GitSec.Cloud
48h to DevSecOps

漏洞修复建议速查手册

(Go 篇)

开源安全研究院

2022/7/28

返回目录

简介

手册简介：

本手册是针对 Go 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

Missing Content Security Policy
Missing HSTS Header
Missing HttpOnly Cookie
Missing Secure Cookie
Open Redirect
Overly Permissive Cross Origin Resource Sharing Policy
Password Management
Password Management:Empty Password
Password Management:Hardcoded Password
Password Management:Null Password
Password Management:Weak Cryptography
Path Manipulation
Path Manipulation:Zip Entry Overwrite
Path Traversal
PBKDF2 Insufficient Iteration Count
PBKDF2 Weak Salt Value
Permissive Content Security Policy
Poor Logging Practice:Use of a System Output Stream
Privacy Violation
Race Condition In Cross Functionality
Reflected XSS All Clients
Resource Injection
Scrypt Weak Salt Value
Server-Side Request Forgery
Setting Manipulation
SQL Injection
Stored XSS All Clients
System Information Leak
System Information Leak:External
System Information Leak:Internal
Use of Cryptographically Weak PRNG
Use of Hardcoded Password
Use of Weak RSA Keys
Weak Cryptographic Hash
Weak Cryptographic Hash:Hardcoded Salt
Weak Cryptographic Hash:User-Controlled Salt
Weak Cryptographic Signature:Insufficient Key Size
Weak Cryptographic Signature:User-Controlled Key Size
Weak Encryption
Weak Encryption:Inadequate RSA Padding
Weak Encryption:Insecure Initialization Vector
Weak Encryption:Insecure Mode of Operation
Weak Encryption:Insufficient Key Size
Weak Encryption:Stream Cipher

返回目录

Weak Encryption:User-Controlled Key Size

XML Injection

返回目录

漏洞名称	**Access Control:Database**
默认严重性	3
摘要	如果没有适当的 Access Control，X(文件) 中的 XX（函数） 方法就会在第 N 行上执行一个 SQL 语句，该语句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的主键的 SQL 语句，从而允许攻击者访问未经授权的记录。
解释	数据库 Access Control 错误在以下情况下发生： 1.数据从一个不可信数据源进入程序。 在这种情况下，X（函数） 会使用 X（文件） 中第 N 行的数据。 2.数据用来指定 SQL 查询中主键的值。 在这种情况下，Y(函数) 会使用 Y(文件) 中第 M 行的数据。 示例 1：以下代码将使用参数化语句，该语句会转义元字符，以防止 SQL Injection 漏洞，并构建和执行一个 SQL 查询。该 SQL 查询可以搜索与指定标识符 [1] 相匹配的清单。您可以从与当前授权用户关联的所有清单中选择该标识符。 ... id := request.FormValue("invoiceID") query := "SELECT * FROM invoices WHERE id = ?"; rows, err := db.Query(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提供的值，还不如在应用程序和数据库层上进行 Access Control。切勿允许用户在没有取得相应权限的情况下检索或修改数据库中的记录。确保每个访问数据库的查询都必须遵守此策略，而通常通过将当前授权用户名作为查询的一部分即可实现前述目的。 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。 ... user := GetAuthenticatedUser() id := request.FormValue("invoiceID") query := "SELECT * FROM invoices WHERE id = ? AND user = ?" rows, err := db.Query(query, id, user) ...
CWE	APSC-DV-000460 CAT I, APSC-DV-000470 CAT II, APSC-DV-002360 CAT II

返回目录

OWASP2017	A4 Insecure Direct Object References

返回目录

漏洞名称	**CGI 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） 中发送修改的数值即可更改输出数据，然后使用 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后此输入无需净化即可经代码写入控制台或 STDOUT。在某些情况下，此输出也会发送回用户的浏览器。 如果代码的控制台输出被应用程序用作网页的一部分（CGI 脚本通常是这种情况），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。 攻击者可能使用应用程序的合法访问权限提交修改后的数据，然后未经正确的净化便作为输出返回。然后，这些会被用于动态生成输出到通用输出编写器——在某些情况下，例如 CGI 脚本，此输出会直接发送到用户浏览器并触发攻击。 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。 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：

返回目录

	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。
CWE	CWE ID 79
OWASP2017	None

返回目录

漏洞名称	**Command Injection**
默认严重性	3
摘要	X(文件) 的第 N 行会利用由不可信赖的数据构建的命令来调用 X（函数）。这种调用会导致程序以攻击者的名义执行恶意命令。执行不可信赖资源中的命令，或在不可信赖的环境中执行命令，都会导致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，我们着重关注第一种情况，即攻击者控制命令的可能性。这种形式的 Command Injection 漏洞在以下情况下发生： 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） 中第 N 行的 X（函数） 之中。 <IfDef var="ConditionalDescriptions"> <ConditionalText condition="taint:number"> 在这种情况下，即使数据为数字类型，由于其未经验证仍可能被视为恶意内容，因此程序仍将报告漏洞，但是优先级值会有所降低。 </ConditionalText> </IfDef> 2.数据作为代表应用程序所执行命令的一个字符串或部分字符串使用。 在这种情况下，Y(函数) 会执行 Y(文件) 中第 M 行的命令。 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：以下代码会运行用户控制的命令。 cmdName := request.FormValue("Command") c := exec.Command(cmdName) c.Run()
建议	请勿让用户直接控制程序执行的命令。如果用户输入一定会影响到命令的运行，请仅从一个安全的命令集合中选择要使用的输入。如果输入中出现了恶意内容，请确保传递给命令执行函数的值默认限制在这一集合的安全选择范围之内，或由程序拒绝继续执行该操作。 在需要将用户输入用作程序执行命令的参数时，由于合法的参数值集合过大或是难以维护，这个方法通常都不切实际。在这种情况下，开发人员通常的做法是执行拒绝列表。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，任何一个定义不安全字符的列表通常都不完整，并且会严重地依赖于执行命令的系统。更好的方式

返回目录

	是创建一个字符列表，允许其中的字符出现在输入中。接受完全由这些经认可的字符组成的输入。 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建一个绝对路径。应对照一系列定义有效值的常量，始终针对从配置文件或者环境中读取的命令值和路径执行健全性检查。 有时您可以执行其他检查来确定这些资源是否被篡改。例如，如果一个配置文件为可写，程序可能会拒绝运行。如果编译时不清楚路径，程序会执行检查来检验这个二进制代码的身份。如果一个二进制代码始终属于某个特定用户，或者被分配了一组特定的访问权限，程序会在执行这个二进制代码前验证这些属性。 尽管可能无法完全阻止强大的攻击者为了控制程序执行的命令而对程序进行的攻击，但只要程序执行外部命令，请务必使用最小授权原则：不给予超过执行该命令所必需的权限。
CWE	CWE ID 77, CWE ID 78
OWASP2017	A1 Injection

返回目录

漏洞名称	**Connection String Parameter Pollution**
默认严重性	3
摘要	X(文件) 文件将未验证的数据传递给第 N 行的数据库连接字符串。攻击者可能会覆盖现有参数值、注入新的参数或利用直接得到的变量。如果在数据库连接中加入未验证的输入，可能会让攻击者覆盖请求参数的值。攻击者可能会覆盖现有参数值、注入新的参数或利用直接得到的变量。
解释	连接字符串参数污染 (CSPP) 攻击包括将连接字符串参数注入到其他现有参数中。这个漏洞类似于也可能会发生参数污染的 HTTP 环境中的漏洞（可能更广为人知）。然而，它可能还适用于其他地方，比如数据库连接字符串。如果应用程序没有正确地检查用户输入，则恶意用户可能会破坏应用程序的逻辑，以执行从窃取凭证到检索整个数据库等各种攻击。如果再向应用程序提交其他参数，且这些参数与现有参数的名称相同，则数据库可能会有下列一种反应： 它可能只从第一个参数中提取数据 它可能从最后一个参数中提取数据 它可能从所有参数中提取数据并将它们连接在一起 这取决于所使用的驱动程序、数据库类型乃至 API 的使用方法。 示例 1：以下代码使用 HTTP 请求中的输入来连接到数据库： ... password := request.FormValue("db_pass") db, err := sql.Open("mysql", "user:" + password + "@/dbname") ... 在这个例子中，程序员并没有考虑到攻击者可以提供 db_pass 参数，比如： "xxx@/attackerdb?foo=" 之后连接字符串变为： "user:xxx@/attackerdb?foo=/dbname" 这样会让应用程序连接到攻击者控制器数据库，让攻击者能够控制哪些数据要传回应用程序。
建议	建议检验可接受字符的允许列表。此外，您还可以使用 API（例如，mysql.Config.FormatDSN）来指定封装这些连接参数的连接字符串值，以防止这种类型的攻击。 示例 2：以下代码会对 HTTP 请求中的输入进行参数化来连接到数据库： ... password := request.FormValue("db_pass") config := mysql.Config{User: "user", Passwd: password, DBName: "dbname"} db, err := sql.Open("mysql", config.FormatDSN()) ...
CWE	CWE ID 235

返回目录

OWASP2017	A1 Injection

返回目录

漏洞名称	**Cookie Security:Cookie not Sent Over SSL**
默认严重性	3
摘要	程序在 X（文件） 中第 N 行创建了 Cookie，但未将 Secure 标记设置为 true。程序创建了 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 都具有 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 Network Sniffing 攻击，因此该 secure 标记有助于保护 Cookie 值的机密性。如果 Cookie 包含私人数据或会话标识符，或带有 CSRF 标记，那么该标记尤其重要。
	在这种情况下，程序会在 X（文件） 中第 N 行创建 cookie，但不会将 Secure 参数传递给 SetCookie()，或使用值 false 进行传递。
	示例 1：以下代码会在未设置 Secure 标记的情况下将 cookie 添加到响应中。
	cookie := http.Cookie{
	Name: "emailCookie",
	Value: email,
	}
	http.SetCookie(response, &cookie)
	...
	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。攻击者随后可截取未加密的网络信息流（通过无线网络时十分容易），从而危及 cookie 安全。

建议	对所有新 Cookie 设置 Secure 标记，以便指示浏览器不要以明文形式发送这些 Cookie。具体做法是将 true 作为关键字参数传递给 Secure。
	示例 2：以下代码会通过将 Secure 标记设置为 true 来更正 Example 1 中的错误。
	cookie := http.Cookie{
	Name: "emailCookie",
	Value: email,
	Secure: true,
	}
	http.SetCookie(response, &cookie)
	...

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	3
摘要	程序在 X(文件) 中的第 N 行创建了 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	浏览器支持 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标记。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。
	cookie := http.Cookie{
	Name: "emailCookie",
	Value: email,
	}
	...

建议	在创建 Cookie 时启用 HttpOnly 属性。具体做法是将 SetCookie() 调用中的 HttpOnly 参数设置为 true。
	示例 2：以下代码创建的 Cookie 与 Example 1 中的代码创建的相同，但这次会将 HttpOnly 参数设置为 true。
	cookie := http.Cookie{
	Name: "emailCookie",
	Value: email,
	HttpOnly: true,
	}
	...
	不要被 HttpOnly 欺骗，产生一种错误的安全感。此外，已开发出了绕过此安全功能的多种机制，因此它不完全有效。

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
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：假设您的一个安全应用程序部署在 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 := http.Cookie{
	Name: "sessionID",
	Value: getSessionID(),
	Domain: ".example.com",
	}
	...
	假设您在 http://insecure.example.com/ 上部署了另一个不太安全的应用程序，并且它存在 Cross-Site Scripting 漏洞。任何浏览到 http://insecure.example.com 的 http://secure.example.com 认证用户都面临着暴露来自 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 Secure.example.com 中的 Cookie。

建议	确保将 Cookie 域设置为具有尽可能高的限制性。
	示例 2：以下代码会显示如何将 Example 1 中的 Cookie 域设置为"secure.example.com"。
	cookie := http.Cookie{
	Name: "sessionID",
	Value: getSessionID(),
	Domain: "secure.example.com",
	}
	...

CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Path**
默认严重性	3
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：
	假设您在 http://communitypages.example.com/MyForum 上部署了一个论坛应用程序，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 := http.Cookie{
	Name: "sessionID",
	Value: sID,
	Expires: time.Now().AddDate(0, 0, 1),
	Path: "/",
	}
	...
	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 /MyForum 中的 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：以下代码显示如何针对“说明”部分中的示例将 Cookie 路径设置为“/MyForum”。
	cookie := http.Cookie{
	Name: "sessionID",
	Value: sID,
	Expires: time.Now().AddDate(0, 0, 1),
	Path: "/MyForum",
	}
	...

CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cross-Site Scripting:Inter-Component Communication**
默认严重性	4
摘要	文件 X(文件) 的第 N 行向一个 Web 浏览器发送了未验证的数据，从而导致该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。在组件间通信 XSS 的情况下，不可信赖的数据是从驻留在同一系统上的其他组件接收的数据。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） 中第 N 行的 X（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。
	在这种情况下，数据通过 Y(文件) 的第 M 行中的 Y(函数) 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 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有数据（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 Go 代码片段可从 HTTP 请求中读取用户名 user，并将其显示给用户。
	`func someHandler(w http.ResponseWriter, r *http.Request){`
	`  r.parseForm()`
	`  user := r.FormValue("user")`
	`  ...`
	`  fmt.Fprintln(w, "Username is: ", user)`
	`}`
	如果 user 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 user 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。

返回目录

	示例 2：下面的 Go 代码片段会根据一个特定的雇员 ID 来查询数据库，并显示出相应的雇员姓名。
	func someHandler(w http.ResponseWriter, r *http.Request){
	...
	row := db.QueryRow("SELECT name FROM users WHERE id =" + userid)
	err := row.Scan(&name)
	...
	fmt.Fprintln(w, "Username is: ", name)
	}
	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，这段代码看似没那么危险，因为 name 的值是从数据库中读取的，而且这些内容显然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，那么攻击者便能在用户的 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，并提高多个用户受此攻击影响的可能性。XSS 漏洞利用会在网站上为访问者提供一个“留言簿”，以此开始攻击。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 响应中反馈数据。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。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。
	- 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器中。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

	- 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读回到应用程序中，并包含在动态内容中。
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 应用程序必须验证所有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，加强应用程序的现有输入验证机制，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。因此，避免 XSS 漏洞的最佳方法是验证进入应用程序以及由应用程序传送至用户的所有数据。 针对 XXS 漏洞进行验证最安全的方式是，创建安全字符允许列表，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，编码时这些字符仍应被视为合法输入，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，许多 Web 浏览器仍会设法更正 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。这就是为何我们不鼓励使用拒绝列表作为阻止 XSS 的方法。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。

返回目录

	– 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，过滤内容的可视化呈现将发生改变。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。应用程序服务器为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以确认是否存在进行 Cross-Site Scripting 攻击必需的字

返回目录

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

返回目录

漏洞名称	**Cross-Site Scripting:Persistent**
默认严重性	4
摘要	文件 X(文件) 的第 N 行向一个 Web 浏览器发送了未验证的数据，从而导致该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 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） 中第 N 行的 X（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。
	在这种情况下，数据通过 Y(文件) 的第 M 行中的 Y(函数) 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 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有数据（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 Go 代码片段可从 HTTP 请求中读取用户名 user，并将其显示给用户。
	func someHandler(w http.ResponseWriter, r *http.Request){
	r.parseForm()
	user := r.FormValue("user")
	...
	fmt.Fprintln(w, "Username is: ", user)
	}
	如果 user 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 user 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。
	示例 2：下面的 Go 代码片段会根据一个特定的雇员 ID 来查询数据库，并显示出相应的雇员姓名。
	func someHandler(w http.ResponseWriter, r *http.Request){
	...

返回目录

	row := db.QueryRow("SELECT name FROM users WHERE id =" + userid) err := row.Scan(&name) ... fmt.Fprintln(w, "Username is: ", name) } 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，这段代码看似没那么危险，因为 name 的值是从数据库中读取的，而且这些内容显然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，那么攻击者便能在用户的 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，并提高多个用户受此攻击影响的可能性。XSS 漏洞利用会在网站上为访问者提供一个“留言簿”，以此开始攻击。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 响应中反馈数据。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。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。 - 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器中。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读回到应用程序中，并包含在动态内容中。

返回目录

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 应用程序必须验证所有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，加强应用程序的现有输入验证机制，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。因此，避免 XSS 漏洞的最佳方法是验证进入应用程序以及由应用程序传送至用户的所有数据。 针对 XXS 漏洞进行验证最安全的方式是，创建安全字符允许列表，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，编码时这些字符仍应被视为合法输入，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，许多 Web 浏览器仍会设法更正 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。这就是为何我们不鼓励使用拒绝列表作为阻止 XSS 的方法。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。

返回目录

	– 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，过滤内容的可视化呈现将发生改变。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。应用程序服务器为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以确认是否存在进行 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

返回目录

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

返回目录

漏洞名称	**Cross-Site Scripting:Poor Validation**
默认严重性	4
摘要	X(文件) 中的第 N 行会使用 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 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） 中第 N 行的 X（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。
	在这种情况下，数据通过 Y(文件) 的第 M 行中的 Y(函数) 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 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有数据（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 Go 代码片段可从 HTTP 请求中读取用户名 user，并将其显示给用户。
	func someHandler(w http.ResponseWriter, r *http.Request){
	r.parseForm()
	user := r.FormValue("user")
	...
	fmt.Fprintln(w, "Username is: ", html.EscapeString(user))
	}
	如果 user 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 user 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。

示例 2：下面的 Go 代码片段会根据一个特定的雇员 ID 来查询数据库，并显示出相应的雇员姓名。

```
func someHandler(w http.ResponseWriter, r *http.Request){
   ...
   row := db.QueryRow("SELECT name FROM users WHERE id =" +
userid)
   err := row.Scan(&amp;name)
   ...
   fmt.Fprintln(w, "Username is: ", html.EscapeString(name))
}
```

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，这段代码看似没那么危险，因为 name 的值是从数据库中读取的，而且这些内容显然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，那么攻击者便能在用户的 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，并提高多个用户受此攻击影响的可能性。XSS 漏洞利用会在网站上为访问者提供一个“留言簿”，以此开始攻击。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 响应中反馈数据。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。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。
- 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器中。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

返回目录

	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读回到应用程序中，并包含在动态内容中。
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 应用程序必须验证所有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，加强应用程序的现有输入验证机制，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。因此，避免 XSS 漏洞的最佳方法是验证进入应用程序以及由应用程序传送至用户的所有数据。 针对 XXS 漏洞进行验证最安全的方式是，创建安全字符允许列表，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，编码时这些字符仍应被视为合法输入，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，许多 Web 浏览器仍会设法更正 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。这就是为何我们不鼓励使用拒绝列表作为阻止 XSS 的方法。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，过滤内容的可视化呈现将发生改变。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。应用程序服务器为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以确认是否存在进行 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
摘要	文件 X(文件) 的第 N 行向一个 Web 浏览器发送了未验证的数据，从而导致该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 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） 中第 N 行的 X（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。
	在这种情况下，数据通过 Y(文件) 的第 M 行中的 Y(函数) 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 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有数据（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 Go 代码片段可从 HTTP 请求中读取用户名 user，并将其显示给用户。
	func someHandler(w http.ResponseWriter, r *http.Request){
	r.parseForm()
	user := r.FormValue("user")
	...
	fmt.Fprintln(w, "Username is: ", user)
	}
	如果 user 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 user 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。
	示例 2：下面的 Go 代码片段会根据一个特定的雇员 ID 来查询数据库，并显示出相应的雇员姓名。
	func someHandler(w http.ResponseWriter, r *http.Request){
	...

返回目录

	row := db.QueryRow("SELECT name FROM users WHERE id =" + userid) err := row.Scan(&name) ... fmt.Fprintln(w, "Username is: ", name) } 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，这段代码看似没那么危险，因为 name 的值是从数据库中读取的，而且这些内容显然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，那么攻击者便能在用户的 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，并提高多个用户受此攻击影响的可能性。XSS 漏洞利用会在网站上为访问者提供一个“留言簿”，以此开始攻击。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 响应中反馈数据。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。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。 - 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器中。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读回到应用程序中，并包含在动态内容中。

返回目录

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 应用程序必须验证所有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，加强应用程序的现有输入验证机制，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。因此，避免 XSS 漏洞的最佳方法是验证进入应用程序以及由应用程序传送至用户的所有数据。 针对 XXS 漏洞进行验证最安全的方式是，创建安全字符允许列表，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，编码时这些字符仍应被视为合法输入，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，许多 Web 浏览器仍会设法更正 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。这就是为何我们不鼓励使用拒绝列表作为阻止 XSS 的方法。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。

返回目录

	– 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，过滤内容的可视化呈现将发生改变。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。应用程序服务器为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以确认是否存在进行 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

返回目录

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

返回目录

漏洞名称	**Denial Of Service Resource Exhaustion**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XXX（方法） 分配的资源 XX（元素） 在被 Y（文件） 文件第 M 行中的 YYY（方法） 使用时容易使资源耗尽。
解释	拒绝服务 (DoS) 攻击会导致应用程序的某些（若非全部）服务方面无法使用。 计算资源受服务器规格的限制，例如内存、存储、CPU、带宽等。若不能正确地生成、管理和释放这些资源，可能导致系统出现意外故障，从而造成拒绝服务攻击。
建议	处理系统资源时一定要小心，例如从内存中的指针到文件系统中的文件，以避免可用资源可能被滥用。
CWE	CWE ID 400
OWASP2017	None

返回目录

漏洞名称	**Denial of Service:Regular Expression**
默认严重性	3
摘要	不受信任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，攻击者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	应用程序从 X（文件） 文件第 N 行的 XXX（方法） 中加载不可信任的对象数据 XX（元素），然后在 Y（文件） 文件第 YY（元素）行中反序列化此数据。
解释	可以修改已序列化的对象数据的攻击者可能能够控制目标代码的加载和执行。这可能使攻击者能够改变内部对象状态；消耗任意数量的数据资源，有效地导致 DoS（拒绝服务）攻击；绕过数据验证，从而利用后续漏洞；甚至可能在服务器上执行任意代码。 应用程序通常需要将对象数据输出为序列化格式，以便持久地保存到磁盘或通过网络传输。然后，应用程序需要反序列化二进制数据，并将其转换回内部对象数据。 但是未对二进制输入进行任何验证，便将不可信任的数据直接加载到任意内部对象中。此外，反序列化流程通常会根据输入数据执行自定义代码。
建议	如果可能，请避免在代码中反序列化不可信任的数据。 如果适用，考虑将对象字段标记为瞬态，以防止它们被反序列化。 实现自定义反序列化，并显式验证所有加载的数据并创建新对象，而不是直接反序列化对象。 覆写 ObjectInputStream.resolveClass() 方法以将可反序列化的类列入白名单，并禁止加载任意类。 对于不该反序列化但继承 Serializable 类的类可以显式声明一个最终 readObject() 方法，这始终抛出一个异常来阻止通过反序列化加载此类对象。 考虑使用数字签名或 HMAC 来执行签名验证，并仅允许使用经过身份验证的数字签名进行反序列化。 如果应用程序包含 commons-collections 库，或者删除此库（如果不需要），或者手动修补 JAR 文件以删除 "InvokerTransformer" 类（并充分测试这不会破坏应用程序）。
CWE	CWE ID 502
OWASP2017	None

返回目录

漏洞名称	**Dynamic Code Evaluation:Code Injection**
默认严重性	3
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支持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（如对活动用户对象执行简单的计算或修改与用户相关的值），就会出现代码注入漏洞。若不经过适当的验证，用户可能会指定程序员不打算执行的操作。
	示例：在此示例中，应用程序从命令行参数中检索要调用的函数的名称。
	
	...
	func beforeExampleCallback(scope *Scope){
	input := os.Args[1]
	if input{
	scope.CallMethod(input)
	}
	}
	...

建议	在任何时候，都应尽可能地避免解析动态代码。如果程序必须动态地解析代码，您可以通过尽可能限制程序中动态执行的代码数量，将这种攻击获得成功的可能性降到最低。将动态代码解析限制为基本编程语言的程序和特定于上下文的子集。该程序不得直接解析未经验证的用户输入。而改为采用间接方法：创建一个合法操作和数据对象的列表，用户必须选择其中的内容。使用这种方法，程序将不会执行用户提供的输入。
CWE	CWE ID 95, CWE ID 494
OWASP2017	A1 Injection

返回目录

漏洞名称	**Email Content Forgery**
默认严重性	4
摘要	用户在 X（文件） 文件第 N 行 XX（元素） 中提供的内容被插入 Y（文件） 文件第 M 行 YY（元素） 中的电子邮件内容中。
解释	允许操纵和伪造电子邮件的组件会使攻击者能够更改电子邮件的内容或主题字段，从而使攻击者能够从可信赖的源（即应用程序的邮件程序）提供恶意、错误或误导性内容对用户执行社会工程攻击。 应用程序通常会生成电子邮件与用户直接通信以进行审批、身份验证、定期更新等。通常，这些电子邮件是根据用户最初提供的值定制的，因此内容和标题都是使用用户提供的值推导的。 如果未在影响应用程序生成的电子邮件内容前对可能污染的输入进行净化，攻击者就可能修改或添加内容到电子邮件正文或其主题行。
建议	一定要使用来自可信任的数据源的值生成电子邮件。 不要依赖用户提供的参数来制作电子邮件。 如果确实需要使用用户输入创建自定义的、用户特定的电子邮件——请一定要净化这些值中的特殊字符和空白区域，并合理地限制电子邮件正文中用户输入的长度。
CWE	CWE ID 116
OWASP2017	None

返回目录

漏洞名称	**File Permission Manipulation**
默认严重性	3
摘要	该程序将调用 X(文件) 中第 N 行的 X（函数），其中含有攻击者可控制的值。允许用户输入以控制文件权限可能导致攻击者能够访问以其他方式保护的系统资源。允许用户输入以直接更改文件权限可能导致攻击者能够访问以其他方式保护的系统资源。

解释	当满足以下任一条件时，就会发生 File Permission Manipulation 错误:
	1.攻击者可能会指定操作中所用的路径，以修改文件系统上的权限。
	2.攻击者可能会指定文件系统上的操作所分配的权限。
	在这种情况下，攻击者可能会控制输入到程序的值（在 X（文件） 中第 N 行上的 X（函数）），该值会传递到 Y(文件) 中第 M 行上的文件系统操作 Y(函数) 中。
	示例 1：以下代码使用系统环境变量中的输入设置文件权限。如果攻击者可更改系统环境变量，则他们可能会使用该程序获得程序所处理文件的访问权限。如果程序依然易受 Path Manipulation 的攻击，那么攻击者可能会利用这一漏洞来访问系统中的任意文件。
	permissions := strconv.Atoi(os.Getenv("filePermissions"));
	fMode := os.FileMode(permissions)
	os.chmod(filePath, fMode);
	...

