Architecture Specification

– IoT Test Bundle

Written by Hagai Solodar ( [hagaisolo@gmail.com](mailto:hagaisolo@gmail.com) ) - Final project

Revision 01 – 17.2.2016 – initial Design

Revision 02 – 1.5.2016 – Final draft

Revision 03 – 22.5.2016 – edit PM section

Revision 04 – 22.5.2016 – edit TPB section

# Introduction

In future times, not so far ahead, as predicted by leading cooperation and personas all over the world IoT is going to be the next technological revolution. IoT, Internet Of Things, is a general name for creating smart, cloud connected objects, “things”. It can relate to a fridge, air conditioner or a washing machines which is control via controller which is connected to the cloud and enable the appliances owner to control it from anywhere in the world, plus smart application can autonomously manages objects for you. As oppose to the simple but not so exciting example given here IoT actually relate to much more impacting things in our personal and social life. It can relate to smart autonomous cars, healthcare devices such as pace makers and imagine a watch which can detect strokes and call medical aid.

IoT is a very wide, distributed and decentralized field and as such it is developed by various companies using different protocols and routines. In a rapidly advancing field the need to quickly release new products forces teams skip deeply and sufficient security checks. It can already be seen in products available in the market that simple well know vulnerabilities in cyber security were not addressed and covered in production. Moreover small and start-up companies sometimes lack the funds and even the understanding for the need to build a secure system.

The IoT test bundle is a Python application meant to help IoT developers check their products for security vulnerabilities in an affordable and obtainable way. The application design is to be invariant to IoT physical design and architecture and focuses on the communication aspect of the product. The bundle contains tests for various known vulnerabilities and it is designed to be extendable.

Python is the chosen programming language for its ease of use and massive modules (libraries) support. The application will import the required Python modules as a part of its initialization. By using Python we keep the code small, readable and easy to manipulate if necessary. We also open a gate way to the Python community which poses and support many features we can use.

# General Architecture – User’s Point of View (Frontend)

The application is designed to help ventures to check their products for security faults and vulnerabilities. Therefore it is designed from a user perspective. Different products will require different tests and tests configuration, the approach taken here is to build each test to be able to act differently according to input parameters, in that way it can test various products and architectures. Nevertheless some tests are irrelevant for some products. As an example you will not check SSH vulnerabilities for a device which does not possess SSH capabilities.

Therefore we require some input from the user about the product being tested to feed the correct parameters to the tests and decide which tests to run. From a user perspective the application is a simple GUI (command line) that navigates between menus, ask for inputs, manage and run tests and produces outputs, hopefully automatically meaningful suggestion about architecture vulnerabilities.

For detailed design I keep in mind the user experience, the setup in which the user will run the application. It is designed for direct connection to the Thing being tested and via cloud connection. By Direct connection I refer to both the device running the application and the Thing present on the same LAN. The LAN is managed possibly by one of them or external device. The application is able to run tests regarding specific LAN implantation as Ethernet or Wi-fi, in this project I will not focus on wireless vulnerabilities. The actual setup for user and for running experiments of the application is an Ethernet link between a computer running the application and the device being tested, Raspberry Pi as an example.

Tests regarding cloud connectivity are tests which the application’s host device can be located anywhere not necessary in the same LAN as the Thing being tested as long as they are both connected to the cloud (The Internet), in this setup we can also check architectures in which the devices act as the gateway between multiple objects and the web and it act as a sort of router. Relevant tools and vulnerabilities regarding this setup are NMAP and its Python implementation, ports scanning, common mistakes in port managing and such. Most of LAN tests planed also to run on this setup but the application is conclusively not designed to check security issues regarding router and other common LAN manager devices.

In order to keep on a modular form, distinguishing between the two setups and any other difference will be inserted as a parameter from the user. On the backend the application can save the tests in an ordered way and use different algorithm to choose which tests to use and what parameter to insert for each test. It is desired to form some sort of algorithm that interact automatically with the tested device and extract more parameters, some of are not expected of the user to know.

