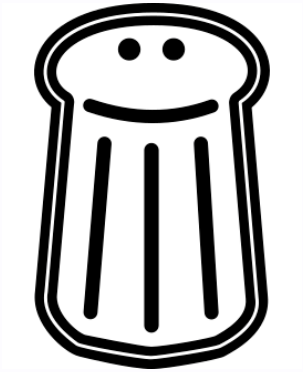
BLE Hacking 101 with WHAD

Romain Cayre, Damien Cauquil



Pass The Salt - July 2, 2025

INSA TOULOUSE LAAS CNRS Quarkslab

Who are we ?

Romain Cayre, EURECOM

- maintainer of *Mirage*, a popular BLE swiss-army tool
- loves *cross-protocol attacks* (*Wazabee*)

Damien Cauquil, Quarkslab

- maintainer of *Btlejack*, another BLE swiss-army tool
- loves reversing stuff, including *embedded systems*

INSA TOULOUSE LAAS CNRS Quarkslab

Agenda

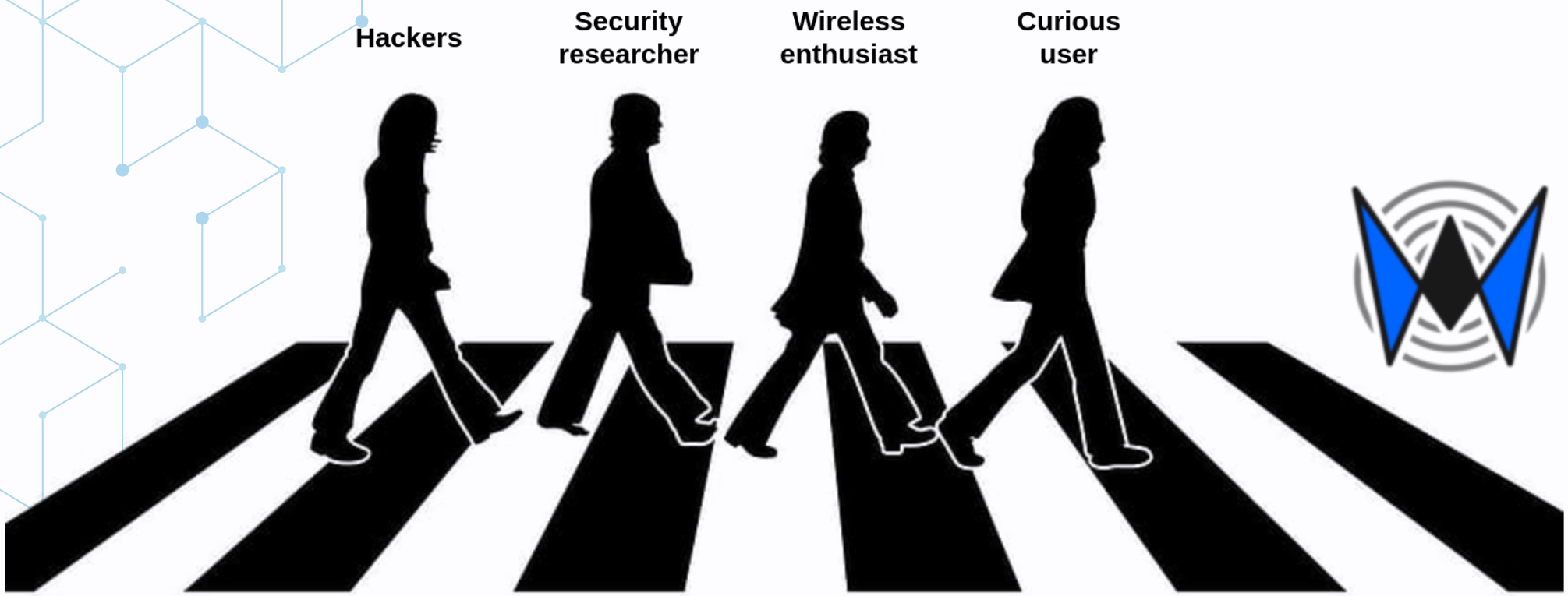
- What is WHAD ?
- Discovering BLE devices
- Interacting with a BLE device
- Creating fake BLE devices
- Breaking BLE legacy pairing
- Python scripting

Workshop goals

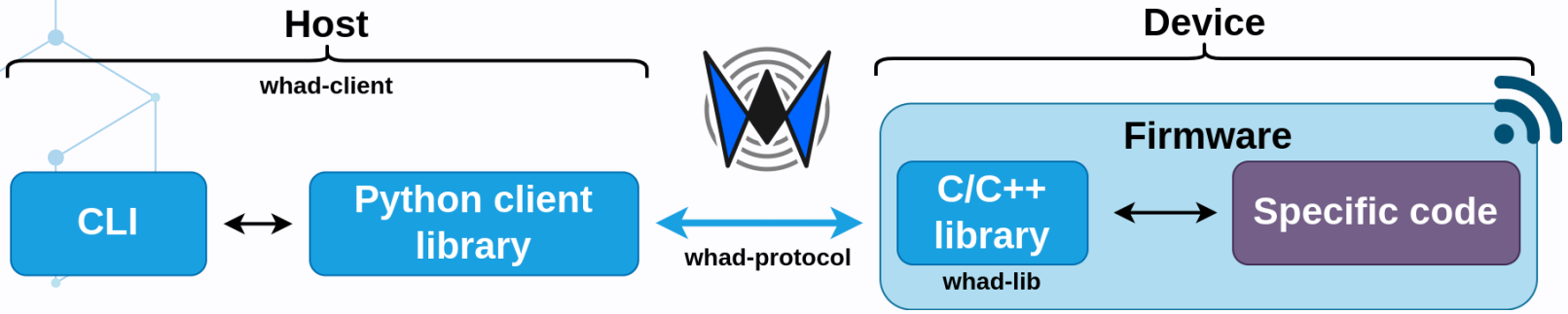
- Introduce WHAD, present its features and why it's cool !
- Demonstrate the basic BLE tools to:
  - handle WHAD interfaces
  - discover BLE devices
  - interact with BLE devices
  - emulate fake BLE devices
- Teach simple Python scripting with WHAD
- Let you experiment with the framework and tools

What is WHAD ?

A tool for everyone

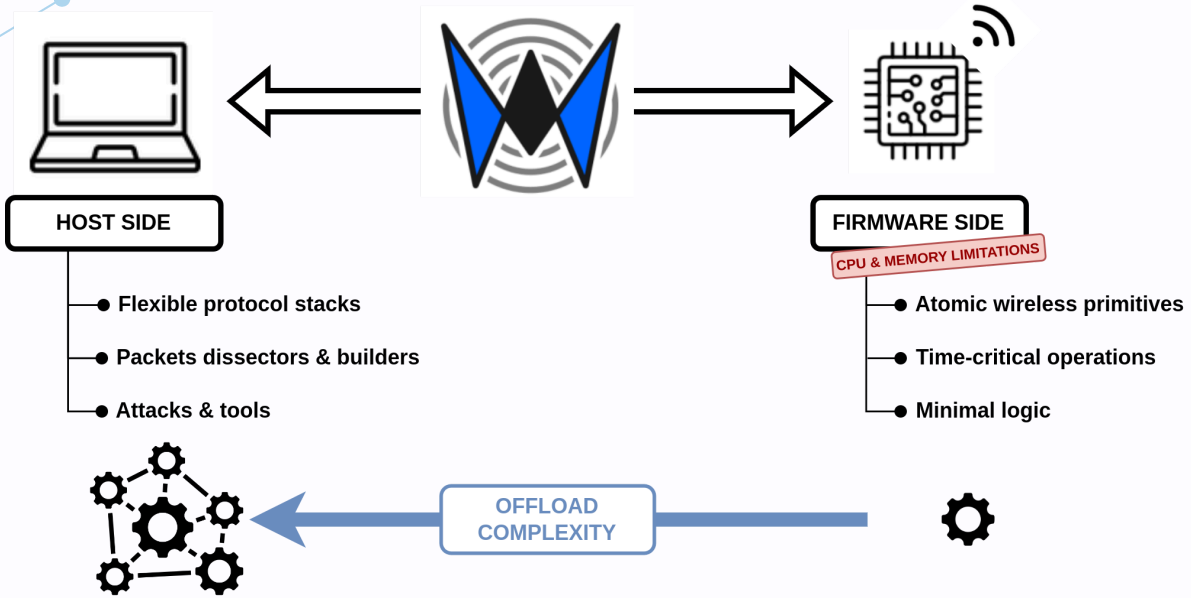


Global overview



- Harmonised Host / RF Hardware **communication protocol**
- **Python library** supporting multiple wireless protocols
- Multiple user-friendly **CLI tools** that can be combined
- Set of firmwares for **various hardware devices**

