RobNet analysis and design

Robbe Bryssinck

# User stories

1. As a user, I can let the application try to hijack a potentially vulnerable web server by inputting the IP of said server, so that I don’t have to do as much work to gain control of the machine.
2. As a user, I can view a list of machines within my botnet, so that I have a visual insight into what machines are at my command.
3. As a user, I can add bots to my botnet, so that I have control of which bots I want to command.
4. As a user, I can remove bots from my botnet, so that I have control of which bots I do not want to command.
5. As a user, I can view what platform my bot is running, so that I can identify the platform’s characteristics easily.
6. As a user, I can view the status of a bot, so that I know which bots are online.
7. As a user, I can remotely control bots within the web app by executing commands, so that I do not have to open a separate terminal to control the machines.
8. As a user, I can send a single command to every bot in my botnet, so that I do not have to contact them individually.
9. As a user, I can see what jobs are running on my botnet, so that I can easily determine whether I can execute a new command.
10. As a user, I can cancel a job at any given time, so that I have control over when to stop a certain job.
11. As a user, I can manually add a machine to the botnet without having to use the built-in HTTP server exploit, so that I can add machines to my botnet that do not run the vulnerable HTTP server.
12. As a user, I can install bot client software provided by the application on one of my bots, so that I can use the built-in commands of the application to perform that bot.

# Non-functional requirements

1. Asynchronous, distributed messaging to control the bots within the botnet.
2. Loosely coupled front-end/back-end API, so that the front-end can easily be replaced, or another front-end application could be attached to the API.
3. Automated tests used to prove the reliability of a botnet.
4. The data must apply to GDPR.
5. The system must be robust: no messages can be missed.
6. The system must be reasonably secure, as outlined in the “Security” section.

# Distributed software architecture

## General architecture

The system is built on an event driven microservices architecture. The main reason behind this architectural design choice is scalability. The application not only has to serve potentially thousands of users at a time, but each user could also potentially have thousands of bots in their network. This means that the system needs to handle many connections at a time, therefore the application should be as connectionless as possible, and as efficient as possible. An event driven microservices architecture ensures that there are few long lived requests, and that the components and the bots are loosely coupled. Initially, this means that the clients will not be as up to date, because the client will have to use polling to get the latest state of the bots, but this could be counteracted by implementing efficient web sockets which keep the client up to date. Given that the system is event driven, the client could be updated instantaneously upon an event.

## Context

### Persistency

Microservices have their own data pool and keep each other up to date through events (eventual consistency). Because of this, each microservice can have different types of persistency technologies (SQL vs NoSQL, in memory database vs persistent database etc.).

### Communication

The microservices communicate by publishing events to an event queue. Other components can subscribe to that event queue and receive and process those events to update their data pool or perform an action (publisher/subscriber pattern). This way, the microservices do not need to know about each other (making the application loosely coupled), the transfer of information is stateless, and many services can receive the same message as efficiently as possible.

The criteria for the technology are as follows:

1. The messaging must be asynchronous as to not clog up the API responsible for delegating messages between the front-end web application and the bots themselves.
2. The messaging service must be able to filter out messages based on criteria supplied by the user, like which botnet to send a message to.
3. The receivers of the messages must be able to carry out arbitrary commands based on the contents of the message itself.

The technology used for brokering messages between services is RabbitMQ. There are other popular choices for async messaging that would have worked as well in this scenario, like Kafka. Ultimately, for this particular use case, the differences were small. I ended up going with RabbitMQ because it was simpler to implement, and there was no specific technological reason to choose Kafka, or any other messaging service, over it. Storing messages for later use is not a requirement for this application. RabbitMQ would not be the best option for persistent messages, Kafka is generally considered stronger in that area. This is not applicable here.

In terms of messaging integration patterns, the application makes use of the “Command message” pattern, used to invoke procedures within the bot clients themselves. It also makes use of the “Pipes and filters” pattern, to distribute the commands to the bots that are part of the given botnet that the user is operating on, and not to other botnets. The “Message bus” pattern is used to distribute the messages themselves, which will go through a different bus for each type of command message.

The BotnetJobs and Bots microservices will make use of RabbitMQ, along with the bot clients themselves. The messages will be used to trigger and stop jobs within the botnet and manage the bots within a botnet itself.

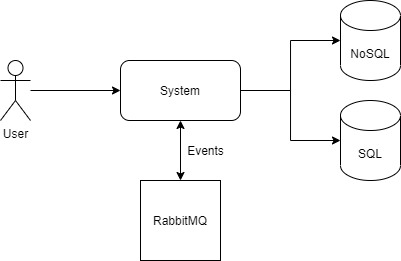


Figure 1: System context diagram

## Components

In this section, the components are defined and given a detailed description, along with visual representation of the relationship between the components.

