svchost.exe* — and injects the decrypted shellcode into the legitimate service. In most cases, the payload is a first-stage backdoor, and its main purpose is to retrieve second-stage payloads — either by connecting to a C&C server or opening a port to wait for external connections and load incoming executables.

## Configuration of the first-stage backdoor

Waterbear's first-stage backdoor configuration contains the information required for the proper execution of and communication with external entities.

- Offset 0x00, Size 0x10: Encryption / decryption key for the functions
- Offset 0x10, Size 0x04: 0x0BB8 (reserved)
- Offset 0x14, Size 0x10: Version (e.g., 0.13, 0.14, 0.16, and so on)
- Offset 0x24, Size 0x10: Mutex or reserved bytes
- Offset 0x34, Size 0x78: C&C server address which is XOR-encrypted by key 0xFF. If the backdoor is intended to listen in on a specific port, this section will be filled with 0x00.
- Offset 0xAC, Size 0x02: Port
- Offset 0xAE, Size 0x5A: Reserved bytes
- Table: The function address table of the payload. The block is initially filled with 0x00 and will be propagated during runtime.
- Table: The sizes of functions
- Table: The API address table. The block is initially filled with 0x00 and will be filled with loaded API addresses during runtime.
- Table: The API hashes for dynamic API loading
- A list of DLL names and the number of APIs to be loaded




XOR (0xFF): C2
Port
Reserved
Function address
Size of every function
API address
API hashing

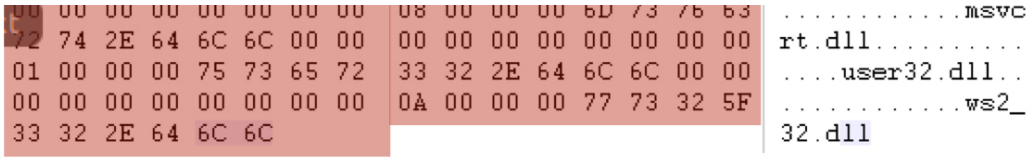



Figure 3. The first-stage backdoor's configuration structure

## Anti-memory scanning of shellcode payload

In order to avoid in-memory scanning during runtime, the payload encrypts all of the function blocks before executing the actual malicious routine. Afterwards, whenever it needs to use a function, it will decrypt the function, execute it, and encrypt the function back again, as can be seen in Figure 4. If a function will not be used on the rest of the execution, it will be scrambled by another mess-up function, as illustrated in Figure 6. The mess-up function muddles up the bytes with random values and makes the input blocks unrecoverable. The purpose of this is to further avoid being detected by a certain cybersecurity solution.

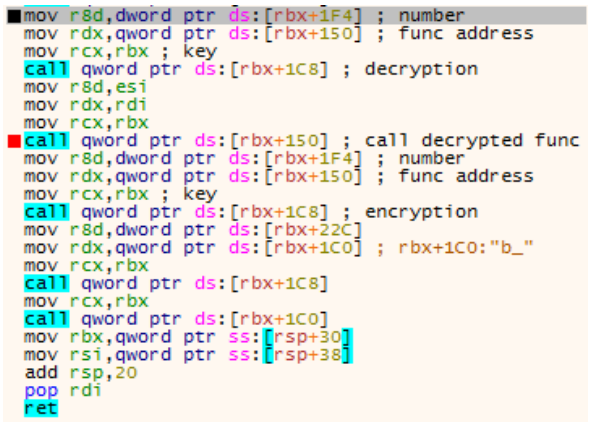



Figure 4. The decryption-execution-encryption flow in the shellcode execution routine

