



OWASP TOP 10 For JavaScript Developers

@LewisArdern



What is the OWASP Top 10?

OWASP Top 10 2017	
A1	Injection
A2	Broken Authentication
A3	Sensitive Data Exposure
A4	XML External Entities (XXE)
A5	Broken Access Control
A6	Security Misconfiguration
A7	Cross-site Scripting
A8	Insecure Deserialization
A9	Using Components with Known Vulnerabilities
A10	Insufficient Logging and Monitoring

- 10 critical web application security risks
- Common flaws and weaknesses
- Present in nearly all applications

Modern, evidence-based risks. Data covers 2014-2017:

- 114,000 apps
- 9000 bug bounties
- 40 security consultancies and 1 bug bounty firm
- 50+ CWEs accepted in raw data

Community-chosen risks

- 500 survey responses

A1:2017 Injection

The Dangers of Mixing Data and Code

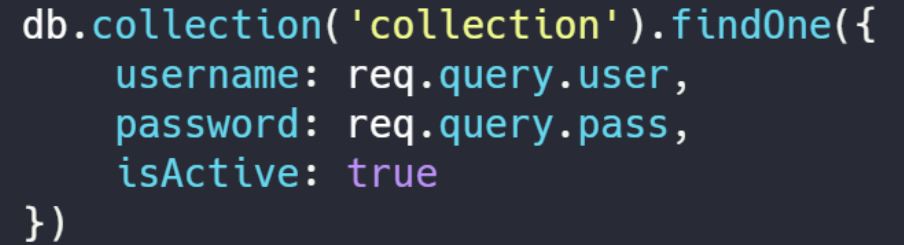
NoSQL Injection

No SQL Injection != No Injection In NoSQL

Official documentation says no SQL Injection

Vulnerable If:

- User input includes a Mongo Query Selector:
 - \$ne, \$lt, \$gt, \$eq, \$regex, etc.
- User input is directly included into a collection method as part of the query:
 - find, findOne, findOneAndUpdate, etc.



```
db.collection('collection').findOne({  
  username: req.query.user,  
  password: req.query.pass,  
  isActive: true  
})
```

<https://docs.mongodb.com/manual/faq/fundamentals/#how-does-mongodb-address-sql-or-query-injection>
<https://docs.mongodb.com/manual/reference/operator/query/>
<https://docs.mongodb.com/manual/reference/method/>

Vulnerable MongoDB Login Example

```
db.collection('collection').findOne({
  username: req.query.user,
  password: req.query.pass,
  isActive: true
}, function (err, result) {
  if (err) {
    console.log('Query error...');
    return err;
  }
  if (result !== null) {
    req.session.authenticated = true;
    res.redirect('/');
  }
  else
    res.redirect('/login?user=' + user);
});
```

Injection:

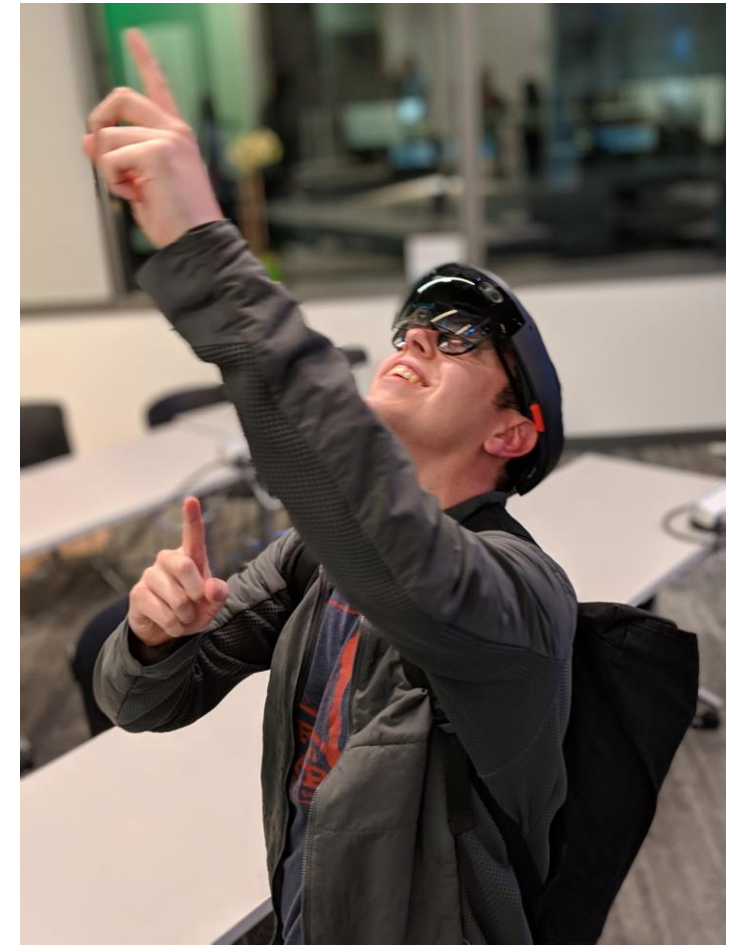
[https://url.to/login?user=admin&pass\[\\$ne\]=](https://url.to/login?user=admin&pass[$ne]=)