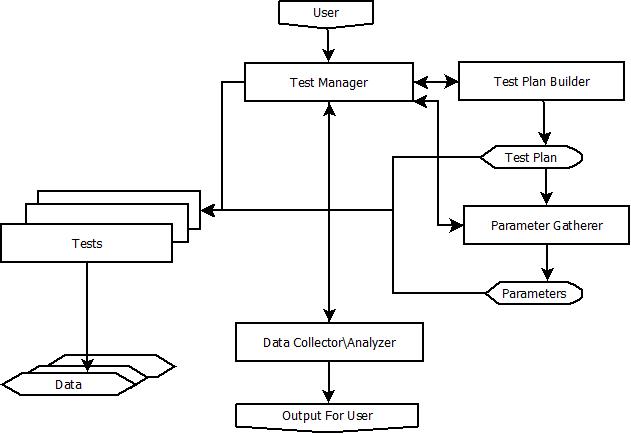


Figure 1: Frontend interaction flow

# General Architecture – Internal Structure (Backend)

In contract to the Frontend architecture which is designed to answer on user perspective demands the Backend architecture focuses on different aspects. First we design the application in a way it can expend easily as new test and feature are being created. In order to achieve this we create an independent unit which we will call a group, a group contains one or more tests regarding a certain subject, each group is independent from other group and contain all the files it need to run. Tests inside a group can interact and call other tests but cannot interact with test from another group in order to keep the independency criterion.

Second, the application is meant to serve as an open source application working under some or another open source license. It is intended to be developed further more by open source communities, therefore the tree structure of the application, its files and directories order must be simple, clear and easy to integrate new files into. Moreover the application development will be managed in GIT for documentation, releases and other key feature in development of an ongoing development effort.

The application is built from three main parts: Core, Groups, UI. The UI (user interference) interact with the user, it can be further developed to a GUI. The core is the framework, it contain the three main features controlling the tests, the test manager, the test plan builder and the data collector. The groups are the test themselves, they will be developed according to minimal criterion specify here and integrated simply by adding the directory to the groups directory.

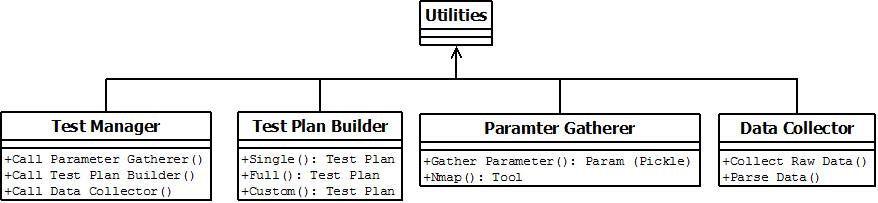


Figure 2: Backend tools architecture

# PM – Parameter Gatherer

## Introduction

The PM is a SW feature responsible for getting the required parameter to feed the tests and the TPB. It is responsible for generating demands to the user and to gather the users input. Furthermore it collects data autonomously, both routes occur simultaneously and initiate further data collection on each other by asking the user for more input after some data gathered and gather more data according to user input.

## Parameter Structure

Each group will contain a data.xml file which specifies among other things the required parameter for this group. The file has to be in the following structure for the PM the parse it:

**<data>**

**<parameters number="4">**

**<ip type="int" question="What is the device IP addess?"/>**

**<port type="int" question="Enter Port"/>**

**<packet\_size type="int" question="Please specify packet size."/>**

**<random\_size type="bool" question="enable random packet creation? "/>**

**</parameters>**

**<testplan>**

**</testplan>**

**<tools number="0"**

**</tools>**

**</data>**

To avoid duplication between groups (i.e. Port, PORT, port or IP ip i.p etc) we name parameters only with lower case letters and an underscore ( \_ ) if multiple words required. Example: for layer 4 protocol one can name it **layer\_4\_protocol**

## Automatic parameter gather

To be planned

## Parameters Synchronization between Groups and PM

Each time we run test the PM collects the parameter demand from all group and start collecting them both by asking the user and autonomously. To avoid collecting common parameter for multiple groups the PM first gathers all parameter demand, then it remove any duplications and then it start collection the parameters.

