# 1. General scheme

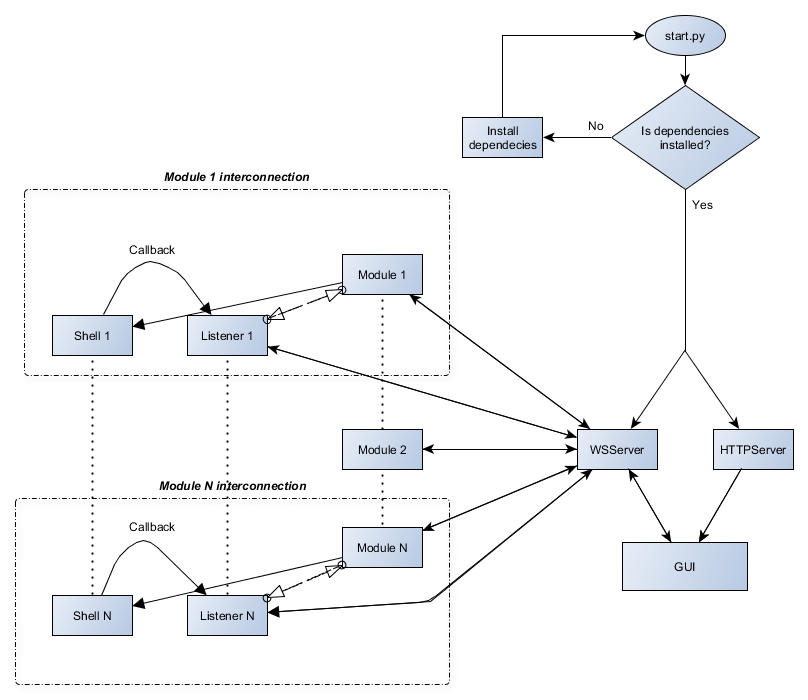


Рис.1.1. Структурная схема фрейморка

При запуске фреймворка выполняется проверка на наличие установленных сторонних модулей python. Если не хватает модуля, то фреймворк автоматически устанавливает его из папки «3rdPartyTools». При наличии всех модулей запускаются: вебсокет сервер(WSServer), который, по сути, является ядром и HTTPServer, который отвечает за отображение ГУИ. В консоль выводится информация (хост и порт) для подключения к ГУИ через веб-браузер.

## 1.1.Система команд.

Websocket сервер слушает определенный порт. К нему могут подключаться клиенты по протоколу websocket. В роли клиентов выступают все сущности фреймворка(кроме самого сервера), такие как: эксплоиты, листенеры и ГУИ. Общий вид запроса имеет вид объекта(словаря) со следующей структурой:

**{**

**“command”: “some\_command”,**

**“args”: {**

**“arg1”: value,**

**“arg2”: value2**

**}**

**}**

Далее этот объект сериализуется в JSON. Сервер имеет определенный список команд. JSON – строка отправляется на сервер. Далее происходит парсинг JSON->dict и появляется словарь с обязательными ключами “***command***” и “***args***”. В “***commands***” находится команда для выполнения на сервере. В зависимости от команды сервер выполняет разные функции, аргументами которых является значение “***args***”, а также объект ***request,*** позволяющий отправить ответ клиенту, вызвавшему текущую команду. Поэтому общий вид функции обработчика команд выглядит так:

**1: def some\_command\_handler(self, args, request):**

**2: a = args[‘a’]**

**3: b = args[‘b’]**

**4: result = self.do\_something(a, b)**

**5: request.send\_message(json.dumps(result))**

*Пояснение*: в строках 2,3 происходит присваивание значений для ключей ‘a’и ‘b’ в словаре args. Обычно перед присваиванием делается проверка на существование ключа. Далее полученные значения используются в качестве аргументов для какой-либо функции, которая может возвращать какое-либо значение. Потом это значение сериализуется в json и отправляется обратно клиенту, который тоже может иметь свой обработчик команд (с подобной структурой).

