GitSec.Cloud
48h to DevSecOps

漏洞修复建议速查手册

(PHP 篇)

开源安全研究院

2022/7/29

返回目录

简介

手册简介：

本手册是针对 PHP 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

Hardcoded Absolute Path
Header Injection
Header Manipulation
Header Manipulation:Cookies
Header Manipulation:SMTP
HTML5:Overly Permissive CORS Policy
HTTP Parameter Pollution
HTTP Response Splitting
HttpOnlyCookies
Improper Control of Dynamically Identified Variables
Improper Exception Handling
Improper Restriction of Stored XXE Ref
Improper Restriction of XXE Ref
Improper Transaction Handling
Inappropriate Encoding for Output Context
Information Exposure Through an Error Message
Information Leak Through Persistent Cookies
Insecure Randomness
Insecure Transport
Insufficiently Protected Credentials
JSON Injection
Key Management:Empty Encryption Key
Key Management:Hardcoded Encryption Key
Key Management:Hardcoded PBE Password
Key Management:Null Encryption Key
LDAP Injection
LDAP Manipulation
Log Forging
Missing HSTS Header
Object Injection
Often Misused:File Upload
Open Redirect
Parameter Tampering
Password Management
Password Management:Empty Password
Password Management:Hardcoded Password
Password Management:Null Password
Password Management:Password in Comment
Password Management:Weak Cryptography
Path Manipulation
Path Traversal
PHP Misconfiguration:allow_url_fopen Enabled
PHP Misconfiguration:allow_url_include Enabled
PHP Misconfiguration:cgi.force_redirect Disabled

返回目录

PHP Misconfiguration:file_uploads Enabled
PHP Misconfiguration:magic_quotes_gpc Enabled
PHP Misconfiguration:magic_quotes_runtime Enabled
PHP Misconfiguration:magic_quotes_sybase Enabled
PHP Misconfiguration:Missing safe_mode_exec_dir Entry
PHP Misconfiguration:Poor open_basedir Configuration
PHP Misconfiguration:register_globals Enabled
PHP Misconfiguration:safe_mode Disabled
PHP Misconfiguration:session_use_trans_sid Enabled
Poor Error Handling:Empty Catch Block
Poor Error Handling:Return Inside Finally
Possible Flow Control
Possible Variable Overwrite:Functional Scope
Possible Variable Overwrite:Global Scope
Privacy Violation
Privacy Violation:HTTP GET
Process Control
Race Condition:PHP Design Flaw
Reflected File Download
Reflected XSS All Clients
Reflection Injection
Reliance on Cookies in a Decision
Reliance on DNS Lookups in a Decision
Remote File Inclusion
Resource Injection
Second Order SQL Injection
Server-Side Request Forgery
Session Fixation
Setting Manipulation
SQL Injection
SQL Injection:Poor Validation
SSL Verification Bypass
Stored Code Injection
Stored Command Injection
Stored File Inclusion
Stored File Manipulation
Stored LDAP Injection
Stored Reflection Injection
Stored Remote File Inclusion
Stored XPath Injection
Stored XSS
System Information Leak
System Information Leak:External
System Information Leak:Internal

返回目录

System Information Leak:PHP Errors
System Information Leak:PHP Version
Trust Boundary Violation
Unsafe Reflection
Unsafe Use Of Target Blank
Use of Broken or Risky Cryptographic Algorithm
Use of Hard coded Cryptographic Key
Use Of Hardcoded Password
Weak Cryptographic Hash
Weak Cryptographic Hash:Empty PBE Salt
Weak Cryptographic Hash:Hardcoded PBE Salt
Weak Cryptographic Hash:Hardcoded Salt
Weak Cryptographic Hash:Insecure PBE Iteration Count
Weak Cryptographic Hash:Predictable Salt
Weak Cryptographic Hash:User-Controlled PBE Salt
Weak Encryption
Weak Encryption:Inadequate RSA Padding
Weak Encryption:Insufficient Key Size
Weak Encryption:User-Controlled Key Size
XML Entity Expansion Injection
XML External Entity Injection
XML Injection
XPath Injection
XQuery Injection
XSLT Injection
XSRF
XSS Evasion Attack

返回目录

漏洞名称	**Access Control:Anonymous LDAP Bind**
默认严重性	4
摘要	如果没有适当的 Access Control，方法 X(方法) 会在包含攻击者控制值的第 N 行上执行 LDAP 语句。这会使攻击者能够访问未经授权的目录条目。如果没有适当的 Access Control，就会执行一个包含用户控制的值的 LDAP 语句，从而允许攻击者访问未经授权的记录。
解释	在匿名绑定下有效地执行 LDAP 查询（没有身份验证）可导致攻击者滥用配置不当的 LDAP 环境。 示例 1：如果用户输入空密码，以下代码将创建一个匿名绑定。 ... $ldapbind = ldap_bind ($ldap, $dn, $password = ""); ... 通过此连接执行的所有后续 LDAP 查询都是在没有身份验证和 Access Control 的情况下执行的。攻击者可能会以意外的方式操纵其中一个查询，从而获取对受目录 Access Control 机制保护的记录的访问权限。
建议	不要依靠表示层来限制用户提交的值，请通过作为特定用户绑定到目录来确保应用程序执行显式验证并强制执行 Access Control 限制。切勿允许用户在没有取得相应权限的情况下检索或修改目录中的记录。每个访问目录的查询都应该遵守此策略。这意味着在匿名绑定下只能执行非常有限的查询，这有效地绕过了 LDAP 系统中内置的 Access Control 机制。 示例 2：以下代码实现的功能与 Example 1 相同，但强制使用密码来实现 Access Control，以便确认仅经过身份验证的用户才能执行后续查询。 ... $ldapbind = ldap_bind ($ldap, $dn, $password); ...
CWE	CWE ID 285
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Access Control:Database**
默认严重性	2
摘要	如果没有适当的 access control，X(文件) 中的 X(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 XX(文件)的第 N 行的 XX(函数) 中。 2. 这个数据用来指定 SQL 查询中主键的值。 在这种情况下，在 YY（文件）中第 M 行的 YY（函数） 使用该数据。 例 1：以下代码用到一个参数化指令，这个指令转义了元字符，以防止 SQL injection 漏洞，并构建和执行一个 SQL 查询。该 SQL 查询指令可以搜索与指定标识符 [1] 相匹配的清单。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 = $_POST['id']; $query = "SELECT * FROM invoices WHERE id = ?"; $stmt = $mysqli->prepare($query); $stmt->bind_param('ss',$id); $stmt->execute(); ... 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。 许多现代 Web 框架都会提供对用户输入执行验证的机制（包括 Struts 和 Struts 2）。为了突出显示未经验证的输入源，Fortify 安全编码规则包会对 Fortify Static Code Analyzer（Fortify 静态代码分析器）报告的问题动态重新调整优先级，即在采用框架验证机制时降低这些问题被利用的几率并提供指向相应证据的指针。我们将这种功能称之为上下文敏感排序。为了进一步帮助 Fortify 用户执行审计过程，Fortify 软件安全研究团队开发了 Data Validation（数据验证）项目模板，该模板根据应用于输入源的验证机制按文件夹对问题进行了分组。
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

返回目录

	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。 ... $mysqli = new mysqli($host,$dbuser, $dbpass, $db); $userName = getAuthenticated($_SESSION['userName']); $id = $_POST['id']; $query = "SELECT * FROM invoices WHERE id = ? AND user = ?"; $stmt = $mysqli->prepare($query); $stmt->bind_param('ss',$id,$userName); $stmt->execute(); ...
CWE	CWE ID 566
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Blind SQL Injections**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行 YYY（方法） 方法中用于数据库查询。这可能导致 SQL 盲注攻击。
解释	攻击者可以直接访问系统的所有数据。攻击者使用现成的工具和文本编辑即可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，包括用户的输入。这样，用户输入未经过数据类型验证或净化，输入中可能包含数据库也做出同样解释的 SQL 命令。即使应用程序的错误响应中不包含数据库的实际内容，也可以使用工具对错误执行一系列布尔测试，从而逐步获得数据库内容。
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 此外，先转义所有用户输入再将其包含在查询中。转义应根据使用的具体数据库进行。 不要使用拼接字符串： 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。 不要让用户动态地提供所查询的表的名称，并尽量完全避免使用动态表名称。 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	**CakePHP Misconfiguration:Debug Information**
默认严重性	3
摘要	当 CakePHP 调试级别在 1 级或以上时，可导致敏感数据写入日志文件。
解释	CakePHP 可配置为公开调试信息，如错误、警告、SQL 指令和堆栈跟踪信息。在生产环境中，不应使用 Debug Information。 例 1: Configure::write('debug', 3); Configure::write() 方法的第二个参数表示调试级别。数值越大，记录的日志信息越冗长。
建议	在生产代码中，仅可使用值 0。 例 2: Configure::write('debug', 0);
CWE	CWE ID 215
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**CakePHP Misconfiguration:Excessive Session Timeout**
默认严重性	3
摘要	如果会话超时时间过长，攻击者就会有更多时间危害用户帐户。
解释	会话持续时间越长，攻击者危害用户帐户的机会就越大。当会话处于活动状态时，攻击者可能会强力攻击用户的密码、破解用户的无线加密密钥或者通过打开的浏览器强占会话。如果创建大量的会话，较长的会话超时时间还会阻止系统释放内存，并最终导致 denial of service。 例 1： 以下示例显示了配置有 low 会话安全级别的 CakePHP。 Configure::write('Security.level', 'low'); Security.level 和 Session.timeout 设置共同定义会话的有效长度。实际会话超时时间等于 Session.timeout 乘以以下倍数之一： 'high' = x 10 'medium' = x 100 'low' = x 300
建议	将会话超时间隔设置为 30 分钟或更少，既能使用户在一段时间内与应用程序互动，又提供了一个限制窗口攻击的合理范围。 例 2：以下示例将会话超时间隔设置为 20 分钟。 Configure::write('Session.timeout', '120'); Configure::write('Security.level', 'high');
CWE	CWE ID 613
OWASP2017	A2 Broken Authentication

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。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Command Injection**
默认严重性	4
摘要	X(文件) 的第 N 行会利用由不可信赖的数据构建的命令来调用 X（函数）。这种调用会导致程序以攻击者的名义执行恶意命令。执行不可信赖资源中的命令，或在不可信赖的环境中执行命令，都会导致程序以攻击者的名义执行恶意命令。
解释	Command Injection 漏洞主要表现为以下两种形式： – 攻击者能够篡改程序执行的命令：攻击者直接控制了所执行的命令。 – 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 XX(文件)的第 N 行的 XX(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 YY（文件）的第 M 行的 YY（函数） 执行。 3. 通过命令的执行，应用程序会授予攻击者一种原本不该拥有的特权或能力。 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 = $_ENV['APPHOME']; $cmd = $home . $INITCMD; 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。 …

返回目录

	$btype = $_GET['backuptype']; $cmd = "cmd.exe /K \"c:\\util\\rmanDB.bat " . $btype . "&&c:\\util\\cleanup.bat\""; 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 = shell_exec("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 Runtime.exec() 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

返回目录

	建一个绝对路径。应对照一系列定义有效值的常量，仔细地检查从配置文件或者环境中读取的命令值和路径。 有时还可以执行其他检验，以检查这些来源是否已被恶意篡改。例如，如果一个配置文件为可写，程序可能会拒绝运行。如果能够预先得知有关要执行的二进制代码的信息，程序就会进行检测，以检验这个二进制代码的合法性。如果一个二进制代码始终属于某个特定的用户，或者被指定了一组特定的访问权限，这些属性就会在执行二进制代码前通过程序进行检验。 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	**Cookie Security:Cookie not Sent Over SSL**
默认严重性	3
摘要	程序在 X（文件） 中第 N 行创建了 cookie，但未将 Secure 标记设置为 true。程序创建了 cookie，但未将 Secure 标记设置为 true。
解释	现今的 Web 浏览器支持每个 cookie 的 Secure 标记。如果设置了该标记，那么浏览器只会通过 HTTPS 发送 cookie。通过未加密的通道发送 cookie 将使其受到网络截取攻击，因此安全标记有助于保护 cookie 值的保密性。如果 cookie 包含私人数据或带有会话标识符，那么该标记尤其重要。 在这种情况下，程序会在 X（文件） 中第 N 行创建 cookie，但不会将 Secure 参数传递给 setcookie()，或使用值 false 进行传递。 示例 1：以下代码会在未设置 Secure 标记的情况下将 cookie 添加到响应中。 ... setcookie("emailCookie", $email, 0, "/", "www.example.com"); ... 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 作为第六个参数传递给 setcookie()，便可完成此配置。 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。 setcookie("emailCookie", $email, 0, "/", "www.example.com", 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
摘要	程序在 X(文件) 中第 N 行上创建了 cookie，但未能将 HttpOnly 标记设置为 true。程序创建了 cookie，但未能将 HttpOnly 标记设置为 true。

解释	所有主要浏览器均支持 HttpOnly Cookie 属性，可阻止客户端脚本访问 Cookie。Cross-Site Scripting 攻击通常会访问 Cookie，以试图窃取会话标识符或身份验证令牌。如果未启用 HttpOnly，攻击者就能更容易地访问用户 Cookie。
	
	在这种情况下，将在 X(文件)的第 N 行中设置 Cookie，但不会设置 HttpOnly 参数，或将其设置为 false。
	
	示例 1：以下代码会在未设置 HttpOnly 属性的情况下创建一个 Cookie。
	setcookie("emailCookie", $email, 0, "/", "www.example.com", TRUE);
	//Missing 7th parameter to set HttpOnly

建议	在创建 Cookie 时启用 HttpOnly 属性。做法是将 setcookie() 调用中的 HttpOnly 参数设置为 true。 示例 2：以下代码创建的 Cookie 与 Example 1 中的代码创建的相同，但这次会将 HttpOnly 参数设置为 true。 setcookie("emailCookie", $email, 0, "/", "www.example.com", TRUE, TRUE); 已开发出了多种绕过将 HttpOnly 设置为 true 的机制，因此它并非完全有效。
CWE	CWE ID 1004
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:HTTPOnly not Set on Session Cookie**
默认严重性	2
摘要	程序在 X(文件) 中第 N 行上创建了会话 cookie，但未能将 HttpOnly 标记设置为 true。程序创建了 cookie，但未能将 HttpOnly 标记设置为 true。
解释	所有主要浏览器均支持 HttpOnly Cookie 属性，可阻止客户端脚本访问 Cookie。Cross-Site Scripting 攻击通常会访问 Cookie，以试图窃取会话标识符或身份验证令牌。如果未启用 HttpOnly，攻击者就能更容易地访问用户 Cookie。 在这种情况下，程序会在 X(文件) 中第 N 行创建会话 cookie，但将 HttpOnly 参数设置为 false。 示例 1：以下代码会在未将 HttpOnly 参数设置为 true 的情况下创建会话 Cookie。 session_set_cookie_params(0, "/", "www.example.com", true, false);
建议	在创建会话 Cookie 时启用 HttpOnly 属性。做法是将 session_set_cookie_params() 调用中的 HttpOnly 参数设置为 true。 示例 2：以下代码创建的 Cookie 与 Example 1 中的代码创建的相同，但这次会将 HttpOnly 参数设置为 true。 session_set_cookie_params(0, "/", "www.example.com", true, true); 已开发出了多种绕过将 HttpOnly 设置为 true 的机制，因此它并非完全有效。
CWE	CWE ID 1004
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Missing SameSite Attribute**
默认严重性	3
摘要	程序无法在会话 Cookie 上设置 SameSite 属性。
解释	浏览器会自动将 Cookie 附加到对设置 Cookie 的站点发出的每个 HTTP 请求中。Cookie 可能存储敏感数据，例如会话 ID 和授权令牌，或者在会话期间对同一站点的不同请求之间共享的站点数据。攻击者可以通过从客户机上加载的第三方网站页面生成对已验证站点的请求来执行模拟攻击，因为浏览器会自动将 Cookie 附加到请求中。SameSite 属性限制了 Cookie 的作用域，因此只有当请求是从第一方或同一站点上下文生成时，才会将 Cookie 附加到请求。这样有助于保护 Cookie 免受 Cross-Site Request Forgery (CSRF) 攻击。SameSite 属性可以有以下三个值： - Strict：当设置为 Strict 时，Cookie 仅在顶级导航时与请求一起发送。 - Lax：当设置为 Lax 时，Cookie 随顶级导航与从第三方站点发送到主机的 GET 请求一起从同一主机发送。例如，假设第三方站点拥有主机站点的 iframe 或 href 标签。如果用户点击链接，请求将包含 Cookie。 - None：Cookie 随对 Cookie 设置的路径和域范围内的站点的所有请求一起发送。因使用 POST 方法提交表单而生成的请求也可以随请求一起发送 Cookie。 在这种情况下，在 X(文件) 的第 N 行上，session.cookie_samesite 属性设置为 "None" 或 ""。 示例 1：以下代码将对会话 Cookie 禁用 SameSite 属性。 ini_set("session.cookie_samesite", "None");
建议	在创建会话 Cookie 时启用 SameSite 属性。为此，请使用 ini_set 在 PHP 配置选项中将 session.cookie_samesite 属性设置为 Strict 或 Lax。 示例 2：以下代码将对会话 Cookie 启用 SameSite 属性。 ini_set("session.cookie_samesite", "Strict"); 此外，Fortify 建议开发人员继续向站点添加传统的 Cross-Site Request Forgery (CSRF) 缓解以及 SameSite 属性。许多用户可能会使用旧的浏览器版本访问该站点。旧的浏览器版本不支持 SameSite 属性。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**Cookie Security:Overly Broad Domain**
默认严重性	2
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 cookie。 例如: setcookie("mySessionId", getSessionID(), 0, "/", ".example.com", true, true); 假设您在 http://insecure.example.com/ 上有另一个不太安全的应用程序，它包含 cross-site scripting 漏洞。任何浏览到 http://insecure.example.com 的 http://secure.example.com 认证用户面临着暴露来自 http://secure.example.com 的会话 cookie 的风险。除了读取 cookie 外，攻击者还可能使用 insecure.example.com 进行 cookie poisoning 攻击，创建自己范围过大的 cookie，并覆盖来自 secure.example.com 的 cookie。
建议	确保将 cookie 域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 域设置为 "secure.example.com"。 setcookie("mySessionId", getSessionID(), 0, "/", "secure.example.com", true, true);
CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Path**
默认严重性	2
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。 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: setcookie("mySessionId", getSessionID(), 0, "/", "communitypages.example.com", true, true); 假设攻击者在 http://communitypages.example.com/EvilSite 上创建了另一个应用程序，并在论坛上发布了该站点的链接。当论坛用户点击该链接时，浏览器会将 /MyForum 设置的 Cookie 发送到在 /EvilSite 上运行的应用程序。通过这种方式窃取会话 ID 后，攻击者就能够危及浏览到 /EvilSite 的任何论坛用户的帐户安全。 除了读取 cookie 外，攻击者还可能使用 /EvilSite 进行 cookie poisoning 攻击，创建自己范围过大的 cookie，并覆盖来自 /MyForum 的 cookie。
建议	确保将 cookie 路径设置为具有尽可能高的限制性。 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"。 setcookie("mySessionId", getSessionID(), 0, "/MyForum", "communitypages.example.com", true, true);
CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Session Cookie Domain**
默认严重性	2
摘要	通过共享相同的基本域的其他应用程序可攻击域范围过大的会话 cookie。
解释	开发人员通常将会话 Cookie 设置为类似“.example.com”的基本域。这会使会话 Cookie 暴露在基本域和任何子域中的所有 Web 应用程序下。在应用程序之间共享会话 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 cookie。 例如: session_set_cookie_params(0, "/", ".example.com", true, true); 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。
建议	将 cookie 域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”部分中的示例将会话 cookie 域设置为 "secure.example.com"。 session_set_cookie_params(0, "/", "secure.example.com", true, true);
CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Broad Session Cookie Path**
默认严重性	2
摘要	可能通过共享相同域的应用程序危及路径范围过大的 cookie 的安全。
解释	开发人员通常将会话 Cookie 设置为根上下文路径（“/”）。这会使 Cookie 暴露在域名相同的所有 Web 应用程序下。泄露会话 Cookie 可能导致危及帐户安全，因为攻击者可能利用域中任何应用程序的漏洞来窃取会话 Cookie。 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 cookie。 例如: session_set_cookie_params(0, "/", "communitypages.example.com", true, true); 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，他的浏览器会将 /MyForum 设置的会话 Cookie 发送到在 /EvilSite 上运行的应用程序。通过这种方式窃取会话 Cookie 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。
建议	将会话 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”部分中的示例将会话 cookie 路径设置为 "/MyForum"。 session_set_cookie_params(0, "/MyForum", "communitypages.example.com", true, true);
CWE	None
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Overly Permissive SameSite Attribute**
默认严重性	3
摘要	会话 Cookie 上的 SameSite 属性未设置为 Strict。
解释	SameSite 属性保护 Cookie 免受 Cross-Site Request Forgery (CSRF) 等攻击。会话 Cookie 代表用户访问站点，以便用户可以执行授权操作。但是，浏览器会自动将 Cookie 与请求一起发送，因此用户和网站会隐式信任浏览器进行授权。攻击者可以滥用此信任，通过在攻击者控制的第三方网站页面中的 link 和 iframe 等标签的 href 和 src 属性中嵌入链接，代表用户向站点发出请求。如果攻击者能够将一位毫无戒备的用户引诱到他们控制的第三方站点，则攻击者可以发出自动包含用于授权该用户的会话 Cookie 的请求，从而可以使攻击者获取有效授权，就像他们是用户一样。 将会话 Cookie 设置为 SameSite 属性的 Strict，这将限制浏览器仅将 Cookie 附加到顶级导航时的请求或来自同一站点的请求。来自第三方站点的请求（通过 iframe、img 和 form 等各种标签中的链接）不含这些 Cookie，因此能阻止站点采取用户可能未授权的操作。 在这种情况下，在 X(文件) 的第 N 行上，session.cookie_samesite 属性设置为 Lax。 示例 1：以下代码将对会话 Cookie 启用 SameSite 属性中的 Lax 模式。 ini_set("session.cookie_samesite", "Lax");
建议	使用 ini_set 在 PHP 配置选项中将 SameSite 属性设置为 Strict。 示例 2：以下代码将对会话 Cookie 启用 SameSite 属性中的 Strict 模式。 ini_set("session.cookie_samesite", "Strict"); 此外，Fortify 建议开发人员继续向站点添加传统的 Cross-Site Request Forgery (CSRF) 缓解以及 SameSite 属性。许多用户可能会使用旧的浏览器版本访问该站点。旧的浏览器版本不支持 SameSite 属性。
CWE	CWE ID 352
OWASP2017	None