### Bots service

The bots service runs a REST API which handles persistency of bot and botnet data. This service serves the information and the current state of the bots/botnets.

### Botnet jobs

This service provides information, manages, and executes commands on the machines within the botnet. The botnet jobs service is written in C#. This language was chosen mainly because the application needs a solid multi-threading implementation. More so, .NET has great support for load balancing, which is important for this service, since it must do a lot of tasks simultaneously. The service has its own data pool to store the jobs currently running on the bots. The database is a NoSQL database. The botnet jobs can have variable amounts and types of arguments, so a more flexible, non-relational database technology fits best.

### Web app

This service runs a front end and a back end. The back end does not do much except for serving pages. The web app relies on the APIs to handle the business logic. The front end can be used by the user to view and manage the botnet. The application is written in React. The front end will be a single page application with many different components in the form of bots and a single component displaying the state of the user’s botnet. React handles these kinds of applications well.

### Bot client

The bot client application needs to be installed on the machine of a bot. This application will handle communication with the botnet job microservice to execute a job, and update the state of the bot. The client is written in Python 3 since portability is the most important factor of a bot client. Since the botnet owner cannot open ports on their bot machines, the bot is not directly approachable from the outside. Therefore, the bot will have these two ways of communicating with the server:

1. By opening a connection to the messaging queue to receive botnet jobs and execute them.
2. By sending HTTP requests to the system to update the bot’s state.

Persistency is not an important factor for this application. The currently executed commands will be stored in memory.

### Exploit executer

This service asks for an IPv4 address, tries to exploit and backdoor the machine pointed to by the IP address, and returns the result of that action. If the exploit succeeds and a backdoor is correctly installed, the service installs the bot client software. Else, the service returns an error code, explaining what went wrong. This program is written in Python 3. Python is a popular language in the field of exploit development, since it handles binary data intuitively, it is easy to connect to servers running a plethora of services, it has great support for running subprocesses, and lastly, it has great libraries for memory corruption/exploit development (Pwntools).

