Mirage: towards a Metasploit-like framework for IoT

Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, Geraldine Marconato

Mirage: towards a Metasploit-like framework for IoT

Romain Cayre*†, Vincent Nicomette*, Guillaume Auriol*, Eric Alata*, Mohamed Kaaniche*, and Geraldine Marconato†
*CNRS, LAAS, 7 avenue du colonel Roche, F-31400
Univ de Toulouse, INSA, LAAS, F-31400
†APSYS.Lab, APSYS
Email: *firstname.lastname@laas.fr, †firstname.lastname@airbus.com

***Abstract*—*Internet of Things (IoT)* devices are nowadays widely used in individual homes and factories. Securing these new systems becomes a priority. However, conducting security audits of these connected objects based on experimental evaluation is a challenging task: it requires the use of heterogeneous hardware components leading to a set of specialised software tools, generally incompatible with each other and often complex to use. In this paper, we present a security audit and penetration testing framework called *Mirage*. This framework, written in Python, is dedicated to the analysis of wireless communications commonly used by IoT devices, and provides a generic, modular, unified and low level audit environment that is easy to adapt to new protocols. The paper describes the software architecture of *Mirage*, its goals and main features, and presents a concrete example of security audit performed with this framework.**

***Keywords*-IoT; wireless; audit; security; framework**

Nowadays, fundamental changes are taking place in the information technology sector: many objects used in day-to-day life become connected and upgraded continuously with new functionalities. This major change, leading to the expansion of the Internet network into the physical world, is called "Internet of Things" (IoT).

These changes occur in a particular context. Indeed, several wireless communication protocols developers are waging a merciless economic war, trying to attract industry to engage with them. In order to address technical requirements related to these new IoT devices, such as low power consumption or mobility, *Zigbee*, *ANT+* or *Bluetooth Low Energy* (which is a variant of *Bluetooth*, designed to provide significantly lower power consumption) are engaged in a keen competition to win new markets, leading to the continuous release of new functionalities, sometimes without adequate attention to security requirements. The heterogeneity of these existing protocols and their rapid expansion is problematic [1], contributing to increasing the attack surface of those devices and consequently of the systems and networks in which these devices are deployed.

In this particular context, reliable and efficient tools and relevant experimental-based methodologies are needed to analyse and assess the security of IoT devices. Many different exploitation frameworks and exploits have been released in recent years, such as Killerbee [2] or BTLEJack [3]. However, conducting security audits of IoT devices is still a challenging task for the following reasons.

The multiplicity and the heterogeneity of wireless communication protocols used by IoT devices has led to the development of various offensive radio frequency (RF) components (e.g. Ubertooth, Micro:Bit, Yard Stick One, ...) [4]–[6]: every hardware component has its own specificities and APIs, leading the security analyst to develop time-consuming and low value source code to perform such experimental analyses. Moreover, two distinct types of hardware components are commonly used to analyse wireless communications: *Software Defined Radios* or low cost *Systems On Chip* adapted to RF communications. However, they have various limitations in terms of efficiency. *Software Defined Radios* are interesting because of their genericity and because they are highly customizable. However, they involve a large amount of effort to implement the protocols stacks. On the other hand, the *System on Chips* are generally not designed for security analysis, implying the development of customized offensive *firmware*, often poorly documented and poorly tested.

As a result, security analysts generally make use of high-level libraries that have not been developed with a security perspective in mind, or create their own libraries, resulting in a lack of modularity and flexibility. This generates costly duplicate developments and sometimes leads to poorly written code.

The code of these tools becomes more complex, and as a consequence, more likely to be errorprone. **Simplicity**, **reusability** and **modularity** are three characteristics of modern software development according to McCall's Factor Model [7], and the previously mentioned constraints have a significant impact on these characteristics (e.g. the development of complex architectures or the use of custom firmware).

The aforementioned problems have a big impact on the reproducibility of security audits based on penetration testing techniques, which is a key requirement in this context [8]. Indeed, the use of non standard libraries and software tools as well as the heterogeneity of hardware components make a systematic approach difficult. It is imperative to provide a tool to efficiently carry out experimental security audits for IoT devices. Unfortunately, to our knowledge, such a tool does not exist yet.

To fill this gap, this paper presents the design and the implementation of an original open source attack-oriented framework to support the security analysis of IoT devices,

named "*Mirage*". This framework, written in Python, targets commonly used wireless IoT communication protocols. The primary objective is to provide a modular and flexible software environment for the development of security assessment tools, similarly to the popular Metasploit framework. However, providing such a framework is a more difficult task because of IoT specificities (e.g. the large amount of protocols and offensive RF hardware components used). As a result, the proposed framework is designed to interface with any kind of RF components thanks to a versatile communication architecture. It provides an unified API to analyse the lower layers of various wireless protocols, and can be easily extended to support new protocols. Finally, *Mirage* allows complex attack scenarios to be implemented, by the combination and chaining of different modular software components.

The paper is organised as follows. Section I describes the related work, Section II briefly introduces the key design principles behind *Mirage*. Section III presents an overview of its architecture. The different protocols and attacks implemented so far in the framework are described in Section IV. Section V presents a concrete illustrative example of a security audit with *Mirage*. Finally, the main conclusions and future work are discussed in Section VI.

I. Related work

This section firstly underlines some defensive approaches, then an overview of some offensive techniques and tools is presented.

Several works present and analyse the rise of new kind of threats targeting IoT devices with a potentially devastating effect, such as the *Mirai* botnet [9] or massive coordinated attacks on the power grid [10]. As a consequence, several studies discuss the relevance of classical security approaches and investigate new mitigation measures, especially intrusion detection systems. For example, *IoT SENTINEL* [11] monitors WiFi and Ethernet traffic through the access point of the *smarthome* and isolates the identified vulnerable devices. Roux et al. [12] provide a protocol-independent approach based on the analysis of the physical layer using *Software Defined Radios*, to monitor wireless communications and detect intrusion attempts. Defensive approaches based on IoT-based Honeypots have also been discussed, such as *IoTPOT* [13] which is dedicated to analysis of Telnet-based attacks.