# 2.Взаимодействие ГУИ – СЕРВЕР

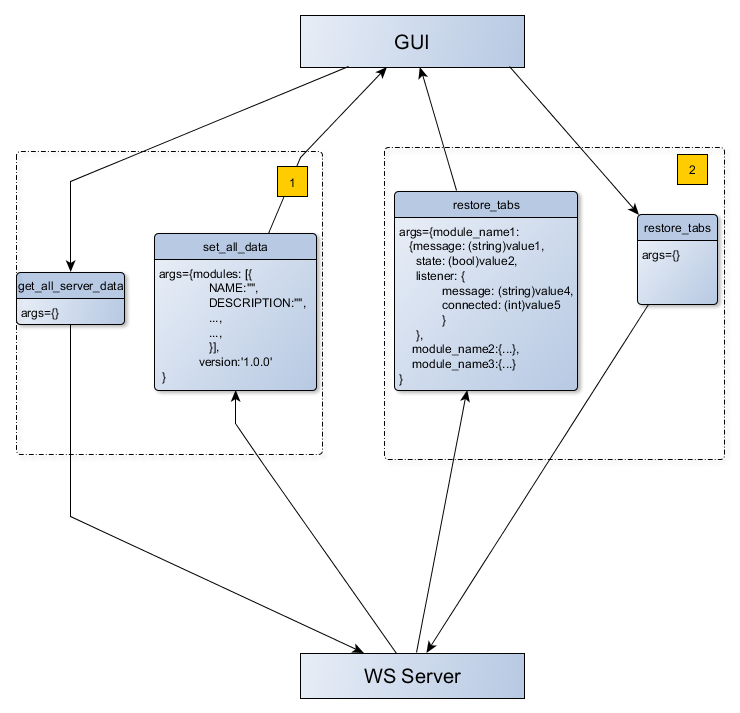


Рис 2.1. Общая схема инициализации ГУИ.

Зайдя в веб-браузере по адресу подключения к ГУИ, загружаются скрипты и контент ГУИ. После загрузки контента, срабатывает событие, по которому ГУИ через websocket подключается к websocket серверу. После подключения отправляется первая команда: «**get\_all\_server\_data**». Получив эту команду сервер собирает данные модулей, которые хранятся в словаре **INFO** каждого модуля. Происходит формирование древа модулей, если в **INFO** есть **ключ** **PATH** со значением пути к модулю. Также из главного файла **start.py** берется текущая версия фреймворка. Все эти данные отправляются обратно ГУИ. Формируется модель данных, с помощью которой ГУИ отображает модули с правильными именами, описанием и т.д. в виде дерева. После этого сразу же отправляется запрос «**restore\_tabs**». С помощью которого сервер загружает информацию о запущенных модулях, листенерах и сообщения от них.

