GitSec.Cloud
48h to DevSecOps

漏洞修复建议速查手册

（C++篇）

开源安全研究院

2022/7/27

返回目录

简介

手册简介：

本手册是针对 C++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

Char Overflow
Cleartext Transmission Of Sensitive Information
Code Correctness:Arithmetic Operation on Boolean
Code Correctness:Erroneous Synchronization
Code Correctness:Function Not Invoked
Code Correctness:Function Returns Stack Address
Code Correctness:Macro Misuse
Code Correctness:Memory Free on Stack Variable
Code Correctness:Premature Thread Termination
Command Injection
Command Injection
Comparison Timing Attack
Connection String Injection
Creation of chroot Jail without Changing Working Directory
Dangerous Function
Dangerous Function:strcpy()
Dangerous Functions
DB Parameter Tampering
Dead Code
Denial of Service
Deprecated CRT Functions VS2005
Directory Restriction
Divide By Zero
DoS by Sleep
Double Free
Download of Code Without Integrity Check
Encoding Used Instead of Encryption
Exposure of Resource to Wrong Sphere
Exposure of System Data to Unauthorized Control Sphere
Float Overflow
Format String
Format String Attack
Format String:Argument Number Mismatch
Format String:Argument Type Mismatch
Hardcoded Absolute Path
Hardcoded password in Connection String
Hashing Length Extension Attack
Heap Inspection
Heap Inspection
Heap Inspection:Swappable Memory
Heuristic 2nd Order Buffer Overflow malloc
Heuristic 2nd Order Buffer Overflow read
Heuristic 2nd Order SQL Injection
Heuristic Buffer Improper Index Access

返回目录

Heuristic Buffer Overflow malloc
Heuristic Buffer Overflow read
Heuristic CGI Stored XSS
Heuristic DB Parameter Tampering
Heuristic NULL Pointer Dereference1
Heuristic NULL Pointer Dereference2
Heuristic Parameter Tampering
Heuristic SQL Injection
Heuristic Unchecked Return Value
Illegal Pointer Value
Improper Exception Handling
Improper Null Termination
Improper Resource Shutdown or Release
Improper Transaction Handling
Inadequate Encryption Strength
Information Exposure Through an Error Message
Insecure Compiler Optimization
Insecure Compiler Optimization:Pointer Arithmetic
Insecure Randomness
Insecure Randomness:Hardcoded Seed
Insecure Randomness:User-Controlled Seed
Insecure Randomness:Weak Entropy Source
Insecure Scrypt Parameters
Insecure SSL:Server Identity Verification Disabled
Insecure Temporary File
Insecure Transport:Weak SSL Protocol
Insufficient BCrypt Cost
Insufficient Output Length
Insufficiently Protected Credentials
Integer Overflow
Integer Overflow
Key Management:Empty Encryption Key
Key Management:Hardcoded Encryption Key
Key Management:Null Encryption Key
LDAP Injection
LDAP Injection
LDAP Manipulation
Least Privilege Violation
Leaving Temporary Files
Leftover Debug Code
Log Forging
Log Forging
Long Overflow
Memory Leak

返回目录

Memory Leak
Memory Leak:Reallocation
MemoryFree on StackVariable
Missing Check against Null
Missing Precision
MultiByte String Length
Null Dereference
NULL Pointer Dereference
Obsolete
Obsolete:Inadequate Pointer Validation
Off by One Error
Off by One Error in Arrays
Off by One Error in Loops
Off by One Error in Methods
Often Misused:Authentication
Often Misused:Exception Handling
Often Misused:File System
Often Misused:Privilege Management
Often Misused:Strings
Out-of-Bounds Read
Out-of-Bounds Read:Off-by-One
Out-of-Bounds Read:Signed Comparison
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
PBKDF2 Insufficient Iteration Count
PBKDF2 Weak Salt Value
Personal Information Without Encryption
Plaintext Storage Of A Password
Poor Style:Redundant Initialization
Poor Style:Value Never Read
Poor Style:Variable Never Used
Portability Flaw
Potential Off by One Error in Loops
Potential Path Traversal
Potential Precision Problem
Privacy Violation
Privacy Violation

返回目录

Process Control
Process Control
Race Condition:File System Access
Race Condition:Signal Handling
Redundant Null Check
Reliance on DNS Lookups in a Decision
Resource Injection
Resource Injection
Scrypt Weak Salt Value
Second Order SQL Injection
Setting Manipulation
Setting Manipulation
Short Overflow
SQL Injection
SQL Injection
Stored Blind SQL Injections
Stored Buffer Overflow boundcpy
Stored Buffer Overflow cpycat
Stored Buffer Overflow fgets
Stored Buffer Overflow fscanf
Stored Command Injection
Stored Connection String Injection
Stored DB Parameter Tampering
Stored Format String Attack
Stored LDAP Injection
Stored Log Forging
Stored Parameter Tampering
Stored Path Traversal
Stored Process Control
Stored Resource Injection
String Termination Error
Symmetric Encryption Insecure Cipher Mode
Symmetric Encryption Insecure Predictable IV
Symmetric Encryption Insecure Predictable Key
Symmetric Encryption Insecure Static IV
Symmetric Encryption Insecure Static Key
System Information Leak
System Information Leak:External
System Information Leak:Internal
TOCTOU
Type Conversion Error
Type Mismatch:Integer to Character
Type Mismatch:Negative to Unsigned
Type Mismatch:Signed to Unsigned

返回目录

Unchecked Return Value
Unchecked Return Value
Undefined Behavior
Undefined Behavior:Redundant Delete
Uninitialized Variable
Unreleased Resource
Unreleased Resource:Database
Unreleased Resource:Synchronization
Unsafe Reflection
Use After Free
Use After Free
Use of a One Way Hash without a Salt
Use Of Deprecated Class
Use of Hard coded Cryptographic Key
Use Of Hardcoded Password
Use of Insufficiently Random Values
Use of Obsolete Functions
Use of Uninitialized Pointer
Use Of Weak Hashing Primitive
Use of Zero Initialized Pointer
Weak Cryptographic Hash
Weak Cryptographic Hash:Hardcoded PBE Salt
Weak Cryptographic Hash:Hardcoded Salt
Weak Cryptographic Hash:Insecure PBE Iteration Count
Weak Cryptographic Hash:Missing Required Step
Weak Cryptographic Hash:User-Controlled PBE Salt
Weak Cryptographic Hash:User-Controlled Salt
Weak Cryptographic Signature:Insufficient Key Size
Weak Cryptographic Signature:Missing Required Step
Weak Encryption
Weak Encryption:Inadequate RSA Padding
Weak Encryption:Insecure Initialization Vector
Weak Encryption:Insecure Mode of Operation
Weak Encryption:Insufficient Key Size
Weak Encryption:Missing Required Step
Weak Mechanism
Weak Randomness Biased Random Sample
Wrong Memory Allocation
Wrong Size t Allocation
XML External Entity Injection
XML Injection
XPath Injection

返回目录

漏洞名称	**Access Control:Anonymous LDAP Bind**
默认严重性	2.0
摘要	在没有适当 access control 的情况下，X(文件) 中的 XX（函数） 函数可以执行第 N 行上的 LDAP 声明，其中可能包含受攻击者控制的值，从而允许攻击者访问未授权的目录条目。在没有适当 access control 的情况下，执行一个包含用户控制值的 LDAP 声明，这会让攻击者访问未授权的记录。

解释	若未经 authentication 便在匿名绑定下有效地执行 LDAP 查询，会导致攻击者滥用低端配置的 LDAP 环境。
	例 1：以下代码使用 ldap_simple_bind_s() 匿名绑定到一个 LDAP 目录。
	...
	rc = ldap_simple_bind_s(ld, NULL, NULL);
	if (rc != LDAP_SUCCESS) {
	...
	}
	...
	针对 ld 执行的所有 LDAP 查询都会在未经 authentication 和 access control 的情况下执行。攻击者可能会采取意想不到的方式操纵其中的某个查询，以便可以访问本应受目录 access control 机制保护的记录。

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，但是附加了一个限制，以验证当前经过身份验证的用户是否具有执行后续查询所需的权限。
	...
	snprintf(username, sizeof(username), "(cn=%s)", user);
	rc = ldap_simple_bind_s(ld, username, password);
	if (rc != LDAP_SUCCESS) {
	...
	}
	...

