WAeUP Identifier Documentation

Release 1.0.dev0

Uli Fouquet

Jul 16, 2017

Contents

CHAPTER 1

Version: 1.0.dev0

WAeUP Identifier is a software for scanning and verifying fingerprints stored on a WAeUP Kofa portal. It is designed to work as a graphical application even on Raspberry PIs.

Contents:

waeup.identifier

Identify WAeUP students biometrically. Scan/verify fingerprints of Kofa students. *waeup.identifier* uploads taken fingerprints and can verify student fingerprints after that. Kofa is an open source student management portal from WAeUP .

Full documentation is available at

https://waeupidentifier.readthedocs.io

Requirements

waeup.identifier runs on Python 3.4.

You need Kivy and fpscan installed.

There are many ways to install *kivy* on your system. Please see the Kivy homepage for a detailed discussion.

waeup.identifier is designed to be run on a RaspberryPI 3, but also runs on ordinary laptops and desktop computers.

While a fingerprint scanner device is not strictly neccessary to run *waeup.identifier*, it also makes not much sense to run it without.

Note: To ease install on RaspberryPI we provide a bunch of playbooks for use with ansible. The ansible playbooks can also be used for provisioning other machines. See the `ansible/` subdir of *waeup.identifier* for details.

User Install

For people that only want to simply use the software:

```
$ sudo pip install waeup.identifier
```

Afterwards the commandline tool (*waeup_identifier*) should be available.

Developer Install

It is recommended to setup sources in a virtual environment:

```
$ virtualenv py34 -p python3.4    # only Python 3.4 is supported currently
$ source py34/bin/activate
(py34) $
```

Then, install *Cython* and Kivy (in that order):

```
(py34) $ pip install Cython
(py34) $ pip install kivy
```

Please note that especially Kivy requires a lot of special libraries to run.

Get the sources:

```
(py34) $ git clone https://github.com/WAeUP/waeup.identifier.git
(py34) $ cd waeup.identifier
```

Install packages for testing/developing:

```
(py34) $ python setup.py dev
```

This will also install the `waeup_identifier` script in your virtual environment `bin/` dir (and a *fake_kofa_server* script, useful for testing).

Running tests:

```
(py34) $ py.test
```

We also support *tox* to run tests for all supported Python versions:

```
(py34) $ pip install tox
(py34) $ tox
```

Of course you must have the respective Python versions installed (currently 3.4 only).

Running the test coverage detector:

```
(py34) $ KIVY_NO_ARGS=1 py.test --cov=waeup.identifier   # for cmdline results
(py34) $ KIVY_NO_ARGS=1 py.test --cov=waeup.identifier --cov-report=html
```

The latter will generate HTML coverage reports in a subdirectory.

The env var setting (*KIVY_NO_ARGS*) at the beginning keeps *kivy* from parsing command line arguments.

Alternatively, you may want to run *coverage* directly. First gather data:

```
(py34) $ coverage run ./py34/bin/py.test
```

Then evaluate the data gathered:

```
(py34) $ coverage report --include='waeup/*,tests/*,setup.py' -m
```

Docs Install

To install/generate the documentation locally, you first have to install the needed tools:

```
(py34) $ python setup.py docs
(py34) $ cd doc
(py34) $ make html
```

Will generate the documentation in a subdirectory.

Misc

There is a fake Kofa XMLRPC server included for use in tests. The server tries to mimic WAeUP Kofa XMLRPC API while being much more lightweight. It can be started using:

```
(py34) $ fake_kofa_server
Starting server at 127.0.0.1:61616
No entries created. Restart with `-- -p' to create.
Press ^C (Ctrl-c) to abort.
```

and will listen for XMLRPC requests on localhost port 61616. It requires basic authentication with `mgr` as username and `mgrpw` as password.

You can also start the server with a prepopulated db (which will vanish when the server stops) like this:

```
(py34) $ fake_kofa_server -- -p
Starting server at 127.0.0.1:61616
Created fake entry: AA11111
Created fake entry: BB11111
Press ^C (Ctrl-c) to abort.
```

Programmatically, the fake kofa server can be started like this:

```
>>> import threading
>>> from waeup.identifier.testing import AuthenticatingXMLRPCServer
>>> server = AuthenticatingXMLRPCServer('127.0.0.1', 16161)
>>> server_thread = threading.Thread(
...     target=server.serve_forever
...     )
>>> server_thread.daemon = True
>>> server_thread.start()
```

When the server runs, you can try to connect to it via *xmlrpclib* (Python 2.x) or *xmlrpc.client* (Python 3.x). Please note, that the *fake_kofa_server* by default listens on localhost port 616161.

```
>>> from xmlrpc.client import ServerProxy  # Python 3.x only
>>> s = ServerProxy("http://mgr:mgrpw@localhost:16161")
>>> s.ping(42)
['pong', 42]
```

See WAeUP Kofa docs or local webservice tests for method details.

```
>>> server.shutdown()
```

Installation

waeup.identifier is based on kivy. Installing kivy, requires several libs to be installed. For convenience we created suitable ansible playbooks that can (at least on Debian-based systems, including raspbian and Ubuntu) install all required stuff at once.