At the same time, several attacks targeting wireless communication protocols commonly used by IoT devices, such as *Wifi*, *Zigbee* or *Bluetooth Low Energy* (*BLE*), were also presented. In [14], Ronen et al. show that a vulnerability affecting Philips Hue light bulbs allows an attacker to control the connected objects and corrupt their *firmware* in order to create an IoT worm. Ryan demonstrates in [15] that an attacker is able to passively eavesdrop a *Bluetooth Low Energy* communication and to bypass its potential encryption. In [16], Armis releases several critical vulnerabilities targeting the *Bluetooth* stack, allowing to take control of billions of *Bluetooth* enabled devices. In [17], Goodspeed et al. demonstrate a vulnerability targeting some RF systems on chip allowing to use them as a low level injection tool. Based on this work, Newlin [18] demonstrates several critical vulnerabilities targeting the *Enhanced ShockBurst* protocol (commonly used by wireless mice and keyboards).

These new attack techniques lead to the development of interesting offensive tools and frameworks. Wright released Killerbee [2], a security framework targeting *Zigbee*-enabled devices. Cauquil provided two offensive tools, *BTLEJuice* [19] and *BTLEJack* [3], allowing to perform *Man-In-The-Middle* attacks or hijacking attacks on *Bluetooth Low Energy* connections, while improving the sniffing technique developed by Ryan [15]. Jasek developed a similar tool called *GATTacker* [20], allowing to conduct a *Man-In-The-Middle* attack on *BLE* devices. Finally, various sniffers and RF hardware components for security attacks [4]–[6] have been released and are dedicated to offensive security.

However, these various tools and techniques have been developed using different languages and libraries, which are not generic. They use different APIs and interact with a specific custom hardware, leading to a lack of modularity and flexibility. An illustrative example is the development of *GATTacker* and *BTLEJuice*. Both are based on two *nodeJS* libraries called *bleno* and *noble*. These libraries allow to implement *Bluetooth Low Energy Peripherals* and *Centrals*, but they are not designed to be used together and imply the use of two different operating systems. To address this problem, *GATTacker*'s developer rewrote the code of these libraries while the architecture of *BTLEJuice* was composed of two software components running on different OS and communicating thanks to a complex WebSockets architecture.

In this context, existing approaches and tools to support experimental security audits must be adapted and improved. In [21], Chung-Kuan et al. underline the fact that testing tools and techniques are fundamental to support IoT security. In [8], Dalalana Bertoglio & Zorzo have analysed 54 primary studies related to Penetration Testing and classified the commonly used tools. They conclude that there is almost no discussion on IoT pentesting and underline the need for reproducibility of security audits. In summary, to our knowledge, none of state-of-the art existing tools provide so far a flexible and modular environment allowing to easily integrate new protocols and attack strategies, while providing a stable and unified API for assessing the security of wireless IoT devices. The main objective of our framework is to fill this gap.

II. Key Principles

The proposed attack-oriented framework named *Mirage* is aimed at developing a modular and flexible software environment allowing to address the main constraints inherent to this type of offensive security tools, especially the heterogeneity of RF hardware components commonly used in the IoT world and the lack of low level attack-oriented libraries covering IoT wireless communication protocols. Four main principles have guided and motivated our work, and are discussed in the following subsections.

A. Providing an unified API

Nowadays, conducting experimental-based security audits of connected objects implies that security analysts must use at the same time several different software tools, each providing its own API and documentation, and potentially using specific file formats, and each of these tools has its own limitations.

As a consequence, the analyst has to learn a lot of technical information that is not directly linked to the audit workflow and is generally neither relevant nor reusable. In addition, they must install the different, and potentially dependent, tools and libraries, leading to increasingly complex solutions. This situation involves rewriting many non-reusable codes to work together or to integrate specific functionalities.

Our framework is designed to seamlessly integrate the different software components and provide a unified API. Indeed, each software component can be configured via a similar interface and uses the same type of display, logging and output mechanisms. The multiple components used to manipulate the wireless communication protocols closely follow the same lines and expose a common API. In addition, the framework architecture requires developers to follow these guidelines while developing audit modules or implementing new protocols.

This approach facilitates the interactions between different pieces of code while harmonising the use and implementation of attacks.

B. Modularity and reusability

Modularity is one of the main features of our framework. Indeed, some attacks targeting a specific protocol implement similar elementary actions and behaviours. For example, fuzzing and cloning a *Bluetooth Low Energy* devices involve the same type of actions: the tool must scan the RF environment to find the target device, then connect to that device and discover the services and features, etc. Finally, the specific behaviour (fuzzing the device or simulating a similar one) can be triggered.

To prevent code redundancy and facilitate maintenance, the framework is designed to divide complex attacks scenarios into small functional modules. Therefore, in the aforementioned use cases, attacks could be broken down into small code units: a scanner, a connection module and a discovery module are provided and can be used in both scenarios, avoiding code redundancy. *Mirage* allows existing modules to be directly reused in a new attack implementation, but also to be executed sequentially using a chaining operator similar to the pipe operator in UNIX environments. This behaviour allows to quickly generate complex attack workflows by controlling the combination of several modules.

This modular approach allows us to cover a large amount of existing attack tools without rewriting them all in our framework. Therefore, a security analyst can really focus on his attack workflow without having to write a lot of irrelevant code.

C. Genericity

Many different RF hardware components are nowadays used to perform experimental based security audits. However, each offers different features and APIs, and a significant part of an analyst's work is devoted to understanding the functionalities provided by a specific component and implementing the corresponding methods. As a result, one of the most important guidelines for us was to design an architecture able to manage these multiple hardware components while following the aforementioned principles, e.g. by providing a unified API to use them.

Each wireless communication protocol commonly used in IoT devices has its own specificities. However, we have identified similar behaviours. As a consequence, we have chosen to design a generic communication architecture that allows new protocols to be easily integrated or existing ones to be manipulated. Currently, this architecture is functional and we have successfully integrated eight hardware components related to several protocols such as *Bluetooth Low Energy*, *Wifi*, *Zigbee*, *Enhanced ShockBurst* and *Infrared radiation* (IR).

