SNE Student project
Information retrieval from a TomTom Nike+ smart watch

Leendert van Duijn & Hristo Dimitrov

June 1, 2014

TomTom Nike+ sport watch
Image: `http://ecx.images-amazon.com/images/I/414rF-Fk7HL._SY300_.jpg`

Abstract

Smart watches can be used to collect data, an example is the Nike+ Sportwatch. It can collect GPS data so you can keep track of your exercises. This information can be a wealth of information in an investigation to show that someone did, or did not, visit a particular location.

While there are methods to recover data from smart phones we sought to investigate the Nike+ Sport watch for which no methods were available. By analyzing the client software and communications protocol we have identified several methods to obtain data. From storing the data streams generated by the official software to low level USB access to the raw commands.

Within our project we have created a proof of concept that, though incomplete, can gather data from the watch in a non destructive manner.

Contents

List of Figures

1 Introduction

Smart watches are portable devices capable of more than a traditional watch, they can integrate with other mobile devices to provide interactivity and use sensors to gather information about the environment or their user. The devices log this information, to be used in reviewing your exercise performance. This data is then stored on the physical device for later analysis, either by software on a workstation or as a service by a vendor.

We intend to investigate the TomTom Nike+ sport watch, which is a smart-watch designed for runners by Nike+ in collaboration with TomTom. It has GPS functionality and it comes along with a foot sensor. It is charged, configured and read out via USB. The device contains information about its owner and also tracking information provided by the GPS functionality. This type of data can be useful in legal or civil cases. The focus of this research is to extract that data from the smart-watch in a forensic way and also investigate if the recovered data can be used as evidence in court.

1.1 Research Questions

Our Research will focus on these research questions:

- What forensically interesting data is being stored on a TomTom Nike+ sport watch?
- How can this data be retrieved in secure and consistent way?
- Can the recovered data be falsified, or is it reliable enough to be used as evidence in court?

2 Related Work

There has been research on mobile device forensics ie. [1],[2]. Methods have been developed to obtain information from these devices. But little work can be found on smart watches as they are new and have not been extensively researched.

There is some work done on the Galaxy Gear Smart watch by [3] by Erin Poremski in 2013, in this project it is shown some data can be revealed about the used. Though the device acts primarily as a secondary screen to an Android phone.

For the Nike+ Sportwatch no such works could be found, nor alternate clients that could be used to collect data from the watch. In this project our goal is to change this, and create the beginnings of a forensically sound method of accessing the Sportwatch data.

3 Approach

3.1 Methodology

For conducting this research the following steps and methods were used.

First of all the USB interfaces that were implemented and enabled on the sport watch were investigated. This was needed to get an overview of what communication channels are available on the device and how can they be used to extract data from it. Then all of the network and USB traffic generated by the watch, the client software and the web user interface was sniffed and investigated. Since the network traffic was encrypted, a Man-in-the-Middle attack had to be performed in order to reveal its content. Once it was clear how the data traffic flows throughout the different components, the next step was reverse engineering of the client software. Executables and DLLs were decompiled and investigated in order to understand the communication with the watch over the USB protocol, what methods to access it.

Finally a Python client was created to start a session with the watch and request data from it over USB.

3.2 Tools

In order to get a better understanding on how the watch communicates with the client software and the back-end servers the different types of traffic that were generated after connecting the watch to a computer needed to be inspected. Since the watch connects to the computer via USB, not only the network traffic, but also the USB traffic needed to be recorded and analyzed. For this to be done, the following tools were installed and set up:

- Burp Suite [4], as an intercepting web proxy
- Tcpdump [5], to capture USB and IP
- Wireshark [6], an aid to do visual analysis of IP and USB traffic
- python pypcap version 1.1 [7], an aid in inspecting USB traffic
- python pyusb version 1.0.0b2 [8], to communicate with the watch
- IDA Freeware version 5.0 [9], to analyze the official client software

4 How does the watch work

4.1 Usage of the sport watch

Since this device is designed for runners and is meant to support them during their training, it works in terms of runs. This means that the GPS and the sensor

functionality of the device is not enabled by default, but the user can only enable it during a so called run. Those runs represent actual trainings separated in time. After a run is completed, the watch will display statistic data about it, like average speed, best lap time or total run duration. GPS and sensor samples during the entire run are also stored on the devise, so that the user can later see his exact route and statistics about every single point at time during the run no his computer.

4.2 Communication and data exchange

The TomTom Nike+ sports watch requires installation of a client software package on the computer to which it is connected. This software is used to exchange information with the watch (See Figure 1.) over Human interface device (HID) USB protocol. The date extracted from the watch is then uploaded to a back-end server over HTTPS. The user interaction with that data is done via a web-based front-end application which communicates with the back-end server also over HTTPS. From that web interface the user can visually observe the data generated on the watch and also create or edit configurations and submit them to the watch.

The user experience flow for the connection of the watch is as follows:

- The user connects the watch to the PC
- The OS detects an HID and USB mass storage device
- The Nike+ Connect Deamon detects the watch
- The OS issues a pop up asking about the new storage device
- The Connect software extracts the data from the watch and uploads it to the server
- The Connect software starts the browser UI

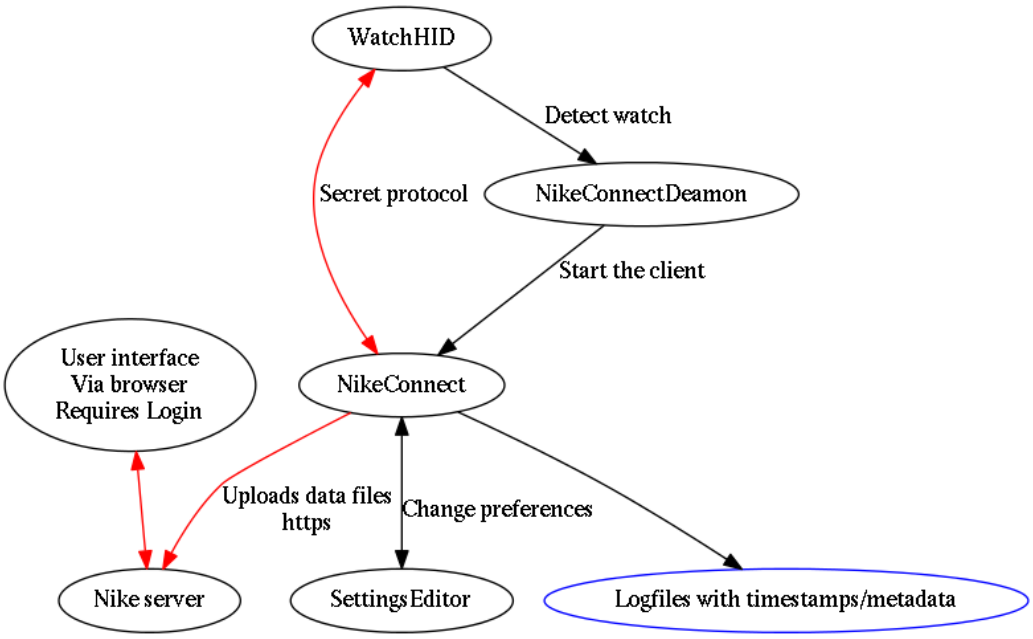

Figure 1: Data exchange between the sports watch and the other components of the platform.

