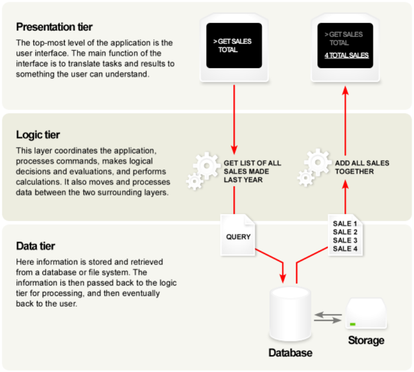
Web Security - HTML, JavaScript and Web Attacks

Stefano Calzavara

Università Ca' Foscari Venezia

Three-Tier Architecture of Web Apps



Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

Key technologies:

- Presentation tier: HTML, CSS and JavaScript
- Logic tier: PHP, JSP, Flask...
- Data tier: MySQL, Postgres, ...

The presentation tier (frontend) is handled by the client, while the logic and data tiers (backend) are handled by the server

Breaking the Tiers

Each tier of a web application can be target of attacks!

- Presentation tier: HTML injection, clickjacking...
- Logic tier: local file inclusion, XML external entities, server-side request forgery...
- Data tier: SQL injection

And this is not even the complete attack surface: web applications are distributed and operate over an untrusted network!

Hyper Text Markup Language (HTML)

Markup language based on elements, defining the document structure in terms of a tree:

- each element is described by a tag (heading, paragraph, image, ...), which is normally opened and closed, e.g., `<h1>Important!</h1>`
- the document is contained within the `<html>` and `</html>` tags, defining the root of the tree
- each element can have multiple attributes, defining any additional properties (e.g., the URL from which an image is fetched is defined in the `src` attribute of the `img` tag)

Tutorial: `https://www.w3schools.com/html/default.asp`

HTML Fundamentals

Many HTML tags are used to define the document structure:

- `<h1>This is the title of the document</h1>`
- `<p>This is a paragraph</p>`
- `<ol><li>First</li><li>Second</li></ol>`

Yet, other tags have relevant security implications, for example because they can trigger network communication:

- `<img src="https://www.foo.com/avatar.png"/>`
- `<a href="https://www.evil.com">Click me!</a>`

HTML Fundamentals: Forms

Example: login form

```
<form action="https://foo.com/login.php" method="POST">
    <input type="text" name="user">
    <input type="password" name="pwd">
    <button type="submit">Login</button>
</form>
```

When a form is submitted, the browser sends an HTTP request to the URL mentioned in the form action with the associated method.

Input elements and their values are also passed in the HTTP request (in the query string for GET requests, in the body for POST requests).

HTML Fundamentals: Frames

An iframe (inline frame) is an HTML element used to embed another HTML document within the current document:

```
<html>
  <head>
    <title>Iframe Example</title>
  </head>
  <body>
    <h1>Welcome to my website!</h1>
    <p>Check out this embedded video:</p>
    <iframe src="https://www.youtube.com/embed/dQ4w9WgXc"
            width="560" height="315"/>
  </body>
</html>
```

HTML Fundamentals: Scripts

HTML documents can also embed JavaScript in multiple forms:

- inline scripts: directly placed in the HTML document

```
<script>alert("Hello")</script>
```

- external scripts: loaded from separate JavaScript files

```
<script src="https://foo.com/bar.js"/>
```

- event handlers: associated to specific DOM events

```
<body onload="alert(1)">
...
</body>
```

HTML Encoding

HTML encoding is used to represent reserved characters so that they can be incorporated into an HTML document with their literal meaning:

- `&quot;` stands for double quotes "
- `&apos;` stands for single quotes '
- `&amp;` stands for ampersand &
- `&lt;` stands for less than <
- `&gt;` stands for greater than >

In addition, any character can be HTML-encoded using its ASCII code in decimal or hexidecimal forms, e.g., " can be encoded as `&#34;` or `&#x22;`

JavaScript

A full-fledged programming language used virtually on every website:

- client-side: parsed and executed by the web browser
- dynamic typing: variables can hold values of any type and their type can be changed during runtime
- prototype-based object-orientation: each object has a prototype, i.e., another object it inherits properties and methods from
- tight integration with the browser: JavaScript has access to powerful APIs granting access to browser assets, e.g., `document.cookie`
- many quirks: unconventional semantics enabling vulnerabilities such as type juggling and dangerous string-to-code transformations

You can find an introductory guide to JavaScript on MDN for reference.