D. Low level analysis

As mentioned above, many different attack tools use high-level libraries. These libraries are generally not designed for security analysis, and they potentially have certain constraints and limitations that may have a significant impact on the tools design.

As a consequence, it is necessary to allow the security analyst to work on the lower layers of the communication protocol stacks. To address this problem, we have implemented flexible and modular protocol stacks, allowing to deeply modify the behaviour of the protocols and easily manipulate the lower layers accessible by software.

This approach has been successful in the implementation of *Bluetooth Low Energy Man-in-the-Middle* attacks. Working at a lower level allowed us to avoid the limitations implied by the previously mentioned libraries: our *Man-in-the-middle* implementation does not require multiple operating systems or need to fully clone the GATT layer of the device to work, allowing us to directly redirect the packets without simulating an entire *BLE* device.

III. Architecture overview

This section describes the key features of our framework architecture. We present the main software components composing our architecture, and we focus on our generic communication architecture. Then, we focus on the modularity of our framework by presenting the concepts of Modules and Scenarios and introducing the chaining operator designed to execute modules sequentially in a pipe, as does the UNIX shell with commands.

A. Main software components

Mirage framework is composed of four main components, as shown in Figure 1:

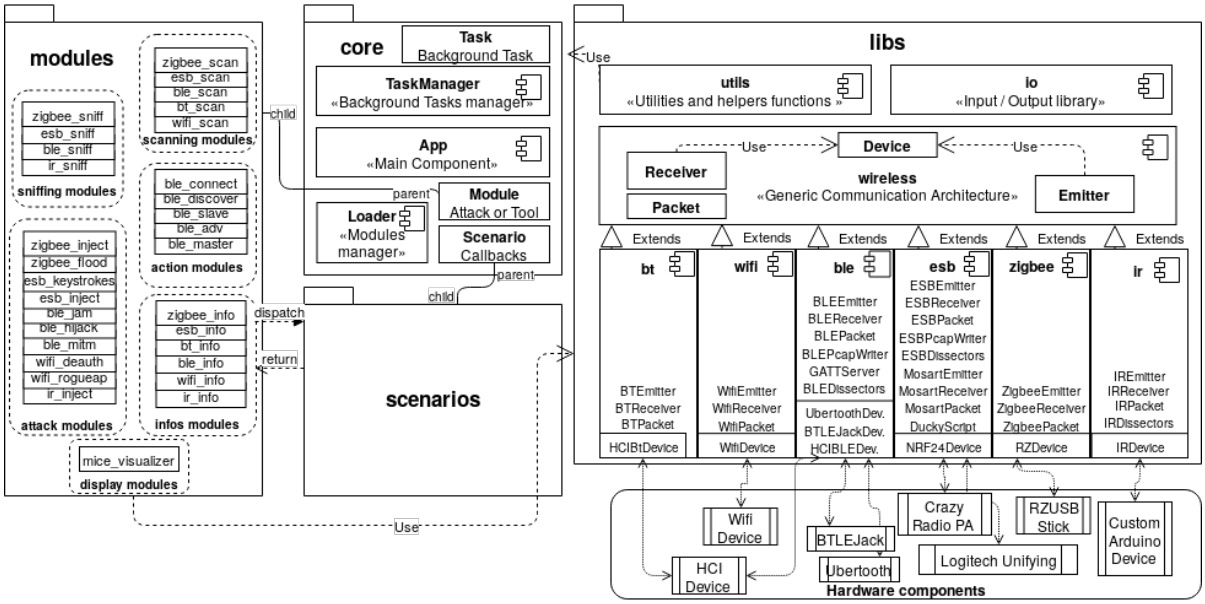

Fig. 1. Global architecture of *Mirage* framework

1) **The core component (“*core*”)**: this component includes the core mechanisms of our framework to load, configure and execute the modules, but also to manage the background tasks, signals and configuration files. It provides an unique entrypoint, allowing the *framework* to be used from a command line interface or directly from the shell environment via the execution of scripts.
2) **The internal libraries (“*libs*”)**: this component is in charge of implementing the wireless protocol stacks, and provides a generic communication architecture to easily integrate new protocols. It provides some display and logging mechanisms and some utilities and helpers functions (e.g. modules and background tasks manipulation, time management, ...).
3) **The attacks and tools (“*modules*”)**: these software components, called ”modules”, are independent and implement the attacks and tools provided by the framework. They provide a specific service such as protocol sniffing or active attacks, and can be used independently or sequentially thanks to the chaining operator.
4) **The callbacks (“*scenarios*”)**: some modules, such as *Man-in-the-Middle* attacks or *devices* simulation, implement some complex behaviours and provide a standardised API to quickly customize their execution. The *scenarios* are specialised classes composed of bindings methods providing simple APIs.

B. Generic communication architecture

One key feature of our framework is the generic communication architecture. Indeed, many of the wireless communication protocols commonly used by IoT devices have their own specificities, but several similar patterns can be extracted and have made it possible to design a generic architecture. Moreover, many different RF hardware components can be used to communicate with a given protocol, and our architecture must be flexible enough to integrate them easily.

Our design defines a generic way to handle multiple protocols in order to provide a unified API, but also allows the implementation of the specific behaviour of each protocol, while handling multiple RF devices and their key characteristics.

Mirage communication architecture is composed of three main software components, depicted in Figure 2.

The **Device** class manages the interfacing with the various RF hardware components. As a result, several classes can inherit this abstract class and implement the main methods for sending or receiving a specific frame as a binary representation, check if the hardware component is connected and ready to use, and initialize the component. However, some devices provide additional features which can be implemented as independent methods. Their method names must be added to **sharedMethods**, a class attribute that is an array listing the specific features available. An instance attribute called **capabilities** can also be provided and defines some high level capabilities of the device, indicating the available functionalities. This class is not directly available in the modules, but the specific behaviour implemented as shared methods can be directly used.

