Omni walkthrough

Index

List of pictures

Disclaimer

I do this box to learn things and challenge myself. I'm not a kind of penetration tester guru who always knows where to look for the right answer. Use it as a guide or support. Remember that it is always better to try it by yourself. All data and information provided on my walkthrough are for informational and educational purpose only. The tutorial and demo provided here is only for those who're willing and curious to know and learn about Ethical Hacking, Security and Penetration Testing.

Reconnaissance

The results of an initial nMap scan are the following:

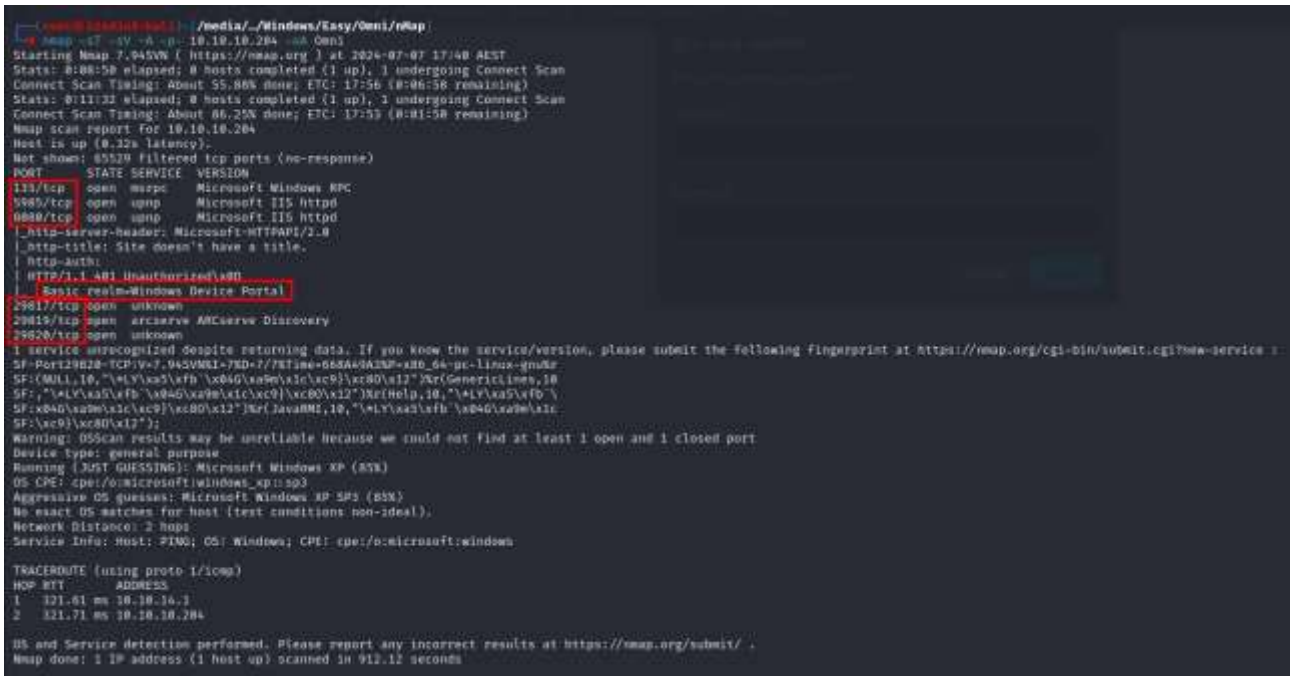

Figure 1 - nMap scan result

Open ports are 135, 5985, 8080, 29817, 29819,29820. So, this box has msrpc service enabled, two web application running on port number 5985 and 8080 and **ARCserve** service enabled on port number 29817, 29819,29820. Also, nMap recognized Windows XP as operative system.

