Prepared for:
Department of Homeland Security

Decider 1.0.0

Admin Guide

This document is a product of the Homeland Security Systems Engineering and Development Institute (HSSEDI™)

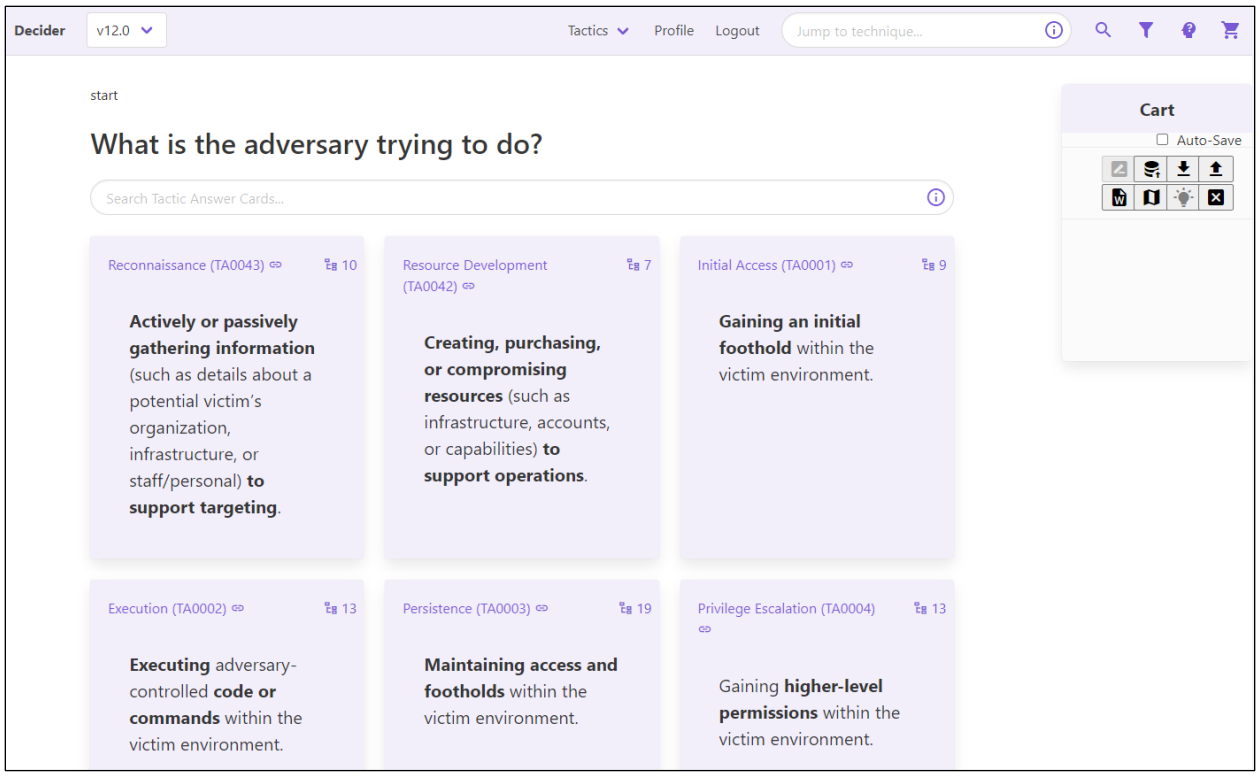
Contents

Introduction

Decider is a web application designed to provide threat analysts with rapid, guided assistance in mapping Tactics, Techniques, and Procedures (TTPs) / adversary behaviors to the MITRE ATT&CK® Enterprise Matrix. The tool features two workflows: an interactive decision tree and a feature-rich search. Users can save their mappings with notes to a "cart" and export the contents of the cart in multiple formats.

This project makes use of MITRE ATT&CK®

[ATT&CK Terms of Use]



Cart:

The cart is an important part of Decider – it is akin to a shopping cart in an online web store – it holds mapped Techniques / notes and persists as the user navigates between pages. The cart is saved locally to the user's browser but can also be saved to Decider's database. The cart allows exporting its contents as JSON (can be imported later, useful for external scripts as well), as an ATT&CK Navigator layer (allows for visualizing ATT&CK heatmaps in relation to defense coverage / the heatmaps of known adversaries), or as a Microsoft Word Document report (contains a table of mapped Techniques and the mapping rationale / notes).

Interactive Decision Tree:

Users click on "answer cards" which answer the question prompt regarding adversary behavior. This process navigates users through the decision tree until successfully creating a mapping to a specific ATT&CK technique or sub-technique.

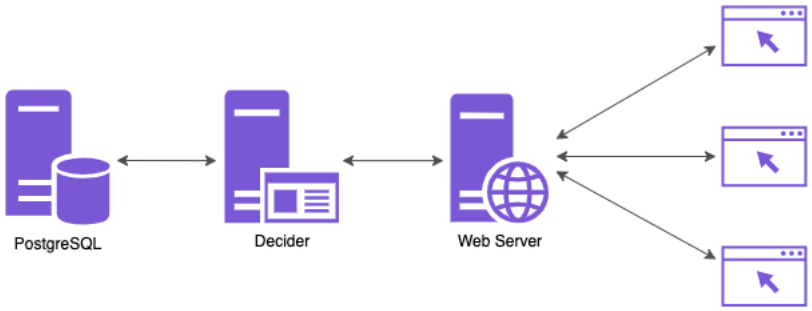
Feature-rich Search:

Users can use the comprehensive and fast search to query the ATT&CK corpus using simple keyword searches or advanced features such as logic operators and word stemming.

ATT&CK Compatibility & Hosting:

Decider ships with content for Enterprise ATT&CK (https://attack.mitre.org/matrices/enterprise/) v11.0 and v12.0. A version selector is present in-app that allows switching between installed versions. Availability of decision tree content and technique co-occurrence data for new releases is subject to the development timeline. Decider can be hosted locally for individual use or hosted in an enterprise environment and managed with the included administrator tools, login, and cart management system.

Architecture



Decider is composed of three components:

- Web Application (Python-Flask, Jinja, JavaScript)
- Web Server (uWSGI)
- Database (Postgres 12)

Using a modern web browser, users send HTTP requests to the uWSGI web server. The web server forwards these requests to the Decider backend, which may query the database if the user is requesting / amending information. The web server forwards the application's response to the client. All of these components must be enabled (and able to communicate) for Decider to run.

Database

PostgreSQL version 12 is used for this application. The database requires an initialization process that creates a database user, the database, and a few extensions. This process is separate from the initial database build.

For loading data into PostgreSQL, there is a build script that pulls data from the ATT&CK Enterprise JSON files and converts them to table columns and rows. The decision tree question / answer data is stored in the Tactic table / Technique table rows (a Tactic's question will lead to all answers of the Techniques living under it). Mismappings, Co-Occurrences, ATT&CK Platforms, and various other pieces of data have their own tables.

Application

Decider is built atop Python/Flask/Jinja and JavaScript/jQuery – these are used for server-side and client-side rendering of webpages respectively. The Flask backend responds to HTTP requests with either JSON data or web pages. For a web page request, Flask will use the Jinja templating engine to populate HTML templates with the queried database information.

For larger pages or pages that require more user interactions, JavaScript/jQuery are used to avoid extended load times and unnecessary page refreshes. Jinja might still be used to render some basic parts of the initial page request, but after the page loads, JavaScript is run to request additional data to build the page. As Jinja is run on the server side, any changes a user makes would not be visible until after a reload, therefore JavaScript is necessary for providing uninterrupted updates.

Secrets

Permanent Secrets

Applying best practices for management of local credentials, Decider leverages the Python package *python-dotenv* which loads configuration information from a `.env` file and sets environment variables.

The `.env` file (to be placed in / generated into the root of the repository) holds 3 variables:

- `DB_USER_NAME`: The username for the DB account Decider uses to run queries.
- `DB_USER_PASS`: The password to go along with the aforementioned username.
- `CART_ENC_KEY`: A key used to encrypt carts that are saved on the DB.

(A template exists for this file, named "`.env.example`")

Note that "`.env`" files are instance-specific - they are intended to be created during deployment, for that environment only. They are not part of the "repository" of code and thus are not shared between environments. The Git utility reads the "`.gitignore`" file in the root of Decider and recognizes the entry for "`/.env`" and thus does not include these files when creating or updating repos.

Also note that other files within the codebase have a similar naming, i.e., "`.env.prod`". However, these files have a different purpose and do not contain sensitive data.

Temporary Secrets

There are two other "`.gitignore`"d files that are to be used temporarily during the setup stage:

- `init.sql`: This is run during install to create a database user, a database, and it also sets up necessary PostgreSQL extensions / functions / dictionary config. It holds the fields:
    - `DB_USER_NAME`: Used to create the DB account Decider uses to run queries.
    - `DB_USER_PASS`: ^-Same. As plaintext.

- `user.json`: This is read by `app.utils.db.actions.full_build` in order to create an admin Decider account that can be used once the setup / build is complete. It holds:
    - `APP_ADMIN_EMAIL`: An email for the initial admin.
    - `APP_ADMIN_PASS`: A hash of the initial admin's entered password.