Another class named **Receiver** can be defined. It is able to instantiate the right **Device** class according to the **interface** parameter, provided by the end user, and communicate directly

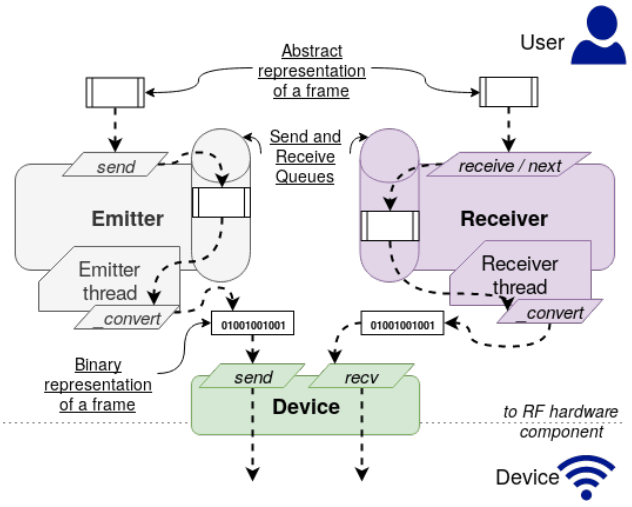

Fig. 2. Generic communication architecture of *Mirage* framework

with it. One main method is needed, called **_convert**. It converts a binary frame provided by the device into an abstract representation. This mechanism allows the programmer to provide a rich interface to user while manipulating frames (e.g. including dissectors, builders or converters). The class **Receiver** exposes two methods to get the received frames, and allows the user to register some callback functions which can be triggered at the reception of a specific frame, at the reception of a given number of frames or at each received frame and can be run in a background thread or in foreground.

The abstract class **Emitter** provides mechanisms similar to **Receiver** for emitting frames. It also includes a **_convert** method, and the child classes have to implement it to convert an abstract frame object into a binary representation. It exposes one main method, **send**, for sending frames from the modules.

The **Emitter** and **Receiver** classes include *First In First Out* data structures (the send and receive queues) to store temporarily the abstract representation of frames. The **Emitter** class includes a background thread for converting these objects into binary frames thanks to the **_convert** method previously mentioned and transmits the resulting bytes array to the corresponding **Device**'s method (**send**). Another thread is launched in background by the **Receiver** class and gets the received frames from **Device**'s **recv** method, converts them into their corresponding abstract representations and populates the receive queue.

Finally, it should be noted that end users cannot directly instantiate the **Device**'s classes. According to the design pattern called *Registry*, devices linked to a specific interface are instantiated only once by an **Emitter** or **Receiver** instance, and the same device can be used by multiple emitters or receivers. Methods corresponding to specific behaviours cannot be directly called by the users, but they are provided by the **Emitter** and **Receiver** classes which implement the design pattern called *Proxy*.

This design presents some interesting properties: 1) the most common actions (e.g. receiving and sending frames) are facilitated, and the API provided is the same for each protocol; 2) it allows specific features provided by the hardware components to easily manipulated, without looking at their respective APIs; 3) the frames are manipulated as an abstract representation, allowing powerful mechanisms such as dissectors to be added.

C. Modules and scenarios

The modules are the key elements of our framework: they are used to implement the attacks and tools. They inherit and extend a class called **core.Module**, to quickly prototype and develop an offensive strategy.

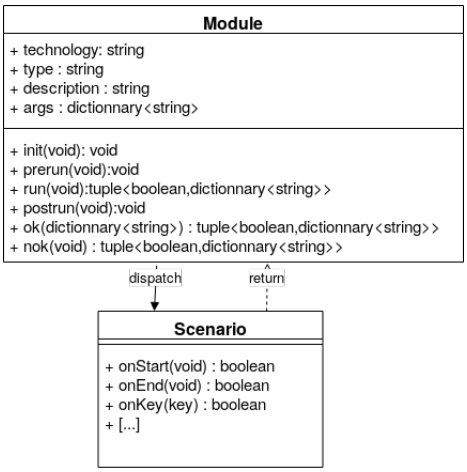

Fig. 3. Architecture overview of a module

As shown in Figure 3, every module must implement an **init** method, allowing the input parameters to be initialized as a dictionary and providing three main instance attributes, used by the core component to classify them:

- **technology**: this attribute indicates the wireless communication protocol targeted by the module. The corresponding emitters and receivers are automatically selected according to its value.
- **type**: this attribute is used to provide the type of tool implemented by the module. It allows the module to be easily classified.
- **description**: this attribute is a short string describing the role of the module.

The module behaviour can be customized by passing named arguments as inputs. These arguments are defined as the keys of a dictionary called **args**, and the corresponding values are used to provide default values.

This **init** method is called directly by the constructor when the module class is instantiated. The modules are dynamically loaded and instantiated by the class **Loader**, included in the **core** component. This class lists the files included in the *modules* sub-directory, instantiates the modules classes found into them and classifies them. This mechanism is automatically executed at the beginning of the execution, allowing the user to focus on the development of the module.

Another main method, called **run**, must be implemented. This is the main method of a module because it contains the

code that implements the attack or tool behaviour. This method is called at the beginning of the module execution. It returns a dictionary composed of a boolean value (indicating whether the module execution is successful or not) and a dictionary (providing the potential output parameters), which can be easily generated thanks to two helpers methods, called **ok** (if the execution is successful) and **nok** (if an error occurred during the module execution). If some specific actions need to be executed before or after the module execution, the developer can implement two additional methods named **prerun** and **postrun**.

Finally, some complex modules such as *Man-in-the-Middle* attacks can be highly customized by filling an input argument named "*SCENARIO*". It allows to provide a name corresponding to a child class of **core.Scenario**. It allows to easily customize the behaviour of a module by providing some callback methods, called if the module triggers the corresponding event. An event named "*onKey*" is automatically triggered if a key is pressed, which provides a basic user interface during execution of a module.

This design allows new attacks or tools to be easily prototyped or developed while ensuring a high level of modularity. Indeed, this approach forces the developers to follow the framework guidelines, which leads to a modular software environment. However, it is flexible enough to allow complex developments and the scenarios allow the developer to provide an elegant way to customize the behaviour of this module without changing the corresponding code.

