Smart Router report

Last build :May 25, 2018
Version 1.0

Nicodème Benjamin
benjamin.nicodeme@icloud.com

benjamin.nicodeme@ulb.ac.be

Contents

1.1 Introduction

The number of IoTs devices present in our houses is constantly growing up. Smart-TVs, smart-lights,IP-cameras and number others are all connected to the Internet to be managed from outside our homes. But how can we, in as SOHO context, monitor these devices to be sure they only do the work they should. So how to be sure they are not botnets ?

1.1.1 Goal

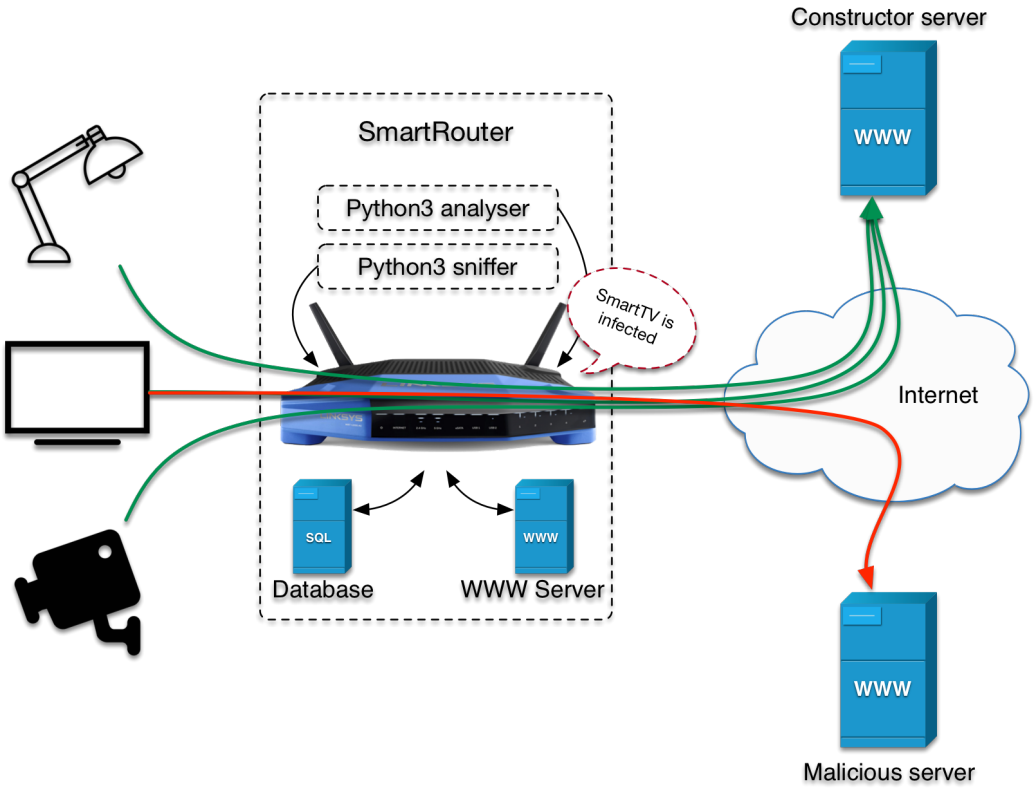
The goal of this project is to monitor IoT traffic of a SOHO (Small Office, Home Office) network. By sniffing and analyzing it, this project can send alerts when an IoT has an unusual network traffic. For example, if it suddenly begins to send random request to random IP addresses around the world.
Only IoT traffic is interesting for this type of project because Iot devices only go to the same servers. Devices like laptops are used for example to surf the web and then it is never the same destinations. So sniffing non-IoT traffic is not the goal of this project.

2.1 Architecture

The smart router is designed to be installed between the connected devices and the Internet, such that it can intercept and analyze the traffic generated by the devices. It is composed of 3 main modules:

- Traffic sniffer,
- Traffic analyzer,
- Web interface

This project was initially designed from Ubuntu machine but was then adapted to work on an OpenWRT distribution which is more adapted from network embedded devices. This project was developed in Python3.