5 Investigation and findings

5.1 Investigated device interfaces

After connecting the watch to a computer two USB devices are being detected.

The first one is a USB mass storage device which is of size 64 kilobytes and its file system is FAT12. Most of this storage is empty and there is a URL of the Nike website at the end of the partition . During all of the experiments nothing was written to that partition.

The last blocks of the USB mass storage device:

```
5b496e7465726e65~7453686f72746375~|[InternetShortcu|
745d0d0a55524c3d~687474703a2f2f67~|t]..URL=http://g|
6f2e6e696b652e63~6f6d2f73706f7274~|o.nike.com/sport|
77617463682d3032~0d0a000000000000~|watch-02........|
```

The second one was an HID device. This was the device which communicates with the Nike+ Connect Deamon software over the HID USB protocol. The extraction of the data from the watch had to be done over that device.

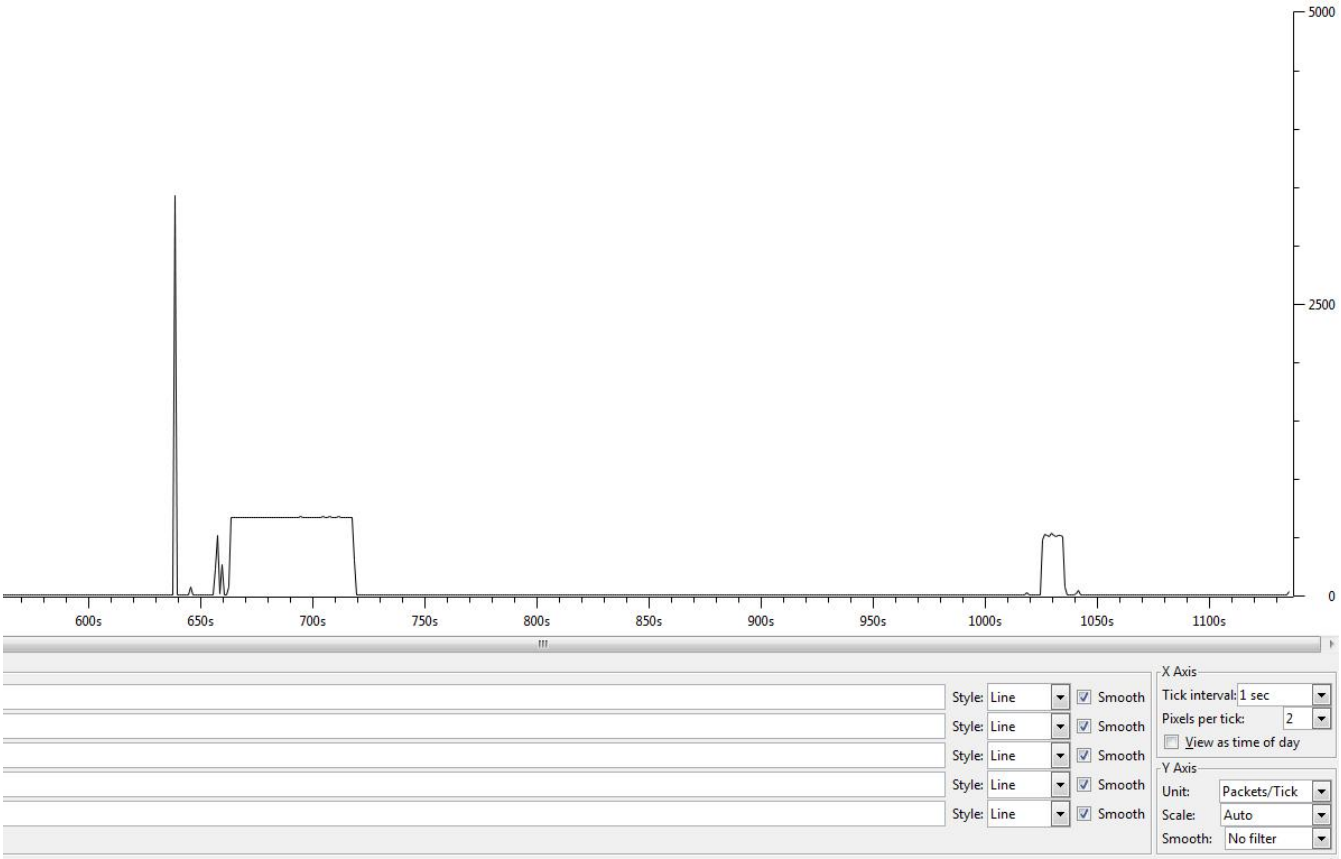

Figure 2: This graph zooms into the data transfer rates over the HID USB protocol after connecting the sport watch to the computer. Vertical axis - number of packets per second; Horizontal axis - time;

5.2 USB traffic inspection

Before the device was connected to a computer for the first time to investigating the traffic it generates, some sample runs were generated on it.

After connecting the sport watch to the set up Virtual Machine, the communication with the client software began. By looking into the captured USB traffic we noticed several things, first of, a concentrated packet spike around the moment the device was connected to our VM. Following this, around the time the Nike software had become active a significant amount of traffic started for the endpoints associated with the watch HID. And then a second smaller event appeared (See Figure 2.).

Filtering out the non HID traffic, we looked at the start of the first traffic event. At this point the Host sent the watch a packet with 64 bytes of unrecognized information, on receiving this packet the watch responds in two parts, this can be seen in Figure 3.

No.	Source	Destinatior	Length	Leftover Capture Data	Info
284	host	12.1	128	09022908000000000000000000000000000000000000000000...	URB_INTERRUPT out
287	12.1	host	64		URB_INTERRUPT out
288	12.1	host	128	0104294300050000000000000000000000000000000000000000...	URB_INTERRUPT in
289	host	12.1	64		URB_INTERRUPT in
296	host	12.1	128	09022321000000000000000000000000000000000000000000...	URB_INTERRUPT out
299	12.1	host	64		URB_INTERRUPT out
302	12.1	host	128	010a23536111b400003840000000000000000000000000000...	URB_INTERRUPT in
303	host	12.1	64		URB_INTERRUPT in
304	host	12.1	128	0902be26000000000000000000000000000000000000000000...	URB_INTERRUPT out
307	12.1	host	64		URB_INTERRUPT out
308	12.1	host	128	0102be01000000000000000000000000000000000000000000...	URB_INTERRUPT in
311	host	12.1	64		URB_INTERRUPT in
312	host	12.1	128	09028431000000000000000000000000000000000000000000...	URB_INTERRUPT out
315	12.1	host	64		URB_INTERRUPT out
316	12.1	host	128	0102840000000000000000000000000000000000000000000...	URB_INTERRUPT in
319	host	12.1	64		URB_INTERRUPT in
320	host	12.1	128	0902e148000000000000000000000000000000000000000000...	URB_INTERRUPT out
323	12.1	host	64		URB_INTERRUPT out
324	12.1	host	128	0102e10000000000000000000000000000000000000000000...	URB_INTERRUPT in

Figure 3: The early packets to the HID device, the highlighted byte is the Opcode

The second packet containing again 64 bytes of payload information. In aligning the request packets with a Log-file from the client software, which was created on the computer, it turns out that the fourth byte is this 'opcode'.