Web Server

Decider uses uWSGI, which supports several methods of integrating with web servers. It is also capable of serving HTTP requests by itself. uWSGI is used to pass requests to Decider. Decider also uses Flask, which includes a simple web server but is not recommended for production.

uWSGI is a production ready server that is efficient, stable, and secure. It is highly configurable and scalable. Additionally, for use with Apache, there are currently three uwsgi-protocol related apache2 modules available.

Installation

Overview

1. Download the Decider repo
2. Create a service account
3. Install the python dependencies
4. Install and configure PostgreSQL
5. Populate secrets files and create the Decider database / user
6. Populate the database
7. Configure uWSGI for SSL

Sample Install Process

Decider has been tested on Ubuntu 20.04 and CentOS 7.

- For detailed information on the installation and configuration process with CentOS 7, please refer to the CentOS 7 Decider Configuration section in the Appendix.

Python must be version 3.8.10.

- Ubuntu 20.04 comes with Python 3.8.10.
- One can always install 3.8.10 if not already present.

PostgreSQL must be version 12 or later.

Ubuntu 20.04 Setup

Before beginning the Decider installation, there are a few pre-requisites that need to be installed.
Following this tutorial requires an active internet connection.
First, download new package information.
Then, install pip / test resources for Python3, and libssl-dev for uWSGI's https capability.

```
sudo apt update

sudo apt install -y git python3-pip python3-testresources libssl-dev
```

When running a web service, the root user should not be used to run the application. A specific unprivileged user without sudo access should be created for this purpose and should only be given access to the Decider application directory.

```
sudo adduser --no-create-home --system --shell /bin/false decider && sudo usermod -L decider

sudo groupadd decider && sudo usermod -aG decider decider
```

Install PostgreSQL.

```
sudo apt install -y postgresql postgresql-contrib

sudo systemctl start postgresql && sudo systemctl enable postgresql
```

Create the Decider application directory, copy the tar into it, and extract the archive. The install directory will need to have the owner and group changed to decider:decider. The permission of the directory and the files will need to be changed. When inflating the Decider archive, the files may be placed into another folder. Please move the contents of this folder to /etc/decider.

(Other option for the top block of commands)
You can also clone the repository by using sudo + git to write into /etc.
Use sudo for git checkout as well (/etc permissions again).
Then you can chown recursively to decider as mentioned below.

The bottom block installs required Python package dependencies globally.

Visit this section for a better method of handling python/pip package installation.
Avoiding "sudo pip" – Python Environment Best-Practices

```
sudo mkdir /etc/decider && sudo mv decider.tar.gz /etc/decider/decider.tar.gz

cd /etc/decider

sudo tar -xvf decider.tar.gz

sudo chown -R decider:decider /etc/decider

sudo find /etc/decider -type d -exec chmod 755 {} +

sudo find /etc/decider -type f -exec chmod 644 {} +


sudo pip3 install wheel==0.37.1

sudo pip3 install -r requirements.txt
```

Decider and its Python requirements have now been installed.
Next, we will edit the configuration of Decider.

Configuration File

Decider is configured primarily by 2 files:

- `.env`:
  - Holds credentials for a DB account used by Decider for queries.
  - Holds a key used to encrypt carts saved in the database.
- `app/conf.py`:
  - Allows creating different configs that can be passed to Decider on startup.
  - Reads DB user/pass from `.env` when making the connection URI.
  - Holds a wide swath of configuration settings (*explained below; snippet included*).

```
• • •
class Config:
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    DECIDER_LOG = "./decider.log"
    LOG_LEVEL = "INFO"
    START_QUESTION = "What is the adversary trying to do?"
    BASE_TECHNIQUE_ANSWER = (
        "There was **not enough context** to identify "
        "a sub-technique, but the **base technique still applies**."
    )
    WTF_CSRF_TIME_LIMIT = None

class DevelopmentConfig(Config):
    SQLALCHEMY_DATABASE_URI = sqlalch.engine.URL.create(
        drivername="postgresql",
        username=DB_USER_NAME,
        password=DB_USER_PASS,
        host="decider-db",
        port=5432,
        database="productiontables",
    )
    LOG_LEVEL = "DEBUG"
    TESTING = True
    WTF_CSRF_CHECK_DEFAULT = False


class PytestConfig(Config):
    SQLALCHEMY_DATABASE_URI = sqlalch.engine.URL.create(
        drivername="postgresql",
        username=DB_USER_NAME,
        password=DB_USER_PASS,
        host="decider-db",
        port=5432,
        database="testing",
    )
    # TESTING = True, off - so auth can be tested
    WTF_CSRF_CHECK_DEFAULT = False
• • •
```

`Config` is the base class that controls defaults across all other configurations. There is a production, development, and default configuration. If Decider is not provided a configuration class on launch using `--config`, then the default configuration (`DefaultConfig`) will be used. (There also exist staging

and testing configs. Staging is a pre-production deployment ground. Testing DB is to be setup and torn down before and after unit tests are ran.)

SQLALCHEMY_TRACK_MODIFICATIONS – This setting will be tracking modifications to objects. Requires more memory usage. It is recommended to set this to False. Default is True.

DECIDER_LOG – Location of Decider's log

LOG_LEVEL – The level of information to output to log. Supported values are INFO, DEBUG, WARNING, ERROR, and CRTITICAL.

START_QUESTION – This is the question that is presented to the user on the first page of Decider (navigates users to a tactic).

BASE_TECHNIQUE_ANSWER – This is the default answer when a sub-technique cannot be mapped and the user must map to the parent technique.

SQLALCHEMY_DATABASE_URI – The database connection string:

- host – The hostname or IP of the database server.
- database – The database within the server to use.
- port – The port that PostgreSQL listens on. The default is 5432
- username, password – Credentials that Decider will use to access the database.
- drivername – What type of database will be connected to. (*Only Postgres is supported.*)

TESTING – This enables testing/development, which disables many security features for easier access. Default value is false.

WTF_CSRF_CHECK_DEFAULT – This disables CSRF checking for easier development. Default value is false.

Now on to setting up and populating the database.

Logging Configuration

Config Location

- Decider uses the de-facto Python Logging Module
- The configuration file for Decider's logging is: **app/logging_conf.yaml**
    - This gets loaded in **decider.py**

Default Configuration

- Decider uses two handlers by default: `StreamHandler` and `FileHandler`
    - `StreamHandler` logs to `stdout`
    - `FileHandler` logs to **decider.log**
        - There is no size limitation or file rotation occurring here
    - Each handler is set to `DEBUG` by default
- This setup is basic and provides a turn-key experience for a local developer
    - Anyone deploying an instance of Decider likely has their own logging requirements

Custom Configuration Advisement

- Exploring what Logging Module Handlers exist is the first step
    - `RotatingFileHandler` or `TimedRotatingFileHandler` would provide rotation where Decider manages the log files
    - `WatchedFileHandler` works when utils such as newsyslog / logrotate are used to manage the logs
    - There exist other options that are network or syslog-based
- From there, the `handlers:` potion of **app/logging_conf.yaml** must be edited to apply the desired setup

Database Setup

1. Run "`sudo -u decider python3 initial_setup.py`",
   it will ask you for information.
   You will be creating two logins and a secret key (a password effectively).
   Below are the 5 pieces of information it wants you to create:
    a. `DB_USER_NAME`
    b. `DB_USER_PASS`
       ^- postgres account that Decider will use to query the database

c. `CART_ENC_KEY`
^- encryption key/password that Decider will use to encrypt carts saved in the database

d. `APP_ADMIN_EMAIL`
e. `APP_ADMIN_PASS`
^- an admin account that you will use to login to the Decider app website itself

2. Run the newly created `init.sql` script using `"sudo -u postgres psql -a -f init.sql"`.
   a. A new database called "`decider`" has just been created.
   b. A user with perms on this new database was made as well.

3. Modify `ProductionConfig` in `app/conf.py`
   `sudo -u decider nano app/conf.py`
   a. `host:` should be set to the hostname / IP where the PostgreSQL server is running.
      i. Will be `localhost` if you install Postgres and Decider on the same machine.
   b. `port:` should be set to the port that PostgreSQL is running on.
      i. (*5432 default is likely fine*)
   c. `database:` should be set to "`decider`" to match the newly created database.

4. Modify content to be built to the DB by adding/removing jsons in `app/utils/jsons/source/`…
   (This step is optional. This just allows removing / adding additional content if desired)

   a. `role.json` is **<u>required</u>** for base app functionality.

   b. `user.json` is **<u>required</u>** so an admin can login and add other users.

   c. ATT&CK & question tree content is in …`/enterprise-attack/` and …`/tree/`
      i. <u>At least one version is required for the app to run.</u>
      ii. An entry in each folder for a given version is **<u>required</u>** for that version to install.
      iii. Removal of an ATT&CK version also means that no AKAs, CoOccurrences, or Mismappings content will install for that version either.

   d. AKA content is in …`/akas/`
      i. Entirely optional per version.
      ii. Provides keywords / tags for Techniques, they're used in the Full Technique Search

   e. CoOccurrence content is in …`/co_occurrences/`
      i. Entirely optional per version.
      ii. Given the occurrence of one Technique, this provides suggestions of other Techniques that likely occurred as well

   f. Mismapping content is in …`/mismappings/`

i. Entirely optional per version.
ii. Allows loading existing Mismapping content (can always add more in-app). Provides instances of when a given Technique was incorrectly mapped. Also suggests what Technique should have been mapped (if it exists).

