Bastion

A THCon 2022 challenge

Alexis Carn
Gautier Ben Aïm

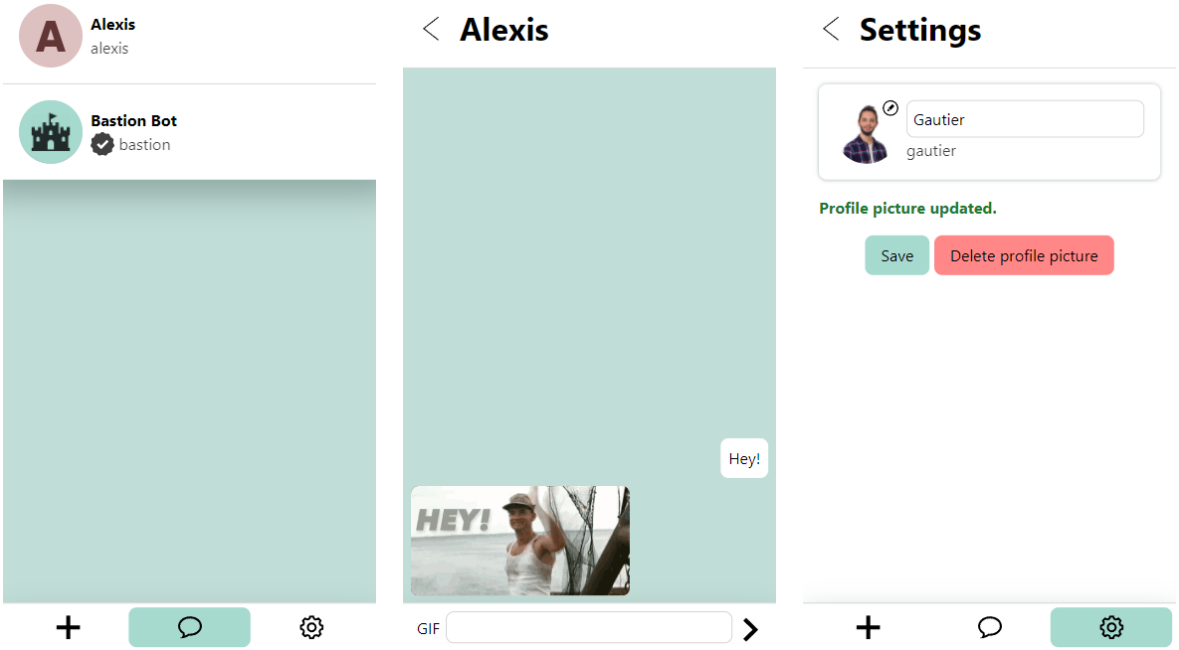
TLS-SEC 2022

What is Bastion?

Bastion is the name of a seemingly secure messaging app, developed as a hacking challenge for the Toulouse Hacking Convention. Its features and design are inspired by many modern messaging applications, such as Telegram and Facebook Messenger. However, many implementation flaws and oversights make *Bastion* deeply insecure, so that CTF competitors can break it. *Bastion* is in the Web category, meaning it is a remote challenge with both a backend and a frontend.

Features

Bastion has the most basic features expected from a real-time messaging application: registration, contact list, automatic refresh, GIFs, changing one's profile picture…



Contact list | Chat | Profile settings

Table 1. Screenshots of Bastion features.

When users open Bastion for the first time, they are asked for a unique username and a display name. A robot named *Bastion Bot* sends a greeting message to all new users. The navigation bar at the bottom allows users to navigate between three pages: "New Conversation", the contact list and "Settings". The settings page allows users to change their display name and their profile picture.

We developed all these features to make the challenge as realistic as possible. The whole application has a very large attack surface, making it very hard to find vulnerabilities; we consider the challenge as difficult to very difficult.

Architecture

Bastion is made of both a frontend, the user interface, and a backend, the server API (application programming interface).