5.3 Network traffic inspection

After the client software retrieves all the data over the USB protocol it connects to the Nike back-end server over HTTPS. Luckily the local Certificate Authority store is being used by the Nike+ Connect software, so Man-in-the-Middle attack was possible to reveal the generated traffic.

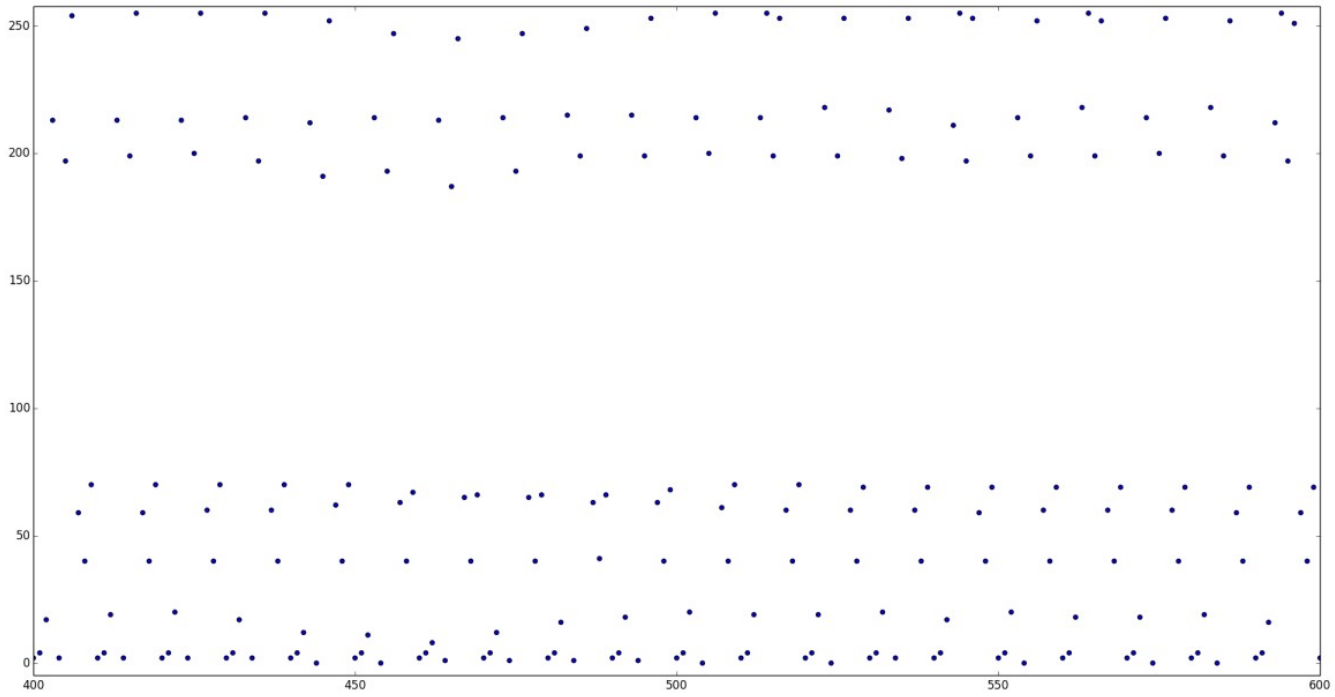

Figure 4: This graph zooms into the visible sample patterns in the rawGPSdata.bin file. Vertical axis - numerical value of the unsigned byte; Horizontal axis - offset in file;

From the sniffed traffic it was visible that a raw GPS data binary file

`rec_<run number>_rawGpsData.bin`

, an acceleration data binary file

`rec_<run number>_accelData.bin`

and a XML file

`rec_<run number>_runXML.xml`

are being sent to the server for every run. The XML file contains meta-data about the run (See Appendix A.). By looking at the graph of the numerical representation of the raw GPS data file a clear repeated pattern is visible (See Figure 4.). An auto-correlation plot (See Figure 5.) shows that there is a big possibility that those patterns represent data samples since they are similar to each other. The size of the pattern according to these numbers is 10 or 110 bytes, this suggests that the samples could very well be raw, uncompressed, GPS data for that run, with space to includes latitude, longitude and elevation.

This theory is supported by the the communication from the server to for web based interface the same XML file with meta-data is being send per run, together

with instead of raw GPS data file, a JSON file containing all the GPS data. The JSON file contains all the GPS samples of the run and each one of them consists of latitude, longitude and elevation. For the displayed run the foot sensor was not used. It is possible that the format changes if it is being used.

Example of the GPS samples of a run that can be found in the JSON file which is being send from the server to the web user interface:

```
{"lat":40.814583,"lon":24.69021,"ele":-5.42},
{"lat":40.814625,"lon":24.690199,"ele":-5.74},
{"lat":40.814667,"lon":24.690187,"ele":-6.06},
{"lat":40.81471,"lon":24.690174,"ele":-6.22}
```

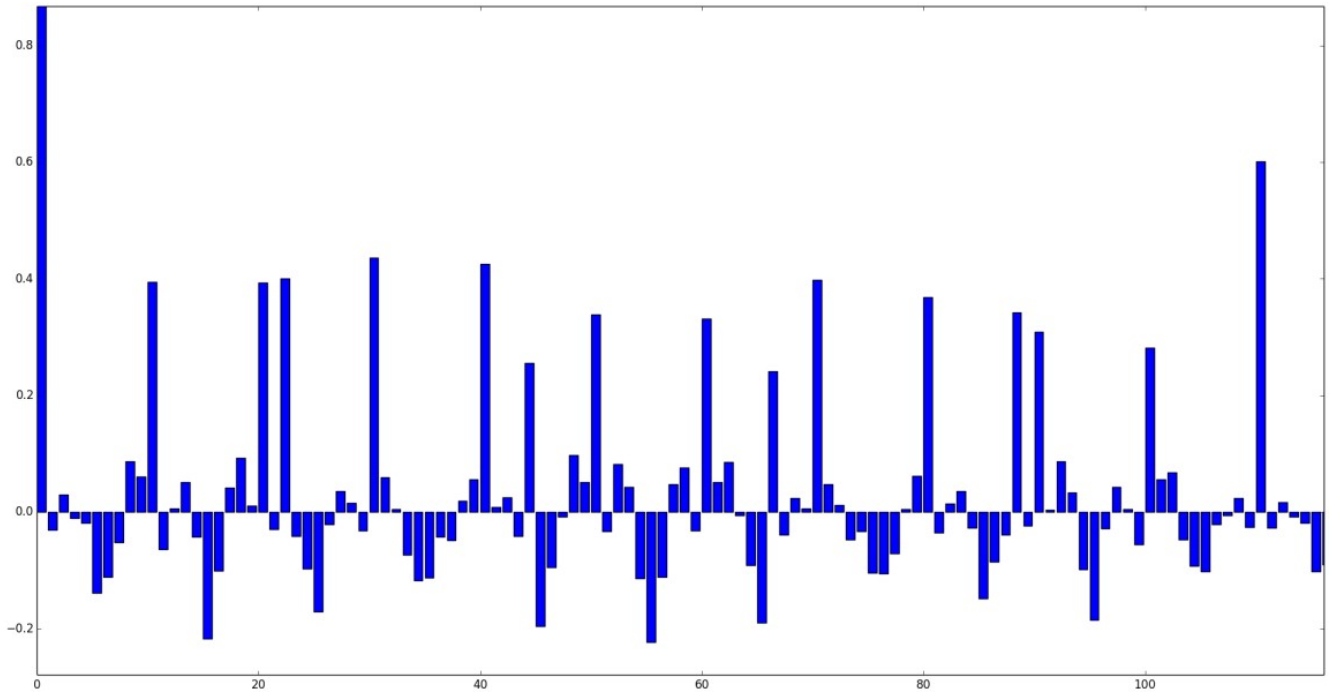

Figure 5: This graph zooms into the auto correlation between the samples in the rawGPSdata.bin file. Vertical axis - correlation coefficient; Horizontal axis - shift in bytes;