Scoping in JavaScript

JavaScript variables live in the global scope by default

- ... even when they are declared within a function!
- this makes information hiding more complicated than expected
- you can make variables local to a function by using the var keyword

```
globalVariable1 = 5;                // A global variable

function globalFunction() {
    var localVariable = 2;          // A local variable
    globalVariable2 = 3;            // Another global variable
    window.globalVariable3 = 4;     // Yet another global variable
}
```

Scoping in JavaScript

While C++ or Java make use of block scoping, JavaScript utilizes the so-called function scoping:

- the JS engine creates a new scope for each encountered function
- an identifier that is locally defined within a function is associated with the function scope, irrespective of blocks
- you can enforce block scoping by using the `let` keyword
- the `const` keyword behaves as `let`, but requires assignment during initialization and prevents reassignment of the variable

Advice: there is nothing of `var` that `let` can't do better...

Scoping in JavaScript

What's the output of the following piece of code?

```
var age = 100;
if (age > 12) {
  var dogYears = age * 7;
  console.log('You are ' + dogYears + ' dog years old!');
}
console.log('Value of dogYears: ' + dogYears);
```

Scoping in JavaScript

JavaScript uses variable and function hoisting, i.e., their declarations are moved higher in their execution context before running the program.

```
sayHello(); // prints "Hello, world!"

function sayHello() {
  console.log("Hello, world!");
}
```

```
console.log(x); // Uncaught ReferenceError

console.log(y); // prints undefined
var y = 5;
```

JavaScript Typing

Dynamic typing: the same variable can store values of different types at different points in time

```
let value = 42; // Number type
value = "JavaScript"; // String type
```

Type coercion: JavaScript performs aggressive implicit type conversions to give a meaning to most programs you write

```
let result = 10 + "5"; // "105" (int coerced to string)
let sum = 5 + true; // 6 (boolean coerced to int)
```

Yes, JavaScript is weird!

JavaScript Typing

Duck typing: if it walks like a duck and it quacks like a duck, then it must be a duck

```
function printName(obj) {
  console.log(obj.name);
}

let person = { name: "John", age: 30 };
let dog = { name: "Bobby", breed: "akita"}
let book = { title: "JavaScript Book", price: 38.40 };

printName(person); // prints John
printName(dog);    // prints Bobby
printName(book);   // prints undefined
```

Type Juggling Vulnerabilities

In JavaScript, there are two different operators that can be used to compare values: loose equality (==) and strict equality (===).

```
"foo" == ["foo"]  // true
"foo" === ["foo"] // false
```

The key difference is that === does not perform type coercion and always returns `false` when the compared values have different types. This is what you normally want when performing comparisons.

Using loose equality is risky from a security perspective: if the attacker has control of one of the compared values, they might be able to alter the control flow or otherwise tamper with the running script!

Type Juggling Vulnerabilities

Assume you do not know the correct password. Can you attack this to get authorized?

```
let password = ***attacker controlled***

if (password == "0e462097431906509019562988736854")
    console.log("Authorized!")
else
    console.log("Unauthorized!")
```

Type Juggling Vulnerabilities

Assume you do not know the correct password. Can you attack this to get authorized?

```
let password = ***attacker controlled***

if (password == "0e462097431906509019562988736854")
    console.log("Authorized!")
else
    console.log("Unauthorized!")
```

Solution: set `password` to the value 0

Type Juggling Vulnerabilities

Using strict equality is useful to prevent type juggling vulnerabilities, but the undisciplined nature of JavaScript hides subtleties...