D. Chaining operator

Another key feature has been added to our framework, to easily combine different modules to set up complex attack workflows: the chaining operator, called "*pipe*". Indeed, several attacks are composed of the same type of actions. For example, cloning a device or launching a fuzzing attack imply the use of similar actions, such as scanning the RF environment or connecting to the device. While it is still possible to use existing modules in a new module, a common need is to sequentially execute existing modules to compose customised attack workflows without writing a module. So, we have included a chaining operator inspired by the pipe operator, commonly used in UNIX environments.

The chaining operator included in our framework operates in a similar manner, allowing a data pipeline to be created between two modules. Every module can be customised by passing named parameters as inputs and can generate named parameters as outputs: as a result, our operator allows to sequentially execute two modules and propagate the outputs from the first module to the inputs of the second one, according to their name. If an output is not used by the next module in the pipeline, it will be stored to be used later by a next module included in the pipeline.

If a module included in the pipeline fails, sequential execution is interrupted. Several modules make use of the classes **Emitter** and/or **Receiver** mentioned previously. Therefore, if a given interface is used by a module, each subsequent module included in the pipeline using an **Emitter** or a **Receiver** based on the same interface does not need to re-instantiate these classes but automatically reuses the existing ones (according to the design pattern called *Registry*). This mechanism allows a complex attack workflow to be divided into simpler actions, leading to a powerful and modular approach. An example of such an execution is shown in Figure 4.

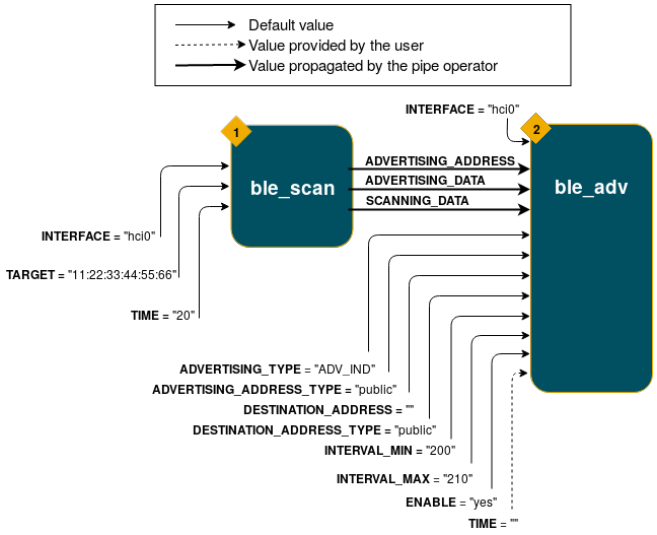

Fig. 4. Example of sequential execution

IV. Protocols and modules

Several protocols commonly used by IoT devices have been integrated into our framework, and several different modules have been developed. In the following subsections, we present an overview of this work by describing the protocol stacks included in *Mirage* and the corresponding modules. Finally, we underline the development process to integrate a new protocol or add a new attack module.

A. Bluetooth and Bluetooth Low Energy

A lot of work has been done to integrate *Bluetooth* devices, especially *Bluetooth Low Energy* devices. Indeed, this technology is often used by connected objects because of its low power consumption and its massive integration in smartphones and tablets.

A partial implementation of a *Bluetooth Classic* stack is included in *Mirage*. It implements a subset of *Bluetooth* layers (especially those used to inquiry and connect to devices) and works by communicating directly with the *Host Controller Interface* (HCI), without requiring the use of an external library. As a result, it makes it easy to use an HCI device such as *Bluetooth* dongles. Moreover, some providers provide an interesting feature in order to develop attack modules: they include in their hardware design some *vendor-specific* HCI frames allowing to change the *BD* address of *dongles*. As a result, this functionality has been included in *Mirage* and makes it easy to spoof a *BD* address and impersonate the identity of a targeted device. However, a lot of additional work

is required to provide a complete stack due to the multiplicity of application layers supported by this technology.

Currently, two main modules related to this technology can be used in our framework: **bt_info** and **bt_scan**. The first module allows useful information about the specified interface to be displayed, while **bt_scan** allows to launch an inquiry scan and identify the visible *Bluetooth* devices in the RF environment.

Many software components have been implemented in our framework, to perform security audits on *Bluetooth Low Energy* devices. Indeed, the *Bluetooth Low Energy* stack inherits from the *Bluetooth Classic* one, allowing to reuse some interesting features (e.g. *BD* address spoofing) while adding an exhaustive *Bluetooth Low Energy* stack implementation. This stack uses directly *Host Controller Interface* without requiring additional libraries, and it also provides several dissectors and helpers functions to easily analyse and generate data from upper layers, such as the Attribute Protocol (*ATT*) and Generic Attribute Profile (*GATT*) layers. A complete *GATT* server has been implemented, allowing to easily simulate a *BLE* device using the *Peripheral* role.

Two main RF hardware components are commonly used to sniff *Bluetooth Low Energy* communications: *Ubertooth* and *BTLEJack*. These hardware components are fully supported by *Mirage* and a unified API to control them is provided. Some additional features have been included in a custom version of *BTLEJack firmware*, allowing to easily sniff and selectively jam *advertisements*, both versions are fully supported by our framework. A PCAP writer is also provided, allowing *Bluetooth Low Energy* sniffed frames to be exported to a PCAP file.

The previously mentioned sniffers can jam *BLE* communications and *BTLEJack* is able to hijack such a communication. These features can be directly used by the framework and the *BTLEJack* hijacking attack can be combined with some active modules commonly used with HCI dongles, to highly customize the attack workflow.

Many different modules, have been included and can be used together to perform complex actions. **ble_info** lists the available interfaces. **ble_mitm**, **ble_hijack** and **ble_jam** allow active attacks to be performed, while **ble_sniff** can be used to passively eavesdrop the communications. Some modules are also provided in order to execute legitimate actions such as **ble_scan**, **ble_adv**, **ble_connect**, **ble_discover**, **ble_master** or **ble_slave**: all these modules expose some specific scenarios events, allowing to deeply customize their behaviour.