建议	防止此类文件权限操纵的最佳方式是允许用户设置文件权限。但是，如果必须确保用户能够指定文件的权限，则应该使用一种间接方法：即创建一个允许用户进行指定的合法文件权限的列表，并且仅允许用户从列表中进行选择。通过这种方式，用户提供的输入将绝对不会直接用于指定文件权限。
	示例 2：以下代码使用了一种间接方法以验证来自系统属性的输入（应设置默认权限掩码）。
	if p, err := strconv.Atoi(os.Getenv("filePermissions")); err == nil {
	if p == 0444 {
	fMode = os.FileMode(0444)
	} else {
	fMode = os.FileMode(0440)
	}
	}/settiu
	os.chmod("file", fMode)
	...
	在 Example 2 中，我们实际不会使用环境变量的用户输入。相反，我们会将它与指定的有效权限进行比较，然后将其用于权限中。此举可防止用于比较相等性的 API 包含 bug，该 bug 会通过如空字节注入等方式允许绕开。

返回目录

CWE	CWE ID 264, CWE ID 732
OWASP2017	None

返回目录

漏洞名称	**Formula Injection**
默认严重性	3
摘要	当调用 X(文件) 中第 N 行的 X（函数） 时，攻击者可能会控制写入到电子表格的数据，借此让用户打开某些电子表格处理器上的文件。攻击者可能会控制写入到电子表格的数据，借此让用户打开某些电子表格处理器上的文件。

解释	常用电子表格处理器（如 Apache OpenOffice Calc 和 Microsoft Office Excel）支持的公式运算非常强大，这可能会使攻击者控制电子表格而在底层系统上运行任意命令或在电子表格上泄漏敏感信息。
	例如，攻击者可能会将以下有效负载作为 CSV 字段的一部分注入：=cmd|'/C calc.exe'!Z0。如果打开电子表格的用户信任文档来源，他们可能就会接受电子表格处理器提供的所有安全提示信息，并使此有效负载（此处为打开 Windows 计算器）在其系统上运行。
	示例：以下示例使用未经检查的用户控制数据写入 csv 文件：
	func someHandler(w http.ResponseWriter, r *http.Request){
	r.parseForm()
	foo := r.FormValue("foo")
	...
	w := csv.NewWriter(file)
	w.Write(foo)
	}

建议	防止注入攻击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。 对于 formula injection，最好使用允许列表以确认仅包含字母数字字符。如果此方法不可行，至少要检验拒绝列表以避免以下字符：=、+、- 和 @。
CWE	CWE ID 1236
OWASP2017	A1 Injection

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。
CWE	CWE ID 547
OWASP2017	None

返回目录

漏洞名称	**Header Manipulation**
默认严重性	3
摘要	X(文件) 中的方法 XX（函数） 包含未经验证的数据，这些数据位于 HTTP 响应标头的第 N 行。这会招致各种形式的攻击，例如 Cache-Poisoning、Cross-Site Scripting、Cross-User Defacement、Page Hijacking、Cookie Manipulation 或 Open Redirect。HTTP 响应标头中包含未经验证的数据会招致 Cache-Poisoning、Cross-Site Scripting、Cross-User Defacement、Page Hijacking、Cookie Manipulation 或 Open Redirect 攻击。

解释	Header Manipulation 漏洞会在以下情况下发生：
	1.数据通过不可信来源进入 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） 中第 N 行的 X（函数） 之中。
	2.数据包含在未经验证就发送给 Web 用户的 HTTP 响应标头中。
	在这种情况下，数据通过 Y(文件) 的第 M 行中的 Y(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：攻击者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 响应标头中。
	示例：以下代码片段会从 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 标头中。
	...
	author := request.FormValue("AUTHOR_PARAM")
	cookie := http.Cookie{
	Name: "author",
	Value: author,
	Domain: "www.example.com",
	}
	http.SetCookie(w, &cookie)
	...
	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

返回目录

	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并将其传播给用户。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有数据（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 将会话和身份验证信息返回给攻击者，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产生两个响应的请求，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标头或正文中的敏感信息。 Cookie Manipulation：当与类似 Cross-Site Request Forgery 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 漏洞是在应用程序的输出中包含恶意数据时出现，因此合乎逻辑的做法是在应用程序输出数据前立即对其进行验证。然而，由于 Web 应用程序通常包含复杂且难以理解的代码，用以动态生成响应。因此，这一方法容易产生遗漏错误（遗漏验

返回目录

	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。 一旦在应用程序中确定了针对 Header Manipulation 攻击执行验证的正确点，以及验证过程中要考虑的特殊字符，下一个难题就是确定在验证过程中该如何处理各种特殊字符。应用程序应拒绝任何要添加到 HTTP 响应标头中且包含特殊字符的输入。
CWE	CWE ID 113
OWASP2017	A1 Injection

返回目录

漏洞名称	**Header Manipulation:Cookies**
默认严重性	3
摘要	X(文件) 中的方法 XX（函数） 包含未经验证的数据，这些数据位于 HTTP Cookie 的第 N 行。这样会招致 Cookie Manipulation 攻击并可能导致其他 HTTP Response Header Manipulation 攻击，如 Cache-Poisoning、Cross-Site Scripting、Cross-User Defacement、Page Hijacking 或 Open Redirect。在 Cookie 中包含未经验证的数据会引发 HTTP Response Header Manipulation 攻击，并可能导致 Cache-Poisoning、Cross-Site Scripting、Cross-User Defacement、Page Hijacking、Cookie Manipulation 或 Open Redirect。
解释	Cookie Manipulation 漏洞会在以下情况下发生： 1.数据通过不可信来源进入 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） 中第 N 行的 X（函数） 之中。 <IfDef var="ConditionalDescriptions"> <ConditionalText condition="taint:number"> 在这种情况下，即使数据为数字类型，由于其未经验证仍可能被视为恶意内容，因此程序仍将报告漏洞，但是优先级值会有所降低。 </ConditionalText> </IfDef> 2.数据包含在未经验证就发送给 Web 用户的 HTTP Cookie 中。 在这种情况下，数据通过 Y(文件) 的第 M 行中的 Y(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 Cross-Site Request Forgery 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 响应标头，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。为了成功实施 HTTP Response Splitting 漏洞，该应用程序必须允许将包含 CR（回车符，也可以由 %0d 或 \r 指定）和 LF（换行符，也可以由 %0a 或 \n 指定）字符的输入包含在标头中。攻击者不仅可以利用这些字符控制应用程序要发送的响应的剩余标头和正文，还可以创建完全受其控制的其他响应。

返回目录

	如今的许多现代应用程序服务器可以防止 HTTP 标头感染恶意字符。例如，如果尝试使用被禁用的字符设置标头，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标头，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，因此在设置带有用户输入的 HTTP 标头时仍需小心谨慎。
	示例：以下代码片段会从 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 标头中。
	...
	author := request.FormValue("AUTHOR_PARAM")
	cookie := http.Cookie{
	Name: "author",
	Value: author,
	Domain: "www.example.com",
	}
	http.SetCookie(w, &cookie)
	...
	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

返回目录

	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：攻击者控制了应用程序传送的响应后，就可以选择多种恶意内容并将其传播给用户。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有数据（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 将会话和身份验证信息返回给攻击者，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，攻击者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产生两个响应的请求，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标头或正文中的敏感信息。 Open Redirect：如果允许未验证的输入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（如 Cookie Manipulation）是在应用程序的输出中包含恶意数据时出现，因此合乎逻辑的做法是在应用程序输出数据前立即对其进行验证。然而，由于 Web 应用程序通常包含复杂且难以理解的代码，用以动态生成响应。因此，这一方法容易

返回目录

	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。 一旦在应用程序中确定了针对 Header Manipulation 攻击执行验证的正确点，以及验证过程中要考虑的特殊字符，下一个难题就是确定在验证过程中该如何处理各种特殊字符。应用程序应拒绝任何要添加到 HTTP 响应标头中且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 标头和 Cookie 的函数提供各种实现方式，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	**HTML5:MIME Sniffing**
默认严重性	3
摘要	应用程序会应用 MIME 探查算法，或不将 X-Content-Type-Options 设置为 nosniff。

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。
	编写 Web 应用程序时，请针对可能包含用户可控内容的每个页面使用 HTTP 标头 X-Content-Type-Options: nosniff。
	编写客户端应用程序时，不应使用 MIME 探查算法来确定服务器的响应 Content-Type。
	示例：以下代码使用 net.http.DetectContentType() 来确定响应 Content-Type：
	...
	resp, err := http.Get("http://example.com/")
	if err != nil {
	// handle error
	}
	defer resp.Body.Close()
	body, err := ioutil.ReadAll(resp.Body)
	content_type := DetectContentType(body)
	...

建议	为了确保服务器应用程序不易遭受 MIME 探查攻击，程序员可以： 1.在应用程序中，为所有页面全局设置 HTTP 标头 X-Content-Type-Options: nosniff。 2.为可能包含用户可控内容的每个页面设置必要标头。 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。 默认情况下，相同的框架全局均包含此标头。在这些框架中，确保不将此保护禁用。 编写客户端应用程序时，不应使用 MIME 探查算法来确定数据 Content-Type，而应靠随数据一同发送来的 Content-Type 标头来判断。
CWE	CWE ID 554
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Improper Error Handling**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 生成的错误未被后续应用程序代码正确地处理。
解释	如果未进行正确的检查，未处理的错误可能会导致意外行为，然后根据具体情况，最可能的场景就是未处理的错误累积时导致拒绝服务，或者在下游继续触发错误。被抑制的错误或未正确记录的错误很可能导致丢失相关信息、妨碍故障排除、影响代码质量和安全相关问题。 如果错误被抛出后未被处理，这表明错误是被有意抑制了，或者是通过某种逻辑流程使其从未得到积极的处理。如果错误被完全抑制或从未使用过，则错误是被显式抑制了——这意味着此错误未记录也未经过应用程序代码的逻辑验证。
建议	一定要记录错误，以保证可以获得与错误相关的信息 处理易出错代码时要执行逻辑检查，以避免操作失败时可能出现的意外行为 考虑所有极端情况，无论这些情况有多不可能发生都要考虑，以避免未来需要进行故障排除 在 Go 中，使用 panic() 或易出现 panic 的代码段（例如第三方库）时，始终考虑以下情况： recover() 必须通过 defer 关键字延迟——否则，recover() 的值将为 nil，然后当发生 panic 调用时没有什么将“捕捉”它 对 recover() 的调用必须直接在 deferred 函数中（匿名或已命名）。如果 deferred 函数调用了另一个含有 recover() 的函数，其值将为 nil 且抛出的 panic 将在延迟码已结束运行之后继续 带 panic 的 Go 例程无法通过其调用代码恢复。如果延迟的 recover() 调用位于在作为一个 Go 例程（通过 go）调用的易出现 panic 的代码调用之前，则延迟代码将被忽略，panic 将终止程序，并且每个正在进行的例程的堆栈跟踪都将被转储 recover() 调用的返回值应作为错误处理——要将这些值记录下来，且这些值应符合代码流以免导致 panic
CWE	CWE ID 248
OWASP2017	None

返回目录

漏洞名称	**Insecure Credential Storage Mechanism**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不安全的机制保存在 Y（文件） 文件第 M 行的 YY（元素） 中
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 不使用经过安全研究人员社区审查的标准化、公开的已知凭证存储解决方案，这会降低用于访问应用程序的凭证存储的安全性。 用于存储凭证的方法不安全。 应用程序没有通过已知的安全机制存储凭证。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 其他凭证存储方式都是不安全的，因为这些方法要么未经过安全研究人员的密码分析，要么已被破解或削弱。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness**
默认严重性	3
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性数据源，会产生 Insecure Randomness 错误。 在这种情况下，生成弱随机数的函数是 X（函数），它位于 X(文件)的第 N 行。 计算机是一种具有确定性的机器，因此无法产生真正的随机性。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：以下代码使用统计学的 PRNG 来创建 RSA 密钥。 import "math/rand" ... var mathRand = rand.New(rand.NewSource(1)) rsa.GenerateKey(mathRand, 2048) 该代码使用 rand.New() 函数生成 RSA 密钥的随机性。由于 rand.New() 是统计学的 PRNG，攻击者很容易猜到其生成的值。
建议	当不可预测性至关重要时，例如，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。）
CWE	CWE ID 338
OWASP2017	None

返回目录

漏洞名称	**Insecure Scrypt Parameters**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不安全的机制保存在 Y（文件） 文件第 M 行的 YY（元素） 中
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 现代 hash 函数通常有成本（或类似）参数，用户可以设置该参数以确定在 hash 密码时应使用多少计算能力。值越高意味着计算越慢，但也越难被攻击，因为每次尝试都需要更长的时间。 虽然 scrypt 专门用于阻止暴力攻击和其他快速攻击，但如果定义了错误的输入参数也会显著弱化算法的输出。 用于存储凭证的方法不安全。 设置太低的成本值可能会显著降低密码 hash 的安全性。 参数 N、r 和 p 未设置安全值。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 选择合适的成本参数时，需要同时考虑底层的 hash 算法和密钥派生函数的目的。最常见的目的是交互式验证登录，然后即可相应地设置查询的成本参数阈值。 在 scrypt 中，这些参数是 N、r 和 p。N 定义 CPU/内存成本参数，r 定义块大小参数，p 是并行化参数。N 和 r 定义执行一次尝试的难度，p 定义应该使用多少处理器来执行计算。确定为安全的当前值组定义是 N >= 2^15、r >= 8 和 p >= 2。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**Insecure SSL:Server Identity Verification Disabled**
默认严重性	3
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：此应用程序没有明确地验证服务器证书。
	
	...
	config := &tls.Config{
	// Set InsecureSkipVerify to skip the default validation
	InsecureSkipVerify: true,
	...
	}>
	conn, err := tls.Dial("tcp", "example.com:443", conf)
	..
	
