PENIOT Final Document

Project acronym: PENIOT
Date : 12/06/2019
Members: Berat Cankar
Bilgehan Bingöl
Doğukan Çavdaroğlu
Ebru Çelebi
Supervisor: Dr. Pelin Angın

The purpose of this document is to clarify and explain the project PENIOT. Respectively, the document will focus on brief product description, features and testing process in a detailed way. Briefly, PENIOT is an extensible penetration testing tool for the Internet of Things (IoT) devices.

1. Product Description

PENIOT is a penetration testing tool for Internet of Things (IoT devices). Penetration testing is the practice of controlled security testing of a digital system to find possible security breaches. It is usually done manually by penetration testers, not by dedicated software. What we are building is a helper tool, a first step for penetration testers that want to do penetration tests on IoT devices. We are not trying to replace penetration testing personnel, rather we want to give them a go-to tool to use in the beginning of their work.

Security is a big problem in IoT, and our tool is designed to alleviate this problem by giving security experts an easy to use, extendable penetration testing tool. Our tool starts with an easy to use graphical user interface (GUI). From that GUI users can choose the type of protocol they want to work on, and the type of the security attack they want to implement. Our tool has four common IoT protocols as default: MQTT, Bluetooth Low Energy (BLE), CoAP and AMQP. On top of that, PENIOT has built in security attacks including fuzzing, denial of service (DoS), sniffing and replay attacks although for some protocols some of these attacks may not be available.

PENIOT can be directly used on hardware devices to test these devices for security breaches. PENIOT is a simple tool that can be used by anyone, but in our mind we had IoT security professionals as main users when we designed it. Therefore, we allow competent users to extend our tool by implementing and adding their own protocols or attacks. Final capability of PENIOT is report generation, after each attack our tool gives the result report in a PDF file.

2. Test Plan

In this section, the testing procedure of PENIOT will be described in a detailed way. For the first six months, PENIOT did not have complete structure that specifies relations between components. We had implemented simple attack scripts for selected protocols. Proceeding this way, we implemented MQTT, CoAP and AMQP protocols. For this part, we generally followed ad-hoc testing. After implementing a security attack for any protocol, we defined simple client and server architecture for that protocol and we performed our attacks on healthy communications to see whether we can expose any component of the protocol with the implemented attacks. We ensured that our attacks performed well on stable communications and worked as expected.

When it comes to implementation of graphical user interface, we needed to define components in the tool so that there could be a reasonable/common interface between different components of PENIOT. Then, we created the following skeleton for PENIOT. This structure will enable us to implement further functionalities in a proper and easy way. There will be predefined interface between components and they will communicate with specified rules.

