hr@wazuh.com
https://wazuh.com

Introduction

Wazuh Inc. is an American IT security company focused on Host-based Intrusion Detection Systems (HIDS). Specifically, we contribute to the world of Open Source Security developing and improving the capabilities of Wazuh software.

Below, there are some tasks that the candidate must perform. The instructions for submitting your responses can be found at the end of this document.
Thanks in advance for your time working on these tasks.

Best regards,

Wazuh, Inc.

hr@wazuh.com
https://wazuh.com

Selection tasks

Task 1: Getting familiar with Wazuh

Wazuh is a Host-based Intrusion Detection System (HIDS). Its architecture consists of the main server (manager) and an installed agent in every host to monitor.

Tasks

- Install a Wazuh manager (Linux host) and connect a Wazuh agent (Linux/Windows host).
- Install Opendistro/Kibana and connect them to the Wazuh manager.
- Generate events in the Wazuh agent and verify the generated alerts from Kibana.

Questions

- What problems did you have?
- Describe the flow from the time a logline is written to a file, to it being visible in Kibana.

Deliverables

- Installation screenshots / logs.
- Answer to the previous questions.
- Screenshots of the final result in Kibana.

Reference

- Wazuh documentation

Task 2: Creating a RESTful API using Python

Task 2.1: Implementation

Using the contents of the files tasks.json and users.json, create an API using Flask that meets the following requirements:

- **Endpoint `GET /tasks`.** Retrieves all tasks listed on the tasks.json file.

  Parameters:

  - completed : boolean - Optional parameter. Filter tasks based on their attribute "completed". When not specified, returns all tasks.
  - title: string - Optional parameter. Displays tasks containing the provided string in their title. Defaults to an empty string.

  Response:

```
{
  total_items: number,
  data: {
    user_id: number,
    id: number,
    title: string,
    completed: boolean
  }[]
}
```

- **Endpoint `GET /tasks/:id`.** Retrieves information from a single task.

  Response:

```
{
  user_id: number,
  id: number,
  title: string,
  completed: boolean
}
```

- **Endpoint `GET /users`.** Retrieves all users listed on the users.json file.

  Response:

```
{
  total_items: number,
  data: {
    id: number,
    name: string,
    username: string,
    email: string,
    address: {
      street: string,
      suite: string,
      city: string,
      zipcode: string,
      geo: {
        lat: number,
        lng: number
      }
    },
    phone: string,
    website: string,
    company: {
      name: string,
      catchPhrase: string,
      bs: string
    }
  }[]
}
```

- **Endpoint `GET /users/:user_id`.** Retrieves information from a single user.

  Response:

```
{
  id: number,
  name: string,
  username: string,
  email: string,
  address: {
```

```
    street: string,
    suite: string,
    city: string,
    zipcode: string,
    geo: {
      lat: number,
      lng: number
    }
  },
  phone: string,
  website: string,
  company: {
    name: string,
    catchPhrase: string,
    bs: string
  }
}
```

- **Endpoint `GET /users/:user_id/tasks.`** Retrieves all tasks from the specified user.

  Parameters:

  - completed : boolean - Optional parameter. Filter tasks based on their attribute “completed”. When not specified, returns all tasks.
  - title: string - Optional parameter. Displays tasks containing the provided string in their title. Defaults to an empty string.

  Response:

```
{
  total_items: number,
  data: {
    user_id: number,
    id: number,
    title: string,
    completed: boolean
  }[]
```

```
}
```

You can additionally include any other endpoints that you think would add value to the API, such as POST, PATCH and DELETE endpoints to perform updates on the dataset.

Questions

- What problems did you have?
- Did you make any changes to the specified API design?
- Explain any additional endpoints that you may have added, and document their parameters and response format.

Deliverables

- Answers to the previous questions on PDF.
- API specification. Some standard format like Swagger is preferred.
- Link to Github repository containing the API code.
- Instructions of use. It must explain at least how to run the API locally.

Reference

- Flask
- Swagger

Task 3: Creating a front-end using React

Task 3.1: Implementation

Create an application using React and Typescript that consumes the previous API and has the following elements:

- Navigation menu.
- Users view: List all users and display the most important information to easily identify them. When you click on a user, the full details should be displayed.
- Tasks view: List all tasks. Must include:
    - Search bar.
    - Filters: Filter by the “completed” attribute, or by the user.
    - Pagination.

Task 3.2: Automated tests

Implement at least five automated tests.

Questions

- What problems did you have?
- What testing tools did you use and why did you choose them?

Deliverables

- Answers to the previous questions on PDF.
- Link to Github repository containing the app's code.
- Instructions of use. It must explain at least how to run the app locally and how to run the tests.

Reference

- React
- Typescript

Task 4: Deploying the service

Deploy the full application, both backend and frontend, so it can be used live. You can use any deployment tool, such as Netlify, Heroku or AWS Amplify.

Questions

- What problems did you have?
- Which tool did you use and why?

Deliverables

- Answers to the previous questions on PDF.
- Screenshots/logs of the deployment process.
- Link to the live version of the application.

Reference

- Netlify.
- Heroku.
- AWS Amplify.

Form of delivery

- PDF containing each task answer, screenshots and any other information you consider.
- ZIP file containing all the deliverables from each task.
- All files must be sent to hr@wazuh.com

Due date

Your completed tasks are due one week from the date of receipt of this document.