B. Zigbee

Since two security frameworks targeting the *Zigbee* protocol (called *Killerbee* and *Secbee*) have been published in recent years, a partial implementation of the *Zigbee* stack developed from scratch is included in the framework as a proof of concept. It allows the user to interact with a *RZUSBStick* from *Atmel* using *Killerbee firmware*. Five corresponding modules are provided, allowing to display information about an interface (**zigbee_info**), scan the RF environment to identify target networks (**zigbee_scan**), sniff *Zigbee* frames on a given channel (**zigbee_sniff**), inject *Zigbee* frames (**zigbee_inject**) or run a Denial of Service attack by flooding a *Zigbee* router with *association* frames (**zigbee_floodassoc**).

C. Enhanced ShockBurst and Mosart

Several relevant works focusing on *Enhanced ShockBurst* and *Mosart* protocols have been published in the recent years. These protocols are widely used by input devices such as mice or keyboards, and many vulnerabilities targeting this kind of hardware have recently been published, allowing to inject keystrokes or mouse-related frames.

As a result, *Mirage* includes a partial *Enhanced ShockBurst* stack and a complete *Mosart* stack, allowing the user to easily sniff and inject frames. It interacts with a *CrazyRadio PA* dongle or a *Logitech Unifying* dongle embedding the *nRF Research firmware* developed and released by *Bastille Networks*. Several dissectors are provided to analyse mouse movements or keystrokes, and a *DuckyScript* interpreter has been added to facilitate attacks targeting keyboards. Finally, a generic component allows to generate a graphical view of mouse movements and can be combined with the previously mentioned sniffer.

Several attack modules are provided for both *Enhanced ShockBurst* and *Mosart* protocols. The interfaces can be enumerated thanks to **esb_info** and **mosart_info**, **esb_scan** and **mosart_scan** allow to scan the channels in order to identify devices, **esb_sniff** and **mosart_sniff** are used to sniff the frames on a given channel. Some active modules are also provided, such as **esb_keystrokes** and **mosart_keystrokes** that allow to generate an attack trace containing unencrypted keystrokes as a pcap file, while **esb_inject** and **mosart_inject** can be used to inject frames. A generic module called **mice_visualizer** allows to visualise the mouse movements for both protocols.

D. Wifi

Wifi is a well-known technology in terms of offensive security, so we have decided to focus on other protocols more specific to IoT and to implement a minimal stack as a proof of concept. Currently, this stack allows to control management frames such as *deauthentication*, *disassociation* or *probe* frames. As a result, four main modules have been included in our framework: **wifi_info** provides some useful information about the interface used, **wifi_scan** allows to discover access points and stations, **wifi_deauth** allows to run a denial of service attacks by injecting *deauthentication* or *disassociation* frames while **wifi_rogueap** simulates an access point (without accepting connections).

E. IR protocols

Infrared Radiations are widely used by manufacturers to control connected objects: it's probably one of the cheapest technology available for short range communications. As a result, we integrate many protocols based on this physical layer in *Mirage* (e.g. *RC5* or *Sony*).

As far as we know, no specific hardware has been released targeting these IR protocols. As a consequence, we have designed and implemented a custom hardware component based on an Arduino Uno, allowing easy sniffing and manipulation of *IR* frames. The corresponding *firmware* and schematics are open source and can be quickly reproduced and improved.

Three main modules are provided by *Mirage* to manipulate these protocols: **ir_info** (displays useful information about an *IR* interface), **ir_sniff** (passively eavesdrop an *IR* frame) and **ir_inject** (injects an *IR* frame).

F. Adding new protocols and modules

One of the main advantages of our framework is that a new protocol can be easily added. First, the developer has to implement a new child class of **libs.wireless.Device**, allowing interaction with a given specific hardware. This class should provide only four main methods: **a) init** (initialising the hardware component), **b) isUp** (indicating if the hardware can be used), **c) recv** (allowing to receive frames) and **d) send** (allowing to transmit frames). As a result, developing a driver is straightforward.

Some specific features can also be added by creating a new method and appending its name to the array **sharedMethods**, allowing them to be called from a module environment.

At least one of the two child classes of **libs.wireless.Receiver** or **libs.wireless.Emitter** must be implemented. They initialize the device previously defined in their constructor and must implement the **_convert** method, allowing to convert a binary frame into an abstraction or a abstract representation of a frame into an array of bytes. As a result, the protocol is fully integrated and can be used from modules.

Developing an attack targeting this new protocol implies creating a new python file in the *modules* sub-directory. A child class of **core.Module** with the same name will be created into this file, allowing the **core.Loader** component to find this new module. The developer has to integrate an **init** method and provide the main necessary attributes (**technology**, **type**, **description** and **args**). Then, he can instantiate the previously created **Emitter** and **Receiver** and implement his attack by developing the corresponding **run** method.

V. Experimentation

This section is dedicated to the presentation of a security audit of a smart connected bulb that we performed with our framework. This experiment is aimed at illustrating the relevance and efficiency of our framework. It also shows how simple it is, using this framework, to perform such security analyses, that have so far been quite complex to carry out, using heterogeneous and sometimes incompatible tools. It should be noted that many other experiments have been performed using our framework, such as security audits of connected objects or the evaluation of an Intrusion Detection System for IoT. The framework also allowed us to discover 20 new vulnerabilities targeting five commercial products.

The main objective of this experiment was first to reverse engineer the communication protocol of the bulb in order to evaluate its attack surface. The bulb is managed, through a *BLE* communication, by an Android application running on a smartphone. This application allows the bulb user to choose its color, change its brightness, turn it on or off, and update its *firmware*.

A. Information gathering

First of all, it was necessary to use this Android application to register the bulb in the application and activate the various legitimate functionalities of the bulb. During these operations, the framework was used to analyse and understand the following behaviours: a) change brightness, b) change temperature, c) switch on/off, d) change color and e) update *firmware*.

After this first analysis, the next step consists in identifying the list of the *ATT* server attributes, stored in the bulb, as well as their *GATT* abstractions, under the form of primary, secondary services and characteristics.

To do this, it was necessary to dump the *ATT* and *GATT* databases. The following modules were used: **a) ble_scan** (in order to scan the environment to identify the *advertisements* of connected objects within radio range), **b) ble_connect** (in order to establish a connection to a specific object), **c) ble_discover** (in order to enumerate the services, characteristics and attributes associated to the *ATT*/*GATT* layers of a specific object).