	尝试连接到 example.com 时，此应用程序会随时接受服务器颁发的任何证书。当服务器被黑客攻击发生 SSL 连接中断时，此应用程序可能会泄漏用户敏感信息。

建议	进行 SSL 连接时，不要忘记服务器验证检查。请务必根据所使用的库验证服务器身份并建立安全的 SSL 连接。
	示例 2：此应用程序明确地验证服务器证书。
	
	...
	config := &tls.Config{
	// Default Value
	InsecureSkipVerify: false,
	RootCAs: rootCAs,
	Certificates: []tls.Certificate{cert},
	...
	}>
	conn, err := tls.Dial("tcp", "example.com:443", conf)
	...

CWE	CWE ID 297
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport**
默认严重性	3
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	所有基于 HTTP、FTP 或 gopher 的通信均未经过验证和加密。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）设置 Web 服务器。
	helloHandler := func(w http.ResponseWriter, req *http.Request) {
	io.WriteString(w, "Hello, world!\n")
	}
	http.HandleFunc("/hello", helloHandler)
	log.Fatal(http.ListenAndServe(":8080", nil))

建议	应尽可能使用 HTTPS 等安全协议与服务器交换数据。 示例 2：以下示例通过 HTTPS 协议设置 Web 服务器。 http.HandleFunc("/", func(w http.ResponseWriter, req *http.Request) { io.WriteString(w, "Hello, TLS!\n") }) log.Fatal(http.ListenAndServeTLS(":8443", "cert.pem", "key.pem", nil))
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insufficient Bcrypt Cost**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 中定义的成本参数过低。
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 现代 hash 函数通常有成本（或类似）参数，用户可以设置该参数以确定在 hash 密码时应使用多少计算能力。值越高意味着计算越慢，但也越难被攻击，因为每次尝试都需要更长的时间。 将 bcrypt 成本参数设置为过低的值会显著降低生成的 hash 的安全性。 用于存储凭证的方法不安全。 设置太低的成本值可能会显著降低密码 hash 的安全性。 bcrypt 函数的成本参数过低。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 选择合适的成本参数时，需要同时考虑底层的 hash 算法和密钥派生函数的目的。最常见的目的是交互式验证登录，然后即可相应地设置查询的成本参数阈值。 Bcrypt 的成本参数应设置为默认值（10）或更高。减少此值会降低根据密码计算 hash 所需的计算工作量，从而削弱产生的 hash，使攻击者更容易破解密码。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**Insufficient Output Length**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不足的输出长度保存在 Y（文件） 文件第 M 行的 YY（元素） 中
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 在这种情况下，密钥派生函数的输出长度太低。这可能导致冲突，使不同密码的 hash 计算结果相同，造成身份验证成功，而这是不应该发生的。 用于存储凭证的方法不安全。 密钥派生函数的输出长度太低。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 然而，使用这些函数无法保证安全的凭证存储机制。为了避免冲突，密钥派生函数输出的安全长度应为 32 字节或更高。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**JSON Injection**
默认严重性	4
摘要	在 X(文件) 的第 N 行中，XX（函数） 方法将未经验证的输入写入 JSON。攻击者可以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.数据从一个不可信数据源进入程序。 在这种情况下，数据进入 X（文件） 中第 N 行的 X（函数） 之中。 2.将数据写入到 JSON 流。 在这种情况下，Y(函数) 会在 Y(文件) 的第 M 行中写入 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，则攻击者可以更改 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。有时，JSON Injection 可以导致 Cross-Site Scripting 或 Dynamic Code Evaluation。 示例 1：以下代码将非特权用户（这些用户具有“默认”角色，与之相反，特权用户具有“管理员”角色）的用户帐户身份验证信息从用户控制的输入变量 username 和 password 序列化为位于 ~/user_info.json 的 JSON 文件： ... func someHandler(w http.ResponseWriter, r *http.Request){ r.parseForm() username := r.FormValue("username") password := r.FormValue("password") ... jsonString := `{ "username":"` + username + `", "role":"default"

返回目录

	"password":"` + password + `",
	}`
	...
	f, err := os.Create("~/user_info.json")
	defer f.Close()
	jsonEncoder := json.NewEncoder(f)
	jsonEncoder.Encode(jsonString)
	}
	
	由于代码使用字符串串联来执行 JSON 序列化，将不会对 username 和 password 中的不可信赖数据进行验证以转义与 JSON 相关的特殊字符。这样，用户就可以任意插入 JSON 密钥，这可能会更改已序列化的 JSON 结构。在本例中，如果非特权用户 mallory（密码为 Evil123!）在输入其用户名时附加了 ","role":"admin，则最终保存到 ~/user_info.json 的 JSON 将为：
	
	{
	"username":"mallory",
	"role":"default",
	"password":"Evil123!",
	"role":"admin"
	}
	
	在没有进一步验证反序列化 JSON 值是否有效的情况下，应用程序会无意中为用户分配 mallory“管理员”特权。

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。 防止 JSON 注入的最佳方法是采用一些间接手段：创建一份必须由用户选择的合法资源名称的列表。通过这种方法，就不能直接使用用户输入来指定资源名称。 有时，这种方法并不可行，因为这样一份合法资源名称的列表太大，管理难度大。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 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**Key Management:Empty Encryption Key**
默认严重性	4
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	请勿使用空加密密钥，因为这样将会大幅减弱由良好的加密算法提供的保护，而且还会大大增加解决问题的难度。一旦问题代码投入使用，要更改空加密密钥，就必须进行软件修补。如果受空加密密钥保护的帐户被盗用，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 X（函数）。 示例 1：以下代码使用空加密密钥执行 AES 加密： ... key := []byte(""); block, err := aes.NewCipher(key) ... 不仅任何可以访问此代码的人可以确定它使用的是空加密密钥，而且任何掌握最基本破解技术的人都更有可能成功解密所有加密数据。应用程序一经发布，要更改空加密密钥，就必须进行软件修补，因为他们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。在系统中的任何位置采用明文的形式存储加密密钥（空或非空），会导致任何有足够权限的人均可读取并无意中误用加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Empty HMAC Key**
默认严重性	3
摘要	空的 HMAC 密钥，如第 N 行的 X(文件) 内 X（函数） 中使用的密钥，可能会以无法轻松修复的方式危及系统安全。空 HMAC 密钥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	请勿使用空 HMAC 密钥。HMAC 的加密强度依赖于密钥的大小，后者用于计算和验证消息的身份验证值。使用空密钥会削弱 HMAC 函数的加密强度。 在这种情况下，X（函数） 在 X(文件) 的第 N 行上使用空 HMAC 密钥。 示例 1：下列代码使用空密钥来计算 HMAC: import "crypto/hmac" ... hmac.New(md5.New, []byte("")) ... Example 1 中的代码可能会成功运行，但有权访问该代码的任何人都能确定它使用的是空 HMAC 密钥。一旦程序发布，除非修补该程序，否则无法更改此空 HMAC 密钥。心怀不轨的雇员可以利用手中掌握的信息访问权限破坏 HMAC 函数。另外，Example 1 中的代码还容易受到伪造和密钥恢复攻击的侵害。
建议	1.请勿使用空 HMAC 密钥。底层散列函数的加密强度依赖于 HMAC 密钥的大小和强度。使用随机种子设置的加密性很强的伪随机生成器随机生成 HMAC 密钥。定期刷新 HMAC 密钥。 2.HMAC 密匙的长度至少应该与底层散列函数输出的长度相匹配。
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	4
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	请勿对加密密钥进行硬编码，因为这样所有项目开发人员都能查看该加密密钥，而且还会大大增加解决问题的难度。一旦代码投入使用，要更改加密密钥，就必须进行软件修补。如果受加密密钥保护的帐户被盗用，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 X（函数）。 示例 1：以下代码使用硬编码加密密钥： ... key := []byte("lakdsljkalkjlksd"); block, err := aes.NewCipher(key) ... 任何可访问该代码的人都能访问加密密钥。应用程序一经发布，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	加密密钥不应为空。加密密钥应加以模糊化，并在外部数据源中进行管理。在系统中的任何位置采用明文的形式存储加密密钥，会导致任何有足够权限的人均可读取并无意中误用加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded HMAC Key**
默认严重性	3
摘要	硬编码 HMAC 密钥（如 X(文件) 内第 N 行的 X（函数） 中使用的密钥）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。 硬编码 HMAC 密钥会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	绝对不要对 HMAC 密钥进行硬编码。HMAC 的加密强度依赖于密钥的保密性，后者用于计算和验证消息的身份验证值。采用硬编码处理 HMAC 密钥允许任何可访问源的人都能查看它，所以会削弱函数的加密强度。 在这种情况下，X（函数） 在 X(文件) 的第 N 行上包含硬编码 HMAC 密钥。 示例 1：下列代码使用硬编码密钥来计算 HMAC: import "crypto/hmac" ... hmac.New(sha256.New, []byte("secret")) ... 此代码会成功运行，但有权访问该来源的任何人都能获得此 HMAC 密钥。一旦程序发布，除非修补该程序，否则无法更改硬编码的 HMAC 密钥“secret”。心怀不轨的雇员可以利用手中掌握的信息访问权限破坏 HMAC 函数。
建议	1.绝对不要对 HMAC 密钥进行硬编码。底层散列函数的加密强度依赖于 HMAC 密钥的大小和强度。使用随机种子设置的加密性很强的伪随机生成器随机生成 HMAC 密钥。定期刷新 HMAC 密钥。 2.如果发现硬编码 HMAC 密钥，则立即修补程序并采取必要的措施，以限制因密钥暴露造成的任何损害。
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	4
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被盗用，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 X（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： ... aes.NewCipher(nil) ... 任何可访问该代码的人都能确定其是否使用了 null 加密密钥。此外，掌握基本破解技术的任何人都更有可能成功解密所有加密数据。应用程序一经发布，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），会导致任何有足够权限的人均可读取并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	**Log Forging**
默认严重性	3
摘要	在 X(文件) 文件第 N 行中，该程序将未经验证的用户输入写入日志。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内容注入日志。

解释	在以下情况下会发生 Log Forging 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） 中第 N 行的 X（函数） 之中。
	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 Y(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的历史记录或查看事务。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趋势信息。
	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，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：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。如果该值未被解析为整数，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。
	func someHandler(w http.ResponseWriter, r *http.Request){
	r.parseForm()
	name := r.FormValue("name")
	logout := r.FormValue("logout")
	...
	if (logout){
	...
	} else {
	log.Printf("Attempt to log out: name: %s logout: %s", name, logout)
	}
	}

返回目录

	如果用户为 logout 提交字符串“twenty-one”，而且他可以创建一个名为“admin”的用户，则日志中会记录以下条目： Attempt to log out: name: admin logout: twenty-one 但是，如果攻击者可以创建用户名"admin+logout:+1++++++++++++++++++++++"，则日志中将记录以下条目： Attempt to log out: name: admin logout: 1 logout: twenty-one
建议	使用间接方法防止 Log Forging 攻击：创建一组与不同事件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。 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。在使用默认的 trigger-error 函数时，在日志文件中将不会显示换行符。
CWE	CWE ID 117
OWASP2017	A1 Injection