5.4 Inspection of executables

The client software can be divided into several components. There is the Connect Deamon, this piece of software runs in the background until such time a watch is connected to the system. On detecting the watch it will start the Connect software. The Connect software then loads its plug-ins, one for each supported device. Once the software is loaded it will automatically connect to the watch and open the

user interface to to inspect and adjust settings for the watch. In the background the software, using the plug-in, will query and extract data from the watch to be uploaded to the Nike services. This process can include updates to the watch and is initiated automatically. After the upload is complete a browser is opened to the Nike web interface to interact with both the uploaded, historical and community data. For this interface authentication is required, though session may remain valid until logging out.

There is authentication of the client software or user to the watch in order to connect to the watch, access the settings or upload data to Nike. Our experiments show the software obeys local Internet settings, a proxy server specified in Internet Explorer is used in both the user interface when interacting with the Nike service, as well as the uploading of data prior to authentication.

The software included 5 plugins, they are loaded in fixed order.

- `EspPlugin.dll`
- `SportBandPlugin.dll`
- `SportWatchPlugin.dll`
- `FuelBandPlugin.dll`
- `FuelBandSEPlugin.dll`

All of these plugins export the base functions

- NikePluginClose
- NikePluginGetAttributes
- NikePluginGetDeviceFamilyAttributes
- NikePluginOpen

The FuelBand and FuelBandSE plug-in also export several functions which appear related to the communications to HID devices, creating and destroying workouts.

The SportWatch plug-in only exporting 4 functions lead us to examine the code using IDA, which reveals that there are more than just these functions being shared with the Connect software. In an attempt to identify any regions of interest in the code the IDA string discovery revealed a large block of text conforming to the XML format. The content of this string (see appendix B) appears to be an interface description revealing several commands to communicate to the watch. Reconstructing an interface to this plug-in should allow read write access to both settings and data on the watch.

When inspected the other 4 plug-ins also contained such an XML document, extracting these using grep on the DLL file obtained mostly correct XML files.

Since each plug-in contains their own interface description it might be possible to create an alternate interface than the Connect software.

An alternate interface using the official plug-ins, though powerful is beyond the scope of this project, if only for the time required to create a compatible environment for the plug-ins to load into. As such analysis of the Sport Watch plug-in was focused on the protocol to communicate with the watch.

5.4.1 Log files

The Connect software and plug-ins generate several log-files with timestamps, they are stored in `C:\ProgramData\Nike`. In this directory we found several files:

- `connect.log`
- `esp.log`
- `fuelband.log`
- `fuelbandse.log`
- `sportband.log`
- `sportwatch.log`

They contain a timestamped header, for the Connect software

```
Log file opened at Tue May 13 12:47:41 2014
Connect version 6.3.18
Built Apr  9 2014 :: 08:40:24
Running on Windows 6.2.9200
```

And for the Sport Watch plug-in

```
Log file opened at Tue May 13 12:47:42 2014
Sport watch plug-in version 6.3.18
Built Apr  9 2014 :: 08:37:04
```

The log is line based, with each event given a severity in the number of leading `*`, a time-stamp, the source file the logging statement originated in, a line number and a message.

These messages vary from a notice that the plug-in is started, the failing of an HTTP request to the progress made in 'Submitting' certain numbers.

```
*[14:52:44.676][SportWatchCommands.cc 415] Submitting 08
*[14:52:44.682][SportWatchCommands.cc 383] Completing 08
*[14:52:44.682][SportWatchCommands.cc 398] Firmware version 'C1280'
...
*[14:52:44.705][SportWatchCommands.cc 1841] Submitting E1
```

```
...
*[14:52:46.336][SportWatchCommands.cc 1785] Completing E1
*[14:52:46.336][SportWatchCommands.cc 1792] Serial number: HA1094M01303
```

In order to translate these codes to actual commands we started exploring the executable code using IDA and looking for the source of the `Submitting` string. As it turns out there are several occurrences of the string `Submitting %02X`. These are referenced in preparing arguments for functions which on closer inspection do indeed append a message to the log-file. This function looks into its arguments for which number to log in place of `%02X`. This argument is pushed onto the stack preceding the reference to the format string.

Several of these references where in code where logically only 2 conditional jumps removed from error messages such as `doSetDesktopData need data parameter`. By inspecting references to these snippets of code we identified a function which appears to initialize several structures which include the functions generating all these log statements closely coupled to a string closely resembling commands from the XML file.

This links functions implemented by the plug-in to their logging statement. Further inspection of these functions did not reveal contradicting error messages. In several cases reaffirming that a certain name does indeed belong to that function.

The search on the logging statements on the word `Submitting` did not exhaust the list of names and coupled functions, further exploration of the code revealed several more commands and codes.

Due to the amount of code we could not fully reconstruct how they work on a higher level. It did become clear that the 'code' is not only used for logging. The code is, after logging it further submitted to low level functions which contained too little debugging statements or known function/library calls for us to reverse engineer. The consistency in using the code as an argument however does reinforce the importance of this single byte value.

During our analysis we gathered enough information to build a translation from these codes to their textual meaning, see appendix C. The full list can be found in appendix D.

When this translation is applied to one of our logfiles we can make sense of what the software is doing, as it is doing it.

- *[14:52:46.336][SportWatchCommands.cc 1785] Completing E1::serial
- *[14:52:46.336][SportWatchCommands.cc 1792] Serial number: HA1094M01303
- *[14:52:46.345][SportWatchCommands.cc 1600] Completing 14::desktop-read
- *[14:52:46.345][SportWatchCommands.cc 1131] Submitting 35::option-age
- *[14:52:46.349][SportWatchCommands.cc 1348] Completing 37::emped-info
- *[14:52:46.352][SportWatchCommands.cc 1093] Completing 35::option-age
- *[14:52:46.360][SportWatchCommands.cc 725] Submitting 21::time
- *[14:52:46.364][SportWatchCommands.cc 646] Completing 21::time
- *[14:52:49.239][SportWatchCommands.cc 544] Submitting 10::readWorkouts

- *[14:52:50.952][SportWatchCommands.cc 519] Completing 10::readWorkouts
- *[14:53:00.644][SportWatchCommands.cc 457] Submitting 11::eeprom-erase
- *[14:53:12.163][SportWatchCommands.cc 427] Completing 11::eeprom-erase

5.4.2 The serial number

The serial number in the logfile looked familiar, close inspection of the watch reveals that it is the same number as printed on the bottom above the TomTom logo. Though our analysis hints that this number might be manipulated it becomes harder to do so when it redundantly printed onto the physical device.