CWE	CCI-000213, CCI-000804, CCI-001084, CCI-002165
OWASP2017	A7 Missing Function Level Access Control

返回目录

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

解释	Database access control 错误在以下情况下发生：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 这个数据用来指定 SQL 查询中主键的值。
	在这种情况下，在 Y（文件）中第 M 行的 Y（函数） 使用该数据。
	示例 1：以下代码使用可转义元字符并防止出现 SQL 注入漏洞的参数化语句，以构建和执行用于搜索与指定标识符 [1] 相匹配的清单的 SQL 查询。您可以从与当前被授权用户有关的所有清单中选择这些标识符。
	...
	CMyRecordset rs(&dbms);
	rs.PrepareSQL("SELECT * FROM invoices WHERE id = ?");
	rs.SetParam_int(0,atoi(r.Lookup("invoiceID").c_str()));
	rs.SafeExecuteSQL();
	...
	问题在于开发者没有考虑到所有可能出现的 id 值。虽然界面生成了属于当前用户的清单标识符列表，但是攻击者可以绕过这个界面，从而获取所需的任何清单。由于此示例中的代码没有执行检查以确保用户具有访问所请求清单的权限，因此它会显示任何清单，即使此清单不属于当前用户。

建议	与其靠表示层来限制用户输入的值，还不如在应用程序和数据库层上进行 access control。任何情况下都不允许用户在没有取得相应权限的情况下获取或修改数据库中的记录。每个涉及数据库的查询都必须遵守这个原则，这可以通过把当前被授权的用户名作为查询语句的一部分来实现。
	示例 2：以下代码实现的功能与 Example 1 相同，但是附加了一个限制，以验证清单是否属于当前经过身份验证的用户。
	...
	ctx.getAuthUserName(&username);
	CMyRecordset rs(&dbms);
	rs.PrepareSQL(
	"SELECT * FROM invoices WHERE id = ? AND user = ?");
	rs.SetParam_int(0,atoi(r.Lookup("invoiceID").c_str()));
	rs.SetParam_String(1,username)

返回目录

	rs.SafeExecuteSQL(); ...
CWE	CWE ID 566
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Access Control:LDAP**
默认严重性	2.0
摘要	在没有适当 access control 的情况下，X(文件) 中的 XX（函数） 函数可以执行第 N 行上的 LDAP 声明，其中可能包含受攻击者控制的值，从而允许攻击者访问未授权的目录条目。在没有适当 access control 的情况下，执行一个包含用户控制值的 LDAP 声明，这可以让攻击者访问未授权的目录条目。

解释	Database access control 错误在以下情况下发生：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据用于在 LDAP 查询中指定一个数据值。
	在这种情况下，在 Y（文件）中第 M 行的 Y（函数） 使用该数据。
	示例 1：以下代码会在使用雇员名称创建 LDAP 查询之前，使用一个允许列表来验证雇员名称的有效性。该验证避免了 LDAP injection 漏洞，但仍可能留下代码漏洞。
	...
	fgets(username, sizeof(username), socket);
	char* regex = "^[a-zA-Z\-\.']$";
	re = pcre_compile(regex, 0, &err, &errOffset, NULL);
	rc = pcre_exec(re, NULL, username, strlen(username), 0, 0, NULL, 0);
	if(rc == 1) {
	snprintf(filter, sizeof(filter), "(employee=%s)", username);
	if ((rc = ldap_search_ext_s(ld, FIND_DN, LDAP_SCOPE_BASE,
	filter, NULL, 0, NULL, NULL, LDAP_NO_LIMIT,
	LDAP_NO_LIMIT, &result)) == LDAP_SUCCESS) {
	...
	}
	}
	问题在于开发人员未能考虑到若攻击者提供可供选择的 username 值，会发生什么情况。因为本例子中的代码是在匿名绑定情况下执行查询，不管当前已验证用户的身份如何，它都会将有效的雇员 ID 返回至目录入口。

建议	应用程序应该执行详细的验证，并以特定用户的身份绑定到目录，从而加强 access control 控制。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，但会使用 LDAP 简单身份验证来确保该查询只影响允许当前用户访问的记录。 ...

返回目录

	snprintf(username, sizeof(username), "(cn=%s)", user);
	rc = ldap_simple_bind_s(ld, username, password);
	if (rc != LDAP_SUCCESS) {
	...
	}
	fgets(username, sizeof(username), socket);
	char* regex = "^[a-zA-Z\-\.']$";
	re = pcre_compile(regex, 0, &err, &errOffset, NULL);
	rc = pcre_exec(re, NULL, username, strlen(username), 0, 0, NULL, 0););
	if(rc == 1) {
	snprintf(filter, sizeof(filter), "(employee=%s)", username);
	if ((rc = ldap_search_ext_s(ld, FIND_DN, LDAP_SCOPE_BASE,
	filter, NULL, 0, NULL, NULL, LDAP_NO_LIMIT,
	LDAP_NO_LIMIT, &result)) == LDAP_SUCCESS) {
	...
	}
	}