- The frontend is written in Svelte, a JavaScript framework similar to React and Vue, which allows developers to quickly create animated graphical interfaces.
- The backend is made with Nest, a Node.js framework designed for building server-side applications. Nest can be used to create both a REST APIs and a WebSocket API, making it a suitable tool to create real-time Node.js applications.
- The database behind the backend is a PostgreSQL instance, interfaced by Prisma, a Node.js ORM (Object–relational mapping). Prisma is an abstraction layer that allows developers to query databases without writing a single line of SQL.

We picked this stack because it allowed us to focus on development, not setup. The whole application has been designed to run in a single Docker container, like all THCon challenges.[1]

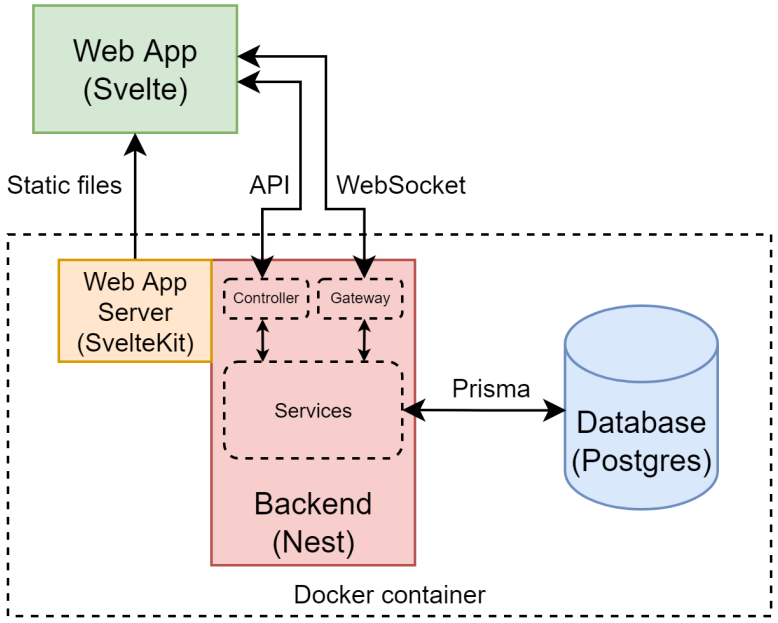


Figure 1. The architecture of Bastion.

The figure above synthesizes the whole architecture of *Bastion*. SvelteKit serves the static files of the web application (HTML, CSS and JavaScript). The web application then connects to the Nest application, either with a REST API or WebSockets, depending on the nature of the messages to exchange. WebSocket is a technology that allows both the client and the server to send a message at any time, which is not possible with a regular REST API.

The database runs in the container and is not exposed to the outside. Only the backend is exposed, and attackers can only attack the backend, which contains several vulnerabilities detailed below.

Complete solution

Bastion is a three-stage challenge, meaning that competitors can acquire up to three flags. A stage consists in a series of common pentest (short for penetration test) tasks, including but not limited to, impersonation, fuzzing, reverse engineering and brute-force. The stages are of increasing difficulty, and while completing a stage is not necessary to start the next one, the competitors are guided to do the three stages in a specific order.

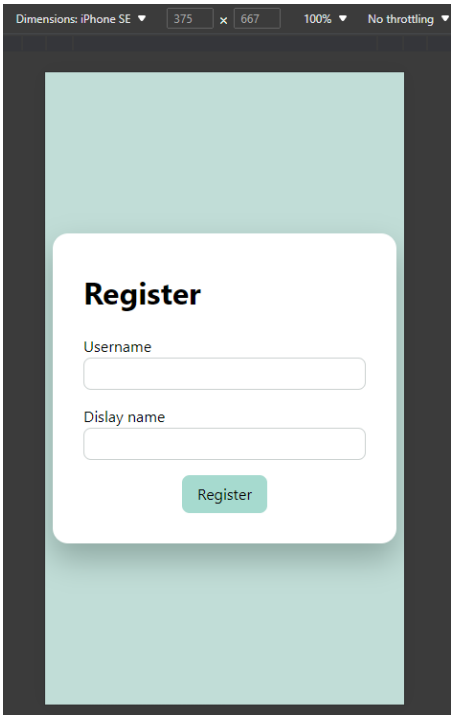
First flag

Walk through

When attackers try to access the application, they are greeted with an error message prompting them to access it from their phone:

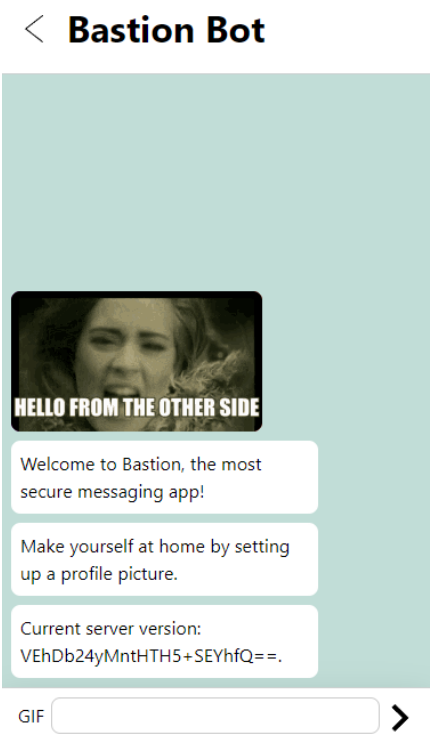
> **Please use Bastion on your phone.**
>
> The desktop version is currently in pre-alpha-insider-limited-turbo-private phase.

There are many ways to circumvent this restriction, one of them being the “Device emulation mode” of most modern browsers. (`F12` then `Ctrl`+`Shift`+`M` on Chromiums)



Once the attacker has created an account, they are redirected to the homepage (`/register` → `/`), featuring a list of all their recent conversations. At this point, there

should only be one person here, *Bastion Bot*, that sends greeting messages to every new user.



The bot sends a message reading *Current server version: VEhDb24yMntHTH5+SEYhfQ==.* This weird and seemingly random version number is a base64-encoded string containing the first flag: **THCon22{GL~~HF!}**.

Implementation details

This vulnerability is a common example of “User-agent spoofing”:[2]

```
request.headers.get('User-Agent')?.toLowerCase().includes('mobile')
```

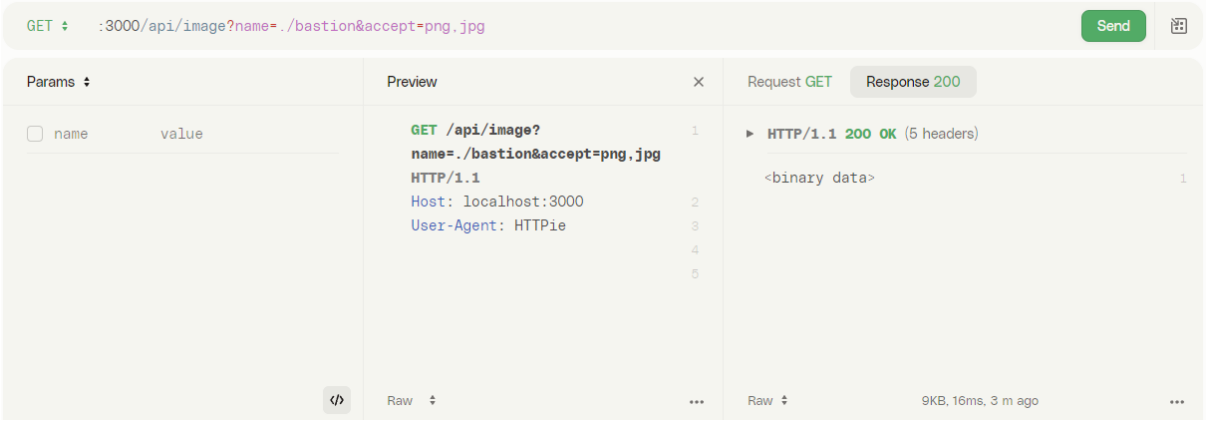
This check is not reliable because the User-Agent header is user-provided, and thus completely editable by an attacker.

