SafeBreach

Whitepaper

Windows IoT Core: RCE as System

A SafeBreach Labs research by
Dor Azouri, Security Researcher, SafeBreach
March 2019

Contents

Intro

In this paper, we will present a **new** exploit that provides you with remote command execution (RCE) as the SYSTEM user. This exploit works on cable-connected Windows IoT Core devices, running Microsoft's official stock image. The achieved execution can be performed as either the SYSTEM user or the currently logged on user (usually "DefaultAccount"), without any needed authentication.

Moreover, we will break down the Sirep/WPCon protocol that is abused for this purpose in granular detail. We will show how this protocol exposes a remote command interface for attackers, including RAT capabilities such as get/put arbitrary files in arbitrary locations, and obtaining system information.

In addition, we will provide an easy-to-use tool, SirepRAT, which implements the techniques laid out in this paper.

The method described in this paper exploits the Sirep Test Service that's built-in and running on the official images offered at Microsoft's site. This service is the client part of the HLK setup one may build in order to perform driver/hardware tests on IoT devices. It serves the Sirep/WPCon protocol[1].

Although Microsoft officially intends for this OS edition to be used by hobbyists and developers, and suggests building a custom image for increased security in commercial products, all other interfaces are password protected (SSH, PowerShell, Web Device Portal...). In contrast, this undocumented method shows a new way to control the device with no authentication required, and provides the simplest known way to run programs as SYSTEM on Windows IoT Core devices.

The research was performed on a Windows IoT Core installed on a Raspberry Pi 3, but is probably not limited to this board as it abuses a Windows service and protocol, which should be platform independent.

[1] The name ambiguity is explained later in this paper

Windows IoT

Windows IoT is Microsoft's operating system for embedded and IoT devices, and already runs in enterprise environments and commercial handheld products, as well as in cool DIY projects.

Windows IoT shares much of the Windows 10 binaries, but it cannot be identical, right? Right! It needs to be efficient resource-wise, ignore irrelevant features, and surely add new IoT-oriented features. Moreover, it is set to be deployed on various boards and sets of hardware, so low-level access for developers is necessary to make it dev-friendly.

It is the first free version of Windows and can be thought of as both the successor of Windows Embedded, and the lightweight version of Windows 10.

Another major novelty Windows IoT features is the support for ARM CPUs, such as the Raspberry Pi's CPU, used in this research.

Core/Enterprise

Windows IoT comes in 2 editions: **IoT Core** and **IoT Enterprise**. The 2 major differences lay in:

1. The target user audience
2. Application format and interface

Another difference to note is that only the IoT Core edition supports the ARM architecture.

The following table, taken from Microsoft's site, details the differences between the editions:

Windows 10 IoT Version Comparison*

	IoT Core	IoT Enterprise**
User Experience	Single UWP app running at startup with supporting background apps & services	Traditional Windows Shell with Advanced Lockdown Features
Headless supported	Yes	Yes
App Architecture supported	UWP only	UWP & Win32
Cortana	Yes (display required)	Yes
Domain Join	Azure Active Directory (AAD) only	AAD & Traditional Domain Join
Management	Mobile Device Management (MDM)	MDM & Traditional Agent Based (e.g. SCCM)
Device Security Technologies	Secure Boot, TPM, BitLocker, Device Guard, Windows as a Service, Windows Firewall & Device Health Attestation	Same as IoT Core and Defender Advanced Threat Protection (ATP)
CPU Architecture support	X86, x64 & ARM	x86 & x64
Licensing	Online licensing terms agreement and Embedded OEM Agreements, Royalty Free	Direct and Indirect Embedded OEM Agreements
Usage Scenarios	Digital signage Smart building IoT Gateway HMI Smart home Wearables	Industry tablets Point of Sale Kiosk Digital signage ATM Medical devices Manufacturing devices Thin client

Which Version to Choose?	For devices that only run a single application, manufacturers should investigate using IoT Core.	For devices that require desktop functionality, multiple apps or access to desktop apps (e.g. Win32, WPF)

*This feature list is not exhaustive but intended to highlight edition differences
** Windows 10 IoT Enterprise is Windows 10 Enterprise with different licensing and distribution

[2] http://wincom.blob.core.windows.net/documents/Windows_10_IoT_Platform_Overview.pdf

Usage Statistics

We were curious to see how popular the Windows IoT OS is in the overall IoT market. One wide and comprehensive survey[3] was conducted by The Eclipse IoT Working Group, AGILE IoT, IEEE, and the Open Mobile Alliance. The above organizations have co-sponsored an online survey to better understand how developers are building IoT solutions.

This survey was conducted annually, 3 years back, and the latest 2018 edition showed us the following relevant conclusions:

- Windows has the second largest (22.9%) share in IoT solutions development (after Linux, 71.8%).
- One of the major IoT sectors in which Windows is used is IoT gateways.
- A large share of the IoT solutions in development are built on top of ARM architecture. This probably justifies the new support of ARM by Windows IoT. We assume this may drive more extensive use of Windows IoT Core edition upon others, given the fact that IoT Core is the only Windows IoT edition that support ARM.
- Security is the top concern for developing IoT solutions.