Query Output:

```
db.collection('collection').findOne({
  username: "admin",
  password: {
    $ne: "",
  },
  isActive: true
})
```

About Me

- Sr. Security Consultant @ Synopsys Software Integrity Group (SIG)
 - Formerly Cigital
- AngularSF Organizer
 - <https://www.meetup.com/Angular-SF/>
- B.Sc. in Computer Security and Ethical Hacking
 - Founder of <http://leedshackingsociety.co.uk/>
- JavaScript Enthusiast!




SYNOPSYS®

Demo

MongoDB Injection

MongoDB Injection Prevention

- Ensure user-input is a String inside a collection method
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
- Perform Custom Data Validation
 - <https://github.com/hapijs/joi>



```
db.collection('collection').findOne({  
    username: String(req.query.user),  
    password: String(req.query.pass),  
    isActive: true  
})
```



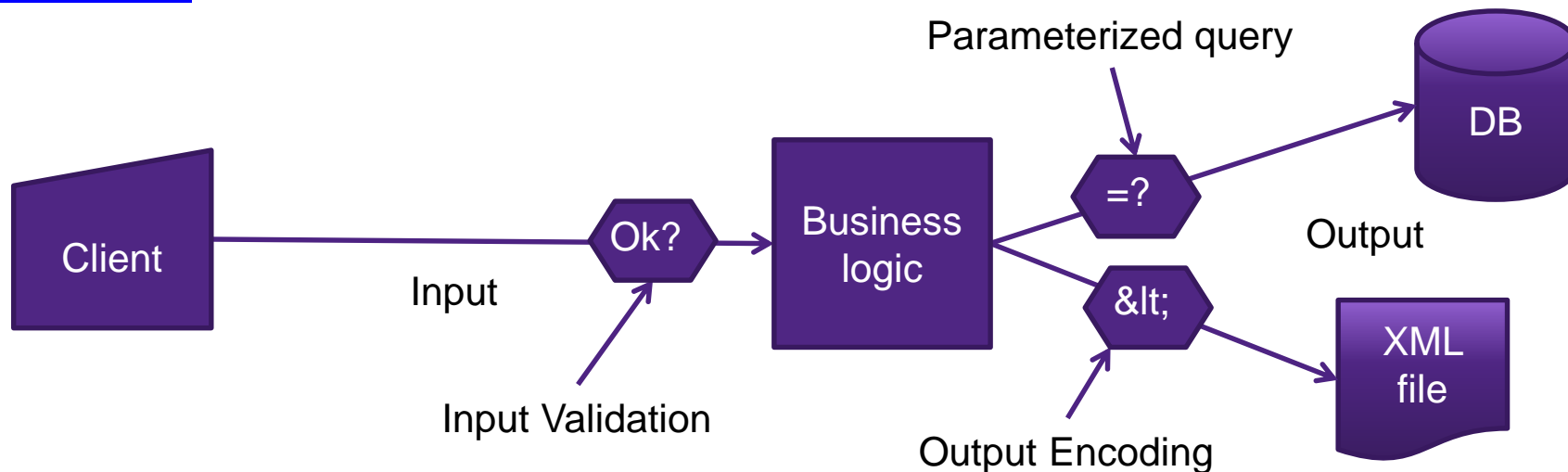

```
connection.query(  
  "SELECT * from users where userid='" + obj.user + "'",  
  function (err, results) {  
    console.log(results);  
  }  
);
```



```
connection.query(  
  'SELECT * from users where userid=?', [obj.user],  
  function (err, results) {  
    console.log(results);  
  }  
);
```

Injection Prevention

- Parameterized Mechanisms
 - <https://github.com/tediousjs/node-mssql#input-name-type-value>
 - <https://github.com/mysqljs/mysql#escaping-query-identifiers>
- Secure APIs
 - <https://github.com/tediousjs/node-mssql#prepared-statements>
- Perform Input Validation & Output Encoding
 - <https://dev.to/azure/pushing-left-like-a-boss-part-5-1-input-validation-output-encoding-and-parameterized-queries-2749>



A2:2017 Broken Authentication

Broken Authentication and Session Management

Secure Session Management Techniques

Browser Enforcement

Prefix the cookie name with the __Secure or __Host to prevent misconfiguration

Transmit Securely

Set the *Secure* attribute to protect in transit (HTTPS only)

Create Securely

- Ensure session ID is cryptographically random
- Renew session ID after login

Use Securely

Set the SameSite attribute to prevent inclusion in cross-origin requests

Destroy Securely

- Ensure session timeout is set appropriately
- Invalidate session after logout

Store Securely

Set the HttpOnly attribute to protect from JavaScript reading cookie stored in browser