5. Build the DB with "`sudo -u decider python3 -m app.utils.db.actions.full_build --config ProductionConfig`"
    a. The database now has content and a starting user.
    b. Please ensure the app works fine / was setup correctly by running it:
        i. `sudo -u decider python3 decider.py --config ProductionConfig`
        ii. You can reference the user guide for features to be checked
            1. Basic litmus test is ensuring that a full search works, navigating question cards works, and that you can add items to a cart / save it to the DB.

6. Delete "`init.sql`" and "`user.json`"
    `sudo -u decider rm init.sql`
    `sudo -u decider rm app/utils/jsons/source/user.json`
    a. "`init.sql`" is in the application root
    b. "`user.json`" is in `app/utils/jsons/source`
    c. Removing these ensures that the only place secrets are stored in is the `.env` file.

uWSGI / Service Setup

OPTIONAL STEP: If the UFW is enabled on the Ubuntu host, the firewall will need to be configured to allow the proper connections.

```
sudo ufw allow 443/tcp
```

Create a uwsgi.ini file in the root of the Decider application. This file is used for the uWSGI configuration and is called every time Decider launches. This guide assumes that you already have an SSL certificate or can obtain one. To generate a self-signed SSL certificate for uWSGI, visit https://uwsgi-docs.readthedocs.io/en/latest/HTTPS.html.

Execute these 3 statements 1-by-1 to generate a self-signed cert if needed (instructions from link above)

```
sudo -u decider openssl genrsa -out decider.key 2048
sudo -u decider openssl req -new -key decider.key -out decider.csr
sudo -u decider openssl x509 -req -days 365 -in decider.csr -signkey decider.key -out decider.crt
```

Copy the certs to the Decider app/utils/certs folder

```
sudo -u decider cp decider.crt /etc/decider/app/utils/certs/
sudo -u decider cp decider.key /etc/decider/app/utils/certs/
```

Create uwsgi.ini – it configures uWSGI to use our certs and to launch Decider on run

`sudo -u decider nano uwsgi.ini`

(paste the textbox in, save)

```
[uwsgi]
chdir=/etc/decider
module = decider:app
master = true
processes = 5
pyargv = --config ProductionConfig
shared-socket = 0.0.0.0:443
uid = decider
gid = decider
https = =0,/etc/decider/app/utils/certs/decider.crt,/etc/decider/app/utils/certs/decider.key
enable-threads = true
```

To set uWSGI to run on system launch, we can create a Unit file to define a systemd service. In your current directory, create a file named decider.service with the following contents.

`sudo -u decider nano decider.service`

(paste the textbox in, save)

```
[Unit]
Description = Decider

[Service]
Type=simple
ExecStart=uwsgi --ini /etc/decider/uwsgi.ini

[Install]
WantedBy=multi-user.target
```

Then, run the following commands to copy the service file to the necessary locations, and enable Decider to start on system launch.

```
sudo cp decider.service /lib/systemd/system/decider.service
sudo cp decider.service /etc/systemd/system/decider.service
sudo chmod 644 /etc/systemd/system/decider.service
sudo systemctl start decider
sudo systemctl status decider
sudo systemctl enable decider
```

Status is used to check that the Decider service launched fine.

Maintenance

Backup

Decider comes with tools to perform backups. There are two ways to back up the database: the Decider dump scripts and the PostgreSQL native dump tool. The dump scripts can be run in order to back up the various modifiable portions of Decider: carts, users, mismapping content, and tree content. Each of the scripts creates a JSON file containing the specified data to dump (varies based on which script was ran, the `--config` used, and the `--version` dumped from). The second method is to use **pg_dump**, the PostgreSQL tool. This tool will dump the database into a file containing SQL commands. This format is a bit more difficult to parse through, but, **it is the recommended route for painless automated backups**.

The Decider dump scripts can be run using:

```
python3 -m app.utils.db.actions.dump_carts --config C → cart.json
python3 -m app.utils.db.actions.dump_users --config C → user.json
python3 -m app.utils.db.actions.dump_mismappings --config C --version V → co-occurrences-V.json
python3 -m app.utils.db.actions.dump_tree_content --config C --version V → tree-content-V.json
```

The produced files are created in `app/utils/db/dumps/`. The dump scripts can be run non-interactively as shown above, or, if no arguments are provided, they will ask the user for selections via prompts. NOTE: dump_carts will dump the carts unencrypted. This set of dump scripts is geared more towards general content management and relocation. One can dump their current set of mismappings and apply them to a new version of ATT&CK using add_version if desired. It is recommended that pg_dump is used for database backups. It does not provide the granular management options of the scripts, but instead gives a convenient single-file solution to backup and restore the whole database at once.

Regular backups of Decider should be made. This can be scheduled via a cron job with pg_dump. This job will run every 6 hours of every day and dump the database named decider into a folder. The files are named by the date and hour the backup was made. An example is provided below:

```
0 */6 * * * pg_dump -U postgres decider > /var/lib/postgresql/backups/decider-`date +\%m\-\%d\-\%H`.bak
```

NOTE: pg_dump will not automatically decrypt any carts stored in the database. The encryption and decryption is done on the python side, so the values stored are the ciphertexts. If pg_dump is used, please DO NOT lose the secret key (`CART_ENC_KEY`) for Decider. To export carts as decrypted, please use the Decider dump script dump_carts.

In regard to backing-up Decider's database content, it is suggested that there be a cut-off date where old backups are deleted. This is not required, and backup frequency / age of backups to keep is wholly up to the admins deploying an instance.

```
0 */6 * * * pg_dump -U postgres decider > /var/lib/postgresql/backups/decider-`date +\%m\-\%d\-\%H`.bak; find /var/lib/postgresql/backups/ -mtime +29 -exec rm {} \;
```

This is an example of the earlier cron job with an additional command that deletes backups that are older than 29 days. This helps place an upper limit on the amount of space used for backups.

NOTE: while user-generated content is small (plaintext), the space used will be multiplied by the total number of backups created.

To provide a perspective on size: we have 2 databases being backed-up, that have essentially no user-added content, backups occur every 6 hours and are deleted after 29 days – this comes to roughly 2 GB of disk usage.

Export Data for External Analysis

There is another script for dumping the information from the database. However, this script is not recommended for backup processes or maintenance actions (as there exists no restore method for the information it dumps).

```
python -m app.utils.dump_tables
```

This script dumps all tables from the database as JSON files, (list of dictionaries format), in the directory app/utils/jsons/tables/. The cart table is excluded from this dump as to restrict avenues of access to cart information. This script allows for exporting the database's content in a simple and accessible manner. The produced JSON files are easy to use with any language without needing to resort to SQLAlchemy setup.

NOTE: *this serves no purpose for the normal use cases of Decider; however, it allows for exploration of content programmatically if desired.*

Database Scripts

Overview

This is the full list of available maintenance scripts:

- python -m app.utils.db.actions.full_build
- python -m app.utils.db.actions.add_version
- python -m app.utils.db.actions.remove_version

- python -m app.utils.db.actions.dump_mismappings
- python -m app.utils.db.actions.dump_tree_content
- python -m app.utils.db.actions.dump_carts
- python -m app.utils.db.actions.dump_users

Please ensure that the Decider environment (from requirements.txt) is properly setup and sourced before using these.
These scripts / modules are to be called from the main /etc/decider/ directory.

Calling any script with the **-h** or **--help** flag will show possible arguments and a short mention of the script's purpose.

All of the scripts have a **--config** flag / option.
This allows specifying which database the script will run against.
Config options are defined inside of, (and can be modified in), app/conf.py

Each script can use either command-line arguments **OR** selections that you type out from the options it offers.

- Using no command-line arguments will cause the prompts to appear.
- Using all command-line arguments will cause the script to run immediately without pause for option selection.
- Using some, (but not all), command-line arguments will cause a warning message that either ALL or NONE of the command-line arguments should be specified.

Example Help Prompt

python -m app.utils.db.actions.remove_version --help
usage: Removes an ATT&CK version from the database. [-h] [--config CONFIG] [--version VERSION]

optional arguments:
-h, --help show this help message and exit
--config CONFIG The database configuration to use (from app/conf.py).
--version VERSION ATT&CK version to be removed.

Example Usage of No Command-line Arguments

python -m app.utils.db.actions.remove_version
Available app/database configs: ['DefaultConfig', 'ProductionConfig', 'StagingConfig', 'DevelopmentConfig', 'PytestConfig']

Which config to use [DefaultConfig]: **(enter was hit, thus selecting this default)**

Versions installed on the database: ['v10.0', 'v8.0']

What version to remove: v8.0

Removal Detail:
- ATT&CK Version v8.0:
  - CoOccurrences
  - AKAs
  - Mismappings

…

Example Usage of All Command-line Arguments:

python -m app.utils.db.actions.remove_version --config DefaultConfig --version v8.0

Removal Detail:
- ATT&CK Version v8.0:
  - CoOccurrences
  - AKAs
  - Mismappings

…

Example Warning of Partial Command-line Argument Usage:

python -m app.utils.db.actions.remove_version --config DefaultConfig ***(--version is missing)***
Either ALL or NONE of the command-line args should be defined. Exiting.