Initial foothold

The first thing I did was try to open the two web application on port 5985 and 8080. The one running on port 5985 provide a 404 page, the one running on port 8080 provide an authentication request. nMap provide me an important information about the authentication of web application running on port 8080. In fact, as we can see in the previous image, nMap identified **Windows Device Portal** as **Basic real**. A real identify an authentication context. Looking for more details about this real in the Internet, I found out the Windows Device Portal real is used to identify HoloLens devices (https://github.com/MicrosoftDocs/mixed-reality/blob/docs/mixed-reality-docs/mr-dev-docs/develop/advanced-concepts/using-the-windows-device-portal.md). This means that nMap has not correctly identified the operative system because I found out that the target is a IoT device. Also, I found a very interesting exploit named **SirepRAT**.

User flag

The next step is obtaining a shell. After I studied the SirepRAT documentation, I learned how to run arbitrary command via the SirepRAT exploit. So, I downloaded netcat for Windows and uploaded it on the target in a temporary folded I created. I accomplished these tasks running the following commands:

Figure 2 - Temporary folder creation

Figure 3 - netcat uploaded on the target

At this point, I can open a listener on my Kali machine and run a shell running the following command:

Figure 4 - Exploit command

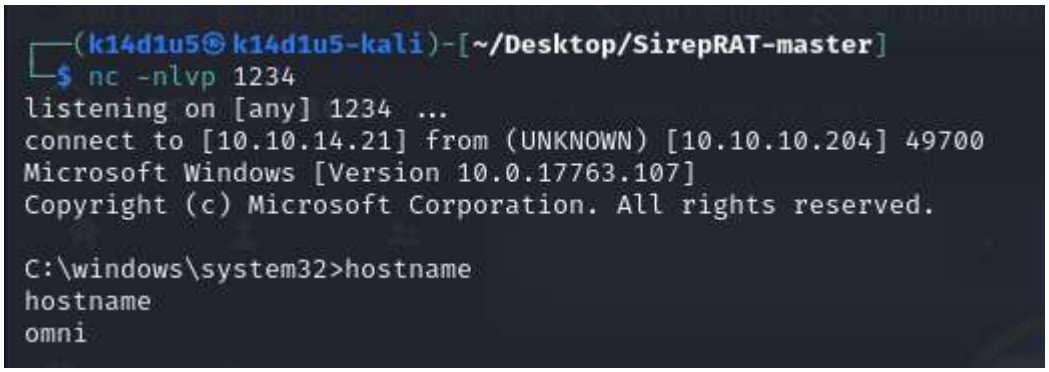

Figure 5 - Shell obtained

Now I need to find flags and information. However, analyzing the system I didn't find so much interesting information. After a lot of time, I remembered that Windows operative system can store credential in some registries or SAM system. So, I tried to exploit these registries, in particular **sam** and **system** one. I was able to extract these two registries running the following commands:

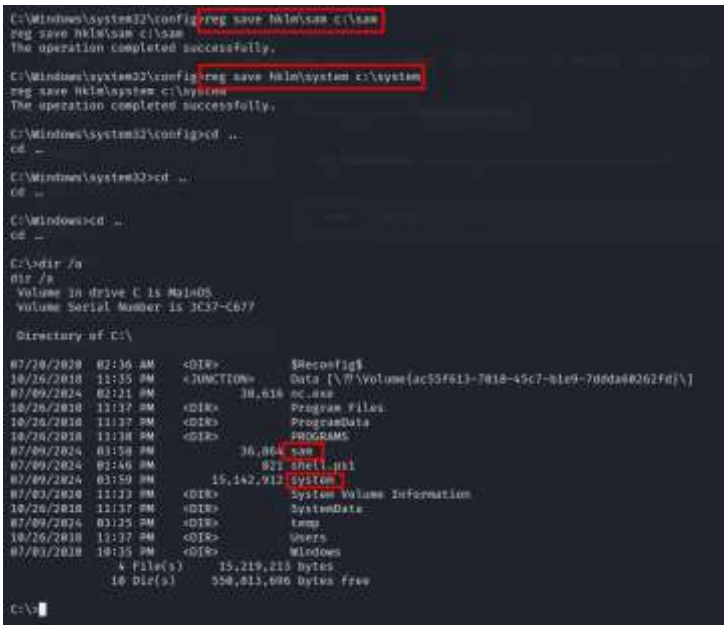

Figure 6 - Extract registies

I downloaded these two files on my Kali local machine as shown in the following pictures:

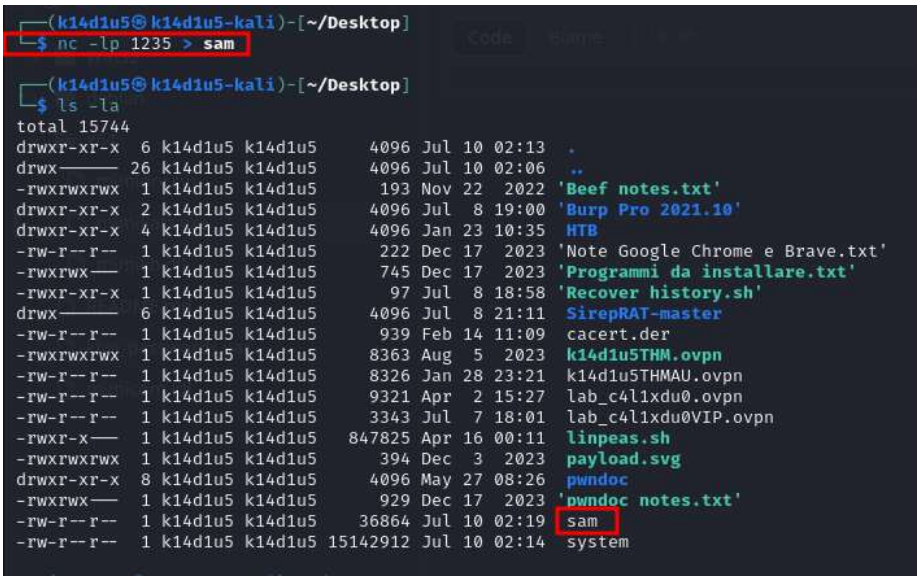

Figure 7 - Download file SAM

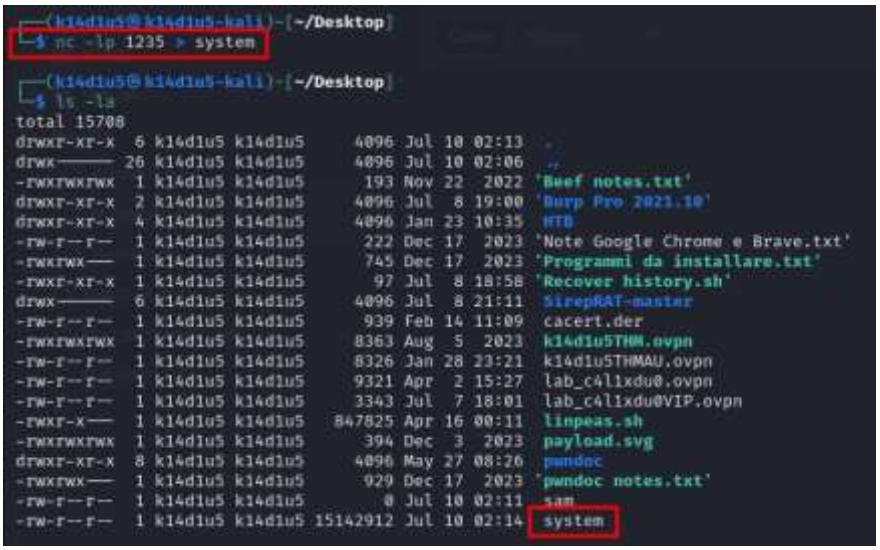

Figure 8 - Download file SYSTEM

These two files can be used to crack password contained in themselves. To accomplish this goal, the first step is running the **secretsdump.py** script. In this way I was able to obtain a list of password hash regarding the target system:

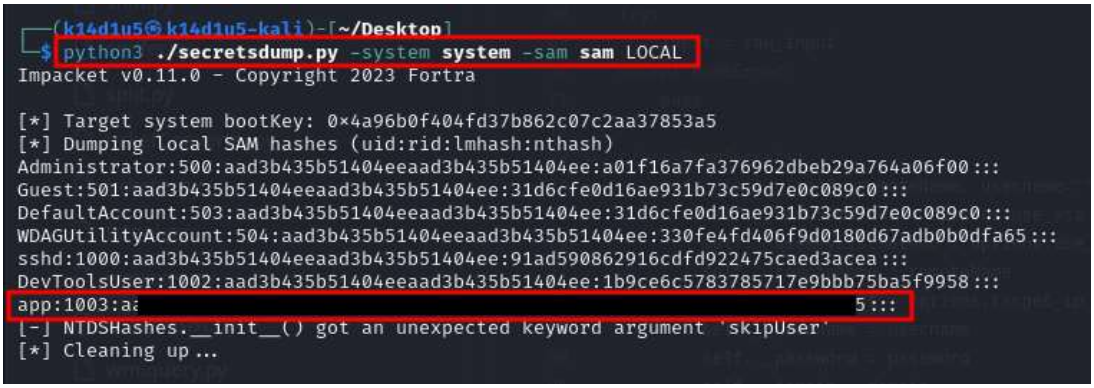

Figure 9 - SAM file dump

Next step is to copy the row regarding **app** user in a file and crack it running John The Ripper:

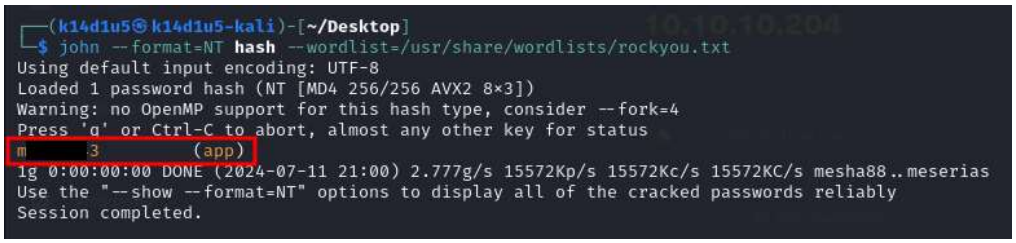

Figure 10 - Hash cracked

I tried these credentials to log in the web application running on port 8080 and them worked! Fro the web application I can run commands. So, I can open a new shell from here opening a new listener and running the following command from the web application itself:

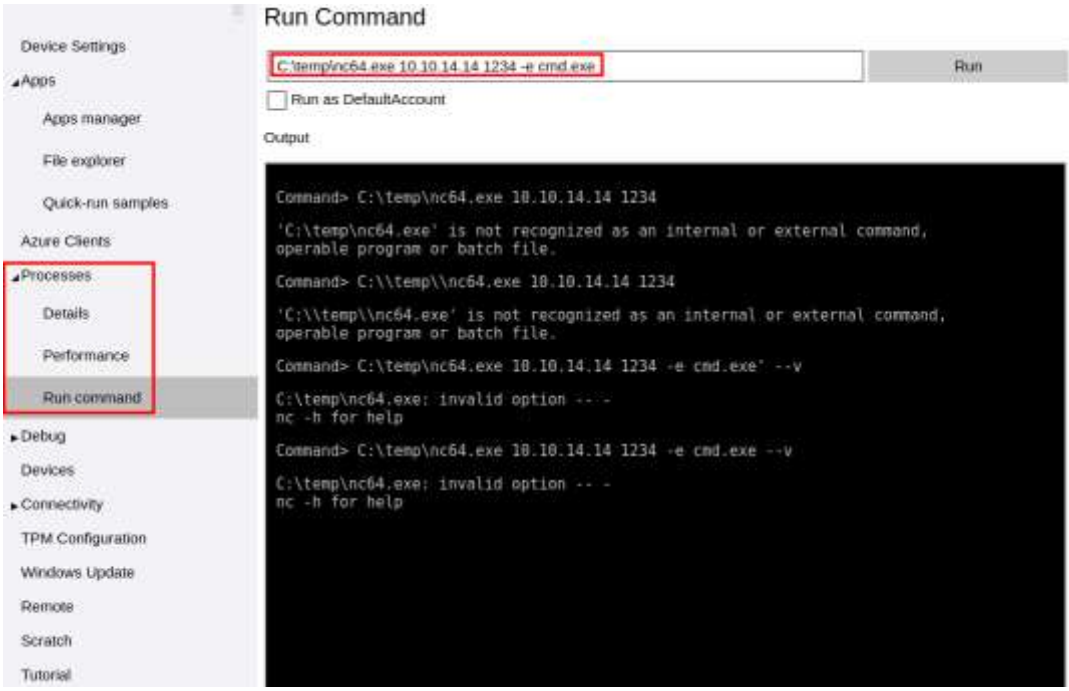

Figure 11 - Command to open a shell from web application

I can convert this new shell in a PowerShell as shown in the following picture:

Figure 12 - PowerShell on the target

Analyzing again the file system, I found two files that contain the user flag and the root flag. The first one is in $C:\Data\Users\app$ path, the second one is in $C:\Data\Users\administrator$. I just need to understand how to decrypt them. After a lot of research in the Internet, I found out I need the following kind of information to decrypt each of them:

Figure 13 - Data to decrypt user flag

All these information are contained in the respective file regarding user flag, root flag or, as we will see later, new credentials. After I set all the needed information, I can decrypt the user flag running the following command:

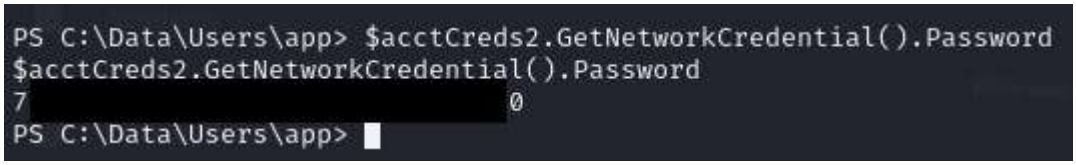

Figure 14 - User flag

Privilege escalation

At this point I need to escalate my privileges to be able to decrypt the root flag too. I still looked for some useful information and I luckily found the following interesting file:

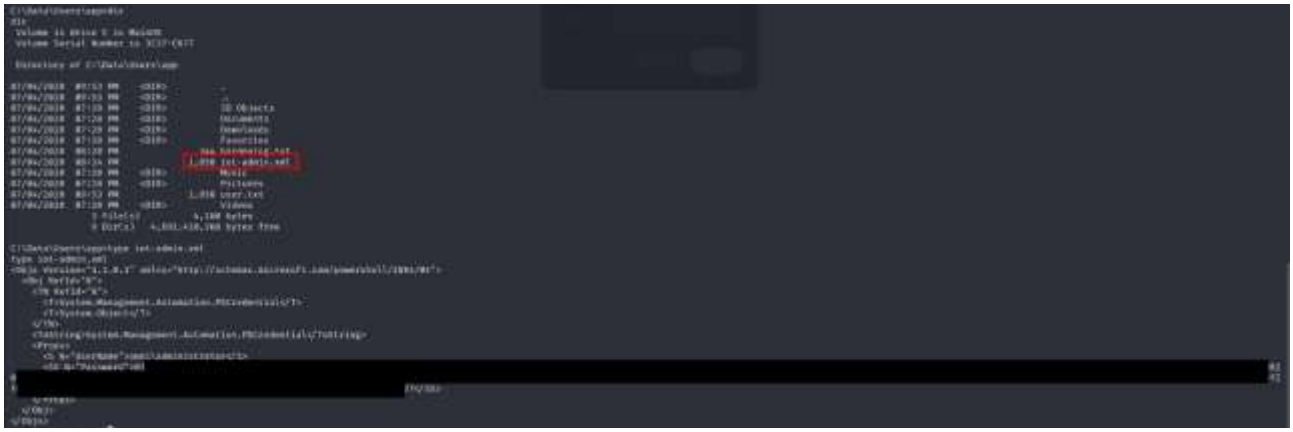

Figure 15 - Data to escalate privileges

So, I applied the same procedure I used before to decrypt the user flag and I obtained new credentials regarding the **Administrator** user (I forgot the screen, I am sorry). Again, these credentials worked to log in the web application and I opened a new shell (running the same command as before from the web application), this time as **Administrator**:

```
┌──(k14d1u5㉿k14d1u5-kali)-[~/Desktop]
└─$ nc -nlvp 1235
listening on [any] 1235 ...
connect to [10.10.14.15] from (UNKNOWN) [10.10.10.204] 49674
Microsoft Windows [Version 10.0.17763.107]
Copyright (c) Microsoft Corporation. All rights reserved.

C:\windows\system32>powershell
powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\windows\system32> [System.Security.Principal.WindowsIdentity]::GetCurrent().Name
[System.Security.Principal.WindowsIdentity]::GetCurrent().Name
OMNI\Administrator
PS C:\windows\system32>
```

Figure 16 - Administrator shell

Since I was logged as **Administrator**, I was able to decrypt the root flag too. Again, I used the same procedure I used to decrypt the user flag. So, the root flag is:

Figure 17 - Root flag

Appendix A – Secure String

SecureString is a string type that provides a measure of security. It tries to avoid storing potentially sensitive strings in process memory as plain text. The value of an instance of SecureString is automatically protected using a mechanism supported by the underlying platform when the instance is initialized or when the value is modified. Your application can render the instance immutable and prevent further modification by invoking the MakeReadOnly method. The maximum length of a SecureString instance is 65,536 characters.

Because the operating system does not directly support SecureString, you must convert the value of the SecureString object to the required string type before passing the string to a native method. The Marshal class has five methods that do this:

- Marshal.SecureStringToBSTR, which converts the SecureString string value to a binary string (BSTR) recognized by COM.
- Marshal.SecureStringToCoTaskMemAnsi and Marshal.SecureStringToGlobalAllocAnsi, which copy the SecureString string value to an ANSI string in unmanaged memory.
- Marshal.SecureStringToCoTaskMemUnicode and Marshal.SecureStringToGlobalAllocUnicode, which copy the SecureString string value to a Unicode string in unmanaged memory.

Each of these methods creates a clear-text string in unmanaged memory. It is the responsibility of the developer to zero out and free that memory as soon as it is no longer needed.

There are significant limitations on how secure a SecureString instance is. These include:

- Platform: on the Windows operating system, the contents of a SecureString instance's internal character array are encrypted. However, whether because of missing APIs or key management issues, encryption is not available on all platforms. Because of this platform dependency, SecureString does not encrypt the internal storage on non-Windows platform. Other techniques are used on those platforms to provide additional protection;

- Duration: even if the SecureString implementation is able to take advantage of encryption, the plain text assigned to the SecureString instance may be exposed at various times:
    - Because Windows doesn't offer a secure string implementation at the operating system level, .NET still has to convert the secure string value to its plain text representation in order to use it;
    - Whenever the value of the secure string is modified by methods such as AppendChar or RemoveAt, it must be decrypted (that is, converted back to plain text), modified, and then encrypted again;
    - If the secure string is used in an interop call, it must be converted to an ANSI string, a Unicode string, or a binary string (BSTR). For more information, see the SecureString and interop section.
- Storage versus usage: more generally, the SecureString class defines a storage mechanism for string values that should be protected or kept confidential. However, outside of .NET itself, no usage mechanism supports SecureString. This means that the secure string must be converted to a usable form (typically a clear text form) that can be recognized by its target, and that decryption and conversion must occur in user space.

Appendix B – SirepRAT

Windows IoT Core, in its default configuration, allows several incoming connections through its firewall. Three of them are the ones used by this service. The service continuously listens on these ports, each for a different purpose.