Assume the `isAdmin` field of the `user` object determines whether you have admin access. Can you attack this to promote Alice into an admin?

```
let user = {"name": "alice", "isAdmin": false}
let prop = ***attacker controlled***

if (prop === "isAdmin") {
    throw new Error("Unauthorized!");
} else {
    user[prop] = true;
}
```

Type Juggling Vulnerabilities

Using strict equality is useful to prevent type juggling vulnerabilities, but the undisciplined nature of JavaScript hides subtleties...

Assume the `isAdmin` field of the `user` object determines whether you have admin access. Can you attack this to promote Alice into an admin?

```
let user = {"name": "alice", "isAdmin": false}
let prop = ***attacker controlled***

if (prop === "isAdmin") {
    throw new Error("Unauthorized!");
} else {
    user[prop] = true;
}
```

Solution: set `prop` to the value `["isAdmin"]`

Document Object Model (DOM)

When the HTML is parsed, the browser builds an internal representation of the document called Document Object Model (DOM).

- The DOM organizes the HTML in a tree structure, where each element is a node of the tree and `document` is the root
- JavaScript programs can select and modify DOM elements by using methods such as `getElementById` and writing into properties such as `document.innerText`
- JavaScript can transitively load more JavaScript, e.g., by injecting `<script>` tags via `document.write` or `document.appendChild`
- JavaScript can also assign event handlers to DOM elements using methods like `addEventListener`, thus allowing the development of reactive programs that process specific inputs

Same Origin Policy (SOP)

The Same Origin Policy (SOP) is the baseline defense mechanism of web browsers, which isolates data controlled by `good.com` from read / write accesses by scripts served from `evil.com`.

Example

A script running on a page served by `https://www.foo.com` cannot:

- access cookies of `https://www.bar.com`
- access the DOM of `https://www.bar.com`
- access the DOM of `https://sub.foo.com`

We will discuss SOP in detail in the next lectures, including techniques to bypass it. For now, this intuitive understanding suffices.

SOP: Important Note

All scripts in a web page normally run in the same global space, namely that of the including page, and all of them run with the same privileges!

```
<html>
<script src="https://evil.com/attack.js/">
<script>console.log("I'm not alone here!");</script>
</html>
```

Isolation of scripts from untrusted sites can be enforced using iframes:

```
<html>
<iframe src="https://evil.com/attack.js/">
<script>console.log("Feeling better...");</script>
</html>
```

Eval and Friends

JavaScript can turn strings into executable code, e.g., by using the infamous `eval` function:

```
let x = 10;
let y = eval("x + 5"); // y is assigned 15
```

Other functions can be used to implement the same behavior, e.g., the `setTimeout` function:

```
setTimeout("console.log('Hello world!')", 1000);
```

Such functions are dangerous when the attacker has full or partial control of the parameters supplied to them!

JSON (JavaScript Object Notation)

JSON is a lightweight data interchange format, which is naturally integrated in JavaScript.

```
var obj = {name: "John", age: 30};
var jsonStr = JSON.stringify(obj);
console.log(jsonStr); // prints {"name":"John","age":30}

var jsonStr = '{"name":"John","age":30}';
var obj = JSON.parse(jsonStr);
console.log(obj.name); // prints John
console.log(obj.age); // prints 30
```

Since the JSON syntax defines valid JavaScript objects, JSON can also be parsed using `eval`, but this is potentially dangerous!

XMLHttpRequest (XHR)

XMLHttpRequest is a powerful JavaScript API used to send HTTP requests and process the corresponding HTTP responses.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       user = JSON.parse(xhttp.responseText);
    }
};
xhttp.open("GET", "https://bar.com/users.php?uid=123");
xhttp.send();
```

Fetch API

Fetch is a more modern, promise-based alternative to XHR. It has a cleaner and more intuitive syntax.

Example

```
fetch("https://bar.com/users.php?uid=123")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

In this course we will primarily focus on XHR for historical reasons. Security differences between XHR and Fetch are limited.