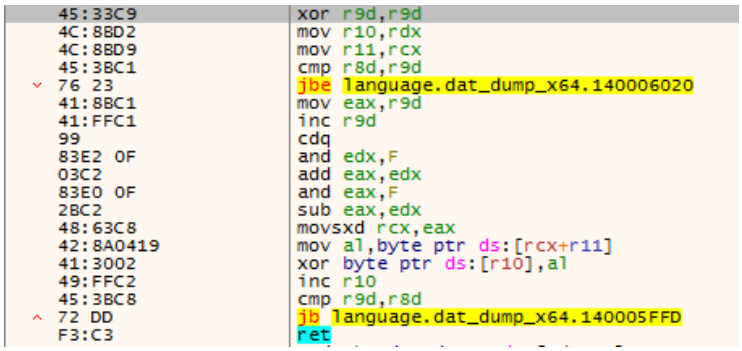



Figure 5. The function for the function block encryption and decryption

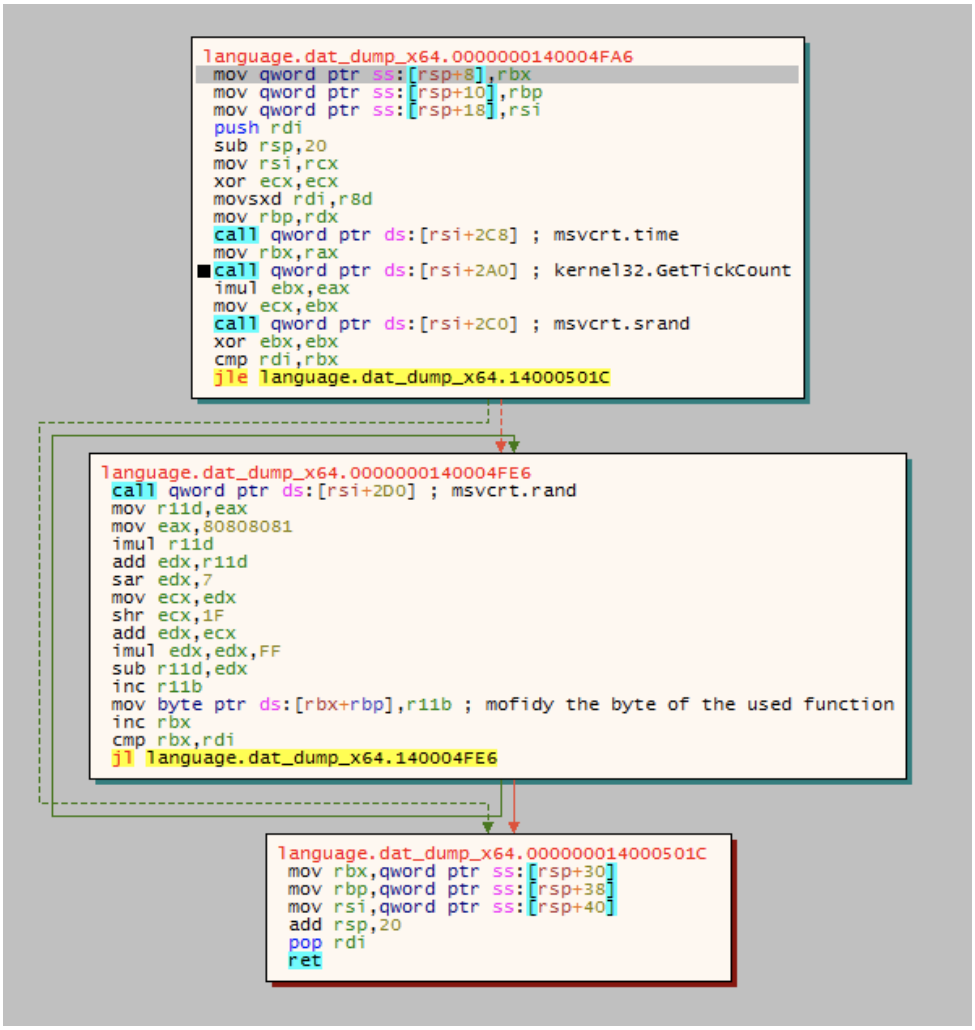



Figure 6. The payload's mess-up function

## Same Waterbear, different story

During our investigation, we found a peculiar incident that stands out from most of the Waterbear infections we've previously seen. This time, the DLL loader loaded two payloads – the first payload performed functionalities we have not seen before: It injected codes into a specific security product to do API hooking in order to hide the backdoor from the product. Meanwhile, the second payload is a typical Waterbear first-stage backdoor.