返回目录

漏洞名称	**Log Forging (debug)**
默认严重性	3
摘要	X(文件) 文件中的方法 XX（函数） 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内容注入日志。

解释	在以下情况下会发生 Log Forging 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） 中第 N 行的 X（函数） 之中。
	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 Y(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趋势信息。
	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，那么攻击者可以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，攻击者可以利用受到破坏的日志文件掩护其跟踪轨迹，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。如果该值未被解析为整数，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 idValue string
	idValue = req.URL.Query().Get("id")
	num, err := strconv.Atoi(idValue)
	if err != nil {
	sysLog.Debug("Failed to parse value: " + idValue)
	}
	...
	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

返回目录

	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。
	以下重写的 Example 1 代码使用与 strconv.NumError 对应的预定义日志条目：
	...
	errMsg := "Failed to parse idValue. The input is required to be an integer value."
	...
	var idValue string
	idValue = req.URL.Query().Get("id")
	num, err := strconv.Atoi(idValue)
	if err != nil {
	sysLog.Debug(errMsg)
	}
	..
	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 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。 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 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 HttpOnly Cookie**
默认严重性	4
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。 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。
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。
CWE	CWE ID 1004
OWASP2017	None

返回目录

漏洞名称	**Missing Secure Cookie**
默认严重性	4
摘要	Y（文件） 文件第 M 行中生成一个 cookie 并发送到了客户端，但缺少安全特性。没有安全标记的 cookie 可能会通过纯文本连接传输，可能会将其暴露给正在观察此通道的任何人。
解释	没有安全特性的 cookie 可能会通过未加密的通道传输，这可能使攻击者可以观察 cookie 并可能冒充用户，然后代表用户执行操作。 Cookie 可以通过加密的和未加密的通道传递。设置安全的 cookie 特性可以指示浏览器永远不要通过不安全的通道提交此 cookie。
建议	总是使用 HTTPS 进行安全的 HTTP 通信 使用安全通道时，cookie 一定要有安全特性标记
CWE	CWE ID 614
OWASP2017	None

返回目录

漏洞名称	**Open Redirect**
默认严重性	3
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入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。如果允许未验证的输入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。
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。应用程序利用重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定向到可能由攻击者控制的任意 URL 时，就会发生 Open Redirect 漏洞。 攻击者可能利用 Open Redirect 漏洞诱骗用户访问某个可信赖的站点的 URL，并将他们重定向到恶意站点。攻击者通过对 URL 进行编码，使最终用户很难注意到重定向的恶意目标，即使将这一目标作为 URL 参数传递给可信赖的站点时也会发生这种情况。因此，Open Redirect 通常被作为网络钓鱼诈骗的一部分被滥用，攻击者通过这种方式来获取最终用户的敏感数据。 这种情况下，系统会通过 X（文件） 中第 N 行的 X（函数） 接受客户端即将被重定向到的 URL。 数据通过 Y(文件) 中第 M 行的 Y(函数) 传送。 示例：以下代码会在用户单击相关链接时，指示用户的浏览器打开从 dest 请求参数解析的 URL。 ... strDest := r.Form.Get("dest") http.Redirect(w, r, strDest, http.StatusSeeOther) ... 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，该用户就有可能会单击该链接，因为他认为该链接会转到可信站点。然而，当受害者单击该链接时，Example 1 中的代码就会将浏览器重定向到 "http://www.wilyhacker.com"。 很多用户都被告知，要始终检查通过电子邮件收到的 URL，以确保链接指向他们知道的可信赖的站点。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%72%2E%63%6F%6D" 那么即使是精明的最终用户也可能被骗至访问该链接。
建议	不要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：创建一份合法 URL 列表，必须由用户选择其中的内容。利用这种方法，就绝不会直接使用用户提供的输入来指定要重定向到的 URL。

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 的列表过于庞大，维护难度过大。在这种情况下，使用类似的方法来限制用户可以重定向到的域，这至少可以防止攻击者向用户发送恶意外部站点。
CWE	CWE ID 601
OWASP2017	None

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	A6-Security Misconfiguration

返回目录

漏洞名称	**Password Management**
默认严重性	3
摘要	X(文件) 中的 XX（函数） 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的属性或配置文件中时，会发生 password management 问题。 在这种情况下，密码会通过 X（文件） 中第 N 行的 X（函数） 读取到程序中，并用于访问 Y(文件) 中第 M 行的 Y(函数) 资源。 示例 1：以下代码可以从 JSON 文件中读取密码，并使用该密码来设置请求的授权标头： ... file, _ := os.Open("config.json") decoder := json.NewDecoder(file) decoder.Decode(&values) request.SetBasicAuth(values.Username, values.Password) ... 这个代码可以顺利运行，但是任何对 config.json 具有访问权限的人都能读取 values.Password 中的值。任何心怀不轨的雇员可以利用手中掌握的信息访问权限入侵系统。
建议	切勿以明文形式存储密码。要求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。至少，密码要先经过 hash 处理再存储。 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，这对于对安全性要求较高的环境来说是远远不够的。较为安全的解决方法是自行创建新机制，而这也是如今唯一可行的方法。
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	4
摘要	空密码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	为密码变量指定空字符串绝非好方法。如果使用空密码成功通过其他系统的验证，那么相应帐户的安全性很可能会被减弱，原因是其接受了空密码。如果在可以为变量指定合法值之前，空密码仅仅是一个占位符，那么它将给任何不熟悉代码的人造成困惑，而且还可能导致出现意外控制流路径方面的问题。 在这种情况下，在对 X(文件) 第 N 行中的 X（函数） 的调用中发现空密码。 示例：以下代码尝试使用空密码连接到数据库。 ... response.SetBasicAuth(usrName, "") ... 如果此示例中的代码成功执行，则表明数据库用户帐户“scott”配置有一个空密码，攻击者可以轻松地猜测到该密码。一旦程序发布，要更新此帐户以使用非空密码，就需要对代码进行更改。
建议	始终从加密的外部资源读取存储的密码值，并为密码变量指定有意义的值。确保从不使用空密码或 null 密码来保护敏感资源。
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	4
摘要	Hardcoded Password 会削弱系统安全性，并会导致无法轻易修正出现的安全问题。
解释	请勿使用硬编码方式处理密码。通过硬编码方式处理密码不仅会让所有项目开发人员都可以看到密码，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，在 X(文件) 中的第 N 行上，对密码进行了硬编码处理。 示例：以下代码对密码进行了硬编码处理： password := "letmein" ... response.SetBasicAuth(usrName, password) 该代码可以正常运行，但是任何有该代码访问权限的人都可以获取该密码。一旦程序发布，除非修补该程序，否则可能无法更改密码“tiger”。雇员可以利用手中掌握的信息访问权限入侵系统。更糟的是，如果攻击者能够访问应用程序的二进制码，他们就可以利用多种常用的反编译器来访问经过反汇编的代码，而在这些代码中恰恰包含着用户使用过的密码值。
建议	绝不能对密码进行硬编码处理，通常应对其进行模糊化，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密码，会造成任何有足够权限的人均可读取和无意中误用密码。
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	4
摘要	Null 密码会削弱安全性。
解释	请勿为密码变量分配 null，因为这可能会使攻击者绕过密码验证，或是表明资源受空密码保护。 在这种情况下，在对 X(文件) 中第 N 行的 X（函数） 的调用中会发现 null 密码。
建议	始终从加密的外部资源读取存储的密码值，并为密码变量指定有意义的值。确保从不使用空密码或 null 密码来保护敏感资源。
CWE	CWE ID 259
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Weak Cryptography**
默认严重性	3
摘要	调用 X（函数） 使密码模糊化并不能提供任何意义上的保护。采用普通的编码方式使密码模糊化无法保护密码。
解释	当密码以明文形式存储在应用程序的属性或配置文件中时，会发生 password management 问题。程序员试图通过编码函数使密码模糊化，以解决 password management 问题，例如使用 64 位基址编码方式，但都无法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 X（函数） 读取到程序中，并用于访问 Y(文件) 中第 M 行的 Y(函数) 资源。 示例：以下代码可以从 JSON 文件中读取密码，并使用该密码来设置请求的授权标头。 ... file, _ := os.Open("config.json") decoder := json.NewDecoder(file) decoder.Decode(&values) password := base64.StdEncoding.DecodeString(values.Password) request.SetBasicAuth(values.Username, password) ... 该代码可以正常运行，但是任何对 config.json 具有访问权限的人都能读取 password 的值，并且很容易确定这个值是否经过 64 位基址编码。任何心怀不轨的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。较为安全的解决方法是自行创建新机制，而这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	**Path Manipulation**
默认严重性	3
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 X（函数） 进入程序的值，这个程序会使用这个值来访问 Y(文件) 中第 M 行的 Y(函数)。 <IfDef var="ConditionalDescriptions"> <ConditionalText condition="taint:number"> 在这种情况下，即使数据为数字类型，由于其未经验证仍可能被视为恶意内容，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 1: 下面的代码使用来自于 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。 rName := "/usr/local/apfr/reports/" + req.FormValue("fName") rFile, err := os.OpenFile(rName, os.O_RDWR|os.O_CREATE, 0755) defer os.Remove(rName); defer rFile.Close() ... 示例 2: 下面的代码使用来自于配置文件的输入来决定打开哪个文件，并返回给用户。如果程序以足够的权限运行，且恶意用户能够篡改配置文件，那么他们可以通过程序读取系统中以扩展名 .txt 结尾的任何文件。 ...

返回目录

	config := ReadConfigFile() filename := config.fName + ".txt"; data, err := ioutil.ReadFile(filename) ... fmt.Println(string(data))
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	CWE ID 22, CWE ID 73
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Path Manipulation:Zip Entry Overwrite**
默认严重性	3
摘要	如果调用 X(文件) 中第 N 行的 X（函数），攻击者将可以在系统上的任何位置随意进行文件写入。如果允许用户输入控制文件系统操作所用的路径，攻击者将可以在系统上对文件进行随意覆盖。

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 Entry Overwrite 错误。
	示例 1：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。
	func Unzip(src string, dest string) ([]string, error) {
	var filenames []string
	r, err := zip.OpenReader(src)
	if err != nil {
	return filenames, err
	}
	defer r.Close()
	for _, f := range r.File {
	// Store filename/path for returning and using later on
	fpath := filepath.Join(dest, f.Name)
	filenames = append(filenames, fpath)
	if f.FileInfo().IsDir() {
	// Make Folder
	os.MkdirAll(fpath, os.ModePerm)
	continue
	}
	
	// Make File
	if err = os.MkdirAll(filepath.Dir(fpath), os.ModePerm); err != nil {
	return filenames, err
	}
	outFile, err := os.OpenFile(fpath, os.O_WRONLY|os.O_CREATE|os.O_TRUNC, f.Mode())
	if err != nil {
	return filenames, err
	}
	
	rc, err := f.Open()
	if err != nil {
	return filenames, err
	}

返回目录

	_, err = io.Copy(outFile, rc)
	// Close the file without defer to close before next iteration of loop
	outFile.Close()
	rc.Close()
	if err != nil {
	return filenames, err
	}
	}
	return filenames, nil
	}
	在 Example 1 中，在对此条目中的数据执行读取/写入函数之前未验证 f.Name。如果 ZIP 文件最初放置在基于 Unix 的计算机的 "/tmp/" 目录中，并且 ZIP 条目为 "../etc/hosts"，而应用程序在必要的权限下运行，则它将覆盖系统的 hosts 文件。从而可能会使该计算机的流量进入攻击者所需的任何位置，例如返回至攻击者的计算机。

建议	防止通过 ZIP 文件进行 Path Manipulation 的最佳方式是采取一种间接的方法。创建 ZIP 条目可以写入的合法路径名称的列表，然后将其写入与 ZIP 条目位置相符的文件。通过这种方式，ZIP 文件中的用户输入绝不会直接用于指定文件位置。
	在此示例中，这种方式可能并不可行。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，任何这样一份黑名单都不可能是完整的，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。
	在这种情况下，诸如 path.filepath.Clean() 等 API 可以用于检索规范形式的文件路径，从而可用于检查文件将写入到的目录。
	示例 2：以下代码通过检查规范形式的文件路径来修复 Example 1。
	func Unzip(src string, dest string) ([]string, error) {
	var filenames []string
	r, err := zip.OpenReader(src)
	if err != nil {
	return filenames, err
	}
	defer r.Close()
	for _, f := range r.File {
	// Store filename/path for returning and using later on
	fpath := filepath.Join(dest, f.Name)
	// Check canonicalized file path
	if !strings.HasPrefix(fpath, filepath.Clean(dest)+string(os.PathSeparator)) {
	return filenames, fmt.Errorf("%s: illegal file path", fpath)
	}

返回目录

	filenames = append(filenames, fpath)
	if f.FileInfo().IsDir() {
	// Make Folder
	os.MkdirAll(fpath, os.ModePerm)
	continue
	}
	