返回目录

漏洞名称	**Cookie Security:Persistent Cookie**
默认严重性	2
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。 示例：以下代码将 cookie 设置为在 10 年后过期。 setcookie("emailCookie", $email, time()+60*60*24*365*10);
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:Persistent Session Cookie**
默认严重性	2
摘要	永久性会话 cookie 可导致危及帐户安全。
解释	永久性会话 cookie 在用户关闭了浏览器后仍然保持有效，并通常用作“记住我的信息”功能的一部分。因此，即使在用户关闭了浏览器后，永久性会话 cookie 也会使他们保持应用程序认证状态 — 假设他们没有明确注销。这就意味着如果第二个用户打开浏览器，他将以上个用户的身份自动登录。除非在受控环境中部署了应用程序，而在该环境中，不允许用户从共享计算机登录，否则即使用户关闭了浏览器，攻击者也可能危及用户帐户的安全。 在这种情况下，在 X(文件)的第 N 行上，session.cookie_lifetime 属性设置为非零数字。 示例：以下代码将会话 cookie 设置为在 10 年后过期。 session_set_cookie_params(time()+60*60*24*365*10, "/", "www.example.com", false, true);
建议	不要使用永久性会话 Cookie。做法是在 PHP 配置文件中将 session.cookie_lifetime 设置为 0。
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
摘要	程序在 X（文件） 中第 N 行创建了会话 cookie，但未将 Secure 标记设置为 true。程序创建了会话 cookie，但未将 Secure 标记设置为 true。
解释	现今的 Web 浏览器支持每个 cookie 的 Secure 标记。如果设置了该标记，那么浏览器只会通过 HTTPS 发送 cookie。通过未加密的通道发送 cookie 将使其受到网络截取攻击，因此安全标记有助于保护 cookie 值的保密性。如果 cookie 包含私人数据或带有会话标识符，那么该标记尤其重要。 在这种情况下，程序会在 X（文件） 中第 N 行创建会话 cookie，但不会将 Secure 标记传递给 setcookie()，或使用值 false 进行传递。 示例 1：以下代码会在未设置 Secure 标记的情况下将 cookie 添加到响应中。 ... setcookie("emailCookie", $email, 0, "/", "www.example.com"); ... 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 作为第六个参数传递给 setcookie()，便可完成此配置。 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。 ... setcookie("emailCookie", $email, 0, "/", "www.example.com", TRUE); ...
CWE	CWE ID 614
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Cookie Security:Session Cookies Disabled**
默认严重性	3
摘要	程序不使用 cookie 传输会话标识符，这可能导致易受 session fixation 和 session hijacking 攻击。
解释	大多数 Web 应用程序使用会话标识符来唯一标识用户，该标识符通常存储在 cookie 中，并在服务器和 Web 浏览器之间进行透明传输。 在这种情况下，在 X(文件) 中第 N 行上，session.use_cookies 或 session.use_only_cookies 属性会设置为 0 或 off，这样可以防止应用程序在 cookie 中存储会话标识符。 没有在 cookie 中存储会话标识符的应用程序有时将这些标识符作为 HTTP 请求参数或 URL 的一部分进行传输。接受 URL 中指定的会话标识符使得攻击者很容易进行 session fixation 攻击。 将会话标识符放在 URL 中还会增加针对应用程序进行 session hijacking 攻击的成功几率。当攻击者控制了受害者的活动会话或会话标识符时，就会发生 session hijacking。对于 Web 服务器、应用程序服务器和 Web 代理而言，通常的做法是存储请求的 URL。如果会话标识符包括在 URL 中，那么也会记录它们。增加能够查看和存储会话标识符的位置就会增加被攻击者攻击的机会。
建议	将会话标识符存储在 cookie 中。在 PHP 配置文件中将 session.use_cookies 或 session.use_only_cookies 属性设置为 1 或 on，便可完成此配置。
CWE	CWE ID 384
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-Site Scripting:Persistent**
默认严重性	4
摘要	文件 X(文件) 的第 N 行向一个 Web 浏览器发送了未验证的数据，从而导致该浏览器执行恶意代码。向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码。
解释	Cross-Site Scripting (XSS) 漏洞在以下情况下发生： 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 XX(文件)的第 N 行的 XX(函数) 中。 2.未经验证但包含在动态内容中的数据将传送给 Web 用户。 在这种情况下，数据通过 YY（文件）的第 M 行中的 YY（函数） 传送。 传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。 例 1: 下面的 PHP 代码片段可根据一个给定的雇员 ID 查询数据库，并显式出相应的雇员姓名。 <?php... $con = mysql_connect($server,$user,$password); ... $result = mysql_query("select * from emp where id="+eid); $row = mysql_fetch_array($result) echo 'Employee name: ', mysql_result($row,0,'name'); ... ?> 如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的盗取行为无能为力。这段代码暴露出的危险较小，因为 name 的值是从数据库中读取的，而且显然这些内容是由应用程序管理的。然而，如果 name 的值是由用户提供的数据产生，数据库就会成为恶意内容沟通的通道。如果不对数据库中存储的所有数据进行恰当的输入验证，那么攻击者就可以在用户的 Web 浏览器中执行恶意命令。这种类型的 Persistent XSS（也称为 Stored XSS）盗取极其阴险狡猾，因为数据存储导致的间接性使得辨别威胁的难度增大，而且还提高了一个攻击影响多个用户的可能性。XSS 盗取会从访问提供留言簿 (guestbook) 的网站开始。攻击者会在这些留言簿的条

返回目录

	目中嵌入 JavaScript，接下来所有访问该留言簿的用户都会执行这些恶意代码。 例 2：下面的 PHP 代码片段可从一个 HTTP 请求中读取雇员 ID eid，并将其显示给用户。 <?php $eid = $_GET['eid']; ... ?> ... <?php echo "Employee ID: $eid"; ?> 如 Example 1 中所示，如果 eid 只包含标准的字母数字文本，此代码将会正确运行。如果 eid 中的某个值包含元字符或源代码，则 Web 浏览器就会在显示 HTTP 响应时执行该代码。 起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易受攻击的网络应用程序，将恶意内容带到了自己的电脑中。这种对易受攻击的 Web 应用程序进行盗取的机制通常被称为反射式 XSS。 正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种： - 如 Example 1 中所示，应用程序将危险数据存储在数据库或其他可信赖的数据存储中。这些危险数据随后会被读回到应用程序中，并包含在动态内容中。在以下情况下会发生 Persistent XSS 漏洞利用：攻击者将危险内容注入到数据存储中，而这些危险内容随后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于显示给许多用户或显示给特定相关用户的区域。这些相关用户通常在应用程序中具备较高的特权，或者可以与敏感数据交互，这些数据对攻击者来说具有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人敏感数据的访问权。 - 如 Example 2 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。

返回目录

	— 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，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。 下面的这些原则适用于属性值:

返回目录

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

返回目录

漏洞名称	**Cross-Site Scripting:Poor Validation**
默认严重性	2
摘要	在 X(文件) 中，程序会使用 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	使用特定的编码函数（例如 htmlspecialchars() 或 htmlentities()）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 <、>、&、" 以及 XML 编码的字符 <、>、&、" 和 ' 之外（仅当已设置 ENT_QUOTES 时），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，不可信赖的数据源大多数情况下为 Web 请求；而对于 Persistent（也称为 Stored）XSS，该数据源通常为数据库查询的结果。 在这种情况下，数据进入 XX(文件)的第 N 行的 XX(函数) 中。 2.未经验证但包含在动态内容中的数据将传送给 Web 用户。 在这种情况下，在 YY（文件）第 M 行的 YY（函数） 中，会利用 HTML 编码对数据进行简单的验证，然后写入到浏览器中。 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：下面的代码片段会从 HTTP 请求中读取 text 参数，使用 HTML 加以编码，并将该参数显示在脚本标签之间的警报框中。 <?php $var=$_GET['text']; ... $var2=htmlspecialchars($var); echo "<script>alert('$var2')</script>"; ?>

返回目录

	如果 text 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 text 具有单引号、圆括号和分号，则会结束 alert 文本框，之后将执行代码。 起初，这个例子似乎是不会轻易遭受攻击的。毕竟，有谁会输入导致恶意代码在自己电脑上运行的 URL 呢？真正的危险在于攻击者会创建恶意的 URL，然后采用电子邮件或社交工程的欺骗手段诱使受害者访问此 URL 的链接。当受害者单击这个链接时，他们不知不觉地通过易受攻击的网络应用程序，将恶意内容带到了自己的电脑中。这种对易受攻击的 Web 应用程序进行盗取的机制通常被称为反射式 XSS。 正如例子中所显示的，XSS 漏洞是由于 HTTP 响应中包含了未经验证的数据代码而引起的。受害者遭受 XSS 攻击的途径有三种： - 如 Example 1 中所示，系统从 HTTP 请求中直接读取数据，并在 HTTP 响应中返回数据。当攻击者诱使用户为易受攻击的 Web 应用程序提供危险内容，而这些危险内容随后会反馈给用户并在 Web 浏览器中执行时，就会发生 Reflected XSS 漏洞利用。发送恶意内容最常用的方法是，将恶意内容作为一个参数包含在公开发布或通过电子邮件直接发送给受害者的 URL 中。以这种手段构造的 URL 已成为多种网络钓鱼阴谋的核心，攻击者会借此诱骗受害者访问指向易受攻击站点的 URL。当该站点将攻击者的内容反馈给受害者后，便会执行这些内容，接下来会将用户计算机中的各种私密信息（比如可能包含会话信息的 Cookie）传输给攻击者，或者执行其他恶意活动。 — 应用程序将危险数据存储在数据库或其他可信赖的数据存储器中。这些危险数据随后会被回写到应用程序中，并包含在动态内容中。Persistent XSS 盗取发生在如下情况：攻击者将危险内容注入到数据存储器中，且该存储器之后会被读取并包含在动态内容中。从攻击者的角度看，注入恶意内容的最佳位置莫过于一个面向许多用户，尤其是相关用户显示的区域。相关用户通常在应用程序中具备较高的特权，或相互之间交换敏感数据，这些数据对攻击者来说有利用价值。如果某一个用户执行了恶意内容，攻击者就有可能以该用户的名义执行某些需要特权的操作，或者获得该用户个人所有的敏感数据的访问权限。 — 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，并储存在动态内容中。
建议	针对 XSS 的解决方法是，确保在适当位置进行验证，并检验其属性是否正确。 由于 XSS 漏洞在应用程序的输出中包含恶意数据时出现，因此，合乎逻辑的方法是在数据流出应用程序的前一刻对其进行验证。然而，由于 Web 应用程序常常会包含复杂而难以理解的代码，用以生成动态内容，因此，这一方法容易产生遗漏错误（遗漏验证）。降低这一风险的有效途径是对 XSS 也执行输入验证。 由于 Web 应用程序必须验证输入信息以避免其他漏洞（如 SQL Injection），因此，一种相对简单的解决方法是，加强一个应用程序现有的输入验证机制，将 XSS 检测包括其中。尽管有一定的价值，但

返回目录

	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。 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。

返回目录

	- "&" 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 82, CWE ID 83, CWE ID 87, CWE ID 692
OWASP2017	A7 Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Cross-Site Scripting:Reflected**
默认严重性	4
摘要	文件 X(文件) 的第 N 行向一个 Web 浏览器发送了未验证的数据，从而导致该浏览器执行恶意代码。向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码。
解释	Cross-Site Scripting (XSS) 漏洞在以下情况下发生： 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 XX(文件)的第 N 行的 XX(函数) 中。 2.未经验证但包含在动态内容中的数据将传送给 Web 用户。 在这种情况下，数据通过 YY（文件）的第 M 行中的 YY（函数） 传送。 传送到 Web 浏览器的恶意内容通常采用 JavaScript 代码片段的形式，但也可能会包含一些 HTML、Flash 或者其他任意一种可以被浏览器执行的代码。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。 例 1：下面的 PHP 代码片段可从一个 HTTP 请求中读取雇员 ID eid，并将其显示给用户。 <?php $eid = $_GET['eid']; ... ?> ... <?php echo "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：下面的 PHP 代码片段可根据一个给定的雇员 ID 查询数据库，并显式出相应的雇员姓名。

返回目录

	<?php... $con = mysql_connect($server,$user,$password); ... $result = mysql_query("select * from emp where id="+eid); $row = mysql_fetch_array($result) echo 'Employee name: ', mysql_result($row,0,'name'); ... ?> 如同 Example 1，如果对 name 的值处理得当，该代码就能正常地执行各种功能；如若处理不当，就会对代码的漏洞利用行为无能为力。同样，这段代码看起来似乎危险较小，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。

返回目录

	— 应用程序之外的数据源将危险数据储存在一个数据库或其他数据存储器中，随后这些危险数据被当作可信赖的数据回写到应用程序中，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。 下面的这些原则适用于属性值:

返回目录

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

返回目录

漏洞名称	**Dangerous File Inclusion**
默认严重性	4
摘要	文件 X(文件) 将一个未验证的文件名传递给了第 N 行的一个动态 include 指令。如果允许未验证的用户输入控制动态包含在 PHP 中的文件，会导致恶意代码的执行。 如果允许未验证的用户输入控制动态包含在 PHP 中的文件，会导致恶意代码的执行。
解释	许多现代网络编写语言都能够在一个封装的文件内包含附加的源文件，从而使代码可以重用和模块化。这种能力经常用于赋予应用程序标准外观（应用模板），因而，人们可以共享各种功能而不需要借助编译的代码，或将代码分解成较小的更好管理的文件。各个包含文件都会作为主文件的一部分进行解析，并采用相同的方式来执行。当未验证的用户输入控制了所包含文件的路径时，就会发生 File inclusion 漏洞。 File inclusion 漏洞是各种 PHP 应用程序中最多见且最为严重的漏洞。在低于 PHP 4.2.0 的版本中，PHP 的安装包附带了默认启用的 register_globals，从而使攻击者很容易重写内部服务器的各种变量。尽管禁用 register_globals 可以从一定程度上减少程序发生 file inclusion 漏洞的风险，但是这些问题仍然存在于各种现代的 PHP 应用程序中。 例 1：以下代码包含了一个文件，该文件位于模板中定义 $server_root 的应用程序下。 ... <?php include($server_root . '/myapp_header.php'); ?$gt; ... 如果将 register_globals 设置为 on，攻击者可以通过提供 $server_root 请求参数覆盖 $server_root 值，从而部分控制动态 include 指令。 例 2：以下代码采用了用户指定的模板名称，并将该名称包含在要呈现的 PHP 页面中。 ... <?php include($_GET['headername']); ?$gt; ... 在 Example 2 中，攻击者可通过为 headername 提供恶意值来完全控制动态 include 语句，从而会使程序包含来自外部站点的文件。 如果攻击者给动态 include 指令指定有效文件，该文件的内容将被传送到 PHP 解释器。对于一个纯文本文件来说，如 /etc/shadow，该文件可能会作为 HTML 输出的一部分来呈现。更为糟糕的是，如果攻击者可以指定一条路径来指向被自己控制的远程站点，那么动态 include 指令就会执行由攻击者提供的任意恶意代码。
建议	通过将以下行包含在 php.ini 文件中来禁用 register_globals 选项：register_globals = 'off'

返回目录

	不要允许未验证的用户输入控制动态包含指令中使用的路径。而应当采取一种间接手段：创建一份合法包含文件列表，仅允许用户从该列表中进行选择。利用这种方法，用户就不能直接从文件系统中指定一个文件。 例 2 可以进一步改进，将用户输入映射至用来选择所需模板的按键，如下所示： <?php $templates = array('main.php' => 1, 'blue.php' => 2, 'red.php' => 3); ?$gt; ... <?php include($templates[$_GET['headername']]); ?$gt; ...
CWE	CWE ID 94, CWE ID 98, CWE ID 494
OWASP2017	A1 Injection

返回目录

漏洞名称	**Dangerous File Injection**
默认严重性	4
摘要	X(文件) 中的第 N 行会使用不可信赖的数据调用 X（函数），这样可致使攻击者在服务器中创建包含任意内容的文件。攻击者将能够在系统中创建包含任意内容的文件。
解释	攻击者将能够在服务器的文件系统中创建包含任意内容的文件。然后，攻击者会使用创建的文件执行其他攻击，具体取决于控制注入文件的内容的能力。 如果攻击者能够控制文件的内容，且文件由 Web 服务器提供服务，则他将能够注入恶意网壳，使其可以在服务器上远程执行任意命令。 如果攻击者能够创建包含 file system 中其他文件的内容的文件，他将能够读取可以使用易受攻击应用程序的权限访问的 file system 中的任意文件。
建议	不要允许用户直接控制程序执行的命令或程序创建的文件的特征，例如文件扩展名和存储目录。在用户的输入会影响命令执行或文件创建的情况下，应将用户输入限制为从预定的安全选项集合中进行选择。 如果输入中出现了恶意的内容，那么程序应当默认传递该集合中的一个安全选项，或者拒绝对不可信赖的数据执行任何进一步操作。
CWE	CWE ID 94, CWE ID 98, CWE ID 494
OWASP2017	A1 Injection

返回目录

漏洞名称	**Dangerous Function**
默认严重性	5
摘要	无法安全地使用函数 X（函数）。不应该使用此函数。永不应该使用那些无法安全使用的函数。
解释	某些函数不论如何使用都有危险性。这一类函数通常是在没有考虑安全问题的情况下就执行了。 在这种情况下，您正在使用的危险函数为 X(文件)的第 N 行中的 X（函数）。
建议	永不应该使用那些无法安全使用的函数。如果这些函数中的任何一个出现在新的或是继承代码中，则必须删除该函数并用相应的安全函数进行取代。
CWE	CWE ID 242
OWASP2017	None

返回目录

漏洞名称	**Dangerous Function:Unsafe Regular Expression**
默认严重性	3
摘要	无法安全地使用函数 X（函数）。不应该使用此函数。永不应该使用那些无法安全使用的函数。
解释	某些函数不论如何使用都有危险性。这一类函数通常是在没有考虑安全问题的情况下就执行了。 在这种情况下，您正在使用的危险函数为 X(文件)的第 N 行中的 X（函数）。
建议	永不应该使用那些无法安全使用的函数。如果这些函数中的任何一个出现在新的或是继承代码中，则必须删除该函数并用相应的安全函数进行取代。
CWE	CWE ID 676
OWASP2017	None

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	**Denial of Service**
默认严重性	3
摘要	通过调用 X(文件) 中第 N 行的 X（函数），攻击者可以造成程序崩溃或让合法用户无法进行使用。攻击者可以造成程序崩溃或使合法用户无法进行使用。 例 1：如果攻击者设置了一个较大的值，那么使用不可信赖的数据设置服务超时可能会使服务变得没有反应。 ... $timeout = $_GET['time']; imap_timeout($time); ...
解释	攻击者可能通过对应用程序发送大量请求，而使它拒绝对合法用户的服务，但是这种攻击形式经常会在网络层就被排除掉了。更加严重的是那些只需要使用少量请求就可以使得攻击者让应用程序过载的 bug。这种 bug 允许攻击者去指定请求使用系统资源的数量，或者是持续使用这些系统资源的时间。
建议	校验用户输入以确保它不会引起不适当的资源利用。
CWE	CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Denial of Service:Regular Expression**
默认严重性	4
摘要	不受信数据被传递至应用程序并作为正则表达式使用。这会导致线程过度使用 CPU 资源。
解释	实施正则表达式评估程序及相关方法时存在漏洞，在评估包含自重复分组表达式的正则表达式时，该漏洞会导致线程挂起。此外，还可以利用任何包含相互重叠的替代子表达式的正则表达式。此缺陷可被攻击者用于执行拒绝服务 (DoS) 攻击。 示例: (e+)+ ([a-zA-Z]+)* (e|ee)+ 已知的正则表达式实现方式均无法避免这种漏洞。所有平台和语言都容易受到这种攻击。
建议	请不要将不可信赖数据作为正则表达式使用。
CWE	CWE ID 185, CWE ID 730
OWASP2017	None

返回目录

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

返回目录

漏洞名称	**Deserialization of Untrusted Data**
默认严重性	4
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。 在 PHP 中，反序列化只会影响作用域中的对象，并且只会触发这些对象的 __construct() 和 __destruct() 方法；这是因为 PHP 中的许多“危险”方法实际上是语言结构，不能通过序列化调用。这意味着反序列化仍然可以在逻辑上绕过或利用现有对象，但前提是它们使用部署后会对应用程序构成威胁的构造函数或析构函数。例如——如果析构函数使用（可以通过反序列化操作的）实例变量创建文件，攻击者就能使用此类析构函数编写有恶意代码的文件。 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。
CWE	CWE ID 502
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	None

返回目录

漏洞名称	**Dynamic Code Evaluation:Code Injection**
默认严重性	4
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 = $_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“exec('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	**Exposure of Resource to Wrong Sphere**
默认严重性	2
摘要	应用程序在 Y（文件） 文件第 M 行中暴露了一个公共字段 YY（元素）。
解释	如果一个类将内部变量公开为公共字段，但未限制访问，变量就可能被意外修改，使类的外部使用者为字段设置任意的、不允许的值。如果该类（或其他使用者）对该变量的值进行假定，就会导致意外行为。这甚至可能导致其他漏洞，具体取决于此值的使用方式。 应用程序的其中一个类将内部变量通过将其作为一个属性来公开为一个公共字段，但未限制访问。也可以公开公共字段但不允许外部修改其值。
建议	避免将内部变量和特定实现暴露为公共字段。 建议将数据作为属性公开，并根据需要在属性代码中实施数据验证和控制。 暴露公共公共字段时，使用 final 修饰符将值限制为只读。
CWE	CWE ID 493
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**File Disclosure**
默认严重性	5
摘要	通过 X（文件） 文件第 N 行中的 XXX（方法） 获取的输入被用于确定 Y（文件） 文件第 M 行中 YYY（方法） 方法要读取的文件，这可能会泄漏该文件的内容。
解释	如果攻击者可以公开攻击者选择的任意文件，攻击者就可能公开包含用户或系统凭证、个人或财务信息的敏感文件，以及服务器上的其他敏感文件。攻击者还可能会公开应用程序源代码以进一步分析，从而研究、改进和升级攻击者对应用程序的攻击。 如果攻击者可以通过提供输入确定要读取的文件，而且这些输入未针对文件系统元字符（例如斜杠）进行净化，攻击者就可能选择读取预期范围外的任意文件，从而泄露这些文件的内容。
建议	考虑使用静态解决方案来读取文件，例如使用允许读取的文件列表，或者使用其他文件存储解决方案（如数据库） 如果确实需要从磁盘读取本地文件，一定要保证从特定的文件夹中读取文件，并限制代码只能访问此文件夹 用户提供要读取的文件名时，要避免用户操纵路径访问意外目录，方法是净化文件名中的文件系统元字符，例如： Slashes (/,\) Dot (.) Tilde (~)
CWE	CWE ID 538
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**File Inclusion**
默认严重性	5
摘要	应用程序在 Y（文件） 文件第 M 行使用 YY（元素） 载入了外部库或源代码文件。攻击者可能会利用此漏洞，导致应用程序加载任意代码。
解释	如果攻击者可以选择库的名称或应用程序加载的代码文件的位置，攻击者就可能让应用程序执行任意代码。这样攻击者便能有效地控制应用程序运行的代码。 应用程序使用不可信任的数据指定库或代码文件，未进行适当的净化。这会导致应用程序加载指定的任意代码。然后执行加载的代码。
建议	不要动态加载代码库，尤其是不要根据用户输入加载代码库。 如果需要使用不可信任的数据来选择要加载的库，请验证所选库名称是否与预定义的已列入白名单的库名称集匹配。也可使用值作为标识符从已列入白名单的库中选择。 对用于加载或处理库或代码文件的不可信任的数据进行验证。
CWE	CWE ID 98
OWASP2017	A1-Injection

返回目录

漏洞名称	**File Manipulation**
默认严重性	5
摘要	通过 X（文件） 文件第 N 行中的 XXX（方法） 获取的输入被 Y（文件） 文件第 M 行中 YYY（方法） 用于确定要写入的文件位置，这可能会使攻击者能够更改或损坏文件的内容，或创建全新的文件。
解释	如果攻击者可以影响攻击者选择的任意文件，攻击者就可能覆写或损坏敏感文件，从而可能导致拒绝服务攻击。如果攻击者还能选择要写入的内容，攻击者就可能将代码注入任意文件，从而可能导致执行恶意代码。 用户提供的输入被用于确定要写入哪个文件，这可能会使用户能够影响或操纵任意文件的内容。
建议	考虑为要写入的文件使用静态解决方案，例如经过验证的可写的文件列表，或者使用其他文件存储解决方案，如数据库。如果确实有需要，可通过正确地净化用户提供的输入来设置文件名以将写入目标限制为单个文件夹，并在程序中设置目标文件夹。可考虑根据应用程序代码的业务需求增加一种检查来验证文件是否存在。
CWE	CWE ID 552
OWASP2017	A1-Injection

返回目录

漏洞名称	**File Permission Manipulation**
默认严重性	3
摘要	该程序将调用 X(文件) 第 N 行中的 X（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会导致攻击者能够访问其他受保护的系统资源。允许用户输入直接更改文件权限会导致攻击者能够访问其他受保护的系统资源。

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 file system 中修改权限的操作所使用的路径。
	2. 攻击者能够指定 file system 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 XX(文件) 第 N 行上的 XX(函数) 处），该值会传递到 YY（文件） 第 M 行上的文件系统操作 YY（函数） 中。
	示例：以下代码旨在为用户设置适当的文件权限以通过 FTP 上载网页。它使用来自 HTTP 请求的输入将文件标记为外部用户可查看的文件。
	$rName = $_GET['publicReport'];
	chmod("/home/". authenticateUser . "/public_html/" . rName,"0755");
	...
	但是，如果攻击者为 publicReport 提供恶意值（例如，../../localuser/public_html/.htpasswd），那么应用程序将允许攻击者读取指定文件。
	示例 2：以下代码使用来自配置文件的输入来设置默认权限掩码。如果攻击者可以篡改配置文件，则他们可以使用该程序获得该程序所处理文件的访问权限。如果程序还容易受路径篡改攻击，那么攻击者可能会利用这一漏洞访问系统中的任意文件。
	...
	$mask = $CONFIG_TXT['perms'];
	chmod($filename,$mask);
	...