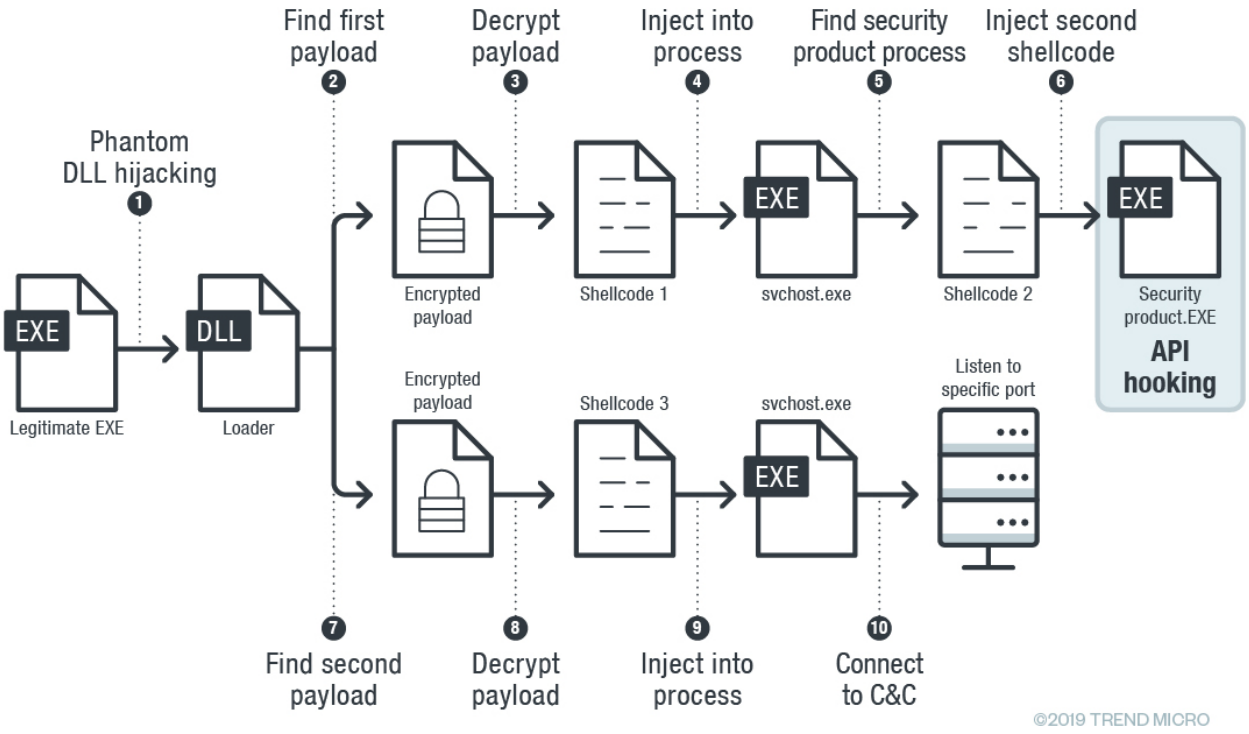



Figure 7. An unusual Waterbear infection chain

Both payloads were encrypted and stored on the victim's disk and were injected into the same service, which was, in this case, LanmanServer. We have observed that the loader tried to read the payloads from the files, decrypted them, and performed thread injections with the following conditions:

1. If the first payload could not be found on the disk, the loader would be terminated without loading the second one.
2. If the first payload was successfully decrypted and injected into the service, the second piece would also be loaded and injected regardless of what happened to the first thread.
3. In the first injected thread, if the necessary executable from the security product was not found, the thread would be terminated without performing other malicious routines. Note that only the thread would be terminated, but the service would still be running.

Regardless if the API hooking was performed or otherwise, the second backdoor would still be executed after having been successfully loaded.

## API hooking to evade detection

In order to hide the behaviors of the first-stage backdoor (which is the second payload), the first payload uses API hooking techniques to avoid being detected by a specific security product and to make an interference in the result of the function execution. It hooks two different APIs, namely **"ZwOpenProcess"** and "**GetExtendedTcpTable"**, to hide its specific processes. The payload only modifies the functions in the memory of the security product process, hence the original system DLL files remain unchanged.