Second flag

Walk through

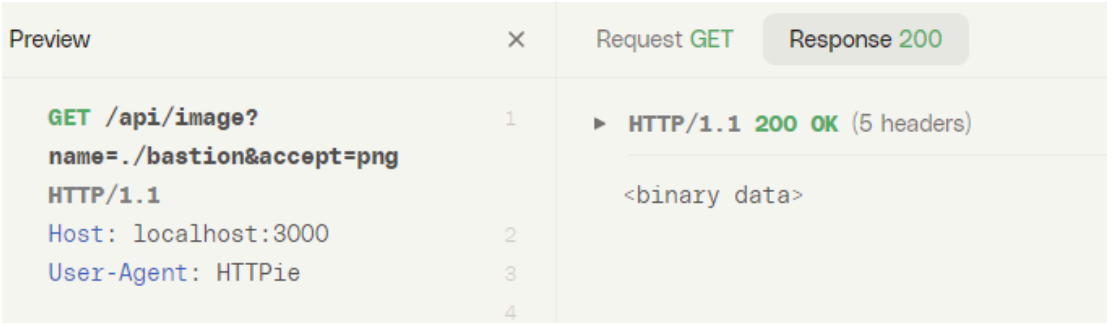
To get the second flag, attackers have to find two different vulnerabilities in no particular order. They are both located in the profile picture module.

The first vulnerability is an arbitrary file download: the API endpoint `/api/image` does not properly sanitize filenames, and allows the characters `.` and `/`. For example, the requests `?name=bastion` and `?name=./bastion` give the same image.

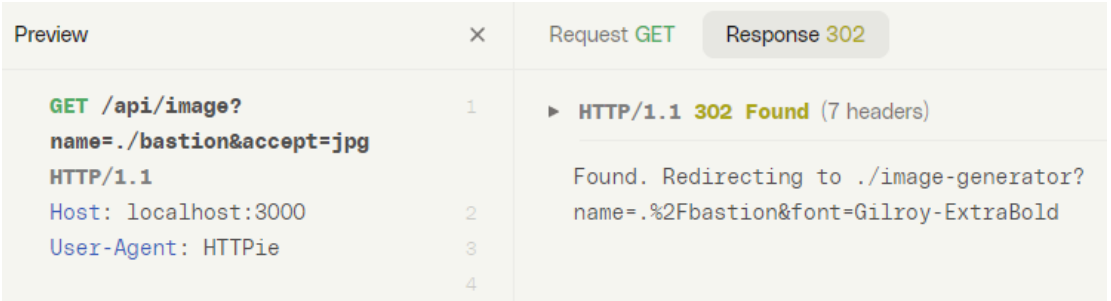


We can also notice that the `accept` parameter is the extension of the underlying stored file:

- With `accept=png` the image is send to the user

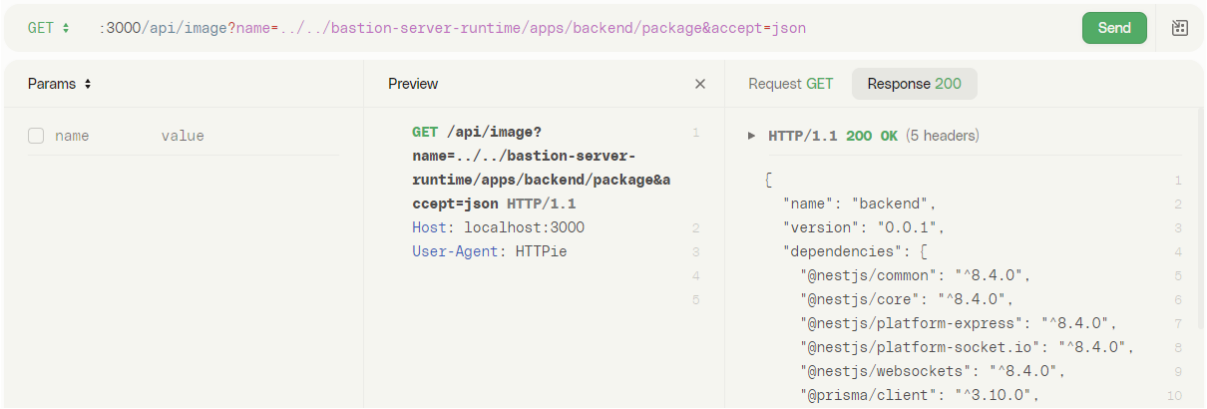


- With `accept=jpg` the user is redirected to the image generator



When no image is found, the endpoint redirects to `/api/image-generator`, which is responsible for creating the big pastel letters. This endpoint takes two parameters: a name and a font. When given another font name than the one already set, the server raises a *400 Bad Request* with an error message containing a full path to where the font should be:

The `path` field gives precious details about where the server source code is, and the attacker can start downloading it. A common resource to find in a Node.js project is a file named `package.json`, which contains all the metadata of a project. Attackers now have to find the right path to download the `package.json` file, for example `../../bastion-server-runtime/apps/backend/package.json`:



There are two `package.json` attackers can find because of the project structure. Both contain a `main` field to give attackers the next file to download. No matter which `package.json` the attackers find, the entrypoint of the backend application is `/bastion-server-runtime/apps/backend/build/app.module.js`:

```
import { AppController } from './app.controller.js';
import { ImageService } from './image/image.service.js';
import { MessageGateway } from './message/message.gateway.js';
import { MessageService } from './message/message.service.js';
import { PrismaService } from './prisma.service.js';
import { UserService } from './user/user.service.js';
```

Here is an extract of this file, showing all the imported files. Attackers will have to download and reverse this code to find the second flag.

The second flag is hidden in the code responsible for the *Bastion Bot* answers:

```
if (Buffer.from(message.body).toString('base64') ===
    'VEhDb24yMntCQURfSEFCSVRTX0ZFQVRVUklOR19CTVRIfQ==') {
    return this.createMessage({
        fromId: 1,
        toId: message.fromId,
        body: "That's a nice flag you have there!",
    });
}
```

The flag is a base64 string, **THCon22{BAD_HABITS_FEATURING_BMTH}**.

Implementation details

The arbitrary file download is caused by an unsanitized user input:

```
const filename = `${storage}/${name}.${ext}`
if (existsSync(filename))
    return new StreamableFile(createReadStream(filename))
```

The stack-trace disclosure is caused by a neglectful exception propagation:

```
try {
    registerFont(`${__dirname}/../../resources/${font}.ttf`)
} catch (error) {
    throw new BadRequestException(error)
}
```

Third flag

To find the third flag, attackers will have to perform two different kinds of brute force attacks.

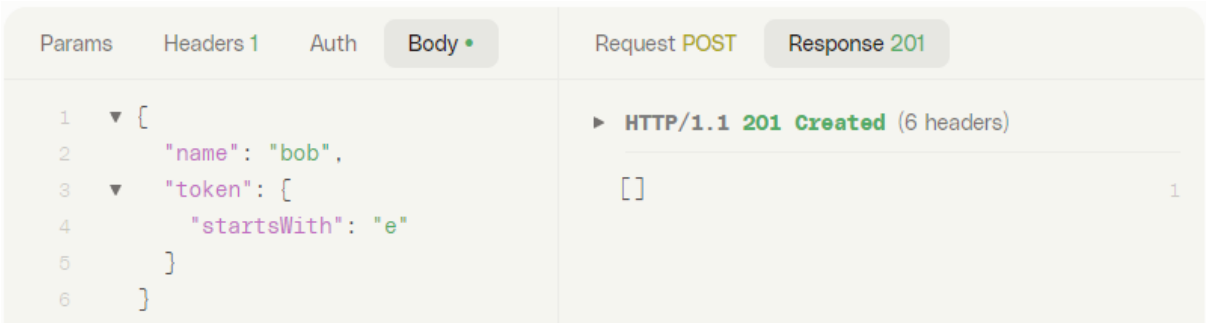
The first one is about breaking 30 bits of entropy to get the admin property on their account:

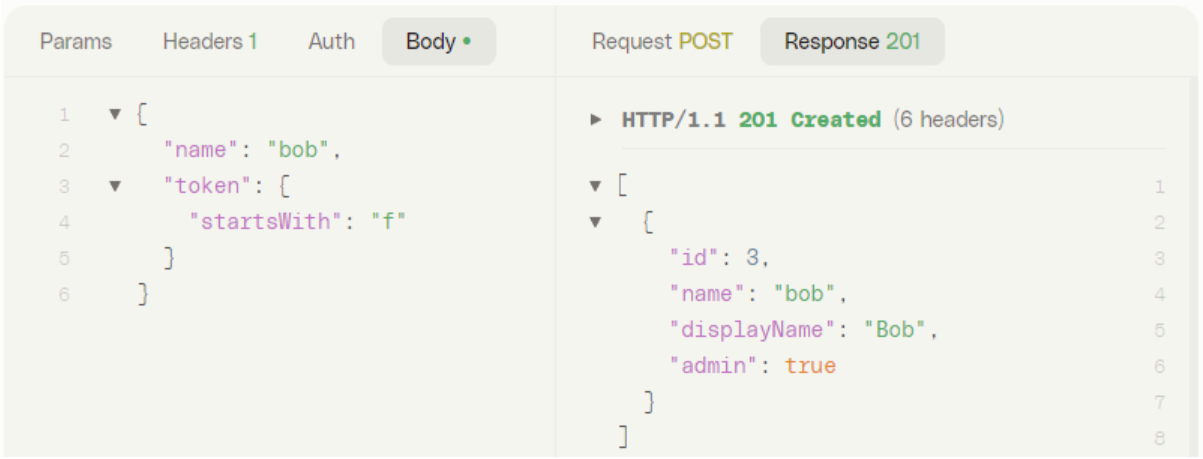
```
async promoteUser({ id, name, token }, key) {
    if (!createHash('md5')
        .update(`${name}/${token}/${key}`)
        .digest('base64')
        .startsWith('admin'))
        throw new UnauthorizedException();
    return this.prismaService.user.update({
        data: { admin: true },
        where: { id },
    });
}
```

This has to be done by pure brute force since no rainbow table can be used. Attackers have to find `key` such as `base64(md5(`${name}/${token}/${key}`))` starts with admin. On an average computer, this should take less than one hour, but attackers who parallelize this task properly among CPU cores and team members should be able to do it in a few

minutes. A collision calculator can be used to compute the probability of breaking the hash before a given amount of time.[3]

The second brute force attack is about finding Alice's or Bob's token byte by byte ; once an attacker is an admin, they can use the `/api/find-users` endpoint (and the `/admin` page as an Easter egg). The request body is not properly sanitized and passed as is to `PrismaClient.user.findMany`, allowing attackers to use the powerful query API offered by Prisma:





Leveraging the `startsWith` option, attackers can guess the token byte by byte.

The third and last flag is in the conversation between Alice and Bob:

The flag is **THCon22{C'est_la_vie_-_Weathers}**.

Conclusion

Bastion is a very difficult challenge, because of both its wide attack surface and the large range of the vulnerabilities involved. We tried our best to make the challenge fun and enjoyable, by mixing different types of vulnerabilities and keeping the guesswork to a minimum, but we also tried to make it as realistic as possible by properly implementing all the basic features messaging applications have.

The whole challenge will be open sourced after the end of the Toulouse Hacking Convention, on April 17th 2022.

References

[1]: CTF challenge expectations https://thcon.party/ctf/
[2]: https://en.wikipedia.org/wiki/User_agent#User_agent_spoofing
[3]: We created one at https://www.desmos.com/calculator/hqgcfbzjjy