Secure express-session Example



```
session = require('express-session')
app.use(session({
  secret: config.SESSION_SECRET,
  name: '__Host-auth-cookie',
  cookie: { httpOnly: true, secure: true, sameSite: 'strict', path: '/' },
  store: new MySQLStore({}, sqlConnection)
}));
```


Never Store The Secret In Source Control




```
session = require('express-session')
app.use(session({
  secret: config.SESSION_SECRET,
  name: '__Host-auth-cookie',
  cookie: { httpOnly: true, secure: true, sameSite: 'strict', path: '/' },
  store: new MySQLStore({}, sqlConnection)
}));
```

Enforce The Browser To Only Transmit Over HTTPs


```
session = require('express-session')
app.use(session({
  secret: config.SESSION_SECRET,
  name: '__Host-auth-cookie',
  cookie: { httpOnly: true, secure: true, sameSite: 'strict', path: '/' },
  store: new MySQLStore({}, sqlConnection)
}));
```

Prevent Inclusion In Cross-Origin Requests



```
session = require('express-session')
app.use(session({
  secret: config.SESSION_SECRET,
  name: '__Host-auth-cookie',
  cookie: { httpOnly: true, secure: true, sameSite: 'strict', path: '/' },
  store: new MySQLStore({}, sqlConnection)
}));
```

Prevent Memory Exhaustion With A Store



```
session = require('express-session')
app.use(session({
  secret: config.SESSION_SECRET,
  name: '__Host-auth-cookie',
  cookie: { httpOnly: true, secure: true, sameSite: 'strict', path: '/' },
  store: new MySQLStore({}, sqlConnection)
}));
```

Insecure Object Comparisons

- What happens if you create your own Authentication middleware?



```
const SESSIONS = {}

const mustBeAuthenticated = (req, res, next) => {
  if(req.cookies) {
    const token = req.cookies.token

    if(token && SESSIONS[token]) {
      //allow it
      next()
    }
  }
  res.send('not authorized!')
}
```


Comparison Table

Value	Return
SESSIONS[' <i>invalidString</i> ']	False
SESSIONS['']	False
SESSIONS[' <i>constructor</i> ']	True
SESSIONS[' <i>hasOwnProperty</i> ']	True

What Happens When You Create an Object in JavaScript?



```
const SESSIONS = {}
```

```
__proto__:
```

```
  constructor: f Object()
```

```
  hasOwnProperty: f hasOwnProperty()
```

```
  isPrototypeOf: f isPrototypeOf()
```

```
  [...]
```

```
SESSIONS['constructor'] === SESSIONS.constructor //returns true
```

Exploit

This issue is trivial to exploit.

- Using cURL we can simply run the following command:
 - curl https://localhost:9000 -H "Cookie: token=constructor"
- Alternatively, you can just set the *document.cookie* value via the browser.

Demo

Insecure Object Comparisons

How Do We Correctly Check?

- Use `crypto.timingSafeEqual(a, b)`
 - https://nodejs.org/api/crypto.html#crypto_crypto_timingsafeequal_a_b
 - It provides a safe comparison and prevents timing attacks
- `Object.hasOwnProperty` or `hasOwnProperty` do not check base properties
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/has



```
SESSIONS.has( '__proto__' )  
  
// false  
  
SESSIONS.has( 'validString' )  
  
// true
```




```
SESSIONS.hasOwnProperty( '__proto__' )  
  
// false  
  
SESSIONS.hasOwnProperty( 'validString' )  
  
// true
```


A3:2017 Sensitive Data Exposure


RESTful API Data Leakage

Its easy to write the following code:



```
try { /*stuff*/ } catch(e) { return e; }
```

But what can happen on exposed routes?



```
{"user": {"id": "DB2 SQL-Error: -104 ILLEGAL SYMBOL SOME SYMBOLS THAT MIGHT BE LEGAL ARE USED AFTER  
'USERID' COLUMN."}}
```

Do Not Include Verbose Error Messages in JSON


- Verbose error messages lead to system information disclosure.
- Although the client-side application displays a generic error message, the JSON response might still contain full error messages.
- Malicious users may use a web proxy to read the stack trace output in JSON.

Caution: Detailed system information might not seem that significant at first sight. However, it can inform attackers on the internals of the system or infrastructure, and help them drive further attacks.

Disclosing Information via Error Messages in JSON

This code shows an SQL error message passed in JSON and the JavaScript code used to mask it.