The payload is composed of a two-stage shellcode. The first-stage shellcode finds a specific security product's process with a hardcoded name and injects the second-stage shellcode into that process. The second-stage shellcode then performs API hooking inside the targeted process.

### Hiding process identifiers (PIDs)

The process identifiers or PIDs to be hidden are stored in the shared memory **"Global\<computer_name>."** If the shared memory doesn't exist, it takes the PID embedded by the first-stage shellcode. In this case, the intention of the malicious code is to hide Waterbear's backdoor activities from the security product. Therefore, the first-stage shellcode takes the PID of the Windows Service — which the first-stage shellcode and the succeeding backdoor both inject into — hides the target process, and embeds that PID into the second-stage shellcode.

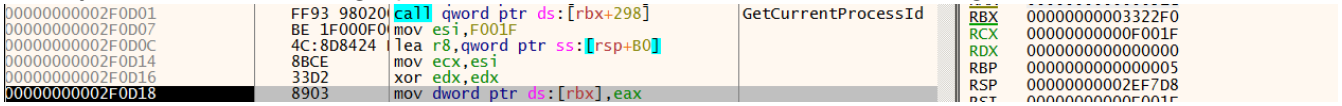



Figure 8. Code that injects current PID into the second-stage shellcode

## Hooking "ZwOpenProcess" in ntdll.dll

The purpose of hooking **"ZwOpenProcess"** is to protect the specific process from being accessed by the security product. Whenever **"ZwOpenProcess"** is called, the injected code will first check if the opened process hits any PIDs in the protected process ID list. If yes, it modifies the process ID, which should open on another Windows Service PID.

First, it builds the hooked function and writes the function at the end of "**ntdll.dll"**. This function includes two parts, as shown in Figure 9:

1. The PID checking procedure. It recursively checks if the PID to be opened by **"ZwOpenProcess"** is in the list of the protected process IDs. If yes, it replaces the PID to be opened with another Windows Service PID that has been written by the Waterbear loader in the beginning.
2. After the PID checking procedure, it executes the original **"ZwOpenProcess"** and returns the result.

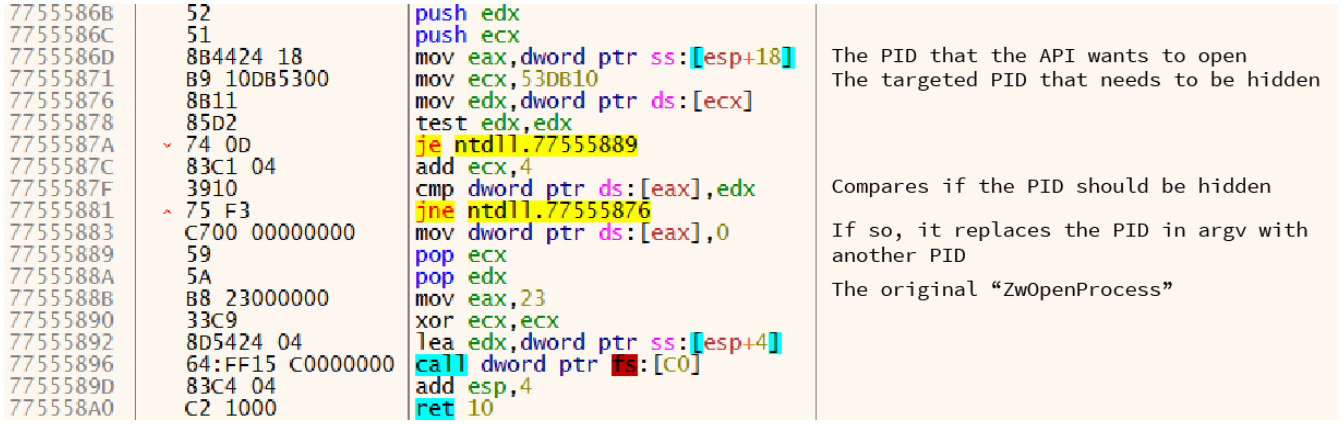



Figure 9. The function hook of **"ZwOpenProcess"** to check and modify the output of the function

Secondly, it writes **"push <ADDRESS> ret"** at the beginning of the original **"ZwOpenProcess**" address. Hence, when **"ZwOpenProcess"** is called, the modified version of **"ZwOpenProcess"** will be executed.