建议	避免出现 file permission manipulation 的最佳方法是采用间接级：创建一份用户可以指定的合法资源名和文件权限的列表，并且规定用户只能从该列表中进行选择。通过这种方法，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、难以跟踪。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

返回目录

	的字符出现在资源名称和有效权限中，且接受由这些认可的字符所组成的输入。
CWE	CWE ID 264, CWE ID 732
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	**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
摘要	X(文件) 文件中的方法 X(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 defacem
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 XX(文件)的第 N 行的 XX(函数) 中。 2. 数据包含在一个 HTTP 响应头文件里，未经验证就发送给了 Web 用户。 在这种情况下，数据通过 YY（文件）的第 M 行中的 YY（函数） 传送。 如同许多软件安全漏洞一样，Header Manipulation 只是通向终端的一个途径，它本身并不是终端。从本质上看，这些漏洞是显而易见的: 一个攻击者将恶意数据传送到易受攻击的应用程序，且该应用程序将数据包含在 HTTP 响应头文件中。 其中最常见的一种 Header Manipulation 攻击是 HTTP Response Splitting。为了成功地实施 HTTP Response Splitting 盗取，应用程序必须允许将那些包含 CR（回车，由 %0d 或 \r 指定）和 LF（换行，由 %0a 或 \n 指定）的字符输入到头文件中。攻击者利用这些字符不仅可以控制应用程序要发送的响应剩余头文件和正文，还可以创建完全受其控制的其他响应。 如今的许多现代应用程序服务器可以防止 HTTP 头文件感染恶意字符。例如，当新行传递到 header() 函数时，最新版本的 PHP 将生成一个警告并停止创建头文件。如果您的 PHP 版本能够阻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，并在 HTTP 响应的位置字段的头文件中对其进行设置。 <?php $location = $_GET['some_location']; ... header("location: $location"); ?>

返回目录

	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 ... 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

返回目录

	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。 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

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:Cookies**
默认严重性	4
摘要	X(文件) 中的方法 X(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、c
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 XX(文件)的第 N 行的 XX(函数) 中。 2. 数据包含在一个 HTTP Cookie 中，该 Cookie 未经验证就发送给了 Web 用户。 在这种情况下，数据通过 YY（文件）的第 M 行中的 YY（函数） 传送。 如同许多软件安全漏洞一样，Cookie Manipulation 只是通向终端的一个途径，它本身并不是终端。从本质上看，这些漏洞是显而易见的：攻击者可将恶意数据传送到易受攻击的应用程序，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。 <?php

返回目录

```
    $author = $_GET['AUTHOR_PARAM'];
    ...
    header("author: $author");
?&gt;
```

假设在请求中提交了一个字符串，该字符串由标准的字母数字字符组成，如“Jane Smith”，那么包含该 Cookie 的 HTTP 响应可能表现为以下形式：

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

然而，因为 cookie 值来源于未经校验的用户输入，所以仅当提交给 AUTHOR_PARAM 的值不包含任何 CR 和 LF 字符时，响应才会保留这种形式。如果攻击者提交的是一个恶意字符串，比如 "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n....."，那么 HTTP 响应就会被分割成以下形式的两个响应：