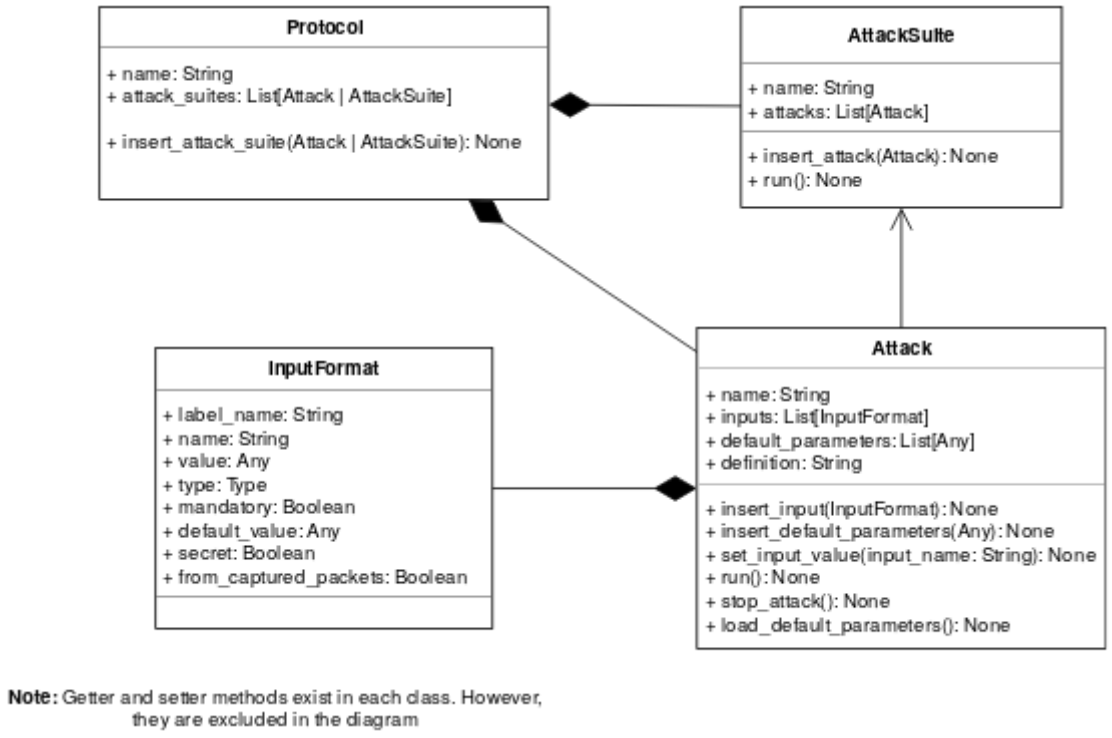

Figure 1: PENIOTt Class Diagram

After having the above structure, easy implementation of unit tests became possible since there are predefined methods to test on each basic component. At the beginning of second semester, we started to implement unit tests for each previously added protocols and their attacks. First, we converted our script-like structure to new class based structures with entities such as *Protocol*, *Attack*, *AttackSuite* and *InputFormat*, then we added protocol

classes for MQTT, AMQP and CoAP. As the project structure, we added all unit test of protocol into their class implementation file after class definition. For example, we test proper initialization of protocols, their names and attack lists. Then, we converted each attack script into attack classes. As you can see from class diagram, attacks have basic overridable methods like *run* and also default implemented other method such as *load_default_parameters*. For each attack class of each protocol entity, we added corresponding unit tests for those attacks. For example, we test initial input list definitions and other tests which are specific for the attacks. To give another example, we added unit test for fuzzing turn count of fuzzing attack of AMQP which checks how many times the fuzzer sent newly generated malicious fuzzing message to target device or behaviour of given seed text to fuzzer. Additionally, we added common unit test like edge cases of non-initialized input. We checked each attack and their behaviour running with non-initialized or unexpected input to get counter measures. Other than these, we added attack performer test for each attack. Generally, they aim our previously implemented healthy communication examples. For another example, we defined MQTT network to check whether our attacks perform correctly. But, we need runner script specific to attacks. Each time we wanted to test any attack, we wrote and deleted runner scripts. After implementation of unit tests, we added those scripts as another unit test to testing module so that we can directly run them. These are the unit testing process of PENIOT. In short, we added entity based structure to our tool and then we tested these entities when they are implemented as protocols or attacks.

While passing to implementation of graphical user interface, we started to do unit tests. When graphical user interface gets sufficient functionalities like home page, menus or other components, we had just completed respective unit tests. Then, the order came to the integration of internally implemented components to graphical embodied frames. For integration part, we added interface and integration testing modules. For each entity, we checked simple communication ways between those entities and graphical user interface. When we ensure that we added necessary tests that could be done at that point, we passed user testing phase since one way for testing of graphical components is trying to figure flaws out via using the tool. We tried lots of tests while adding new features to our tool. We also tried to do exceptional cases like giving invalid path for saving our reports or trying to continue attacks with invalid input value.

Also, we made user tests for overall system every week by trying every possible functionality implemented so far to see if there are any errors. This helped us to find many errors in the beginning so we could correct them early before they could create further problems.

These were the main testing procedure that we followed while implementing our tool. Also, there are other components which are mostly dependent on external libraries in PENIOT. Since they are pretty dependent to external libraries, we performed user testing on these modules as the followings,

- Import/Export Module: We tried exporting internal entity templates from PENIOT in different formats like tar.gz or zip. Also, we tried importing protocols or attack which are implemented by user with different file hierarchy or different compression

formats. For extending PENIOT, our tool should be able to export main class structure to the user (so the user can write code that is in harmony the general PENIOT class structure) and then it should be able to import (and incorporate) user made code to its source code. After implementing the import/export functionality, we created units tests to make sure that attacks are performed correctly.

- Capture Packet Module: We tried to save captured packets with invalid target location or file names.
- Reporting Module: We tested different protocol with various attacks with different lines of reports. Also, we visually test report template.