## 2.1. Интерфейс

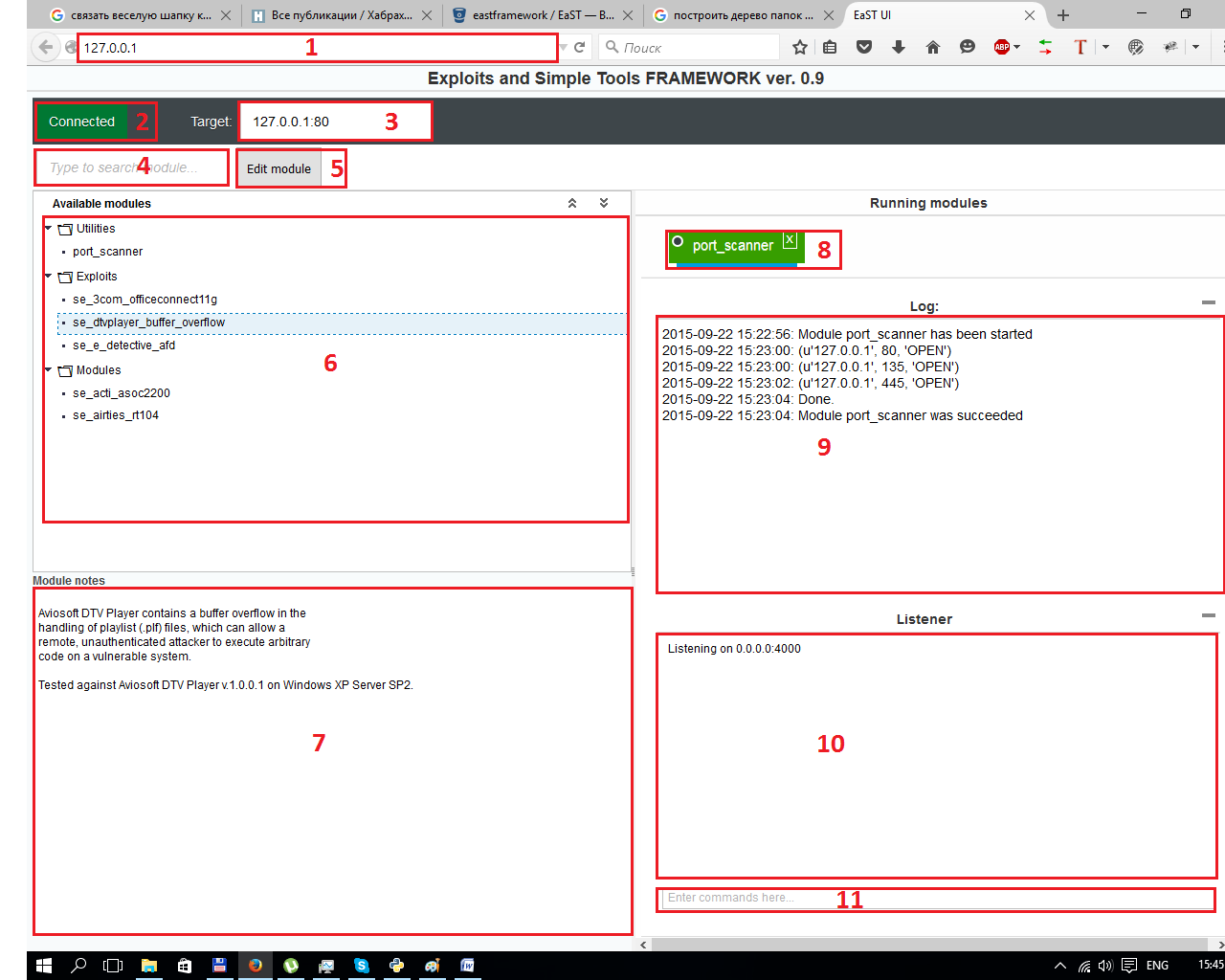
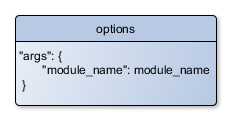


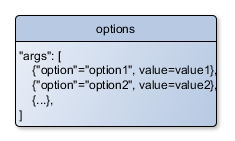
Рис. 2.2. GUI. 1) адрес подключения; 2) кнопка для переподключения(отображает текущий статус соединения); 3) поле для ввода хоста и порта, которые будут использоваться при запуске модуля по умолчанию; 4) поле для поиска модуля по частичному совпадению; 5) кнопка, запускающая редактор выбранного в дереве модуля; 6) древо модулей; 7) информация о выбранном модуле; 8) вкладка модуля(показывает цветом статус модуля); 9) сообщения модуля; 10) сообщения из листенера; 11) поле для отправки команд листенеру.

## 2.2. Запуск эксплойтов.

По двойному клику на выбранном модуле в древе модулей ГУИ отправляет серверу запрос на получение доступных опций для данного модуля. Запрос выглядит следующим образом:



, где module\_name – имя модуля, для которого необходимо получить опции. Сервер обрабатывает запрос и возвращает объект, содержащий словарь со следующей структурой:



Далее, с помощью этого объекта, ГУИ отрисовывает диалог с опциями модуля (рис. 2.3).