This link might be of value in linking the watch to a computer, with a timestamp of when the number was obtained.

5.5 Communicating with the USB protocol

In order to show it is possible to build an alternative client we attempted to make a proof of concept to talk to the watch, having concluded that using the official binaries would be an complex and Windows only solution a low level approach was chosen.

5.5.1 Packet Replaying

Using the python toolset `https://bitbucket.org/dwaley/usb-reverse-engineering` an attempt was made to replay an entire conversation between the watch and client software. However this did not succeed as the machine running the experiment ran into a kernel panic before any results where observed.

The experiment was run only once in favor of attempting to crafting packets based on the capture, in a simpler attempt to get interaction with the watch, even if limited to simpler features.

Having identified the first request submitted packet to the watch, which according to the log file is the 'Get the MSP firmware version number' command. This command has an opcode of 8. In the packet this opcode is present, at the same offset where the following stream of packets the opcodes appear in order, as logged.

Listing 1 : Replay the Version command

```
#!/usr/bin/python2
NIKE_VENDOR_ID = 0x11ac
NIKE_PRODUCT_ID = 0x5455

import usb.core
import usb.util
import usb.control
```

```
dev = usb.core.find(idVendor=NIKE_VENDOR_ID, idProduct=NIKE_PRODUCT_ID)

if dev.is_kernel_driver_active(0):
    print('Detaching kernal driver\n')
    dev.detach_kernel_driver(0)

cfg = usb.util.find_descriptor(dev, bConfigurationValue=1)
iface = cfg[(0,0)]

bytes_num = [
        0x09,0x02,0x29,0x08,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00]
data_raw = "".join(map(chr, bytes_num))

iface[1].write(data_raw)
version = iface[0].read(64)

letter = chr(version[3])
code = version[5]<<8 + version[4]

print("Version: %s%d"%(letter,code))
```

With this code python will replay the packet to obtain a version number, and will print what our analysis shows is the version from the response. The byte at offset 3 an ASCII character, and 4 and 5 a big-endian 16Bit integer[1].

After this success, after countless failed attempt ranging from simple errors, to kernel panics, to non responsive watch, an attempt was made to replay a more interesting command the 'read-eeprom' command.

From a dump where actual data was synced the packet was copied, and noted that there was more than one response to read all the data. Adjusting the previous snippet to do just this becomes

Listing 2 : Replay the ReadEeprom command

```
#!/usr/bin/python2
NIKE_VENDOR_ID = 0x11ac
```

[1]This is based on the output logged to file, we did not have an other version device to confirm this

```
NIKE_PRODUCT_ID = 0x5455

import usb.core
import usb.util
import usb.control

dev = usb.core.find(idVendor=NIKE_VENDOR_ID, idProduct=NIKE_PRODUCT_ID)

if dev.is_kernel_driver_active(0):
    print('Detaching kernal driver\n')
    dev.detach_kernel_driver(0)

cfg = usb.util.find_descriptor(dev, bConfigurationValue=1)
iface = cfg[(0,0)]

bytes_num = [
        0x09,0x05,0xb3,0x10,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00]
data_raw = "".join(map(chr, bytes_num))

while True:
    #Clean outstanding
    try:
        outs = iface[0].read(64)
        print("Outstanding:")
        print(outs)

    except:
        break
print("Writing packet")
iface[1].write(data_raw)

o = open("raw1.OUT","wb")
while True:
    data = iface[0].read(64)[:]

    print(data[:])
```

```
o.write("".join(map(chr, data)))
```

When this is executed the watch should respond with eeprom data, the first response was quite short, likely due to the eeprom being 'erased' by the official client on an earlier experiment.

After making a small run, the experiment was repeated. This time the resulting datastream consisted of several packets. In order to confirm that this data is indeed eeprom data containing GPS coordinates we looked into its contents. Running the strings utility reveals an interesting line of ASCII text.

- `GSD4e_1.0.0-P1_RPATCH.03.2-P0- 11/11/2011 315`

This string has been encountered in the file uploaded to the Nike API, under the filename `rawGpsData.bin` (As seen in section 5.3). Though in this raw USB stream there are 65 bytes extra data prior to this string as compared to the file recovered earlier.

Due to time constraints we have not been able to further decode the response, to remove any USB/transfer headers from each packet.

In order to verify the results, and confirm the non-destructive nature of the read-eeprom command prior to creating any new run data or any other experiments the watch was disconnected from the workstation for several hours. When reconnected the experiment was repeated and the raw packet streams hashed. Though some header data from the communications protocol could be expected to differ, regardless of the identical payload, the output of each experiment is a bit perfect match. Whereas the official client will alter the watch state, by erasing the eeprom data that is uploaded.

```
The sha256sum of both extractions, in order of extraction
26cbcb357848a67109f565fdfdce100b33e12c843575544d7a27dbb8c09a7df5  eeprom_shortwalk.respo
26cbcb357848a67109f565fdfdce100b33e12c843575544d7a27dbb8c09a7df5  raw1.OUT
```

6 Watch data reliability

An interesting fact about this particular GPS enabled watch is that it does not obtain its time over GPS, instead the time and date are being set manually. An experiment was conducted to check if run data can be falsified and runs can be created in the past. By setting the time to 1 month ago, a valid run was created in the past. By intercepting the traffic created for that run and reviewing it in the web user interface it seamed that the run was indeed created one month ago. This means that run data could be faked and this should be taken in consideration when such data is being used in court.

During conducting the experiments at one point the sport watch was connected to a newly clean install virtual machine. Without having to enter any identifying or authenticating information the client software was able to connect to the server

and upload the newly generated run data. From the sniffed traffic it was visible that user identifying data like e-mail address, user name, date of birth, gender and wight along with authenticating cookie are passed to the server in a session initiating request. Since the VM was newly installed, this data must have been stored on the watch itself. This means that the watch reports user identifying data and a cookie used for authenticating to he server as being the watch to every computer that it is connected to. Because of this one could steal that data by connecting the watch to untrusted machine and if he could create fake run data, he might be able to upload those runs to the Nike server. Since this concerns adding fake data to the server and not to the watch, it falls outside the scope of this research and it has not been tested.

7 Future work

There are multiple directions in which this research can be extended.

Customized software can be build with the purpose of extracting data from the watch. This can be done in two different ways depending on if the client software could be trusted or not and if there is enough knowledge to decode the raw format in which data is being send over the USB protocol.
The first way would be to implement a USB driver interface which communicates with the watch over USB and extracts all needed information form it. This should be done in a forensic way, meaning that all the extracted data should be hashed, write protected and backed up. This method excludes the usage of the client software, so also a tool for interpreting the data should be build. The second way would be to implement a Mock server and have the client software connect to it and report all of the collected data to it. This is possible, because the client software uses DNS to retrieve the IP address of the server to which it connects and it also uses the local CA store to verify the identity of the server. So a CA for the Mock server can be included and DNS can be used to redirect the client software to it. With this implementation the client software will be used to decode the USB communication data to XML and raw GPS data files, but still some decoding will be needed for the raw data format which will holds the entire run.

Due to time restrictions no physical attempts on the device were made to obtain privileged access or access internal memory. Disassembling the device may reveal interfaces to which one can connect extract the data directly. This would be proffered over the previously discussed methods, since the data will not have to be parsed via the communication functions which may alter it.