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Wiley Hacker
HTTP/1.1 200 OK
...
```

显然，第二个响应已完全由攻击者控制，攻击者可以用任何所需标头和正文内容构建该响应。攻击者可以构建任意 HTTP 响应，从而发起多种形式的攻击，包括：cross-user defacement、web and browser cache poisoning、cross-site scripting 和 page hijacking。

Cross-User Defacement：攻击者可以向一个易受攻击的服务器发出一个请求，导致服务器创建两个响应，其中第二个响应可能会被曲解为对其他请求的响应，而这一请求很可能是与服务器共享相同 TCP 连接的另一用户发出的。这种攻击可以通过以下方式实现：攻击者诱骗用户，让他们自己提交恶意请求；或在远程情况下，攻击者与用户共享同一个连接到服务器（如共享代理服务器）的 TCP 连接。最理想的情况是，攻击者通过这种方式使用户相信自己的应用程序已经遭受了黑客攻击，进而对应用程序的安全性失去信心。最糟糕的情况是，攻击者可能提供经特殊技术处理的内容，这些内容旨在模仿应用程序的执行方式，但会重定向用户的私人信息（如帐号和密码），将这些信息发送给攻击者。

Cache Poisoning： 如果多用户 Web 缓存或者单用户浏览器缓存将恶意构建的响应缓存起来，该响应的破坏力会更大。如果响应缓存在共享的 Web 缓存（如在代理服务器中常见的缓存）中，那么使用该缓存的所有用户都会不断收到恶意内容，直到清除该缓存项为止。同样，如果响应缓存在单个用户的浏览器中，那么在清除该缓存项以前，该用户会不断收到恶意内容。然而，影响仅局限于本地浏览器的用户。

返回目录

	Cross-Site Scripting：一旦攻击者控制了应用程序传送的响应，就可以选择多种恶意内容来传播给用户。Cross-Site Scripting 是最常见的攻击形式，这种攻击在响应中包含了恶意的 JavaScript 或其他代码，并在用户的浏览器中执行。基于 XSS 的攻击手段花样百出，几乎是无穷无尽的，但通常它们都会包含传输给攻击者的私人数据（如 Cookie 或者其他会话信息）。在攻击者的控制下，指引受害者进入恶意的网络内容；或者利用易受攻击的站点，对用户的机器进行其他恶意操作。对于易受攻击的应用程序用户，最常见且最危险的攻击就是使用 JavaScript 将会话和 authentication 信息返回给攻击者，而后攻击者就可以完全控制受害者的帐号了。 Page Hijacking：除了利用一个易受攻击的应用程序向用户传输恶意内容，还可以利用相同的根漏洞，将服务器生成的供用户使用的敏感内容重定向，转而供攻击者使用。攻击者通过提交一个会导致两个响应的请求，即服务器做出的预期响应和攻击者创建的响应，致使某个中间节点（如共享的代理服务器）误导服务器所生成的响应，将本来应传送给用户的响应错误地传给攻击者。因为攻击者创建的请求产生了两个响应，第一个被解析为针对攻击者请求做出的响应，第二个则被忽略。当用户通过同一 TCP 连接发出合法请求时，攻击者的请求已经在此处等候，并被解析为针对受害者这一请求的响应。这时，攻击者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

返回目录

	只接受完全由这些受认可的字符组成的输入。例如，有效的用户名可能仅包含字母数字字符，帐号可能仅包含 0-9 的数字。 更灵活的方法是执行拒绝列表，但其安全性较差，该列表会在使用输入之前有选择地拒绝或避免潜在的危险字符。为了创建这样的列表，首先需要了解在 HTTP 响应头文件中具有特殊含义的一组字符。尽管 CR 和 LF 字符是 HTTP Response Splitting 攻击的核心，但其他字符，如":"（冒号）和 '='（等号），在响应标头中同样具有特殊的含义。 在应用程序中确定针对 Header Manipulation 攻击执行验证的正确要点，以及验证过程中要考虑的特殊字符之后，下一个难点就是确定验证过程中处理各种特殊字符的方式。应用程序应拒绝任何要添加到 HTTP 响应头文件中的包含特殊字符的输入，这些特殊字符（特别是 CR 和 LF）是无效字符。 许多应用程序服务器都试图避免应用程序出现 HTTP Response Splitting 漏洞，其做法是为负责设置 HTTP 头文件和 cookie 的函数提供各种执行方式，以检验是否存在进行 HTTP Response Splitting 攻击必需的字符。不要依赖运行应用程序的服务器，以此确保该应用程序的安全。开发了某个应用程序后，并不能保证在其生命周期中它会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
摘要	X(文件) 中的 X(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 XX(文件)的第 N 行的 XX(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 YY（文件）的第 M 行的 YY（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： $subject = $_GET['subject']; $body = $_GET['body']; mail("support@acme.com", "[Contact us query] " . $subject, $body); 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： ... 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

返回目录

	... 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:Overly Permissive CORS Policy**
默认严重性	3
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。如果不实施同源策略，恶意网站可能就会使用客户端凭据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。 <?php header('Access-Control-Allow-Origin: *'); ?> 将 * 作为 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。 <?php header('Access-Control-Allow-Origin: www.trusted.com'); ?>
CWE	CWE ID 942
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**HTTP Parameter Pollution**
默认严重性	3
摘要	X(文件) 文件将未验证的数据传递给第 N 行的 HTTP URL 重定向。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 URL 中加入未验证的输入，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	HTTP Parameter Pollution (HPP) 攻击包含将编码的查询字符串分隔符注入其他现有的参数中。如果 Web 应用程序没有正确地检查用户输入，恶意用户可能会破坏应用程序的逻辑，进行客户端侧或服务器侧攻击。如果再向 Web 应用程序提交参数，并且如果这些参数与现有参数的名称相同，则 Web 应用程序可能会有下列一种反应： 它可能只从第一个参数中提取数据 它可能从最后一个参数中提取数据 它可能从所有参数中提取数据并把它们连接起来 Technology/HTTP back-end Overall Parsing Result Example ASP.NET/IIS All occurrences of the specific parameter par1=val1,val2 ASP/IIS All occurrences of the specific parameter par1=val1,val2 PHP/Apache Last occurrence par1=val2 JSP,Servlet/Apache Tomcat First occurrence par1=val1 JSP,Servlet/Oracle Application Server 10g First occurrence par1=val1 IBM HTTP Server First occurrence par1=val1 mod_perl/Apache Becomes an array ARRAY(0x8b9059c) 例 1：根据应用程序服务器和应用程序自身的逻辑，下列请求可能造成与身份验证系统的混淆，使攻击者能模拟别的用户。 http://www.server.com/login.php?name=alice&name=hacker 例 2：下列代码使用来自于 HTTP 请求的输入来呈现两个超链接。 <% ... $id = $_GET["id"];

返回目录

	header("Location: http://www.host.com/election.php?poll_id=" . $id); ... %> URL：　http://www.host.com/election.php?poll_id=4567 链接 1：<a href="vote.php?poll_id=4567&candidate=white">White 先生的投票<a> 链接 2：<a href="vote.php?poll_id=4567&candidate=green">Green 女士的投票<a> 程序员没有考虑到攻击者可能提供 poll_id（例如“4567&candidate=green”）的可能性，届时得到的页面中将包含以下注入链接，因此，Green 女士将始终投票选择捡出第一个参数的应用程序服务器。 <a href="vote.php?poll_id=4567&candidate=green&candidate=white">White 先生的投票<a> <a href="vote.php?poll_id=4567&candidate=green&candidate=green">Green 女士的投票<a>
建议	在编译 URL 时，未经验证的用户输入应过滤掉多余的参数（删除"&" 字符）。
CWE	CWE ID 235
OWASP2017	A1 Injection

返回目录

漏洞名称	**HTTP Response Splitting**
默认严重性	4
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行 YYY（方法） 方法中用于一个 HTTP 响应头中。在某些无法防御 HTTP 响应拆分攻击的旧版本中，这会导致此类攻击。
解释	如果标头设置代码是有漏洞的版本，则攻击者可能： 通过操纵标头，任意将应用程序服务器的响应标头更改为受害者的 HTTP 请求 通过注入两个连续的换行符任意更改应用程序服务器的响应主体，这可能导致跨站点脚本 (XSS) 攻击 导致缓存中毒，可能控制所有网站的 HTTP 响应（这些响应需要通过与此应用程序相同的代理）。 因为 HTTP 响应头中使用了用户输入，所以攻击者可以添加 NewLine 字符使标头看起来像是包含已经过工程设计的内容的多个标头，从而可能使响应看起来像多个响应（例如，通过设计重复的内容长度标头）。这可能导致组织的代理服务器为受害者的后续请求提供第二个经过工程设计的响应；或者，如果代理服务器也执行响应缓存，攻击者可以立即向另一个站点发送后续请求，使代理服务器将设计的响应缓存为来自该第二站点的响应，稍后再将该响应提供给其他用户。 许多现代 Web 框架默认对插入标头的字符串中的换行符进行净化，从而可以解决此问题。但是，因为许多旧版本的 Web 框架无法自动解决此问题，所以可能需要手动净化输入。
建议	验证所有来源的所有输入（包括 cookie）。验证应使用白名单：仅接受适合指定结构的数据，而不是排除不符合要求的模式。检查： 数据类型 大小 范围 格式 预期值 此外，将用户输入添加到响应标头前，先删除或 URL 编码所有特殊（非字母）字符。 一定要使用最新的框架。
CWE	CWE ID 113
OWASP2017	A1-Injection

返回目录

漏洞名称	**HttpOnlyCookies**
默认严重性	4
摘要	Web 应用程序的 X（文件） 文件第 N 行的 XXX（方法） 方法创建了一个 cookie XX（元素） 并在响应中返回此 cookie。但是，应用程序未配置为自动设置 cookie 的 "httpOnly" 特性，代码也未明确将此添加到 cookie。
解释	用户侧脚本（例如 JavaScript）通常可以访问含有用户会话标识符的 Cookies 或者其他敏感应用 cookies。除非 Web 应用使用 "httpOnly" cookie 标志显式禁止这种情况，否则这些 cookie 可能被恶意客户端脚本（如跨站脚本 (XSS)）读取和访问。根据"深度防御"，这个标志可以减少 XSS 漏洞被发现时造成的损害。 默认情况下，Web 应用框架不会为应用程序的 sessionid cookie 和其他敏感应用程序 cookie 设置 "httpOnly" 标志。同样，应用程序也不会显式使用 "httpOnly" cookie 标志，因此使客户端脚本默认可以访问 cookie。
建议	- 一定要为所有敏感的服务器侧 cookie 设置 "httpOnly" 标志。 - 强烈建议部署 HTTP 严格传输安全 (HSTS)，以确保将在已安全的通道上发送 cookie。 - 由应用程序为每个 cookie 显式设置 "httpOnly" 标志。 - 特别是任何时候将一个 cookie 添加到带 setcookie() 或 setrawcookie() 响应时，要将 $httponly 参数显式设置为 true。 - 建议是，将 PHP 配置为自动为所有 cookies 设置 httpOnly，方法是添加 session.cookie_httponly = true 到 php.ini，或调用 ini_set('session.cookie_httponly', true); - 也可以通过为每个会话设置会话参数来强制执行强默认值，方法是使用 session_set_cookie_params()，将 $httpOnly 设置为 true。 - 如果将 cookie 直接写到响应头，例如使用 "Set-Cookie:" 头名称的 header() 方法，请在 cookie 值末尾追加 ";httpOnly;"。
CWE	CWE ID 1004
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Improper Control of Dynamically Identified Variables**
默认严重性	4
摘要	Y（文件） 文件第 M 行的 YYY（方法） 以不安全的方式解包用户提供的值，可能会使攻击者使用意外值覆盖非预期的变量。
解释	攻击者可能会利用不安全的值解包方式来覆盖全局变量或设置，更改其会话状态或通过更改非预期参数执行带外操作。 某些用于解包变量的方法可能会不加区别地覆盖其他变量。如果不能保证解包的数据集是经过验证的和可信任的，可能会导致意外值被覆盖，从而导致意外或恶意行为。
建议	不要从用户输入中广泛地提取值。使用更精确的方法，只解包和设置预期的和经过验证的值。
CWE	CWE ID 914
OWASP2017	None

返回目录

漏洞名称	**Improper Exception Handling**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法执行预计可能抛出异常的操作，且未正确地封装在 try-catch 块中。这就是“不正确的异常处理”。
解释	攻击者可能会恶意造成异常来导致应用程序崩溃，并可能在特定异常情况下导致拒绝服务 (DoS) 攻击或意外行为。即使没有恶意控制也可能发生异常，造成总体上的不稳定性。 应用程序执行一些可能引发异常的操作，例如访问数据库或文件。因为应用程序并非设计用于正确地处理异常，所以应用程序可能会崩溃。
建议	任何可能导致异常的方法都应该封装在 try-catch 块中： 显式处理预计的异常 包括一个显式处理意外异常的默认解决方案
CWE	CWE ID 248
OWASP2017	None

返回目录

漏洞名称	**Improper Restriction of Stored 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。此 XML 是根据从 X（文件） 文件第 N 行的数据库或文件中读取的已储存数据 XX（元素） 构造的。注意 YY（元素） 已被设置为自动加载和替换 XML 中的所有 DTD 实体引用，包括对外部文件的引用。
解释	应用程序如果能解析和替换不可信任的任意 XML 文档中的 DTD 实体引用，攻击者就能制作 XML 文档来读取任意服务器文件，并将其加载到数据库中。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。 攻击者可能将包含 DTD 声明的 XML 文档（或文档片段）保存到应用程序数据库。特别是此文档可能含有引用服务器磁盘上的本地文件的实体定义时，例如<!ENTITY xxe SYSTEM "file:///c:/boot.ini">。然后，攻击者可以添加一个再引用回实体定义的 XML 实体引用，例如 &xxe; 。 然后，应用程序将从数据库中读取此文档，并将其作为 XML 读取和解析。这将自动将实体值分解为文件，并将其内容直接嵌入到加载的 XML 文档中。然后如果已完全加载的 XML 文档返回用户，结果中就可能包含敏感系统文件的内容。 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 一定要验证所有来源的数据。 即使来自内部数据库或服务器文件，也不要未经验证便自动处理不可信任的数据。 如果确实需要从不可信任的数据加载 XML，请确保 XML 解析器受到限制和约束。 特别是要禁用 DTD 解析和实体分解时。在服务器上应用严格的 XML 模式，并对输入 XML 进行相应验证。 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 解析或实体分解。
CWE	CWE ID 611
OWASP2017	A4-XML External Entities (XXE)

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	**Improper Transaction Handling**
默认严重性	3
摘要	应用程序在 X（文件） 中的 XXX（方法） 方法创建并打开了一个到数据库的连接，并将其加入一次事务中。虽然应用程序将连接封装在一个 try{} 块中以处理异常，但数据库事务处理出现错误时并不是总能回滚。
解释	丢弃的数据库事务（如果其关联连接在提交或回滚事务处理之前关闭）可能会有几个不同的结果，具体取决于使用的实现和特定技术。虽然在某些情况下，如果连接关闭，数据库将自动回滚事务，但更常见的是，它会自动提交中止状态的事务，或者无限期地保持事务打开状态（取决于配置的超时值）。 对于第一种情况下，出现 runtime 异常后提交的事务可能会处于不一致的状态，与当前运行时条件不兼容。这可能会导致损坏系统完整性甚至稳定性的情况。 对于第二种情况，无限期保持活动的事务会导致数据库服务器保留所有受事务影响的记录和表上的锁。这可能会导致一般可靠性和可用性问题，造成延迟、性能下降，甚至因为一个线程一直在等待锁被释放而形成死锁。 不管哪种情况，这都会导致意外状态，并且取决于外部因素，使得应用程序无法控制结果。 应用程序创建了一个到数据库的连接，并在合适的时候提交此连接以显式管理数据库事务。但是，代码不会显式回滚失败的事务，例如在异常的情况下。这会导致应用程序依赖于实现特定的行为，取决于特定技术组合（例如数据库服务器）和结果配置。
建议	一定要在 try{} 块中打开数据库连接并开始事务处理。 关闭数据库连接之前确保没有活动的未提交事务。 出现异常时一定要回滚活动事务。 处理事务后，一定要在 catch{} 块或 finally{} 块中回滚事务。
CWE	CWE ID 460
OWASP2017	None

返回目录

漏洞名称	**Inappropriate Encoding for Output Context**
默认严重性	4
摘要	应用程序使用 X（文件） 文件第 N 行的 XX（元素） 函数编码先前收到的用户输入。但是，编码函数有问题，不能正确地编码所有输入。然后，应用程序将部分编码的输入发送到响应网页，然后此网页输出回客户端。这可能导致跨站脚本攻击。
解释	在某些情况下，使用有问题的编码函数可能会导致跨站点脚本 (XSS) 攻击。这将使攻击者能够窃取用户的密码、请求用户的信用卡信息、提供虚假信息或运行恶意软件。从受害者的角度来看，这是原始网站，受害者会认为其遭受的损失应由原始网站负责。 应用程序创建包含先前用户输入的数据的网页。用户输入被直接嵌入到页面的 HTML 中，使浏览器将其作为网页的一部分显示。此输入在嵌入之前会先编码为 HTML。 但是，编码函数没有 "ENT_QUOTES" 和 "ENT_HTML5" 标志，所以嵌入的撇号、反引号 (`) 和反斜杠未正确编码。如果输入是在撇号或反引号起重要作用的上下文中的输出，例如 HTML 特性值或 JavaScript，则编码将不充分，而未编码的输入将等同于上下文环境进行处理。如果两个参数在同一行中，并且该行是 Javascript，则第一个参数可以使用反斜杠结尾，转义引用的附件，并且后面的参数可能包含 XSS 负载。这样，攻击者将能够在页面中嵌入内容，就像是原始的源页面一样。
建议	通用指南： 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 完全编码所有动态数据，然后再将其嵌入到输出中。 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 建议使用语言或平台内置的函数。 在 Content-Type HTTP 响应标头中，显式定义整个页面的字符编码（字符集）。 在会话 cookie 上设置 httpOnly 标志，以防止 XSS 漏洞窃取 cookie。 具体建议：

返回目录

	调用 htmlentities() 或 htmlspecialchars() 时，一定要将 $flags 参数设置为 ENT_HTML5 | ENT_QUOTES
CWE	CWE ID 838
OWASP2017	A6-Security Misconfiguration

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	**Information Leak Through Persistent Cookies**
默认严重性	3
摘要	应用程序从 X（文件） 文件第 N 行获取密码 XX（元素），然后将其发送回客户，保存在常规 cookie YY（元素）（Y（文件） 文件第 M 行）中。这可能会因保存位置不安全而泄漏敏感数据。
解释	攻击者如果能访问用户桌面或不受保护的网络连接，就可以窃取用户的密码和存储在 cookie 中的其他敏感数据。攻击者窃取密码后，就可以冒充受害者访问应用程序，并做用户可以做的所有事情。因为使用的是用户自己的密码，所以应用程序无法识别差异。 应用程序将敏感用户数据（包括密码）嵌入 cookie 中并发送回用户浏览器。持久性 cookie 保存在用户计算机上的文件中，可能会被未经授权的用户访问，从而暴露用户的密码。
建议	不要在 cookie 中保存除 sessionid 之外的密码或任何其他秘密数据。
CWE	CWE ID 539
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Randomness**
默认严重性	3
摘要	由 X（函数） 实施的随机数生成器不能抵挡加密攻击。标准的伪随机数值生成器不能抵挡各种加密攻击。

解释	在对安全性要求较高的环境中，使用能够生成可预测值的函数作为随机数据源，会产生 Insecure Randomness 错误。
	在这种情况下，生成弱随机数的函数是 X（函数），它位于 X(文件)的第 N 行。
	电脑是一种具有确定性的机器，因此不可能产生真正的随机性。伪随机数生成器 (PRNG) 近似于随机算法，始于一个能计算后续数值的种子。
	PRNG 包括两种类型：统计学的 PRNG 和密码学的 PRNG。统计学的 PRNG 提供很多有用的统计属性，但其输出结果很容易预测，因此容易复制数值流。在安全性所依赖的生成值不可预测的情况下，这种类型并不适用。密码学的 PRNG 生成的输出结果较难预测，可解决这一问题。为保证值的加密安全性，必须使攻击者根本无法、或几乎不可能鉴别生成的随机值和真正的随机值。通常情况下，如果并未声明 PRNG 算法带有加密保护，那么它很可能就是统计学的 PRNG，因此不应在对安全性要求较高的环境中使用，否则会导致严重的漏洞（如易于猜测的密码、可预测的加密密钥、Session Hijacking 和 DNS Spoofing）。
	示例： 下面的代码可利用统计学的 PRNG 为购买产品后仍在有效期内的收据创建一个 URL。
	`function genReceiptURL($baseURL) {`
	`$randNum = rand();`
	`$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 Transport**
默认严重性	4
摘要	调用 X(文件) 中第 N 行的 X（函数） 会使用 HTTP（而非 HTTPS）协议将数据发送到服务器。调用使用 HTTP（而非 HTTPS）协议将数据发送到服务器。
解释	所有数据均通过 HTTP 以明文形式发送，容易受到攻击。 例 1：以下示例通过 HTTP 协议（而不是 HTTPS 协议）发送数据。 ... $client = new Zend_Http_Client('http://www.example.com/fetchdata.php'); $client->request(Zend_Http_Client::POST); ...
建议	使用 HTTPS 将数据发送至服务器。 例 2：以下示例通过 HTTPS 协议（而不是 HTTP 协议）发送数据。 ... $client = new Zend_Http_Client('https://www.example.com/fetchdata.php'); $client->request(Zend_Http_Client::POST); ...
CWE	CWE ID 319
OWASP2017	A3 Sensitive Data Exposure

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	**JSON Injection**
默认严重性	4
摘要	在 X(文件) 的第 N 行中，X(方法) 方法将未经验证的输入写入 JSON。攻击者可以利用此调用将任意元素或属性注入 JSON 实体。该方法会将未经验证的输入写入 JSON。攻击者可以利用此调用将任意元素或属性注入 JSON 实体。

解释	JSON injection 会在以下情况中出现：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 XX(文件)的第 N 行的 XX(函数) 中。
	2. 将数据写入到 JSON 流。
	在这种情况下，由 YY（文件）的第 M 行的 YY（函数） 编写 JSON。
	应用程序通常使用 JSON 来存储数据或发送消息。用于存储数据时，JSON 通常会像缓存数据那样处理，而且可能会包含敏感信息。用于发送消息时，JSON 通常与 RESTful 服务一起使用，并且可以用于传输敏感信息，例如身份验证凭据。
	如果应用程序利用未经验证的输入构造 JSON，则可以更改 JSON 文档和消息的语义。在相对理想的情况下，攻击者可能会插入无关的元素，导致应用程序在解析 JSON 文档或请求时抛出异常。在更为严重的情况下，例如涉及 JSON Injection，攻击者可能会插入无关的元素，从而允许对 JSON 文档或请求中对业务非常关键的值执行可预见操作。还有一些情况，JSON Injection 可以导致 Cross-Site Scripting 或 Dynamic Code Evaluation。
	例 1：以下 PHP 代码将非特权用户（这些用户具有“默认”角色，与之相反，特权用户具有“管理员”角色）的用户帐户身份验证信息从用户控制的 URL 参数 username 和 password 序列化为位于 ~/user_info.json 的 JSON 文件：
	...
	$username = $_GET['username'];
	$password = $_GET['password'];
	$user_info_json_string = '{"role":"default","username":"' . $username . '","password":"' . $password . '"}';
	$user_info_json_file = fopen('~/user_info.json', 'w');
	fwrite($user_info_json_file, $user_info_json_string);
	fclose($user_info_json_file);
	但是，由于 JSON 序列化使用字符串串联来执行，将不会对 username 和 password 中的不可信赖数据进行验证以转义与 JSON 相关的特殊字符。这样，用户就可以任意插入 JSON 密钥，可能会更改已序列化的 JSON 的结构。在本例中，如果非特权用户 mallory（密码为 Evil123!）将 %22,%22role%22:%22 附加到其用户名中，并将

返回目录

	该值传递到 username URL 参数，则最终保存到 ~/user_info.json 的 JSON 将为：
	{
	"role":"default",
	"username":"mallory",
	"role":"admin",
	"password":"Evil123!"
	}
	如果之后使用 PHP 的本地 json_decode() 函数对已序列化的 JSON 文件进行反序列化，如下所示：
	$user_info_json_string = file_get_contents('user_info.json', 'r');
	$user_info_json_data = json_decode($user_info_json_string);
	$user_info_json_data 中 username、password 和 role 的最终值将分别为 mallory、Evil123! 和 admin。在没有进一步验证反序列化 JSON 中的值是否有效的情况下，应用程序会错误地为用户分配 mallory“管理员”特权。

建议	在将用户提供的数据写入 JSON 时，请遵循以下准则：
	1.不要使用从用户输入派生的名称创建 JSON 属性。
	2. 确保使用安全的序列化函数（能够以单引号或双引号分隔不可信赖的数据，并且避免任何特殊字符）执行对 JSON 的所有序列化操作。
	示例 2：以下 PHP 代码实现的功能与 Example 1 相同，但会使用 json_encode() 而不是字符串连接来对数据进行序列化，从而确保正确地分隔和转义任何不可信数据：
	...
	$username = $_GET['username'];
	$password = $_GET['password'];
	$user_info_array = array('role' => 'default', 'username' =>
	$username, 'password' => $password);
	$user_info_json_string = json_encode($user_info_array);
	$user_info_json_file = fopen('~/user_info.json', 'w');
	fwrite($user_info_json_file, $user_info_json_string);
	fclose($user_info_json_file);

CWE	CWE ID 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**Key Management:Empty Encryption Key**
默认严重性	4
摘要	空加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。
解释	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 X（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。 ... $encryption_key = ''; $filter = new Zend_Filter_Encrypt($encryption_key); $filter->setVector('myIV'); $encrypted = $filter->filter('text_to_be_encrypted'); print $encrypted; ... 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。 从 Microsoft(R) Windows(R) 2000 开始，Microsoft(R) 提供 Windows Data Protection Application Programming Interface (DPAPI)，这是一个操作系统层级的服务，用以保护敏感的应用程序数据，如密码或者私人密钥 [1]。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded Encryption Key**
默认严重性	4
摘要	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 X（函数）资源。 示例：下列代码使用 hardcoded 加密密钥来加密信息： ... $encryption_key = 'hardcoded_encryption_key'; //$filter = new Zend_Filter_Encrypt('hardcoded_encryption_key'); $filter = new Zend_Filter_Encrypt($encryption_key); $filter->setVector('myIV'); $encrypted = $filter->filter('text_to_be_encrypted'); print $encrypted; ... 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_encryption_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 PBE Password**
默认严重性	3
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 X（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： ... $hash = hash_pbkdf2('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
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 null，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) 中第 N 行的 X（函数） 的调用中会发现 null 加密密钥。 示例：以下代码会将加密密钥变量初始化为 null。 ... $encryption_key = NULL; $filter = new Zend_Filter_Encrypt($encryption_key); $filter->setVector('myIV'); $encrypted = $filter->filter('text_to_be_encrypted'); print $encrypted; ... 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。 从 Microsoft(R) Windows(R) 2000 开始，Microsoft(R) 提供 Windows Data Protection Application Programming Interface (DPAPI)，这是一个操作系统层级的服务，用以保护敏感的应用程序数据，如密码和加密密钥 [1]。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**LDAP Injection**