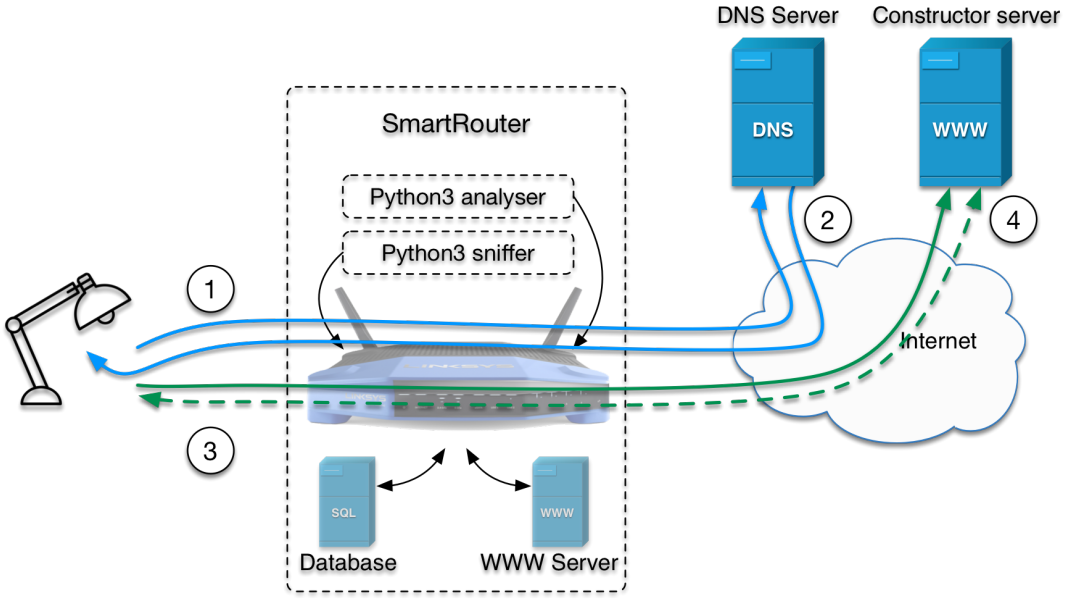


2.1.1 Traffic sniffer

The traffic sniffer is a process that captures all SYN packets with destination port TCP/80 && TCP/443. This allows to detect all connection tentatives from the devices on the local network. For each tentative, following fields are stored in the database:

- Source MAC address,
- Destination IP address,
- Timestamp of the request

To be able to know to which domain corresponds each destination IP, DNS packets are also sniffed and then data are correlated to make IP correspond to a domain. If no domain corresponds to an IP, IPs are reversed lookup to try to have a domain.
Schematically, the process is the following :



1. First, the IoT sent a QNS query to *constuctor.com* (assuming that this domain is the domain used for the IoT management, and which, in this case is legit).
2. Second, the DNS server will reply to the previous request with the corresponding IP address of the previously asked domain. This request is sniffed, the domain and corresponding IP address are stored in the database
3. Third, the IoT will contact the server behind the previously asked domain. Only the SYN packet will be sniffed and an entry will be created into the database,
4. Finally, the rest of the request (non only SYN packets) will not be stored into the database.

2.1.2 Traffic analyzer

A process is launched every X minutes (depending on the setup) to analyze all traffic and determine if the traffic is legit or malicious. This process is managed by a custom *init.d* script which is run at startup.

A "learning period" (which depend on the setup) is defined. Between the first request of an IoT and the end of the learning period, each packet and domains are considered as legit.

Beyond the period, if an IoT contact an unknown domain (which is a domain not contacted in the learning period), it will be considered like malicious. Is this case an alert will be generated and stored in the database.

Moreover, additional modules may be triggered when an alert occurs, like a Slack notification (see below), a notification on a dedicated app, a webhook etc.

2.1.3 Web interface

Actually the web interface displays only a txt file containing alerts.
In the next version of this project, a web interface could be used to configure the router and have a better view of the alerts.

3.1 Implementation

To test the detection system, we built a complete implementation, using OpenWRT, Python and Scapy.
The complete source code of the smart router is available at `https://github.com/RUCD/smart-router/`.

3.1.1 OpenWRT [1]

The OpenWrt Project is a Linux operating system targeting embedded devices. So it was one of the possible choices for the Linux distribution. More specifically, OpenWRT was a compatible distribution for the router used for this project. The routers used are Linksys WRT1200ac.

3.1.2 Scapy

We used python and scapy to implement the traffic sniffer and the traffic analyzer. *Scapy is a packet manipulation tool for computer networks, written in Python by Philippe Biondi. It can forge or decode packets, send them on the wire, capture them, and match requests and replies.*

3.1.3 Web server

Due to portability issues and high resource consumption, we implemented a very simple web interface on the OpenWRT router: alerts are simply read on a text file and displayed vie a simple web server.