The following list provides both versions of names are given regarding the ports used: first name is the IoT oriented name, the second one is the Windows Phone oriented name:

1. 29820: **Sirep-Server-Protocol2/WPConProtocol2.** Used for the command communication;
2. 29819: **Sirep-Server-Ping/WPConTCPPing/WPPingSirep**. Used for the simple echo service;
3. 29817: **Sirep-Server-Service/WPCon**. Its unique purpose was not investigated and is left for future research (after objectives were achieved using services on port **WPConProtocol2**).

The service listens on the **Sirep-Server-Protocol2** port and accepts commands sent to it in a unique binary structure. Results are sent back to the command initiator in a simple binary structure. Surprisingly, the filtering is not based on any form of authentication or even identification. Basically, any remote client can send commands to the device, with the only requirement being that the device's relevant network interface is connected with an Ethernet cable (not wirelessly). The check is based solely on the details of the local socket that received the new TCP connection. This authorization form is very permissive.

A service routine accepts command packets in a binary form. This is the gate that routes the packet buffers to the right path in code, in a switch manner. The routing is done based on the first integer of the received packet, that represents the command code. Each command code is mapped to its handling function.

The general top-level structure of the packet is the common Type-Length-Value format:

1. The 1st integer represents the command type to perform;
2. The 2nd is the overall payload length, starting from the 3rd Integer;
3. The rest of the payload forms the command data, and is command-type-specific.

Some of the inner content structures are described below in detail. The simplest command supported is the **GetSystemInformationFromDevice** command. It requires no special arguments, and actually no data at all. The packet is nothing more than 2 integers: the first being the command type ($0x32$) and the second is the payload length ($0x0$):

00	01	02	03	04	05	06	07
Command Type				Payload Length			
32	00	00	00	00	00	00	00

Figure 18 - GetSystemInformationFromDevice packet

Sending this through a TCP connection to port 29820 of the Windows IoT device returns a binary block that represents different properties of the target system.

The launch command is probably the strongest of all the RAT-like abilities that the Sirep service exposes. It gets a program path, command line parameters and other arguments that correspond to some of the parameters needed for the API call. Since the service is the one spawning the process, the created process is given the LocalSystem user context which means it runs with SYSTEM privileges. Alternatively, it supports running it as the currently logged on user. The packer has the following structure:

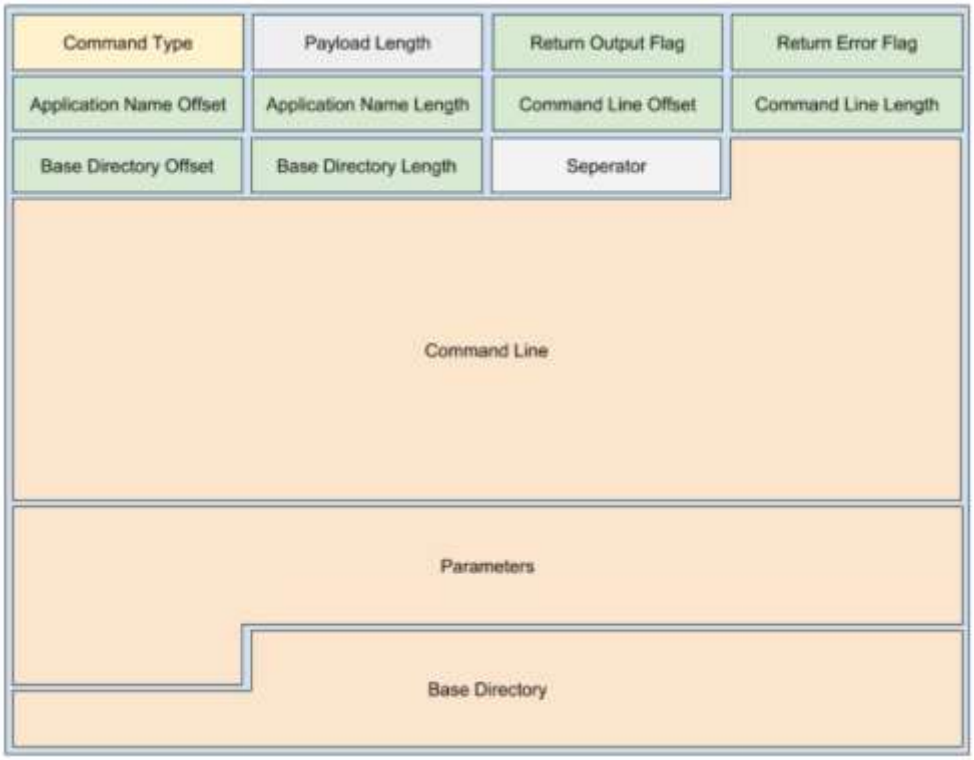

Figure 19 - LaunchCommandWithOutput

In general, string arguments are specified using a pair of integers representing <offset, length>. The offset is calculated starting from offset 0x9 of the whole packet (the green boxes above):

Name	Value	Start	Size
CommandType	LaunchCommand (Ah)	0h	4h
PayloadLength	AEh	4h	4h
ReturnOutputFlag	1h	8h	4h
ReturnErrorFlag	1h	Ch	4h
ApplicationNameOffset	24h	10h	4h
ApplicationNameLength	66h	14h	4h
CommandLineOffset	8Ah	18h	4h
CommandLineLength	6h	1Ch	4h
BaseDirectoryOffset	90h	20h	4h
BaseDirectoryLength	1Eh	24h	4h
Separator	0h	28h	4h
ApplicationName[51]	<AS_LOGGED_ON_USER>C:\Windows\System32\hostname.exe	2Ch	66h
CommandLine[3]	/?	92h	6h
BaseDirectory[15]	C:\Users\Public	98h	1Eh

Figure 20 - LaunchCommandWithOutput offset table

The **GetFileFromDevice** command is very simple. It require only a remote path of a file to download:

00	01	02	03	04	05	06	07	08	...	47
Command Type				Payload Length				FilePath		
1E	00	00	00	40	00	00	00	C:\Windows\System32\hostname.exe		

Figure 21 - GetFileFromDevice packet

The **GetFileInformationFromDevice** command is almost identical to the former GetFileFromDevice, but has a different command code. And of course, it returns information about the specified remote file, and not the file data:

00	01	02	03	04	05	06	07	08		47
Command Type				Payload Length				FilePath		
3C	00	00	00	40	00	00	00	C:\Windows\System32\hostname.exe		

Figure 22 - GetFileInformationFromDevice packet

The **PutFileOnDevice** command lets the client specify a remote path along with data to write to that path. The path is a regular Sirep packed string, and the data is represented with WriteRecords structure that are described below:

```
enum WRITE_RECORD_TYPE {
    RegularChunk = 0x15,
    LastChunk = 0x16,
} WriteRecordType <bgcolor=cAqua>;
```

Figure 23 - WriteRecord structure

Its packet structure is the following:

00	01	02	03	04	05	06	07	08	...	47
Command Type				Payload Length				FilePath		
14	00	00	00	40	00	00	00	C:\Windows\System32\hostname.exe		

48	49	4A	4B	4C	4D	4E	4F	50	...	67
WriteRecord Type				Data Length				Data		
15	00	00	00	18	00	00	00	HELLO WORLD!		

Figure 24 - PutFileOnDevice packet

The result is returned as multiple records of different types. Each record is built in the TLV format. There are several record types, the main ones are listed in the following table:

Command	Record Type	Code	Notes
GetSystemInformation	SystemInformation	0x33	
Launch command	HResult	0x01	Mandatory, represents the `HRESULT`
	OutputStream	0x0B	Optional, can't be set if error stream is not set
	ErrorStream	0x0C	Optional
GetFileFromDevice	File	0x1F	
PutFileOnDevice	HResult	0x01	represents the `HRESULT`
GetFileInformationFromDevice	FileInformation	0x3D	

Figure 25 - Result packet structure