The first step was to launch the **ble_scan** module, whose outputs were the following ones:

```
$ sudo ./mirage.py ble_scan
[INFO] Module ble_scan loaded !
[SUCCESS] HCI Device (hci0) successfully instanciated !
```

Devices found

BD Address	Name	Company
XX:XX:XX:39:8E:07	Salon	Texas Instruments Inc.

This scan enabled to obtain three interesting information items: the BD address, the manufacturer of the system as well as the name of the object. These information items were extracted from the *advertising* packets. In this study, the name of the smart bulb is "**Salon**", its BD address is $XX{:}XX{:}XX{:}39{:}8E{:}07$ and the manufacturer name is "**Texas Instruments Inc.**". The next step was to perform a connection to the object and then dump the services, characteristics of the object (at the *GATT* level). The chaining operator integrated in *Mirage* allowed us to easily combine two existing modules (**ble_connect** and **ble_discover**) to obtain the structure of the high level protocol layers, i.e. *GATT* layer. It is also possible to export this information in a .cfg file, by setting the *GATT_FILE* parameter of the **ble_discover** module.

```
$ sudo ./mirage.py "ble_connect|ble_discover" //
ble_connect1.TARGET=XX:XX:XX:39:8E:07 //
ble_discover2.GATT_FILE=/tmp/gatt.cfg
```

The output of the module indicates that three services (quite common for most connected object) are available on the smart bulb: **a) Generic Access** (*handles* 0x0001 to 0x000b), **b)**

Generic Attribute (*handles* 0x000c to 0x000f), **c) Device Information** (*handles* 0x0010 to 0x001e).

Three other services, specific to the bulb, are also available from *handles* 0x001f to 0x002f, 0x0030 to 0x0039 and 0x003a to 0xFFFF.

Furthermore, two interesting features are associated with the first service: **DataTransmit** (*handle* 0x0020) and **DataReceive** (*handle* 0x0023).

B. Reverse-engineering of the communication protocol

In order to accurately identify the behaviour of the object, a *Man-In-The-Middle* attack was performed, while the different functionalities of the bulb were activated thanks to the Android application on the smartphone. The *Man-In-The-Middle* attack allowed us to analyse the traffic corresponding to these functionalities. At first, since no specific scenario was loaded in the *Man-In-The-Middle* module, the default behaviour was applied (redirection and logging of the packets).

```
$ sudo ./mirage.py ble_mitm //
TARGET=XX:XX:XX:39:8E:07 //
SHOW_SCANNING=no //
ADVERTISING_STRATEGY=btlejuice
```

This attack allowed us to identify the format of the command messages. They are triggered by a **Write Request** to the *handle* of value 0x0021 (which corresponds to the **DataTransmit** characteristics, identified during the previous step of the analysis).

The messages format is as follows:

0x55	identifier – 1 byte	parameter	0x0d 0x0a

Every action is performed using a specific identifier (e.g. 0x10 for switching on or off, 0x13 for color modification ...) and the corresponding parameter. As an example, the messages intended to modify the color include the identifier **0x13**, followed by three bytes corresponding to the hexadecimal *RGB* code of the required color, as shown in Table I.

TABLE I
MESSAGES FORMAT RELATED TO COLOR MODIFICATION

Color modification (Red)	55 13 ff 00 00 0d 0a
Color modification (Green)	55 13 00 ff 00 0d 0a
Color modification (Blue)	55 13 00 00 ff 0d 0a

To confirm our assumptions, we performed a connection to the bulb and executed the **ble_master** module. Then, another *Man-In-The-Middle* attack was also performed, in which our framework was able to modify on the fly the different commands sent by the smartphone application to the bulb (for instance, the color Red and Green were exchanged, as well as the switch on/off action).

C. Obtaining a firmware *dump*

The last step of our security audit was to analyse the *firmware* update procedure of the bulb. Indeed, when connecting the smartphone application, a dialog box proposes to update the *firmware* of the bulb *over the air*. To analyse this update process, we used the **ble_sniff** module:

```
sudo ./mirage.py ble_sniff //
CHANNEL=37 SNIFFING_MODE=newConnections //
INTERFACE=microbit0
```

The data dumped during this sniffing attack allowed us to identify eight different steps of the update process (mostly *Read Requests* and *Write Commands* on different *handles*).

Once these messages are exchanged, the Master starts to write, by means of **Write Commands**, in the *handle 0x0040*, values formatted as follows:

```
000017defffff0500007c42424242ffffffff
0100ffffffffffffffffffffffffffffffff
020000010203040506070809a0bffffffff
[...]
```

By analysing the format of these messages, we we were able to see that the first two bytes represent a counter, followed by the contents of the *firmware*, sent by 16 bytes data blocks. The attack scenario **slave_lightbulb** was then built, (based on the **ble_slave** module) in order to dump the whole *firmware* in the "firmware.bin" file. Creating an identical clone of the bulb can offer many advantages. First, it enables to simulate the behaviour of the object to be audited. It may also be used to perform a denial of service attack of the legitimate object.

Such a strategy could easily be instantiated in our framework, thanks to the following chained execution: **a) ble_scan** (dumping of *advertisement* data),**b) ble_connect** (connection to the bulb), **c) ble_discover** (dumping of GATT services and characteristics in a .cfg file), **d) ble_adv** (sending *advertisements*) and **e) ble_slave** (creation of a *BLE Slave* using the same GATT data and implementing the *slave_lightbulb* scenario). This example illustrates the relevance of introducing a chaining operator, which allows complex attack workflow to be designed without writing a single line of code.

```
$ sudo ./mirage.py //
"ble_scan|ble_connect|ble_discover|ble_adv|ble_slave" //
ble_scan1.TARGET=XX:XX:XX:39:8E:07 //
ble_discover3.GATT_FILE=/tmp/gatt.cfg //
ble_adv4.INTERFACE=hci1 //
ble_slave5.SCENARIO=slave_lightbulb
```

After the information collection phase and the creation of the *BLE* slave, the latter was executed and produced the following output:

```
[...]
[SUCCESS] HCI Device (hci1) successfully instanciated !
[INFO] Importing GATT layer datas from /tmp/gatt.cfg ...
[INFO] Scenario loaded !
[INFO] Updating connection handle : 68
[INFO] Master connected : 73:5E:A2:21:C7:9D
[...]
[INFO] Sending notification (1)...
[INFO] Sending notification (2)...
[INFO] Starting firmware recuperation ...
[INFO] Writing #0...
[INFO] Write Command : handle = 0x40 /
value = 000017defffff0500007c42424242ffffffff
[...]
```

After this operation, the *firmware* was dumped and available in the "/tmp/firmware.bin" file.

