DUCTF 2024: Prisoner Processor

Vie

DUCTF 2024 has concluded this summer, and I decided to take a look at it with Maple Bacon and solve a few challenges. This is specifically a writeup for "Prisoner-Processor", the hardest web challenge available. My teammate Angus and I solved this together. It was a great challenge and ventured into alot of interesting things about Bun and TypeScript, so kudos to the authors for making such an interesting challenge!

SPECS

start.sh

The start.sh script is more sus than other conventional start.sh scripts as it will attempt to restart the server 5 times, then copy a backup to the app folder if it couldn't restart, and retry again in a perpetual loop.

It's technically not *wrong* to have a start.sh auto-restart a server if it goes down, but there's a few ways to do that that's considered "cleaner". On the infrastructural side, enabling `restart: always` in the docker-compose.yml file does the same thing, with the added benefit of restarting the whole container, on a clean serving of the app data (no need to copy a backup). Basically, having the app do the restarts in the challenge itself seems weird. This is important for later.

index.ts

Some ground truths about the challenge:

- It runs on TypeScript, and the runtime is Bun
- It's a prisoner?? database with an "examples" endpoint that gives you a JSON of all of Oceania's prisoners and their crimes. Each prisoner JSON has properties that are prefixed with `signed.`, and these signed properties are used for signature generation.
- The app can accept submitted JSON, validate its signature, then use `stringify` from a yaml module to convert into yaml. The JSON-to-YAML is yeeted into a new file, and the app checks for the existence of an `outputPrefix` in the JSON data. If it exists, it's put into the filename as `outputPrefix-<random sequence of chars>.yaml` and saved into a folder called `/app-data/`. That file will not be used anywhere else in the app after it's created.

Exploits

Bypassing Signature Validation with Top-level Prototype Pollution

In `/src/index.ts`:

The above functions are responsible for the signature checking. The signature is used to stop us from creating our own JSON objects. The signature is computed with a secure I'm-not-gonna-try-to-bruteforce method and is based on all properties in the JSON whose key value starts with `signed.`. If we modify the signed properties in any way, the signatures won't match and the app will reject our JSON. However, the functions don't check // don't care for properties in the JSON whose key value *don't* start with `signed.`, so we can freely add arbitrary properties as long as their key doesn't start with `signed.`, and those additional properties won't affect the signature. We still wouldn't be able to make a JSON from scratch due to the I'm-not-gonna-try-to-bruteforce method to make a new signature, but we can copy one from the "examples" endpoint and reuse its signature.

An example JSON file looks like this:

The function `getSignedData` will be given JSON like this and look for any keys using `Object.entries` that start with `signed.`, and assign it to an empty object. That object, after having been given all the `signed.` properties, is then returned.

The assigning is vulnerable to top-level prototype pollution:

`signedParams[keyName] = data[param];` allows you to assign arbitrary properties but only one-layer down. This is not a recursive assignment, so you can't give whole objects to the data. So we can define a `signed.__proto__` as a key and add additional properties in there - and because `Object.entries()` doesn't go up the prototype chain during the signature check - but the `for (const param in data)` loop to assign props DOES go up the prototype chain, it will be added to the object but it won't violate the signature check. This function is used later on in this snippet of code:

As in, the data is stored in the variable `signedData`. Because the pollution is shallow, we will need to look for properties that the code is checking in the object itself for, AKA, is `signedData` expected to have any other properties that were not previously defined? The answer is yes: `const outputPrefix = z.string().parse(signedData.outputPrefix ?? "prisoner")`. `outputPrefix` will be "prisoner" if it wasn't defined, otherwise we have full control over that value if we do define it. This is useful for us because `outputPrefix` is next used to create the filename in `outputFile`, which is created as a template string with `outputPrefix` prepended to the rest of the filename. We have, ostensibly, semi-control over the filename. We can certainly LFI and traverse the directories by making `outputPrefix` a bunch of `../` characters, but before we get into how we're gonna use it, we gotta review the other bugs first.

Nullbytes

The function `convertJsonToYaml` is defined here. Remember that the app doesn't actually use the yaml file anywhere, it's just stored somewhere.

Tracing the logic from above, the arg `outputFileString` will be the `outputFile` that we can control the prefix of. First, it's checked to see if we don't have any banned words:

Then, Bun runtime will create a "file" object to perform file I/O stuff on, and the app checks if the file object points to something that already exists. This barricades us from doing too much directory traversal with `outputPrefix`, because we can't use any words in `BANNED_STRINGS` and we also can't overwrite some other file. I also didn't really touch on the fact that the file has a bunch of random characters appended to it, and finally, the filetype `.yaml` added as well, so we can only ever define yaml files, anyway. This situation seems like a good case for null-byte termination. Luckily for us, `Bun.file()` will seemingly(?) not notice the nullbyte, so something like `viewie.txt\00<randomcharacters>.yaml` will *look like* `viewie.txt\00<randomcharacters>.yaml`. But when `Bun.write()` is called, something like `viewie.txt\00<randomcharacters>.yaml` is instead seen as `viewie.txt` as the write operation implicitly terminates at the null byte. I may be wrong on who's recognizing the nullbyte, but the point is, we can inject a nullbyte into our `outputPrefix` to remove the `<randomcharacters>.yaml` suffix to the filename, *AND* bypass the `existsSync` check so we can overwrite important files, and they can be of any extension (not just yaml).