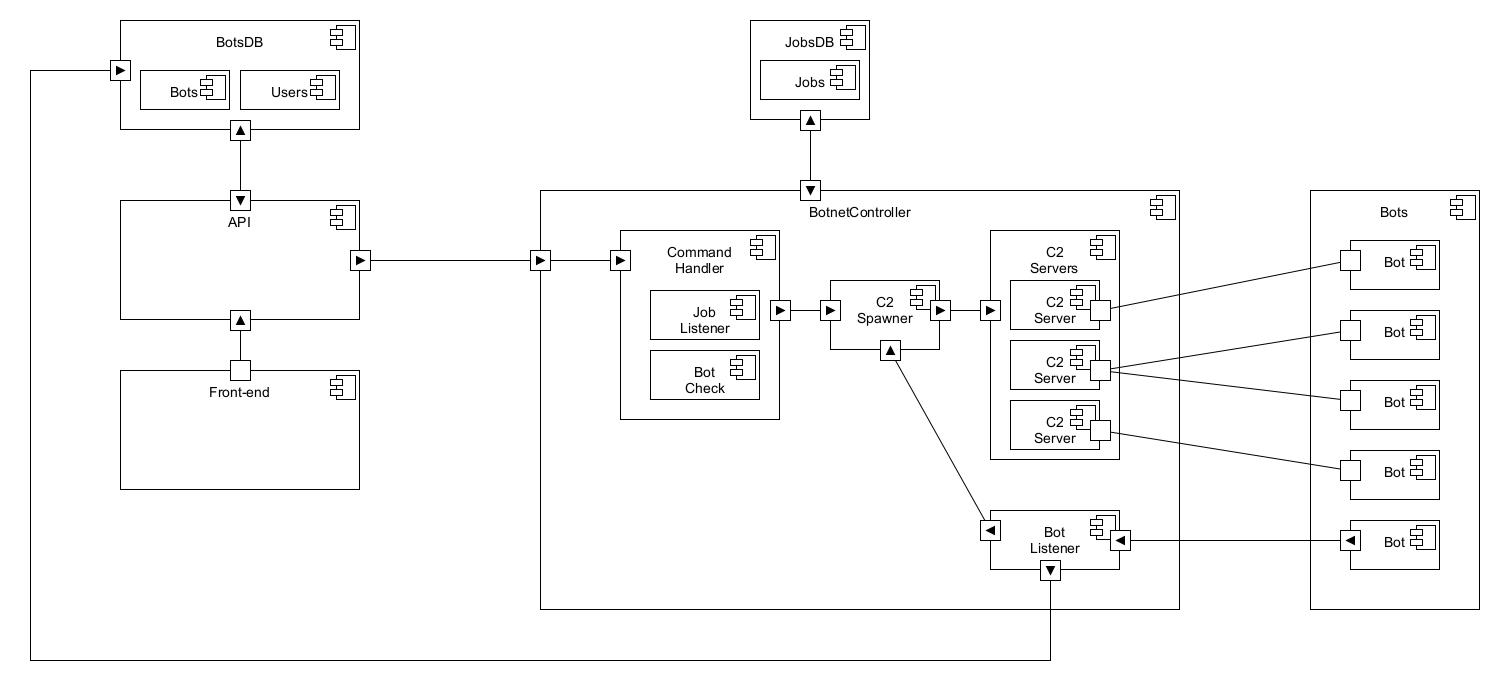


Figure 2: botnet controller architecture

## Component architecture

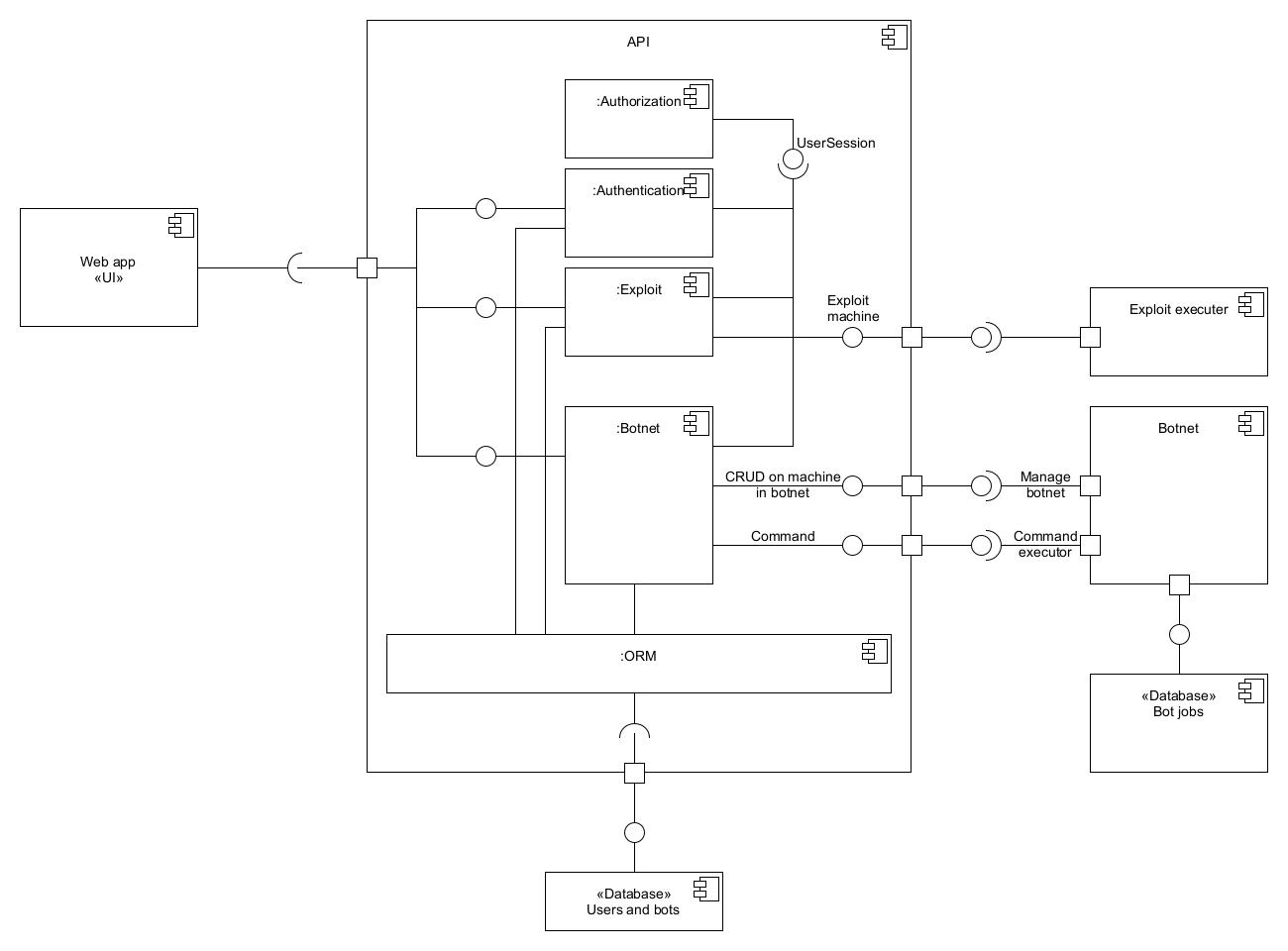


Figure 3: component diagram

## ERD

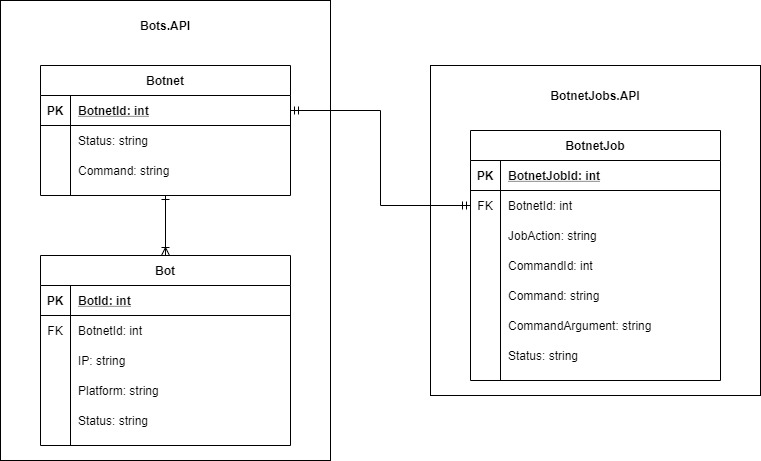


Figure 4: ERD

# Security

To ensure that the application is secure when it hits production, the application will be built with security in mind. The security of the system will be ensured in several ways:

* Static analysis tools will be used to make sure the application itself is secure.
  + Visual studio will generate warnings about potential security vulnerabilities like storing passwords in config files.
  + Github statically analyzes dependencies for both the front-end web application and the back-end API, and will warn the programmer about any outdated, vulnerable dependencies.
* Authentication and authorization will be implemented using hardened, battle-tested frameworks provided by third parties who specialize in security.
  + Identity framework will be used to restrict access and handle logins.
  + JWTs will be used to handle generation and validation of authorization tokens.
* An ORM, Entity framework, will be used. The main motivation for this is convenience and cleanliness of code, but it also prevents SQL injections.
* The programmer is informed about the most common vulnerabilities in web development (OWASP top 10).
* Automated web application vulnerability scanners can be used to check for vulnerabilities within the web application and/or the web API themselves.
* Before the application is deployed to production for its initial release, a professional pen tester should try to find vulnerabilities in the system.

## GDPR and privacy

As with any application that requires the user to register, submit information, and have that information stored by a third party (i.e. our system), the design must comply to the rules of GDPR and any other relevant privacy laws. To ensure this remains true for the entirety of the system, the system will be checked against the GDPR guidelines to see if it still applies the necessary rules. This will be done at the end of every sprint, as defined in the definition of done.

# Definition of done

The definition of done exists for two reasons:

1. It is used to ensure the quality and completeness of the deliverables on a sprint-per-sprint basis.
2. It is used to ensure that all the non-functional requirements are met.

Some of these requirements can be ensured through automated testing alone, but not all of them. Therefore, a definition of done was set up. This checklist must be checked in the last week of each sprint.

1. Run static analysis tools to ensure code quality and security.