默认严重性	4
摘要	方法 X(方法) 可以调用通过未经验证的输入动态生成的 LDAP 筛选器，从而使攻击者可以修改指令的含义。通过用户输入构造的动态 LDAP 筛选器允许攻击者修改指令的含义。
解释	LDAP injection 错误在以下情况下出现： 1.数据从一个不可信赖的数据源进入程序。 在这种情况下，数据进入 XX(文件)的第 N 行的 XX(函数) 中。 2.该数据用于动态构造 LDAP 筛选器。 在这种情况下，数据将传递到 YY（文件）的第 M 行中的 YY（函数）。 示例 1：以下代码动态构造一个 LDAP 查询，并对其加以执行，该查询可以检索所有报告给指定经理的雇员记录。该经理的名字是从 HTTP 请求中读取的，因此不可信任。 ... $managerName = $_POST["managerName"]]; //retrieve all of the employees who report to a manager $filter = "(manager=" . $managerName . ")"; $result = ldap_search($ds, "ou=People,dc=example,dc=com", $filter); ... 在正常情况下，诸如搜索向 John Smith 经理报告的雇员，该代码执行的筛选器如下： (manager=Smith, John) 但是，由于筛选器是通过连接一个常数基本查询串和一个用户输入串动态构造而成的，因此，该查询只在 managerName 不包含任何 LDAP 元字符时才能正常运行。如果攻击者为 managerName 输入字符串 Hacker, Wiley)(|(objectclass=*)，则该查询会变成： (manager=Hacker, Wiley)(|(objectclass=*)) 根据执行查询的权限，增加 |(objectclass=*) 条件会导致筛选器与目录中的所有输入都匹配，而且会使攻击者检索到有关用户输入池的信息。根据执行 LDAP 查询的权限大小，此次攻击的影响范围可能会有所差异，但是如果攻击者可以控制查询的命令结构，那么这样的攻击至少会影响执行 LDAP 查询的用户可以访问的所有记录。
建议	LDAP injection 漏洞的根本原因是攻击者提供了可以改变 LDAP 查询含义的 LDAP 元字符。构造 LDAP 筛选器后，程序员会清楚哪些字符应作为命令解析，而哪些字符应作为数据解析。 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 LDAP 查询中由用户控制的数值完全来自于预定的字符集合，不包含任何上下文中所已使用的 LDAP 元字符。如果由用户控制的数值范围要求它必须包含 LDAP 元字符，则使用相应的编码机制删除这些元字符在 LDAP 查询中的意义。

返回目录

	示例 2：上述例子可以进行重写，以便使用 ldap_escape 构造一个保护指令命令结构的滤字符串。
	...
	$manager = $_POST['manager'];
	$filter = ldap_escape("(manager=" . $manager . ")", null, LDAP_ESCAPE_FILTER);
	if (ldap_search($ds, "ou=People,dc=example,dc=com", $filter)) {
	...
	}

CWE	CWE ID 90
OWASP2017	A1 Injection

返回目录

漏洞名称	**LDAP Manipulation**
默认严重性	3
摘要	X(文件) 中的 X(方法) 方法执行包含未经验证输入的 LDAP 指令，可能会允许攻击者改变指令的含义或执行非法 LDAP 命令。执行一个 LDAP 指令，该指令包含了由用户控制的数值字符串，且使用滤字符串以外的字符，这可能会使攻击者改变指令的含义或执行非法 LDAP 命令。

解释	LDAP manipulation 错误在以下情况中出现:
	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 XX(文件)的第 N 行的 XX(函数) 中。
	2.数据在一个动态 LDAP 指令的滤字符串之外使用。
	在这种情况下，在 YY（文件）中第 M 行的 YY（函数） 使用该数据。
	示例 1: 以下代码从用户处读取 dn 字符串，然后使用该字符串来执行 LDAP 查询。
	$dn = $_POST['dn'];
	if (ldap_bind($ds)) {
	...
	try {
	$rs = ldap_search($ds, $dn, "ou=People,dc=example,dc=com", $attr);
	...
	由于基 dn 来源于用户输入，且在匿名绑定的情况下执行查询，因此攻击者可能会通过指定一个意外的 DN 字符串来篡改查询结果。问题在于开发人员没能充分利用适当的访问控制机制来限制随后的查询，使其只能读取那些允许当前用户读取的雇员记录。

建议	应用程序应该执行精确验证和通过绑定到具体的用户目录的方式来加强 access control 约束，而不是仅仅依赖于表示层来限制用户提交的值。在任何情况下，只要没有适当的权限，都不应该允许用户检索或修改目录中的相关记录。访问目录的每个查询应该执行该策略，这就意味着只有完全受限的查询在匿名绑定的情况下执行，从而有效避免了在 LDAP 系统上建立 access control 机制。
	示例 2: 以下代码实现的功能与 Example 1 相同，但会验证从用户处读取的值，以确保它对应于有效的 dn，并且该代码会使用 LDAP 身份验证来确保该查询只影响允许当前用户访问的记录。check_dn 函数将在预定义的有效基 check_dn 值列表中查找 dn 字符串。
	...
	if (ldap_bind($ld, $username, $password)) {
	$dn = $_POST['dn'];
	if (check_dn($dn)) {
	$rc = ldap_search($ds, $dn, $filter)
	...

返回目录

	} }
CWE	CWE ID 90
OWASP2017	None

返回目录

漏洞名称	**Log Forging**
默认严重性	3
摘要	在 X(文件) 文件第 N 行中，该程序将未经验证的用户输入写入日志。攻击者可以利用这一行为来伪造日志条目或将恶意内容注入日志。将未经验证的用户输入写入日志文件可致使攻击者伪造日志条目或将恶意信息内容注入日志。

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 XX(文件)的第 N 行的 XX(函数) 中。
	2. 数据写入到应用程序或系统日志文件中。
	在这种情况下，数据通过 YY（文件）的第 M 行的 YY（函数） 记录下来。
	为了便于以后的审阅、统计数据收集或调试，应用程序通常使用日志文件来储存事件或事务的历史记录。根据应用程序自身的特性，审阅日志文件可在必要时手动执行，也可以自动执行，即利用工具自动挑选日志中的重要事件或带有某种倾向性的信息。
	如果攻击者可以向随后会被逐字记录到日志文件的应用程序提供数据，则可能会妨碍或误导日志文件的解读。最理想的情况是，攻击者可能通过向应用程序提供包括适当字符的输入，在日志文件中插入错误的条目。如果日志文件是自动处理的，那么攻击者就可以通过破坏文件格式或注入意外的字符，从而使文件无法使用。更阴险的攻击可能会导致日志文件中的统计信息发生偏差。通过伪造或其他方式，受到破坏的日志文件可用于掩护攻击者的跟踪轨迹，甚至还可以牵连第三方来执行恶意行为 [1]。最糟糕的情况是，攻击者可能向日志文件注入代码或者其他命令，利用日志处理实用程序中的漏洞 [2]。
	示例： 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。
	<?php
	$name =$_GET['name'];
	...
	$logout =$_GET['logout'];
	if(is_numeric($logout))
	{
	...
	}
	else
	{
	trigger_error("Attempt to log out: name: $name logout: $val");
	}
	?>

返回目录

	如果用户为 logout 提交字符串“twenty-one”，而且他可以创建一个名为“admin”的用户，则日志中会记录以下条目： PHP Notice: Attempt to log out: name: admin logout: twenty-one 但是，如果攻击者可以创建用户名"admin+logout:+1++++++++++++++++++++++"，则日志中将记录以下条目： PHP Notice: 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	**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	**Object Injection**
默认严重性	4
摘要	X(文件) 中的第 N 行使用不可信赖的数据调用 X（函数）。这种调用允许注入任意 PHP 对象，会导致程序代表攻击者执行恶意命令。反序列化不可信赖的数据会允许注入任意 PHP 对象，导致程序代表攻击者执行恶意命令。

解释	如果不可信赖的数据未经过正确清除即被传递给 unserialize() 函数，则会出现 Object injection 漏洞。攻击者可以将经特殊技术处理的序列化字符串传递到易受攻击的 unserialize() 调用，导致任意 PHP 对象注入应用程序范围。这种漏洞的严重性取决于应用程序范围中可用的类。攻击者会对实施 PHP 幻数方法（如 __wakeup 或 __destruct）的类感兴趣，因为他们可以执行这些方法中的代码。
	例 1：以下代码显示了实施 __destruct() 幻数方法并执行定义为类属性的系统命令的 PHP 类。还有使用用户提供的数据对 unserialize() 进行的不安全调用。
	...
	class SomeAvailableClass {
	public $command=null;
	public function __destruct() {
	system($this->command);
	}
	}
	...
	$user = unserialize($_GET['user']);
	...
	在 Example 1 中，应用程序可能预期获得一个序列化的 User 对象，但攻击者实际上可能提供 SomeAvailableClass 的序列化版本，并为其 command 属性提供一个预定义值：
	GET REQUEST:
	http://server/page.php?user=O:18:"SomeAvailableClass":1:{s:7:"command";s:8:"uname -a";}
	一旦没有对 $user 对象的其他引用，析构函数方法将被调用并执行攻击者提供的命令。
	攻击者可以使用称为“面向属性编程”的技术链接在调用易受攻击的 unserialize() 时声明的不同类，该技术由 Stefan Esser 在 BlackHat 2010 会议上提出。利用该技术，攻击者可以重复使用现有代码以生成其自己的负载。

建议	应当禁止用户直接控制程序未序列化的命令。如果必须序列化用户输入，请使用其他序列化标准，如 JSON。 不要因为不存在实施危险幻数方法的类，就毫无顾忌地使用 unserialize()，尤其是在可以使用插件扩展的模块化应用程序中，因

返回目录

	为新类可能会加载到不同的环境中，致使攻击者能够重复使用现有代码生成恶意负载。
CWE	CWE ID 502
OWASP2017	A8 Insecure Deserialization

返回目录

漏洞名称	**Often Misused:File Upload**
默认严重性	2
摘要	X(文件) 文件中的函数 X(方法) 会调用第 N 行的 X（函数）。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 PHP 解释器，他们就能促使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。攻击者可以将恶意的 PHP 源文件上传到该程序中，并随后从服务器中请求这些文件，这会促使 PHP 解析器执行这些文件。
	<?php
	$udir = 'upload/'; // Relative path under Web root
	$ufile = $udir . basename($_FILES['userfile']['name']);
	if (move_uploaded_file($_FILES['userfile']['tmp_name'], $ufile)) {
	echo "Valid upload received\n";
	} else {
	echo "Invalid upload rejected\n";
	} ?>
	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	除非程序特别要求用户上传文件，否则请通过在 php.ini 文件中包含以下条目来禁用 file_uploads 选项： file_uploads = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 file_uploads 选项： php_flag file_uploads off 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。 例 2：下面的代码演示了一种严重受限的文件上传机制，这种机制将上传的内容存储在一个无法通过 Web 访问的目录中。 ` <?php

返回目录

	$udir = '/var/spool/uploads/'; # Outside of Web root
	$ufile = $udir . basename($_FILES['userfile']['name']);
	if (move_uploaded_file($_FILES['userfile']['tmp_name'], $ufile)) {
	echo "Valid upload received\n";
	} else {
	echo "Invalid upload rejected\n";
	} ?>
	尽管这种机制可阻止攻击者直接请求上传的文件，但它对针对程序中存在的其他漏洞发起的攻击没有任何作用，这些漏洞也能让攻击者利用上传的内容。阻止这种攻击的最好方法是，使攻击者难以破译上传的文件的名称和位置。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 [3]。

CWE	CWE ID 434
OWASP2017	A1 Injection

返回目录

漏洞名称	**Open Redirect**
默认严重性	4
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 XX(文件) 的第 N 行中的 XX(函数) 接受客户端即将被重定向到的 URL。 数据通过 YY（文件） 的第 M 行中的 YY（函数） 传送。 例 1：以下 PHP 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 = $_GET["dest"]; header("Location: " . $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.php?dest=www.wilyhacker.com”链接，该用户就有可能会单击该链接，因为他会以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.php?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

返回目录

	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。
	例 2：以下代码引用了一个通过有效 URL 传播的数组。用户单击的链接将通过与所需 URL 对应的数组索引来传递。
	<%
	...
	$strDest = intval($_GET["dest"]);
	if(($strDest >= 0) && ($strDest <= count ($strURLArray) - 1))
	{
	$strFinalURL = $strURLArray[strDest];
	header("Location: " . $strFinalURL);
	}
	...
	%>
	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	**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 Management**
默认严重性	3
摘要	X(文件) 中的 X(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 XX(文件) 中第 N 行的 XX(函数) 读取到程序中，并用于访问 YY（文件） 中第 M 行的 YY（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 = file('config.properties', FILE_IGNORE_NEW_LINES | FILE_SKIP_EMPTY_LINES);
	$password = $props[0];
	$link = mysql_connect($url, $usr, $password);
	if (!$link) {
	die('Could not connect: ' . mysql_error());
	}
	...
	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
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 X（函数） 的调用中发现空密码。 示例：以下代码尝试使用空密码连接到数据库。 <?php ... $connection = mysql_connect($host, 'scott', ''); ... ?> 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。 从 Microsoft(R) Windows(R) 2000 开始，Microsoft(R) 提供 Windows Data Protection Application Programming Interface (DPAPI)，这是一个操作系统层级的服务，用以保护敏感的应用程序数据，如密码或者私人密钥 [1]。
CWE	CWE ID 259
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Hardcoded Password**
默认严重性	4
摘要	Hardcoded password 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，密码用于访问 X(文件)中第 N 行的 X（函数） 资源。
	示例：以下代码用 hardcoded password 来连接数据库：
	...
	$link = mysql_connect($url, 'scott', 'tiger');
	if (!$link) {
	die('Could not connect: ' . mysql_error());
	}
	...
	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。 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 259, CWE ID 798
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Null Password**
默认严重性	4
摘要	Null 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 X（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。
	<?php
	...
	$storedPassword = NULL;
	if (($temp = getPassword()) != NULL) {
	$storedPassword = $temp;
	}
	if(strcmp($storedPassword,$userPassword) == 0) {
	// Access protected resources
	...
	}
	...
	?>
	如果 readPassword() 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。 从 Microsoft(R) Windows(R) 2000 开始，Microsoft(R) 提供 Windows Data Protection Application Programming Interface (DPAPI)，这是一个操作系统层级的服务，用以保护敏感的应用程序数据，如密码或者私人密钥 [1]。
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
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
摘要	调用 X（函数） 给密码加密并不能提供任何意义上的保护。采用普通的编码方式给密码加密并不能有效地保护密码。

解释	当密码以明文形式存储在应用程序的属性文件或其他配置文件中时，会发生 password management 漏洞。程序员试图通过编码函数来遮蔽密码，以修补 password management 漏洞，例如使用 64 位基址编码方式，但都不能起到充分保护密码的作用。
	在这种情况下，密码会通过 XX(文件) 中第 N 行的 XX(函数) 读取到程序中，并用于访问 YY（文件） 中第 M 行的 YY（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 = file('config.properties', FILE_IGNORE_NEW_LINES | FILE_SKIP_EMPTY_LINES);
	$password = base64_decode($props[0]);
	$link = mysql_connect($url, $usr, $password);
	if (!$link) {
	die('Could not connect: ' . mysql_error());
	}
	...
	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
摘要	攻击者可以控制 X(文件) 中第 N 行的 X（函数） 文件系统路径参数，借此访问或修改原本受保护的文件。允许用户输入控制文件系统操作所用的路径会导致攻击者能够访问或修改其他受保护的系统资源。
解释	当满足以下两个条件时，就会产生 path manipulation 错误： 1.攻击者能够指定某一文件系统操作中所使用的路径。 2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。 例如，在某一程序中，攻击者可以获得特定的权限，以重写指定的文件或是在其控制的配置环境下运行程序。 在这种情况下，攻击者可以指定通过 XX(文件) 中第 N 行的 XX(函数) 进入程序的值，这一数值可以通过 YY（文件） 中第 M 行的 YY（函数） 访问文件系统资源。 示例 1：以下代码使用来自于 HTTP 请求的输入来创建一个文件名。程序员没有考虑到攻击者可能使用像“../../tomcat/conf/server.xml”一样的文件名，从而导致应用程序删除它自己的配置文件。 $rName = $_GET['reportName']; $rFile = fopen("/usr/local/apfr/reports/" . rName,"a+"); ... unlink($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"; $handle = fopen($filename,"r"); $amt = fread($handle, filesize($filename)); echo $amt; ...
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

返回目录

	其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	CWE ID 22, CWE ID 73
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Path 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	**PHP Misconfiguration:allow_url_fopen Enabled**
默认严重性	3
摘要	对远程文件进行操作可能会让攻击者将恶意内容注入到程序中。

解释	如果启用 allow_url_fopen 选项，就可以通过 HTTP 或 FTP URL 在远程文件上执行接受文件名的 PHP 函数。该选项是在 PHP 4.0.4 中引入的，并且默认情况下是启用的，由于它可能会导致攻击者将恶意内容引入程序中，因此具有危险性。最理想的情况是，攻击者篡改了远程文件来引入到恶意内容，因此对远程文件进行操作导致应用程序容易受到攻击。最糟糕的情况是，攻击者控制了运行应用程序的 URL，这样一来，攻击者就可以将一个 URL 提供给远程服务器，从而可以将任意恶意内容注入到应用程序中。
	例 1：以下代码可打开一个文件并读取其内容，文件名由一个请求参数控制。由于 $file 的值由一个请求参数控制，因此攻击者有可能违背程序员的假设，将一个 URL 提供给远程文件。
	<?php
	$file = fopen ($_GET["file"], "r");
	if (!$file) {
	// handle errors
	}
	while (!feof ($file)) {
	$line = fgets ($file, 1024);
	// operate on file content
	}
	fclose($file);
	?>

建议	不要允许任意 file system 函数访问远程文件。而应使用来自 libcurl 的 cURL 函数在必要时显式授予对远程文件的访问权限。 例 2：以下条目来自 php.ini 文件，用于禁用 allow_url_fopen 选项： allow_url_fopen = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 allow_url_fopen 选项： php_flag allow_url_fopen off
CWE	CWE ID 94
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:allow_url_include Enabled**
默认严重性	4
摘要	包含那些引用远程文件的指令可能会造成攻击者将恶意内容注入程序中。
解释	如果启用 allow_url_include 选项，会导致用于指定在当前页面包含某个文件的 PHP 函数，如 include() 和 require()，接受指向远程文件的 HTTP 或 FTP URL。该选项是在 PHP 5.2.0 中引入的，并且默认情况下被禁用，由于它可能会导致攻击者将恶意内容引入程序中，因此具有危险性。最理想的情况是，攻击者篡改了远程文件来引入恶意内容，因此包含远程文件导致应用程序容易受到攻击。最糟糕的情况是，攻击者控制了应用程序用来指定要包含的远程文件的 URL，这样一来，攻击者就可以将一个 URL 提供给远程服务器，从而可以将任意恶意内容注入应用程序中。
建议	如果启用 allow_url_include 选项，会导致用于指定在当前页面包含某个文件的 PHP 函数，如 include() 和 require()，接受指向远程文件的 HTTP 或 FTP URL。该选项是在 PHP 5.2.0 中引入的，并且默认情况下被禁用，由于它可能会导致攻击者将恶意内容引入程序中，因此具有危险性。最理想的情况是，攻击者篡改了远程文件来引入恶意内容，因此包含远程文件导致应用程序容易受到攻击。最糟糕的情况是，攻击者控制了应用程序用来指定要包含的远程文件的 URL，这样一来，攻击者就可以将一个 URL 提供给远程服务器，从而可以将任意恶意内容注入应用程序中。 建议： 不允许使用远程包含指令。以下条目来自 php.ini 文件，用于禁用 allow_url_include 选项： allow_url_include = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 allow_url_include 选项： php_flag allow_url_include off
CWE	CWE ID 94
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:cgi.force_redirect Disabled**
默认严重性	4
摘要	如果允许用户通过 Web 直接调用 PHP 解释器，可能会让攻击者绕过权限检查而访问服务器上的受保护文件。
解释	如果 PHP 解释器作为 CGI 二进制码来安装，则对 PHP 资源的请求通常会被 Web 服务器重定向到该解释器。在这种情况下，当用户请求 http://www.example.com/content/page.php 时，服务器首先会对 /content 目录执行必要的 access control 检查，然后将访问控制重定向到 PHP 解释器，同时发出对 http://www.example.com/cgi-bin/php/content/page.php 的请求。 如果禁用 cgi.force_redirect 选项（该选项默认情况下是启用的），对 /cgi-bin/php 目录具有访问权限的攻击者就可以利用 PHP 解释器权限访问任意 Web 文档，从而绕过服务器将执行的任何 access control 检查。
建议	如果您将 PHP 作为 CGI 二进制码运行，并且您的服务器支持该操作，请启用 cgi.force_redirect 选项。PHP 支持一种适用于多种服务器的直接模块接口，这些服务器包括 Apache、Microsoft IIS、Netscape 和 iPlanet。如果服务器要求您禁用 cgi.force_redirect 选项，以便将 PHP 作为 CGI 二进制码来运行，请慎重考虑将 PHP 作为 CGI 二进制码运行所获得的功能是否值得您冒此安全风险。 以下条目来自 php.ini 文件，用于启用 cgi.force_redirect 选项： cgi.force_redirect = 'on' 也可以通过在 Apache httpd.conf 文件中包含以下条目来启用 cgi.force_redirect 选项： php_flag cgi.force_redirect on
CWE	CWE ID 305
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:file_uploads Enabled**
默认严重性	2
摘要	允许用户上传文件可能会让攻击者注入危险内容或者执行恶意代码

解释	如果启用 file_uploads 选项，可能会让 PHP 用户将任意文件上传到服务器中。允许用户上传文件这一功能本身并不代表一种安全漏洞。但是，这一功能可能会引起一系列的攻击，因为它让恶意用户能够随意向服务器环境中引入数据。
	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 PHP 解释器，他们就能促使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。攻击者可以将恶意的 PHP 源文件上传到该程序中，并随后从服务器中请求这些文件，这会促使 PHP 解析器执行这些文件。
	<?php
	$udir = 'upload/'; // Relative path under Web root
	$ufile = $udir . basename($_FILES['userfile']['name']);
	if (move_uploaded_file($_FILES['userfile']['tmp_name'], $ufile)) {
	echo "Valid upload received\n";
	} else {
	echo "Invalid upload rejected\n";
	} ?>
	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	除非程序特别要求用户上传文件，否则请通过在 php.ini 文件中包含以下条目来禁用 file_uploads 选项： file_uploads = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 file_uploads 选项： php_flag file_uploads off 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。大多数依赖于上传的内容所发起的攻击都需要攻击者能够提供他们所选的内容，限制这种内容可以大大减小构成攻击的可能性。 例 2：下面的代码演示了一种严重受限的文件上传机制，这种机制将上传的内容存储在一个无法通过 Web 访问的目录中。 `

返回目录

	<?php
	$udir = '/var/spool/uploads/'; # Outside of Web root
	$ufile = $udir . basename($_FILES['userfile']['name']);
	if (move_uploaded_file($_FILES['userfile']['tmp_name'], $ufile)) {
	echo "Valid upload received\n";
	} else {
	echo "Invalid upload rejected\n";
	} ?>
	尽管这种机制可阻止攻击者直接请求上传的文件，但它对针对程序中存在的其他漏洞发起的攻击没有任何作用，这些漏洞也能让攻击者利用上传的内容。阻止这种攻击的最好方法是，使攻击者难以破译上传的文件的名称和位置。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 [3]。

CWE	CWE ID 434
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:magic_quotes_gpc Enabled**
默认严重性	4
摘要	避免自动输入这一方法收效甚微，对许多攻击类型起不到任何防范作用。
解释	如果启用 magic_quotes_gpc 和 magic_quotes_runtime 配置选项，它们会导致 PHP 自动转义输入中所有的 '（单引号）、"（双引号）、\（斜杠）及 NUL 字符，输入内容可通过 Web（GET、POST、以及 Cookie 或 GPC）和其他输入源（数据库、file system 等或运行时环境）分别读取。这种在服务器级别执行输入验证的做法非常软弱无力，难以防范 SQL injection 类型的攻击（尽管它就是被设计成针对这种漏洞提供保护的），对于那些针对 Web 应用程序发起的其他攻击更没有任何防范作用。正是由于认识到它们的不足之处，从 PHP 6 开始已经弃用并删除了这两个选项。
建议	不要使用或依赖于 magic_quotes_gpc 和 magic_quotes_runtime 配置选项。在早于 6.0 的 PHP 版本中，通过在 php.ini 文件中包含以下条目来显式禁用这两个配置选项: magic_quotes_gpc = 'off' magic_quotes_runtime = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 magic_quotes_gpc 和 magic_quotes_runtime 选项: php_flag magic_quotes_gpc off php_flag magic_quotes_runtime off 不要奢望能够自动避开故障，而应基于目前可用的最强的验证机制，针对上下文执行适当的输入验证。最强的输入验证形式采用一种迂回战术：创建一个合法值列表，用户可以指定其中的值，并且只能从该列表中选择值。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。 在某些情况下，这种方法是不切实际的，因为合法输入值集过大，或者很难跟踪。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 20