Another point of investigation, which may be interesting will be to perform a security audit of the watch and the service. This will prove or disprove the scenario of someone stealing an account by borrowing the watch, which will have an effect

over the validity of the retrieved data.

And finally ways to disprove falsified data by changing the time of the watch should be investigated. To do this most likely privileged access to the device will be needed. An approach for this would be to look at file fragmentation on the memory of the device and continuity and logical ordering of the log files on the device.

8 Conclusion

In conclusion, the TomTom Nike+ sport watch stores GPS and sensor data along with calculated meta data about runs that the user generates. This data can be used to get a good overview of how and where the owner of the watch was moving in the timeperiod of the recorded run.

This research reviled two ways to obtain the data which is being stored on the watch, although there might be other possible ways if phisical methods are used. The first way is to communicate to the device over the HID USB protocol and record the data which is being send from the watch. This can be done by creating a tool which requests the data from the watch and writes it to files which are them hashed and write protected or by having the Connect Nike client software get the data and sniffing the generated USB traffic and extracting the needed files from it. Unfortunately this way of extracting the data produces files in binary format, which were not decoded during this research. The second way is set up a Man in the Middle attack on the communication between the client software and the server to which it reports the extracted data and extract the data files from the generated network traffic. In this scenario the data passes through the client software, it should be noted that this software is considered trusted. This method results in readable meta data about the runs, but still all of the detailed sample data from the GPS and the sensors is in binary format and some further research need to be conducted for that to be decoded. There another way in which all of the run data can be made readable. This is possible by setting up a Man in the Middle attack on the communication between the server and the web user interface. However we do not consider that a forensically suitable way of extracting the data, because of two reasons. The first one is that the data is not being retrieved from the smart watch, but from the Nike servers instead. And there are too many point in which it can be altered until it reaches the web interface. One way to do that is by sending fake data to the server with stolen watch credentials. The second reason is that the user credentials are required in order to log on to the web interface, otherwise the data can not be obtained from it.

The integrity of the data that the watch produces can not be verified with the device access provided to a normal user. Also the indicated time of its creation can be manipulated. This makes the retrieved date from the watch not very trustworthy, but still useful in forensic cases.

References

[1] Eoghan Casey, Adrien Cheval, Jong Yeon Lee, David Oxley, and Yong Jun Song. Forensic acquisition and analysis of palm webos on mobile devices. *Digital Investigation*, 8(1):37 – 47, 2011. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2011.04.003. URL `http://www.sciencedirect.com/science/article/pii/S1742287611000405`.

[2] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3–4):175 – 184, 2012. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2011.10.003. URL `http://www.sciencedirect.com/science/article/pii/S1742287611000879`.

[3] Erin Poremski. Midterm, galaxy gear smartwatch forensics. CNIT 58100 CFM, Dr. Sam Liles, Purdue University.

[4] Burp suite, the leading toolkit for web application security testing. URL `http://portswigger.net/burp/`.

[5] The the official web site of tcpdump and libpcap. URL `http://www.tcpdump.org/`.

[6] The wireshark homepage. URL `http://www.wireshark.org/`.

[7] pypcap simplified object-oriented python extension module for libpcap. URL `http://code.google.com/p/pypcap/`.

[8] Pyusb 1.0 is a python library allowing easy usb access. URL `http://sourceforge.net/apps/trac/pyusb/`.

[9] Hex-rays home > ida. URL `https://www.hex-rays.com/products/ida/index.shtml`.

A XML run meta data

Here is an example of the XML file which is being send from the client software to the server and contains meta data about a run.

Listing 3 : rec_4_runXML.xml - meta data about run number 4

```
<?xml version="1.0" encoding="UTF-8"?>
<sportsData type="run">
  <vers>2</vers>

  <runSummary>
    <time>2014-05-02T11:30:20-08:00</time>
    <duration>2008000</duration>
    <distance unit="km">6.9369</distance> <!-- 4.3104 mi -->
    <pace>7:46 min/mi</pace>
    <calories>5267</calories>
    <gps>on</gps>
    <emped>off</emped>
  </runSummary>

  <activity>
    <type>other</type>
    <name>Other</name>
  </activity>

  <userInfo>
    <device>SportWatch</device>
    <muid>HA1094M01303</muid>
    <fwVer>C1280</fwVer>
    <empedID>HA1094M01303</empedID>
    <localSoftware version="6.3.18">Nike+ Connect</localSoftware>
    <weight>733.0</weight> <!-- 1616.0 pounds -->
    <age>21</age>
    <gender>male</gender>
  </userInfo>

  <startTime>2014-05-02T11:30:20-08:00</startTime>

  <snapShotList snapShotType="userClick">
    <snapShot event="stop">
      <duration>2007000</duration>
      <distance>6.9369</distance>
      <pace>0</pace> <!-- 0.00 min/mi -->
```

```
    </snapShot>
  </snapShotList>

  <snapShotList snapShotType="kmSplit">
    <snapShot>
      <duration>255000</duration>
      <distance>1.0035</distance>
      <pace>241545</pace> <!-- 6.47 min/mi -->
    </snapShot>
...Some data is omitted...
    <snapShot>
      <duration>1311000</duration>
      <distance>6.0141</distance>
      <pace>144092</pace> <!-- 3.85 min/mi -->
    </snapShot>
  </snapShotList>

  <snapShotList snapShotType="mileSplit">
    <snapShot>
      <duration>382000</duration>
      <distance>1.6115</distance>
      <pace>221729</pace> <!-- 5.93 min/mi -->
    </snapShot>
...Some data is omitted...
    <snapShot>
      <duration>1446000</duration>
      <distance>6.4397</distance>
      <pace>473933</pace> <!-- 12.70 min/mi -->
    </snapShot>
  </snapShotList>

  <extendedDataList>
    <extendedData dataType="distance" intervalType="time" intervalUnit="s" i
 0.0304,  0.0656,  0.1008,  0.1365,  0.1620,  0.1620,
 0.2026,  0.2281,  0.2541,  0.2894,  0.3299,  0.3806,
 0.3907,  0.4497,  0.4889,  0.5136,  0.5841,  0.6295,
 0.6801,  0.7012,  0.7834,  0.8241,  0.8707,  0.9327,
 0.9786,  1.0200,  1.0634,  1.0735,  1.0958,  1.1163,
...Some data is omitted...
 6.6972,  6.7548,  6.7964,  6.7964,  6.8020,  6.8493,
 6.8567,  6.8591,  6.9085,  6.9339,  6.9341,  6.9341,
 6.9341,  6.9341,  6.9341,  6.9341,  6.9341,  6.9341,
 6.9341,  6.9341
```

```
    </extendedData>
  </extendedDataList>
</sportsData>
```

B Commands.xml

In all five of the plugins within the Connect software an XML document can be found. It can be recognized by looking for the `<commands>` `</commands>` header and footer.

To recover this file we wrote the following snippet of code, though crude should be effective.