CWE	CWE ID 639
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Android Bad Practices:Use of Internal APIs**
默认严重性	3.0
摘要	调用 X(文件) 中第 N 行的 X(函数) 时会调用内部或隐藏的 API。应用程序调用内部或隐藏的 API。
解释	不建议开发人员使用未记录或隐藏的 API 构建其应用程序。 由于无法保证 Google 未来不会删除或更改这些 API，因此应避免使用它们，并且使用此类方法或字段具有破坏应用程序的较高风险。
建议	请勿使用内部或隐藏的 API，而是坚持使用 SDK API。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Asymmetric Encryption Improper Padding**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的加密函数使用了 Y（文件） 文件第 M 行中的弱填充方案 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 应用程序为 RSA 加密使用了不安全的填充方案。为 RSA 使用弱填充会使攻击者能够解密捕获的密文。要成功利用，应用程序必须泄漏密文解密因无效填充而失败。这也被称为 Bleichenbacher 攻击。 应用程序的加密相关功能有问题。 为 RSA 使用不安全的填充方案会使攻击者能够对解密机执行自适应选择密文攻击（自适应 CCA 攻击）。这意味着被动攻击者捕获到受害者的加密消息后，可以使用主动攻击来尝试破译截获的消息。通过对有效消息进行轻微调整，攻击者即可通过将这些修改的消息发送到解密机并监听“无效填充”错误来迭代缩小每个消息的可能性空间。自适应 CCA 方法只需要调用几万次解密机即可有效地解密消息。
建议	使用 RSA 加密，确保使用的填充方案是 OAEP。虽然 PKCS#1 v1.5 可能默认使用多种库、框架或语言，但众所周知此方法很弱，是不应使用的。
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Asymmetric Encryption Insufficient Key Size**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的 RSA 密钥生成流程使用了 Y（文件） 文件第 M 行中不足的密钥大小 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 如果用于加密数据的密钥大小不足，攻击者在为捕获的密文找到实际密钥前必须尝试的密钥总数就会减少。如果所使用的密钥足够小，攻击者可能轻松就能破解密钥。 应用程序的加密相关功能有问题。 应用程序使用小密钥进行非对称加密。
建议	密钥大小对增加攻击者破解加密密钥的难度来保护密文有重要意义。可接受的最小密钥大小取决于具体情况和应用程序的安全模型。 一般有两个主要建议：NIST 和 IAD-NSA 建议。 为保证合理的安全性，NIST 声明非对称密钥的最小分解模数为 2048 位，而 IAD-NSA 建议最小 3072 位。
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Asymmetric Encryption RSA Low Public Exponent**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的 RSA 密钥生成流程使用了 Y（文件） 文件第 M 行中的低公共指数 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 使用低公共指数的 RSA 密钥会使攻击者能够解密捕获的消息。 应用程序的加密相关功能有问题。 在某些情况下，使用有低公共指数的 RSA 密钥会使攻击者能够解密捕获的密文。最大的威胁是如果消息未填充且长度较小，这允许攻击者逆向已执行的 RSA 加密。
建议	
CWE	CWE ID 326
OWASP2017	None

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	**Boolean Overflow**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的值未经验证便被用于 Y（文件） 文件第 M 行 YY（元素） 中的算术运算中，这可能导致算术溢流（通常称为“整数溢出”）。
解释	算术溢出会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断或丢失数据；使用溢出的数值执行内存操作时，甚至导致缓冲区溢出，从而导致执行任意代码。 所有数字数据类型都按位表示。如果在算术运算后，某个值超过了其按位表示的位数，那么被加的最重要的数字将被截断，此截断后剩余的值将是回绕操作的剩余值——这称为“算术溢出”，另常误称为——“整数溢出”，此误称是有误导的，因为此名称还适用于许多其他类型。如果可能溢出的数据类型低于其最小值，则它们将负向回绕，通常称为下溢。 例如，如果无符号的 32 位整数设置为 4,294,967,295，然后加 1，它就会溢出并回绕到 0。如果有符号的 32 位整数值为 2,147,483,647，然后加 1，则它会溢出并回绕到 -2,147,483,648。 为了保证代码正确，一定要检查值是否在预期范围内，确保算术运算结果不会溢出或下溢。
建议	对可能包含任意值的数据执行算术运算时，请考虑添加一个检查以保证数据在范围内，且这些操作的结果不会导致溢出或下溢。 考虑为所有算术运算创建封装器，以对特殊情况进行特定处理；例如，如果检查显示已经或将要发生溢出，则抛出异常。
CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**Buffer Improper Index Access**
默认严重性	5
摘要	X（文件） 文件第 N 行的数组索引 XX（元素） 被用于引用 Y（文件） 文件第 M 行的数组 YY（元素） 单元的索引。
解释	数组的元素通常会映射一个数值，表示元素在数组中的位置。如果无法保证用于从数组中读取元素的索引值在数组的范围内，会导致引用超出数组边界。
建议	将值用作数组索引之前，一定要检查值是否非负且小于数组大小。
CWE	CWE ID 129
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow**
默认严重性	4.0
摘要	X(文件) 中的 XX（函数） 函数在第 N 行中分配的内存边界之外写入数据，这可能会破坏数据、引起程序死机或导致恶意代码的执行。在一块分配的内存边界之外写入数据可能会破坏数据、造成程序崩溃或导致恶意代码的执行。
解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。 在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。 虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。 在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可以轻易地覆盖缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。 Buffer overflow 漏洞通常出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 在本实例中，我们主要考虑第一种情况，因为： — 外部数据经由 X（文件） 的第 N 行进入程序 X（函数）。 — 该数据可影响 Y（文件） 中第 M 行 Y（函数） 中的内存处理。 以下例子分别演示了上面三种情况。 例 1.a：以下示例代码显示了简单的 buffer overflow，它通常由第一种情况所导致，即依靠外部数据来控制行为的代码。该代码使用 gets() 函数将一个任意大小的数据读取到堆栈缓冲区中。因为没有什么方法

返回目录

	可以限制该函数读取数据的量，所以代码的安全性就依赖于用户始终输入比 BUFSIZE 少的字符数量。
	...
	char buf[BUFSIZE];
	gets(buf);
	...
	例 1.b：这一例子表明模仿 C++ 中 gets() 函数的不安全行为是如此的简单，只要通过使用 >> 运算符将输入读取到 char[] 字符串中。
	...
	char buf[BUFSIZE];
	cin >> (buf);
	...
	例 2：虽然本例中的代码也是依赖于用户输入来控制代码行为，但是它通过使用边界内存复制函数 memcpy() 增加了一个间接级。该函数接受一个目标缓冲区、一个起始缓冲区和要复制的字节数。虽然输入缓冲区由 read() 的边界调用填充，但是 memcpy() 复制的字节数需要由用户指定。
	...
	char buf[64], in[MAX_SIZE];
	printf("Enter buffer contents:\n");
	read(0, in, MAX_SIZE-1);
	printf("Bytes to copy:\n");
	scanf("%d", &bytes);
	memcpy(buf, in, bytes);
	...
	注：该类型的 buffer overflow 漏洞（程序可读取数据，然后对剩余数据随后进行的内存操作中的一个数值给予信任）已在图像、音频和其他的文件处理库中频繁地出现。
	例 3：这是一个关于第二种情况的例子，代码受未在本地校验的数据属性的影响。在本例中，名为 lccopy() 的函数将一个字符串作为其变量，然后返回一个堆分配字符串副本，并将该字符串的所有大写字母转化成了小写字母。因为该函数认为 str 总是比 BUFSIZE 小，所以它不会对输入执行任何边界检查。如果攻击者避开对调用 lccopy() 代码的检查，或者如果更改代码，使得程序员对 str 长度的原有假设与实际不符，那么 lccopy() 就会通过无边界调用 strcpy() 溢出 buf。
	char *lccopy(const char *str) {
	char buf[BUFSIZE];
	char *p;
	strcpy(buf, str);
	for (p = buf; *p; p++) {
	if (isupper(*p)) {
	*p = tolower(*p);
	}