--Version Option Explained
This allows you to select which ATT&CK version the current interaction script is working with (assuming you are using a script where version matters). full_build is version agnostic as it builds all versions into

the database as they are found on the disk. dump_users and dump_carts are version agnostic as users aren't tied to a version and a cart can exist for any version.

dump_tree_content and dump_mismappings are version-tied, as their content is directly associated with their version. Trying to use --version V with a script that doesn't take it will cause it to error out with an invalid argument warning.

Initial Deployment (Full Build)

python -m app.utils.db.actions.full_build --config CONFIG

This script takes all source JSON data from app/utils/jsons/source and builds the database from it.
This is **DESTRUCTIVE**, running this on an existing instance will overwrite everything currently on the specified database.

For the initial deployment of Decider, this script is to be ran 2 times, once for the development/testing database, and once for the production/deployment database.

The ATT&CK Enterprise JSON can be downloaded at https://github.com/mitre/cti/blob/master/enterprise-attack/enterprise-attack.json.
The release tags can be used to select a desired version.
Make sure to rename the file to enterprise-attack-VERSION.json so it can be read by the build scripts.

Adding New ATT&CK Versions

python -m app.utils.db.actions.add_version --config CONFIG --version VERSION

This script adds ATT&CK content for a version not already present in the database.
Trying to add a version already present in the database causes the script to exit with a warning.

Please ensure that JSON sources are added to app/utils/jsons/source before running this script.

Required Sources:

- app/utils/jsons/source/enterprise-attack/enterprise-attack-**VERSION**.json
- app/utils/jsons/source/tree/tree-content-**VERSION**.json

Sources Explained:

- enterprise-attack-**VERSION**.json: This provides the ATT&CK content itself – Tactics, Techniques, and their descriptions.
- tree-content-**VERSION**.json: This provides the question and answer card content corresponding to a given version of ATT&CK. This allows for navigation through the decision tree in Decider.

Optional Sources:

- app/utils/jsons/source/co_occurrences/co-occurrences-**VERSION**.json
- app/utils/jsons/source/akas/akas-**VERSION**.json
- app/utils/jsons/source/mismappings-**VERSION**.json

Sources Explained:

- co-occurrences-**VERSION**.json: This provides information about Techniques that are likely to appear together. Including this will provide the "Frequently Appears With" section on the success page of Techniques that strongly imply other Techniques occurring. This also enables the feature to see all implied Techniques for a whole cart at once.
- akas-**VERSION**.json: This provides extra search / tagging information. This dataset provides keywords / phrases and which Techniques they typically imply. Aside from increasing searchability of content – the AKAs associated with a Technique will be displayed on its success page.
- mismappings-**VERSION**.json: This dataset provides in-app editable content that describes when Techniques are commonly mis-mapped with one another. Information describing the originally mapped Technique, the corrected Technique (if the behavior can be mapped at all), and a description as to why the original mapping was incorrect.

The add_version script will give a print-out of features added when adding a version to the database. If a feature is not being added as expected – check the script output, when an optional feature cannot be located – it will be mentioned. Run remove_version, ensure the source data is in the proper places, and then attempt add_version again.

Removing Old ATT&CK Versions

python -m app.utils.db.actions.remove_version --config CONFIG --version VERSION

This script removes all content for a specified version of ATT&CK on the specified database (/config). This is **DESTRUCTIVE**, any DB-saved carts and content pertaining to this version will be deleted.

Dumping Content for a Rebuild

python -m app.utils.db.actions.dump_mismappings --config CONFIG --version VERSION
python -m app.utils.db.actions.dump_tree_content --config CONFIG --version VERSION
python -m app.utils.db.actions.dump_carts --config CONFIG
python -m app.utils.db.actions.dump_users --config CONFIG

A rebuild consists of running these scripts first, moving the dumped content into the sources folder, and then running full_build.
The purpose of dumping this content as part of the rebuild process is that this content is user-editable. Calling full_build without dumping content and storing it in the appropriate place will cause deletion of content on the database but not the disk.

Each of these scripts dumps a file to the folder app/utils/db/dumps.
These dumped files should be placed in their appropriate location within app/utils/jsons/source.

- mismappings.json, cart.json, and user.json all go straight into the app/utils/jsons/source folder.
- tree-content-VERSION.json should be placed into the app/utils/jsons/source/tree folder.

Finally, full_build can be called to rebuild the database.
Instead of being deleted, the dumped content will now be included as part of the build process.

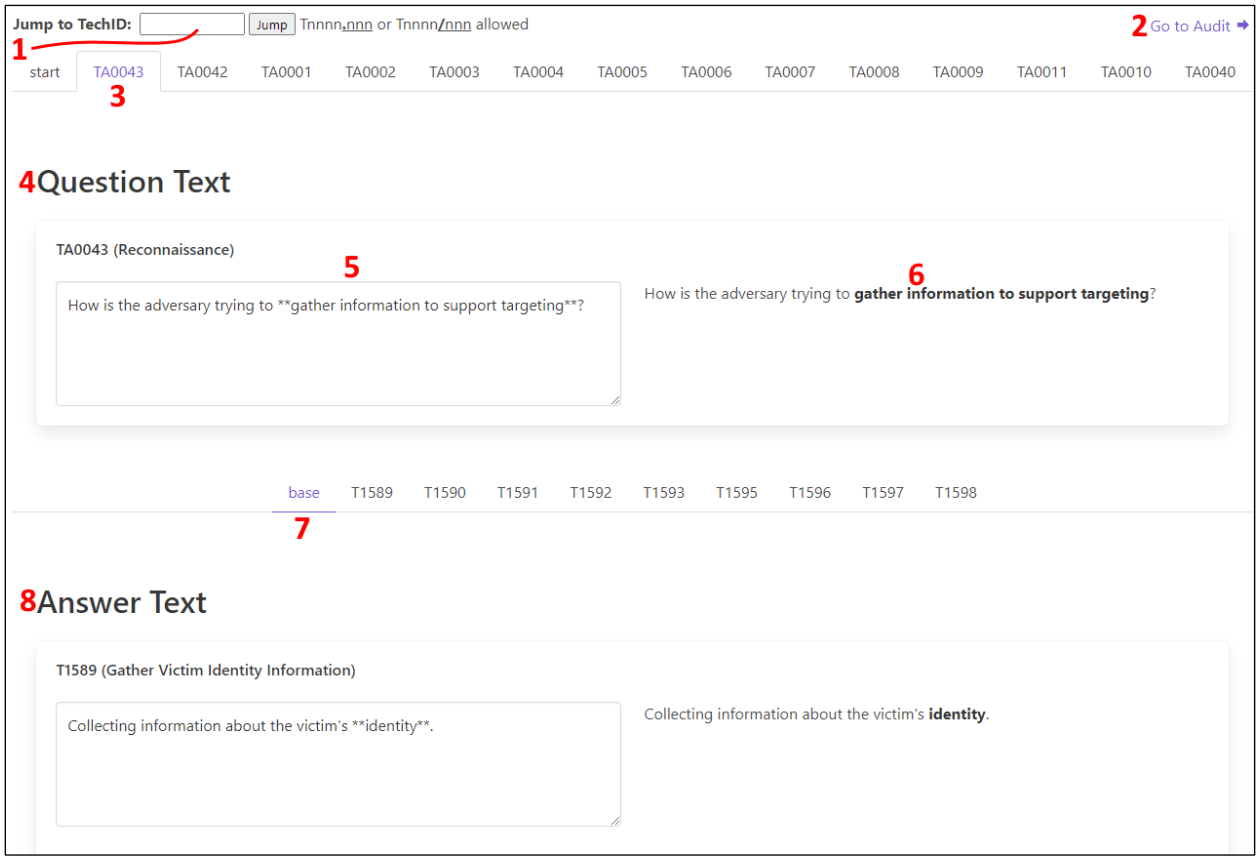
Content Editing

Mismappings, questions, answers and users can be modified from within Decider. Normal users are only able to view the content. Editors can modify questions, answers and mismappings, while admin users are able to do all of the above and add or delete other users. Additionally, user passwords can be changed by the administrator. All users are able to change their own passwords.

Content beyond those listed can be modified directly in the JSON files. Once those have been changed, the build script can be run again to build the entire database with the updated changes. ***NOTE:*** *This will wipe any existing changes made in the database through the administrative tools. Please see the database scripts section for more details.*

Decision Tree – Question / Answer Editing

Location: Profile > Editor Tools > Edit Card Content



1. **Jump to TechID:** This input box allows you to jump directly to a Technique's Answer Card editing box. Simply type / paste the Technique ID into this box and hit [Enter] or click "Jump". If you wish to get to the Technique's Question content editing box – you can click on the Technique's Tab after doing a jump to the proper Tactic page.
2. **Link to Audit Page:** This is a link to the "Audit Card Content" page. The audit page lists any issues that must be resolved in order to make all question and answer content fit style requirements. The audit page has a link back to this editing page. This allows for easily checking progress during a content editing process.
3. **Tactic Tabs:** "start" allows editing the Tactic answers that appear on the first page of the decision tree. The other tabs here are for each Tactic. Clicking one will allow editing the question for that Tactic and the answers for the Techniques under it.
4. **Question Text:** This is the question that is found on the top of each page. For the start page, the question is provided in the conf.py file in the source code. This can be changed by a server administrator. Questions for all other tactics and techniques can be changed here.
5. **Edit Box:** This box is where the user can enter in content. On each change, the content inside the text area will be sent to the server and rendered in the preview section. The cards can

support Markdown syntax for styling the content. More information about Markdown can be found in the appendix.