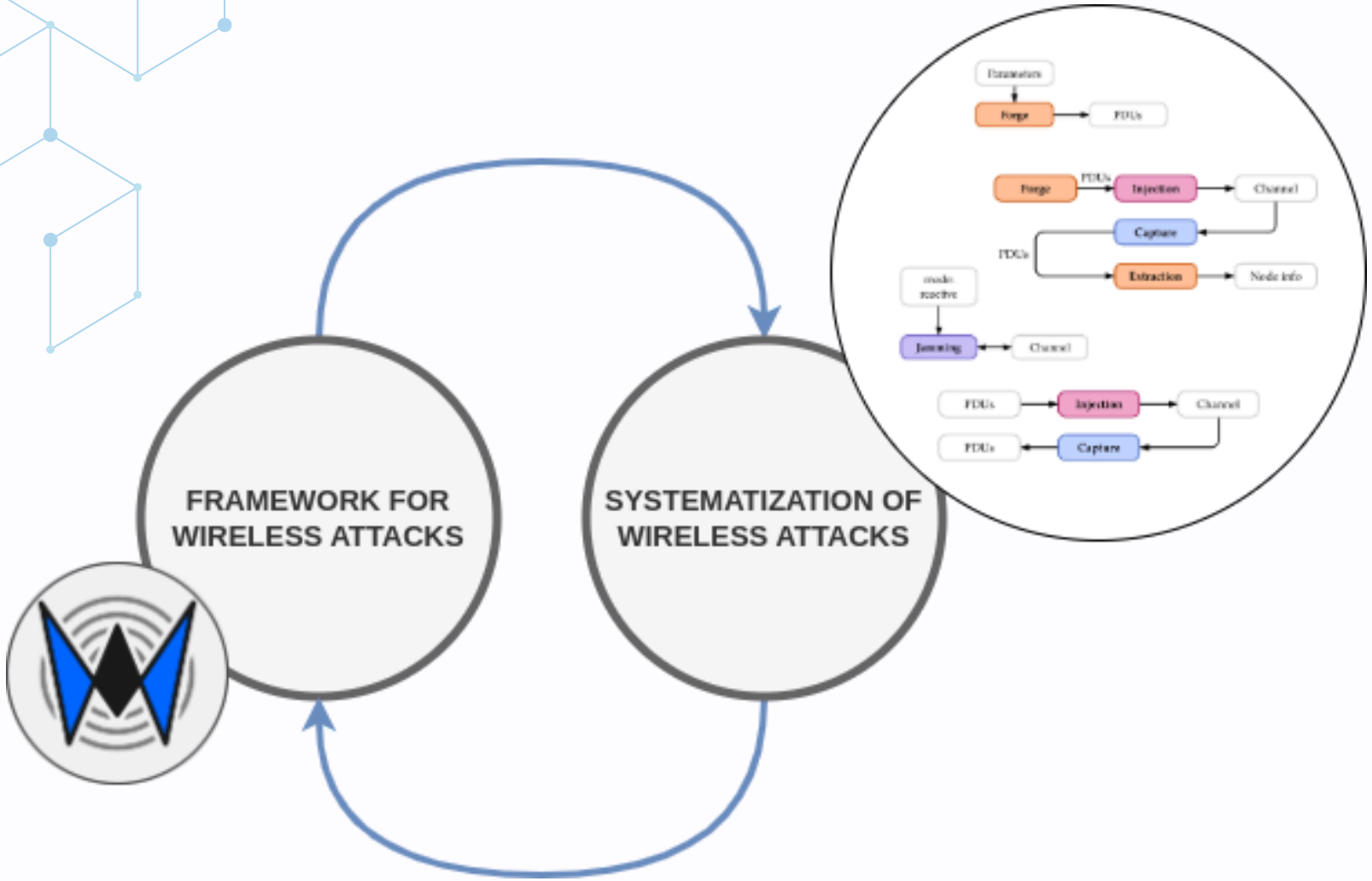
Core concepts



Offload complexity as much as possible:

- Protocol stacks implemented on host side
- **Hardware does hardware stuff:** timing-critical tasks, RF

Core concepts



Core concepts

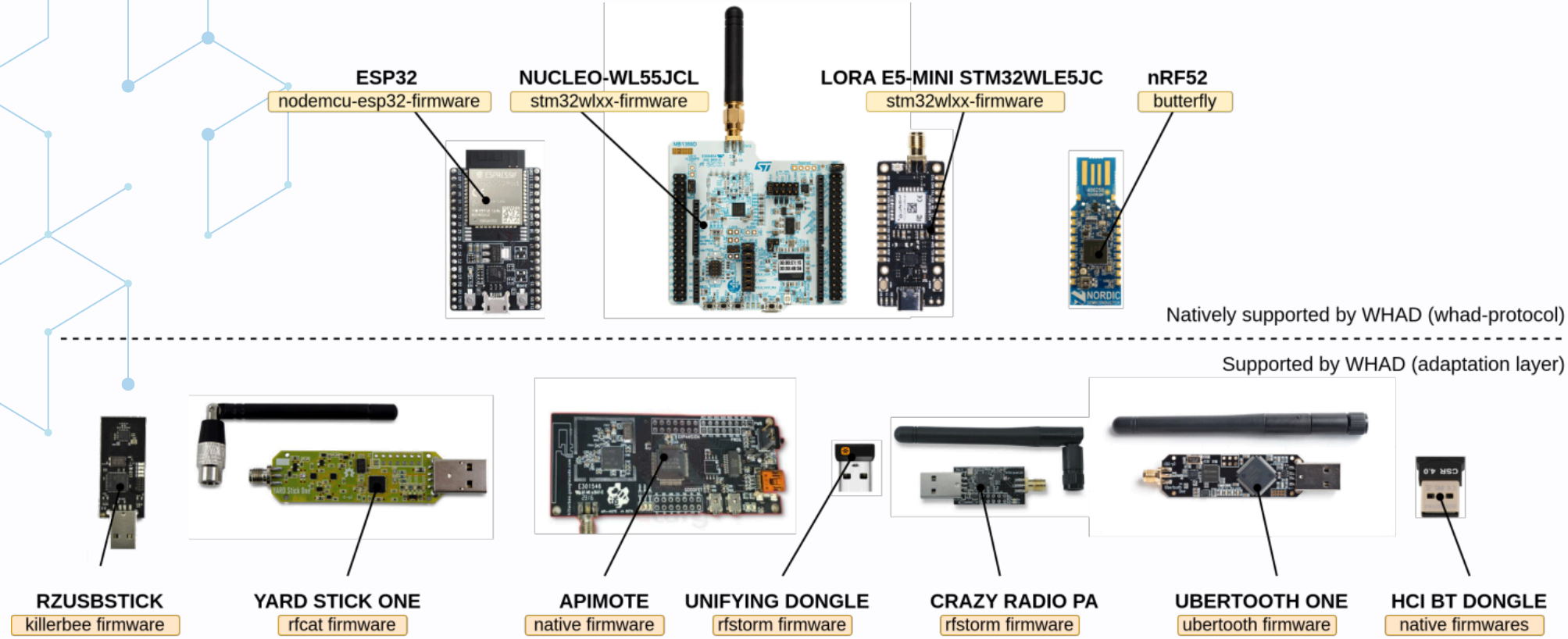
Generic tools to perform generic tasks/attacks:

- Tools work on multiple protocols
- Common and standardized file format (PCAP)
- Generic tools that can be chained to create complex tools (inspired by UNIX philosophy)