返回目录

	}
	return strdup(buf);
	}
	例 4：以下代码演示了第三种情况，代码过于复杂，以致于程序员无法准确预测它的行为。本代码来自于常用的 libPNG 图象解码器，这种解码器被广泛应用于许多应用程序中，包括 Mozilla 和某些 Internet Explorer 版本。
	该代码似乎可以安全地执行边界检查，因为它检测变量长度的大小，该变量长度会在之后用来控制 png_crc_read() 复制的数据量。然而，在测试长度前，该代码会立即对 png_ptr->mode 执行检查，如果检查失败，便会发出一个警告，然后会继续进行处理。因为 length 测试在 else if 块中进行，如果针对该代码的首次测试失败，那么就不会再测试 length，而将其盲目地用于调用 png_crc_read()，因此很容易引起堆栈 Buffer Overflow。
	虽然本例中的代码不是我们所遇见的代码中最复杂的，但是它足以说明为什么要尽可能地降低执行内存操作代码的复杂度。
	if (!(png_ptr->mode & PNG_HAVE_PLTE)) {
	/* Should be an error, but we can cope with it */
	png_warning(png_ptr, "Missing PLTE before tRNS");
	}
	else if (length > (png_uint_32)png_ptr->num_palette) {
	png_warning(png_ptr, "Incorrect tRNS chunk length");
	png_crc_finish(png_ptr, length);
	return;
	}
	...
	png_crc_read(png_ptr, readbuf, (png_size_t)length);
	例 5：本例同样演示了第三种情况，程序过于复杂，使其暴露出 buffer overflow 的问题。在这种情况下，问题出现的原因在于其中某个函数的接口不明确，而不是代码结构（同上一个例子中描述的情况一样）。
	getUserInfo() 函数采用一个定义为多字节字符串的用户名和一个指向用户信息结构的指针，这一结构由该用户的相关信息填充。因为 Windows authentication 中的用户名使用 Unicode，所以 username 参数首先要从多字节字符串转换成 Unicode 字符串。然后，这个函数便会错误地将 unicodeUser 的长度以字节形式而不是字符形式传递出去。调用 MultiByteToWideChar() 可能会把 (UNLEN+1)*sizeof(WCHAR) 宽字符或者 (UNLEN+1)*sizeof(WCHAR)*sizeof(WCHAR) 字节，写到 unicodeUser 数组，该数组仅分配了 (UNLEN+1)*sizeof(WCHAR) 个字节。如果 username 字符串包含了多于 UNLEN 的字符，那么调用 MultiByteToWideChar() 将会溢出 unicodeUser 缓冲区。
	void getUserInfo(char *username, struct _USER_INFO_2 info){
	WCHAR unicodeUser[UNLEN+1];

返回目录

	MultiByteToWideChar(CP_ACP, 0, username, -1,
	unicodeUser, sizeof(unicodeUser));
	NetUserGetInfo(NULL, unicodeUser, 2, (LPBYTE *)&info);
	}

建议	绝不要使用自身安全性较差的函数，如 gets()，避免使用那些难以安全使用的函数，如 strcpy()。使用相应的边界函数（如 strncpy() 或在 strsafe.h [4] 中定义的 WinAPI 函数）来取代无边界函数（如 strcpy()）。 虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	CWE ID 787, CWE ID 120, CWE ID 129, CWE ID 131
OWASP2017	A1 Injection

返回目录

漏洞名称	**Buffer Overflow AddressOfLocalVarReturned**
默认严重性	4
摘要	Y（文件） 文件第 M 行的指针 YY（元素） 在释放后又被使用。
解释	内存释放后使用错误会导致代码使用之前已分配特定值的内存区域，但该区域已被释放并且可能已被另一个值覆写。此错误可能导致意外行为、内存损坏和崩溃错误。在某些情况下，释放和使用的内存部分会被用于确定执行流程，而攻击者可以引发错误，导致执行恶意代码。 指向变量的指针使代码可以有设定大小的到动态分配变量的地址。最终，指针的目标会被释放——这可能是在代码中显式完成的，例如以编程的方式释放此变量时；或者隐式完成，例如在返回局部变量时——返回后，变量的范围也被释放。释放后，内存将被应用程序重用，并被新数据覆写。此时，取消引用此指针可能会解析新写入的和意外的数据。
建议	不要返回局部变量或指针 检查代码以确保显式释放指针后没有流程可以使用指针
CWE	CWE ID 562
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow boundcpy WrongSizeParam**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 121
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow boundedcpy**
默认严重性	5
摘要	Y（文件） 文件第 M 行中的 Size 参数 YY（元素） 受 X（文件）文件第 N 行中用户输入 XX（元素） 的影响。这可能会导致缓冲区溢出漏洞，被执行恶意代码。
解释	允许污染的输入设置从源复制到目标的字节数的大小可能会导致内存损坏、意外行为、不稳定和数据泄漏。在某些情况下，例如存储器的其他特定区域也受用户输入控制时，这可能导致代码被执行。 如果从源复制到目标的字节数的大小大于目标大小，则会发生溢出，超出预期缓冲区的内存将被覆盖。这个大小值是根据用户输入导出的，而用户可能提供无效且危险的缓冲区大小。
建议	不要信任用户提供的内存分配大小；要从复制的值中导出。 如果确实需要根据提供的值分配内存，请仅将此大小限制为安全值。特别要确保此值不超过目标缓冲区的大小。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow boundedcpy2**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow cin**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow cpycat**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow fgets**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow Indexes**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow IndexFromInput**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 787
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow LongString**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow Loops**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow Loops Old**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow LowBound**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow OutOfBound**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow scanf**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow StrcpyStrcat**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow unbounded**
默认严重性	5
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Buffer Overflow Unbounded Buffer**
默认严重性	5
摘要	X（文件） 文件第 N 行创建的缓冲 XX（元素） 未经合适的边界检查便被 YY（元素） 写入 Y（文件） 文件第 M 行中的缓冲区。
解释	如果未限制内存写入到预期缓冲区，则攻击者可能通过使用恶意数据溢出目标缓冲区将任意数据写入内存，从而破坏内存，导致不可预测的行为并可能执行恶意代码。 如果代码尝试对分配的内存缓冲区执行原始内存写入，但无法保证写入不超过、“丢失”或逸出给定缓冲区，这就会发生缓冲区溢出。利用这个漏洞，内存写入管理错误最终会导致将数据写入分配的缓冲区之外，从而破坏内存。如果攻击者可以启动、控制或操纵此内存损坏，攻击者就能破坏特定的数据，转移应用程序逻辑流并执行恶意代码。 如果缓冲区被可能不安全的方法写入，未通过任何检查确保写入的数据在其范围内，就可能导致溢出。
建议	写入内存前一定要根据情况执行严格的边界检查，确保数据不会超出目标缓冲区边界。 在可能的情况下，考虑使用有自己的内部边界检查、或者要求在函数参数中提供边界的替代函数。如果边界是作为函数参数的一部分提供的，请确保这些边界是从源缓冲区正确地推导出来的——不正确的边界也可能导致缓冲区溢出。
CWE	CWE ID 120
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow Unbounded Format**
默认严重性	5
摘要	X（文件） 文件第 N 行创建的参数 XX（元素） 未经恰当边界检查便被 YY（元素） 写入 Y（文件） 文件第 M 行中的格式字符串。
解释	如果未限制内存写入到预期缓冲区，则攻击者可能通过使用恶意数据溢出目标缓冲区将任意数据写入内存，从而破坏内存，导致不可预测的行为并可能执行恶意代码。 如果未限制内存写入到格式字符串再写到缓冲区，则攻击者可能通过使用恶意数据溢出目标缓冲区将任意数据写入内存，从而破坏内存，导致不可预测的行为并可能执行恶意代码。 如果代码尝试对分配的内存缓冲区执行原始内存写入，但无法保证写入不超过、“丢失”或逸出给定缓冲区，这就会发生缓冲区溢出。利用这个漏洞，内存写入管理错误最终会导致将数据写入分配的缓冲区之外，从而破坏内存。如果攻击者可以启动、控制或操纵此内存损坏，攻击者就能破坏特定的数据，转移应用程序逻辑流并执行恶意代码。
建议	写入格式字符串前一定要根据情况执行严格的边界检查，确保数据不会超出目标缓冲区边界。 在可能的情况下，考虑使用有自己的内部边界检查、或者要求在函数参数中提供边界的替代函数。如果边界是作为函数参数的一部分提供的，请确保这些边界是从源缓冲区正确地推导出来的——不正确的边界也可能导致缓冲区溢出。
CWE	CWE ID 120
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow Wrong Buffer Size**
默认严重性	5
摘要	X（文件） 文件第 N 行创建的缓冲区 XX（元素） 被 YY（元素）写入 Y（文件） 文件第 M 行的缓冲区，但缓冲区分配计算错误会允许 XX（元素） 超出此缓冲区的大小，导致溢出。
解释	如果未限制内存写入到预期缓冲区，则攻击者可能通过使用恶意数据溢出目标缓冲区将任意数据写入内存，从而破坏内存，导致不可预测的行为并可能执行恶意代码。 如果代码尝试对分配的内存缓冲区执行原始内存写入，但无法保证写入不超过、“丢失”或逸出给定缓冲区，这就会发生缓冲区溢出。利用这个漏洞，内存写入管理错误最终会导致将数据写入分配的缓冲区之外，从而破坏内存。如果攻击者可以启动、控制或操纵此内存损坏，攻击者就能破坏特定的数据，转移应用程序逻辑流并执行恶意代码。 某些函数根据传递给它们的参数计算要复制的数据的大小。如果这些参数计算错误或与写入缓冲区无关（例如用户输入），则从源写入的数据量可能超出目标的边界，从而导致缓冲区溢出。
建议	尽量根据数据本身确定要写入的数据的大小。 切勿根据用户输入提供数据大小，因为这些输入可能是恶意的。 使用硬编码值不能成为不使用数据大小验证的借口；将数据写入缓冲区之前一定要验证数据的大小。
CWE	CWE ID 131
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow:Format String**
默认严重性	4.0
摘要	X(文件) 第 N 行中 X(函数) 的 format string 参数未能正确限制该函数可以写入的数据量，这会允许程序在分配的内存边界之外写入数据。这种行为会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。该程序使用了界定不当的 format string，允许其在分配的内存边界之外写入数据。这种行为会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这里，一个结构不良的 format string 会导致程序在分配的内存边界之外写入数据。
	示例：以下代码会溢出 c，因为 double 类型需要的空间超过分配给 c 的空间。
	void formatString(double d) {
	char c;
	scanf("%d", &c)
	}