Listing 4 : Recovery of commands.xml

```
#!/usr/bin/python2

#Look for start to end, dump the snippet if found
sts = "<commands>"
eds = "</commands>"
import sys
print("Run as %s INPUT1 [INPUT2] [...] "%sys.argv[0])
for fn in sys.argv[1:]:
        print("Processing %s"%fn)
        of = "%s_output_commands.xml"%fn
        infile = open(fn,"rb")
        indata = infile.read()
        infile.close()
        print("File read, looking for payload")
        st = indata.find(sts)
        fs = indata.count(sts)
        print("marker at %d, found %d"%(st,fs))
        if fs == 1:
                ndm = indata.find(eds)
                ndm += len(eds)
                print("end marker at %d"%ndm)
                comdata = indata[st:ndm]
                off = open(of,"wb")
                off.write(comdata)
                off.close()
                print("Wrote %s, %d bytes"%(of, len(comdata)))
        else:
                print("failure ... '%s' %d"%(fn, fs))
```

C Log file annotation

In order to annotate a logfile generated by the SportWatch plugin we used the following python code, which was made based on the Opcodes in appendix D, and relies on the code in listing 6.

Listing 5 : Annotation of sportwatch.log

```
#!/usr/bin/python2

from opcodes import opcode_mapping

import sys
inputname = "input.log"
outputname = "output.log"
if len(sys.argv) == 3:
    inputname = sys.argv[1]
    outputname = sys.argv[2]

print("Run as %s [INPUT OUTPUT]"%sys.argv[0])
print("Reading from %s\nWriting to %s"%(inputname,outputname))

logfile = open (inputname,"rb")
data = logfile.read()
print("Input %d Bytes"%len(data))
logfile.close()

out = open (outputname, "wb")
def mkit (k, v):
    #Make the mask using "ing "
    #this keeps timestamps intact should it 'look' like an opcode
    return ("ing %s" % k.upper (), "ing %s::%s" % (k.upper (), v))

#For all opcodes, replace any found in the input
for k, v in opcode_mapping.items ():
    old, new = mkit (k, v)
    data = data.replace (old, new)

print("Output %d bytes"%len(data))
out.write(data)
out.close()
print("Output closed")
```

D Opcodes

During our analysis of the executable, the USB traffic and logs we constructed a list of opcodes, command names and for some even documentation. In order to help in analyzing data packets and logfiles we prepared some python code, containing the command to opcode mapping.

Listing 6 : Python opcodes for sportwatch plugin

```
#!/usr/bin/python2

def __init__(self):
    pass

#Opcode to text mapping
opcode_mapping = {
'E1': 'serial',                 '54': 'gpspatch-update',
'E0': 'model',                  '51': 'ephemeris-query',
'E3': 'emped-setup',            '53': 'gpspatch-query',
'E2': 'hwrevision',             '3C': 'option-clock',
'EE': 'test-reminders',         '3E': 'option-reminder-mode',
'ED': 'test-lcd',               '3D': 'option-laps',
'EA': 'test-adc',               '02': 'restore-defaults',
'EC': 'test-sflash',            '03': 'latchup',
'EB': 'test-accel',             '26': 'flags-sync',
'38': 'hrs-info',               '01': 'Unknown',
'E7': 'test-buzzer',            '21': 'time',
'4A': 'option-fuel',            '04': 'bootblock',
'11': 'eeprom-erase',           '05': 'console-write',
'10': 'readWorkouts',           '46': 'option-speed',
'13': 'battery',                '47': 'option-ui-mode',
'12': 'eeprom-query',           '08': 'version',
'15': 'desktop-write',          '09': 'upgrade',
'14': 'desktop-read',           '42': 'option-records',
'33': 'option-weigth',          '40': 'option-sounds',
'32': 'option-metric',          '41': 'writeAttaboy',
'31': 'option-24hour',          'E9': 'test-button',
'37': 'emped-info',             'E8': 'test-backlight',
'36': 'option-gender',          'E5': 'test-nordic',
'35': 'option-age',             'E4': 'test-gps',
'52': 'ephemeris-update',       '3A': 'wo-loop',
'48': 'option-lap-metric',      'E6': 'test-power',
'44':'option-clockset-mode','45': 'wo-loop-auto'
}
```

```
opcode_ints = {}
for (k,v) in opcode_mapping.items():
    val = int(k, base=16)
    opcode_ints[val] = v
```

The following is the table of opcodes and the corresponding command, and where available documentation from commands.xml.

Hex	Opcode	Command	Commands.xml entry
01	1	Unknown	
02	2	restoreDefaults	<command name='restoreDefaults' description='Restore device to factory settings and reset device when finished'/>
02	2	restore-defaults	<command name='restore-defaults' description='Restore device to factory defaults'/>
03	3	latchup	<command name='latchup' description='Turn off battery'/>
04	4	bootblock	<command name='bootblock' description='Go to boot block'></command>
05	5	console-write	<command name='console-write' description=''> <input name='text' description='text to write to device'' required='true' type='text'/> </command>
08	8	version	<command name='version' description='Get the MSP firmware version number'></command>
09	9	upgrade	<command name='upgrade' description='Upgrade application firmware'> <input name='device.firmware.image' description='firmware image data bytes' required='true' type='binary' ></input> </command>
10	16	eeprom-read	<command name='eeprom-read' description='Read the workout storage data'> <input name='workout.raw.data.offset' description='the 24-bit offset at which to read workout data, default 0' required='false' type='unsigned' max='16777215' ></input> </command>
10	16	readWorkouts	<command name='readWorkouts' description='Reads workouts from the device into attribute workout.raw'></command>
11	17	eeprom-erase	<command name='eeprom-erase' description='Erase all workout storage data'> <input name='magic' description='magic number required to call this command' required='true' type='unsigned' max='65535' ></input> </command>
12	18	eeprom-query	<command name='eeprom-query' description='Get the total available bytes of workout data storage, and the number of bytes in use'></command>
13	19	battery	<command name='battery' description='Get the battery state and charge level'></command>
14	20	desktop-read	<command name='desktop-read' description='Read raw binary image of the desktop data block'/>
15	21	desktop-write	<command name='desktop-write' description='Write raw binary image of the desktop data block'> <input name='data' description='data to be written' required='true' type='binary' ></input> </command>
21	33	time	<command name='time' description='Set or get the time'> <input name='device.clock.time' description='count of seconds since 1/1/1970. Unix time, UTC' required='false' type='unsigned'/> <input name='device.clock.gmtOffset' description='The offset from GMT time in seconds' required='false' type='signed'/> <input name='device.clock.dstOffset' description='Offset in minutes due to daylight savings time' required='false' type='signed'/> </command>
26	38	flags-sync	
31	49	option-24hour	<command name='option-24hour' description='Get or set the 24 hour clock display setting'> <input name='device.clock.is24HourMode' description='If the clock should be in 24-hour mode' required='false' type='boolean' ></input> </command>
32	50	option-metric	<command name='option-metric' description='Get or set the display mode (kilometers or miles)'> <input name='device.display.metric' description='if 24-hour mode is employed' required='false' type='boolean' ></input> </command>
33	51	option-weigth	<command name='option-weight' description='Get or set the weight of the user'> <input name='athlete.1.weight' description='Weight of the athlete in Kilograms' required='false' type='real' ></input> </command>
35	53	option-age	<command name='option-age' description='Get or set the age of the user'> <input name='athlete.1.age' description='age of the athlete in years' required='false' type='unsigned' ></input> </command>
36	54	option-gender	<command name='option-gender' description='Get or set the gender of the user'> <input name='athlete.1.gender' description='gender value' required='false' type='text' ></input> </command>
37	55	emped-info	<command name='emped-info' description='Get information on linked empeds'> <input name='sensor.type.1.linked.index' description='the index (0-7) of the linked emped' required='true' type='unsigned' max='7' ></input> </command>
38	56	hrs-info	<command name='hrs-info' description='Get information on linked heart rate straps'> <input name='sensor.type.2.linked.index' description='the index (0-7) of the linked heart rate strap' required='true' type='unsigned' max='7' ></input> </command>
3a	58	wo-loop	<command name='wo-loop' description='Get or set the workout loop'> <input name='device.workout.loop' description='comma delimited list of bottom stat and top workout stats' required='false' type='text' > <enums> <enum>Distance</enum> <enum>Pace</enum> <enum>Elapsed_Time</enum> <enum>Heart_Rate</enum> <enum>Calories</enum> <enum>Time</enum> </enums> </input> </command>
3C	60	option-clock	<command name='option-clock' description=''> <input name='style' description='clock face style' required='false' type='unsigned' ></input> </command>