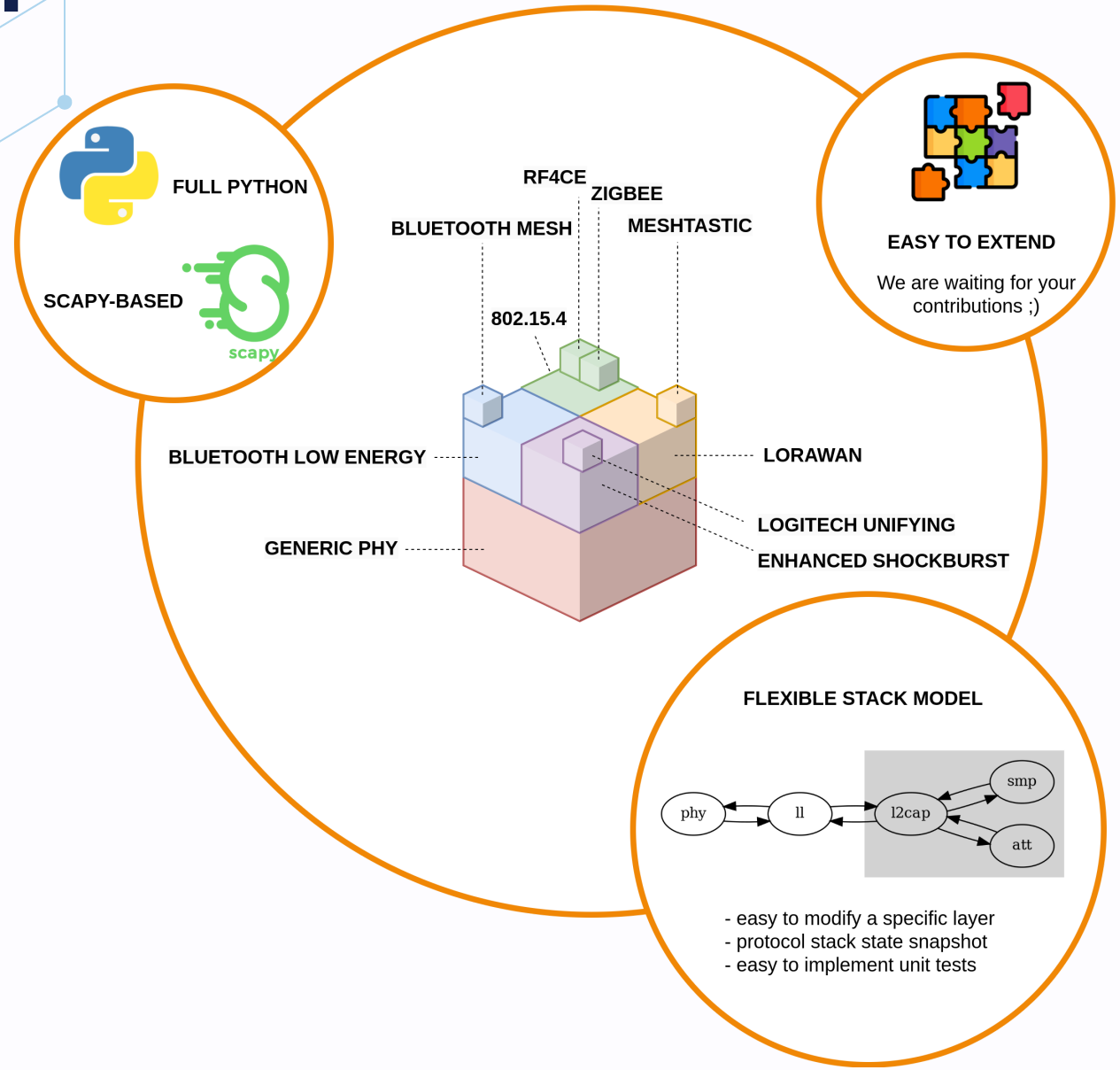
Compatible hardware

- **Hardware is discoverable** and exposes its *capabilities*
- Generic and custom tools can **tune attacks or tasks** to hardware
- Anyone can develop a compatible firmware **without bothering about tools**