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。

返回目录

	当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	CWE ID 787, CWE ID 134
OWASP2017	A1 Injection

返回目录

漏洞名称	**Buffer Overflow:Format String (%f/%F)**
默认严重性	4.0
摘要	X(文件) 第 N 行中 X(函数) 的 format string 参数使用了一个界定不当的 format string，该字符串包含一个 %f 或 %F 浮点说明符。特别大的浮点值会导致该程序在分配的内存边界之外写入数据，这会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。该程序使用了界定不当的 format string，其中包含一个 %f 或 %F 浮点说明符。特别大的浮点值会导致该程序在分配的内存边界之外写入数据，这会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这里，一个结构不良的 format string 会导致程序在分配的内存边界之外写入数据。
	示例： 以下代码会溢出 buf，因为根据 f 的大小，格式字符串说明符 "%d %.1f ... " 可能会超出分配的内存大小。
	void formatString(int x, float f) {
	char buf[40];
	sprintf(buf, "%d %.1f ... ", x, f);
	}

返回目录

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow:Off-by-One**
默认严重性	4.0
摘要	X(文件) 中的 XX（函数） 函数在第 N 行的 X（缓冲区） 边界之外写入数据，这可能会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。该程序在分配的内存边界之外写入数据，这可能会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的 off-by-one 错误在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆栈 buffer overflow 和堆 buffer overflow 等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	示例：以下代码包含一个 off-by-one 缓冲区溢出，当 recv 返回的字节数达到最大允许读取的 sizeof(buf) 字节数时，便会发生此溢出。在这种情况下，随后对 buf[nbytes] 的间接引用会将 null 字节写入到所分配内存的边界之外。
	`void receive(int socket) {`
	`  char buf[MAX];`
	`  int nbytes = recv(socket, buf, sizeof(buf), 0);`
	`  buf[nbytes] = '\0';`
	`  ...`
	`}`

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。

返回目录

	当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Buffer Overflow:Signed Comparison**
默认严重性	4.0
摘要	X(文件) 中的 XX（函数） 函数使用带符号的比较来检查第 N 行的变量 X（变量），但随后会将 X（变量） 视为不带符号的变量。这会导致该程序从第 N 行的 X（缓冲区） 边界之外读取数据。该程序使用带符号的比较来检查稍后会被视为不带符号的值。这将会导致程序在分配的内存边界之外写入数据，这可能会损坏数据、引起程序崩溃或为恶意代码的执行提供机会。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	示例：以下代码尝试通过检查从 getInputLength() 中读取的不可信的值，验证其是否小于目标缓冲区 output 的大小，来避免 off-by-one buffer overflow。然而，因为 len 和 MAX 之间比较的是带符号的值，所以如果 len 为负值，在其转换为 memcpy() 不带符号的参数时，将会变成一个超级大的正数。
	void TypeConvert() {
	char input[MAX];
	char output[MAX];
	fillBuffer(input);
	int len = getInputLength();

返回目录

	if (len <= MAX) {
	memcpy(output, input, len);
	}
	...
	}

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Buffer Size Literal**
默认严重性	2
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 118
OWASP2017	None

返回目录

漏洞名称	**Buffer Size Literal Condition**
默认严重性	2
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 118
OWASP2017	None

返回目录

漏洞名称	**Buffer Size Literal Overflow**
默认严重性	2
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 118
OWASP2017	None