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:magic_quotes_runtime Enabled**
默认严重性	4
摘要	避免自动输入这一方法收效甚微，对许多攻击类型起不到任何防范作用。
解释	如果启用 magic_quotes_gpc 和 magic_quotes_runtime 配置选项，它们会导致 PHP 自动转义输入中所有的 '（单引号）、"（双引号）、\（斜杠）及 NUL 字符，输入内容可通过 Web（GET、POST、以及 Cookie 或 GPC）和其他输入源（数据库、file system 等或运行时环境）分别读取。这种在服务器级别执行输入验证的做法非常软弱无力，难以防范 SQL injection 类型的攻击（尽管它就是被设计成针对这种漏洞提供保护的），对于那些针对 Web 应用程序发起的其他攻击更没有任何防范作用。正是由于认识到它们的不足之处，从 PHP 6 开始已经弃用并删除了这两个选项。
建议	不要使用或依赖于 magic_quotes_gpc 和 magic_quotes_runtime 配置选项。在早于 6.0 的 PHP 版本中，通过在 php.ini 文件中包含以下条目来显式禁用这两个配置选项： magic_quotes_gpc = 'off' magic_quotes_runtime = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 magic_quotes_gpc 和 magic_quotes_runtime 选项： php_flag magic_quotes_gpc off php_flag magic_quotes_runtime off 不要奢望能够自动避开故障，而应基于目前可用的最强的验证机制，针对上下文执行适当的输入验证。最强的输入验证形式采用一种迂回战术：创建一个合法值列表，用户可以指定其中的值，并且只能从该列表中选择值。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。 在某些情况下，这种方法是不切实际的，因为合法输入值集过大，或者很难跟踪。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 20
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:magic_quotes_sybase Enabled**
默认严重性	4
摘要	避免自动输入这一方法收效甚微，对许多攻击类型起不到任何防范作用。
解释	如果启用，且 magic_quotes_gpc 或 magic_quotes_runtime 同时启用，则 magic_quotes_sybase 配置选项会用单引号而不是反斜杠对单引号进行转义。它还会完全覆盖 magic_quotes_gpc。在这种情况下，即使启用 magic_quotes_gpc ，无论双引号、反斜杠还是 NUL 均不会进行转义。 如果启用 magic_quotes_gpc 和 magic_quotes_runtime 配置选项，它们会导致 PHP 自动转义输入中所有的 '（单引号）、"（双引号）、\（斜杠）及 NUL 字符，输入内容可通过 Web（GET、POST、以及 Cookie 或 GPC）和其他输入源（数据库、文件系统等或运行时环境）分别读取。这种在服务器级别执行输入验证的做法非常软弱无力，难以防范 SQL injection 类型的攻击（尽管它就是被设计成针对这种漏洞提供保护的），对于那些针对 Web 应用程序发起的其他攻击更没有任何防范作用。正是由于认识到它们的不足之处，此功能已从 PHP 5.3.0 开始弃用并从 PHP 5.4.0 开始删除。magic_quotes_gpc 和 magic_quotes_runtime 已从 PHP 6 中弃用并删除。
建议	不要使用或依赖于 magic_quotes_sybase 配置选项。在早于 5.3.0 的 PHP 版本中，通过在 php.ini 文件中包含以下条目来显式禁用这两个配置选项: magic_quotes_sybase = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 magic_quotes_sybase 选项: php_flag magic_quotes_sybase off 不要奢望能够自动避开故障，而应基于目前可用的最强的验证机制，针对上下文执行适当的输入验证。最强的输入验证形式采用一种迂回战术：创建一个合法值列表，用户可以指定其中的值，并且只能从该列表中选择值。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。 在某些情况下，这种方法是不切实际的，因为合法输入值集过大，或者很难跟踪。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 20

返回目录

OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:Missing safe_mode_exec_dir Entry**
默认严重性	4
摘要	如果将 PHP 配置为调用任意程序，会导致攻击者能执行恶意命令。
解释	如果启用 safe_mode，safe_mode_exec_dir 选项会限制 PHP 仅从指定的目录执行命令。尽管禁用 safe_mode_exec_dir 条目这一做法本身并不是一种安全漏洞，但这一画蛇添足的行为可能会被攻击者利用，并结合其他漏洞制造更大的危机。
建议	启用 safe_mode，为 safe_mode_exec_dir 指定尽可能严格的值，从而允许程序执行任何所需的本机程序，但同时又阻止攻击者执行敏感或危险程序。以下条目来自 php.ini 文件，用于限制 PHP 仅执行 /usr/bin/php 目录下的程序: safe_mode = 'on' safe_mode_exec_dir = '/usr/bin/php' 也可以通过在 Apache httpd.conf 文件中包含以下条目来设置这些选项: php_flag safe_mode on php_flag safe_mode_exec_dir /usr/bin/php
CWE	CWE ID 553
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:Poor open_basedir Configuration**
默认严重性	3
摘要	open_basedir 配置可指定可以更改的工作目录。
解释	如存在，open_basedir 配置选项将试图阻止 PHP 程序在 php.ini 中指定的目录树外的文件上运行。如果使用 . 指定工作目录，则攻击者可能会使用 chdir() 更改此目录。 尽管 open_basedir 选项从总体上有利于保证系统的安全性，但它的实施效果却受到 race condition 的不利影响，这种状况可能会允许攻击者在某些情况下绕过该选项所定义的限制条件 [2]。在 PHP 执行访问权限检查与打开文件的两个时刻之间，存在一种 TOCTOU（检查时间，使用时间）race condition。与其他语言中 file system 的 race condition 一样，这种紊乱状况会导致攻击者能够将指向一个通过 access control 检查的文件的 symlink 替换为另一个本不能通过测试的文件，从而获得对受保护文件的访问权限。 这种攻击针对的漏洞大小取决于执行访问检查与打开文件这两个时刻之间的时间间隙。即使连续执行调用，现今的操作系统也无法确保在进程让出 CPU 之前将执行的代码数量。攻击者掌握了多种扩大该时间间隙的技术，以便更加容易地发起攻击，但即使是一段很短的间隙，该攻击企图也可以不断地重复，直到成功为止。
建议	确定您的程序需要访问的最小目录集，并使用 open_basedir 选项来限制程序只能访问这些目录，而无需指定当前的工作目录。 为了防止在实施 open_basedir 的过程中针对 race condition 发起的攻击，请通过在 php.ini 文件中包含以下条目来禁用 PHP symlink() 函数： disable_functions = symlink 与其依赖内置的 PHP 机制，不如考虑在操作系统中创建一个底层 chroot 监牢，限制 PHP 或 Web 服务器进程可访问的文件系统区域 [4]。
CWE	CWE ID 284
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:register_globals Enabled**
默认严重性	5
摘要	如果将 PHP 配置为对所有的 Environment、GET、POST、Cookie 及 Server 变量进行全局注册，会导致意外行为，使系统容易受到攻击者的攻击。

解释	如果启用 register_globals 选项，会导致 PHP 对 EGPCS（Environment、GET、POST、Cookie 及 Server）变量进行全局注册，这样一来，任何用户在任何 PHP 程序中都将能访问这些变量。如果程序员在编写程序时启用此选项，或多或少都会导致程序察觉不到它们所依赖于的数值来源，这会导致运行正常的环境发生意外行为，使系统容易受到恶意环境中的攻击者发起的攻击。由于认识到 register_globals 所隐含的安全隐患，在 PHP 4.2.0 中默认禁用了该选项，而在 PHP 6 中弃用并删除了该选项。
	示例 1：以下代码容易受到 cross-site scripting 攻击。程序员假定 $username 的值来源于由服务器控制的会话，但是攻击者可能会为 $username 提供一个恶意值来代替请求参数。如果启用 register_globals 选项，此代码会在它所生成的 HTML 内容中包含由攻击者提交的恶意值。
	<?php
	if (isset($username)) {
	echo "Hello $username";
	} else {
	echo "Hello Guest
	";
	echo "Would you like to login?";
	}
	?>

建议	不要使用或依赖于 register_globals 选项。在早于 6.0 的 PHP 版本中，通过在 php.ini 中包含以下条目显式禁用了 register_globals 选项： register_globals = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 register_globals 选项： php_flag register_globals off
CWE	CWE ID 473
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:safe_mode Disabled**
默认严重性	4
摘要	如果允许 PHP 脚本访问服务器上的任意文件，会使得攻击者能够操纵敏感文件或者执行恶意命令。
解释	safe_mode 选项是 PHP 中最重要的 security features 之一。如果禁用 safe_mode 选项，PHP 会以调用它的用户所具有的权限对文件进行操作，而该用户通常是一个特权用户。尽管将 PHP 配置为禁用 safe_mode，这一做法本身不会引发安全漏洞，但这一画蛇添足的行为可能会被攻击者利用，并结合其他漏洞制造更大的危机。
建议	启用 safe_mode 选项，以防止攻击者获取对敏感文件的访问权限；可通过在 php.ini 中包含以下条目来启用该选项: safe_mode = 'on' 也可以通过在 Apache httpd.conf 文件中包含以下条目来启用该选项: php_flag safe_mode on
CWE	CWE ID 552
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**PHP Misconfiguration:session_use_trans_sid Enabled**
默认严重性	4
摘要	如果将 PHP 配置为通过 URL 传递会话 ID，将容易受到 session fixation 和 session hijacking 攻击。
解释	如果启用 session.use_trans_sid，会导致 PHP 通过 URL 传递会话 ID，这样一来，攻击者就更容易劫持当前会话，或者哄骗用户使用已被攻击者控制的现有会话。 通过 URL 传递的参数比 POST 参数和 cookie 值的可见性更好，这是因为它们通常出现在浏览器历史记录、书签、日志文件及其他高度明显的位置。如果攻击者了解到系统上某一当前会话的机密会话标识符，他们就有可能将会话标识符返回给程序，以便劫持用户会话。 除了 session hijacking 外，通过 URL 暴露会话 ID 还会引发 session fixation 攻击。在通常情况下，攻击者会利用 Session Fixation 漏洞在 Web 应用程序中创建一个新会话，并且记录与之关联的会话标识符。然后，攻击者会设法使受害者在服务器中的验证失败，从而通过当前会话来访问用户帐号。通过 URL 传递会话标识符使得攻击者有可能攻击一大批受害者，这种攻击的方式是将包含受到危害的会话标识符的 URL 通过电子邮件发送给受害者，或另外一种大规模通信机制来实现。
建议	禁用 session.use_trans_sid 选项，以防止攻击者获取对会话标识符的访问权限和发动 session fixation 攻击；可通过在 php.ini 中包含以下条目来禁用该选项： session.use_trans_sid = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用该选项： php_flag session.use_trans_sid off
CWE	CWE ID 384
OWASP2017	A2 Broken Authentication

返回目录

漏洞名称	**Poor Error Handling:Empty Catch Block**
默认严重性	2
摘要	X(文件) 中的 X(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。
	示例 1：下面摘录的代码会忽略一个由 doExchange() 抛出的罕见异常。
	try {
	doExchange();
	}
	catch (exception $e) {
	// this can never happen
	}
	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。
	示例 2：Example 1 中的代码可通过以下方式重写：
	try {
	doExchange();
	}
	catch (exception $e) {
	error_log("A RareException has occurred!", 3, "/var/log/app.log")
	}

CWE	CWE ID 1069
OWASP2017	None

返回目录

漏洞名称	**Poor Error Handling:Return Inside Finally**
默认严重性	3
摘要	X(文件) 中的 X(方法) 方法从 finally 块中的第 N 行返回，这会导致异常的丢失。从 finally 块中返回会导致异常的丢失。

解释	finally 块中的返回指令会导致从 try 块中抛出的异常丢失。
	示例 1：在下列代码中，第二次调用 doMagic 方法，同时将参数 True 传递给该方法会导致抛出 exception 异常，该异常将不会传递给调用者。finally 块中的返回指令会导致异常的丢弃。
	<?php
	echo "Watch as this magical code makes an exception " .
	"disappear before your very eyes!" . PHP_EOL;
	echo "First, the kind of exception handling " .
	"you're used to:" . PHP_EOL;
	try {
	doMagic(False);
	} catch (exception $e) {
	// An exception will be caught here
	echo $e->getMessage();
	}
	echo "Now, the magic:" . PHP_EOL;
	try {
	doMagic(True);
	} catch (exception $e) {
	// No exception caught here, the finally block ate it
	echo $e->getMessage();
	}
	echo "Tada!" . PHP_EOL;
	function doMagic($returnFromFinally) {
	try {
	throw new Exception("Magic Exception" . PHP_EOL);
	}
	finally {
	if ($returnFromFinally) {
	return;
	}
	}
	}
	?>

建议	将返回指令移到 finally 块之外。如果必须要 finally 块返回一个值，可以简单地将该返回值赋给一个本地变量，然后在 finally 块执行完毕后返回该变量。

返回目录

CWE	CWE ID 584
OWASP2017	None

返回目录

漏洞名称	**Possible Flow Control**
默认严重性	3
摘要	Y（文件） 文件第 M 行中发现可能有流程控制。这可能被攻击者用于控制程序的流程并造成意外行为。
解释	能够控制程序流程的攻击者可能会控制程序的输出并导致意外输出。 如果控制流程的决策中使用了来自用户输入的变量，就会发生这种情况。
建议	程序控制流程中不应使用用户输入。
CWE	CWE ID 691
OWASP2017	None

返回目录

漏洞名称	**Possible Variable Overwrite:Functional Scope**
默认严重性	2
摘要	此程序会调用 X（函数），它能够覆盖当前作用域中的变量，并可能为攻击者打开方便之门。此程序会调用可覆盖当前作用域中变量的函数，这可能会为攻击者打开方便之门。
解释	对于可覆盖当前作用域中已初始化的变量的函数，攻击者能够通过它影响依赖被覆盖变量的代码的执行情况。 在这种情况下，X(文件) 中第 N 行中的函数 X（函数） 可以覆盖当前范围内的变量。 示例 1：如果攻击者为下面这段 PHP 代码中的 str 提供恶意值，则对 parse_str() 的调用可能会覆盖当前作用域中的所有任意变量，包括 first。在这种情况下，如果包含 JavaScript 的恶意值覆盖 first，则该程序会很容易受到 cross-site scripting 攻击。 <?php $first="User"; ... $str = $_SERVER['QUERY_STRING']; parse_str($str); echo $first; ?>
建议	使用此方法时必须附加额外的参数，防止它覆盖变量。 - 调用 parse_str(string $encoded_string [, array &$result]) （具有第二个参数），这样可以捕获操作结果以及防止函数覆盖当前作用域中的变量。 - 调用 extract(array $var_array [, int $extract_type [, string $prefix]] ）（其中第二个参数设为 EXTR_SKIP），这样可以防止函数覆盖当前作用域中的变量。 示例 2：以下代码会在 parse_str() 中使用另一个参数来缓解 Example 1 中的漏洞。 <?php $first="User"; ... $str = $_SERVER['QUERY_STRING']; parse_str($str, $output); echo $first; ?>
CWE	CWE ID 473
OWASP2017	None

返回目录

漏洞名称	**Possible Variable Overwrite:Global Scope**
默认严重性	3
摘要	此程序会调用 X（函数），它会覆盖全局变量，并可能为攻击者打开方便之门。此程序会调用可覆盖全局变量的函数，这可能会为攻击者打开方便之门。
解释	对于可覆盖已初始化的全局变量的函数，攻击者能够通过它影响依赖被覆盖变量的代码的执行情况。 在这种情况下，X(文件) 中第 N 行中的函数 X（函数） 可以覆盖全局变量。 示例 1：如果攻击者为下面这段 PHP 代码中的 str 提供恶意值，则对 mb_parse_str() 的调用可能会覆盖所有任意变量，包括 first。在这种情况下，如果包含 JavaScript 的恶意值覆盖 first，则该程序会很容易受到 cross-site scripting 攻击。 <?php $first="User"; ... $str = $_SERVER['QUERY_STRING']; mb_parse_str($str); echo $first; ?>
建议	对于能够覆盖全局变量的函数，可通过下列方式避免它们覆盖全局变量： - 调用 mb_parse_str(string $encoded_string [, array &$result])（具有第二个参数），这样可以捕获操作结果以及防止函数覆盖全局变量。 - 调用 extract(array $var_array [, int $extract_type [, string $prefix]])（其中第二个参数设为 EXTR_SKIP），这样可以防止函数覆盖已定义的全局变量。 示例 2：以下代码会在 mb_parse_str() 中使用另一个参数来缓解 Example 1 中的漏洞。 <?php $first="User"; ... $str = $_SERVER['QUERY_STRING']; mb_parse_str($str, $output); echo $first; ?>
CWE	CWE ID 473
OWASP2017	None

返回目录

漏洞名称	**Privacy Violation**
默认严重性	3
摘要	X(文件) 文件会错误地处理第 N 行的机密信息，从而危及到用户的个人隐私，这是一种非法行为。对机密信息（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 XX(文件) 中第 N 行的 XX(函数)。 2. 数据被写到了一个外部介质，例如控制台、file system 或网络。 这种情况下，数据被传递给 YY（文件） 的第 M 行中的 YY（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。 <?php $pass = getPassword(); trigger_error($id . ":" . $pass . ":" . $type . ":" . $tstamp); ?> 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:HTTP GET**
默认严重性	4
摘要	调用 X(文件) 中第 N 行的 X（函数） 会使用 HTTP GET（而非 POST）方法将数据发送到服务器。标识调用使用 HTTP GET（而非 POST）方法将数据发送到服务器。
解释	使用 GET 方法的 HTTP 请求允许将 URL 和请求参数缓存到浏览器的 URL 缓存、中间代理和服务器日志中。这可能将敏感信息泄露给不具备相应数据权限的人。 示例 1：以下代码使用 GET HTTP 方法而非 POST 来发送 HTTP 请求。 ... $client = new Zend_Http_Client('https://www.example.com/fetchdata.php'); $client->request(Zend_Http_Client::GET); ...
建议	结合使用 HTTP POST 方法和 HTTPS 将数据发送至服务器。这会限制请求中泄露给该 URL 的数据。应在 POST 正文中发送请求参数。敏感数据不应置于 URL 请求参数中，因为它们仍会在 HTTP POST 请求中泄露。 示例 2：以下代码按照建议使用 POST HTTP 方法来发出 HTTP 请求。 ... $client = new Zend_Http_Client('https://www.example.com/fetchdata.php'); $client->request(Zend_Http_Client::POST); ...
CWE	CWE ID 359
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Process Control**
默认严重性	4
摘要	X(文件) 中的 X(方法) 函数会调用第 N 行的 X（函数），进而从一个不可信赖的数据源或在不可信赖的环境中加载库。该调用会导致应用程序以攻击者的名义执行恶意代码。从一个不可信赖的数据源或是不可信赖的环境中加载库，会导致应用程序以攻击者的名义执行恶意代码。

解释	Process control 漏洞主要表现为以下两种形式：
	— 攻击者可以篡改程序加载的库的名称：攻击者直接地控制库所使用的名称。
	— 攻击者可以篡改库加载的环境：攻击者间接控制库名称的含义。
	在这种情况下，我们着重关注第一种情况，即攻击者有可能控制加载的库的名称。这种类型的 Process Control 漏洞会在以下情况下出现：
	1. 数据从不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 XX(文件)的第 N 行的 XX(函数) 中。
	2. 数据作为代表应用程序所加载的库的一个或部分字符串使用。
	在这种情况下，库由 YY（文件）中第 M 行的 YY（函数） 加载。
	3. 通过在库中执行代码，应用程序授予攻击者在一般情况下无法获得的权限或能力。
	例 1：以下代码来自于一个具有特别权限的系统实用程序，使用系统属性 APPHOME 来确定系统的安装目录，然后用基于指定目录的相对路径加载本地库。
	...
	$home = getenv("APPHOME");
	$lib = $home + $LIBNAME;
	dl($lib);
	...
	这段代码允许攻击者加载库，并通过修改系统属性 APPHOME，指向一个包含恶意版本 LIBNAME 的不同路径，从而使用较高的应用程序权限去执行任意代码。由于程序不会验证从环境中读取的值，所以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：下列代码使用 dl() 来从名为 sockets.dll 的库加载代码，此库可从多个位置加载，具体取决于您的安装和配置。
	...
	dl("sockets");
	...
	这里的问题是，对于要加载的库来说，dl() 只会接受库的名称，而不接受库的路径。
	在搜索顺序方面，如果攻击者将一份恶意的 sockets.dll 放在高于应用程序需要加载的文件的位置，那么应用程序就会加载该恶意代码，而

返回目录

	不会选择加载最初需要的文件。由于应用程序的这一特性，它会以较高的权限运行，这就意味着攻击者的 sockets.dll 内容将以较高的权限运行，从而可能导致攻击者完全控制整个系统。
建议	不要允许用户控制由程序加载的库。若加载库的选择一定要涉及用户输入的话，通常情况下，应用程序会期望这个特定的输入是一个很小的数值集合。而不是依赖输入的安全性以及不包含任何恶意信息。因此，应用程序所使用的输入应该仅从一个预先决定的、安全的库的集合中进行选择。如果输入看上去是恶意的，则应该将即将加载的库限制在这一集合的安全数值范围之内，或由程序拒绝继续执行该操作。 攻击者可以通过篡改环境间接地控制程序加载的库。我们不应当完全信赖环境，还需采取预防措施，防止攻击者利用某些控制环境的手段进行攻击。应尽可能避免对库进行动态加载。如果无法避免动态加载，应对照一系列定义有效参数的不变量对从环境中读取的库名称和路径进行仔细的检查。 有时还可以执行其他校验，以检查环境是否已被恶意篡改。例如，如果一个配置文件为可写，程序可能会拒绝运行。如果事先已知有关要加载的库的信息，程序就会执行检测，以校验文件的有效性。如果一个库应始终归属于某一特定用户，或者被分配了一组特定的权限，则可以在加载库之前，对这些属性进行校验。 最终，程序也许无法完全防范神通广大的攻击者控制其所加载的库。因此，对输入值和环境可能执行的任何操作，都应努力鉴别并加以防范。这样做是为了尽可能地防范各种攻击。
CWE	CWE ID 114, CWE ID 494
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Race Condition:PHP Design Flaw**
默认严重性	3
摘要	PHP 配置选项 open_basedir 存在一个设计缺陷，使该选项容易发生文件访问 race condition，从而可能使攻击者绕过 file system 上的 access control 检查。
解释	如果启用 open_basedir 配置选项，该选项会试图阻止 PHP 程序对 php.ini 文件中所指定的目录结构以外的文件进行操作。尽管 open_basedir 选项从总体上有利于保证安全性，但它的实施效果却受到 race condition 的不利影响，这种状况可能会允许攻击者在某些情况下绕过该选项所定义的限制条件 [2]。在 PHP 执行访问权限检查与打开文件的两个时刻之间，存在一种 TOCTOU（检查时间，使用时间）race condition。与其他语言中 file system 的 race condition 一样，这一漏洞会导致攻击者能够将指向一个通过 access control 检查的文件的 symlink 替换为另一个本不能通过测试的文件，从而获得对受保护文件的访问权限。 这种攻击针对的漏洞大小取决于执行访问检查与打开文件这两个时刻之间的时间间隙。即使连续执行调用，现今的操作系统也无法确保在进程让出 CPU 之前将执行的代码数量。攻击者掌握了多种扩大该时间间隙的技术，以便更加容易地发起攻击，但即使是一段很短的间隙，该攻击企图也可以不断地重复，直到成功为止。
建议	确定您的程序需要访问的最小目录集，并使用 open_basedir 选项来限制程序只能访问这些目录 [3]。 为了防止在实施 open_basedir 的过程中针对 race condition 发起的攻击，请通过在 php.ini 文件中包含以下条目来禁用 PHP symlink() 函数： disable_functions = symlink 不要奢望 PHP 机制中具备该功能，而应考虑在操作系统中创建一个底层 chroot 监牢，以限制 PHP 或 Web 服务器进程可访问的 file system 区域 [5]。
CWE	CWE ID 362, CWE ID 367
OWASP2017	A6 Security Misconfiguration

返回目录

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

返回目录

漏洞名称	**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。 考虑可以净化值的多种本机 PHP 方法（例如 htmlspecialchars 和 htmlentities），不要习惯性地为 Javascript 上下文对值进行编码并忽略某些包围字符，例如撇号 (')，引号 (") 和反引号 (`)。选择净化函数之前，一定要考虑输入的输出上下文。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Reflection Injection**
默认严重性	5
摘要	X（文件） 文件第 N 行中 XXX（方法） 获取的输入影响 Y（文件） 文件第 M 行 YYY（方法） 中的反射的调用。
解释	如果反射受用户输入的直接影响，那么最好的情况也是导致意外或不稳定的行为。在最坏的情况下，它可能会允许攻击者注入和执行恶意代码，从代码中调用意外的方法或类来绕过逻辑流或操纵数据等。 反射就是一种编码方法，其中的类、方法或内置函数都是通过名称在程序中调用的。如果名称是根据用户输入动态确定的，则这些输入可能会改变代码流、调用意外或非预期的代码、有时还会注入新的恶意代码。
建议	除非绝对有需要，否则避免使用任何形式的动态代码评估，特别是要避免使用反射。 如果不需要动态评估，可以使用逻辑流程来确定要运行哪些功能 如果需要动态评估，可使用允许的代码段的白名单，以确保无法执行任意代码
CWE	CWE ID 470
OWASP2017	A1-Injection

返回目录

漏洞名称	**Reliance on Cookies in a Decision**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 获取的 cookie XX（元素）使用其值在 Y（文件） 文件第 M 行的 YYY（方法） 中制定决策，但未正确地对内容进行验证。
解释	未进行验证便将 cookie 值用于做出决定——这可能会使用户能够在未经授权的情况下篡改这些 cookie 以影响此决定的结果。具体取决于实现情况，这可能会导致绕过授权和身份验证，从而导致账户盗窃、权限提升等等。 Cookie 值通常是由服务器通过 Set-Cookie 标头或客户端代码设置的——导致这些值对最终用户来说不是透明的。但是，这些 cookie 值很容易被篡改，所以 cookie 的内容应视为潜在的恶意或污染的输入。
建议	任何与会话相关的决定都不要单独依赖用户输入——在依赖 cookie 做出决定之前，一定要根据可信任的凭证（例如会话变量）验证用户的权限和真实性。
CWE	CWE ID 784
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Reliance on DNS Lookups in a Decision**
默认严重性	3
摘要	XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 执行了反向 DNS 查询。Y（文件） 文件第 M 行中的 YY（元素） 应用程序（根据反向 DNS 查询到的主机名）做出了安全决定，尽管此主机名是不可靠的而且可以很容易地被欺骗。
解释	不通过加密证书或协议验证域所有权，仅依靠反向 DNS 记录不是一种充分的验证机制。根据注册的主机名做出安全决定可能会使外部攻击者控制应用程序流程。攻击者可能会执行受限的操作、绕过访问控制、甚至冒充用户身份、将伪造的主机名注入安全日志，以及其他可能的逻辑攻击。 应用程序根据远程 IP 地址执行反向 DNS 解析，并根据返回的主机名执行安全检查。但是，根据具体的环境，可能很容易就可以冒充 DNS 名称或导致这些 DNS 被错误地报告。如果远程服务器被攻击者控制，远程服务器就可能被配置为报告虚假主机名。此外，攻击者可以通过控制关联的 DNS 服务器、或者攻击合法的 DNS 服务器、或者使服务器 DNS 缓存中毒、或者修改到服务器的不受保护的 DNS 流量来伪造主机名。无论是哪种方式，远程攻击者都可以更改检测到的网络地址，伪造身份验证信息。
建议	不要依靠 DNS 记录、网络地址或系统主机名作为身份验证形式或做出与安全相关的决定。 不要在未进行记录验证的情况下对未受保护的协议执行反向 DNS 解析。 实施合适的身份验证机制，例如密码、加密证书或公钥数字签名。 考虑使用提议的协议扩展来加密保护 DNS，例如 DNSSEC（但要注意有限的支持和其他缺点）。
CWE	CWE ID 350
OWASP2017	None

返回目录

漏洞名称	**Remote File Inclusion**
默认严重性	5
摘要	应用程序在 Y（文件） 文件第 M 行使用 YY（元素） 载入了外部库或源代码文件。攻击者可能会利用此漏洞，导致应用程序加载任意代码。 请注意，此库是根据 X（文件） 文件第 N 行上不可信任的用户输入 XX（元素） 动态选择的。攻击者可以（甚至从远程服务器）影响此输入，使应用程序执行任意代码。
解释	如果攻击者可以选择库的名称或应用程序加载的代码文件的位置，攻击者就可能让应用程序执行任意代码。这样攻击者便能有效地控制应用程序运行的代码。 这包括执行系统命令，甚至可以完全控制服务器。特别是，此漏洞甚至可使攻击者检索和加载完全在攻击者控制下的远程库。 应用程序使用不可信任的数据指定库或代码文件，未进行适当的净化。这会导致应用程序加载指定的任意代码。然后执行加载的代码。 对目标库的引用是通过用户输入接收的，使外部攻击者能够控制应用程序下载并执行远程服务器上托管的远程代码文件。