Compatible hardware

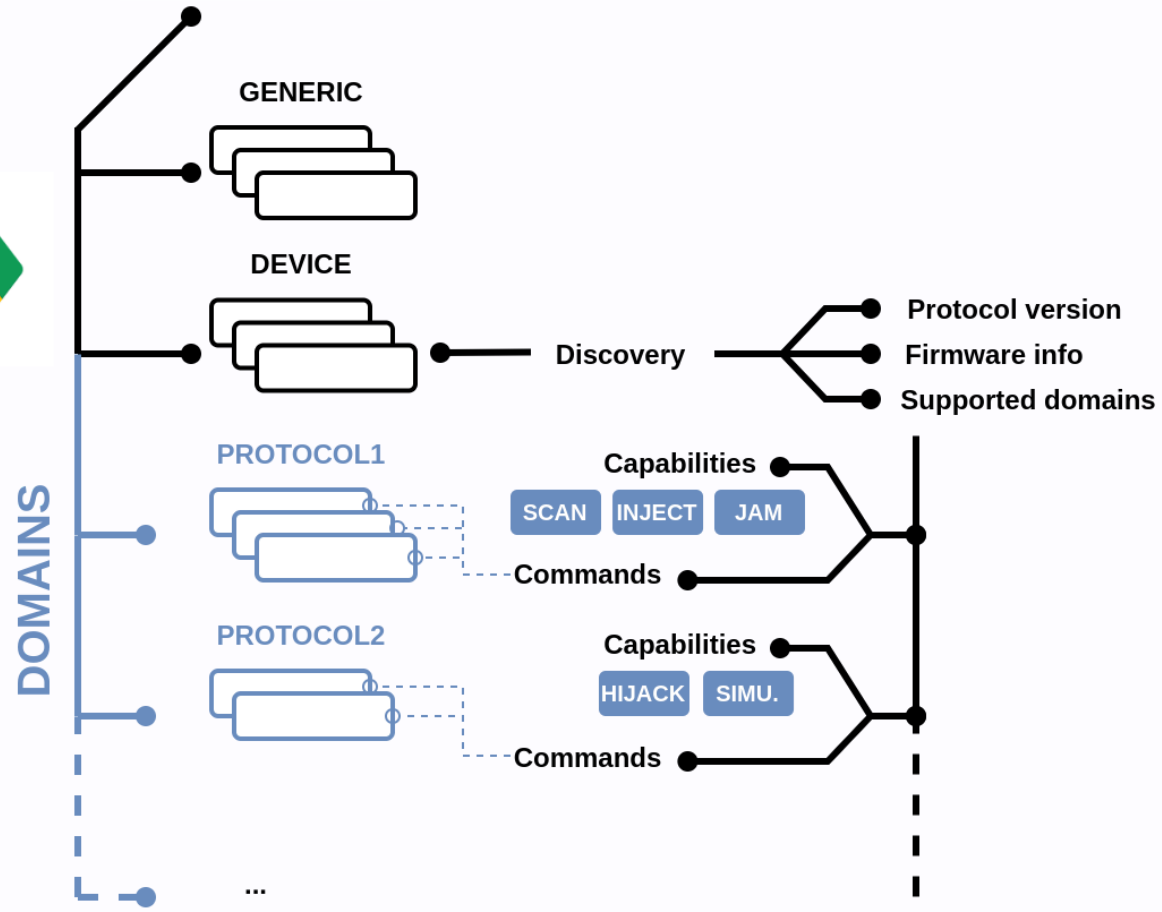


Supported protocols

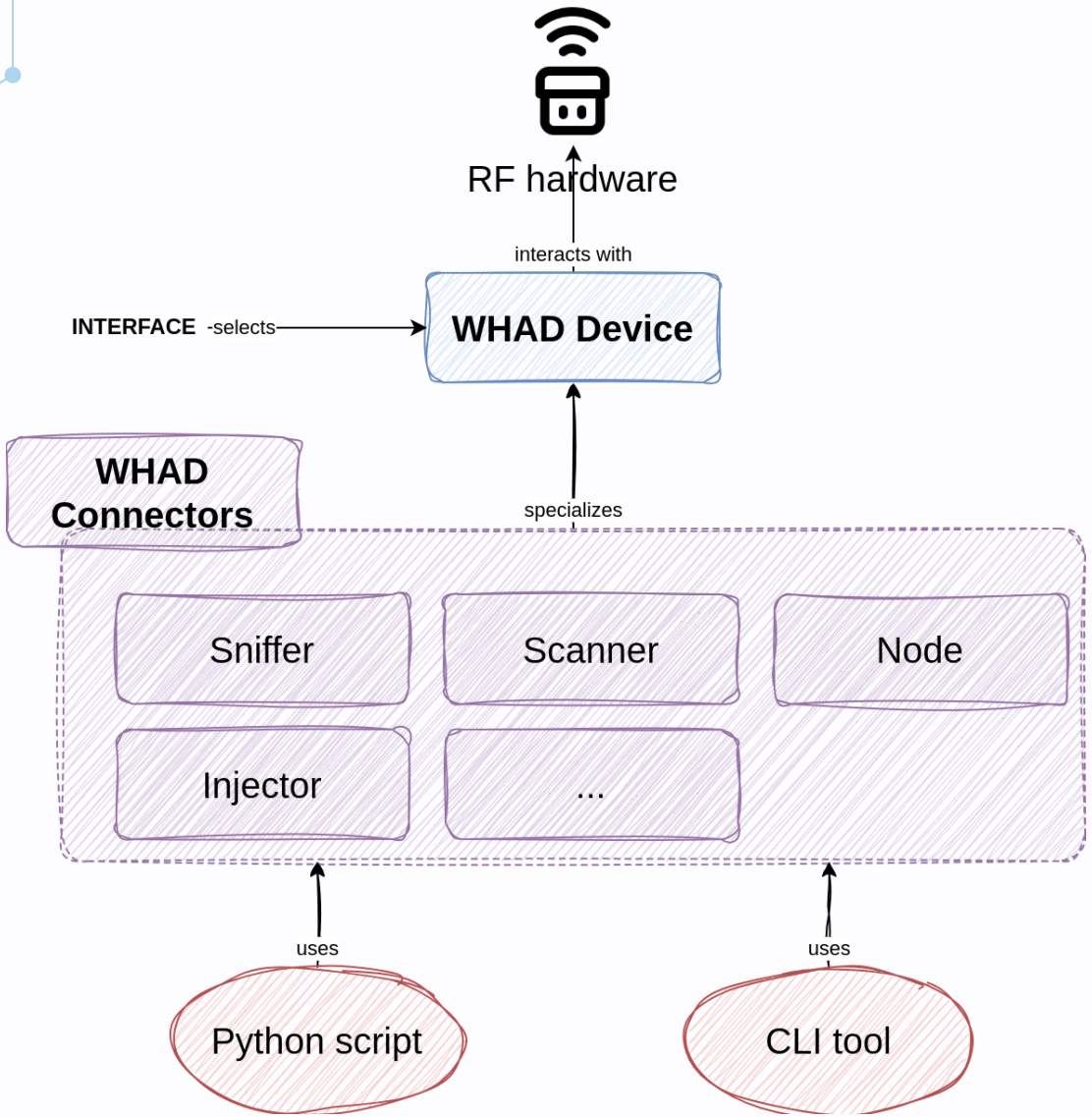


WHAD protocol

protobuf
Protocol Buffers

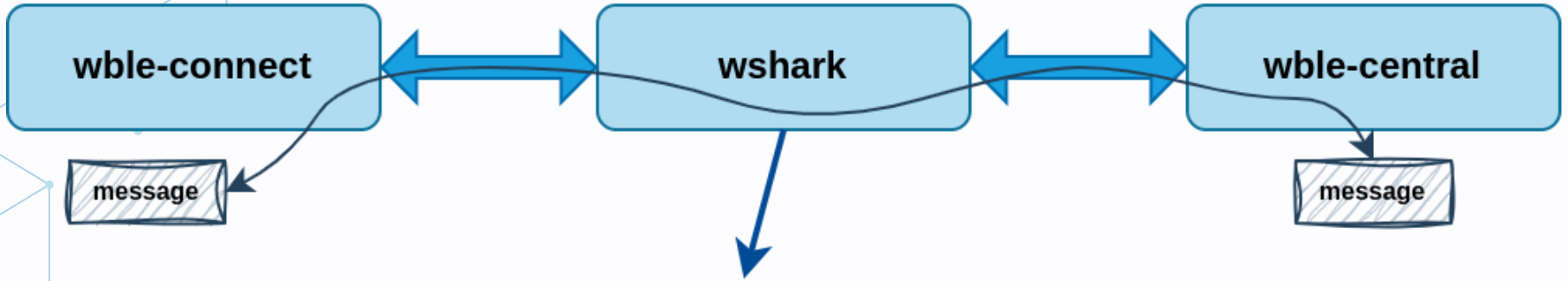


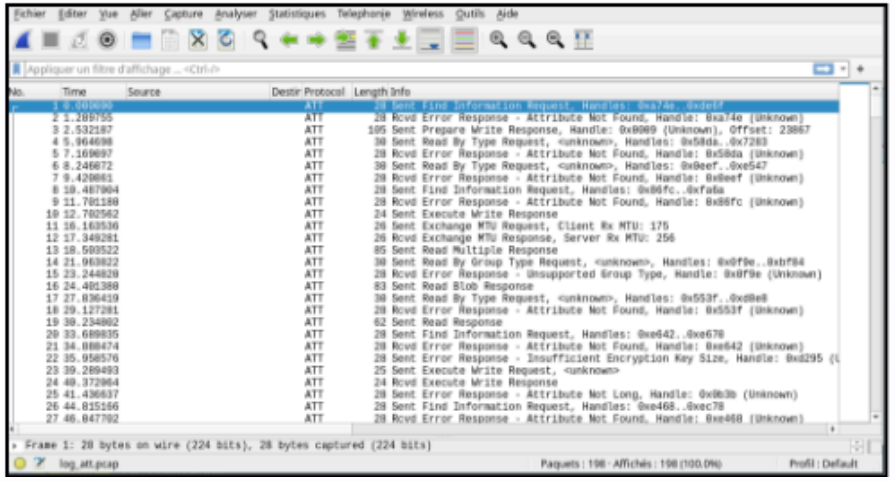
Connectors and interfaces



Tool chaining

```
$ wble-connect -i hci0 00:11:22:33:44:55 | wshark | wble-central profile
```





Python scripting

WHAD provides an user-friendly Python API to implement your own custom scripts:

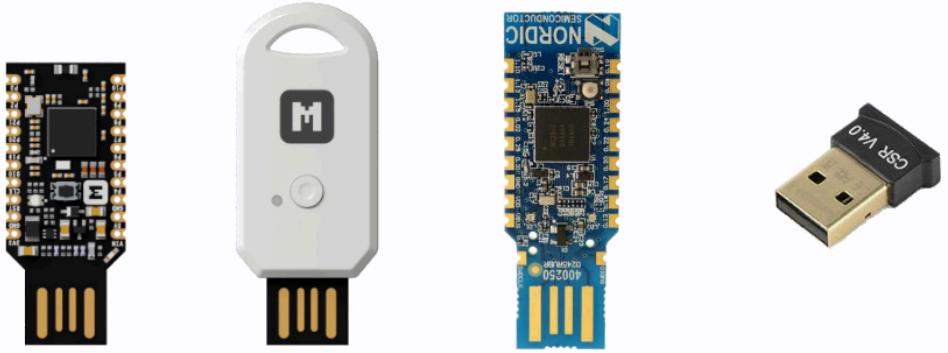
```
import sys
from whad.ble import Central
from whad.ble.profile import UUID
from whad.device import WhadDevice

# Create the Central connector & the WHAD device
central = Central(WhadDevice.create("hci0"))
device = central.connect('74:da:ea:91:47:e3', random=False)
if device is not None:
    device.discover()
    device_name = device.get_characteristic(UUID(0x1800), UUID(0x2A00)
    print("[i] Device name: ", device_name.value)
    central.close()
else:
    print("Usage: ", sys.argv[0]+" <interface>")
```