JavaScript code to handle the error:



```
connection.query('SELECT * from users where userid=?', [obj.user], (err, results) => {  
  if (err) {  
    logger.log(`SQL Error:  ${err}`);  
    return 'Could not complete query';  
  }  
  return results;  
})
```

A4:2017 XML External Entities (XXE)

XML External Entities (XXE) Injection

Two examples of parsing libraries vulnerable to XXE

- node-expat
 - 48,353 weekly downloads
 - Vulnerable by default
 - No way to configure parser to disable DTD
 - <https://help.semmle.com/wiki/display/JS/XML+internal+entity+expansion>
- libxmljs
 - 47,876 weekly downloads
 - Vulnerable if noent is set to true
 - <https://help.semmle.com/wiki/display/JS/XML+external+entity+expansion>

```
var expat = require('node-expat')
var parser = new expat.Parser('UTF-8')

parser.on('text', function (text) {
  console.log(text)
})

parser.write(xmlSrc)
```

```
const libxml = require('libxmljs');
var doc = libxml.parseXml(xmlSrc, { noent: true });
```

XML External Entities (XXE) Vulnerable Example

Libxmljs can be vulnerable to XXE

User-input:
req.files

```
module.exports.bulkProducts = function (req, res) {  
  if (req.files.products && req.files.products.mimetype == 'text/xml') {  
    var products = libxmljs.parseXmlString(req.files.products.data.toString('utf8'),  
      { noent: true, noblanks: true })  
    products.root().childNodes().forEach(product => {  
      var newProduct = new db.Product()  
      // ..snip..  
      newProduct.description = product.childNodes()[3].text()  
      newProduct.save()  
    })  
    res.redirect('/app/products')  
  } // ...snip...  
}
```

Misconfiguration:
noent: true


XML Injection Prevention

- Consider using a library which does not process DTDs
 - <https://github.com/isaacs/sax-js>
- Use libraries with safe defaults, such as libxmljs (apart from its sax parser)
 - <https://github.com/libxmljs/libxmljs>
- If entities such as & or > need to be expanded use lodash, underscore, or he
 - <https://lodash.com/docs/4.17.11#unescape>
 - <https://underscorejs.org/#unescape>
 - <https://github.com/mathiasbynens/he>
- Alternatively, strict input validation/output encoding must be performed before parsing

A5:2017 Broken Access Control

Do Not Rely on Client-Side Controls


- Client-side routing and authorization should only be implemented for user experience
- Authentication and authorization controls implemented client-side can be bypassed
- All authorization, authentication, and business logic controls must be enforced server-side:
 - npm packages - <https://github.com/casbin/node-casbin>
 - Frameworks - <https://sailsjs.com/documentation/concepts/policies/access-control-and-permissions>
 - Writing custom middleware:



```
app.get ('/api/users/:id', auth.requiresRole('admin'), admin.getUserById);
app.get('/api/users/edit/:id', auth.requiresRole('admin'));
app.put('/api/users/:id', auth.requiresRole('admin'), admin.updateUser);
app.delete('/api/users/:id', auth.requiresRole('admin'), admin.deleteUser);
app.post('/api/admin/logs', auth.requiresRole('admin'), admin.checkTmpFolder);
```

Angular Example

- Angular Route Guards are for Boolean display aesthetics



```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(public router: Router) {}

  canActivate(): boolean {
    alert("Unauthorized! Only administrators are allowed.");
    return false;
  }

}
```

<https://angular.io/guide/router#milestone-5-route-guards>

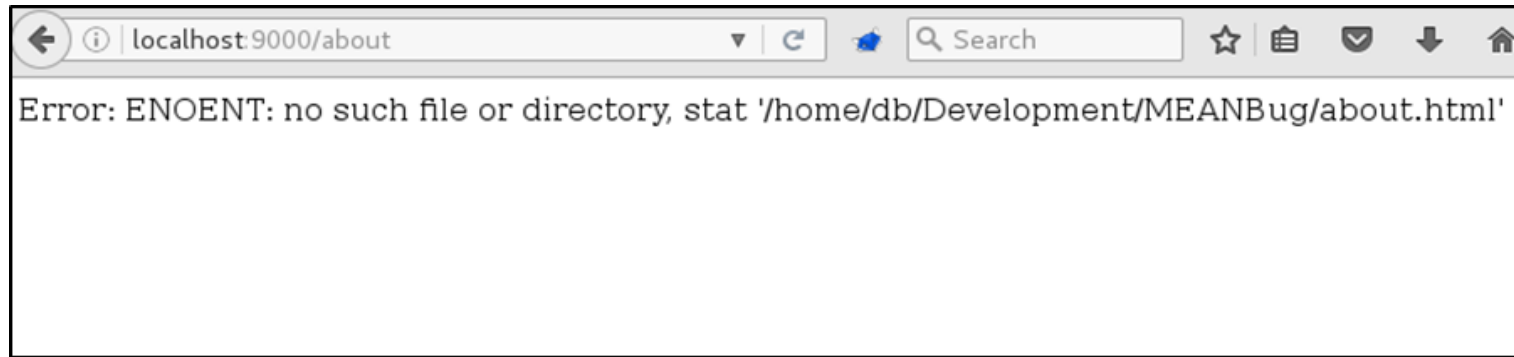
<https://nvisium.com/blog/2019/01/17/angular-for-pentesters-part-2.html>