6. **Content Preview:** This section displays how the content will render. If content does not show up in the preview after the content has been modified in the edit box, that means there is an issue saving the content on the server.
7. **Technique Tabs:** "base" allows editing the answers for the base techniques directly under the current tactic question selected (in the higher tactic tabs row). The other tabs allow editing content for techniques that have sub-techniques. Clicking a tab will allow editing the base technique question, as well as the answers for the sub-techniques under it.
8. **Answer Text:** The answer section is the content for the cards. Under the base tab, only the card content will be shown. Under the tab for a technique, the user will have an opportunity to modify the question at the top of the page for a technique.

NOTE: Techniques can be associated with multiple Tactics in ATT&CK (basically: "this behavior is related to these goals"). In Decider, a Technique can only have one answer. So, it is not possible to have different answers for a Technique under two different Tactic questions. This can influence how content edits are handled.

Decision Tree – Question / Answer Auditing

Location: Profile > Editor Tools > Audit Card Content

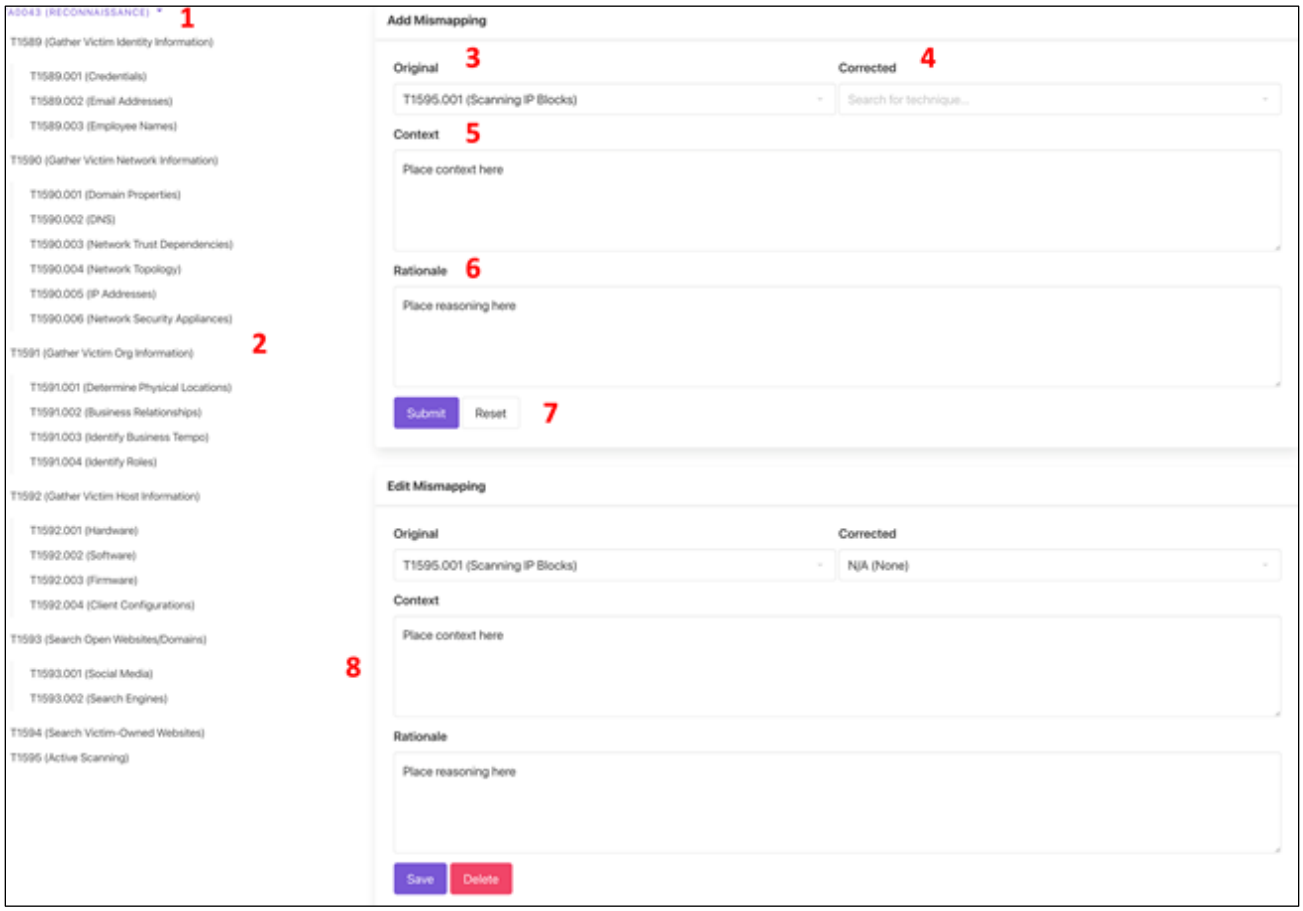
This auditing page lists every issue that exists in the current decision tree content.
Resolving every issue will bring the question / answer content to parity with the style requirements.

1. **Link to Question / Answer Editing Page:** This links to the tree content editing page. The editing page itself has a link pointing back to here. This allows for easily checking progress during a content editing process.
2. **ID:** This is the Technique ID or Tactic ID associated with the current issue entry. The issue rows in the table are ordered by this column. Tactics before Techniques, and then numerically ascending from there.

3. **Name:** This is the name of the Technique or Tactic the issue row is referencing. This is also a link to the MITRE ATT&CK page for said Technique / Tactic. This allows for quickly pulling up reference material to be used during the editing process.
4. **Issue Description:** This explains what is wrong with the current Technique/Tactic question/answer content.
   Reported Issues Include:
   a. Any cards that are entirely missing content *(happens when new Techniques are added)*
   b. Any Answer cards containing a question mark
   c. Any Question cards missing a question mark
   d. Any cards that are entirely missing bolding

Mismappings

Location: Profile > Editor Tools > Edit Mismappings



1. **Menu:** Each technique is listed under its associated tactics. By clicking on the dropdown, the user is able to show / hide the list of techniques. This panel is scrollable to show additional tactics and techniques.

2. **Technique List:** When a technique has been clicked on, the mismapping form is populated automatically. The “Original” field is changed to match the technique that was selected.
3. **Original:** This is the technique that has been mapped incorrectly. There is a dropdown menu that allows the user to select a technique by searching or by scrolling through and clicking. This technique field is always defined (no N/A option available).
4. **Corrected:** This is the technique that the original is being corrected to. Using the N/A value in this dropdown means that the originally mapped behavior does not have a technique to map to.
5. **Context:** The context field provides details describing the situation that led to the original technique being mapped. This, in tandem with the Rationale, helps describe why mapping a different technique is more appropriate, or, why the adversary’s behavior is not mappable.
6. **Rationale:** The rationale provides the reasoning behind why the mismapping exists.
7. **Reset:** The reset button will clear all fields except for original. Although a user could overwrite the original technique field and save, the saved mismapping will not display underneath the add form.
8. **Edit Mismapping:** This section displays all previously submitted mismappings for a given technique. The user can edit the corrected technique, context, and rationale as well as delete the mismapping.

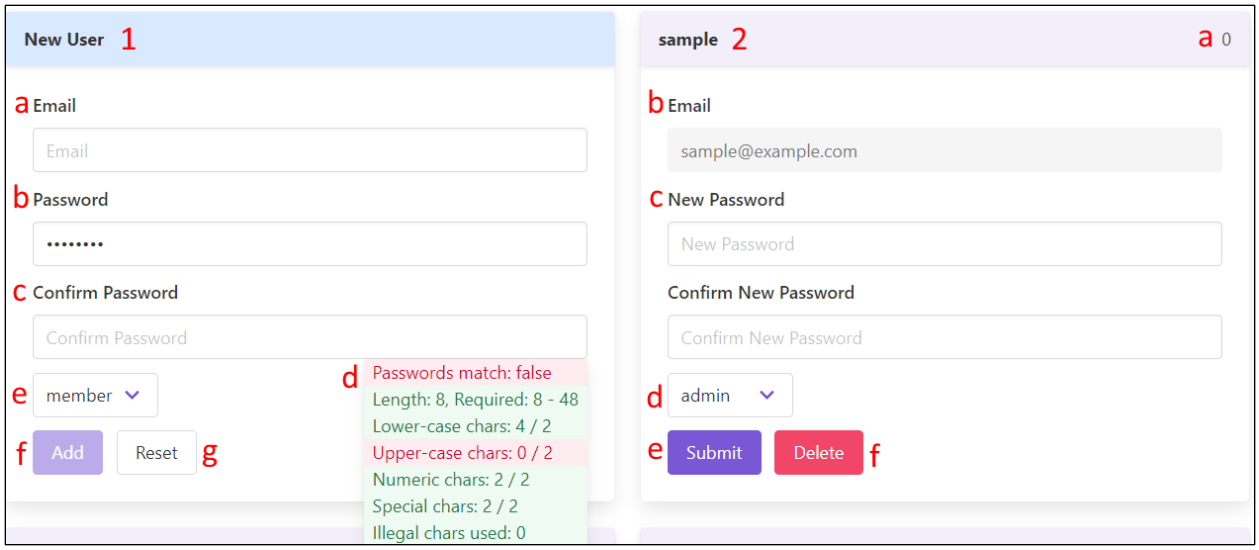
User Management

Location: Profile > Administrator Tools > User Management

Users can be added, deleted, or modified from within Decider.