Setup

Hardware requirements

- Computer or VM running a **Linux** operating system
- nRF52 USB Dongle with **ButteRFly** installed
- USB Bluetooth Low Energy **dongle** or **embedded Bluetooth adapter**



Resources

- Online repository with examples and code templates: https://github.com/whad-team/whad-workshop
- Virtual machine image: https://drive.google.com/file/d/10wKAWIcy5zyRrwxYV8BikEAq5SCYjIOP/view?usp=sharing

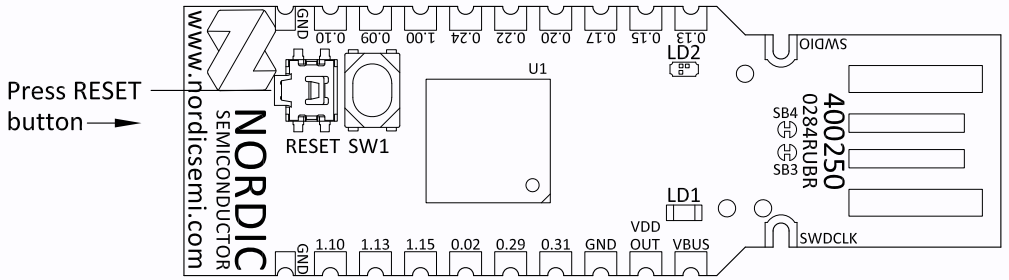
Instaling WHAD locally

Installing whad-client is as simple as running:

```
$ mkdir whad-workshop && cd whad-workshop
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install whad
(venv) $ winstall --rules all
```

Flashing ButteRFly firmware

Set the dongle in **programming mode** by pressing the side button *RESET*:



Run the following command:

```
$ winstall --flash butterfly
```

WHAD interfaces

`wup` / `whadup` is the easiest way to detect compatible interfaces on your system:

- automatically detect compatible interfaces (USB and internal)
- query any interface to determine its capabilities
- show capabilities and explain what the device is capable of

WHAD interfaces

List available interfaces:

```
$ wup
```

Enumerate the capabilities of a specific interface *"uart0"*:

```
$ wup uart0
```

Hands-on

WHAD interfaces

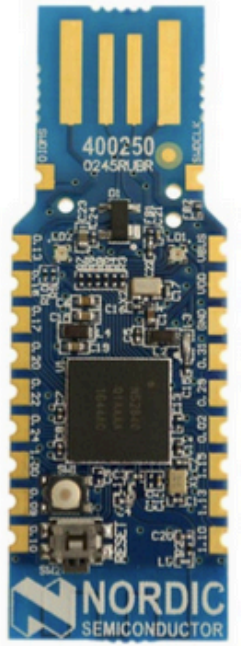
```
(venv) virtualabs@virtubox:~$ wup uart0
[i] Connecting to device ...
[i] Device details

Device ID: c3:9e:c3:80:c2:80:c3:98:c3:90:08:c3:86:6c:7d:c2:a3:c3:9f:c3:bb:c3:b4:c3:9a:10:c2:9b
Firmware info:
- Author : Romain Cayre
- URL : https://github.com/whad-team/butterfly
- Version : 1.0.1

[i] Discovering domains ...
[i] Domains discovered.

This device supports Bluetooth LE:
- can sniff data
- can inject packets
- can hijack communication
- can simulate a role in a communication

List of supported commands:
 - SetBdAddress: can set BD address
```



whadup / wup

Use `wup` to list all the available interfaces:

```
$ wup
```

Use `wup` to show more information about an interface:

```
$ wup hci0
```

Hands-on

Discovering BLE devices

BLE device advertisements

- Two types of PDU sent by BLE devices:
  - **Advertisement PDUs**: *ADV_IND*, *ADV_DIRECT_IND*, *ADV_NONCONN_IND*, etc...
  - **Scan-related PDUs** (active mode): *SCAN_REQ* and *SCAN_RSP*
- Advertisement and Scan PDUs are sent (at least) on **advertising channels** (**37**, **38**, **39**)

Sniffing advertising PDUs

`wsniff` provides a dedicated mode when domain is set to **ble**:

```
$ wsniff -i uart0 ble -a
```

The `-a` option enables *wsniff*'s BLE advertisement sniffing mode.

It listens on channel 37 by default, but you can specify another with the `-c / --channel` option.

Sniffing is only supported by nRF52840 dongles.

Scanning for BLE devices

`wble-central` provides a scanning feature to discover surrounding devices:

```
$ wble-central -i hci0 scan
 RSSI Lvl  Type  BD Address          Extra info
[-058 dBm] [PUB] 2c:be:eb:XX:XX:XX
[-076 dBm] [RND] f2:ea:48:d1:48:c9 name:"ZeFit4 HR#17757"
[-058 dBm] [PUB] a4:c1:38:XX:XX:XX name:"8eyvxxxx"
[-064 dBm] [PUB] d0:d0:03:XX:XX:XX
```

Unlike sniffing, scanning loops on every advertising channel

Profiling BLE devices

- *GATT enumeration procedure* discovers services and characteristics for a specific device but:
  - It may be slow depending on the target device
  - Performed **multiple times** when required by different tools
- WHAD can **save a BLE device's GATT profile into a JSON file** for later use

Profiling BLE devices

`wble-central` provides a specific `profile` command to connect to a specific device and save its *profile* to a JSON file:

```
$ wble-central -i hci0 -b f2:ea:48:d1:48:c9 -r profile my_device.json
```

`-r` option is mandatory when dealing with a device that uses a *random* BD address

Hands-on

BLE profile example

```
{
    "services": [
        {"uuid": "1800", "type_uuid": "2800", "start_handle": 1,
            "end_handle": 7, "characteristics": [...]},
        {"uuid": "1801", "type_uuid": "2800", "start_handle": 8,
            "end_handle": 8, "characteristics": []},
        ...
    ],
    "devinfo": {
        "adv_data": "10095...",
        "bd_addr": "f2:ea:48:d1:48:c9",
        "addr_type": 1,
        "scan_rsp": ""
    }
}
```

Interacting with BLE devices

wble-central interactive mode

Provides a set of commands to:

- scan for BLE devices
- connect to a BLE device
- enumerate its GATT profile
- interact with its exposed characteristics

To use it, simply run:

```
$ wble-central -i hci0
```

Scanning and connecting to a device

Scan and connect to a device:

```
wble-central> scan
 RSSI Lvl  Type  BD Address         Extra info
[-078 dBm] [RND] f2:ea:48:d1:48:c9 name:"ZeFit4 HR#17757"
```

```
wble-central> connect f2:ea:48:d1:48:c9
```

Successful connection:

```
Successfully connected to target f2:ea:48:d1:48:c9
wble-central|f2:ea:48:d1:48:c9>
```

GATT enumeration

Once connected, use the `profile` command:

```
wble-central|f2:ea:48:d1:48:c9> profile
```

```
Service Generic Access (0x1800)

 Device Name (0x2A00) handle: 2, value handle: 3
  | access rights: read, write

...
```

GATT enumeration is **required to use UUIDs** in future operations.