A6:2017 Security Misconfiguration

Ensure Node Is Not Running in Development Mode

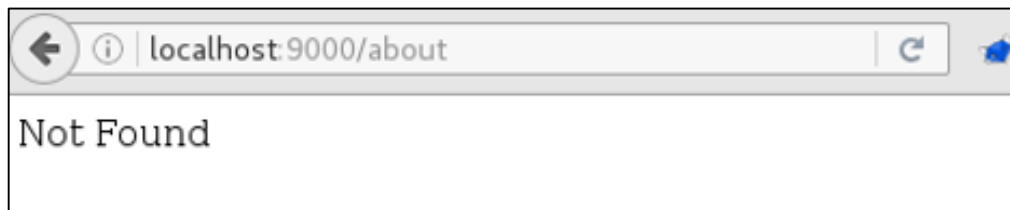
NodeJS applications run in development mode by default

- NodeJS and most frameworks that run on it return verbose errors if left in development mode




- When deploying to production, set the `NODE_ENV` variable to a value other than *development* to avoid verbose errors
 - <https://expressjs.com/en/advanced/best-practice-performance.html>

```
[02.26.2017] db@kali-VM1:MEANBug$ NODE_ENV=production node server.js  
Server running on localhost, port 9000 in production mode.
```



Ensure Node Is Not Running with sudo Privileges

- A Node.js application running with sudo privileges has a greater chance of modifying the underlying server system through malicious code execution.
 - On Linux systems, sudo is required to bind to ports under 1000 (e.g., 80)
 - If sudo is required, after the port has been bound, change the privileges to a less privileged user and group:



```
app.listen(80, 'localhost', null, function() {  
  process.setgid('users');  
  process.setuid('www-data');  
});
```

<https://nodejs.org/api/process.html>

A7:2017 Cross-Site Scripting (XSS)

XSS Is Easy To Introduce



```
const userName = location.hash.match(/userName=([^;&]*)/)[1];  
// ...  
div.innerHTML += `Welcome ${userName}">`
```

Script Execution:

http://www.vulnerable.site?userName=

XSS Prevention Is HARD

- DOM XSS is hard to prevent in today's developer ecosystem
 - <https://hackerone.com/reports/158853>
 - <https://hackerone.com/reports/405191>
 - <https://hackerone.com/reports/164821>
- Each browser parses and renders HTML differently
 - <https://www.youtube.com/watch?v=IG7U3fuNw3A>
 - <http://shazzer.co.uk>
- Various execution contexts and character sets
 - <https://html5sec.org>
 - <https://github.com/cure53/XSSChallengeWiki/wiki/Puzzle-1-on-kcal.pw>
 - <http://polyglot.innerht.ml/>
- Script Gadgets
 - <https://github.com/google/security-research-pocs/tree/master/script-gadgets>



web

```
<noscript><p title="</noscript><img src=x onerror=alert(1)>">
```



@LiveOverflow

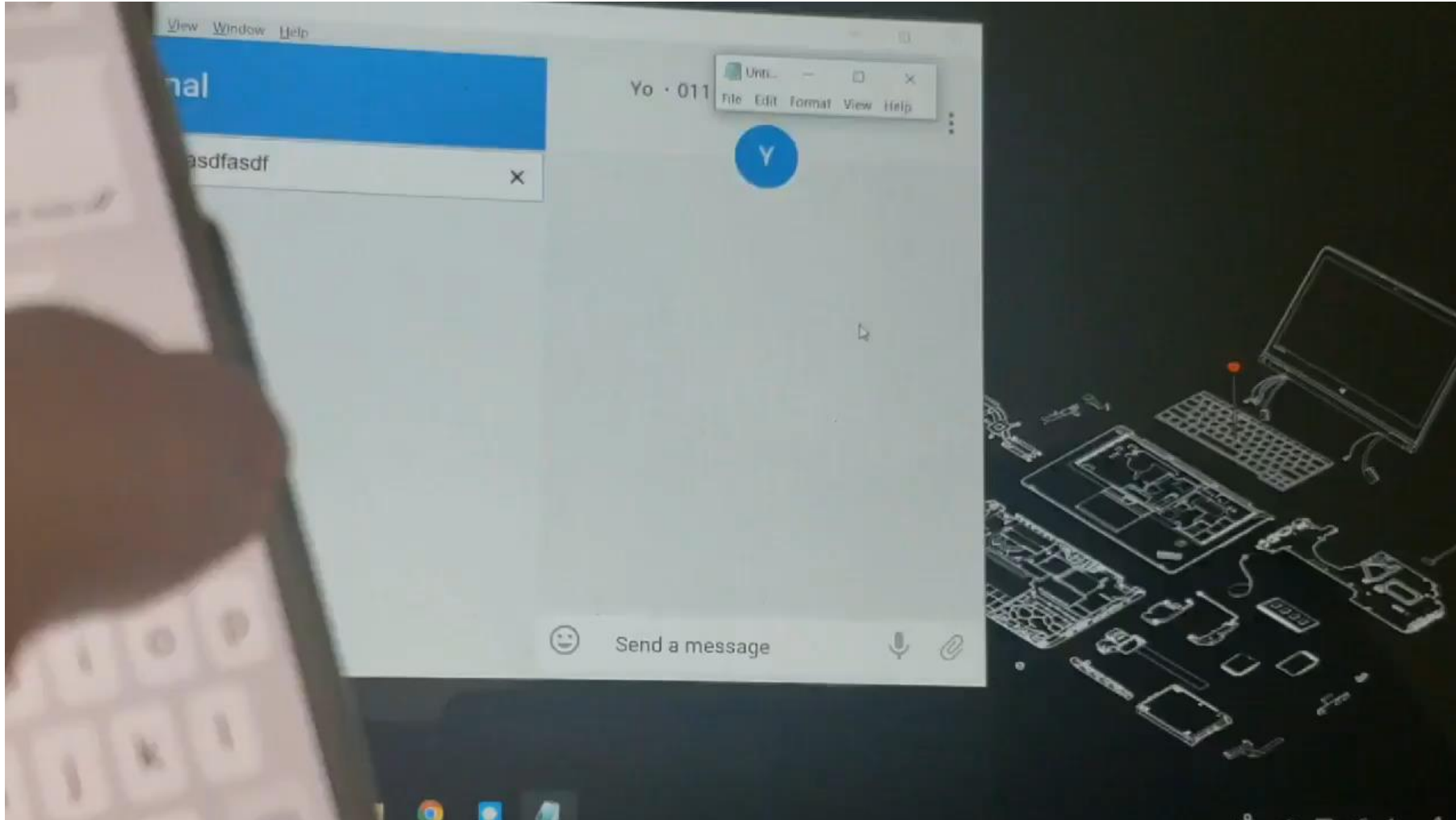
Frameworks Reduce The Attack Surface Until:

- Combining templating engines, third-party libraries, and frameworks
 - <https://jsfiddle.net/015jxu8s/>
- Disabling security controls
 - [https://docs.angularjs.org/api/ng/provider/\\$sceProvider](https://docs.angularjs.org/api/ng/provider/$sceProvider)
- Using Insecure APIs
 - [trustAs](#), [v-html](#), [bypassSecurityTrust](#), or [dangerouslySetInnerHTML](#)
- Allowing JavaScript URIs in ``
 - <https://medium.com/javascript-security/avoiding-xss-in-react-is-still-hard-d2b5c7ad9412>
- Direct access to the DOM
 - <https://angular.io/api/core/ElementRef>
- Server-Side Rendering
 - <https://medium.com/node-security/the-most-common-xss-vulnerability-in-react-js-applications-2bdffbcc1fa0>
- Caching mechanisms such as `$templateCache`
 - <https://docs.angularjs.org/guide/security>

Note: This is not an exhaustive list.

Signal Creates a Lot of Noise

What happens if you bypass React controls for insecure use?



Source: https://ivan.barreraoro.com.ar/wp-content/uploads/2018/05/poc1.mp4?_=1

What Went Wrong?

Signal developers utilized *dangerouslySetInnerHTML* for phone and desktop leading to RCE in the desktop and Cross-Site Scripting (XSS) in iOS/Android

```
7 + import { MessageBody } from './MessageBody';
8 +
7 9 interface Props {
8 10   attachments: Array<QuotedAttachment>;
9 11   authorColor: string;
@@ -111,7 +113,9 @@ export class Quote extends React.Component<Props, {}> {
111 113
112 114   if (text) {
113 115     return (
114 -     <div className="text" dangerouslySetInnerHTML={{ __html: text }} />
116 +     <div className="text">
117 +       <MessageBody text={text} />
118 +     </div>
115 119   );
116 120 }
117 121
```

General Prevention Techniques

- Libraries and frameworks for automatic output encoding and sanitization:

- [Pug](#), [Mustache](#), [EJS](#)
- [Angular](#), [React](#), [Vue](#)
- [secure-filters](#)

- Sanitization for HTML, MathML and SVG with DOMPurify

- <https://github.com/cure53/DOMPurify>

- Default to safe APIs

- [innerText](#)
- [encodeURIComponent](#)

Templating Engine	HTML Output
Mustache {{code}}	Input
Jade/Pug #{code}	Input
EJS <%=code%>	Input



```
const createDOMPurify = require('dompurify');
const { JSDOM } = require('jsdom');

const window = (new JSDOM('')).window;
const DOMPurify = createDOMPurify(window);

const clean = DOMPurify.sanitize(dirty);
```

Caution: Always use the correct encoding context, [in the correct order](#).