Users can fit into one of three roles: member, editor, and admin. The member can only view content. The editor can edit questions, answers, and mismappings. The admin can perform all the mentioned actions and manage users. The roles can be modified in the user management page by an admin.

Additional roles cannot be added by an admin. To add additional roles, a developer must add to the roles table and create the associated permissions to restrict or allow these roles on the routes.



1. **New User:** This form allows adding a new user.
    a. **Email**: must be unique and valid. (just used as an identifier, not for communications)
    b. **Password**: must match (c) and fit requirements of (d). (will be changed to SAML 2.0 soon)
    e. **Role:** new user can be set as a “member”, “editor”, or “admin”.
    f. **Add:** adds the new member.
    g. **Reset:** Clears the form.
2. **Existing User:** this form allows managing an existing user (name from email is displayed)
    a. **User ID:** this user’s unique ID from the database.
    b. **Email:** their email cannot be changed.
    c. **New Password:** allows updating the password. Same requirements process as in 1.
    d. **New Role:** allows updating their role.
    e. **Submit:** pressing this will apply the selected changes to the user.
    f. **Delete:** pressing this will remove this user entirely.

Appendix

Flask Server Routes

This is a listing of every route in Decider; it helps give an overview of the potential attack surface.
The following snippets are from the code itself.

Decorators are present to indicate:

- The URL(s) the route is accessible by
- The REST method(s) the route responds to
- Permissions needed to visit the route (all are "member" by default, **@public_route** allows login)
- **ErrorDuringHTMLRoute** signals that the route is to be visited as a page in the browser
- **ErrorDuringAJAXRoute** signals that the route is to be called while already on a page

admin.py

```
@admin_.route("/admin/user", methods=["GET"])
@admin_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def admin_user_get():
```

```
@admin_.route("/admin/user", methods=["POST"])
@admin_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def admin_user_post():
```

```
@admin_.route("/admin/user", methods=["PATCH"])
@admin_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def admin_user_patch():
```

```
@admin_.route("/admin/user", methods=["DELETE"])
@admin_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def admin_user_delete():
```

api.py

```
@api_.route("/api/versions", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def get_versions():
```

```
@api_.route("/api/mismappings", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def get_mismappings():
```

```
@api_.route("/api/tactics", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
```

```
def get_tactics():
```

```
@api_.route("/api/techniques", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def get_techniques():
```

```
@api_.route("/api/user_version_change", methods=["PATCH"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def user_version_change():
```

```
@api_.route("/api/answers/", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def answers_api():
```

```
@api_.route("/api/cooccurrences", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def cooccurrences_api():
```

```
@api_.route("/api/techid_to_valid_tactid_map/<version>", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def techid_to_valid_tactid_map(version):
```

auth.py

```
@auth_.route("/login", methods=["GET"])
@public_route
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def login():
```

```
@auth_.route("/login", methods=["POST"])
@public_route
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def login_post():
```

```
@auth_.route("/logout", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def logout():
```

docs.py

```
@docs_.route("/changelog", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def changelog():
```

edit.py

```
@edit_.route("/edit/mismapping", methods=["GET"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def edit_mismapping_get():
```

```
@edit_.route("/edit/mismapping", methods=["POST"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def edit_mismapping_post():
```

```
@edit_.route("/edit/mismapping", methods=["DELETE"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def edit_mismapping_delete():
```

```
@edit_.route("/edit/tree", methods=["GET"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def edit_tree_get():
```

```
@edit_.route("/edit/tree/api", methods=["GET"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def edit_tree_api_get():
```

```
@edit_.route("/edit/tree/api", methods=["POST"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def edit_tree_api_post():
```

```
@edit_.route("/edit/tree/audit/<version>", methods=["GET"])
@edit_permission.require(http_exception=403)
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def audit_tree_content(version):
```

misc.py

```
@misc_.route("/favicon.ico", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def favicon():
```

```
@misc_.route("/word_export", methods=["POST"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def export_cart_to_word():
```

```
@misc_.route("/suggestions/<version>", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def suggestions(version):
```

profile.py

```
@profile_.route("/profile", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def profile():
```

```
@profile_.route("/profile/save_cart", methods=["POST"])
```

```
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def save_cart():
```

```

@profile_.route("/profile/load_cart", methods=["POST"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def load_cart():
```

```

@profile_.route("/profile/delete_cart", methods=["POST"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def delete_cart():
```

```

@profile_.route("/profile/carts", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def get_carts():
```

```

@profile_.route("/profile/change_password", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def change_password_get():
```

```

@profile_.route("/profile/change_password", methods=["POST"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def change_password_post():
```

question.py

```

@question_.route("/", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def home():
```

```

@question_.route("/question/", methods=["GET"])
@question_.route("/question/<version>", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def question_start_page(version=None):
```

```

@question_.route("/question/<version>/<tactic_id>", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def question_tactic_page(version, tactic_id):
```

```

@question_.route("/question/<version>/<tactic_id>/<path:dest>", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def question_further_page(version, tactic_id, dest=""):
```

```

@question_.route("/no_tactic/<version>/<path:subpath>", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def notactic_success(version, subpath=""):
```

search.py

```

@search_.route("/search/mini/<version>", methods=["POST"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def mini_search(version):
```

```

```

```
@search_.route("/search/page", methods=["GET"])
@wrap_exceptions_as(ErrorDuringHTMLRoute)
def search_page():
```

```
@search_.route("/search/full", methods=["GET"])
@wrap_exceptions_as(ErrorDuringAJAXRoute)
def full_search():
```

Markdown Quick Start

Markdown is a popular markup language used by Decider to format text. It is used specifically in the cards and description of the techniques. This document will list which elements are supported with some basic examples.

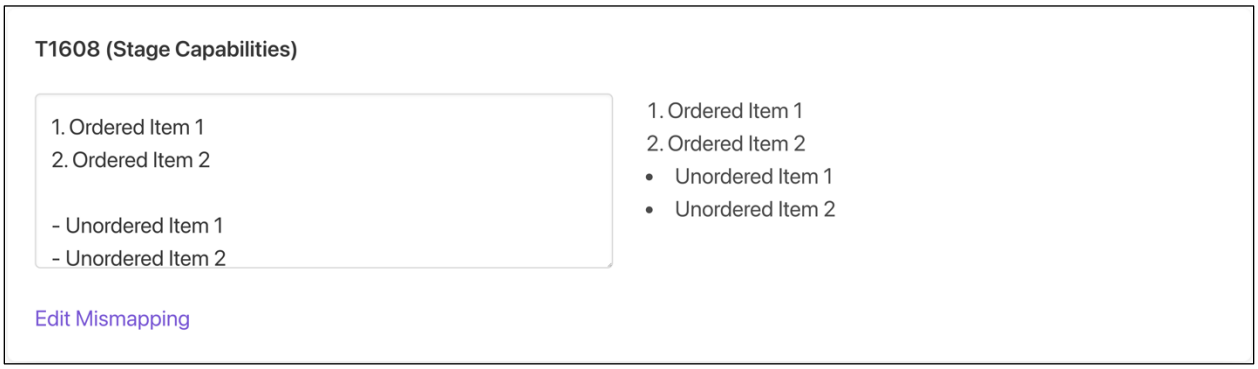
Supported: bold, italics, lists (ordered and unordered), and links. More information on Markdown can be found here: https://guides.github.com/features/mastering-markdown/

Examples:

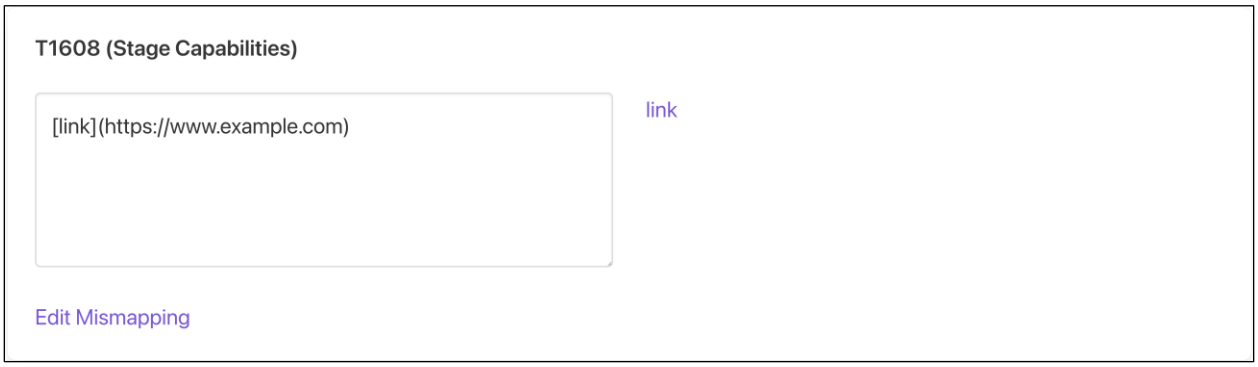
1. Bold Text

2. Italics

3. Lists



4. Links



CentOS 7 – Example Decider Install Process

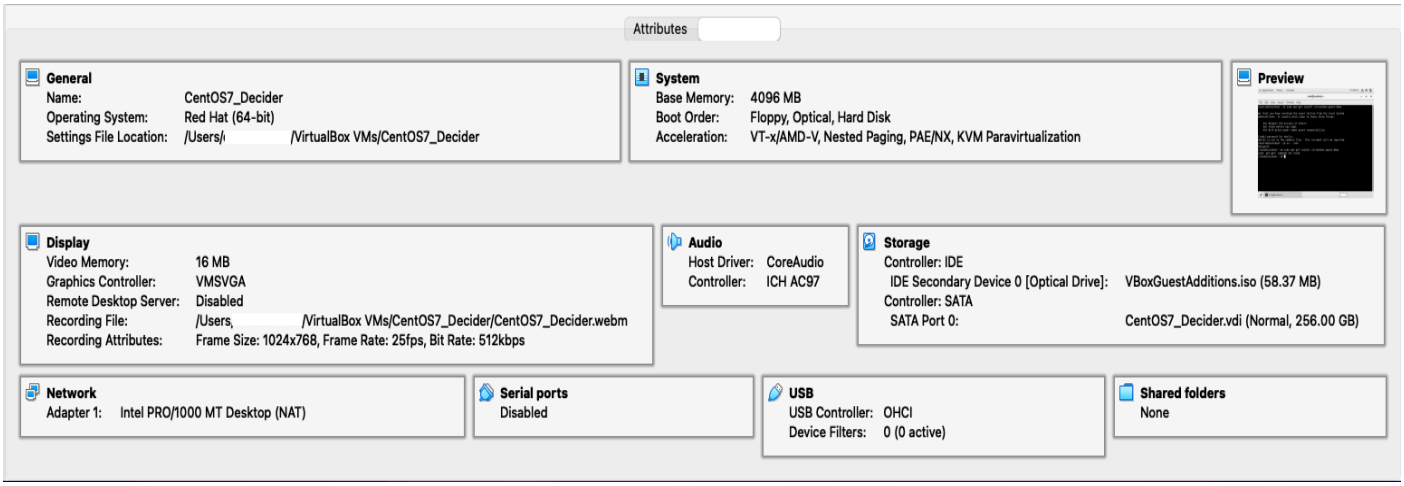
A test instance of Decider was created in a CentOS 7 environment, running under VirtualBox. This section will walk through the installation steps taken.

The first step of this process was to install a CentOS 7 VM in Oracle VirtualBox.

Commands performed included: downloading Python, downloading PostgreSQL, downloading the Decider GitLab repo, and installing necessary Decider requirements and other configurations.

Screenshots of the CentOS 7 VM configuration:

Hardware configuration in VirtualBox



The following are Linux commands for configuring and running Decider on a CentOS 7 VM. Each section is broken down below by its purpose.

1. Preform an update check to CentOS.

```
##(Performed Updates via GUI + Restarted)

# Update Package List
yum check-update
```

2. Adding Decider administrator as `sudo` to eliminate the need to use root level permissions.

```
# Add Self to Sudoers

EDITOR=nano visudo
```

```
    # Under Allow root to run any commands anywhere
    # Under "root   ALL=(ALL) ALL"
    #   Add "USERNAME ALL=(ALL) ALL"
    # Save file
```

3. Add Postgres to repositories , list available (adds GPG keys too), install Postgres, enable service

```
# Add Postgres to repos, list available (adds GPG keys too), install Postgres, enable
service

sudo yum -y install https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-
redhat-repo-latest.noarch.rpm
sudo yum -y list postgres*
sudo yum -y install postgresql12 postgresql12-server postgresql12-contrib postgresql12-libs
```

```
sudo systemctl enable postgresql-12
```

4. Initialize and (re)start Postgres service

```
# Initialize Postgres and (re)start

sudo /usr/pgsql-12/bin/postgresql-12-setup initdb
sudo systemctl restart postgresql-12
```

5. Clone Decider from MITRE Gitlab repository.

```
# Clone Decider

sudo yum install -y git
cd /etc
sudo git clone https://gitlab.mitre.org/dhs-cisa-cti/decider_tool.git decider
```

6. Make Decider User & Group, Own Decider Dir, Ensure Perms are Normal

```
# Make Decider User & Group, Own Decider Dir, Ensure Perms are Normal

sudo adduser --no-create-home --system --shell /bin/false decider && sudo usermod -L decider
sudo groupadd decider && sudo usermod -aG decider decider
sudo chown -R decider:decider /etc/decider
sudo find /etc/decider -type d -exec chmod 755 {} +
sudo find /etc/decider -type f -exec chmod 644 {} +
```

7. Build and Install Python 3.8.10

```
# Build & Install Python 3.8.10

sudo yum install -y yum-utils openssl-devel  # openssl-devel likely already in yum-builddep
for Py
sudo yum-builddep python -y
cd /usr/src
sudo wget https://www.python.org/ftp/python/3.8.10/Python-3.8.10.tar.xz
sudo tar -xvf Python-3.8.10.tar.xz
cd Python-3.8.10
sudo ./configure --enable-optimizations
sudo make install
sudo ln -s /usr/local/bin/python3 /usr/bin/python3
```

8. Install Decider requirements

Visit this section for a better method of handling python/pip package installation.
[Avoiding "sudo pip" – Python Environment Best-Practices]

```
# Install Decider requirements

sudo python3 -m pip install wheel==0.37.1
cd /etc/decider
sudo python3 -m pip install -r requirements.txt
```

9. Setup Secrets, Create DB, Build DB

```
# Setup Secrets, Create DB, Build DB

sudo -u decider python3 initial_setup.py     # define secrets
sudo -u postgres psql -a -f init.sql         # create DB + extensions
sudo -u decider nano app/conf.py
```

```
    # class ProductionConfig(Config) point local { host="localhost", database="decider" }
    # Save file
```

```
sudo nano /var/lib/pgsql/12/data/pg_hba.conf
# We don't use mappings between OS and DB user names
```

```
    # change "ident" to "password" for connection method (local IPv4/6)
    # Save file
```

```
sudo systemctl restart postgresql-12
sudo -u decider python3 -m app.utils.db.actions.full_build --config ProductionConfig  #
build
# (need to remove init.sql and user.json here)
```

10. Generate self-signed SSL cert and stick into proper directory

```
# Generate self-signed SSL cert and stick into proper directory
# FROM: https://uwsgi-docs.readthedocs.io/en/latest/HTTPS.html

sudo RANDFILE=/etc/decider/.rnd -u decider openssl genrsa -out decider.key 2048
sudo RANDFILE=/etc/decider/.rnd -u decider openssl req -new -key decider.key -out
decider.csr
sudo RANDFILE=/etc/decider/.rnd -u decider openssl x509 -req -days 365 -in decider.csr -
signkey decider.key -out decider.crt
sudo -u decider cp decider.crt /etc/decider/app/utils/certs/
sudo -u decider cp decider.key /etc/decider/app/utils/certs/
```

11. Create and Set uWSGI config file

```
# Set uWSGI config

sudo -u decider nano uwsgi.ini
```

```
    # Copy the below text into uwsgi.ini file
```

```
[uwsgi]
chdir=/etc/decider
module = decider:app
master = true
processes = 5
pyargv = --config ProductionConfig
shared-socket = 0.0.0.0:443
uid = decider
gid = decider
https =
=0,/etc/decider/app/utils/certs/decider.crt,/etc/decider/app/utils/certs/decider.key
enable-threads = true
# Save file
```

12. Create Decider service

```
# Create Decider service

sudo -u decider nano decider.service
```

```
# Copy the below text into uwsgi.ini file

[Unit]
Description = Decider

[Service]
Type=simple
ExecStart=/usr/local/bin/uwsgi --ini /etc/decider/uwsgi.ini

[Install]
WantedBy=multi-user.target
# Save file
```