	// Make File
	if err = os.MkdirAll(filepath.Dir(fpath), os.ModePerm); err != nil {
	return filenames, err
	}
	outFile, err := os.OpenFile(fpath, os.O_WRONLY|os.O_CREATE|os.O_TRUNC, f.Mode())
	if err != nil {
	return filenames, err
	}
	
	rc, err := f.Open()
	if err != nil {
	return filenames, err
	}
	_, err = io.Copy(outFile, rc)
	// Close the file without defer to close before next iteration of loop
	outFile.Close()
	rc.Close()
	if err != nil {
	return filenames, err
	}
	}
	return filenames, nil
	}

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	None

返回目录

漏洞名称	**PBKDF2 Insufficient Iteration Count**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 中定义的迭代参数过低。
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 现代 hash 函数通常有成本（或类似）参数，用户可以设置该参数以确定在 hash 密码时应使用多少计算能力。值越高意味着计算越慢，但也越难被攻击，因为每次尝试都需要更长的时间。 将 PBKDF2 迭代参数设置为过低的值会显著降低生成的 hash 的安全性。 用于存储凭证的方法不安全。 设置太低的成本值可能会显著降低密码 hash 的安全性。 PBKDF2 函数的迭代参数过低。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛 (PHC) 获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 选择合适的成本参数时，需要同时考虑底层的 hash 算法和密钥派生函数的目的。最常见的目的是交互式验证登录，然后即可相应地设置查询的成本参数阈值。 PBKDF2 的迭代参数至少应设置为 10.000 或更高的值。降低此值会降低根据密码计算 hash 所需的计算工作量，从而削弱产生的 hash，使攻击者更容易破解密码。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**PBKDF2 Weak Salt Value**
默认严重性	4
摘要	在 Y（文件） 文件第 M 行 YY（元素） 处为 PBKDF2 提供的加密 salt 不安全。
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 为 PBKDF2 salt 使用弱伪随机数生成器的值或静态值将大幅降低生成的 hash 的安全性。 用于存储凭证的方法不安全。 凭证存储函数的 salt 参数不安全。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 各种 salt 用于防止攻击者创建彩虹表并执行 hash 搜索（避免对用户密码 hash 的暴力破解）。各个密码使用唯一且不可预测的 salt 也可防止攻击者知道哪些用户共享相同的密码。最佳做法是使用密码加密伪随机数生成器（CSPRNG）获取 salt 值。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**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。 应用程序代码已经设置了内容安全策略；但是，它设置的策略过于宽松。
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	**Poor Logging Practice:Use of a System Output Stream**
默认严重性	3
摘要	使用 os.Stdout 或 os.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。使用 os.Stdout 或 os.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：通常，开发人员学习编写的第一个 Go 程序如下所示：
	...
	func main(){
	fmt.Println("Hello World")
	}
	多数开发人员深入了解 Go 的许多精妙之处后，一部分人员始终使用 fmt.Println() 编写进行标准输出的消息。
	问题在于，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、日志标识符、时间戳，以及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。
	开发者普遍接受结构化日志记录，但许多人在“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初始开发阶段之后生成的，那么使用 os.Stdout 或 os.Stderr 进行日志记录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 Go 日志记录 API，而不使用 os.Stdout 或 os.Stderr。
	示例 2：例如，您可以使用 Logrus 在 Example 1 中重写“hello world”程序，如下所示：
	import (
	"os"
	"github.com/sirupsen/logrus"
	)
	func main(){
	var logger = logrus.New()
	file, err := os.OpenFile("logrus.log",
	os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
	if err == nil {
	logger.Out = file
	} else {
	logger.Info("Failed to log to file, using default stderr")
	}
	logger.Info("Hellow World")

返回目录

	}
CWE	CWE ID 398
OWASP2017	None

返回目录

漏洞名称	**Privacy Violation**
默认严重性	3
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。对私人信息（如客户密码或社会保障号码）处理不当会损害用户隐私，这通常是一种非法行为。

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 X（函数）。
	2.程序将数据写到一个外部位置，例如控制台、文件系统或网络。
	在这种情况下，数据将传递到 Y(文件) 中第 M 行的 Y(函数)。
	示例 1：以下代码包含一个语句，该语句以日志文件的形式将各条记录信息写入数据库。存储的值中，有一个是 GetPassword() 函数的返回值，该函数会返回与该帐户关联且由用户提供的明文密码。
	pass = GetPassword();
	...
	if err != nil {
	log.Printf('%s: %s %s %s', id, pass, type, tsstamp)
	}
	Example 1 中的代码会将明文密码记录到应用程序的事件日志中。虽然许多开发人员认为事件日志是存储数据的安全位置，但这不是绝对的，特别是涉及到隐私问题时。
	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器
	- 从合作伙伴或其他第三方间接获取
	有时，未贴上私人标签的数据在其他上下文中也有可能成为私人信息。例如，学生的学号通常不视为私人信息，因为学号中并没有明确且公开可用的信息来反映学生的个人信息。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。
	虽然不安全地处理私人数据有多种形式，但是常见的风险来自于盲目的信任。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]。

返回目录

	为能够妥善应对此类备受瞩目的信息盗取事件，私人数据的收集与管理正日益规范化。各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] 这些规范业已存在，但侵犯隐私的情况仍时有发生。
建议	安全和隐私需求出现冲突时，通常应优先考虑隐私。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如何处理各种私人数据。若贵组织受到联邦或者州法律的制约，应确保贵组织的隐私保护原则足够严格，能够满足相关法律法规的要求。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	**Race Condition In Cross Functionality**
默认严重性	3
摘要	Y（文件） 文件中的 YYY（方法） 方法使用了 YY（元素），会被其他并发功能以对线程不安全的方式访问，从而导致此资源产生竞争条件。
解释	最好的情况是，竞争条件可能导致准确性问题、值被覆盖或可能导致拒绝服务攻击的意外行为。在最坏的情况下，它可能会使攻击者通过重用可控制的竞争条件获得有利条件，从而检索数据或绕过安全机制。 条件竞争就是一个公共资源实例被多个并发逻辑进程使用。如果这些逻辑进程都尝试检索和更新资源，但没有及时的管理系统（例如锁定），就会发生条件竞争。 例如如果一个资源可以返回特定的值给进程以进行进一步的编辑，但值又被第二个进程更新，导致原始进程的数据不再有效，这就是一种条件竞争。原始进程编辑不正确的值并将其更新回资源后，第二个进程的更新就会被覆盖并丢失。
建议	在应用程序的各个并发进程之间共享资源时，一定要保证这些资源的线程安全，或者实现锁定机制以保证预期的并发活动。
CWE	CWE ID 362
OWASP2017	None

返回目录

漏洞名称	**Reflected XSS All Clients**
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	**Resource Injection**
默认严重性	3
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 问题：
	1.攻击者可以指定用于访问系统资源的标识符。
	例如，攻击者也许可以指定用于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，程序可能允许攻击者将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 X（函数） 进入程序的值，这一数值可以通过 Y(文件) 中第 M 行的 Y(函数) 访问系统资源。
	注意：如果 Resource Injection 涉及存储在文件系统中的资源，则可以将其报告为名为 Path Manipulation 的不同类别。有关这一漏洞的更多详细信息，请查看 Path Manipulation 说明。
	示例：以下代码使用从 HTTP 请求中读取的设备名称来进行连接，以便将与 fd 关联的套接字绑定至设备。
	`func someHandler(w http.ResponseWriter, r *http.Request){`
	`r.parseForm()`
	`deviceName := r.FormValue("device")`
	`...`
	`syscall.BindToDevice(fd, deviceName)`
	`}`
	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文件系统交互的方法中使用时，具有很大风险。同样，对于创建远程连接的函数而言，包含 URL 和 URI 的数据也具有很大风险。

建议	防止资源注入的最佳方法是采用一些间接手段：创建一份用户可以指定的合法资源名列表，并且规定用户只能从该列表中进行选择。通过这种方法，就不能直接使用由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	CWE ID 99

返回目录

OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Scrypt Weak Salt Value**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 为 scrypt 提供的 salt 不安全。
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 为 scrypt salt 使用弱伪随机数生成器的值或静态值将大幅降低生成的 hash 的安全性。 用于存储凭证的方法不安全。 凭证存储函数的 salt 参数不安全。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 各种 salt 用于防止攻击者创建彩虹表并执行 hash 搜索（避免对用户密码 hash 的暴力破解）。各个密码使用唯一且不可预测的 salt 也可防止攻击者知道哪些用户共享相同的密码。最佳做法是使用密码加密伪随机数生成器（CSPRNG）获取 salt 值。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**Server-Side Request Forgery**
默认严重性	3
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) 中的第 N 行调用 X（函数）。 示例：在下列示例中，攻击者将能够控制服务器连接至的 URL。 url := request.Form.Get("url") res, err =: http.Get(url) ... 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:// - 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 系统上使用 file:// 方案和 UNC 路径来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：例如创建一份必须由客户选择的合法资源名的列表，并且规定用户只能选择其中的文件名。通过这种方法，就不能直接使用由用户提供的输入来指定资源名称。

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可以造成的损害。
CWE	CWE ID 918
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Setting Manipulation**
默认严重性	3
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 X（函数） 进入程序，并流向 Y(文件) 中第 M 行的 Y(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：以下代码片段使用用户控制的数据设置环境变量。 ... catalog := request.Form.Get("catalog") path := request.Form.Get("path") os.Setenv(catalog, path) ... 在本例中，攻击者可以设置任何任意环境变量，并影响其他应用程序的运行方式。 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赖的数据，以防止攻击者控制某些敏感数值。虽然攻击者控制这些数值的影响不一定显现，但是不要低估攻击者的攻击力。
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，应用程序预期输入会在相对小的数值范围内。如果可能的话，请从预定的安全数值集合中选择输入值。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个安全选项的默认设置。即使无法事先了解安全数值集合，通常也可以验证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	**SQL Injection**
默认严重性	3
摘要	文件 X(文件) 中第 N 行调用了通过不可信赖的来源的输入构建的 SQL 查询。通过这种调用，攻击者能够修改语句的含义或执行任意 SQL 命令。通过不可信赖的数据源的输入构建动态 SQL 语句，攻击者就能够修改语句的含义或者执行任意 SQL 命令。
解释	SQL Injection 错误会在以下情况下出现: 1.数据从一个不可信数据源进入程序。 在这种情况下，数据进入 X（文件） 中第 N 行的 X（函数） 之中。 2.数据用于动态地构造 SQL 查询。 在这种情况下，数据将传递到 Y(文件) 中第 M 行的 Y(函数)。 示例 1: 下列代码可动态地构建并执行一个 SQL 查询，用来搜索与指定名称相匹配的条目。该查询仅会显示条目所有者与当前经过身份验证的用户的名称一致的条目。 ... rawQuery := request.URL.Query() username := rawQuery.Get("userName") itemName := rawQuery.Get("itemName") query := "SELECT * FROM items WHERE owner = " + username + " AND itemname = " + itemName + ";" db.Exec(query) ... 查询计划执行以下代码: SELECT * FROM items WHERE owner = <userName> AND itemname = <itemName>; 但是，由于该查询是由代码动态构造的，由一个常数基本查询字符串和一个用户输入字符串连接而成，因此只有在 itemName 不包含单引号字符时，该查询才能正常运行。如果一个用户名为 wiley 的攻击者为 itemName 输入字符串“name' OR 'a'='a”，则该查询会变成: SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name' OR 'a'='a'; 如果添加条件 OR 'a'='a'，where 子句的值将始终为 true，这样该查询在逻辑上就等同于一个更为简单的查询: SELECT * FROM items; 通常，查询必须仅返回已通过身份验证的用户所拥有的条目，而通过以这种方式简化查询，攻击者就可以规避这一要求。现在，查询会返回存储在 items 表中的所有条目，而不论其指定所有者是谁。

返回目录

	示例 2：此示例说明了将不同的恶意值传递给 Example 1.中构造和执行的查询所带来的影响。如果一个用户名为 wiley 的攻击者为 itemName 输入字符串“name'; DELETE FROM items; --”，则该查询就会变为以下两个查询： SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name'; DELETE FROM items; --' 许多数据库（包括 Microsoft(R) SQL Server 2000）允许同时执行由多个用分号分隔的 SQL 语句。在不允许批量执行用分号分隔的语句的 Oracle 和其他数据库服务器上，此攻击字符串只会导致错误；但是在支持批量执行的数据库上，此类型攻击可以使攻击者针对数据库执行任意命令。 注意末尾的一对连字符（--）；这在大多数数据库服务器上都表示该语句剩余部分将视为注释，不会加以执行。[4]。在这种情况下，可通过注释字符删除修改后的查询遗留的末尾单引号。而在不允许通过这种方式使用注释的数据库上，攻击者通常仍可使用类似于 Example 1 中所用的技巧进行攻击。如果攻击者输入字符串 "name'; DELETE FROM items; SELECT * FROM items WHERE 'a'='a"，将创建以下三个有效语句： SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name'; DELETE FROM items; SELECT * FROM items WHERE 'a'='a'; 避免 SQL injection 攻击的传统方法之一是，作为一个输入验证问题来处理，只接受列在安全值允许列表中的字符，或者识别并避免列在潜在恶意值列表（拒绝列表）中的字符。检验允许列表是一种非常有效的方法，它可以强制执行严格的输入验证规则，但是参数化的 SQL 语句所需的维护工作更少，而且能提供更好的安全保障。而对于通常采用的执行拒绝列表方式，由于总是存在一些小漏洞，所以并不能有效地防止 SQL Injection 攻击。例如，攻击者可以： - 将未引用的字段作为目标 - 寻找方法以绕过某些需要转义的元字符 - 使用存储过程隐藏注入的元字符 手动转义 SQL 查询输入中的字符有一定的帮助，但是并不能完全保护您的应用程序免受 SQL Injection 攻击。 防范 SQL Injection 攻击的另外一种常用解决方法是使用存储过程。虽然存储过程可以阻止某些类型的 SQL Injection 攻击，但是对于绝大多数攻击仍无能为力。存储过程有助于避免 SQL Injection 攻击的常用方式是限制可传入存储过程参数的语句类型。但是，有许多方法都可以绕过这一限制，许多危险的语句仍可以传入存储过程。所以再次强