D. Conclusion

In summary, this security audit was carried out relatively easily using our framework and quickly revealed enough

useful information to produce a relevant security assessment report. No security mechanism seems to be implemented on this smart object. The sending of commands as well as the *firmware* update *Over The Air* are carried out through unencrypted data. Reverse engineering of the *firmware*, even if out of scope of this paper, makes it relatively easy for an attacker to develop a malicious *firmware* that could be loaded on the bulb, and which could possibly enable the set up of some form of smart bulbs based botnets. Such malware propagation through smart objects is today a real threat and a framework like *Mirage* could be very useful to ease the audit of smart objects and prevent the development of such botnets by applying adequate protection mechanisms.

VI. Conclusion

In this paper, we presented a new security audit and penetration testing framework called *Mirage* dedicated to IoT devices, focusing on the analysis of widely used wireless communication protocols. It offers a flexible software environment for developing new tools and attacks thanks to a modular architecture and the introduction of a chaining operator. We also described a generic communication architecture that allows new protocols to be easily integrated and provides a unified API for multiple technologies. Then, we highlighted multiple protocols and modules already included in *Mirage*, demonstrating that several existing attacks could be easily integrated into our framework and sometimes improved, thanks to a low level architecture. Finally, we described an experiment to demonstrate its usability. It should be noted that our framework has been successfully used to evaluate an Intrusion Detection System dedicated to IoT. It has been useful in easily automating the generation of intrusion attempts in a smart-homes context, leading to an efficient evaluation process.

The framework is publically available as an open-source project[1], and we intend to actively maintain and extend it by including additional and new wireless protocols. As future work, we plan to use *Mirage* as a passive eavesdropping tool, in order to monitor wireless traffic and perform intrusion detection. Indeed, many different passive modules are already implemented in the framework and allow us to easily analyse frames from various protocols. As a result, we plan to integrate new protocols such as ZWave, ANT+ or LoRaWAN, and add relevant modules to analyse a complete wireless environment such as for network topology inference. Such a monitoring and intrusion detection tool dedicated to wireless network communications of connected objects would constitute, to our viewpoint, a novel and relevant contribution to support the security analysis of IoT wireless communications and develop appropriate protection mechanisms.

[1] **Repository**: https://redmine.laas.fr/projects/mirage

References

[1] Z.-K. Zhang *et al.*, "Iot security: Ongoing challenges and research opportunities," in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pp. 230–234, 11 2014.

[2] J. Wright, "KillerBee: Practical ZigBee Exploitation Framework," 2009. http://www.willhackforsushi.com/presentations/toorcon11-wright.pdf.

[3] D. Cauquil, "You'd better secure your BLE Devices or we'll kick your butts !," 2018. https://media.defcon.org/DEFCON26/DEFCON26presentations/DamienCauquil-Updated/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf.

[4] D. Spill, "Ubertooth," 2012. http://ubertooth.sourceforge.net/.

[5] D. Cauquil, "Weaponizing the BBC Micro:Bit," 2017. https://media.defcon.org/DEFCON25/DEFCON25presentations/DEFCON-25-Damien-Cauquil-Weaponizing-the-BBC-MicroBit.pdf.

[6] Atlas, "SubGHz or Bust," 2012. https://media.blackhat.com/bh-us-12/Briefings/Atlas/BH_US_12_Atlas_GHZ_Workshop_Slides.pdf.

[7] J. P. Cavano and J. A. McCall, "A framework for the measurement of software quality," *ACM SIGSOFT Software Engineering Notes*, vol. 3, pp. 133–139, 11 1978.

[8] D. Dalalana Bertoglio and A. Zorzo, "Overview and open issues on penetration test," *Journal of the Brazilian Computer Society*, vol. 23, 12 2017.

[9] M. Antonakakis *et al.*, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 1093–1110, USENIX Association, 2017.

[10] S. Soltan, P. Mittal, and H. V. Poor, "Blackiot: Iot botnet of high wattage devices can disrupt the power grid," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 15–32, USENIX Association, 2018.

[11] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," *CoRR*, vol. abs/1611.04880, 2016.

[12] J. Roux, E. Alata, G. Auriol, M. Kaâniche, V. Nicomette, and R. Cayre, "Radiot: Radio communications intrusion detection for iot - A protocol independent approach," *CoRR*, vol. abs/1811.03934, 2018.

[13] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: Analysing the rise of iot compromises," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, (Washington, D.C.), USENIX Association, 2015.

[14] E. Ronen, A. Shamir, A.-O. Weingarten, and C. OFlynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 195–212, 05 2017.

[15] M. Ryan, "Bluetooth: With low energy comes low security," in *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2013.

[16] Armis, "Blueborne Technical White Paper," 2017.

[17] T. Goodspeed, S. Bratus, R. Melgares, R. Shapiro, and R. Speers, "Packets in packets: Orson welles' in-band signaling attacks for modern radios," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pp. 7–7, 08 2011.

[18] M. Newlin, "MouseJack : White Paper," 2016. https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/DEFCON-24-Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.whitepaper.pdf.

[19] D. Cauquil, "BtleJuice, un framework d'interception pour le Bluetooth Low Energy," 2017. https://www.slideshare.net/NetSecureDay/nsd16-btle-juice-un-framework-dinterception-pour-le-bluetooth-low-energy-damien-cauquil.

[20] S. Jasek, "Gattacking Bluetooth Smart Devices," 2017. https://github.com/securing/docs/raw/master/whitepaper.pdf.

[21] C.-K. Chen, Z.-K. Zhang, S.-H. Lee, and S. Shieh, "Penetration testing in the iot age," *Computer*, vol. 51, pp. 82–85, 04 2018.