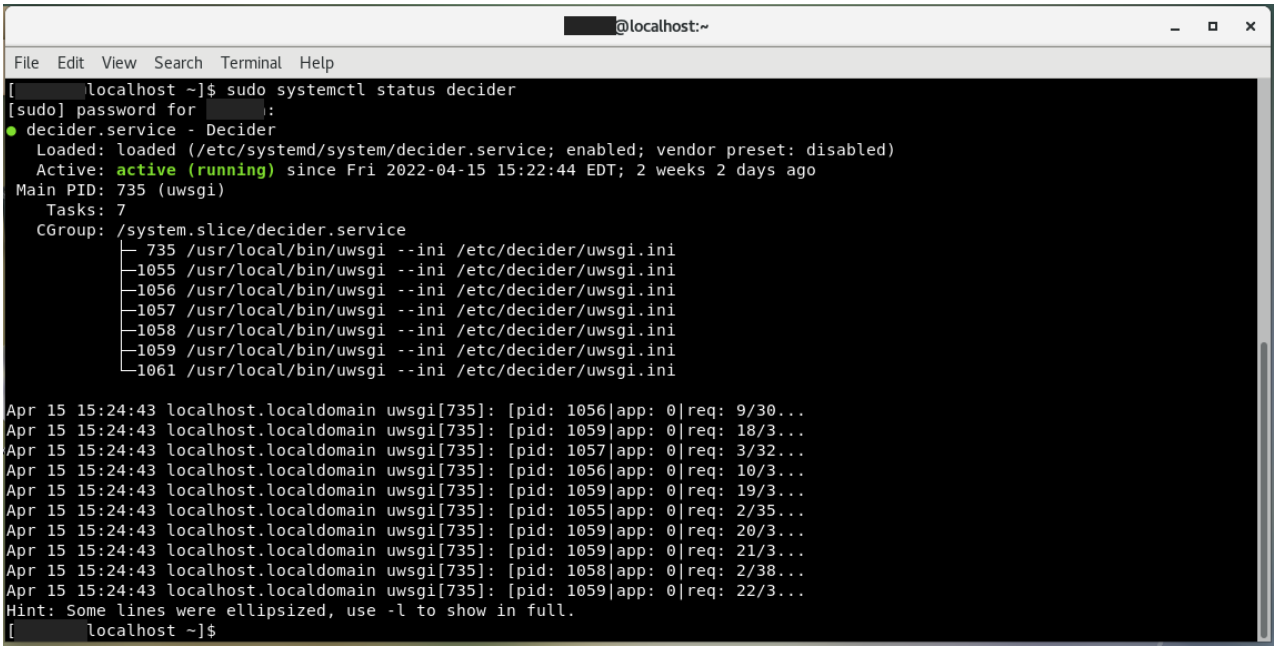
13. Copy service file and launch Decider

```
# Copy service file and launch Decider

sudo cp decider.service /lib/systemd/system/decider.service
sudo cp decider.service /etc/systemd/system/decider.service
sudo chmod 644 /etc/systemd/system/decider.service
sudo systemctl start decider
sudo systemctl status decider
sudo systemctl enable decider
```

14. Allow Requests on Port 443 (HTTPS)

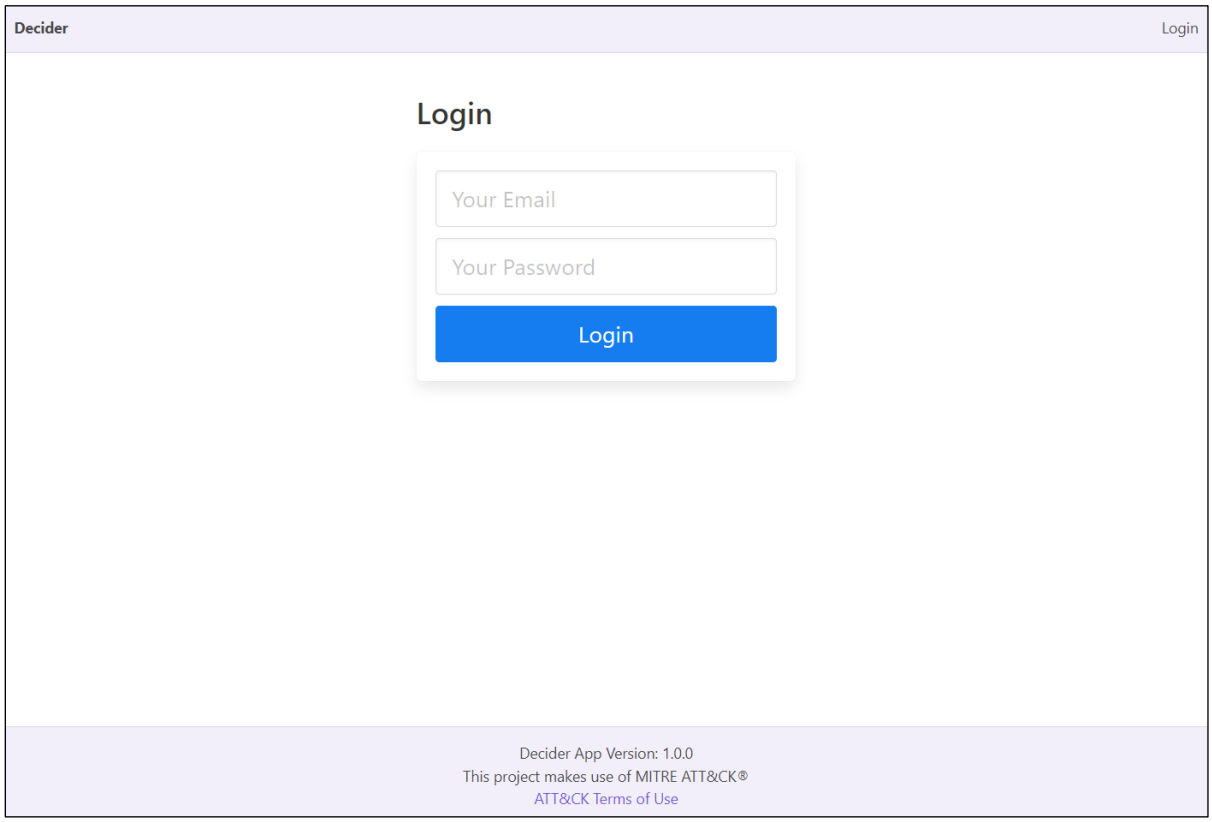
```
# Allow Requests on Port 443 (HTTPS)

sudo firewall-cmd --zone=public --add-port=443/tcp --permanent
sudo firewall-cmd --reload
```

Command and result of checking Decider is running.

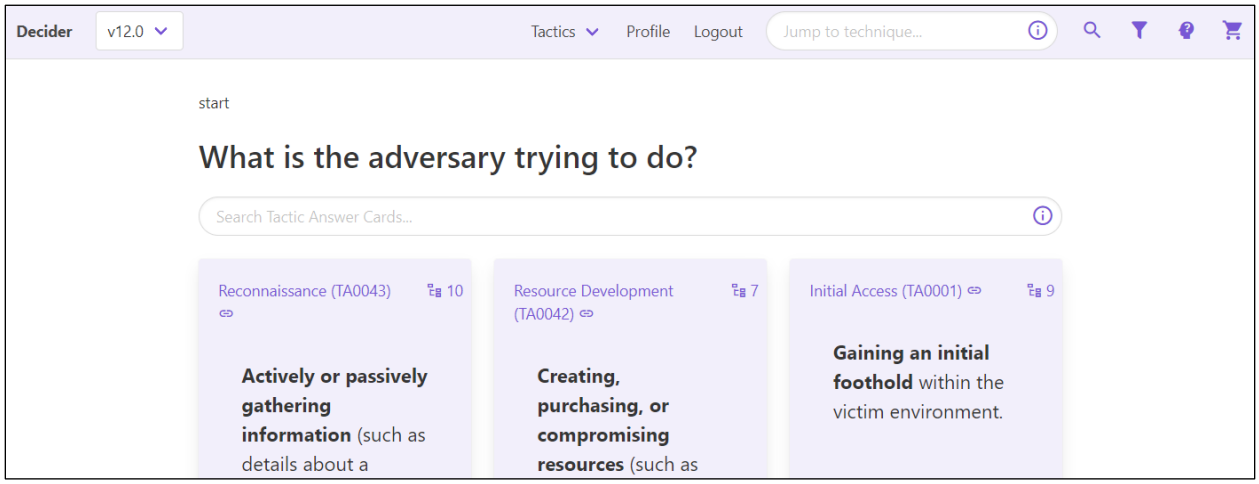


Screenshots for initially accessing the successfully deployed Decider application

1. The log-in screen to Decider.



2. The user interface after logging into the Decider application.



Avoiding "sudo pip" – Python Environment Best-Practices

Preface

`sudo pip` was originally used as a means of side-stepping Pip's caching of packages in the home directory – which fails when paired with the `--no-create-home` option of a cut-back application-dedicated user.

This, however, is a bad practice – that litters system packages and makes removal/upgrade of packages in the future more cumbersome.

The Solution

- Take ownership of `/etc/decider` *(using the account performing the installation)*
- Create a virtual environment *(`venv`, `virtualenv`, `pyenv`, et cetera)* and `source` it
- `pip install -r` requirements into the virtual environment
- Change ownership *(`chown`)* of `/etc/decider` back to the decider application account
- Configure uwsgi.ini to use this virtual environment *(below)*

```
[uwsgi]

virtualenv = /etc/decider/decider_tool/venv

chdir = /etc/decider/decider_tool
module = decider:app
master = true
processes = 5
pyargv = --config ProductionConfig
```

```
shared-socket = 0.0.0.0:443
uid = decider
gid = decider
https =
=0,/etc/decider/decider_tool/app/utils/certs/decider.crt,/etc/decider/decider_tool/app/utils/c
erts/decider.key
enable-threads = true
```