返回目录

漏洞名称	**CGI Reflected XSS**
默认严重性	5
摘要	X（文件） 文件第 N 行中发现未经验证的输入。第 M 行的 YY（元素） 中发现可能有 XSS 漏洞。
解释	特别设计的 HTTP 请求可能会检索系统信息并通过 CGI（通用网关接口）利用系统。 CGI 规范提供了在服务器计算机及其连接的主机上读取文件、获取 shell 访问权限和损坏文件系统的可能性。 获取访问权限的方法有：利用脚本的假设，利用服务器环境中的弱点，以及利用其他程序和系统调用中的弱点。 CGI 脚本的主要弱点是输入验证不充分。
建议	如果不必要，避免将用户输入绑定到系统变量。 验证并编码所有用户输入。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**CGI Stored XSS**
默认严重性	5
摘要	X（文件） 文件第 N 行中发现未经验证的数据库输出。第 M 行的 YY（元素） 中发现可能有 XSS 漏洞。
解释	保存的恶意数据可能检索系统信息并通过 CGI（通用网关接口）利用系统。 CGI 规范提供了在服务器计算机及其连接的主机上读取文件、获取 shell 访问权限和损坏文件系统的可能性。 获取访问权限的方法有：利用脚本的假设，利用服务器环境中的弱点，以及利用其他程序和系统调用中的弱点。 CGI 脚本的主要弱点是输入验证不充分。
建议	不要提供不必要的文件权限。 验证并编码所有数据库输出。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Char Overflow**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的值未经验证便被用于 Y（文件） 文件第 M 行 YY（元素） 中的算术运算中，这可能导致算术溢流（通常称为“整数溢出”）。
解释	算术溢出会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断或丢失数据；使用溢出的数值执行内存操作时，甚至导致缓冲区溢出，从而导致执行任意代码。 所有数字数据类型都按位表示。如果在算术运算后，某个值超过了其按位表示的位数，那么被加的最重要的数字将被截断，此截断后剩余的值将是回绕操作的剩余值——这称为“算术溢出”，另常误称为——“整数溢出”，此误称是有误导的，因为此名称还适用于许多其他类型。如果可能溢出的数据类型低于其最小值，则它们将负向回绕，通常称为下溢。 例如，如果无符号的 32 位整数设置为 4,294,967,295，然后加 1，它就会溢出并回绕到 0。如果有符号的 32 位整数值为 2,147,483,647，然后加 1，则它会溢出并回绕到 -2,147,483,648。 为了保证代码正确，一定要检查值是否在预期范围内，确保算术运算结果不会溢出或下溢。
建议	对可能包含任意值的数据执行算术运算时，请考虑添加一个检查以保证数据在范围内，且这些操作的结果不会导致溢出或下溢。 考虑为所有算术运算创建封装器，以对特殊情况进行特定处理；例如，如果检查显示已经或将要发生溢出，则抛出异常。
CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**Cleartext Transmission Of Sensitive Information**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XX（元素） 包含可能敏感的个人信息，通过 Y（文件） 文件第 M 行 YYY（方法） 方法中的 YY（元素）在不安全网络上发送。这可能会暴露个人数据并使其被盗。
解释	通过网络传输敏感的个人详情（例如密码、社会安全号码、信用卡数据和其他形式的 PII（个人身份信息））时，一定要进行保护。在未加密的网络上发送这些私密数据，如果不使用 SSL/TLS 或其他形式的加密，可能会暴露用户的秘密信息，使其面临被假冒身份、身份盗窃和财务欺诈的风险。 如果 SSL/TLS 通道终止于前端 Web 服务器、反向代理等此类设施，则可以忽略此问题。 应用程序以多种方式处理敏感的私密信息。从某个点开始，这些秘密数据会发送到全网络，就是因为应用程序在不受保护的通道上发送数据前，未使用 SSL/TLS 或任何其他安全协议，也没有确保数据已加密。
建议	通用指南： 一定要保护所有 PII 和其他敏感数据，尤其是通过网络发送数据时更要这样。 只要传输敏感数据时都要使用 SSL/TLS。也可以使用其他加密协议，例如 IPsec 或 SSH。 仔细考虑应用程序是否确实需要这些个人信息。没有收到的数据就不会暴露。 具体建议： 在 Web 应用程序中，如果没有先验证通道是否有 SSL 保护，不要将个人信息直接输出到响应。 不要将个人信息直接写入标准套接字。而始终要使用 SSLSocket 确保通道使用了 SSL/TLS。
CWE	CWE ID 319
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Arithmetic Operation on Boolean**
默认严重性	3.0
摘要	在 X(文件) 的第 N 行，函数 XX（函数） 对布尔值执行算术运算，这可能不会得到程序员想要的结果。该程序对布尔值执行算术运算，这可能不会得到程序员想要的结果。
解释	对布尔值执行算术运算可能与对整型值执行运算的操作方式不同，这可能会导致意外的结果。
建议	请避免对布尔值执行算术运算。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Erroneous Synchronization**
默认严重性	3.0
摘要	如果 X(文件) 中的 XX（函数） 函数向其他线程发出信号后并未解除锁定 mutex，则其他线程将保持锁定并继续等待 mutex。如果线程向其他线程发出信号后并未解除锁定 mutex，则其他线程将保持锁定并继续等待 mutex。

解释	在线程向其他等待 mutex 的线程发出信号后，它必须调用 pthread_mutex_unlock() 来解除锁定 mutex，然后其他线程才能开始运行。如果发出信号的线程无法解除锁定 mutex，则在第二个线程中调用 pthread_cond_wait() 函数时不会返回任何值，该线程也将无法执行。
	例 1：以下代码通过调用 pthread_cond_signal() 向其他等待 mutex 的线程发出信号，但无法解除锁定 mutex，而其他线程会继续等待 mutex。
	…
	pthread_mutex_lock(&count_mutex);
	// Signal waiting thread
	pthread_cond_signal(&count_threshold_cv);
	…

建议	向其他线程发出信号后，务必立即解除锁定相关的 mutex。
	示例 2：以下代码是经过重写的 Example 1 代码，用于在调用 pthread_cond_signal() 之后解除锁定互斥体。
	…
	pthread_mutex_lock(&count_mutex);
	// Signal waiting thread
	pthread_cond_signal(&count_threshold_cv);
	pthread_mutex_unlock(&count_mutex);
	…

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Function Not Invoked**
默认严重性	3.0
摘要	因为缺少结尾插入语，X(文件) 中第 N 行的表达式会引用函数指针的值，而不是函数的返回值。因为缺少结尾插入语，该表达式会引用函数指针的值，而不是函数的返回值。
解释	该表达式永远不会为 NULL，因为它会引用函数的指针，而不是函数的返回值。 例 1：以下条件永远不会触发。语句 getChunk == NULL 将永远是 false，因为 getChunk 是程序中定义的一个函数名称。 if (getChunk == NULL) return ERR;
建议	添加插入语，使该函数得到调用。 示例 2：将前面的示例更改为调用 getChunk() 函数，如下所示： if (getChunk() == NULL) return ERR;
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Function Returns Stack Address**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数会在第 N 行返回一个堆栈变量的地址。这可能会引起意料之外的程序行为，通常是程序崩溃。返回堆栈变量的地址会引起意料之外的程序行为，通常是程序崩溃。

解释	因为局部变量分配在堆栈上，所以当程序返回一个指向局部变量的指针时，它返回的是堆栈地址。随后的函数调用可能会重复使用同一堆栈地址，因而会覆盖指针的值，因为函数的堆栈框架在返回时已经失效，所以这个指针不再指向原来的变量。最好的情况这会使指针的值发生意外变更。在大多数情况下，这会导致程序在下一次间接引用该指针时发生崩溃。而且此类问题难以调试，因为引发问题的原因通常早已从症状中删除。
	例 1：以下函数会返回一个堆栈地址。
	char* getName() {
	char name[STR_MAX];
	fillInName(name);
	return name;
	}

建议	返回指针必须指向该堆才能保持有效。在大多数情况下，也就是指动态分配的内存，如下例所示。
	例 2:
	char* getName() {
	char* name = (char*) malloc(STR_MAX);
	fillInName(name);
	return name;
	}
	（确保内存由调用者正常释放，否则这会在程序中引入 memory leak。）
	在某些情况下，也可以返回一个指向静态变量的指针，如以下代码所示。这是可以接受的，因为静态变量是在堆上进行分配的。
	例 3:
	char* getName() {
	static char name[STR_MAX];
	fillInName(name);
	return name;
	}
	如果您返回一个指向静态变量的指针，请确保代码是线程安全的。