Supported Boards

Microsoft officially suggests[4] using one of the following boards as development devices:

1. AAEON Up Squared
2. DragonBoard 410c
3. MinnowBoard Turbot
4. Raspberry Pi 2
5. Raspberry Pi 3B

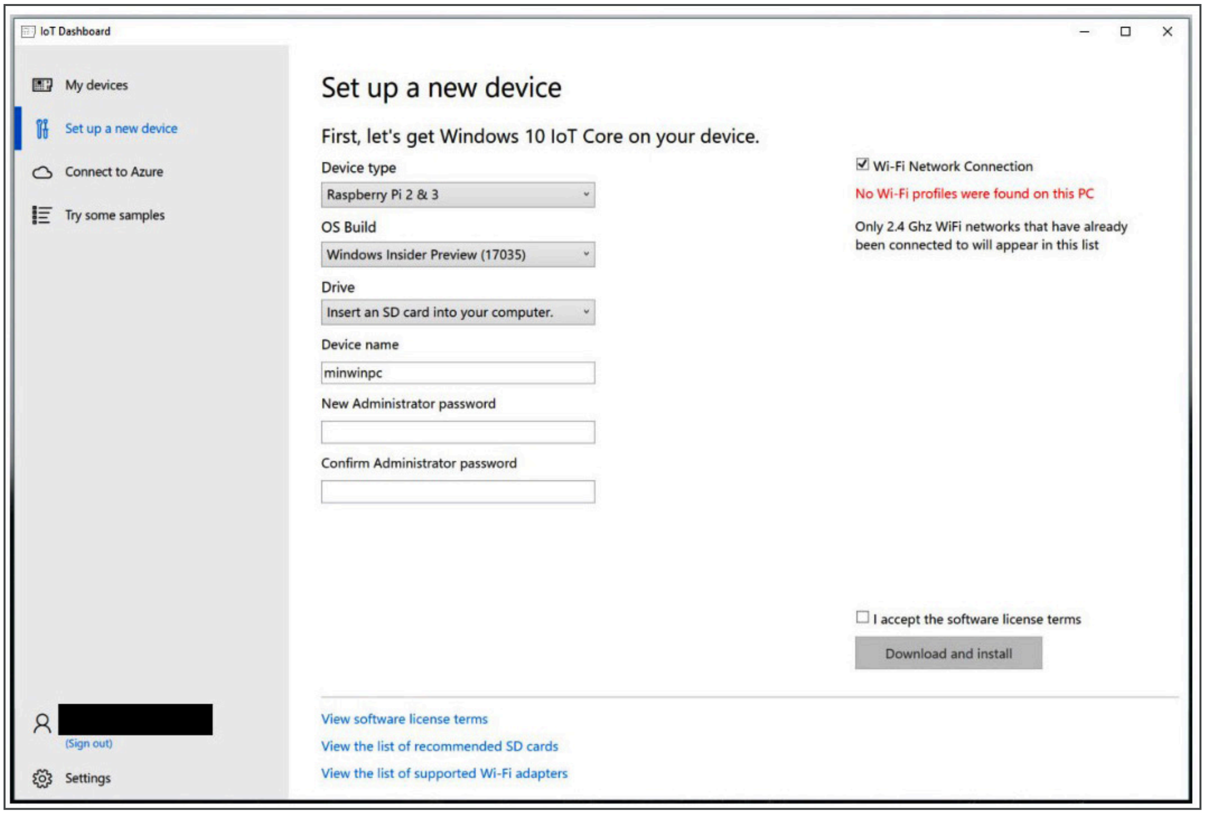
Microsoft provides the most comprehensive documentation around these devices.

Official Installation

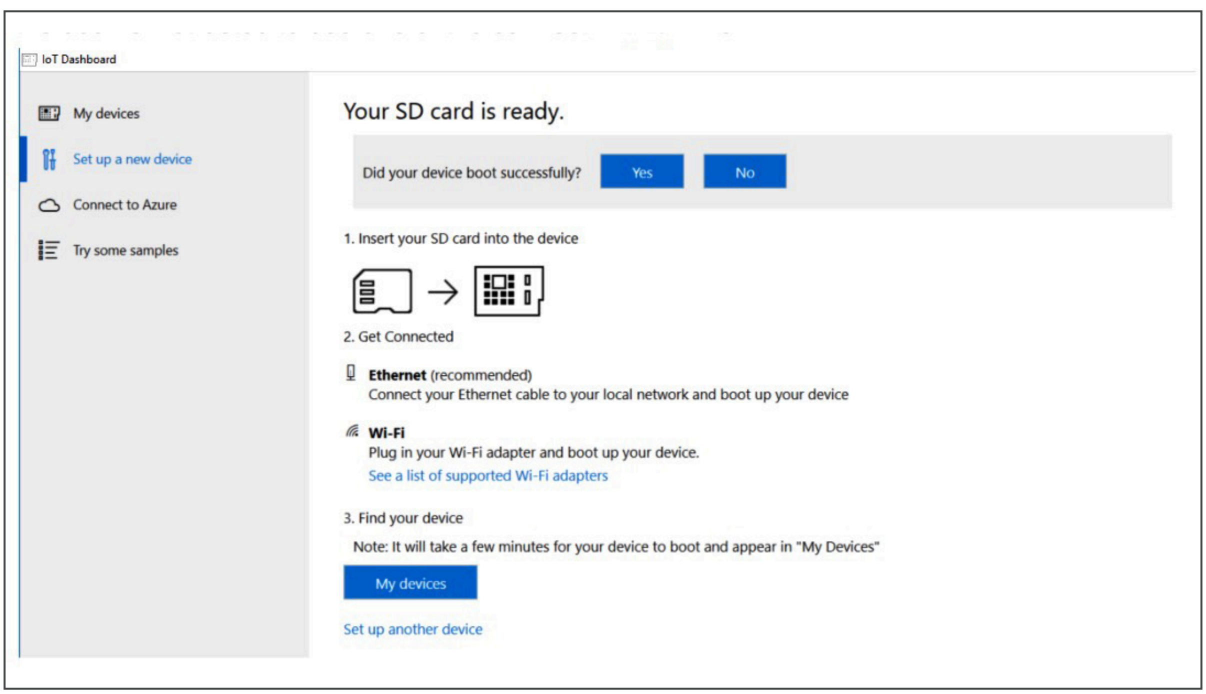
Windows 10 IoT Core Dashboard

Apart from the documentation support for the suggested development boards, Microsoft streamlines the installation process for 4 out of the 5 by providing the IoT Dashboard, a wizard tool for managing devices and OS images:

[3] https://www.slideshare.net/kartben/iot-developer-survey-2018
[4] https://docs.microsoft.com/en-us/windows/iot-core/tutorials/quickstarter/prototypeboards



Using this simple tool, one may choose an image to install on an SD card and define initial configuration for it. As the method described in this paper requires the IoT device to be connected by Ethernet and not WiFi, the following screenshot shows the relevant step where the user is instructed to use one of the connectivity options:



Stock Image / Custom Image

Windows IoT Core installation is performed using bootable images that are flushed to the board's SD card. Different images are built with various sets of features enabled. In general, there are 2 categories of images: stock images, and custom images. Given that the exploit presented in this paper is directed to the stock images and may not influence custom images, it is crucial to present the role, use cases and availability of both categories.

The **stock images** are the ones available publicly by Microsoft, and come shipped with a load of developer features enabled (called test images). One single image is available for each OS build release + Windows Insider Preview builds[5].

These images are also the only ones available for installation when using the IoT Dashboard.

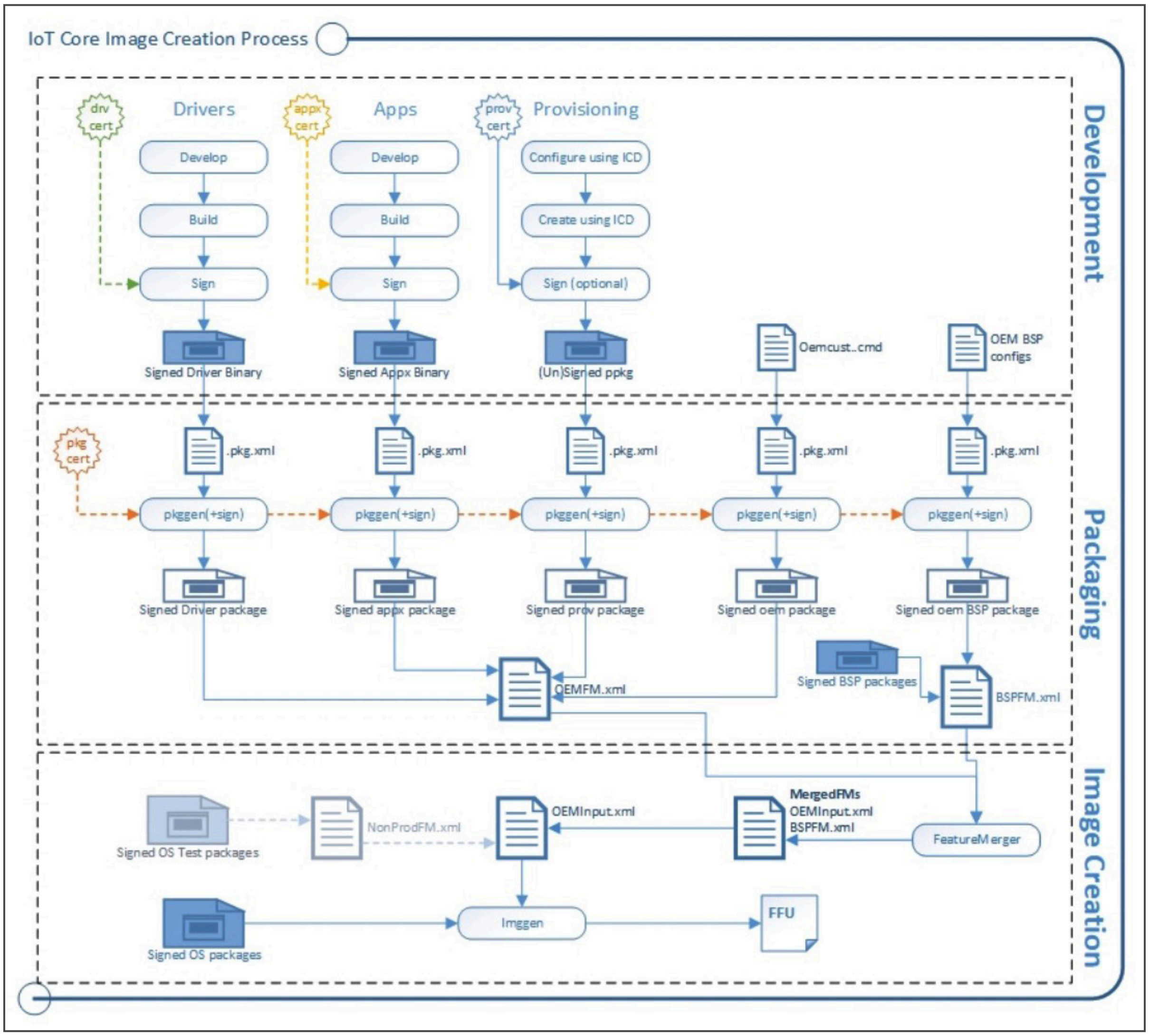
Due to the fact that this method exploits the Sirep protocol (that is handled by the Sirep service) the `IOT_SIREP` feature must be enabled as a prerequisite.

The full list of available features is maintained here.

Note: All stock images we have tested[6] have the `IOT_SIREP` feature enabled, and thus are prone to this attack.

