OracleOciLib retrieves the data inside it and loads the corresponding library. If the value doesn't exist, service during boot-up. We saw a Waterbear loader named Waterbear's first-stage backdoor specific case we observed during our analysis. The 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.

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 is a typical Waterbear first-stage backdoor, which we will attempt to dissect first based on a specific case we observed during our analysis.

The 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.

A Waterbear 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.
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 0x18, Size 0x10: Mutex or reserved bytes
- Offset 0x38, 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 0x54, Size 0x02: Port
- Offset 0x56, Size 0x04: 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 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
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.
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.

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 (PDs)

The process identifiers or PDs to be hidden are stored in the shared memory “Global\computer_name.” If the shared memory doesn’t exist, it takes the PD 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 PD of the Windows Service — which the first-stage shellcode and the succeeding backdoor both inject into — hides the target process, and embeds that PD 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.

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.

The API hooking of “ZwOpenProcess” will only be triggered if “…\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.

The rest of the routine is all placed in “GetRTTAndHopCount.” In the first part of the code, it pushes “[GetRTTAndHopCount]+0xC3E” 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.

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.
Figure 13. The first part of injected code in "GetRTTAndHopCount," which executes "GetExtendedTcpTable" and returns back to the next instruction

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.

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Technique</th>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>Execution through Module Load</td>
<td>T1129</td>
<td>Dynamically loads the DLLs through the shellcode</td>
</tr>
<tr>
<td>Execution through API</td>
<td>T1106</td>
<td></td>
<td>Dynamically loads the APIs through the shellcode</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hooking</td>
<td>T1179</td>
<td>Hooks security product's commonly used APIs</td>
</tr>
<tr>
<td>Privilege Escalation</td>
<td>Process Injection</td>
<td>T1055</td>
<td>Injects the decrypts payload into svchost.exe process</td>
</tr>
<tr>
<td>Hooking</td>
<td>T1179</td>
<td></td>
<td>Hooks security products' commonly used APIs</td>
</tr>
<tr>
<td>Defense Evasion</td>
<td>Binary Padding</td>
<td>T1009</td>
<td>Adds junk data to evade anti-virus scan</td>
</tr>
<tr>
<td>Disabling Security Tools</td>
<td>T1120</td>
<td></td>
<td>Targets a specific security product’s process for injection purposes</td>
</tr>
<tr>
<td>Deobfuscate/Decode Files or Information</td>
<td>T1140</td>
<td></td>
<td>Uses TROJ_WATERBEAR to decrypt encrypted payload</td>
</tr>
<tr>
<td>Execution Guards</td>
<td>T1140</td>
<td></td>
<td>Targets specific software in the victim’s environment</td>
</tr>
<tr>
<td>DLL Side-Loading</td>
<td>T1013</td>
<td></td>
<td>Uses modified legitimate DLL to load the malicious DLL</td>
</tr>
<tr>
<td>Process Injection</td>
<td>T1055</td>
<td></td>
<td>Injects the decrypted payload into svchost.exe process</td>
</tr>
<tr>
<td>Exfiltration</td>
<td>Exfiltration Over Command and Control Channel</td>
<td>T1041</td>
<td>Possibly sends collected data to attackers via C&amp;C channel</td>
</tr>
</tbody>
</table>

SHA256                Detection Name
6496758a91f2b31f1fca42e2859015e831ca1b7b7f6e435e5f66e5e9d851   BDR_WATERBEAR,ZTGF
3909e837f3a6e36974e787a84b5657e57974dbc7777bd58f85d8da8999a401b   TROJ_WATERBEAR,ZTGD
fcd6879a5b6c1a92e5599028f389b16a4194186e66a2a1a9e294327eab5e8     TROJ_WATERBEAR,ZTGF
3268b73e3f3976c7f4d4664db1a1d9f83a30c74d1d7bb841a9f5c5f6d9     TROJ_WATERBEAR,ZTGF
ab81b03d4511a323275e5f94d7a01b96b97a53b0c0cc63b2b6c335        TROJ_WATERBEAR,ZTGF
bda68123bba3b85558a2234e8353b0a2b11b1cd92e1a5e9be28884ac1e8e1   TROJ_WATERBEAR,ZTGI