建议	不要动态加载代码库，尤其是不要根据用户输入加载代码库。 如果需要使用不可信任的数据来选择要加载的库，请验证所选库名称是否与预定义的已列入白名单的库名称集匹配。也可使用值作为标识符从已列入白名单的库中选择。 对用于加载或处理库或代码文件的不可信任的数据进行验证。 特别是，切勿载入远程服务器的任意代码。 一般来讲，要尽量通过远程代码加载本地代码文件。 切勿使用匿名相对路径添加文件，而要使用绝对路径或根相对路径。 将 allow_url_fopen 设置为 false，以限制从远程位置添加文件的功能。
CWE	CWE ID 98
OWASP2017	A1-Injection

返回目录

漏洞名称	**Resource Injection**
默认严重性	3
摘要	攻击者可以控制 X(文件) 中第 N 行的 X（函数） 的资源标识符参数，借此访问或修改其他受保护的系统资源。使用用户输入控制资源标识符，借此攻击者可以访问或修改其他受保护的系统资源。
解释	当满足以下两个条件时，就会发生 resource injection： 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 XX(文件) 中第 N 行的 XX(函数) 进入程序的值，这一数值可以通过 YY（文件） 中第 M 行的 YY（函数） 访问系统资源。 注意：如果资源注入涉及存储在文件系统中的资源，则可以将其报告为名为路径篡改的不同类别。有关这一漏洞的详细信息，请参见 path manipulation 的描述。 示例：下列代码使用从 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。 <?php $host=$_GET['host']; $dbconn = pg_connect("host=$host port=1234 dbname=ticketdb"); ... $result = pg_prepare($dbconn, "my_query", 'SELECT * FROM pricelist WHERE name = $1'); $result = pg_execute($dbconn, "my_query", array("ticket")); ?> 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

返回目录

	其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	CWE ID 99
OWASP2017	A5 Broken Access Control

返回目录

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

解释	当攻击者可以影响应用程序服务器建立的网络连接时，将会发生 Server-Side Request Forgery。网络连接源自于应用程序服务器内部 IP 地址，因此攻击者将可以使用此连接来避开网络控制，并扫描或攻击没有以其他方式暴露的内部资源。
	在这种情况下，X(文件)的第 N 行调用 X（函数）。
	示例：在下列示例中，攻击者将能够控制服务器连接至的 URL。
	$url = $_GET['url'];
	$c = curl_init();
	curl_setopt($c, CURLOPT_POST, 0);
	curl_setopt($c,CURLOPT_URL,$url);
	$response=curl_exec($c);
	curl_close($c);
	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

返回目录

	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。 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	**Session Fixation**
默认严重性	3
摘要	函数 X(方法) 在未释放当前会话标识符的情况下验证用户，会给攻击者窃取验证会话的机会。在未释放当前会话标识符的情况下验证用户会给攻击者窃取验证会话的机会。
解释	Session Fixation 漏洞会在以下情况中出现： 1.当 Web 应用程序验证某一用户时，没有首先释放当前会话，而是继续使用已经与该用户关联的会话。 2.攻击者能够夺取已知的用户会话标识符，因此，一旦该用户进行验证，攻击者便可以访问经过验证的会话。 在通常情况下，攻击者会利用 Session Fixation 漏洞在 Web 应用程序中创建一个新会话，并且记录与之关联的会话标识符。然后，攻击者会设法使受害者在服务器中的验证失败，从而通过当前会话来访问用户帐号。 在这种情况下，在 X(文件) 的第 N 行上，session.use_strict_mode 属性设置为 0 或 off。 示例 1：以下代码将对会话 Cookie 禁用 use_strict_mode 属性。 ini_set("session.use_strict_mode", "0");
建议	为了防止 Session Fixation 攻击，基于 Web 的应用程序必须在验证用户的同时生成新的会话标识符。许多应用程序服务器都提供独立的授权管理机制和会话管理机制，这大大增加了 session fixation 攻击的难度。 在创建会话 Cookie 时启用 use_strict_mode 属性。做法是使用 ini_set 在 PHP 配置选项中将 session.use_strict_mode 属性设置为 1 或 on。 示例 2：以下代码将对会话 Cookie 启用 use_strict_mode 属性。 ini_set("session.use_strict_mode", "1"); 唯一有效的解决方案是：通过执行专有代码来进行身份验证，并确保在处理登录请求前已释放现有会话。只要改变了关联的会话标识符，任何存放在会话中的用户信息都可以安全地移动到新会话中。切记操作顺序十分重要：在处理登录请求前，必须释放当前会话。如果等到用户登录之后再释放会话，攻击者就可以在身份验证后的一段时间内知晓会话标识符，并与合法用户发生争用情况。
CWE	CWE ID 384
OWASP2017	A2 Broken Authentication

返回目录

漏洞名称	**Setting Manipulation**
默认严重性	3
摘要	攻击者可以控制 X(文件) 中第 N 行的 X（函数） 的一个参数，从而导致服务中断或意外的应用程序行为。允许对系统设置进行外部控制可以导致服务中断或意外的应用程序行为。
解释	当攻击者能够通过控制某些值来监控系统的行为、管理特定的资源、或在某个方面影响应用程序的功能时，即表示发生了 Setting Manipulation 漏洞。 在这种情况下，潜在的恶意数据会通过 XX(文件) 中第 N 行的 XX(函数) 进入程序，并流向 YY（文件） 中第 M 行的 YY（函数）。 由于 Setting Manipulation 漏洞影响到许多功能，因此，对它的任何说明都必然是不完整的。与其在 Setting Manipulation 这一类中寻找各个功能之间的紧密关系，不如往后退一步，考虑有哪些系统数值类型不能由攻击者来控制。 示例 1：以下 PHP 代码片段从 HTTP 请求中读取参数，并将该参数设置为数据库连接的当前目录。 <?php ... $table_name=$_GET['catalog']; $retrieved_array = pg_copy_to($db_connection, $table_name); ... ?> 在该示例中，攻击者通过提交一个不存在的目录名，或者连接到数据库中未授权的部分，从而可以引出一个错误。 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
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 XX(文件)的第 N 行的 XX(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 YY（文件）的第 M 行中的 YY（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 = $_SESSION['userName']; $itemName = $_POST['itemName']; $query = "SELECT * FROM items WHERE owner = '$userName' AND itemname = '$itemName';"; $result = mysql_query($query); ... 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。 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

返回目录

	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。

返回目录

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。 当连接到 MySQL 时，可以将前面的例子改写为使用参数化 SQL 指令（而不是联结用户输入的字符串），如下所示： ... $mysqli = new mysqli($host,$dbuser, $dbpass, $db); $userName = $_SESSION['userName']; $itemName = $_POST['itemName']; $query = "SELECT * FROM items WHERE owner = ? AND itemname = ?"; $stmt = $mysqli->prepare($query); $stmt->bind_param('ss',$username,$itemName); $stmt->execute(); ... MySQL Improved 扩展 (mysqli) 适用于 MySQL 中的 PHP5 用户。依赖于不同数据库的代码应该检查相似的扩展名。 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	**SQL Injection:Poor Validation**
默认严重性	2
摘要	X(文件) 的第 N 行会调用通过不可信来源的输入构建的 SQL 查询，并使用 HTML、XML 或其他并非始终足以防止恶意字符被插入该查询的编码。如果依靠 HTML、XML 及其他类型的编码来验证不受信任的输入，攻击者也许就能够修改语句的含义或者执行任意 SQL 命令。
解释	使用 mysql_real_escape_string() 等编码函数可避免一部分 SQL Injection 漏洞，但不能完全避免。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 SQL Injection 攻击，并且可能允许攻击者修改语句的含义或者执行任意 SQL 命令。由于在动态解释代码的给定部分中不可能总是静态确定输入显示的位置，因此 Fortify 安全编码规则包可能会将经过验证的动态 SQL 数据显示为“SQL Injection: Poor Validation”问题，即使验证可能足以防止在该上下文中出现 SQL Injection 攻击时也是如此。 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 XX(文件)的第 N 行的 XX(函数) 中。 2. 数据用于动态地构造 SQL 查询。 在这种情况下，数据将传递到 YY（文件）的第 M 行中的 YY（函数）。 示例 1： 以下示例演示数据库配置如何影响 mysqli_real_escape_string() 的行为。 将 SQL 模式设置为“NO_BACKSLASH_ESCAPES”时，反斜杠字符会被视为正常字符，而不是转移字符[5]。 由于 mysqli_real_escape_string() 将此情况考虑在内，因此鉴于数据库配置，当 '' 不再转义为 \'' 时，以下查询易受到 SQL Injection 的攻击。 mysqli_query($mysqli, 'SET SQL_MODE="NO_BACKSLASH_ESCAPES"'); ... $userName = mysqli_real_escape_string($mysqli, $_POST['userName']); $pass = mysqli_real_escape_string($mysqli, $_POST['pass']); $query = 'SELECT * FROM users WHERE userName="' . $userName . '"AND pass="' . $pass. '";'; $result = mysqli_query($mysqli, $query); ... 如果攻击者将 password 字段留空并为 userName 输入 '' OR 1=1;-- ，则不对引号进行转义，最后的查询如下所示： SELECT * FROM users WHERE userName = "" OR 1=1;

返回目录

	-- "AND pass=""; 由于 OR 1=1 会导致 where 子句的值始终为 true 且双连字符会导致剩余语句被视为注释，因此该查询在逻辑上就等同于一个更为简单的查询： SELECT * FROM users; 避免 SQL injection 攻击的传统方法之一是，作为一个输入验证问题来处理，只接受列在安全值允许列表中的字符，或者识别并避免列在潜在恶意值列表（拒绝列表）中的字符。检验允许列表是一种非常有效的方法，它可以强制执行严格的输入验证规则，但是参数化的 SQL 语句所需的维护工作更少，而且能提供更好的安全保障。而对于通常采用的执行拒绝列表方式，由于总是存在一些小漏洞，所以并不能有效地防止 SQL Injection 攻击。例如，攻击者可以： - 将没有被黑名单引用的字段作为目标 - 寻找方法以绕过某些需要转义的元字符 - 使用存储过程隐藏注入的元字符 手动转义 SQL 查询输入中的字符有一定的帮助，但是并不能完全保护您的应用程序免受 SQL Injection 攻击。 防范 SQL Injection 攻击的另外一种常用解决方法是使用存储过程。虽然存储过程可以阻止某些类型的 SQL Injection 攻击，但是对于绝大多数攻击仍无能为力。 存储过程有助于避免 SQL Injection 攻击的常用方式是限制可传入存储过程参数的语句类型。 但是，有许多方法都可以绕过这一限制，许多危险的语句仍可以传入存储过程。 所以再次强调，存储过程在某些情况下可以避免一些漏洞，但是并不能完全保护您的应用程序免受 SQL Injection 攻击。
建议	造成 SQL Injection 漏洞的根本原因在于攻击者可以改变 SQL 查询的上下文，使程序员原本要作为数据解释的数值却被解释为命令了。 构造 SQL 查询后，程序员知道哪些字符应作为命令解释，哪些字符应作为数据解释。 参数化 SQL 语句可以防止直接窜改上下文，避免几乎所有的 SQL Injection 攻击。 参数化 SQL 语句是用常规的 SQL 字符串构造的，但是当需要添加用户提供的数据时，它们就需要使用捆绑参数，这些捆绑参数是一些占位符，用来存放随后插入的数据。 换言之，捆绑参数可以使程序员明确告知数据库哪些字符应被视为命令，哪些字符应被视为数据。 这样，当程序准备执行某个语句时，它可以明确告知数据库，每个捆绑参数所使用的运行时值，而不会存在数据被篡改为命令的风险。 示例 2： 连接到 MySQL 后，应始终使用参数化的 SQL 语句，而非用户提供的动态连接输入，如下所示： ... $mysqli = new mysqli($host,$dbuser, $dbpass, $db); $userName = $_SESSION['userName']; $itemName = $_POST['itemName']; $query = "SELECT * FROM items WHERE owner = ? AND itemname = ?"; $stmt = $mysqli->prepare($query);

返回目录

	$stmt->bind_param('ss',$username,$itemName); $stmt->execute(); ... MySQL Improved 扩展 (mysqli) 适用于 MySQL 中的 PHP5 用户。 依赖于不同数据库的代码应该检查相似的扩展名。 更加复杂的情况常常出现在报表生成代码中，因为这时需要通过用户输入来改变 SQL 语句的结构，比如在 WHERE 子句中添加动态约束条件。 不要因为这一需求，就无条件地接受连续的用户输入，从而创建查询字符串。 当必须要根据用户输入来改变命令结构时，可以使用间接的方法来防止 SQL Injection 攻击：创建一个合法的字符串集合，使其对应于可能要加入到 SQL 语句中的不同元素。 在构造语句时，可使用来自用户的输入，以便从应用程序控制的值集合中进行选择。
CWE	CWE ID 89
OWASP2017	A1 Injection

返回目录

漏洞名称	**SSL Verification Bypass**
默认严重性	4
摘要	Y（文件） 模块在 YYY（方法） 中发送和接收 HTTPS 请求。第 M 行的 YY（元素） 参数有效地禁用了 SSL 证书信任链验证。
解释	如果 SSL/TLS 库被配置为禁用证书信任链验证，这可能使应用程序接受伪造证书。这使攻击者能够拦截客户端请求、伪造服务器证书，并主动执行中间人攻击，甚至可以通过 HTTPS 执行。这样，攻击者对请求和响应有完全的访问权限，可以读取任何秘密信息并修改任何敏感数据，包括用户凭证。 应用程序显式将 HTTPS 请求的参数设置为禁用证书信任链验证。如果未对证书签署方一路验证到可信的证书颁发机构，就可能颁发伪造的证书，并被应用程序识别。因为已在系统受信任的根证书存储中配置了签名证书，所以只要使用有任意用户名或服务器名的自签名证书，应用程序就会信任该证书，而不强制执行验证。
建议	通用指南： 正确地实施所有必要的检查，以确保加密通信中涉及的实体身份正确。 正确地配置 HTTPS 的所有参数。 具体建议： 不要禁用证书验证。 显式执行验证，将"rejectUnauthorized" 设置为 True。
CWE	CWE ID 599
OWASP2017	A3-Sensitive Data Exposure

返回目录

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

返回目录

	根据最小权限原则，如果可行，应独立出来所有的动态执行，以使用一个单独的专用用户帐户来运行，该帐户仅对特定操作和动态执行所涉及的文件有权限。
CWE	CWE ID 94
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Command Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 方法使用 Y（文件） 文件第 M 行的 YY（元素） 调用了 OS (shell) 命令，这使用了不可信任的字符串让命令执行。 这使攻击者能够注入任意命令，可以发动命令注入攻击。 攻击者可能将执行的命令注入数据库，然后由应用程序在 X（文件）文件第 N 行的 XXX（方法） 方法中使用 XX（元素） 检索到。
解释	攻击者可以在应用程序服务器主机上运行任意系统级 OS 命令。根据应用程序的操作系统权限，这可能包括: 文件操作（读取/创建/修改/删除) 打开到攻击者服务器的网络连接 启动和停止系统服务 修改运行的应用程序 完全控制服务器 应用程序运行 OS 系统级命令而不是通过应用程序代码完成其任务。该命令包括不可信任的数据，这些数据可能受到攻击者的控制。此不可信任的字符串中可能包含攻击者设计的恶意系统级命令，使攻击者就像直接在应用程序服务器上运行一样执行该命令。 在这种情况下，应用程序读取数据库中的命令，并将其作为字符串传递给操作系统。攻击者可能提前将恶意负载加载到数据库字段中，从而导致应用程序加载该命令。然后，该未经验证的数据由操作系统作为系统命令执行，使用与应用程序相同的系统权限运行。
建议	重构代码以避免直接执行 shell 命令。可以使用平台提供的 API 或库调用代替。 如果必须执行命令，则仅执行不包含用户控制的动态数据的静态命令。 无论来源如何，一概验证所有输入。验证应基于白名单：仅接受适合指定格式的数据，而不是排除不符合要求的模式（黑名单）。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: 数据类型 大小 范围 格式 预期值 作为深度防御措施，为了尽量减少损失，可将应用程序配置为使用没有不必要的 OS 权限的受限用户帐户运行。 如果可能，应根据最小权限原则，隔离出所有 OS 命令，以使用单独的专用用户帐户，且该帐户只有应用程序使用的特定命令和文件的权限。

返回目录

	如果确实需要使用用户输入调用系统命令或执行外部程序，请不要未经净化便将用户输入与命令拼接。视适用情况，使用专为此目的设计的方法，例如 escapeshellarg 和 escapeshellcmd。
CWE	CWE ID 77
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored File Inclusion**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行使用 YY（元素） 载入了外部库或源代码文件。攻击者可能会利用此漏洞，导致应用程序加载任意代码。
解释	如果攻击者可以选择库的名称或应用程序加载的代码文件的位置，攻击者就可能让应用程序执行任意代码。这样攻击者便能有效地控制应用程序运行的代码。 应用程序使用不可信任的数据指定库或代码文件，未进行适当的净化。这会导致应用程序加载指定的任意代码。然后执行加载的代码。
建议	不要动态加载代码库，尤其是不要根据用户输入加载代码库。 如果需要使用不可信任的数据来选择要加载的库，请验证所选库名称是否与预定义的已列入白名单的库名称集匹配。也可使用值作为标识符从已列入白名单的库中选择。 对用于加载或处理库或代码文件的不可信任的数据进行验证。
CWE	CWE ID 98
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored File Manipulation**
默认严重性	4
摘要	通过 X（文件） 文件第 N 行中的 XXX（方法） 获取的输入被 Y（文件） 文件第 M 行中 YYY（方法） 用于确定要写入的文件位置，这可能会使攻击者能够更改或损坏文件的内容，或创建全新的文件。
解释	如果攻击者可以影响攻击者选择的任意文件，攻击者就可能覆写或损坏敏感文件，从而可能导致拒绝服务攻击。如果攻击者还能选择要写入的内容，攻击者就可能将代码注入任意文件，从而可能导致执行恶意代码。 用户提供的输入被用于确定要写入哪个文件，这可能会使用户能够影响或操纵任意文件的内容。
建议	考虑为要写入的文件使用静态解决方案，例如经过验证的可写的文件列表，或者使用其他文件存储解决方案，如数据库。如果确实有需要，可通过正确地净化用户提供的输入来设置文件名以将写入目标限制为单个文件夹，并在程序中设置目标文件夹。可考虑根据应用程序代码的业务需求增加一种检查来验证文件是否存在。
CWE	CWE ID 552
OWASP2017	A1-Injection

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 Reflection Injection**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XXX（方法） 获取的输入影响 Y（文件） 文件第 M 行 YYY（方法） 中的反射的调用。
解释	如果反射受用户输入的直接影响，那么最好的情况也是导致意外或不稳定的行为。在最坏的情况下，它可能会允许攻击者注入和执行恶意代码，从代码中调用意外的方法或类来绕过逻辑流或操纵数据等。 反射就是一种编码方法，其中的类、方法或内置函数都是通过名称在程序中调用的。如果名称是根据用户输入动态确定的，则这些输入可能会改变代码流、调用意外或非预期的代码、有时还会注入新的恶意代码。
建议	除非绝对有需要，否则避免使用任何形式的动态代码评估，特别是要避免使用反射。 如果不需要动态评估，可以使用逻辑流程来确定要运行哪些功能 如果需要动态评估，可使用允许的代码段的白名单，以确保无法执行任意代码
CWE	CWE ID 470
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Remote File Inclusion**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行使用 YY（元素） 载入了外部库或源代码文件。攻击者可能会利用此漏洞，导致应用程序加载任意代码。
解释	如果攻击者可以选择库的名称或应用程序加载的代码文件的位置，攻击者就可能让应用程序执行任意代码。这样攻击者便能有效地控制应用程序运行的代码。 应用程序使用不可信任的数据指定库或代码文件，未进行适当的净化。这会导致应用程序加载指定的任意代码。然后执行加载的代码。
建议	不要动态加载代码库，尤其是不要根据用户输入加载代码库。 如果需要使用不可信任的数据来选择要加载的库，请验证所选库名称是否与预定义的已列入白名单的库名称集匹配。也可使用值作为标识符从已列入白名单的库中选择。 对用于加载或处理库或代码文件的不可信任的数据进行验证。
CWE	CWE ID 98
OWASP2017	A1-Injection

返回目录

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

返回目录

漏洞名称	**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。 考虑可以净化值的多种本机 PHP 方法（例如 htmlspecialchars 和 htmlentities），不要习惯性地为 Javascript 上下文对值进行编码并忽略某些包围字符，例如撇号 (')，引号 (") 和反引号 (`)。选择净化函数之前，一定要考虑输入的输出上下文。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**System Information Leak**
默认严重性	3
摘要	对于 X(文件)，在调用第 N 行上的 X（函数） 过程中，程序可能会显示系统数据或调试信息。由 X（函数） 揭示的信息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输出流或者日志功能流出程序时，就会发生信息泄漏。 在这种情况下，XXX（函数） 会产生系统数据或调试信息，而 X(文件) 第 N 行的 YYY（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标准错误流： <?php ... echo "Server error! Printing the backtrace"; debug_print_backtrace(); ... ?> 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 497
OWASP2017	None

返回目录

漏洞名称	**System Information Leak:External**
默认严重性	3
摘要	对于 X(文件)，在调用第 N 行上的 X（函数） 过程中，程序可能会显示系统数据或调试信息。由 X（函数） 揭示的信息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，XXX（函数） 会产生系统数据或调试信息，而 X(文件) 第 N 行的 YYY（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： <?php ... echo "Server error! Printing the backtrace"; debug_print_backtrace(); ... ?> 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
摘要	对于 X(文件)，在调用第 N 行上的 X（函数） 过程中，程序可能会显示系统数据或调试信息。由 X（函数） 揭示的信息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	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，XXX（函数） 会产生系统数据或调试信息，而 X(文件) 第 N 行的 YYY（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标准错误流： <?php ... echo "Server error! Printing the backtrace"; debug_print_backtrace(); ... ?> 根据这一系统配置，该信息可能会转储到控制台、写入日志文件或公开给用户。在某些情况下，该错误消息会告诉攻击者该系统易遭受的确切攻击类型。例如，数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他错误消息可以揭示有关该系统的更多间接线索。在 Example 1 中，泄露的信息可能会暗示有关操作系统类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施的信息。
建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。调试踪迹有时可能出现在不明显的位置（例如，嵌入在错误页 HTML 的注释中）。 即便是并未揭示栈踪迹或数据库转储的简短错误消息，也有可能帮助攻击者发起攻击。例如，"Access Denied"（拒绝访问）消息可以揭示系统中存在一个文件或用户。
CWE	CWE ID 497
OWASP2017	None

返回目录

漏洞名称	**System Information Leak:PHP Errors**
默认严重性	3
摘要	泄漏数据或调试信息将会帮助攻击者了解系统和制定攻击计划。
解释	如果启用 display_errors 选项，Web 中会显示发生的错误，这些错误会向攻击者明示系统的薄弱之处。例如，一个数据库错误消息可以揭示应用程序容易受到 SQL Injection 攻击。其他的错误消息可以揭示有关该系统的更多间接线索。
建议	在 PHP 5.2.4 版及更高版本中，将 display_errors 选项设置为 stderr，以便将错误输出到 stderr 而非 stdout 中。 以下条目来自 php.ini 文件，用于设置 disable_errors 选项以便将错误输出到 stderr 中： disable_errors = 'stderr' 在早期的 PHP 版本中，可完全禁用 display_errors 选项。 以下条目来自 php.ini 文件，用于禁用 disable_errors 选项： disable_errors = 'off' 如果禁用 display_errors 选项，可使用 log_errors 和 error_log 配置选项将错误重定向到系统日志文件中，这控制着 PHP 如何在日志中记录不显示到系统输出流中的错误。 以下条目来自 php.ini 文件，这些条目会使 PHP 将错误记录到由 error_file 选项指定的日志文件中。 log_errors = 'on' error_file = '/hwxx/daxx/uwnetid/phperrors.log' 也可以通过在 Apache httpd.conf 文件中包含以下条目来设置 disable_errors、log_errors 和 error_file 选项： php_flag disable_errors off php_flag log_errors on php_flag error_file /hwxx/daxx/uwnetid/phperrors.log
CWE	CWE ID 209, CWE ID 215
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**System Information Leak:PHP Version**
默认严重性	3
摘要	泄漏详细系统信息将会帮助攻击者了解系统和制定攻击计划。
解释	如果启用 expose_php 选项，那么由 PHP 解释器生成的每个响应都会包含主机系统上所安装的 PHP 版本。了解到远程服务器上运行的 PHP 版本后，攻击者就能针对系统枚举已知的盗取手段，从而大大增加成功发动攻击的机会。
建议	如果利用 php.ini 文件中的以下条目禁用 expose_php 选项，可阻止攻击者了解关于系统的详细信息： expose_php = 'off' 也可以通过在 Apache httpd.conf 文件中包含以下条目来禁用 expose_php 选项： php_flag expose_php off
CWE	CWE ID 497
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Trust Boundary Violation**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从元素 XX（元素） 获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终保存在 Y（文件） 文件第 M 行的 YYY（方法）中。这构成了信任边界违规。
解释	从 Session 变量读取的代码可能将其作为服务器侧变量来信任，但这些代码可能已被用户输入污染。这可能导致用于验证或授权用户的参数被篡改。此外，受污染的 Session 变量会给应用程序提供额外的攻击面——如果不可信任的数据污染了一个 Session 变量，而然后该 Session 变量未经净化即作为可信任变量在其他地方使用，就可能导致跨站点脚本攻击、SQL 注入等攻击。 服务器侧 Session 变量或对象是分配到与特定用户关联的特定会话的值。通常这些内容保存着与该用户会话相关的值，例如特定标识符、用户类型、授权、验证信息等。因此，通常会认为 Session 对象的内容是可信任的，因为用户通常无法自己设置这些值。 应用程序将用户输入（不可信任的数据）放在了视为可信任位置的服务器侧 Session 对象中。这可能导致开发人员将不可信任的数据作为可信任的数据来处理。
建议	验证并净化所有来源的所有输入。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 不要将不可信任的用户输入与可信任的数据混在一起。
CWE	CWE ID 501
OWASP2017	A5-Broken Access Control

返回目录

漏洞名称	**Unsafe Reflection**
默认严重性	3
摘要	攻击者可以控制 X(文件) 中第 N 行的反射方法 X（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 XX(文件)的第 N 行进入 XX(函数) 的程序。传递到 YY（文件） 的第 M 行中的 YY（函数） 的反射 API。
	示例：应用程序使用反射 API 的一个共同理由是实现自己的命令发送器。以下例子显示了一个没有使用反射的命令发送器:
	$ctl = $_GET["ctl"];
	$ao = null;
	if (ctl->equals("Add")) {
	$ao = new AddCommand();
	} else if ($ctl.equals("Modify")) {
	$ao = new ModifyCommand();
	} else {
	throw new UnknownActionError();
	}
	$ao->doAction(request);
	程序员可能会修改这段代码，以便按照如下情况使用反射:
	$ctl = $_GET["ctl"];
	$args = $_GET["args"];
	$cmdClass = new ReflectionClass(ctl . "Command");
	$ao = $cmdClass->newInstance($args);
	$ao->doAction(request);
	乍一看，这种修改似乎具有许多优点。代码的行数比原先少了，if/else 代码段也完全删除了，而且还可以在不改变命令发送器的情况下增加新的命令类型。

返回目录