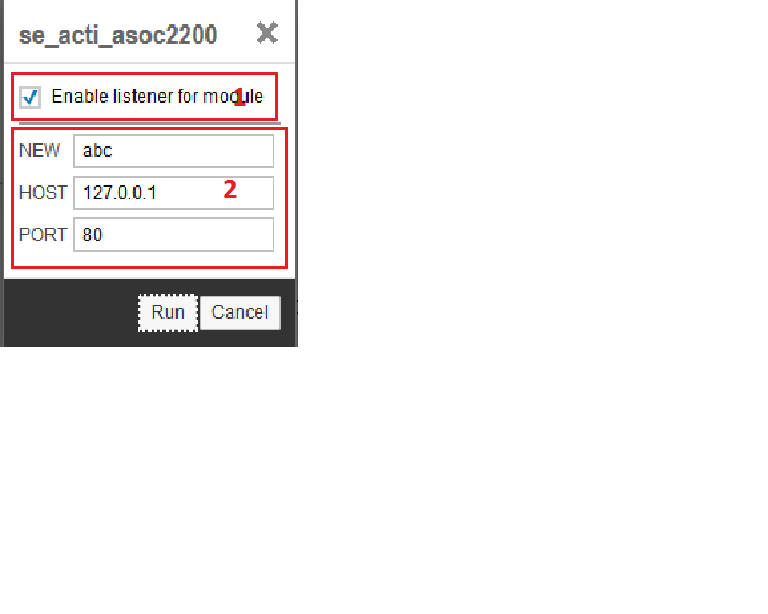
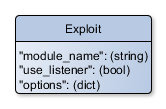


Рис. 2.3. Диалог запуска модуля с выбранными опциями. 1) чекбокс, включающий использование листенера для выбранного модуля; 2) контролы, сформировавшиеся из объекта с опциями.

Существует выбор типа опций. Это помогает создавать более информативные диалоги запроса опций. *Подробней в разделе для разработчиков.*

По нажатию на кнопку **[Run]** ГУИ оправляет команду с параметрами:



, где module\_name – имя запускаемого модуля, а isListenerEnabled – булевская переменная указывающая включать или не включать листенер для данного модуля, options – ассоциативный массив, ключами которого являются названия опций модуля, а значениями являются значения полей, заполненных в GUI(рис.2.3.).

Сервер запускает указанный модуль как отдельный процесс, а если необходим листенер, то и его тоже. Опции хранятся в объекте запущенного процесса. Потом модуль будет иметь доступ к этим опциям(см*. раздел для разработчиков*).

Для каждого запущенного модуля создается собственная структура данных(рис 2.4). Доступ к этой структуре осуществляется с помощью имени запущенного модуля или с помощью PID запущенного модуля:

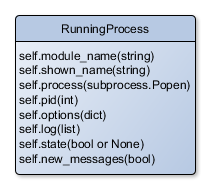
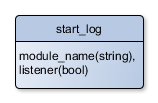


Рис. 2.4. Модель запущенного процесса.

На рисунке 2.4.:

* ***module\_name*** – уникальное имя модуля, которое сервер дает каждому запущенному модулю. Для уникальности используется суффикс «(n)», где n – обычно равно количеству одинаковых запущенных модулей.
* ***shown\_name*** – оригинальное название модуля.
* ***process*** – экземпляр процесса, запущенного с помощью subprocess.Popen(). С помощью этого атрибута возможно завершать работу модуля или, другими словами, убивать процесс.
* ***pid*** – уникальный ID процесса.
* ***options*** – опции модуля, которые могут меняться с помощью ГУИ. Используются запущенными модулями для установки значений каких-либо параметров.
* ***log*** – структура «список». Хранит в себе экземпляры класса ModuleMessageElement, который имеет 2 атрибута: время сообщения и сообщение. Экземпляры этого класса создаются, когда модуль посылает сообщение серверу.
* ***state*** – показывает текущее состояние модуля: None – модуль выполняется, True – модуль завершился успешно, False – модуль завершился неуспешно.
* ***new\_messages*** – True – есть сообщения от этого модуля, которые еще не отправлялись в ГУИ; False – нет новых сообщений.

После запуска модуля сервер отправляет ГУИ команду:



, где module\_name – уникальное имя запускаемого модуля, а listener – переменная, отвечающая за отображения окна листенера в ГУИ.