Inheritance in JavaScript

Inheritance in JavaScript is not based on classes, but directly on objects known as prototypes. Each object has a prototype, accessible using the `__proto__` property.

```
var o1 = {a: 1};    // prototype is Object.prototype

var o2 = Object.create(o1);  // prototype is o1

console.log(o2.a);  // prints 1

console.log(o2.__proto__);  // prints {a: 1}
```

Method invocations and property accesses traverse the prototype chain to look for a valid definition, up to the root `Object.prototype`.

Prototype Pollution

Prototype pollution is a JavaScript vulnerability that enables an attacker to add arbitrary properties to global object prototypes, which may then be inherited by user-defined objects.

```
let user = {};

console.log(user.isAdmin)  // prints undefined

user.__proto__.isAdmin = true

console.log(user.isAdmin)  // prints true
```

Prototype Pollution

Since __proto__ is a writable property, you can arbitrarily redefine the prototype as you like!

```
let x = {};

x.__proto__ = String.prototype

x.charAt(1); // Uncaught TypeError: 'this' must be String
```

By updating __proto__, the attacker can redefine the inherited behavior of objects to make them behave unexpectedly!

Prototype Pollution

Remarkably, methods are also properties in JavaScript, i.e., a method is a property with a function value.

```
let x = [1,2,3];

Array.prototype.toString = function() { return "foo"; }

x.toString();  // prints foo
```

As you can see from the example, even native methods can be redefined!

Object Properties

JavaScript objects are highly mutable:

- new properties can be added after creating the object
- existing properties can be deleted or reconfigured

Each property has its own configuration:

- `value`: the value of the property
- `writable`: can the value of the property be changed?
- `enumerable`: can the property be enumerated, e.g., in loops?
- `configurable`: can the property be deleted and reconfigured?

Properties can be configured using `Object.defineProperty()`

Object Freezing

How to protect objects against tampering?

- `Object.preventExtensions()`: suppresses the addition of new properties and the object's prototype from being reassigned
- `Object.seal()`: same as `Object.preventExtensions()`, plus it makes existing properties non-configurable
- `Object.freeze()`: same as `Object.seal()`, plus it makes existing properties read-only

Alert!

Freezing an object does not freeze its prototype!

Making JavaScript Better: Strict Mode and TypeScript

JavaScript is definitely not the best language in the world, but strict mode promotes the writing of more disciplined code:

- It disallows the usage of unsafe language features (e.g., variable declarations in `eval`)
- It converts silent mistakes into errors (e.g., it forbids assignments to undeclared variables)

```
(function () {
  "use strict";
  undeclaredVariable = 42; // throws ReferenceError
 })();
```

TypeScript is a strongly typed extension of JavaScript, which allows static detection of many type errors. It must be compiled to JavaScript.

TypeScript: Example

The following code does not type-check:

```
let user = {"name": "alice", "isAdmin": false}
let prop = ["isAdmin"]

if (prop === "isAdmin") {  // type error here
    throw new Error("Unauthorized!");
} else {
    user[prop] = true;     // type error also here
}
```

TypeScript forces the `prop` variable to have type `string` to mark the code as well-typed, thus forbidding the dangerous assignment at line 2.

TypeScript: Example

```
type User = {
    name: string;
    isAdmin: boolean;
};

let user: User = {"name": "alice", "isAdmin": false};
let prop: string = "isHappy";

if (prop === "isAdmin") {
    throw new Error("Unauthorized!");
} else {
    user[prop] = true;
}
```

Mini-Lab: Playing as a Web Attacker

We now practice with JavaScript by simulating how a web attacker would steal cookies from www.unive.it through a script injection attack, where we simulate inclusion of an attacker-controlled script:

1. Create an HTTP dump at https://httpdump.app
2. Load your dump in a new browser tab and observe that the GET request is correctly logged in the original tab
3. Browse to https://www.unive.it and open the Javascript console
4. Write and execute a script which reads Unive's cookies and leaks them to the dump, for example using an `<img>` tag

Notice that also a network attacker might be able to sniff the cookies, if the request to the dump was performed over plain HTTP. Try to do that and see what happens!