Custom images are built by a vendor, OEM, or a manufacturing company that plans to commercialize its product. Building a custom image is a non-trivial process described here, that includes purchasing a code-signing certificate from a Certificate Authority (CA) and signing the final files. Microsoft draws the process in the following flow chart:

[5] available only to users who register to the Windows Insider Program
[6] 16299, 17134, 17661, 17763, 17692 (Windows Insider Preview)



In summary, users such as hobbyists who build DIY projects, along with internal utilizations an organization might build on top of the IoT Core platform - will likely use the stock images, that are vulnerable to SirepRAT. When commercialization comes into place, the developers are instructed to build their own custom image responsibly, disabling any developer features. Thus, it will be unlikely to exploit a released IoT Core based product.

Remote Interfaces

Out of the box, Windows IoT Core exposes several remote control and administrative interfaces. A thorough research by IBM[7] presented all the public interfaces and examined possible attack surfaces on each. Let us list these interfaces in short. An important point to notice is that using all of these public interfaces requires Administrator authentication.

[7] https://www.blackhat.com/docs/us-16/materials/us-16-Sabanal-Into-The-Core-In-Depth-Exploration-Of-Windows-10-IoT-Core-wp.pdf

Windows Web Device Portal

A web interface is served on port 8080[8]. WDP[9] lets you configure and manage your device remotely over your network. It features apps and process management, file explorer, networking information, and much more.

WDP requires HTTP authentication with an Administrator user.

Windows IoT Remote Server (Remote display)

The Windows IoT Remote Client app, installed on a connected Windows 10 desktop machine, shows the display output of the IoT Core device and allows MSTSC-like control over it. A known bug[10] currently prevents it from working on Raspberry Pi boards, but it should be operational on other boards.

This feature is installed on the stock image by default, but it must be enabled using WDP, that requires Administrator privileges.

SSH

Windows IoT Core serves SSH connection, for remote administration.

As expected, it requires Administrator login.

PowerShell

A PowerShell command can be obtained from a remote Windows computer, using the "Enter-PSSession" cmdlet.

Connection requires Administrator credentials.

Visual Studio Debugging

The Visual Studio Remote Debugger allows a remote VS instance on a debugger computer to debug running processes on the device.

Similar to the Remote Display feature, it must also be enabled from WDP by an Administrator.

Windows Hardware Lab Kit (HLK)

This is the less known command interface exposed for testing hardware using HLK. It only exists to internally serve test execution, and it is at the heart of this research paper.

---

8. https://docs.microsoft.com/en-us/windows/uwp/debug-test-perf/device-portal
9. https://docs.microsoft.com/en-us/windows/iot-core/manage-your-device/deviceportal
10. https://docs.microsoft.com/en-us/windows/iot-core/release-notes/commercial/fallcreatorsupdate

Let us introduce HLK in the next section.

What is HLK?

The Windows Hardware Lab Kit[11] (Windows HLK) is a test framework used to test hardware devices for Windows 10 and Windows Server 2006. It is composed of a server and client software: the server is called the HLK Controller and the client is a software installed on the target test device.

Vendors who aim to get their hardware and drivers officially compatible with those Windows versions, must qualify through the Windows Hardware Compatibility Program. This program requires the product to pass a set of tests composed by Microsoft. Passing those tests ensures that the product meets Microsoft's requirements.

In fact, HLK is the successor of HCK[12] (Hardware Certification Kit), and plays the exact same role. The only difference is the list of target Windows versions. While HLK addresses Windows 10 and Windows Server 2006, HCK addresses the following previous versions:

- Windows
- Windows 8
- Windows Server 2008 R2
- Windows Server 2012

The HLK controller runs a server software called HLK Studio, that provides a GUI for managing the test devices, building test scenarios (playlists), and running the actual tests.

Test Setup

Running tests on any system requires a dedicated setup. Firstly, one HLK Controller must connect to the target devices, in one of many ways. The connection can be done through Ethernet, USB, or by using Aries (a dedicated Ethernet-to-USB bridge dongle by Microsoft).

Microsoft states that choosing the right setup depends mainly on the test network and the number of test devices. The environment may either be domain-joined, or workgroup-joined.

When testing mobile/embedded/IoT devices, an HLK proxy client must be used. This proxy handles the communication between the HLK Controller and other test devices. Along with other roles, the proxy aids in discovering applicable mobile/IoT devices on the network. This is done using the device advertisement mechanism described later in this paper.

More detailed proxy setup explanation is available on Microsoft HLK docs.

[11] https://docs.microsoft.com/en-us/windows-hardware/test/hlk/windows-hardware-lab-kit
[12] And HCK, in its turn, is the successor of WTT (Windows Test Technologies)

Running Tests

After assembling the correct test setup, one can use the HLK Studio program, installed on the HLK Controller, in order to onboard test systems. When the onboarding process ends, you can run the desired tests.

HLK Studio offers management and control of the test devices. It allows grouping test devices into pools. and provides the user with full freedom in choosing the desired tests (playlists) to run on desired test systems.

Microsoft provides predefined test playlists for the certification process, along with other playlists anyone can use to test their product. Moreover, a vendor might choose to use this framework to run his own tests, making HLK an integral part of the development and testing cycle of the product.

In general, tests and their results are delivered over IP communication. While some setups involve USB connections, an IPOverUSB protocol is used. The HLK communication is done using a proprietary protocol. This is the protocol that we abuse in this paper.

Protocol Name Ambiguity: WPCon/Sirep/TShell

The correct name of the protocol addressed in this paper is somewhat unclear, at least from the point we managed to get to. It is referenced in multiple names across different sources: **WPCon**, **Sirep** and **TShell**. Let us do our best in trying to explain this ambiguity:

The different names are found on the internet and in the local firewall configuration, as well as in testsirepsvc.dll's symbols.