Apply Defence In Depth Strategies

- Create a strong Content Security Policy (CSP)
 - <https://speakerdeck.com/lweichselbaum/csp-a-successful-mess-between-hardening-and-mitigation>
 - <https://twitter.com/LewisArdern/status/1112926476498698240>
 - <https://csp.withgoogle.com>
- Experiment with Trusted Types
 - <https://developers.google.com/web/updates/2019/02/trusted-types>

```
script-src 'strict-dynamic' 'nonce-rAnd0m123'  
object-src 'none';  
base-uri 'none';  
report-uri https://csp.example.com;
```

```
const templatePolicy = TrustedTypes.createPolicy('template', {  
  createHTML: (templateId) => {  
    const tpl = templateId;  
    if (/^[0-9a-z-]$/i.test(tpl)) {  
      return `<link rel="stylesheet" href="./templates/${tpl}/style.css">`;  
    }  
    throw new TypeError();  
  }  
});  
  
const html = templatePolicy.createHTML(location.hash.match(/tplid=([^&]*)/)[1]);  
// html instanceof TrustedHTML  
document.head.innerHTML += html;
```

A8:2017 Insecure Deserialization

Avoid Unsafe Deserialization

NodeJS applications are vulnerable to RCE (Remote Code Execution) exploits if attacker-controlled data is deserialized via reflection

- Avoid passing untrusted data to deserialization functions
- Apply security patches for software that contains known deserialization vulnerabilities
- Encrypt or hash serialized objects
 - Prevents tampering, but not replay
- Check the object type is as expected

```
var serialize = require("node-serialize");
... snip ...
app.get("/", function(req, res){
  if(req.cookies.user){
    var user = Buffer.from(req.cookies.user, 'base64').toString();
    console.log("Received user object %s", user);

    var userObj = serialize.unserialize(user);

    res.send("Hello " + escape(userObj.username) + "!");
  }
}
```

Eval is Evil



new Function, setTimeout, setInterval also dynamically evaluate JavaScript

- The core issue behind node-serialize is the use of eval
 - <https://github.com/luin/serialize/blob/master/lib/serialize.js#L76>

```
for(key in obj) {
  if(obj.hasOwnProperty(key)) {
    if(typeof obj[key] === 'object') {
      obj[key] = unserialize(obj[key], originObj);
    } else if(typeof obj[key] === 'string') {
      if(obj[key].indexOf(FUNCFLAG) === 0) {
        obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
      } else if(obj[key].indexOf(CIRCULARFLAG) === 0) {
        obj[key] = obj[key].substring(CIRCULARFLAG.length);
        circularTasks.push({obj: obj, key: key});
      }
    }
  }
}
```

A9:2017 Using Components with Known Vulnerabilities

Security Issues with Third-Party Components

- Perform a security audit against 3rd party code
- If you find a security issue, notify the project maintainer
 - <https://github.blog/2019-05-23-introducing-new-ways-to-keep-your-code-secure/#open-source-security>
- Use automated tools to audit dependencies in your CI/CD pipeline:

Example	Command
npm https://docs.npmjs.com/cli/audit	npm audit --fix
yarn https://yarnpkg.com/en/docs/cli/audit	yarn audit --fix
bower https://www.npmjs.com/package/auditjs	auditjs --bower bower.json
Client-Side JavaScript https://github.com/retirejs/retire.js/	retire --js /path/
Node.js Open-Source https://snyk.io/test/	snyk test

Examples of Components with Known Vulnerabilities

These are examples of popular components with known vulnerabilities:

- **Prototype Pollution In Lodash:** [CVE-2018-3721](#) in [Lodash](#) impact in some cases was denial of service (DoS), Remote Code Execution (RCE), and even bypass security controls.
- **Directory Traversal in Next.js:** [CVE-2018-6184](#) in [Next.js](#) allowed for arbitrary read of the file system
- **Cross-Site-Scripting (XSS) in Next.js:** [CVE-2018-18282](#) in [Next.js](#) allowed for XSS on the `/_error` page
- **Privilege Escalation in auth0-js:** [CVE 2018-6873](#) in [auth0-js](#) did not validate JWT audience which allowed for Privilege Escalation
- **Arbitrary Command Injection in Kibana:** [CVE-2018-17246](#) in [Kibana](#) allowed for arbitrary command execution in the Console Plugin.

Mitigation Techniques

Track use of outdated third-party components and update where necessary:

- Maintain a technology assets inventory to track components and dependencies
 - <https://medium.com/uber-security-privacy/code-provenance-application-security-77ebfa4b6bc5>
 - <https://yarnpkg.com/lang/en/docs/cli/why/> and <https://yarnpkg.com/lang/en/docs/cli/list>
 - <https://docs.npmjs.com/cli/lis.html>
 - <https://bower.io/docs/api/#list>
- Review the inventory on a regular basis for known vulnerabilities
- Track known risks and vulnerabilities in the environment
- Develop a process to update, and regression test external components
- Pin Dependency versions where possible
 - Reduce the risk of another [event-stream](#) affecting your organization
 - <https://docs.npmjs.com/files/shrinkwrap.json>
 - <https://yarnpkg.com/lang/en/docs/yarn-lock>

A10:2017 Insufficient Logging and Monitoring

Insufficient Logging and Monitoring

Insufficient logging and monitoring of computer systems, applications, and networks provide multiple gateways to probes and breaches that can be difficult or impossible to identify and resolve without a viable audit trail.