返回目录

	调，存储过程可以避免一些漏洞，但是并不能完全确保您的应用程序不受 SQL Injection 攻击。
建议	造成 SQL Injection 漏洞的根本原因在于攻击者可以更改 SQL 查询的上下文，导致程序员原本要作为数据解释的数值却被解释为命令。构造 SQL 查询后，程序员知道哪些字符应作为命令解释，哪些字符应作为数据解释。参数化 SQL 语句可以防止直接窜改上下文，避免几乎所有的 SQL Injection 攻击。参数化 SQL 语句是用常规的 SQL 字符串构造的，但是当需要添加用户提供的数据时，它们就需要使用捆绑参数，这些捆绑参数是一些占位符，用来存放随后插入的数据。换言之，捆绑参数可以使程序员明确告知数据库哪些字符应被视为命令，哪些字符应被视为数据。这样，当程序准备执行某个语句时，它可以为数据库指定运行时值，供每一个捆绑参数使用，而不存在将数据解释为修改命令的风险。 下面是对前面例子的重写，改为使用参数化 SQL 语句（替代用户输入连续的字符串），如下所示： ... rawQuery := request.URL.Query() username := rawQuery.Get("userName") itemName := rawQuery.Get("itemName") ... db.Exec("SELECT * FROM items WHERE owner = ? AND itemname = ?", username, itemName) ... 更加复杂的情况常常出现在报表生成代码中，因为这时需要通过用户输入来改变 SQL 语句的结构，比如在 WHERE 子句中添加动态约束条件。不要因为这一需求，就无条件地接受连续的用户输入，从而创建查询字符串。当必须要根据用户输入来改变命令结构时，可以使用间接的方法来防止 SQL Injection 攻击：创建一个合法的字符串集合，使其对应于可能要加入到 SQL 语句中的不同元素。在构造语句时，可使用来自用户的输入，以便从应用程序控制的值集合中进行选择。
CWE	CWE ID 89
OWASP2017	A1 Injection

返回目录

漏洞名称	**Stored XSS All Clients**
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	**System Information Leak**
默认严重性	3
摘要	X(文件) 中的 XX（函数） 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	当程序通过输出流或者日志功能揭示系统数据或调试信息时，就会发生信息泄漏。 在这种情况下，X(文件) 中的第 N 行调用 X（函数）。 示例 1：以下代码会将路径环境变量输出到标准错误流： path := os.Getenv("PATH") ... log.Printf("Cannot find exe on path %s\n", path) 根据系统配置，此信息可能会转储到控制台、写入日志文件或公开给用户。有时，攻击者可以基于错误消息确定应用程序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，搜索路径可能会暗示操作系统的类型、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。
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。请留意，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	**System Information Leak:External**
默认严重性	3
摘要	对于 X(文件)，在调用第 N 行上的 X（函数） 过程中，程序可能会显示系统数据或调试信息。由 X（函数） 揭示的信息可能帮助攻击者制定攻击计划。揭示系统数据或调试信息可帮助攻击者了解系统并制定攻击计划。

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。
	在这种情况下，XX（函数） 会产生系统数据或调试信息，而 X(文件) 中第 N 行的 YY（函数） 会泄露前述信息。
	示例 1：以下例子会通过 HTTP 响应泄露系统信息。
	func handler(w http.ResponseWriter, r *http.Request) {
	host, err := os.Hostname()
	...
	fmt.Fprintf(w, "%s is busy, please try again later.", host)
	}
	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有哪些。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。请留意，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 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
摘要	X(文件) 中的 XX（函数） 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	通过日志或打印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） 中第 N 行的 X（函数） 中的数据会通过 Y(文件) 中第 M 行的 Y(函数) 流出程序。 示例 1：以下代码会将路径环境变量输出到标准错误流： path := os.Getenv("PATH") ... log.Printf("Cannot find exe on path %s\n", path) 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有哪些。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，搜索路径可能会暗示操作系统的类型、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。
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。请留意，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	**Use of Cryptographically Weak PRNG**
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 338
OWASP2017	None

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	None

返回目录

漏洞名称	**Use of Weak RSA Keys**
默认严重性	4
摘要	应用程序使用 Y（文件） 文件第 M 行的弱加密算法 YY（元素） 保护 X（文件） 文件第 N 行的敏感信息 XX（元素）。
解释	使用弱的或过时的加密不能为敏感数据提供足够的保护。获得访问加密数据权限的攻击者可能会使用密码分析或强力攻击来破解加密。这样，攻击者可能会窃取用户密码和其他个人数据。这可能导致用户假冒身份或身份盗用。 应用程序使用了被认为过时的弱算法，因为相对容易破解。这些过时的算法容易受到多种类型的攻击，包括暴力破解。
建议	通用指南： 一定要使用强的现代算法进行加密、进行 hash 计算等。 不要使用弱的、过时的或淘汰的算法。 一定要根据具体要求选择正确的加密机制。 应使用专用密码保护方案保护密码，例如 bcrypt、scrypt、PBKDF2 或 Argon2。 具体建议： 不要使用 SHA-1、MD5 或任何其他弱的 hash 算法来保护密码或个人数据。相反，需要安全 hash 时，要使用较强的 hash，例如 SHA-256。 不要使用 DES、3DES、RC2 或任何其他弱加密算法来保护密码或个人数据。而要使用较强的加密算法来保护个人数据，例如 AES。 不要使用 ECB 等弱加密模式，也不要依赖不安全的默认值。要显式设置较强的加密模式，例如 GCM。 对于对称加密，请使用至少 256 位的密钥长度。
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Weak Cryptographic Hash**
默认严重性	2
摘要	不要在安全性关键的上下文中使用弱加密散列值，因为无法保证数据完整性。
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。密码分析研究揭示了这些算法中存在的根本缺陷，不建议将其用于安全性关键的上下文中。不要依赖 MD 和 RIPEMD 散列算法来保证安全性。有效破解 MD 和 RIPEMD 散列的技术已得到广泛使用。对于 SHA-1，目前的破坏技术仍需要极高的计算能力，因此难以实现。然而，攻击者已发现了该算法的弱点，破坏该算法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，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。未来研究也可能会揭示这些算法的安全缺陷。
CWE	CWE ID 328
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Hardcoded Salt**
默认严重性	3
摘要	Hardcoded salt 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	采用硬编码处理 salt 绝非一个好方法。这不仅是因为所有项目开发人员都可以使用 hardcoded salt 来查看该 salt，而且还会使解决这一问题变得极其困难。一旦该代码投入使用，则无法轻易更改该 salt。如果攻击者知道 salt 的值，他们就可以计算出该应用程序的“彩虹表”，并轻松地确定散列值。 在这种情况下，在对 X(文件) 第 N 行中的 X（函数） 的调用中发现硬编码 salt。 示例 1：下列代码使用了 hardcoded salt： ... salt := "2!@$(5#@532@%#$253I5#@$" password := get_password() sha256.Sum256([]byte(salt + password) ... 此代码成功运行，但有权访问此代码的任何人都可以访问 salt。一旦程序发布，将无法更改名为“2!@$(5#@532@%#$253I5#@$”的 salt。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 salt 进行硬编码。最好是对 salt 加以模糊化，并在外部数据源中进行管理。在系统中的任意位置采用明文形式存储 salt 会使拥有足够权限的任何人都能够读取并可能误用该 salt。 示例 2：下列代码使用由系统管理员配置的 salt 变量： ... salt := get_salt() password := get_password() sha256.Sum256([]byte(salt + password) ...
CWE	CWE ID 760
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:User-Controlled Salt**
默认严重性	3
摘要	X(文件) 中的 XX（函数） 方法包括在第 N 行的加密散列中使用的用户控制的 salt 值。攻击者可以指定一个空 salt，既能轻松确定散列值，又可以泄露有关程序如何执行加密散列的信息。不要接受生成加密散列的方法中的 salt 的用户输入。

解释	在以下情况下会发生 Weak Cryptographic Hash: 用户控制 salt 问题将在以下情况下出现：
	1.数据通过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） 中第 N 行的 X（函数） 之中。
	2.用户控制的数据包括在 salt 中，或完全用作加密散列函数中的 salt。
	在这种情况下，在 Y(文件) 中第 M 行的 Y(函数) 使用该数据。
	如同许多软件安全漏洞一样，Weak Cryptographic Hash: 用户控制 Salt 是到达终点的一个途径，其本身并不是终点。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递到应用程序，然后这些数据被用作加密散列函数中的全部或部分 salt。
	用户定义的 salt 可以实现各种不同的攻击：
	1.攻击者可以利用这一漏洞，指定一个空 salt 作为散列数据。然后攻击者可以使用许多不同的散列算法快速控制数据，以泄露有关您的应用程序中使用的散列实现的信息。这样，通过限制所用散列的特定变体，可以更轻松地“破解”其他数据值。
	2.如果攻击者可以操纵其他用户的 salt，或者诱骗其他用户使用空 salt，这使他们能够计算应用程序的“彩虹表”，并轻松地确定散列值。
	示例 1：以下代码使用用户控制 salt 进行密码散列：
	func someHandler(w http.ResponseWriter, r *http.Request){
	r.parseForm()
	salt := r.FormValue("salt")
	password := r.FormValue("password")
	...
	sha256.Sum256([]byte(salt + password))
	}
	Example 1 中的代码将成功运行，但任何有权使用此功能的人可以通过修改 salt 环境变量来操纵用于对密码执行散列的 salt。此外，此代码还会使用 Sum256 加密散列函数，而该函数不应该用于对密码执行加密散列。一旦程序发布，撤消与用户控制的 salt 相关的问题就会非常困难，因为很难知道恶意用户是否确定了密码散列的 salt。

建议	绝不要实施用户控制的 salt，即使是部分也不可以，也不能是硬编码的。通常情况下，必须对 salt 加以模糊化，并在外部数据源中进行管

返回目录

	理。在系统中的任意位置采用明文形式存储 salt 会使拥有足够权限的任何人都能够读取并可能误用该 salt。
CWE	CWE ID 328, CWE ID 760
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Signature:Insufficient Key Size**
默认严重性	4
摘要	X(文件) 文件中的 XX（函数） 方法使用了强大的加密签名算法，但密钥长度不够，从而导致签名更容易受到强力攻击。原本强大的加密签名算法如果使用的密钥长度不够，就会更加容易受到强力攻击。
解释	当前的密码指南建议，RSA 和 DSA 算法使用的密钥长度至少为 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 1024 位 DSA 签名密钥。 ... dsa.GenerateParameters(params, rand.Reader, dsa.L1024N160) privatekey := new(dsa.PrivateKey) privatekey.PublicKey.Parameters = *params dsa.GenerateKey(privatekey, rand.Reader) ...
建议	确保 RSA 和 DSA 签名密钥的长度不少于 2048 位。 示例 2：以下代码可生成 2048 位 DSA 签名密钥。 ... dsa.GenerateParameters(params, rand.Reader, dsa.L2048N224) privatekey := new(dsa.PrivateKey) privatekey.PublicKey.Parameters = *params dsa.GenerateKey(privatekey, rand.Reader) ...
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Signature:User-Controlled Key Size**
默认严重性	3
摘要	X(文件) 中的 XX（函数） 函数包括的用户输入在第 N 行的加密签名算法所使用的密钥大小参数的范围内。采用密钥大小的加密签名函数可以接收受污染的密钥大小值。
解释	通过允许用户控制的值来确定密钥大小，攻击者可以指定一个空密钥，从而可以修改确保加密数据完整性的加密签名。即使要求使用非零值，攻击者也仍然可以指定尽可能低的密钥大小值，降低加密数据的完整性。 在以下情况下会发生 Weak Cryptographic Hash: 用户控制密钥大小问题将在以下情况下出现： 1.数据通过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） 中第 N 行的 X（函数） 之中。 2.用户控制的数据被用作加密签名函数中密钥大小参数的全部或一部分 在这种情况下，在 Y(文件) 中第 M 行的 Y(函数) 使用该数据。 如同许多软件安全漏洞一样，Weak Cryptographic Signature: 用户控制密钥大小是到达终点的一个途径，其本身并不是终点。从本质上看，这些漏洞是显而易见的：攻击者将恶意数据传送到应用程序，这些数据随后被用作用于执行加密的密钥大小值的全部或一部分。 当前的密码指南建议，RSA 和 DSA 算法使用的密钥长度至少为 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。即使将用户输入用作部分密钥大小，也会削弱签名的安全性，进而削弱加密数据的完整性。 示例 1：下面的代码通过用户控制的密钥大小参数生成 DSA 签名密钥： ... dsa.GenerateParameters(params, rand.Reader, key_len) privatekey := new(dsa.PrivateKey) privatekey.PublicKey.Parameters = *params dsa.GenerateKey(privatekey, rand.Reader) ... 用户几乎不需要指定 key_len 的能力。在此类情况下，您应该验证它不仅是一个数值而且还处于密钥大小值的合理范围内。对于大多数用例，请选择一个足够大的硬编码密钥大小。
建议	确保 RSA 和 DSA 签名密钥的长度不少于 2048 位。 示例 2：以下代码可生成 2048 位 DSA 签名密钥： ... dsa.GenerateParameters(params, rand.Reader, dsa.L2048N224)