CWE	APSC-DV-002400 CAT II
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Macro Misuse**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数运行于共享资源上，且在部分平台上作为宏来执行，因此，应在 XX（函数） 中同时调用其搭档函数。对于在共享资源上运行且在部分平台上作为宏执行的函数系列，必须在相同的程序范围内进行调用。

解释	有些函数家族在有些平台上作为函数执行，而在其他平台上作为宏执行。如果函数依赖于某个内部维护的（而不是在调用时传入的）共享资源，则它们必须在同一程序范围内使用，否则会无法访问该共享资源。
	示例 1：以下代码使用 pthread_cleanup_push() 函数将 routine 函数推至调用线程清除堆栈之上，然后返回。由于 pthread_cleanup_push() 及其搭档函数 pthread_cleanup_pop() 在 IBM AIX 之外的平台上作为宏来执行，因此随后调用 pthread_cleanup_pop() 将无法访问 pthread_cleanup_push() 创建的数据结构。在将这些函数作为宏执行的平台上，该代码将无法编译，或者不能正确运行。
	void helper() {
	...
	pthread_cleanup_push (routine, arg);
	}

建议	请务必在相同的程序范围内调用 pthread_cleanup_push() 和 pthread_cleanup_pop()。
	示例 2：以下代码是经过重写的 Example 1 代码，用于在相同范围内加入对 pthread_cleanup_pop() 的相应调用。
	void helper() {
	...
	pthread_cleanup_push(routine, arg);
	pthread_cleanup_pop(1);
	}

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Memory Free on Stack Variable**
默认严重性	3.0
摘要	释放堆栈缓冲区可导致意外的程序行为。

解释	不要明确地取消分配堆栈内存。定义堆栈缓冲区的函数将在该函数返回时自动取消分配缓冲区。
	在这种情况下，将在 XX（文件） 中第 M 行上定义堆栈缓冲区，并在第 <Replace key="LastTraceLocation.line" /> 行上释放它。
	示例：
	void clean_up()
	{
	char tmp[256];
	...
	free(tmp);
	return;
	}
	明确地释放堆栈内存可能会损坏内存分配数据结构。这将导致程序异常终止或对数据造成更加严重的损坏。

建议	不要明确地释放堆栈内存，应使程序在函数返回时自动回收它。
	示例：
	void clean_up()
	{
	char tmp[256];
	...
	/* no memory management for tmp */
	return;
	}

CWE	APSC-DV-002400 CAT II
OWASP2017	None

返回目录

漏洞名称	**Code Correctness:Premature Thread Termination**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数早于其产生的线程结束执行，从而造成线程过早地终止。如果父进程早于其产生的线程结束执行，线程会过早终止。

解释	对于通过从父进程的 main() 函数调用 pthread_create() 而产生的线程，如果父进程没有调用 pthread_exit() 而早于任何线程结束执行，则这些线程会过早终止。调用 pthread_exit() 可保证在其所有线程执行完毕之前，父进程保持活动状态。此外，父进程可以调用所有子线程上的 pthread_join，并且确保它们将在进程结束之前完成。
	例 1：以下代码会使用 pthread_create() 函数创建一个线程，然后正常退出。如果子线程在 main() 函数返回时仍未结束执行，则该线程会被过早地终止。
	void *Simple(void *threadid)
	{
	...
	pthread_exit(NULL);
	}
	int main(int argc, char *argv[]) {
	int rc;
	pthread_t pt;
	rc = pthread_create(&pt, NULL, Simple, (void *)t);
	if (rc){
	exit(-1);
	}
	}

建议	当您从某个调用过 pthread_create() 的函数中退出之前，务必先调用 pthread_exit()。
	示例 2：以下代码是经过重写的 Example 1 代码，其中包括对 pthread_exit() 的调用。
	void *Simple(void *threadid)
	{
	...
	pthread_exit(NULL);
	}
	int main(int argc, char *argv[]) {
	int rc;
	pthread_t pt;
	rc = pthread_create(&pt, NULL, Simple, (void *)t);
	if (rc){
	exit(-1);

返回目录

	}
	pthread_exit(NULL);
	}

CWE	None
OWASP2017	None

返回目录

漏洞名称	**Command Injection**
默认严重性	4.0
摘要	X(文件) 中的 XX（函数） 函数通过调用第 N 行的 X(函数) 来执行利用不可信赖数据构建的命令。这会允许攻击者向应用程序中注入恶意命令。执行包含无效用户输入的命令，会导致应用程序以攻击者的名义执行操作。

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，既攻击者显式地控制了所执行的命令。这种类型的 Command Injection 漏洞会在以下情况下出现：
	1. 数据从不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据是字符串的一部分，应用程序将该字符串作为命令加以执行。
	在这种情况下，命令经由 Y（文件）的第 M 行的 Y（函数） 执行。
	3. 通过命令的执行，应用程序会授予攻击者一种原本不该拥有的特权或能力。
	示例 1：以下这个简单的程序将文件名作为命令行参数加以接受，并将文件的内容回显给用户。该程序是按照 setuid root 安装的，因为其最初的用途是一种学习工具，以便让那些仍在正在接受培训的系统管理员查看特权系统文件，而不授予其篡改权限或损坏系统的权力。
	int main(char* argc, char** argv) {
	char cmd[CMD_MAX] = "/usr/bin/cat ";
	strcat(cmd, argv[1]);
	system(cmd);
	}
	因为程序是利用 root 权限运行的，所以也会以 root 权限来调用 system()。如果用户指定了标准的文件名，那么调用就可按照您期望的方式进行。然而，如果攻击者传递了一个 ";rm -rf /" 形式的字符串，由于缺少参数，对 system() 的调用无法成功地执行 cat，然后程序会逐层删除根分区中的内容。
	示例 2：以下代码来自于一个特权程序，该程序使用环境变量 $APPHOME 来确定应用程序的安装目录，然后在该目录中执行一个初始化脚本。
	...
	char* home=getenv("APPHOME");
	char* cmd=(char*)malloc(strlen(home)+strlen(INITCMD));
	if (cmd) {

返回目录

```
        strcpy(cmd,home);
        strcat(cmd,INITCMD);
        execl(cmd, NULL);
    }
    ...
```

如 Example 1 中所示，该示例中的代码允许攻击者使用更高的应用程序权限来执行任意命令。在此示例中，攻击者可以篡改环境变量 $APPHOME 以指定包含恶意版本 INITCMD 的其他路径。由于程序不会验证从环境中读取的值，因此攻击者可通过控制该环境变量来诱骗应用程序去运行恶意代码。

因为攻击者使用环境变量来控制程序调用的命令，所以在本例中，环境的影响是不言而喻的。现在，我们转移一下注意力，看看如果攻击者能够改变命令的解析方式，会发生什么情况。

示例 3：以下代码来自一个基于 Web 的 CGI 实用程序，用户可以利用此实用程序修改其密码。通过 NIS 执行的密码更新过程包括在 /var/yp 目录中运行 make。请注意，由于程序更新了密码记录，因此它已按照 setuid root 安装。

程序会调用 make，如下所示：

```
system("cd /var/yp &amp;&amp;amp; make &amp;&amp;gt; /dev/null");
```