Reading a characteristic's value

Use the `read` command with the characteristic's UUID:

```
wble-central|f2:ea:48:d1:48:c9> read 2a00
```

```
00000000: 5A 65 46 69 74 34 20 48  52 23 31 37 37 35 37     ZeFit4 HR#
```

Use the `read` command with the characteristic's value handle (3):

```
wble-central|f2:ea:48:d1:48:c9>read 3
```

Writing to a characteristic's value

Use the `write` command with the characteristic's UUID:

```
wble-central|f2:ea:48:d1:48:c9> write 2A00 "Hello"
```

Use the `write` command with the characteristic's value handle:

```
wble-central|f2:ea:48:d1:48:c9> write 3 "Hello"
```

Write data in hex ("ABCD"):

```
wble-central|f2:ea:48:d1:48:c9> write 2A00 hex 41 42 43 44
```

Hands-on

Writing to a characteristic's value

Try to write into a non-writeable characteristic's value:

```
wble-central|f2:ea:48:d1:48:c9> >write 19 "Hello"
[!] ATT error: write operation not allowed
```

Write without waiting a response with `write-cmd` :

```
wble-central|f2:ea:48:d1:48:c9> writecmd 3 "Hacked"
```

Subscribing to notifications

Use the `sub` command with the characteristic's UUID:

```
wble-central|f2:ea:48:d1:48:c9> sub 2a37
```

or with the characteristic's handle:

```
wble-central|f2:ea:48:d1:48:c9> sub 22
```

Real-time monitoring

Use the `wireshark` command to start a live wireshark monitor:

```
wble-central|f2:ea:48:d1:48:c9> wireshark on
```

Run a `profile` command and let the magic happens

Hands-on

Scripting with WHAD, a primer

Scripting with WHAD

- WHAD's interactive shell supports scripting **by default**
- Scripting is useful when:
  - Some devices expect to receive data **in a short time window**
  - you want to **automate** one or more tasks/attacks
- Scripts are **simple text files** with *.whad* extension
  - commands are executed one after the other
  - basic scripting language

Manipulating the environment

- WHAD interactive shells use a dedicated *environment* to create, store and recall **variables**
- To create a variable: `set NAME VALUE`

```
set TARGET "f2:ea:48:d1:48:c9"
```

- To use a variable: `$VAR_NAME`

```
connect $TARGET
```

Manipulating the environment

To delete a variable: `unset VAR_NAME`

```
unset TARGET
```

To list current environment: `env`

```
wble-central> env
TOTO=tralala
TARGET=f2:ea:48:d1:48:c9
```

Some useful scripting commands

To print some text or value: `echo TEXT`

```
echo "Connecting to " $TARGET
```

Wait for the user to press a key: `wait MESSAGE`

```
wait "Press a key to disconnect from target"
```

Scripting with *wble-central*

Quick whad script to connect to a target (*example.whad*):

```
set TARGET "f2:ea:48:d1:48:c9"
echo "Connecting to " $TARGET "..."
connect $TARGET random
```

Run script with `wble-central`, using the `-f` option:

```
$ wble-central -i hci0 -f ./example.whad
```

Automating wble-central

Write a script that connects to your watch and:

- discovers its services and characteristics
- read the *DeviceName* characteristic from its *Generic Access* service and prints it
- writes "0wn3d" into the same *DeviceName* characteristic value
- disconnects properly from the device

Speeding things up !

1. Export your watch GATT profile into a JSON file using `wble-central`
2. Modify the previous script to load this JSON file using the interactive shell's `profile` command
3. Verify that this new script runs much faster

Creating fake BLE devices

wble-periph interactive mode

- Provides a **set of commands** to:
  - configure the device's advertising data
  - create GATT services and characteristics
  - modify characteristics's properties and values
  - start and stop advertising
- **Displays every client GATT operation** in real-time
- Allows user to **modify characteristics's values** even when a GATT client is connected !

Creating a fake peripheral by hand

Start `wble-periph` in interactive mode:

```
$ wble-periph -i hci0
```

Use the `name` command to set the device name:

```
wble-periph> name "EmulatedDevice"
```

Add a *Generic Access* service using the `service` command:

```
wble-periph> service add 1800
```

Creating an emulated peripheral by hand

Create a *Device Name* characteristic using the `char` command:

```
wble-periph|service(1800)> char add 2a00 read write notify
```

This characteristic is declared as readable, writeable and supports notifications.

Set the characteristic value with `write` :

```
wble-periph|service(1800)> write 2a00 "EmulatedDevice"
```

Check the device's GATT profile

Use the `back` command to return to main menu:

```
wble-periph|service(1800)> back
```

Use the `service` command to print the current GATT profile:

```
wble-periph> service

Service 1800 (Generic Access) (handles from 1 to 4):
└─ Characteristic 2a00 (Device Name)
   └─ handle:2, value handle: 3, props: read,write,notify
   └─ Descriptor 2902 (handle: 4)
```

Start advertising

Use the `start` command to tell `wble-periph` to start advertising:

```
wble-periph> start
```

When the emulated device is advertising, no more changes can be made to the GATT profile configured previously.

Connect to your emulated device

Using *nRF Connect*, connect to your emulated device and read its *Device Name* characteristic

`wble-periph` should display something like:

```
wble-periph> start
New connection handle:24
Reading characteristic 2a00 of service 1800
 00000000: 46 61 6B 65 44 65 76 69  63 65                          FakeDevic
```

Notifications

Subscribe to notifications for the *Device Name* characteristic

Use the `write` command to update the characteristic's value:

```
wble-periph[running]> write 2a00 "EmulatedDevice!"
```

Notice the value has changed in *nRF Connect*

Hands-on

Monitoring with Wireshark

Use the `wireshark` command to spawn Wireshark and monitor live GATT operations:

```
wble-periph[running]> wireshark on
```

Device cloning

Use a saved GATT profile (JSON) with `wble-periph` :

```
$ wble-periph -i hci0 -p profile.json
```

It will automatically copy the emulated device profile including:

- its services, characteristics, descriptors and values
- its advertising data and scan response (if used)

Hands-on

Scripting *wble-periph*

Automate peripheral creation

```
name "MyEmulatedDevice"
echo "Configuring services and characteristics ..."
service 18a00
char add 2a00 read write notify
write 2a00 "MyEmulatedDevice"
back
echo "Starting emulated device ..."
start
wait "Press any key to exit."</code></pre>
```

Hands-on

To run this script:

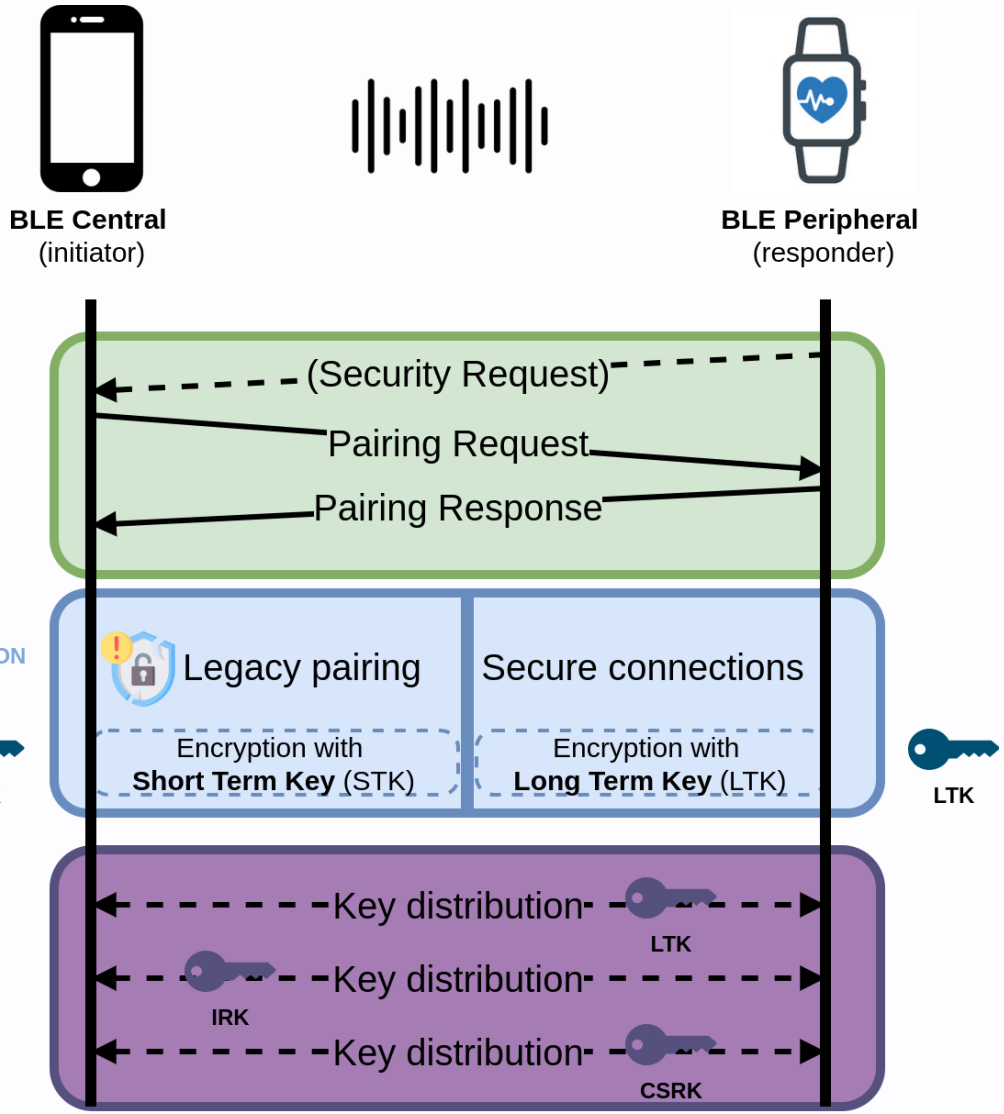
```
$ wble-periph -i hci0 -f emulated.whad
```