New (1.5.2016) – **add scripts to make all lower case letter first then all spaces underscore**

It may be good to add some feature to synchronize same parameter with different name such as “Port”, “PORT” and “port”. We do not desire to ask the user three times for port in such case.

The solution can come it the form of a SW feature or regulation on parameter names.

# Test Plan Builder – TPB

## Introduction

The TPB is responsible for building a test plan. It is called from the main page and building with the user a test plan. When building a test plan it verifies that the group requirements are satisfied.

## Supported Capabilities

The TPB is currently supported for various capabilities:

1. Single – creates a test plan of a single group.
2. Full – creates a test plan including all groups.
3. Custom – creates a custom test plan.
4. List – lists all available groups and tests
5. Help – print to terminal the help menu
6. Quit – exists the TPB menu back to main
7. Smart – a smart test plan builder

### Flow of Single Test Plan Creation

Demonstration for single test:

User Enters **TPB** and chooses **Single** 🡺

User is asked to type a group name 🡺

A **TestPlan** object is instantiate listing this group 🡺

The object is returned to the **Test Manager** via main.

### 5.2.2. Flow of Full Test Plan Creation

Pretty much the same as single, they are both predefined “constant” test plans.

### 5.2.3. Flow of Custom Test Plan Creation

The user type the groups it wants to run with one space between the group, etc:

To run the group Ping and DDOS the user type:

***Ping DDOS***

## Smart Test Plan

From the characterization of IoT device we derive a method to create test plans.

Each group will contain in its data.xml file a list of flags. In each layer it will raise the flag it supports (nothing means “all”), building a smart test plan is according to defining a device in this 6 aspects.

A group that support several devices but not all will posses several flags, it can also posses negative flags (maybe, could be hard to implement) or if it is invariant for difference in this layer in hold the “all” flag.

When using smart test plan building the application will collect all available flags from al groups, ask user about them and generate test plan accordingly.

### Classification method:

We describe a device in 6 layers (**TBD:** find another name to avoid confusion with OSI model).

**Layer 1**

The device itself, parameters regarding the device such as: hardware (raspberry pi\Arduino\esp8266\else) firmware (don’t have examples) and O.S. (the Linux image\windows\could have non)

**Layer 2**

Setup, way of connecting to the cloud, basically if it resides in some network, detail about the network, connected to the cloud via router or directly via 4G 5G etc.

**Layer 3**

Low level communication protocols (OSI layer 2-4) such as IP, TCP\UDP and maybe some cellular protocols such as RNS.

**Layer 4**

High level communication protocols, HTTP, MQTT, CoAP, AMQP, DDS, etc.

**Layer 5**

Representation, date representation and encryption, SSH

### Mechanism

For building the smart test plan the TPB first collect all available flags from all groups without the values. Then the user characterization phase start and the TPB asks the user to define the device layer by layer, filling the values. Then it checks each group if it is valid for this specific device classification and if it is it adds the group to the test plan.

(TPD – one can add an automatic mechanism that check automatically certain definition as response to different protocols or find which setup.)

### Groups data

Each group in its data.xml file will have some flags for the Smart test plan, example:

**<data>**

**<parameters number="2">**

**</parameters>**

**<testplan>**

**<layer1 layer="1"/>**

**<layer2 layer="2">**

**<char type="setup">router</char>**

**</layer2>**

**<layer3 layer="3">**

**<char type="protocol">tcp</char>**

**<char type="protocol">udp</char>**

**</layer3>**

**<layer4 layer="4"/>**

**<layer5 layer="5"/>**

**<layer6 layer="6"/>**

**</testplan>**

**<tools number="0">**

**</tools>**

**</data>**

For each layer we give an attribute **layer** with the layer’s number as value (**layer=”3”)**. Each layer can host several characterizations, for each one the **type** attribute describe what it is and the value is the characterization (**type=”protocol”**), then the **text** inside is the classification.

(**<char type=”protocol”>tcp</char>**) means there is a flag for protocol with tcp value, the group will only run if tcp (or all\non) is filled by the user at the user characterization phase. Note that at this example the group will run with both **tcp** and **udp**.