So what about the `BANNED_STRINGS` list?

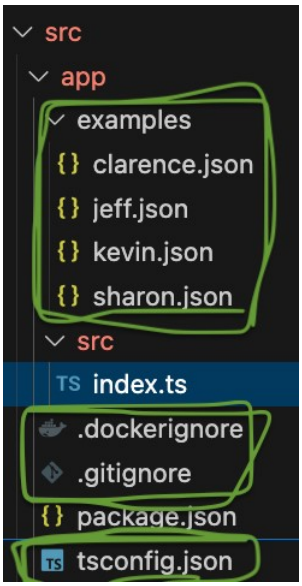
"Unfortunately, tsconfig.json isn't real JSON"

So, to recap:

1. We can add arbitrary properties to a JSON and include a `__proto__` key which contains `outputPrefix`.
2. You can make `outputPrefix` navigate to any file and overwrite it, and you just need to add a null byte to cancel out the random garbage appended to it.

So we have the ability to write to any writable directory in the app. The final few caveats are the fact that while we control the filename and write to anywhere, remember the contents are JSON that has been YAML-ified. So even if we rename our file to `somethingelse.js`, the file contents will still be a YAML file of prisoner data, NOT js.

Looking at the `BANNED_WORDS` list and cross-referencing with the directory structure of the challenge, we can't really go anywhere except for the files in green.



Actually just kidding, all those files are in `/app`, which is part of the `BANNED_WORDS` list. Shame.

Good thing that `proc/self/cwd` is a symlink to `/app`.

```
$ ls -la /proc/self/cwd
lrwxrwxrwx 1 bun bun 0 Jul  7 15:51 /proc/self/cwd -> /app
```

Minor tangent aside, the only remotely interesting file in there to use, is `tsconfig.json`. Right now, it doesn't have much to it. But when looking through the documentation for it, we came across an interesting property, path re-mapping:

> the Bun runtime will re-map import paths according to the compilerOptions.paths field in tsconfig.json.

More specifically, according to the official TypeScript documentation:

> A series of entries which re-map imports to lookup locations relative to the baseUrl if set, or to the tsconfig file itself otherwise. There is a larger coverage of paths in the moduleResolution reference page. paths lets you declare how TypeScript should resolve an import in your require/imports.

Any `require` or `import` in your typescript will be affected by this:

Then, `import 'jquery'` in your TS file will make the runtime look for the `jquery` module in `./vendor/jquery/dist/jquery`. We can freely overwrite this file and have it applied when the *app crashes*. Remember the start.sh script? It will attempt to restart the app if it detects a crash, and when Bun restarts, it will look for a `tsconfig.json` to apply. If we modify the `tsconfig.json` to include this property, we can basically tell Bun what code to run for certain imports, allowing us arbitrary code exec this way.

If we were to use path re-mapping, what would be a good candidate to re-map? Our first answer to this was the `yaml` library, as while it's used in the app to YAML-ify the JSON data, ther are no other use cases in the app for YAML to be parsed, loaded, etc etc. So, the re-mapping candidate for us will be `yaml`. When Bun comes across the `import { stringify } from 'yaml';` line in index.ts, the `tsconfig.json` will tell Bun to look to where we tell it to go instead.

So, gathering all our bugs, if we can make the filename `tsconfig.json\00<randomcharacters>.yaml`, it will resolve to and overwrite `tsconfig.json`. Wait, our file is still YAML. So when Bun attempts to read the JSON, it will error out! `tsconfig.json` needs to be valid JSON after all, right?

Right?

```
// Unfortunately "tsconfig.json" isn't actually JSON. It's some other
// format that appears to be defined by the implementation details of the
// TypeScript compiler.
//
// Attempt to parse it anyway by modifying the JSON parser, but just for
// these particular files. This is likely not a completely accurate
// emulation of what the TypeScript compiler does (e.g. string escape
// behavior may also be different).
```

I'll spare the details of how we tried to find all the weird and interesting properties that `tsconfig.json` will allow for, and how the compiler will parse it.

Here are some things that make it different from regular JSON:

1. It can support comment syntax like `/**/` which technically shouldn't be allowed in JSON formats.
2. Yaml allows `//` to be used as a key and naturally, Bun will ignore whatever comes after it
3. You can give it a valid JSON structure, then invalid JSON, and the compiler won't complain.

Anyway, the following "JSON" is a valid, readable and parsable `tsconfig.json`.

We can finally combine all of this into the final solve.

SOLVE

1. Copy an example prisoner JSON file and add your tsconfig path re-mapping:

Which will be valid `tsconfig.json`. We put our js file into the examples folder simply cause it's an rwx folder. Then add:

The nullbyte to remove the `<randomcharacters>.yaml` filename, the directory traversal and `proc/self/cwd` to get the symlink to `/app` and bypass the BANNED_WORDS list.

2. Make a javascript file and yeet into `/examples` for RCE (exec getflag and whatever)

3. Crash the app (we did this by trying to write to `proc/self/mem`) to force a restart and have our overwritten `tsconfig.json` apply to the app

```
flag: DUCTF{bUnBuNbUNbVN_hOn0_tH15_aPp_i5_d0n3!!!one1!!!!}
```