2. All tests described in the test plan must pass.
3. The build should properly pass CI/CD.
4. Privacy rules and GDPR compliance must be checked to see if it is still valid.
5. Manual user testing must be performed to see if the system is still performant and robust.
6. The automated tests require 90% code coverage as a minimum.
7. The deliverable must contain solid proof of progress in the form of a demo or presentation.

# Test plan

To test the functionality within my distributed application, there are several types of testing systems put in place, namely:

* Unit testing. Since certain components in my system have quite a bit of logic, unit tests are important to implement to ensure that the logic behaves as expected. Used tools are XUnit for .NET components, Jest for front end testing and pytest for the Python bot.
* Integration testing. Given that the system makes use of microservices and is therefore loosely coupled, good integration tests are invaluable. The integration tests will be performed with the same technologies used in unit testing.
* Acceptance testing. The acceptance testing will be done manually. To realistically emulate a user, the acceptance testing will be done through black box testing.
* Performance testing. For the front-end, “Google Lighthouse” will be used for automated performance testing/dynamic analysis.

## Unit/integration tests

The unit and integration tests will cover most of the code, except for the services that directly interact with the RabbitMQ message broker. This has two main reasons:

1. The code is not very complex, and only really makes calls to the RabbitMQ API, which does all the heavy lifting in delivering the messages. If these services were to be tested, the tests would in practice only really cover the RabbitMQ API.
2. For these tests to pass in the CI/CD pipeline, there would need to be a separate RabbitMQ container set up. For this project, I made the decision not to implement that based on some research, given that the cost in effort does not match up to the added benefit to the project.

The automated tests will be performed on the CI pipeline.

## Acceptance tests

The acceptance tests will be performed manually by the developer before pushing out a new release. The exact tests will simply be performing the user stories, and making sure that the stories described are doable, and that there are no critical defects.

## Performance testing

The performance testing will be performed through a tool called “Lighthouse”, which is test tooling developed by Google. This tool can be used to analyze not only the performance of the front-end itself, but it can test the end-to-end performance too.