Basic vulnerabilities include:

- Unlogged events, such as failed login credentials
- Locally stored logs without cloud backup
- Misconfigurations in firewalls and routing systems
- Alerts and subsequent responses that are not handled effectively
- Malicious activity alerts not detected in real time

Many reported breaches were only discovered years after the first intrusion happened. In the case of one famous university, a data breach from 2008 was not discovered until 2018!

Secure Logging and Monitoring

1. Plan

Enumerate, scope, define policies

2. Monitor

Analyze, collect, store

3. Action

Escalate, collect

Key steps to implement a cybersecurity logging and monitoring capability:


- Develop a logging and monitoring plan
- Carry out requirements for logging and monitoring
- Determine sources of potential compromise indicators
- Design the logging and monitoring capability
- Build the logging and monitoring capability
- Integrate the capability into your security framework
- Maintain the logging and monitoring capability

Use Winston to Enable Secure Logging Practices

- Winston is a logging library that escapes data to prevent log injection:

```
{ "level": "info", "message": "Received: hello\\r\\nReceived: fakeentry", "timestamp": "2015-11-16T21:09:57.024Z" }
```

- Winston also provides various built-in security controls, such as content filters:



```
var logger = new Winston.logger({filters: [function(level, msg, meta){  
  return maskCardNumbers(msg);  
}]};  
winston.log('info', "Card 123456789012345 processed.");
```

- Resulting in the following output:

```
info: Card 123456****2345 processed.
```

Note: node-bunyan is another logging library that logs JSON strings securely.

<https://www.npmjs.com/package/winston> & <https://github.com/trentm/node-bunyan>

Thank you!

Email: lewis@ardern.io

Website: <https://ardern.io>

Twitter: <https://twitter.com/LewisArdern>

GitHub: <https://github.com/LewisArdern>

LinkedIn: <https://www.linkedin.com/in/lewis-ardern-83373a40>

Recommended Reading:

OWASP Top 10

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

OWASP Application Security Verification Standard

https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

OWASP Proactive Controls

https://www.owasp.org/index.php/OWASP_Proactive_Controls

OWASP Testing Guide

https://www.owasp.org/index.php/OWASP_Testing_Project

OWASP Cheat Sheet Series

https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series

BSIMM

<https://www.bsimm.com/>

https://www.owasp.org/index.php/OWASP_SAMM_Project

SafeCode

<https://safecode.org>

Microsoft Agile SDL

<https://www.microsoft.com/en-us/SDL/discover/sdlagile.aspx>

Vulnerable Machines



https://www.owasp.org/index.php/OWASP_Juice_Shop_Project

https://www.owasp.org/index.php/OWASP_Node_js_Goat_Project

<https://github.com/dbohannon/MEANBug>

<https://github.com/appsecco/dvna>

Cheat Sheets & Best Practices



The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL injection), but are out of scope for this cheat sheet. Hence, they are omitted.

<https://cheatsheets.pragmaticwebsecurity.com/cheatsheets/angularOWASPTop10.pdf>

<https://github.com/i0natan/nodebestpractices>

Recommended Open Source Analysis Tools

Referencing only projects that are either open-source or scan open-source:

Products that perform JavaScript data flow analysis:

- [Coverity Scan](#)
- [LGTM](#)

Tools that look for areas of interest:

- [Tarnish](#)
- [JSHint](#)
- [JSLint](#)
- [ESLint](#)
 - [Code Climate - nodesecurity plugin](#)
- [TSLint](#)
 - [tslint-config-security](#)
 - [tslint-angular-security](#)

Tools that look for known issues in JavaScript libraries:

- [Retire.js](#)
- [npm audit](#)
- [yarn audit](#)
- [GitHub](#)
- [Snyk](#)
- [auditjs](#)

Tools that deobfuscate JavaScript:

- [Closure Compiler](#)
- [JStillery](#)
- [Unminify](#)
- [Jsnice](#)
- [jsdetox](#)
- [prepack.io](#)

ESLint Security Rules

- ESLint can help identify security issues
- Default security rule configs
 - NodeJS <https://github.com/nodesecurity/eslint-config-nodesecurity>
 - VanillaJS <https://github.com/mozfreddyb/eslint-config-scanjs>
 - AngularJS <https://github.com/LewisArdern/eslint-plugin-angularjs-security-rules>
 - React <https://github.com/yannickcr/eslint-plugin-react#list-of-supported-rules>
- Security rules
 - [eslint-plugin-scanjs](#)
 - [eslint-plugin-security](#)
 - [eslint-plugin-react](#)
 - [eslint-plugin-angularjs-security](#)
 - [eslint-plugin-no-wildcard-postmessage](#)
 - [eslint-plugin-no-unsafe-innerhtml](#)
 - [vue/no-v-html](#)
 - [eslint-plugin-prototype-pollution-security-rules](#)