For development we recommend to run the ansible driven variant on localhost as described below.

You can of course install all parts manually.

Installing and Provisioning with ansible

We provide some ansible playbooks to ease installation and provisioning on RaspberryPI (raspbian) or other Debian-based distros.

The RaspberryPI is assumed to have the default 7" touch screen attached.

All ansible-related stuff can be found in the projects `ansible/` subdir. In the following we show how to provision a running RaspberryPI connected to the same network as your local machine.

Overview

If you want to provision a RaspberryPI device, start with the *setup_raspi_playbook.yml* playbook. This will prepare fresh Raspbian installs for upcoming unstalls and do some safety measures.

Afterwards, continue with the other playbooks.

Get ansible

On Ubuntu/Debian:

```
$ sudo apt-get install ansible
```

Please make sure, your *ansible* is not too old. Version 2.x+ would be nice:

```
$ ansible --version
ansible 2.1.1.0
```

If it is too old, Ubuntu users can get a more recent version from the *ansible/ansible* PPA. It can be activated with *add-apt-repository* command.[1]

```
$ sudo apt-get install software-properties-common  #  Ubuntu 14.04+
$ sudo add-apt-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

[1] On Ubuntu 12.04 you have to install *python-software-properties* instead of *software-properties-common*

Check SSH Connectivity

First, make sure you can SSH into your RaspberryPI as user `pi`[2]

```
$ ssh -l pi 192.168.45.244
```

(but with the real IP of your RaspberryPI).

Then, unless done already, change the password of the *pi* user:

```
$ passwd
```

Do not skip these steps as they prepare SSH to flawlessly connect to your device (storing the host id, etc.).

Check Basic Ansible Connectivity

Then see, if ansible can connect as user `pi` to your RaspberryPI:

```
$ ansible -i 192.168.32.86, all -k -u pi -m setup
```

Please note the trailing comma after the IP number.

Again, use the real IP of your RaspberryPI instead of `192.168.32.86` (don't forget the trailung comma). You will be asked for the SSH password and replace `pi` with your real user if you do not use the default.

If you have an *hosts* file with appropriate settings (IP and username) and configured passwordless login on your Raspberry PI, you can instead do:

```
$ ansible -i hosts all -m setup
```

These should list plenty of infos about your raspberry in green color. Red means: something went wrong.

The *hosts* file is an ansible inventory file. The IP set in it must match the real IP of your local Raspberry PI.

It also sets a username to connect to the device. If you use another one, replace *pi* with the username you really use.

Initial Provisioning of a RaspberryPI

Preparation:

Your RaspberryPI should be

- up and running
- have an internet connection
- have a changed password or accept one of your local SSH keys

You should have run the steps above at least once.

Then carefully adjust the variables set in the *setup_raspi_playbook.yml* header using your editor. Currently there are four of them:

- *repo_path*: leave it untouched
- ***device_id*: if you are about to provision multiple devices, you can give** each one a different name. This makes it easier to distinguish their configs later on. The name should be one string without whitespaces and made of ASCII chars only.

[2] `pi` is the default user in Raspbian. If you created a different user to connect to your Raspberry PI, you should of course use that.

- ***reverse_ssh_host*: ip number or hostname of a machine for reverse SSH.** This is the machine to which we want to connect to and create a tunnel at to our device. Ignore, if you do not understand what this is for.
- ***reverse_ssh_port*: the port on the remote host, we will use for reverse** ssh. We will use this port to connect back to the device, once the reverse tunnel was started. Again, ignore, if you don't know what all that means.
- ***new_password*: the password to set for the connecting ssh user. The** default password we set is `fading-remedy-pony`. It is not safe to leave this password unchanged.
- *set_password*: set to `false` if you don't want to set the password.
- ***remove_bloatware*: set to `false` if you want to keep large** softwarepackages like *mathematica*, *wolfram-engine*, or *libreoffice*. We remove these otherwise, if they should be installed.

Then you can run the local ansible setup playbook like this:

```
$ ansible-playbook -i 192.168.122.12, -u pi -k setup_raspi_playbook.yml
```

Please note the trailing comma after the IP number.

The username `pi` and the IP number must, of coursei, be set to match your local settings.

The playbook will ask for the SSH password of the user set with `-u` (default: `raspberry`).

This playbook will also clone the *waeup.identifier* repository.

Note: The raspberry-pi setup will take *huge* amounts of time for updating, depending on your internet connection and SD-card quality/speed.

If you ssh into your raspi device once and run:

```
$ sudo apt-get update
$ sudo aptitude safe-upgrade
```

you will be able to track changes and can check whether everything is still working. A later *ansible* run will be much shorter then.

Remote Maintenance (optional)

Once the fingerprinting device is up and running, we might want to get access to it from remote locations. For this purpose we provide some sort of "poor mans Teamviewer" deploying reverse SSH tunnels.

It roughly works like this: the RaspberryPI connects via SSH to a remote server, laying a reverse tunnel open on that remote machine. As long as the tunnel exists, any admin on the remote server then can connect to your device w/o the need to know about the IP address, just as if the remote machine would be a local one.

How does the setup for this type of remote maintenance work?

Remote Maintenance Client Setup (optional)

If you want to prepare your freshly provisioned RaspberryPI for remote maintenance, it is sufficient to run the *setup_raspi_playbook.yml* playbook. It prepares your device to create a reverse ssh tunnel to a remote server and also runs the *setup_ssh_playbook.yml* automatically to harden the SSH server config on your device.

The raspi setup will also create a local EC25519 SSH key for logging into the maintenance machine (and starting a reverse ssh tunnel).

Note: The public key will be copied to the local *keys* directory (`.../.ssh/id_ed25519.pub`) and must be copied to the maintenance servers `authorized_keys` file manually.

Remote Maintenance Server Setup (optional)

The server setup makes no sense without one or more RaspberryPI devices to maintain over the server.

The remote box has to be prepared as well for the new client. Therefore, on the remote box, we normally allow only creation of an SSH reverse tunnel back to the RaspberryPI device. This poor-mans' teamviewer allows us to log into the RaspberryPI from some central machine if only the device has an internet connection.

The remote machine (not the RaspberryPI) can be provisioned for this purpose with the *setup_maintbox_playbook.yml*:

```
$ ansible-playbook -i <REMOTE-BOX-IP>, -u <REMOTE-USER> -k -K setup_maintbox_playbook.
↪yml
```

Here we have to provide an SSH password (`-k`) and a sudo password (`-K`). Leave these options out, if you have other authentication methods activated on your remote server.

The playbook will create a user *reverse* that is only allowed to connect to create a reverse SSH tunnel back to itself. Issuing any commands is forbidden.

Inside the new users home we create a *.ssh* directory that contains an empty *authorized_keys* file. The permissions of these files are set properly to ease later addition of real keys.

On the server we need the *public* key of the RaspberryPI to be added to the *reverse* user's `authorized_keys` file. For each device provisioned above you can find a public in the local *keys* directory. It is named `id_ed25519.pub` and its content has to be added to the *authorized_keys* file on the server. That could rougly be done like this:

```
(laptop) $ scp keys/<PATH-TO>/id_ed25119.pub remote_user@server:/home/remote_user
(laptop) $ ssh remote_user@server
(server) $ sudo bash -c "cat id_ed25519.pub >> /home/reverse/.ssh/authorized_keys"
```

Install fpscan

Preparation:

- Ansible must be installed locally
- The target system should be reachable via ansible (see above)

The fpscan commandline utility is a little C program for creating fingerprint scans. *waeup.identifier* deploys it to do the actual scans.

Because fpscan is available as source code only, the `install_fpscan_playbook.yml` creates a local build dir in the SSH users home, then builds and installs fpscan.

Install kivy

Preparation:

- Ansible must be installed locally
- The target system should be reachable via ansible (see above)

This playbook installs kivy in a virtualenv on the target machine.

The virtualenv will be created by ansible and is by default located in the remote user's home dir. It can be set via the playbook var `venv_path`.

Local dev environment

To install kivy in a local virtualenv on the local host, run the respective ansible playbook like this:

```
$ ansible-playbook -i "localhost," -c local -K ansible/install_kivy_playbook.yml
```

This will ask for a SUDO password (`-K`) and install kivy in a local virtualenv in `/home/<USERNAME>/venv34/`.

If you want to install in a custom dir on localhost, do:

```
$ ansible-playbook -i "localhost," -c local -e "venv_path=`pwd`/venv34" -K ansible/
↪install_kivy_playbook.yml
```

I.e., set the *venv_path* variable to a path where you want to install everything.

Please note, that we use Python 3.4 for kivy install.