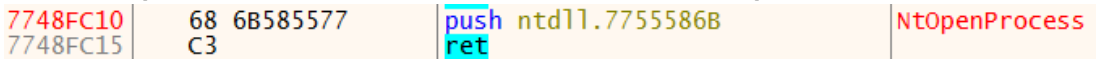



Figure 10. **"ZwOpenProcess"** after modification

The API hooking on **"ZwOpenProcess"** will only be triggered if **"%temp%\KERNELBASE.dll"** exists on the host. It is possible that this check is designed according to the nature of the security product it targets.

## "GetExtendedTcpTable" and "GetRTTAndHopCount" hooks in iphlpapi.dll

The second part of API hooking hooks on **“GetExtendedTcpTable.” “GetExtendedTcpTable”** is used for retrieving a table that contains a list of TCP endpoints available to the application, and it is frequently used in some network-related commands, such as netstat. The purpose of the hook is to remove TCP endpoint records of certain PIDs. In order to achieve that, it modifies two functions: **“GetExtendedTcpTable”** and **“GetRTTAndHopCount.”** The second function, **“GetRTTAndHopCount,”** acts as the place to put the injected hooking code.

**“GetExtendedTcpTable”** only writes a jump to **“GetRTTAndHopCount”** in the beginning of the function. Only the first 5 bytes of the code of the API "GetExtendedTcpTable" are changed, as shown in Figure 11.

```
74B01A8A | ^ E9 D3E8FFFF    | JMP IPHLPAPI.GetRTTAndHopCount
```

Figure 11. Only 5 bytes changed in the **“GetExtendedTcpTable”**

The rest of the routine is all placed in **“GetRTTAndHopCount.”** In the first part of the code, it pushes [**“GetRTTAndHopCount”**+0x3E] as the return address and then executes the first four instructions of the original **“GetExtendedTcpTable”** function (which has already been replaced by the jump instruction in Figure 11). After that, it jumps to **“GetExtendedTcpTable”** to execute the function normally and to catch its return values. The code is shown in Figure 12.

```
74370362    60              pushad                                  “GetRTTAndHopCount”
74370363    8B4424 24       mov eax,dword ptr ss:[esp+24]
74370367    8B4C24 30       mov ecx,dword ptr ss:[esp+30]
7437036B    50              push eax
7437036C    51              push ecx
7437036D    B9 18000000     mov ecx,18
74370372    29CC            sub esp,ecx
74370374    8D7424 44       lea esi,dword ptr ss:[esp+44]
74370378    89E7            mov edi,esp
7437037A    FC              cld
7437037B    F3:A4           rep movsb
7437037D    E8 00000000     call iphlpapi.74370382                  call $0
74370382    5B              pop ebx
74370383    83EB 20         sub ebx,20
74370386    8D43 3E         lea eax,dword ptr ds:[ebx+3E]           Push the following function hook’s addr as return addr
74370389    50              push eax
7437038A    8BFF            mov edi,edi                             The original beginning of “GetExtendedTcpTable”
7437038C    55              push ebp
7437038D    8BEC            mov ebp,esp
7437038F    6A 02           push 2
74370391    90              nop
74370392    90              nop
74370393    90              nop
74370394    90              nop
74370395    90              nop
74370396    90              nop
74370397    90              nop
74370398    90              nop
74370399    90              nop
7437039A  ˇ FFA3 B2000000   jmp dword ptr ds:[ebx+B2]              jmp to “GetExtendedTcpTable” + 7
```

Figure 12. The first part of injected code in **“GetRTTAndHopCount,”** which executes **“GetExtendedTcpTable”** and returns back to the next instruction

After **“GetExtendedTcpTable”** is executed and the process returns back to the second part of the code, it iteratively checks every record in the returned Tcp table. If any record contains the PID Waterbear wants to hide, it will remove the corresponding record, modify the record number inside the table, and continue to check the succeeding records.