## Test Plan Class

The TestPlan class instantiations are test plan returned from the TPB to the Test Manager (TM – chapter 6). It should specify in an exact way the tests that the TM needs to run.

It currently have get\_list, set\_list and \_\_init\_\_ with list methods.

## Tool Presence Check (TPC)

Each group will contain a file named data.xml, in this file there is a field which contains a list of the tools this group expects to find. The group relies on the existence of the tools for it to run. In case there is a mismatch between tools demand and the existing tools the group will not initialized. Some example tools are NMAP, WIRESHARK and such which we do not want to be a part of the group and then to be duplicated for each group.

The TPB verifies the tool requirements existence then the tests inside the group just use the tools with varying API’s.

The “tool” name should be the **module** the group tries to import for this example:

**#!usr/bin/python**

**Import nmap**

The tool name should be **nmap**, **data.xml:**

<data>

.

.

.

<tools number="1">

<nmap/>

</tools>

</data>

# Test Manager – TM

## Introduction

The Test Manager is responsible for running the tests. It receives a test plan from the **TPB** (Test Plan Builder) and then it calls the **PG** (Parameter Gatherer) to gather the needed parameters. After it start running the test by some not yet determined protocol and calls another not yet exist feature to collect and display the output data from the tests.

It receives a list of group to execute and returns the tests outputs.

# Data Collector

# Introduction

The Data Collector navigate parse and display the output data from the tests. Currently I believe it is best that the test themselves will analyze the data and generate readable output file. The Data Collector (DC) tool only responsible to help the user navigate all the tests output and find what the user need easily.

# Group Structure

## Introduction

A Group is an independent set of one or more tests and more required content for the application to proper activate the tests. The group design was choose in order to keep the application modular and extendable. Following the independency criterion we are able to simply add, remove and update groups without further modification to the applications source code.

## Organization and obligatory content

The design of structure of the files and folders within the OS has to be simple modular and comply with several demands rising from Python architecture. Therefore within the main directory we build a general group’s directory with holds a directory for each group (example: /home/bundleApp/groups/brute\_force\_resistance\_tests/).

For proper integration of a group with the application a few files which specify the groups demand from the application must exist.

Each group must contain a python file with the group name to run from the TM since it run the group as follows:

Popen([executable, **'Groups//'**+item+**'//'** + item + **'.py'**], creationflags=CREATE\_NEW\_CONSOLE)

So for group called ping we have Groups/ping/ping.py file that will be run.

## Data.xml file

Each group contain a data.xml file which specifies the parameter it needs, device classification for smart test plan and tool requirement. Example:

*<?***xml version="1.0" encoding="UTF-8"***?>*<**data**>  
 <**parameters number="2"**>  
 <**ip type="int" question="What is the device IP addess?"**/>  
 <**port type="int" question="Enter Port"**/>  
 <**packet\_size type="int" question="Please specify packet size."**/>  
 <**random\_size type="bool" question="Enable random packet creation."**/>  
 </**parameters**>  
  
 <**testplan**>  
 <**layer1 layer="1"**/>  
 <**layer2 layer="2"**>  
 <**char type="setup"**>router</**char**>  
 </**layer2**>  
 <**layer3 layer="3"**>  
 <**char type="protocol"**>tcp</**char**>  
 <**char type="protocol"**>udp</**char**>  
 </**layer3**>  
 <**layer4 layer="4"**/>  
 <**layer5 layer="5"**/>  
 <**layer6 layer="6"**/>  
 </**testplan**>  
  
 <**tools number="0"**>  
 <**nmap**></**nmap**>  
 </**tools**>  
</**data**>

## Bringing parameter to group

The file that host the parameters is located at: ../testbundle/groups/parameters

And is created with “pickle”, since the group is called from the TM the proper relative location of the parameters file is:

param\_file = open (‘Groups//parameters’, ‘rb’)

(**Not** open(‘../parameters’, ‘rb’) which I use for debugging)

The Tools.core\_get\_parameters bring the parameter to the group, the group can use it, it is a part of a pack of utilities for the groups (Tools).