4.1 Installation

The installation is very simple thanks to auto-deploy scripts. A Readme is present to the Github the guide the installation.
Like routers used for this project do not have a lot of storage, a USB stick is needed to expand data storage. All this procedure is also explained in the Github(`https://github.com/RUCD/smart-router`).
One of the auto-deploy script is the following :

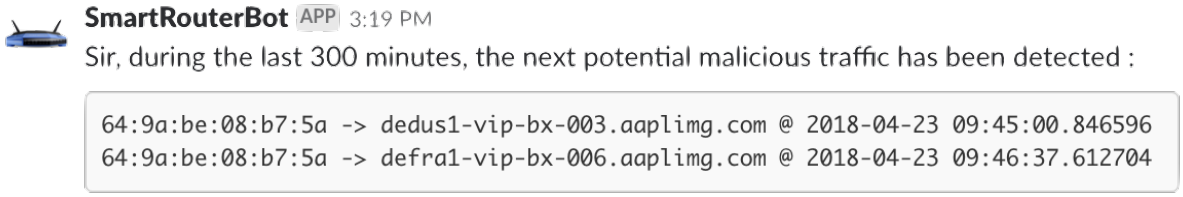
```
sh -c "$(curl -fsSL https://raw.githubusercontent.com/RUCD\
/smart-router/master/docs/setupScripts/setupSR.sh)"
```

5.1 Testing

Initially, two different types of alerting had been designed. On via a web browser, and another one, optional, like an application or something like that.
We tested the smart router in various real SoHo networks. To be able to easily collect the alerts produced by the different routers, we implemented a slack application. This is a simple slack integration working with tokens. All is implemented by slack.

[1] `https://openwrt.org`

Slack alert have the role of the initial application alert. A slack application has been integrated in a workspace. So with it, the Smart Router send alert to a specific channel, thanks to slack tokens.
This slack integration allow the receive notification like initially wanted, but without having to develop a specific application for it.



6.1 Future Work

Lots of modules can be developed in future work, for example we can imagine :

- More protocol sniffed : for now, only web and DNS traffic is sniffed, but it could be interesting to have a generic sniffer which allowed to sniff everything traffic,
- Web configurations : all configuration is done in command line, so a more user-friendly web interface configuration page could be a great idea,
- Target sniffing : a good idea will be to be able to detect IoTs and filter only this traffic. Actually, all traffic is sniffed, but some can be blacklisted not to be sniffed,
- Target analysis : actually, all traffic is reanalyzed every time an analysis is run. A better analysis will be to analyze only new traffic since the last analysis,
- Lots of other integration are possible to have better interfaces and configuration modules.

6.2 Actual code

The following section explains the project code structure.

6.2.1 python

alert.py
A simple class to format alerts.

analyser.py
All logic behind analyzing the previously sniffed data.

database.py
All logic to interact with the database (MySQL was at first used, then SQLite replaced it(the MySQL code has not been tested)).

dnsquery.py
A simple class to format DNS queries.

firstActivityFaker.py
A class to fake activity before the first detected activity (used for fill fake data for testing).

host.py
A simple class to format hosts.

httpquery.py
A simple class to format HTTP queries.

main.py
The main class, where sniffing and alerting threads are launched.

sniffer.py
All the logic concerning the sniffing process.

6.2.1.1 Tests
Tests can be found at directory 'python/tests'. The following tests class can be found :

- tu_allprocess.py: this class is used to test all process of the smart router (including sniffing and analyzing). Sniffing and analyzing begin a very coupled process, it can be interesting to comment 'testAnalyser' or 'testSniffer' if we want to test only one feature at a time.
- tu_basics.py, tu_db.py, tu_readdb.py: most of the code of these classes is about testing the database. Basic tests are done int 'tu_basics.py' while more specific tests are realized into 'tu_db.py'

6.2.2 php
All the php code (Laravel code) remains in th 'ui' directory. It was used during the development process. But when the code was adapted to fit in embedded OpenWRt device, this code was not used. So it was only used to display DB content at the time, and must certainly be updated considering actual DB tables if it must be reused in the future. If this code won't be used anymore, it must certainly be deleted.

6.2.2.1 Tests
No real tests have been implemented for this part of the code like it was only used in early development version of the code and like it was only used to display simple data.

6.2.3 Documentation
Documentation, including, schemas, scripts(setup scripts('docs/setupScripts')), OpenWRT images and readmes can be found at the 'docs' directory.
The certainly most useful .md file containing all info about how to install OpenWRT and the project on Linksys WRT1200 AC can be found at 'docs/FirstInstallSR.md'.

6.2.4 Database
The 'db' folder is only used for the SQLite file storage.

6.2.5 Vagrant
At the time of the development on Ubuntu, Vagrant was used for this project. So Vargant files can be found at the root folder of this project.

6.2.6 Travis
Travis was used during the development process. Due to a misconfiguration of Travis, the code is always failing. **Furthermore, Travis environment is maybe not really optimized to create traffic and sniff it.**

7.1 Conclusion
This project has a lot of other possibilities then experimented here. It also has a lot of optimization and machine learning possible.