In the end, it returns the modified table.

```
743703A0    833C24 02       cmp dword ptr ss:[esp],2
743703A4  ~ 75 60           jne iphlpapi.74370406
743703A6    85C0            test eax,eax
743703A8  ~ 75 5C           jne iphlpapi.74370406
743703AA    50              push eax
743703AB    8B6C24 08       mov ebp,dword ptr ss:[esp+8]
743703AF    8B4D 00         mov ecx,dword ptr ss:[ebp]
743703B2    8B5424 08       mov edx,dword ptr ss:[esp+8]
743703B6    83C2 04         add edx,4
743703B9    31F6            xor esi,esi
743703BB    8B83 B6000000   mov eax,dword ptr ds:[ebx+B6]
743703C1    8B00            mov eax,dword ptr ds:[eax]
743703C3    85C0            test eax,eax                    Loops through the results to
743703C5  ~ 74 39           je iphlpapi.74370400            check every PID
743703C7    8B7A 14         mov edi,dword ptr ds:[edx+14]
743703CA    39C7            cmp edi,eax
743703CC  ~ 74 11           je iphlpapi.743703DF            If the PID matches, it jumps to
743703CE    83FE 0A         cmp esi,A                       the removal part
743703D1  ~ 7D 2D           jge iphlpapi.74370400
743703D3    46              inc esi
743703D4    8B83 B6000000   mov eax,dword ptr ds:[ebx+B6]
743703DA    8B04B0          mov eax,dword ptr ds:[eax+esi*4]
743703DD  ^ EB E4           jmp iphlpapi.743703C3
743703DF    51              push ecx                        Removes the hit record
743703E0    49              dec ecx
743703E1    89D7            mov edi,edx
743703E3    89D6            mov esi,edx
743703E5    83C6 18         add esi,18
743703E8    6BC9 18         imul ecx,ecx,18
743703EB    FC              cld
743703EC    F3:A4           rep movsb
743703EE    59              pop ecx
743703EF    51              push ecx
743703F0    31C0            xor eax,eax
743703F2    B9 18000000     mov ecx,18
743703F7    F3:AA           rep stosb
743703F9    59              pop ecx
743703FA    83EA 18         sub edx,18
743703FD    FF4D 00         dec dword ptr ss:[ebp]          Reduces the record number in
74370400    83C2 18         add edx,18                      the returned structure
74370403    E2 B4           loop iphlpapi.743703B9
74370405    58              pop eax
74370406    894424 2C       mov dword ptr ss:[esp+2C],eax   Returns the modified table
7437040A    5A              pop edx
7437040B    5A              pop edx
7437040C    61              popad
7437040D    8B4424 04       mov eax,dword ptr ss:[esp+4]
74370411    C2 1800         ret 18
```

Fig. 13. The injected code's second part in **"GetRTTAndHopCount"** that checks and removes returned records of certain PIDs

Rather than directly disabling these two functions, this method of using API hooking makes noticing malicious behaviors more difficult, especially since both functions still work and return results normally. Although in this case, the affected process is specified in the first-stage shellcode, the API hooking logic is quite generic that the same piece of shellcode can be used to hook other vendors' products.

## Conclusion

This is the first time we've seen Waterbear attempting to hide its backdoor activities. By the hardcoded product name, we infer that the attackers are knowledgeable of the victims' environment and which security product(s) they use. The attackers might also be familiar with how security products gather information on their clients' endpoints and networks, so that they know which APIs to hook. Since the API hooking shellcode adopts a generic approach, the similar code snippet might be used to target other products in the future and make the activities of Waterbear harder to detect.