	然而，这种修改允许攻击者将任意实现了 Worker 接口的对象实例化。如果命令发送器仍对 access control 负责，那么只要程序员创建实现 Worker 接口的新类，就务必要修改发送器的 access control 代码。如果未修改 access control 代码，那么一些 Worker 类就没有任何 access control 权限。 解决这种 access control 问题的一种方法是让 Worker 对象负责执行 access control 检查。下面是一段重新修改的代码示例： $ctl = $_GET["ctl"]; $args = $_GET["args"]; $cmdClass = new ReflectionClass(ctl . "Command"); $ao = $cmdClass->newInstance($args); $ao->checkAccessControl(request); ao->doAction(request); 虽然有所改进，但它鼓励了采用分散化的手段进行 access control，这使程序员在 access control 上更加容易犯错误。 这段代码还突出反映了通过反射构建命令发送器所引发的安全问题。攻击者能为任意种类的对象调用默认构造函数。实际上，攻击者甚至不会局限于使用实现了 Worker 接口的对象；系统中所有对象的默认构造函数都可以调用。如果对象没有实现 Worker 接口，则会在分配到 $ao 前抛出 ClassCastException。但如果构造函数执行了一些有利于攻击者的操作，则说明已经造成损害。对于简单的应用程序来说，这种情况的影响并不大，但是对于日趋复杂的大型应用程序来说，攻击者利用构造函数发动攻击并非没有可能。
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	**Unsafe Use Of Target Blank**
默认严重性	3
摘要	使用 X（文件） 文件第 N 行的 XX（元素） 但未正确地设置 "rel" 特性，或取消新窗口与其父窗口的关联，这是不安全的新窗口打开方式。
解释	毫无戒备的用户可能会点击攻击者准备的有漏洞的外观合法的链接，从而打开恶意页面。打开的新页面可以将原始页面重定向到另一个恶意页面，并滥用用户的信任进行非常有迷惑性的网络钓鱼攻击。 使用 HTML 元素（带任意值的 "target" 特性）或使用 JavaScript 中的 window.open() 打开新页面时，新页面可以通过 window.opener 对象访问原始页面。这可能导致被重定向到恶意的钓鱼页面。
建议	对于 HTML: 除非有需要，否则不要为用户创建的链接（用任意值）设置 "target" 特性。 如果有需要，使用 "target" 特性时，要同时将 "rel" 特性设置为 "noopener noreferrer": 对于 Chrome 和 Opera，则为 "noopener"， 对于 Firefox 和旧浏览器，则为 "noreferrer" Safari 无类似解决方案 对于 JavaScript: 使用 "var newWindow = window.open()" 调用不可信任的新窗口时，先设置 "newWindow.opener=null"，再将 "newWindow.location" 设置为可能不可信任的站点，这样在新窗口打开新站点时，它无法访问其原始的 "opener" 特性
CWE	CWE ID 1022
OWASP2017	None

返回目录

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

返回目录

漏洞名称	**Use of Hard coded Cryptographic Key**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XX（元素） 变量分配了硬编码的文本值。此静态值用作加密密钥。
解释	源代码中的静态、不可更改的加密密钥会被能访问源代码或应用程序二进制文件的攻击者窃取。攻击者拥有加密密钥后，即可用其访问任何已加密的秘密数据，从而使数据失去保密性。此外，被窃取后，还无法更换加密密钥。请注意，如果这是可以多次安装的产品，则加密密钥将总是相同，使攻击者能够以相同的方式破解所有实例。 应用程序代码使用一个加密密钥来加密和解密敏感数据。尽管此加密密钥遵循了随机创建且要保密的重要法则，但应用程序会在源代码中以纯文本形式嵌入一个静态密钥。 攻击者可能获得对源代码的访问权——无论是在源代码控制系统、开发人员工作站中、还是在服务器文件系统或产品二进制文件本身中。攻击者获得源代码的访问权限后，即可轻松检索纯文本的加密密钥，并用它解密应用程序保护的敏感数据。
建议	通用指南： 请不要使用纯文本格式存储任何敏感信息，例如加密密钥。 切勿硬编码应用程序源代码中的加密密钥。 使用合适的密钥管理，包括动态生成随机密钥、保护密钥、并根据需要更换密钥。 具体建议： 删除应用程序源代码中硬编码的加密密钥。相反，从外部受保护的存储中检索密钥。
CWE	CWE ID 321
OWASP2017	A3-Sensitive Data Exposure

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	3
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
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 X（函数） 的调用中发现空 salt。 示例 1：下列代码使用了空 salt： ... $hash = hash_pbkdf2('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。 示例 2：下列代码使用由系统管理员配置的 salt 变量： ... $dk = hsdh_pbkdf2('sha256', $password, $salt, 100000); ...
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
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 X（函数） 的调用中发现硬编码 salt。 示例 1：下列代码使用了 hardcoded salt： ... $hash = hash_pbkdf2('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。 示例 2：下列代码使用由系统管理员配置的 salt 变量： ... $hash = hash_pbkdf2('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	2
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 X（函数） 的调用中发现硬编码 salt。 例 1：下列代码使用了 hardcoded salt： … crypt($password, '2!@$(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： … crypt($password, $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
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 X（函数） 的调用指定的重复计数过低。 示例 1：以下代码使用 50 的迭代计数： ... $hash = hash_pbkdf2('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。 示例 2：以下代码使用 100,000 的迭代计数： ... $hash = hash_pbkdf2('sha256', $password, $salt, 100000); ...
CWE	CWE ID 916
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Predictable Salt**
默认严重性	3
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。 示例 1：以下代码重新使用密码作为 salt： function register(){ $password = $_GET['password']; $username = $_GET['username']; $hash = hash_pbkdf2('sha256', $password, $password, 100000); ...

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。
	示例 2：以下代码使用 random_bytes 创建足够随机的 salt 值：
	...
	function get_random_salt(){
	return random_bytes(64);
	}
	...
	function register(){
	$password = $_GET['password'];
	$username = $_GET['username'];
	$salt = get_random_salt();
	$hash = hash_pbkdf2('sha256', $password, $salt, 100000);
	...

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
摘要	X(文件) 中的 X(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 XX(文件)的第 N 行的 XX(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 (PBKDF) 中的 salt 使用。 在这种情况下，在 YY（文件）中第 M 行的 YY（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 PBKDF 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。 示例 1：以下代码使用用户控制 salt： function register(){ $password = $_GET['password']; $username = $_GET['username']; $salt = getenv('SALT'); $hash = hash_pbkdf2('sha256', $password, $salt, 100000); ... 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 Encryption**
默认严重性	4
摘要	调用 X(文件) 中第 N 行的 X（函数） 会使用弱加密算法，无法保证敏感数据的保密性。识别调用会使用无法保证敏感数据的保密性的弱加密算法。
解释	过时的加密算法（如 DES）无法再为敏感数据的使用提供足够的保护。加密算法依赖于密钥大小，这是确保加密强度的主要方法之一。加密强度通常以生成有效密钥所需的时间和计算能力来衡量。计算能力的提高使得在合理的时间内获得较小的加密密钥成为可能。例如，在二十世纪七十年代首次开发出 DES 算法时，要破解在该算法中使用的 56 位密钥将面临巨大的计算障碍，但今天，使用常用设备能在不到一天的时间内破解 DES。
建议	使用密钥较大的强加密算法来保护敏感数据。DES 的一个强大替代方案是 AES（高级加密标准，以前称为 Rijndael）。在选择算法之前，首先要确定您的组织是否已针对特定算法和实施进行了标准化。
CWE	CWE ID 327
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Inadequate RSA Padding**
默认严重性	4
摘要	X(文件) 中的 X(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：
	function encrypt($input, $key) {
	$output='';
	openssl_public_encrypt($input, $output, $key, OPENSSL_NO_PADDING);
	return $output;
	}

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：
	function encrypt($input, $key){
	$output='';
	openssl_public_encrypt($input, $output, $key, OPENSSL_PKCS1_OAEP_PADDING);
	return $output;
	}

CWE	CWE ID 780
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insufficient Key Size**
默认严重性	3
摘要	X(文件) 中的 X(方法) 方法使用了密钥长度不够的加密算法，导致加密数据容易受到强力攻击。另外，当使用的密钥长度不够时，强大的加密算法便容易受到强力攻击。
解释	当前的密码指南建议，RSA 算法使用的密钥长度至少应为 2048 位。但是，计算能力和因子分解技术方面的持续进步 [1] 意味着未来将不可避免地需要提高建议的密钥大小。 示例 1：以下代码可生成 1024 位 RSA 加密密钥。 ... $keysize = 1024; $options = array('private_key_bits' => $keysize, 'private_key_type' => OPENSSL_KEYTYPE_RSA); $res = openssl_pkey_new($options); ... 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。 示例 2：以下代码可生成 2048 位 RSA 加密密钥。 ... $keysize = 2048; $options = array('private_key_bits' => $keysize, 'private_key_type' => OPENSSL_KEYTYPE_RSA); $res = openssl_pkey_new($options); ... 同样，如果使用对称加密，请确保特定密钥的长度至少为 128 位（适用于 AES）和 168 位（适用于 Triple DES）。
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:User-Controlled Key Size**
默认严重性	5
摘要	X(文件) 中的 X(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 XX(文件)的第 N 行的 XX(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 YY（文件）中第 M 行的 YY（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）。 示例 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： ... $hash = hash_pbkdf2('sha256', $password, $random_salt, 100000, strlen($password)); ... 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。 示例 2：以下代码可从密码衍生密钥，但使用由系统管理员确定的值： ... $hash = hash_pbkdf2('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
摘要	在 X(文件):N 中配置的 XML 解析器不阻止或限制文档类型定义 (DTD) 实体解析。这会使解析器暴露在 XML Entity Expansion Injection 之下使用配置为不阻止或限制文档类型定义 (DTD) 实体解析的 XML 解析器可能会使解析器暴露在 XML Entity Expansion Injection 之下。
解释	XML Entity Expansion Injection（也称为 XML Bombs）为 DoS 攻击，受益于有效且结构良好的 XML 块，这些块以指数方式扩展，直到耗尽服务器分配的资源。XML 允许定义充当字符串替换宏的自定义实体。通过嵌套重复出现的实体解析，攻击者可能很容易使服务器资源崩溃。 以下 XML 文档显示了 XML Bomb 的一个示例。 <?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> 此测试通过将小型 XML 文档的内存扩展到 3GB 以上来使服务器崩溃。
建议	在将用户提供的数据写入 XML 时，请遵循以下准则： - 通过所用的 XML 解析器禁用实体扩展。 libxml_disable_entity_loader(true); 或者 $dom->resolveExternals=false; - 写入到 XML 之前，先对用户输入进行 XML 实体编码。 - 如果需要允许实体扩展，则： a. 验证输入以确保扩展的实体是适当且允许的。 b. 限制该解析器在加载 XML 时可以执行的功能： $doc = XMLReader::xml($badXml,'UTF-8',LIBXML_NONET);
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
摘要	在 X(文件) 的第 N 行中，X(方法) 方法将写入未经验证的 XML 输入。攻击者可以利用此调用将任意元素或属性注入 XML 文档的开头。XML External Entity (XXE) Injection 之所以不同于 XML injection，是因为攻击者通常会控制插入到 XML 文档开头的输入。如果处理未经验证的 XML 文档，可能会使攻击者更改 XML 的结构和内容、对主机服务器进行端口扫描或对内部网络进行主机扫描、在文件系统中加入任意文件或导致拒绝应用程序的服务。
解释	在以下情况下会发生 XML External Entity (XXE) 注入： 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 XX(文件)的第 N 行的 XX(函数) 中。 2. 数据写入到 XML 文档中。 在这种情况下，由 YY（文件）的第 M 行的 YY（函数） 编写 XML。 应用程序通常使用 XML 来存储数据或发送消息。当 XML 用于存储数据时，XML 文档通常会像数据库一样进行处理，而且可能会包含敏感信息。XML 消息通常在 web 服务中使用，也可用于传输敏感信息。XML 消息甚至还可用于发送身份验证凭据。 如果攻击者能够写入原始 XML，则可以更改 XML 文档和消息的语义。在危害最轻的情况下，攻击者可能能够插入嵌套的实体引用，导致 XML 解析器不断消耗越来越多的 CPU 资源。在 XML External Entity Injection 危害更大的情况下，攻击者可能能够添加 XML 元素以公开本地文件系统资源的内容、泄漏内部网络资源的存在状态或公开后端内容本身。 示例 1：下面是一些易受 XXE 攻击的代码： 假定攻击者能控制以下代码的输入 XML： ... <?php $goodXML = $_GET["key"]; $doc = simplexml_load_string($goodXml); echo $doc->testing; ?> ... 现在假设攻击者将以下 XML 传递到 Example 2 中的代码： <?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE foo [<!ELEMENT foo ANY > <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

返回目录

	在处理此 XML 时，将使用系统 boot.ini 文件的内容填充 <foo> 元素的内容。攻击者可能会利用返回到客户端的 XML 元素来窃取数据或获取有关网络资源是否存在的信息。
建议	将用户提供的数据写入 XML 时，应该遵守以下准则： - 通过所用的 XML 解析器禁用实体扩展。 libxml_disable_entity_loader(true); or $dom->resolveExternals=false; - 写入到 XML 之前，先对用户输入进行 XML 实体编码。 - 如果需要允许实体扩展，则： a. 对输入进行验证，以确保扩展实体没有问题并允许使用。 b. 限制该解析器在加载 XML 时可以执行的功能： $doc = XMLReader::xml($badXml,'UTF-8',LIBXML_NONET);
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
摘要	在 X(文件) 的第 N 行中，X(方法) 方法将写入未经验证的 XML 输入。攻击者可以利用此调用将任意元素或属性注入 XML 文档的正文中。XML Injection 之所以不同于 XML External Entity (XXE) Injection，是因为攻击者通常会控制插入到 XML 文档中部或末尾的输入。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 XX(文件)的第 N 行的 XX(函数) 中。 2. 数据写入到 XML 文档中。 在这种情况下，由 YY（文件）的第 M 行的 YY（函数） 编写 XML。 应用程序通常使用 XML 来存储数据或发送消息。当 XML 用于存储数据时，XML 文档通常会像数据库一样进行处理，而且可能会包含敏感信息。XML 消息通常在 web 服务中使用，也可用于传输敏感信息。XML 消息甚至还可用于发送身份验证凭据。 如果攻击者能够写入原始 XML，则可以更改 XML 文档和消息的语义。危害最轻的情况下，攻击者可能会插入无关的标签，导致 XML 解析器抛出异常。XML injection 更为严重的情况下，攻击者可以添加 XML 元素，更改身份验证凭据或修改 XML 电子商务数据库中的价格。还有一些情况，XML injection 可以导致 cross-site scripting 或 dynamic code evaluation。 示例 1： 假设攻击者能够控制下列 XML 中的 shoes。 <order> <price>100.00</price> <item>shoes</item> </order> 现在假设，在后端 Web 服务请求中包含该 XML，用于订购一双鞋。假设攻击者可以修改请求，并将 shoes 替换成 shoes</item><price>1.00</price><item>shoes。新的 XML 如下所示： <order> <price>100.00</price> <item>shoes</item><price>1.00</price><item>shoes</item> </order>

返回目录

	当使用 XML 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。 如果攻击者控制了已解析 XML 文档的前部或全部内容，则可能会发生这种攻击的一种更严重的形式，称为 XML External Entity (XXE) Injection。 示例 2：下面是一些易受 XXE 攻击的代码： 假定攻击者能控制以下代码的输入 XML： ... <?php $goodXML = $_GET["key"]; $doc = simplexml_load_string($goodXml); echo $doc->testing; ?> ... 现在假设攻击者将以下 XML 传递到 Example 2 中的代码： <?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE foo [<!ELEMENT foo ANY > <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo> 在处理此 XML 时，将使用系统 boot.ini 文件的内容填充 <foo> 元素的内容。攻击者可能会利用返回到客户端的 XML 元素来窃取数据或获取有关网络资源是否存在的信息。
建议	将用户提供的数据写入 XML 时，应该遵守以下准则： 1.不要使用从用户输入派生的名称创建标签或属性。 2.写入到 XML 之前，先对用户输入进行 XML 实体编码。 3. 将用户输入包含在 CDATA 标签中。 为了缓解 XML External Entity (XXE) Injection，可以使用以下选项： 将用户提供的数据写入 XML 时，应该遵守以下准则： 1.通过所用的 XML 解析器禁用实体扩展。 libxml_disable_entity_loader(true); or $dom->resolveExternals=false; 2.写入到 XML 之前，先对用户输入进行 XML 实体编码。 3. 如果需要实体扩展，则应： a. 对输入进行验证，以确保扩展实体没有问题并允许使用。 b. 限制该解析器在加载 XML 时可以执行的功能： $doc = XMLReader::xml($badXml,'UTF-8',LIBXML_NONET);
CWE	CWE ID 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**XPath Injection**
默认严重性	4
摘要	在 X(文件) 的第 N 行，X(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 XX(文件)的第 N 行的 XX(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 YY（文件）的第 M 行中的 YY（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。
	...
	<?php
	load('articles.xml');
	
	$xpath = new DOMXPath($doc);
	$emailAddrs = $xpath-
	>query("/accounts/account[acctID='" .
	$_GET["test1"] .
	"']/email/text()");
	//$arts = $xpath-
	>evaluate("/accounts/account[acctID='" .
	$_GET["test1"] .
	"']/email/text()")
	
	foreach ($emailAddrs as $email)
	{
	echo $email->nodeValue."";
	}
	?>
	...
	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，

返回目录

	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
	...
	<?php
	load('articles.xml');
	$accId = intval($_GET["accountId"]);
	
	$xpath = new DOMXPath($doc);
	$emailAddrs = $xpath-
	>query("/accounts/account[acctID='" . $accId . "']/email/text()");
	//$arts = $xpath-
	>evaluate("/accounts/account[acctID='" . $accId . "']/email/text()")
	
	foreach ($emailAddrs as $email)
	{
	echo $email->nodeValue."";
	}
	?>
	...
	intval() PHP 函数有一点需要特别注意。当送入 intval() 函数的值不能被转换成数字时（即“abc”），intval() 函数将返回零。所以当 intval() 函数返回零时，应使应用程序采取正确的动作。

CWE	CWE ID 643
OWASP2017	A1 Injection

返回目录

漏洞名称	**XQuery Injection**
默认严重性	4
摘要	在 X(文件) 的第 N 行，X(方法) 方法调用 XQuery 表达式，该表达式是用未经验证的输入创建的。通过这种调用，攻击者能够修改指令的含义或执行任意 XQuery 表达式。通过利用用户输入构建动态 XQuery 表达式，攻击者就能够修改指令的含义。

解释	XQuery 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 XX(文件)的第 N 行的 XX(函数) 中。
	2. 数据用于动态构造一个 XQuery 表达式。
	在这种情况下，查询将传递到 YY（文件）的第 M 行中的 YY（函数）。
	示例 1：下列代码可动态构造和执行 XQuery 表达式，以根据给定用户名和密码的组合检索用户帐户。由于用户名和密码是从 HTTP 请求中读取的，因此不可信。
	...
	<?php
	require_once 'zorba_api.php';
	$memstor = InMemoryStore::getInstance();
	$z = Zorba::getInstance($memstor);
	try {
	// get data manager
	$dataman = $z->getXmlDataManager();
	// load external XML document
	$dataman->loadDocument('users.xml',
	file_get_contents('users.xml'));
	// create and compile query
	$express =
	"for \$user in doc(users.xml)//user[username='" . $_GET["username"] .
	"'and pass='" . $_GET["password"] . "'] return \$user"
	$query = $zorba->compileQuery($express);
	// execute query
	$result = $query->execute();
	?>
	...
	在正常情况下，诸如搜索采用相应用户名和密码的用户帐户，该代码执行的表达式如下：
	for \$user in doc(users.xml)//user[username='test_user' and
	pass='pass123'] return \$user
	但是，由于这个表达式是动态构造的，由一个常数基查询字符串和一个用户输入字符串连接而成，因此只有在 username 或 password 不

返回目录

	包含单引号字符时，才会正确执行这一查询。如果攻击者为 username 输入字符串 admin' or 1=1 or ''='，则该查询会变成：
	for \$user in doc(users.xml)//user[username='admin' or 1=1 or ''='' and password='x' or ''=''] return \$user
	添加条件 admin' or 1=1 or ''=' 会使 XQuery 表达式永远评估为 true。因此，该查询在逻辑上等同于更简单的查询：
	//user[username='admin']
	如此简化的查询会使攻击者绕过查询与密码匹配的要求；现在查询会返回文档中存储的管理用户，无论输入什么样的密码。

建议	造成 XQuery injection 漏洞的根本原因在于攻击者能够改变 XQuery 表达式查询的上下文，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 XQuery 表达式后，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，确保 XQuery 表达式中由用户控制的数值完全来自于预定的字符集合，不包含任何上下文中所已使用的 XQuery 元字符。如果由用户控制的数值要求它包含 XQuery 元字符，则使用相应的编码机制删除这些元字符在 XQuery 表达式中的意义。
CWE	CWE ID 652
OWASP2017	A1 Injection

返回目录

漏洞名称	**XSLT Injection**
默认严重性	4
摘要	在 X(文件) 的第 N 行中，X(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 XX(文件)的第 N 行的 XX(函数) 中。 2. 数据写入到 XSL 样式表中。 在这种情况下，由 YY（文件）的第 M 行的 YY（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。 如果攻击者完全控制了提交给应用程序的样式表，则他们可能还会执行 XXE (XML External Entity) Injection 攻击。 示例 1: 下面是一些易受 XSLT 注入攻击的代码: ... <?php $xml = new DOMDocument; $xml->load('local.xml'); $xsl = new DOMDocument; $xsl->load($_GET['key']); $processor = new XSLTProcessor; $processor->registerPHPFunctions(); $processor->importStyleSheet($xsl); echo $processor->transformToXML($xml); ?> ... 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" xmlns:php="http://php.net/xsl"> <xsl:template match="/"> <script>alert(123)</script> </xsl:template>

返回目录

	</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： <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:php="http://php.net/xsl"> <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。 3. 执行任意 PHP 代码： XSLT 处理器通过启用“registerPHPFunctions”，能将本机 PHP 语言方法暴露为 XSLT 函数。 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:php="http://php.net/xsl"> <xsl:template match="/"> <xsl:value-of select="php:function('passthru','ls -la')"/> </xsl:template> </xsl:stylesheet> 上述样式表将在服务器上输出“ls”命令的结果。
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

漏洞名称	**XSS Evasion Attack**
默认严重性	3
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。 嵌入此数据前，X（文件） 文件第 N 行的 XX（元素） 解码方法会被此操作调用，然后传递到 Y（文件） 文件第 M 行 YY（元素） 中的输出。这可能允许攻击者向该解码方法提供编码数据，这将避开先前的所有净化，并导致跨站点脚本 (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。 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。 通过在此流程中解码数据，攻击者可能提供已绕过过滤恶意字符防御机制（Web 应用程序防火墙、正则表达式和其他服务器端验证）的先前编码的数据；然后这些数据会在数据有效性验证检查后解码，绕过防御机制，并导致跨站点脚本 (XSS) 攻击。
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。 如果需要在使用前解码数据，请避免发生双重编码的情况——一定要重新编码数据。一定要保证任何解码数据和任何输出之间至少有一种净化机制，以避免危险字符通过隐藏在编码数据中绕过净化。 考虑可以净化值的多种本机 PHP 方法（例如 htmlspecialchars 和 htmlentities），不要习惯性地为 Javascript 上下文对值进行编码并忽略某些包围字符，例如撇号 (')，引号 (") 和反引号 (`)。选择净化函数之前，一定要考虑输入的输出上下文。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录