An official explanation for this ambiguity was not found on the internet, but it seems to come from the evolution of this service and protocol, that apparently started as a dev tool for Windows Embedded/Windows Mobile: TShell (Texus Shell). This is the protocol used for running commands from a debugger computer over IP.

The "WP" prefixes and "mobile" string found in symbol names support its existence on Windows Phone devices.

Due to the unique hardware those devices have, Microsoft offered test images for OEMs of mobile phones, with an implementation of the IPOverUSB stack. Microsoft then offered dev and debugging tools on top of this IP over USB stack.

We reached Microsoft for clarification but it refuses to share this kind of internal information to the public.

HLK on Windows IoT

The IoT device takes the role of a test system in the HLK test setup described above. It accepts test tasks from the HLK controller, and sends back the results. It also serves special ping requests (service-specific pings, not ICMP), and advertises itself using UDP broadcast packets.

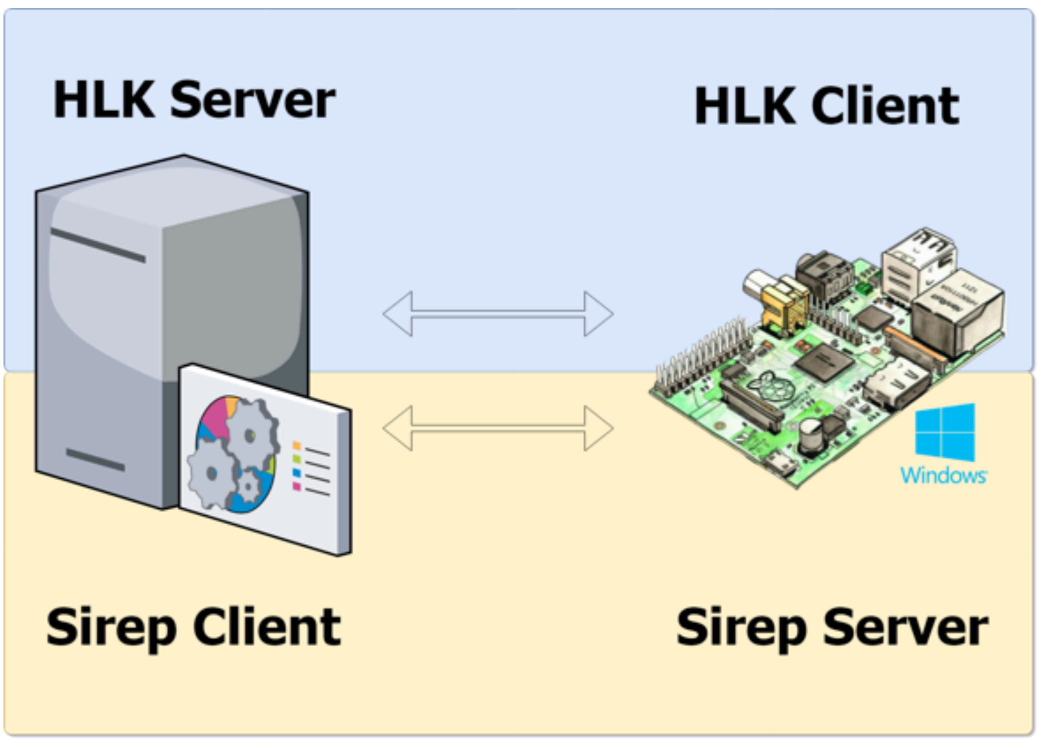
All of the above functionality is implemented in one DLL, using 3 TCP ports, that are allowed by default on the device's Windows firewall:

```
reg query
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Defaults\FirewallPolicy\Fi
rewallRules | findstr -i sirep
    Sirep-Server-Service    REG_SZ
v2.28|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=29817|App=%systemroot%\system32\svcho
st.exe|Name=Sirep Server (Service)|Desc=Sirep Server (Service)|EmbedCtxt=Sirep Server|
    Sirep-Server-Ping    REG_SZ
v2.28|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=29819|App=%systemroot%\System32\svcho
st.exe|Name=Sirep Server (Ping)|Desc=Sirep Server (Ping)|EmbedCtxt=Sirep Server|
    Sirep-Server-Protocol2    REG_SZ
v2.28|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=29820|App=%systemroot%\System32\svcho
st.exe|Name=Sirep Server (Protocol 2)|Desc=Sirep Server (Protocol 2)|EmbedCtxt=Sirep Server|
```

The Service's Main Dll: testsirepsvc.dll

The Sirep service's image path leads to this DLL at *C:\Windows\System32\testsirepsvc.dll*. The core service functionality is implemented in it, including the communication and execution of tasks sent from the HLK controller.

Notice the confusing ambiguity of the "client" and "server" terms here. The HLK server acts as a Sirep client of this Windows IoT service. To avoid confusion, we will refer to the Sirep service on the Windows IoT device as the "Sirep server" and the HLK controller as the "Sirep client", as described in the following drawing:



13

We will go over the logic and implementation of the main features this service offers.

[13] https://www.pinterest.com/pin/416301559284668200/

Network Signature

Device Advertisement

The service has a very bold network signature, that actually acted as the bait to start digging, resulting in this research paper.

By default, Windows IoT core sends periodic gratuitous UDP packets to advertise the device on the LAN. It is a simple broadcast packet with the unique device ID: a 12-character hex string (Unicode). It is sent to the broadcast address on relevant subnets and interfaces. The filtering of relevant interfaces is done similarly to how the command connections are filtered, as described further below in this paper.