Breaking BLE legacy pairing

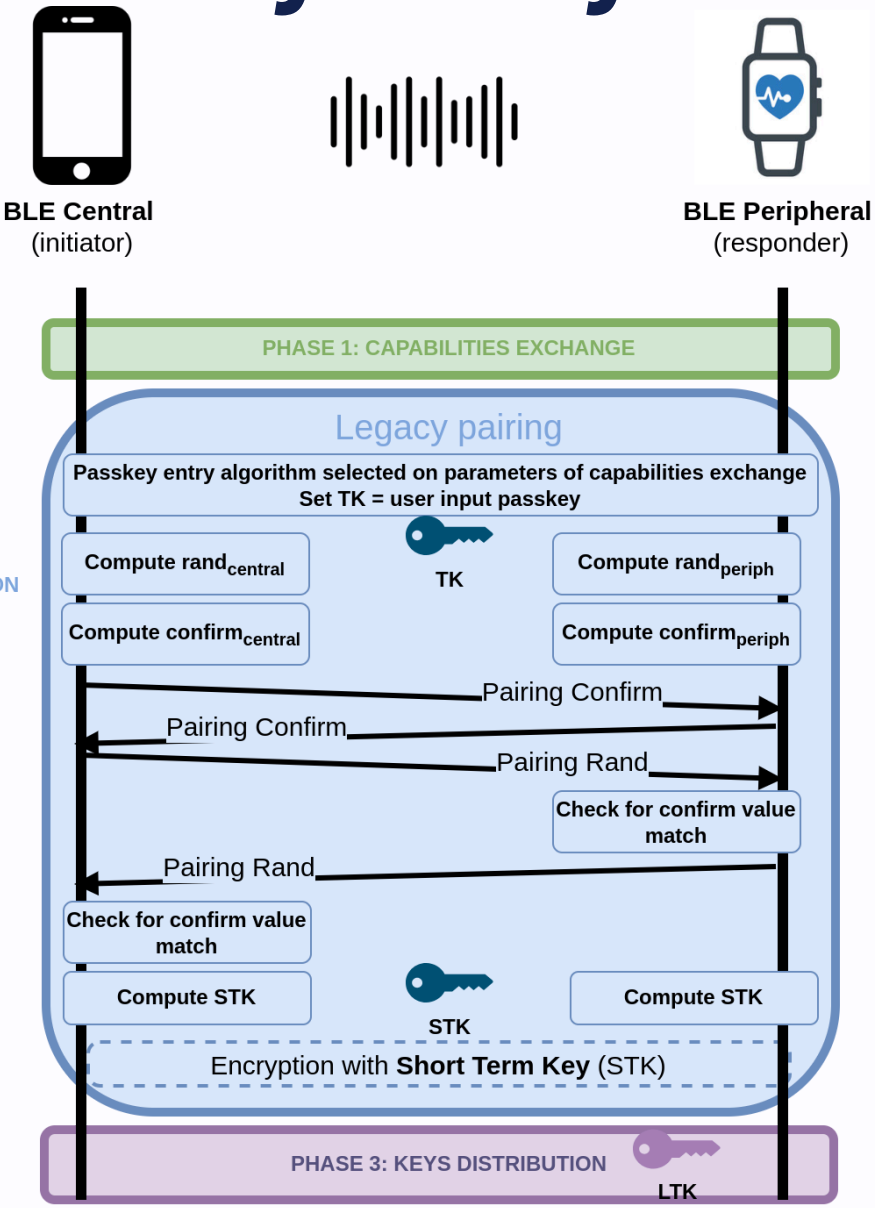
BLE pairing vs bonding

- Bluetooth Low Energy **pairing** allows to negotiate security keys (e.g., encryption) to encrypt and authenticate the link
- Bluetooth Low Energy **bonding** is a variant of BLE pairing where the devices will store the distributed keys for later use

BLE pairing overview



Legacy pairing - passkey entry



CrackLE

- Legacy Pairing is known to be vulnerable to a key recovery attack: **crackLE**, that allows an attacker to guess or very quickly brute force the TK (Temporary Key).
- With the TK and other data collected from the pairing process, the STK (Short Term Key) and later the LTK (Long Term Key) can be collected.
- With the STK and LTK, all communications between the Central and the Peripheral can be decrypted.

CrackLE attack with WHAD

Sniffing the pairing process and the encrypted traffic:

```
$ wsniff -i uart0 ble -f | wdump pairing.pcap
```

Recovering the Short Term Key (STK):

```
$ wplay pairing.pcap | wanalyze legacy_pairing_cracking
[✓] legacy_pairing_cracking → completed
  - tk:  00000000000000000000000000000000
  - stk:  11223344112233441122334411223344
```

Hands-on

CrackLE attack with WHAD

Recovering the distributed keys (LTK, IRK, CSRK):

```
$ wplay --flush pairing.pcap -d -k 11223344112233441122334411223344 \
 | wanalyze
[...]
[✓] ltk_distribution → completed
  - ltk:  2867a99de17e3548cc17cf16ef96050e
  - rand:  38a7dcd10a1a93c6
  - ediv:  29507

[✓] irk_distribution → completed
  - address:  74:da:ea:91:47:e3
  - irk:  13c3a68f113b764cc8e73f55fc52c002

[✓] csrk_distribution → completed
  - csrk:  c3062f93c91eef96354edcd70a1a0306
[...]
```

wanalyze

Use `wplay` and `wanalyze` to recover the Long Term Keys distributed in the following PCAP file:

https://github.com/whad-team/whad-client/raw/refs/heads/main/whad/resources/pcaps/ble_pairing.pcap

Python scripting

Python API 101

- WHAD provides an user-friendly Python API if you want to write your custom scripts
- Very convenient way to automate things or implement complex behaviours !

Devices & connectors

Start by initating the communication with your RF hardware:

```
from whad.device import WhadDevice
dev = WhadDevice.create("uart0")
```

Devices & connectors

Then, you can use a dedicated **connector** (Scanner, Central, Peripheral, Sniffer...) that will expose a *specialized API*:

```
from whad.ble import Scanner
scanner = Scanner(dev)
```

```
from whad.ble import Central
central = Central(dev)
```

Closing a connector

You can properly close a connector using the `close()` method:

```
from whad.device import WhadDevice
from whad.ble import Central

try:
  dev = WhadDevice.create("uart0")
  central = Central(dev)
  while True:
    pass
except KeyboardInterrupt:
  central.close()
```

Discovering BLE devices

To discover surrounding BLE devices, instantiate a `Scanner` connector and use the `discover_devices()` method:

```
from whad.device import WhadDevice
from whad.ble import Scanner

scanner = Scanner(WhadDevice.create("hci0"))
for device in scanner.discover_devices():
  print(device.address, "->", device.name)
```

Connecting to a device

Connecting to a BLE device is as simple as instantiating a `Central` connector and calling the `connect()` method:

```
from whad.device import WhadDevice
from whad.ble import Central

central = Central(WhadDevice.create("hci0"))
target = central.connect('11:22:33:44:55:66', random=True)
```

Discovering a GATT profile

Once connected, you can discover services and characteristics using the `discover` method:

```
from whad.device import WhadDevice
from whad.ble import Central

central = Central(WhadDevice.create("hci0"))
target = central.connect('11:22:33:44:55:66')

# Discover and display the profile
target.discover()
print("[i] Discovered profile")
for service in target.services():
    print('-- Service %s' % service.uuid)
    for charac in target.characteristics():
        print(' + Characteristic %s' % charac.uuid)
```

Find characteristic by UUID

You can easily access a specific characteristic from its UUID:

```
from whad.ble.profile import UUID

# [...]

# Retrieve the DeviceName characteristic object
device_name = target.get_characteristic(UUID(0x1800), UUID(0x2A00))
print(device_name.name)
```

Reading a characteristic value

Once you got your characteristic object, you can read the characteristic value using:

```
# Reading the remote device name
device_name = target.find_characteristic_by_uuid(UUID(0x2A00))
print(device_name.value)
```

It will trigger a *read request* for the corresponding characteristic (or several if values are longer than the MTU), even if you have no read access !