3. Feature Implementations

In this section, we list the master features of our project and explain how each of these features are implemented. Below is a list of our master features. The ones implemented in PENIOT's final form are starting with →, the last four items (starting with ●) are not included in PENIOT's final form, reason is explained below their respective headers.

In the graphical user interface of the product, we display all available attacks for the selected protocol. After choosing the attacks, the user should provide the inputs which are necessary to perform the attack. This is how some of the master features (attack implementations) are integrated to the end-product.

→ **Sniffing AMQP Message Traffic:** We sniff AMQP communication and save the captured packets. We incorporated this attack to the GUI so the user can easily run it.
→ **AMQP Attacks (at least 1):** We implemented DoS Attack, Random Payload Fuzzing Attack and Payload Size Fuzzer Attack. Definitions of these attacks are explained in the MQTT attacks part.
→ **Sniffing MQTT Message Traffic:** We sniff MQTT communication and save the captured packets. We incorporated this attack to the GUI so the user can easily run it.
→ **MQTT attacks (at least 1):** DoS Attack (we send a large number of packets to clog valid MQTT communication) , Replay Attack (we resend a captured, valid MQTT packets), Topic Name Fuzzing Attack (there are certain rules that MQTT messages should follow, we try to breach these rules in this attack), Generation Based Fuzzing Attack (in this attack we send deliberately corrupted MQTT packets), Payload Size Fuzzer Attack (in this attack we send packets that are larger than MQTT central broker is prepared to handle) and Random Payload Fuzzing attack (in this attack we send randomly generated inputs in valid MQTT packets) are the other attacks we implemented for MQTT protocol.

We implemented each of these and incorporated them to our GUI so that the user can easily call them.
→ **Sniffing CoAP Message Traffic:** We sniff CoAP communication and save the captured packets. We incorporated this attack to the GUI so the user can easily run it.

→ **CoAP Attacks (at least 1):** DoS Attack, Replay Attack, Random Payload Fuzzing Attack and Payload Size Fuzzer Attack are the other attacks we implemented for CoAP protocol. Definitions of these attacks are explained in the MQTT attacks part.

We implemented each of these and incorporated them to our GUI so that the user can easily call them.

→ **Sniffing BLE Message Traffic:** We sniff the BLE communication by using some additional hardware such as Adafruit BLE Sniffer. We save the captured packets. We incorporated this attack to the GUI so the user can easily call it.

→ **BLE Attacks (at least 1):** Apart from sniff attack, we have Replay Attack. We simply resend the captured packets. These packets may include passwords so they may lead to some serious security breaches. This attack uses an additional hardware called BLE dongle. We incorporated this attack to the GUI so the user can easily call it.

→ **Easy-to-use Interface:** Alongside with developing attacks we created a GUI with Python's Tkinter package. This GUI interacts with the user and then runs the source code below in accordance with user's demands.

Our GUI allows user to choose the protocol and attack they want to work on. Then before implementing the selected attack, it asks for required inputs. PENIOT's GUI checks if the compulsory inputs are supplied (compulsory inputs can change depending on the selected protocol and attack) and makes input validation on these inputs. After all inputs are correctly entered, user interface runs the necessary Python module to implement the respective attack.

→ **Report Generation with respect to Test Cases:** We display the results of test cases and enable the users to save the results as a pdf file.

→ **External Module Integration Capability:** As told before, we allow users to extend PENIOT by adding their own protocols or attacks. However, our protocols and attacks follow a well defined class structure and newly added protocols or attacks should conform to these to function. Therefore, as the first step of the extension protocol, the general class structure of the PENIOT is exported as a set of files so that the user can write their implementation in harmony with them. After completing their code, the user can export their code files to the PENIOT. This happens by pressing the respective button in the GUI and uploading their code files

● **Sniffing Zigbee Message Traffic:** We had to drop this feature because it requires some specialized hardware to do it. We could not find cheap versions of these devices. Given we were running on a tight budget, we thought it would be safer to ask department to buy BLE devices since BLE is a more common protocol.

● **Zigbee Attacks:** We had to drop this feature for the same reason above.

● **Sniffing RPL Message Traffic:** Given time restrictions, with our advisor we had agreed to implement four different protocols. Although initially we had added RPL protocol in our master features list with time we saw that there are more common protocols (like CoAP), so we decided to add one of them rather than implementing RPL.

● **RPL Attacks:** We had to drop this feature for the same reason above.