In its turn, the HLK controller waits for these advertisement packets and builds a list of the available devices' IDs. These discovered devices will be available for the user to choose from, in the HLK studio app that runs on the HLK test server.

Tracing the source of those packets uncovers the sender identity: Sirep Test Service.

More specifically, it is sent from a dedicated thread, running `ControllerWSA::NameBroadcasterThreadProc`. This eventually calls `ControllerWSA::SendBroadcastForDevice`, which uses `ws2_32!sendto` as follows:

```
WS2_32!sendto:
7730b260 e92d4ff0 push        {r4-r11,lr}
0: kd> db r1 L?0x74
0324f7e0  00 c0 ff ee 42 00 38 00-32 00 37 00 45 00 42 00  ....B.8.2.7.E.B.
0324f7f0  33 00 44 00 42 00 44 00-39 00 36 00 00 00 00 00  3.D.B.D.9.6.....
0324f800  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0324f810  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0324f820  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0324f830  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0324f840  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0324f850  00 00 00 00                                      ....
0: kd> k
 # Child-SP RetAddr  Call Site
00 0324f7c0 711b7cb8 WS2_32!sendto
01 0324f7c0 711b7e3c testsirepsvc!ControllerWSA::SendBroadcastForDevice+0xd0
02 0324f880 711b7abc testsirepsvc!ControllerWSA::NameBroadcasterThread+0xb0
03 0324fad8 77ae97e2 testsirepsvc!ControllerWSA::NameBroadcasterThreadProc+0xc
04 0324fae0 00000000 ntdll!RtlUserThreadStart+0x22
```

Open Ports in the Firewall

Windows IoT Core, in its default configuration, allows several incoming connections through its firewall. 3 of them are the ones used by this service. The service continuously listens on these ports, each for a different purpose.

As mentioned above, the descriptions found for the different ports are ambiguous due to the protocol evolution. In the following list, both versions of names are given: first the IoT oriented name, then the Windows Phone oriented name:

1. **29820:** Sirep-Server-Protocol2/WPConProtocol2
   Used for the command communication exploited in this paper
2. **29819:** Sirep-Server-Ping/WPConTCPPing/WPPingSirep
   Used for the simple echo service described above
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)

Interesting Functions and Logic

Before describing the RCE method and RAT abilities exposed by the protocol, let us present the core parts of the service flow, that are at the heart of the abuse presented here. Surprisingly, the flow is very insecure and exposes a network interface for remote clients to send commands to the device, and get back their output, with no authentication at all.

Ultimately, an attacker can send commands to execute arbitrary programs on the device, that will be run by the SYSTEM account or by the logged on user, to the attacker's choosing.

Incoming Connection Authorization

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.

Let us describe the logic that decides whether a new connection is allowed to send commands to the device. This logic is implemented in the `ControllerWSA::IsConnectionAllowed` function.

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. More specifically, the function performs its checks on the SOCKADDR_IN structure that's returned from an API call to getsockname:

```
; Attributes: bp-based frame

; int __fastcall ControllerWSA::IsConnectionAllowed(#20 *__hidden this, unsigned int)
public: int __cdecl ControllerWSA::IsConnectionAllowed(unsigned int)

var_28= -0x28
var_20= -0x20
var_1C= -0x1C

PUSH.W          {R4,R5,R11,LR}
ADD.W           R11, SP, #8
BL              __security_push_cookie
SUB             SP, SP, #0x1C
MOV             R3, R1
MOV             R5, R0   ; The ControllerWSA object
MOV             R0, R3
LDR             R3, =__imp_getsockname
MOVS            R2, #0x10
STR             R2, [SP,#0x28+var_28]
LDR             R3, [R3]
MOV             R2, SP
ADD             R1, SP, #0x28+var_20
MOVS            R4, #0
BLX             R3
CBZ             R0, loc_100094C8
```

```
loc_100094C8
ADD             R1, SP, #0x28+var_20
ADD.W           R0, R5, #0x2C ; this
BL              NetworkInterfaces::IsInterestingAddress(sockaddr_in *) ; R0 - socket address
CBNZ            R0, loc_100094E8 ; SUCCESS = r0 non-zero
```

This authorization form is very permissive, and we spent time looking for the reasoning behind it. Our best guess relies on the fact described above, that is: this IoT service is a kind of a porting made from the old Windows Phone test service. At that time, testing and debugging tools were made functional only using IPOverUSB, when the phone is connected to the dev computer through a USB cable. Now, the IoT equivalent of this limiting requirement, is cable connection (as opposed to the easier option of WiFi connection).

We couldn't confirm this assumption with Microsoft as they naturally refuse disclosing this kind of internal information.

Commands Interface

A service routine accepts command packets in a binary form that's revealed below. 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:

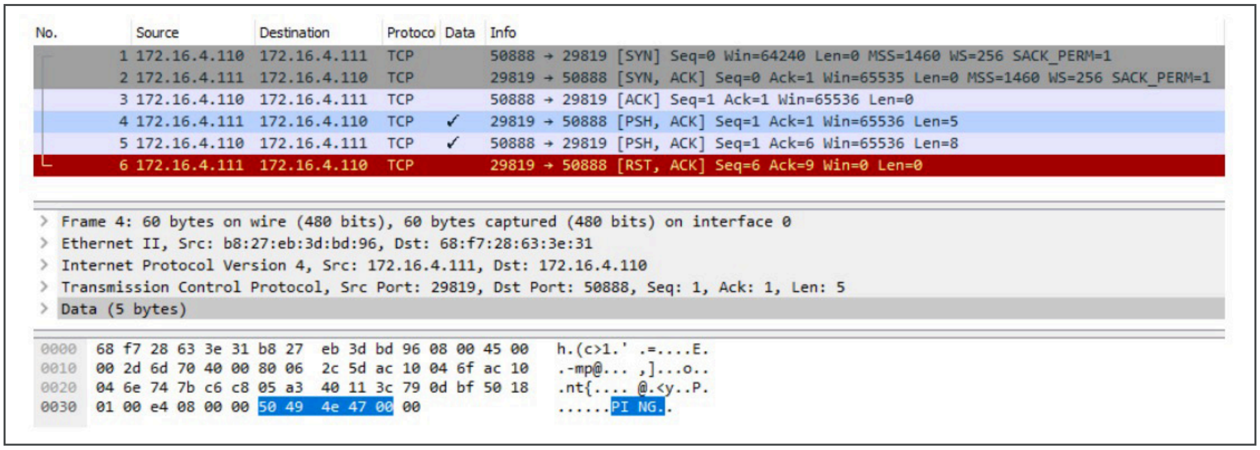
```
BL      SirepProtocol2ReceivePacketWithTimeout
MOV     R4, R0   ; if receive succeeded, r0=0
CMP     R4, #0
BLT     RecvPacketFailed ; branch if recv above failed
LDR     R3, [SP,#8] ; first packet byte (command type)
CMP     R3, #0x1E
BGT     SwitchCommandsSet ; branch to first set of commands
BEQ     SwitchGetFile ; branch to get file command
CMP     R3, #0xA
BEQ     SwitchLaunch ; branch to launch command
CMP     R3, #0x14
BEQ     SwitchPutFile ; branch to put file command
CMP     R3, #0x17
BEQ     SwitchPutFile2 ; branch to put file command #2
CMP     R3, #0x18
BNE     SwitchPipeClose ; branch to pipe close command
```

The switch tree continues to the second level:

```
SwitchCommandsSet                       ; CODE XREF: .text:1000D2D2↑j
                CMP             R3, #0x28
                BEQ             SwitchPipeClose2 ; branch to pipe close command #2
                CMP             R3, #0x32
                BEQ             SwitchGetSystemInfo ; branch to get sys info command
                CMP             R3, #0x3C
                BNE             SwitchPipeClose ; branch to pipe close command
                LDR             R1, [SP,#0xC]
                MOV             R0, R5
                BL              SirepGetFileInformationFromDevice(void *,ulong)
                B               loc_1000D35C
; ---------------------------------------------------------------------------
```

Ping

The controller listens on the Sirep-Server-Ping (29819) port and serves as an echo server, simply responding with a “PING” payload to every incoming TCP connection, then terminates the connection with RST:



Sirep/WPCon protocol abuse

In this part we will describe the structure of the different commands packets, that are then sent through a regular TCP connection to the device, in order to mimic a test server and send commands to the IoT device. The result packets that are sent back from the IoT device also come in a unique structure, that will be described.

Protocol “Handshake”

On any new connection, the Sirep service immediately responds with a packet containing a GUID in its payload. This GUID is a hard-coded constant in the service DLL, and represents the Sirep Protocol Version Guid. In our tests, the used GUID is:

```
SirepProtocolVersionGuid = 2a 4c 59 a5 fb 60 04 47 a9 6d 1c c9 7d c8 4f 12
```

So before sending the desired command packet, the attacker side must receive these 0x10 bytes.

Packet Structure

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.

00	01	02	03	04	05	06	07	08	...	Payload Length
Command Type				Payload Length				Command Data		

Command Structures

In this section, we will describe in detail some of the binary command structures. The details here are accompanied by a convenient definition file - "010 Editor" template file. It allows one to parse and build their own packet payloads with the right structure.

Listed below are the commands supported by the Sirep service and their corresponding result codes:

	Name	Code	Result Code
1	GetSystemInformationFromDevice	0x32	0x33
2	LaunchCommandWithOutput	0x0A	0x0C
3	GetFileFromDevice	0x1E	0x1F
4	PutFileOnDevice	0x14	0x01
5	GetFileInformationFromDevice	0x3C	0x3D

Info Command: GetSystemInformationFromDevice

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

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. Full details about the result structure will be given in the next section, but for the sake of this example, let's present the result data:

```
00000010  33 00 00 00 40 00 00 00  00 00 00 00 0a 00 00 00   3...@... ........
00000020  00 00 00 00 ee 42 00 00  02 00 00 00 00 00 00 00   .....B.. ........
00000030  00 00 05 05 00 00 00 00  00 10 00 00 0f 00 00 00   ........ ........
00000040  04 00 00 00 00 00 00 00  00 00 01 00 03 0d 00 00   ........ ........
00000050  04 00 00 00 05 00 00 00                            ........
```