3D	61	option-laps	<command name='option-laps' description=''> <input name='device.laps.mode' description='laps mode' required='false' type='unsigned' ></input> <input name='device.laps.auto.units' description='auto laps units' required='false' type='unsigned' ></input> <input name='device.laps.auto.value' description='auto laps value' required='false' type='unsigned' ></input> <input name='device.laps.run.units' description='run interval units' required='false' type='unsigned' ></input> <input name='device.laps.run.value' description='run interval value' required='false' type='unsigned' ></input> <input name='device.laps.rest.units' description='rest interval units' required='false' type='unsigned' ></input> <input name='device.laps.rest.value' description='rest interval value' required='false' type='unsigned' ></input> </command>
3E	62	option-reminder-mode	<command name='option-reminder-mode' description='Set reminder mode and time of day for reminders'> <input name='enabled' description='enabled' required='false' type='boolean'/> <input name='time' description='time of day for reminders' required='false' type='unsigned'/> </command>
40	64	option-sounds	<command name='option-sounds' description='Get or set the sounds mode'> <input name='device.sounds.mode' description='sounds mode' required='false' type='unsigned' ></input> </command>
41	65	writeAttaboy	
42	66	option-records	<command name='option-records' description='set or get user records'> <input name='totalDistance' description='total distance in meters' required='false' type='unsigned'/> <input name='1mile' description='fastest mile in milliseconds' required='false' type='unsigned'/> <input name='1km' description='fastest kilometer in milliseconds' required='false' type='unsigned'/> <input name='5km' description='fastest 5km' required='false' type='unsigned'/> <input name='10km' description='fastest 10km' required='false' type='unsigned'/> <input name='marathon' description='fastest marathon' required='false' type='unsigned'/> <input name='longest' description='longest single run in centimeters' required='false' type='unsigned'/> </command>
44	68	option-clockset-mode	<command name='option-clockset-mode' description='get or set method of setting clock'> <input name='mode' description='clock set mode, 0=PC and GPS, 1 = PC only, 2 = manual' required='false' type='unsigned'/> </command>
45	69	wo-loop-auto	
46	70	option-speed	
47	71	option-ui-mode	
48	72	option-lap-metric	
4A	74	option-fuel	
51	81	ephemeris-query	<command name='ephemeris-query' description='Get information about ephemeris data'/>
52	82	ephemeris-update	<command name='ephemeris-update' description='Update ephemeris data'></command> <command name='updateEphemeris' description='Upgrade ephemeris'> <input name='image' description='Ephemeris image data bytes' required='true' type='binary'/> <input name='expiration' description='Expiration date of ephemeris data' required='true' type='unsigned'/> </command>
53	83	gpspatch-query	<command name='gpspatch-query' description='Get information about currently installed GPS patch'/>
54	84	gpspatch-update	<command name='gpspatch-update' description='Update GPS patch'> <input name='patch' description='GPS patch' required='true' type='binary'/> </command>
E0	224	model	<command name='model' description='Get or set model number'> <input name='model' description='model number' required='false' type='text' ></input> </command>
E1	225	serial	<command name='serial' description='Get or set the device serial number'> <input name='serialnumber' description='serial number for set command' requried='false' type='text'/> </command>
E2	226	hwrevision	<command name='hwrevision' description='Get the device hardware revision'> <input name='hardwareRevision' description='hardware revision number' required='false' type='unsigned'/> </command>
E3	227	emped-setup	<command name='emped-setup' description='Commands for factory setup of empeds'> <input name='action' description='0=stop pairing, 1=start pairing, 2=get paired emped, 3=set factory emped, 4=get factory emped, 5=remove all empeds' required='true' type='unsigned' min='0' max='5'/> <input name='empedID' description='emped radio ID' requried='false' type='unsigned'/> <input name='payloadType' description='payload type' requried='false' type='unsigned'/> </command>
E4	228	test-gps	<command name='test-gps' description='Test GPS receiver'> <input name='test' description='test to execute (0-4)' required='true' type='unsigned' min='0' max='4'/> <input name='svid' description='satalite ID' required='false' type='unsigned'/> <input name='period' description='test duration in seconds' required='false' type='unsigned'/> </command>
E5	229	test-nordic	<command name='test-nordic' description='Test Nordic radio'> <input name='test' description='test to execute (0-1)' required='true' type='unsigned' min='0' max='1'/> </command>
E6	230	test-power	<command name='test-power' description='Test power consumption'> <input name='test' description='test to execute (0-2)' required='true' type='unsigned' min='0' max='2'/> </command>

E7	231	test-buzzer	<command name='test-buzzer' description='Test buzzer'> <input name='test' description='test to execute (1 to play tone, 0 to stop)' required='true' type='unsigned' min='0' max='1'/> <input name='frequency' description='frequency in Hertz' required='false' type='unsigned'/> <input name='duration' description='duration in milliseconds' required='false' type='unsigned'/> </command>
E8	232	test-backlight	<command name='test-backlight' description='Test the backlight'> <input name='test' description='1 to turn on backlight, 2 to turn off, 0 to exit test' required='true' type='unsigned' min='0' max='1'/> </command>
E9	233	test-button	<command name='test-button' description='Test the buttons'> <input name='test' description='1 to read button state, 0 to exit test' required='true' type='unsigned' min='0' max='1'/> </command>
EA	234	test-adc	<command name='test-adc' description='Test ADC'> <input name='test' description='1=sample VBAT_DIVIDE, 2=sample THERM_MEAS, 3=sample thermal sensor, 0=exit' required='true' type='unsigned' min='0' max='3'/> </command>
EB	235	test-accel	<command name='test-accel' description='Test accelerometer'> <input name='test' description='1=sample accelerometer, 0=exit test' required='true' type='unsigned' min='0' max='1'/> </command>
EC	236	test-sflash	<command name='test-sflash' description='Test serial flash'> <input name='test' description='1=query device Ids, 0=exit test' required='true' type='unsigned' min='0' max='1'/> </command>
ED	237	test-lcd	<command name='test-lcd' description='Test LCD'> <input name='test' description='1=all black, 2=all white, 0=exit test' required='true' type='unsigned' min='0' max='2'/> </command>
EE	238	test-reminders	<command name='test-reminders' description='Set time of last run reminder'> <input name='time' description='time of last run reminder' required='false' type='unsigned'/> </command>