| Tactic | Technique | ID | Description |
|---|---|---|---|
| Execution | Execution through Module Load | T1129 | Dynamically loads the DLLs through the shellcode |
| Execution through API | T1106 | Dynamically loads the APIs through the shellcode | |
| Persistence | Hooking | T1179 | Hooks security product's commonly used APIs |
| Privilege Escalation | Process Injection | T1055 | Injects the decrypts payload into *svchost.exe* process |
| Hooking | T1179 | Hooks security products' commonly used APIs | |
| Defense Evasion | Binary Padding | T1009 | Adds junk data to evade anti-virus scan |
| Disabling Security Tools | T1089 | Targets a specific security product's process for injection purposes | |
| Deobfuscate/Decode Files or Information | T1140 | Uses TROJ_WATERBEAR to decrypt encrypted payload | |
| Execution Guardrails | T1480 | Targets specific software in the victim's environment | |
| DLL Side-Loading | T1073 | Uses modified legitimate DLL to load the malicious DLL | |
| Process Injection | T1055 | Injects the decrypted payload into svchost.exe process | |
| Exfiltration | Exfiltration Over Command and Control Channel | T1041 | Possibly sends collected data to attackers via C&C channel |

## Indicators of Compromise (IoCs)

| SHA256 | Detection Name |
|---|---|
| 649675baef92381ffcdfa42e8959015e83c1ab1c7bbfd64635ce5f6f65efd651 | BKDR_WATERBEAR.ZTGF |
| 3909e837f3a96736947e387a84bb57e57974db9b77fb1d8fa5d808a89f9a401b | TROJ_WATERBEAR.ZTGD |
| fcfdd079b5861c0192e559c80e8f393b16ba419186066a21aab0294327ea9e58 | TROJ_WATERBEAR.ZTGJ |
| 3f26a971e393d7f6ce7bf4416abdbfa1def843a0cf74d8b7bb841ca90f5c9ed9 | TROJ_WATERBEAR.ZTGH |
| abb91dfd95d11a232375d6b5cdf94b0f7afb9683fb7af3e50bcecdb2bd6cb035 | TROJ_WATERBEAR.ZTGH |
| bda6812c3bbba3c885584d234be353b0a2d1b1cbd29161deab0ef8814ac1e8e1 | TROJ_WATERBEAR.ZTGI |
| 53402b662679f0bfd08de3abb064930af40ff6c9ec95469ce8489f65796e36c3 | TROJ_WATERBEAR.ZTGH |
| f9f6bc637f59ef843bc939cb6be5000da5b9277b972904bf84586ea0a17a6000 | TROJ_WATERBEAR.ZTGI |
| 3442c076c8824d5da065616063a6520ee1d9385d327779b5465292ac978dec26 | BKDR_WATERBEAR.ZTGD |

| | |
|---|---|
| 7858171120792e5c98cfa75ccde7cba49e62a2aeb32ed62322aae0a80a50f1ea | TROJ64_WATERBEAR.ZTGI |
| acb2abc7fb44c2fdea0b65706d1e8b4c0bfb20e4bd4dcee5b95b346a60c6bd31 | BKDR_WATERBEARENC.ZTGF |
| b9f3a3b9452a396c3ba0ce4a644dd2b7f494905e820e7b1c6dca2fdcce069361 | BKDR64_WATERBEAR.ZTGD |
| 7c0d2782a33debb65b488893705e71a001ea06c4eb4fe88571639ed71ac85cdd | BKDR_WATERBEARENC.ZTGH |
| c7c7b2270767aaa2d66018894a7425ba6192730b4fe2130d290cd46af5cc0b7b | BKDR_WATERBEARENC.ZTGI |
| 7532fe7a16ba1db4d5e8d47de04b292d94882920cb672e89a48d07e77ddd0138 | BKDR_WATERBEARENC.ZTGI |
| dea5c564c9d961ccf2ed535139fbfca4f1727373504f2972ac92acfaf21da831 | BKDR_WATERBEARENC.ZTGI |
| 05d0ab2fbeb7e0ba7547afb013d307d32588704daac9c12002a690e5c1cde3a4 | BKDR64_WATERBEARENC.ZTGJ |
| 39668008deb49a9b9a033fd01e0ea7c5243ad958afd82f79c1665fb73c7cfadf | BKDR_WATERBEARENC.ZTGD |