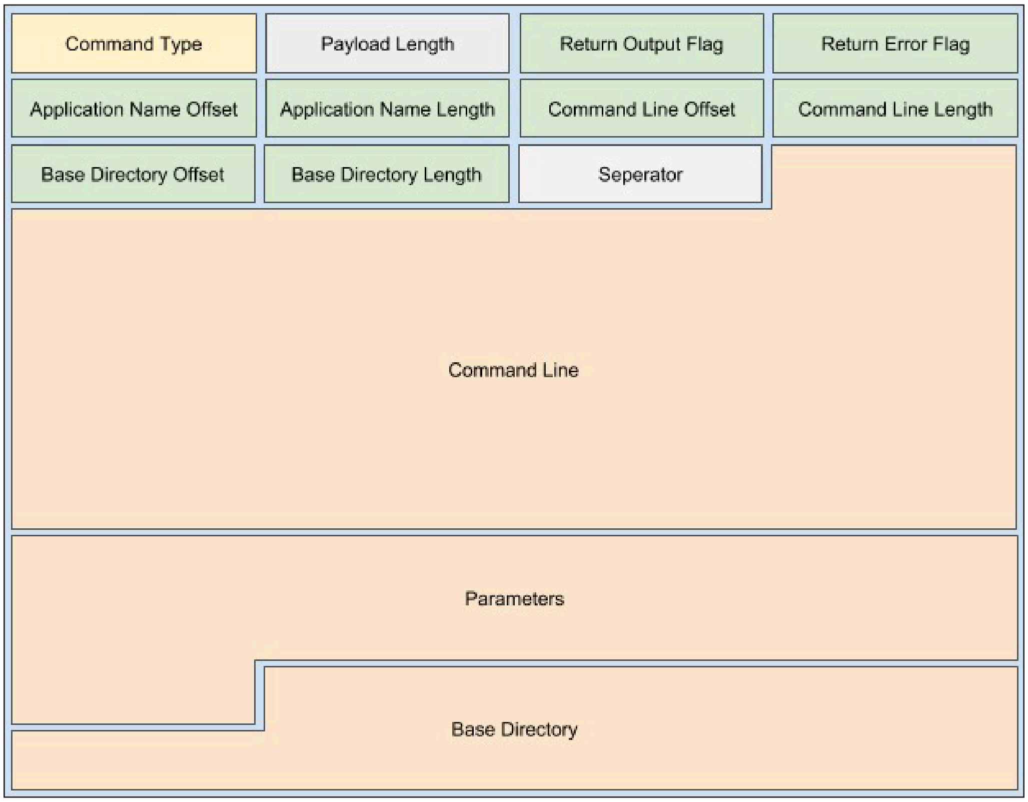
Notice the 0x0A byte - this is the Windows OS version: 10. In general, the output consists of the buffers returned from API calls to GetVersionExW (OSVERSIONINFOEXA) and GetSystemInfo. The SirepRAT tool takes care of parsing this result struct into meaningful properties.

RCE Command: LaunchCommandWithOutput

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 to CreateProcess/CreateProcessAsUser. 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.

Launch Command Packet Structure

The following drawing and table describe the packet structure an attacker can build to run processes on the target device.



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).

The following table specifies the exact offsets of each field, and an example value for each. In this example, "*hostname.exe* /?" will be run in a new process with the currently logged on user, having a current directory set as "*C:\Users\Public*". Both the output and error streams will be returned in the result (result structure explained below), as specified in the ReturnOutputFlag and ReturnErrorFlag fields.

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

All that is left to do, is send this payload through a TCP connection to port 29820 of the target device (after performing the protocol "handshake" described above).

Advanced Options

The Launch command supports the following advanced options as well:

1. Command line parameters - corresponds to `lpCommandLine`
2. Base directory to run from - corresponds to `lpCurrentDirectory`
3. Result records - it is possible to specify which of the output/error streams we want in the response (details further below)
4. Execution user context:
    a. Run as SYSTEM - the default behaviour
    b. Run as the currently logged on user - corresponds to using `CreateProcessAsUser`
    To use this feature, the command line string must start with the following constant substring:

```
<AS_LOGGED_ON_USER>
```

Download Command: GetFileFromDevice

This is a simple command, requiring 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		

File Command: GetFileInformationFromDevice

This 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 (the returned data is similar to the WIN32_FIND_DATAA structure and is parsed by SirepRAT).

The command structure is as follows:

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		

Upload Command: PutFileOnDevice

This 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 that are described below.

WriteRecord Structure

The data to write is given in WriteRecord structures that are appended to the command payload.

These records have 2 possible types: one is a regular chunk of data, and the other is the last chunk, as shown in the following snippet from the Sirep template file:

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

Command Structure

Unlike all other commands, the “Payload Length” header field specifies the remote file path length and not the whole payload length.

The remote path and the WriteRecord data records are composed in the following structure, to form the full command:

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!		

Result Packet Structure

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	

For example, after running the Launch command example given before, we get 3 result records:

```
1    00000010  01 00 00 00 04 00 00 00  00 00 00 00              ........ ....

2    0000001C  0b 00 00 00 36 00 00 00                          ....6...
     00000024  0d 0a 50 72 69 6e 74 73  20 74 68 65 20 6e 61 6d  ..Prints  the nam
     00000034  65 20 6f 66 20 74 68 65  20 63 75 72 72 65 6e 74  e of the  current
     00000044  20 68 6f 73 74 2e 0d 0a  0d 0a 68 6f 73 74 6e 61   host... ..hostna
     00000054  6d 65 0d 0a 0d 0a                                 me....

3    0000005A  0c 00 00 00 04 00 00 00  01 00 00 00              ........ ....
```

Record #1 should be interpreted as follows:

1. Result record type is 00000001
2. Result record data length is 00000004 bytes
3. Result record data is 00000000 - meaning that running the command ended with `HRESULT = S_OK`

SirepRAT

Based on the findings we have extracted from this research about the service and protocol, we built a simple python tool that allows exploiting them using the different supported commands. We called it SirepRAT.

The tool code, along with related artifacts, can be found on SafeBreach-Labs github repository.

It features an easy and intuitive user interface for sending commands to a Windows IoT Core target. It works on any cable-connected device running Windows IoT Core with an official Microsoft image, or any other image with `IOT_SIREP` feature enabled.

SirepRAT Features full RAT capabilities without the need of writing a real RAT malware on target.

Command Parsing: Template Files

Along with the python tool comes a set of 010 Editor template file, used to parse payloads for the different command supported. Example command payloads are also given.