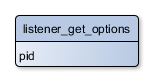
После обработки этой команды ГУИ запускает таймер, по которому серверу шлётся команда «status» с пустым объектом в качестве аргументов один раз в 300мс. Сервер берет все экземпляры запущенных процессов(рис. 2.4) и формирует из них большой словарь, первичными ключами которого являются уникальные имена модулей. Также, если модуль использует листенер, то в словаре к этому модулю дополнительно добавляются данные о его листенере.

При закрытии вкладки модуля в ГУИ серверу отправляется команда «kill\_process» c аргументом «module\_name», где module\_name – уникальное имя убиваемого модуля. Сервер ищет структуру данных, связанную с этим модулем и с помощью атрибута process убивает процесс.

# 3. Взаимодействие листенер - сервер.

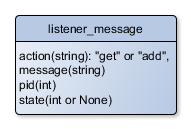
Листенер представляет собой асинхронный сокет сервер, который может слушать определенный порт и взаимодействовать с подключившимся к этому порту клиентом (шеллом). После соединения с клиентом возможна передача сообщений в любую сторону.

Листенер запускается, когда в сервер из гуи поступает команда на запуск модуля с использование листенера. Сервер автоматически устанавливает порт, который будет слушать листенер, и добавляет эту информацию к опциям модуля(см. раздел для разработчиков). Листенер, как один из вебсокет-клиентов подключается к серверу и посылает команду:



, где pid – PID листенера. На что получает ответ от сервера, в котором передаются опции данного листенера. Далее он ожидает подключение к своему порту.

Также у листенера есть универсальная команда, с помощью которой он получает сообщения от ГУИ и передает их шеллу или посылает сообщение от шелла серверу, которое затем передается ГУИ:



, где action – строка(“get” или “add”), отвечающая за то, принимать сообщение от гуи или передавать сообщение для гуи; message – сообщение для ГУИ(когда action=”add”); pid – PID листенера, state – состояние листенера(None – ждет подключение, 1 – шелл подключен, 2 – связь с шеллом разорвана).

Также листенер имеет защиту от кратковременного подключения (например, когда работает сканер портов и для проверки открытости порта пробует подключиться к нему).

# 4. Разработка модулей

Все модули наследуются от базового класса Sploit. Этот класс содержит основные методы для взаимодействия с сервером.

## 4.1.Основные правила написания модулей:

1. Модуль должен содержать словарь INFO, с обязательными ключами «NAME», «DESCRIPTION», «NOTES», в которых хранятся имя модуля, краткое описание и подробное описание соответственно.
2. Модуль может содержать словарь OPTIONS, ключами которого являются какие-либо параметры, которые можно будет показать и дать возможность изменить с помощью ГУИ.
3. Метод Sploit.args(self, OPTIONS) позволяет получить измененные при помощи ГУИ опции и использовать их в модуле. Также, если модуль запускался с включенным листенером, то можно получить порт листенера: listener\_port = Sploit.args(self, OPTIONS)[‘listener’][‘PORT’]
4. Метод self.log(msg) шлёт сообщения в ГУИ и записывает в текстовый файл лога.
5. Метод self.finish(state) используется, когда модуль завершил свою работу. Если state равно True, то это означает, что модуль выполнился успешно, если False, то модуль выполнился неуспешно

Также существует такое понятие, как тип опции. В зависимости от типа, опции в ГУИ отображаются по-разному. Для простых типов опции достаточно просто присвоить значение, например: OPTION[‘int’] = 10 или OPTION[‘bool’]=True. Есть тип, позволяющий выбирать из нескольких элементов и записывается следующим образом: OPTION[‘list’] = dict(options=[a,b,c], selected=c), где ключу «options» присваивается список из элементов, а ключу «selected» присваивается один из этих элементов, который является текущим значением для данного типа опций.

## 4.2. Вспомогательные классы

Разработчикам эксплоитов доступны несколько вспомогательных классов:

* **PortScannerMT**. Простой многопоточный сканер портов. Проверяет открыт порт или закрыт при попытке подключиться к нему.
* **Shellcodes.** Генератор различных шеллкодов под разные системы.