Writing into a characteristic value

Writing to a characteristic's value is quite as simple as reading it, we just set the characteristic's value attribute and it starts a GATT write operation:

```
device_name = target.find_characteristic_by_uuid(UUID(0x2A00))
device_name.value = b"pwnd"
```

By default, it will trigger a *write request*. If you want to use *write command* (no response), use:

```
device_name = target.find_characteristic_by_uuid(UUID(0x2A00))
device_name.write(b"pwnd", without_response=True)
```

Subscribing for notifications

To subscribe for notification, start by writing a callback:

```
def notification_callback(charac, value: bytes, indication=False):
  print((
    f"Characteristic {charac.name} value has been "
    f"changed to {value.hex()}"
  ))
```

Then, subscribe for notification using:

```
device_name = target.find_characteristic_by_uuid(UUID(0x2A00))
if device_name.can_notify():
    if device_name.subscribe(notification = True,
                             callback = notification_callback):
      print("[i] Successfully susbcribed !")
    else:
      print("[i] An error occured.")
```

Subscribing for indication

To subscribe for indication, callback is very similar:

```
def indication_callback(charac, value: bytes, indication=False):
  # indication parameter equals True
  print(f"Characteristic {charac.name} value has been changed to {valu
```

Then, subscribe for indication:

```
device_name = target.find_characteristic_by_uuid(UUID(0x2A00))
if device_name.can_indicate():
    if device_name.subscribe(indication=True,
                             callback = indication_callback):
      print("[i] Successfully susbcribed !")
    else:
      print("[i] An error occured.")
```

Unsubscribing

Both for indication and notification, you can unsubscribe at any time using:

```
# Unsubscribe from notifications or indications
if device_name.unsubscribe():
    print((
        "Successfully unsubscribe from characteristic "
        f"{device_name.uuid}"))
```

Synchronous mode

Sometimes it may be convenient to disable the stack temporarily and handle the PDUs processing by yourself. It can be done by enabling the synchronous mode:

```
# Enable synchronous mode: we must process any incoming BLE packet.
central.enable_synchronous(True)
```

Then, all the PDUs will not be forwarded to the stack but appended to a queue instead.

Hands-on

Sending handcrafted PDUs

It's then possible to inject your own handcrafted PDUs:

```
central.send_pdu(BTLE_DATA()/BTLE_CTRL()/LL_VERSION_IND(
    version = 0x08,
    company = 0x0101,
    subversion = 0x0001
))
```

Then, wait for an answer using something like this:

```
while central.is_connected():
    pdu = central.wait_packet()
    if pdu.haslayer(LL_VERSION_IND):
        pdu[LL_VERSION_IND].show()
        break
```

Hands-on

Monitoring in and out traffic

You can attach a callback to monitor all the incoming and outgoing traffic while using a connector:

```
def processing_callback(pdu):
  pdu.show()

central.attach_callback(processing_callback)
```

Using PCAP monitor

You can also easily export the traffic into a PCAP file using the PCAP monitor:

```
from whad.common.monitors import PCAPMonitor

pcap_monitor = PCAPMonitor("out.pcap")

# Attach & start the monitor
pcap_monitor.attach(central)
pcap_monitor.start()

# [...]
# Stop the monitor
pcap_monitor.stop()
```

Using Wireshark monitor

Similarly, you can watch the live traffic through wireshark using another monitor:

```
from whad.common.monitors import WiresharkMonitor

ws_monitor = WiresharkMonitor()

# Attach & start the monitor
ws_monitor.attach(central)
ws_monitor.start()

# [...]
# Stop the monitor
ws_monitor.stop()
```

Pick your ~~poison~~ target

LIGHTBULB
clients/lightbulb.py

TRACKER ZETRACK
clients/tracker.py

HEART RATE MONITOR
clients/hrm.py

B36 WATCH
clients/b36.py

KEYFOB
clients/itag.py

ZEFIT WATCH
clients/watch.py

DIFFICULTY

Hands-on

Discover the clients

Scan for a compatible device:

```
$ ./lightbulb.py
Lightbulb:  74:da:ea:91:47:e3
```

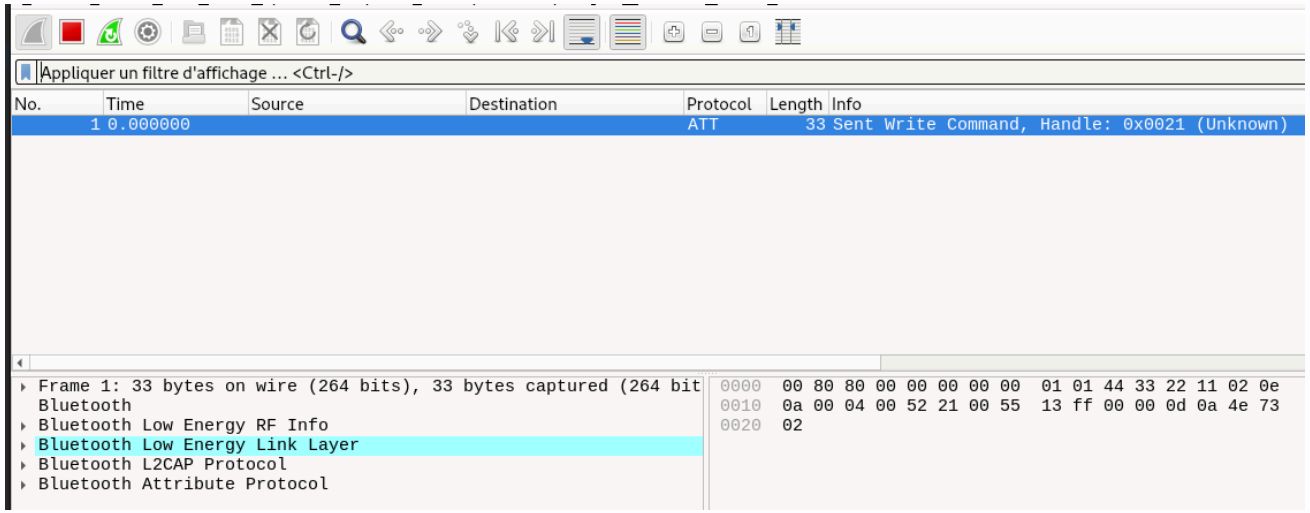
Connect to the device and play with commands:

```
./lightbulb.py -a 74:da:ea:91:47:e3
[i] Connected to  74:da:ea:91:47:e3 !
[i] Available commands:
        - wireshark [start|stop]: start or stop wireshark
        - on: turn bulb on
        - off: turn bulb off
  [...]
[[ 74:da:ea:91:47:e3 ]] ~> on
[[ 74:da:ea:91:47:e3 ]] ~> off
```

Analyze a feature

Choose some features to reproduce and analyze them by monitoring the traffic with wireshark:

```
[[ 74:da:ea:91:47:e3 ]] ~> wireshark start
[[ 74:da:ea:91:47:e3 ]] ~> color 255 0 0
```



Hands-on

Hack the world !

- Try to reproduce the features and control your target manually using `wble-central`
- Implement your own client using a python script or a WHAD script
- Emulate your device using a python script and connect it using your client !

Thank you !