返回目录

	privatekey := new(dsa.PrivateKey) privatekey.PublicKey.Parameters = *params dsa.GenerateKey(privatekey, rand.Reader) ...
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption**
默认严重性	3
摘要	调用 X(文件) 中第 N 行的 X（函数） 使用的是弱加密算法，无法保证敏感数据的保密性。标识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 位密钥要克服巨大的计算障碍，但今天，攻击者借助常用设备就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:Inadequate RSA Padding**
默认严重性	4
摘要	X(文件) 中的 XX（函数） 方法执行不带 OAEP 填充模式的公钥 RSA 加密，因此加密机制比较脆弱。公钥 RSA 加密在不使用 OAEP 填充模式下执行，因此加密机制比较脆弱。
解释	实际中，使用 RSA 公钥的加密通常与某种填充模式结合使用。该填充模式的目的在于防止一些针对 RSA 的攻击，这些攻击仅在执行不带填充模式的加密时才起作用。 示例 1：以下代码通过使用 PKCS#1 v1.5 填充模式的 RSA 公钥执行加密： ... import "crypto/rsa" ... plaintext := []byte("Attack at dawn") cipherText, err := rsa.EncryptPKCS1v15(rand.Reader, &k.PublicKey, plaintext) ...
建议	为安全使用 RSA，在执行加密时必须使用 OAEP（最优非对称加密填充模式）。 示例 2：以下代码通过使用 OAEP 填充模式的 RSA 公钥执行加密： ... import "crypto/rsa" ... plaintext := []byte("Attack at dawn") cipherText, err := rsa.EncryptOAEP(sha256.New(), rand.Reader, &k.PublicKey, plaintext, []byte("message")) ...
CWE	CWE ID 780
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insecure Initialization Vector**
默认严重性	3
摘要	使用加密伪随机数生成器创建初始化矢量。
解释	使用加密伪随机数生成器创建初始化矢量 (IV)。否则，生成的密码文本可预测性会更高，也更容易受到 Dictionary 攻击。 示例 1：以下代码可重新使用密钥作为 IV： import ("crypto/aes" "crypto/cipher" "crypto/rand") ... block, err := aes.NewCipher(key) ... mode := cipher.NewCBCEncrypter(block, key) mode.CryptBlocks(ciphertext[aes.BlockSize:], plaintext) 当使用密钥作为 IV 时，攻击者能够恢复密钥，从而能够解密数据。

建议	将足够长度的初始化矢量 (IV) 与适当的随机数据源中的字节结合使用。
	示例 2：以下代码使用 crypto/rand 程序包创建足够随机的 IV：
	import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
	"io"
	)
	...
	block, err := aes.NewCipher(key)
	...
	iv := ciphertext[:aes.BlockSize]
	if _, err := io.ReadFull(rand.Reader, iv); err != nil {
	panic(err)
	}
	mode := cipher.NewCBCEncrypter(block, iv)
	mode.CryptBlocks(ciphertext[aes.BlockSize:], plaintext)

CWE	CWE ID 329
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insecure Mode of Operation**
默认严重性	4
摘要	X(文件) 中的函数 XX（函数） 在第 N 行上将密码加密算法用于不安全的操作模式。请勿将密码加密算法用于不安全的操作模式。

解释	块密码操作模式是一种算法，用来描述如何重复地应用密码的单块操作，以安全地转换大于块的数据量。一些操作模式包括电子代码本 (ECB)、密码块链 (CBC)、密码反馈 (CFB) 和计数器 (CTR)。
	ECB 模式本质上较弱，因为它会对相同的明文块生成一样的密文。
	CBC 模式容易受到密文填塞攻击。CTR 模式由于没有这些缺陷，使之成为一个更好的选择。
	示例 1：以下代码将 AES 密码用于 CBC 模式：
	...
	block, err := aes.NewCipher(key)
	if err != nil {
	panic(err)
	}
	ciphertext := make([]byte, aes.BlockSize+len(plaintext))
	iv := ciphertext[:aes.BlockSize]
	if _, err := io.ReadFull(rand.Reader, iv); err != nil {
	panic(err)
	}
	mode := cipher.NewCBCEncrypter(block, iv)
	mode.CryptBlocks(ciphertext[aes.BlockSize:], plaintext)
	...

建议	加密大于块的数据时，避免使用 ECB 和 CBC 操作模式。CBC 模式效率较低，并且在和 SSL 一起使用时会造成严重风险 [1]。请改用 CCM (Counter with CBC-MAC) 模式，或者如果更注重性能，则使用 GCM（Galois/Counter Mode）模式（如可用）。
	示例 2：以下代码将 AES 密码用于 GCM 模式：
	...
	block, err := aes.NewCipher(key)
	if err != nil {
	panic(err.Error())
	}
	nonce := make([]byte, 12)
	if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
	panic(err.Error())
	}
	aesgcm, err := cipher.NewGCM(block)
	if err != nil {
	panic(err.Error())

返回目录

	} ciphertext := aesgcm.Seal(nil, nonce, plaintext, nil) ...
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
摘要	X(文件) 中的 XX（函数） 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	当前的加密指南建议，RSA 算法使用的密钥长度至少应为 2048 位。对于对称加密，密钥长度应至少为 128 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 1024 位 RSA 密钥： ... myPrivateKey := rsa.GenerateKey(rand.Reader, 1024); ...
建议	最低限度下，确保 RSA 密钥长度不少于 2048 位。针对接下来几年对加密有严格要求的应用程序，请使用密码长度为 4096 位的密钥。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： ... myPrivateKey := rsa.GenerateKey(rand.Reader, 2048); ... 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	**Weak Encryption:Stream Cipher**
默认严重性	4
摘要	X(文件) 中的方法 XX（函数） 使用流密码。如果加密数据要存储在磁盘上，或者多次使用相同的密钥，那么这种方法就不太安全。如果加密数据要存储在磁盘上，或者多次使用密钥，那么使用流密码就比较危险。
解释	流密码容易受到“密钥重新使用”攻击，又名“两次性密码本”攻击。当多次使用相同的密钥时，会产生这种类型的漏洞，因为可以轻易地对两个密码文本字符串进行异或运算并使密钥失效，从而只剩下异或后的明文。由于人类语言的格式化方式，通常可以轻松恢复两条原始消息。 为了阻止上述攻击，需要使用新的初始化向量 (IV)，因此流密码不适合于存储加密数据，因为它意味着： 1) 磁盘扇区用作 IV： 由于在每次需要修改存储数据时都需要重新使用相同的 IV，因此这种方法并不安全。 2) 使用可将新的 IV 映射到磁盘扇区的复杂系统： 维护难度大。这种方法需要 IV 不断更新，需要使用户无法读取，需要密码文本占用的磁盘空间比未加密的明文更多。 基于上述两点，使用流密码而不是块密码来存储加密数据十分不利。 流密码的另一个问题是，它们不会提供身份验证，因此容易受到“位翻转”攻击。一些块密码（例如“CTR”）同样容易受到这些攻击，因为它们与流密码的工作原理相似。 示例 1：以下代码可创建流密码，然后将其与常量 IV 一起用于加密数据，再存储到磁盘上： import ("crypto/aes" "crypto/cipher" "os") ... iv = b'1234567890123456' CTRstream = cipher.NewCTR(block, iv) CTRstream.XORKeyStream(plaintext, ciphertext) ... f := os.Create("data.enc") f.Write(ciphertext) f.Close() 在 Example 1 中，由于 iv 设置为常量初始化向量，因此容易受到重用攻击。
建议	不要使用流密码来存储加密数据，而是始终使用随机初始化向量。

返回目录

	示例 2：以下代码可借助随机 IV 创建分组密码，然后用于加密数据，之后再存储该数据：
	import (
	"crypto/aes"
	"crypto/rand"
	"crypto/cipher"
	"os"
	"io"
	)
	...
	iv := make([]byte, aes.BlockSize)
	if _, err := io.ReadFull(rand.Reader, iv); err != nil {
	panic(err)
	}
	CBCencrypt := cipher.NewCBCEncrypter(block, iv)
	CBCencrypt.CryptBlocks(ciphertext[aes.BlockSize:], plaintext)
	...
	f := os.Create("data.enc")
	f.Write(ciphertext)
	f.Close()
	始终使用随机 IV 和密钥。如果需要存储加密数据（尤其是随后可能会修改的数据），请使用与流密码功能不一样的分组密码。

CWE	CWE ID 327
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:User-Controlled Key Size**
默认严重性	5
摘要	X(文件) 中的 XX（函数） 函数包括的用户输入在第 N 行的加密算法所使用的密钥大小参数的范围内。采用密钥大小的加密函数可以接收受污染的密钥大小值。
解释	通过允许用户控制的值确定密钥大小，攻击者可以指定一个空密钥，从而可以相对容易地解密任何使用空密钥进行加密的数据。即使要求使用非零值，攻击者也仍然可以指定尽可能低的值，降低加密的安全性。 弱加密：用户控制密钥大小问题将在以下情况下出现： 1.数据通过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） 中第 N 行的 X（函数） 之中。 2.用户控制的数据包含在密钥大小参数中，或完全用作加密函数中的密钥大小参数。 在这种情况下，在 Y(文件) 中第 M 行的 Y(函数) 使用该数据。 如同许多软件安全漏洞一样，弱加密：用户控制密钥大小是到达终点的一个途径，其本身并不是终点。从本质上看，这些漏洞是显而易见的：攻击者将恶意数据传送到应用程序，这些数据随后被用作执行加密的密钥大小值的全部或一部分。 用户控制密钥大小的问题在于，它可以实现各种不同的攻击： 1.攻击者可以利用此漏洞，为涉及他们可以访问的任何数据的加密操作指定零密钥大小。由此，可以轻易地使用多个不同的算法以及空密钥，尝试对他们自己的数据进行解密，以泄露有关应用程序中使用的加密实现的信息。通过允许攻击者在破解期间仅专注于特定算法，这使攻击者可以更容易地解密其他用户的加密数据。 2.攻击者可以操纵其他用户的加密密钥大小，或诱骗其他用户使用零（或其他较小数字）加密密钥大小，从而使攻击者可能可以读取其他用户的加密数据（攻击者知晓所使用的加密算法后）。 示例 1：以下代码可生成使用用户控制的衍生密钥长度的 RSA 密钥： ... rsa.GenerateKey(random, user_input) ... Example 1 中的代码将成功运行，但任何有权使用此功能的人将能够操纵加密算法的密钥大小参数，因为变量 user_input 可由用户控制。一旦软件发布，撤消与用户控制的密钥大小相关的问题就会非常困难。很难知道恶意用户是否控制了给定加密操作的密钥大小。
建议	加密算法的密钥大小参数绝不能由用户控制，即使部分控制也不行。通常情况下，应该根据使用中的加密算法，将密钥大小参数手动设置为相应的密钥大小。一些 API 甚至提供一套与各种加密算法相对应的密钥大小常量。

返回目录

	示例 2：以下代码可生成使用由衍生密钥的使用所决定的值的 RSA 密钥： ... rsa.GenerateKey(random, 2048) ...
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**XML Injection**
默认严重性	4
摘要	在 X(文件) 的第 N 行中，XX（函数） 方法将写入未经验证的 XML 输入。攻击者可以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.数据从一个不可信数据源进入程序。 在这种情况下，数据进入 X（文件） 中第 N 行的 X（函数） 之中。 2.数据写入到 XML 文档中。 在这种情况下，Y(函数) 会在 Y(文件) 的第 M 行中写入 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。在危害最轻的情况下，攻击者可能会插入无关的标签，并导致 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。有时 XML injection 可以导致 cross-site scripting 或 dynamic code evaluation。 示例 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> 当使用 SAX 解析器时，第二个 <price> 标签中的值将会覆盖第一个 <price> 标签中的值。这样，攻击者就可以只花 1 美元购买一双价值 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