与上一个示例不同，因为本例中的命令采用硬编码形式，所以攻击者不能控制传输到 system() 的参数。但是，因为程序没有指定 make 的绝对路径，而且没有在调用命令之前清除任何环境变量，所以攻击者就能够篡改它们的 $PATH 变量，以便指向一个名为 make 的恶意二进制代码，并在 shell 提示符中执行 CGI 脚本。而且，因为程序已按照 setuid root 安装，所以攻击者的 make 目前会在 root 的权限下执行。

在 Windows 中，还存在其他风险。

示例 4：直接或通过调用 _spawn() 家族中的某项函数调用 CreateProcess() 时，如果可执行文件或路径中存在空格，必须谨慎操作。

```
...
LPTSTR cmdLine = _tcsdup(TEXT("C:\\Program Files\\MyApplication -L -S"));
CreateProcess(NULL, cmdLine, ...);
...
```

CreateProcess() 解析空格时，操作系统尝试执行的第一个可执行文件将是 Program.exe，而不是 MyApplication.exe。因此，如果攻击者能够在系统上安装名称为 Program.exe 的恶意应用程序，任何使用 Program Files 目录错误调用 CreateProcess() 的程序将运行此恶意应用程序，而非原本期望的应用程序。

环境在程序的系统命令执行中扮演了一个十分重要的角色。由于诸如 system()、exec() 和 CreateProcess() 之类的函数利用调用这些函数的程序的环境，因此攻击者有可能影响这些调用行为。

返回目录

建议	应当禁止用户直接控制由程序执行的命令。如果用户输入影响到命令的运行，那么请仅从一个预先决定的、安全的命令的集合中进行选择。如果输入中出现了恶意的内容，那么程序应当传递给函数一个默认的安全参数去执行，或者拒绝执行任何命令。 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，程序员往往又会退而采用执行拒绝列表方法，以便在使用输入之前，有选择性地拒绝或避免潜在的危险字符。但是，任何这样一个定义不安全字符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系统。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。 另外一个抵御恶意输入的防线是避免使用执行 shell 解析的函数。例如，不要使用 system()，该函数会执行它自己的命令 shell。 留意外部环境，看环境会对您所执行命令的行为产生何种影响。特别要注意 $PATH、$LD_LIBRARY_PATH 和 $IFS 变量在 Unix 和 Linux 机器中的使用方式。 请注意，Windows APIs 使用的是一个特殊的搜索顺序，它不仅仅基于一系列目录，而且还基于在没有特别指定时自动添加的文件扩展名列表。例如，对于 _spawn() 家族中的函数，如果命令名称参数没有文件扩展名或不以句号结束，那么它将尝试按照以下顺序执行文件扩展名：首先 .com，其次 .exe，然后 .bat，最后 .cmd。此外，由于执行命令时，函数将解析代表可执行文件和路径的参数中的空格，这使得 Windows 中存在其他风险。 示例 5：以下代码将重写 Example 4，以使用引号将可执行路径括起来，从而避免无意中执行恶意应用程序。 ... LPTSTR cmdLine[] = _tcsdup(TEXT("\"C:\\Program Files\\MyApplication\" -L -S")); CreateProcess(NULL, cmdLine, ...); ... 另一种方法也能达到同样的效果，即将可执行文件的名称作为第一个参数传递，而不是传递 NULL。 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	**Command Injection**
默认严重性	5
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。 攻击者可通过用户输入 XX（元素） 注入执行的命令，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，应用程序从用户输入接收数据，并将其作为字符串传递给操作系统。然后，该未经验证的数据由 OS 作为系统命令执行，使用与应用程序相同的系统权限。
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，请不要将用户输入与命令拼接。而要使用支持相关功能的平台函数将参数与命令分离。 不要调用 system() 或其变体，因为这不支持将数据参数与系统命令分离。 而要使用独立接收来自命令的参数并进行验证的函数之一。这包括 ShellExecute()、execve() 或其变体之一。 一定要将用户控制的数据作为 lpParameters 或 argN 参数（或等效值）传递给函数，并正确地进行引用。切勿将用户控制的数据作为 cmdname 或 filePath 的第一个参数传递。 不要使用用户控制的输入直接执行任何 shell 或命令解释器，例如 bash、cmd 或 make。
CWE	CWE ID 77
OWASP2017	A1-Injection

返回目录

漏洞名称	**Comparison Timing Attack**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证对比操作使用了 Y（文件） 文件第 M 行中 YY（元素） 的不安全比较操作符
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 在 hash 或密码之间使用非恒定时间比较可能会导致侧信道数据泄漏。有经验的攻击者可通过建立精心选择的身份验证尝试之间时间差异的统计模型来检索质询字符串。 用于存储凭证的方法不安全。 定期比较操作通常会针对性能进行优化。这些比较方法用于字符串时，这些比较操作会逐个对比两个字符串上的各个字符，一旦发现差异，这些比较操作会立即中断并返回 false 比较结果。这里的问题就是，攻击者如果可以控制其中一个比较项，就可以创建一个非预期的侧信道向量。我们假设要破解的挑战是 "AAA"。了解时间侧信道的攻击者可以发送 "BAA"，然后再发送 "ABA"，然后注意到返回比较结果所花费的平均时间略有增加，这仅仅是因为现在要在字符串上执行两次比较循环而不是只执行一次（第一个字符失败）。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛 (PHC) 获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性）
CWE	CWE ID 208
OWASP2017	None

返回目录

漏洞名称	**Connection String Injection**
默认严重性	5
摘要	应用程序的 YYY（方法） 方法收到不可信任的、用户控制的数据，并使用这些数据通过 Y（文件） 文件第 M 行的 YY（元素） 元素连接数据库。这可能导致“连接字符串注入”攻击。 攻击者可通过用户输入 XX（元素） 注入连接字符串，这会被应用程序在 X（文件） 文件第 N 行的 XXX（方法） 方法中检索到。
解释	如果攻击者可以操作应用程序到数据库服务器的连接字符串，则攻击者可能也能够执行以下任何操作: 破坏应用程序性能（通过增加 MIN POOL SIZE） 篡改网络连接（例如，通过受信任连接） 将应用程序指向攻击者的伪装数据库 （通过暴力攻击）获得数据库中的系统帐户密码。 为了与数据库或其他外部服务器（例如 Active Directory）通信，应用程序会动态构造连接字符串。此连接字符串中包含不可信任的数据，此漏洞可能被恶意用户利用。因为此不可信任的数据未经过限制或适当净化，可能被恶意利用来操纵连接字符串。
建议	无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项: 数据类型 大小 范围 格式 预期值 不允许用户控制数据库连接字符串。避免根据不可信任的数据（特别是用户输入）来动态创建连接字符串。 以合适的配置机制来保存所有的连接字符串。如果 runtime 时需要动态构建连接字符串，请不要直接在连接字符串中添加不可信任的数据；而要让用户从预定义好的连接字符串中选择。
CWE	CWE ID 99
OWASP2017	A1-Injection

返回目录

漏洞名称	**Creation of chroot Jail without Changing Working Directory**
默认严重性	3
摘要	XXX（方法） 函数调用 X（文件） 文件第 N 行的 XX（元素） 创建 chroot jail。但是，该函数没有显式将工作目录更改为 chroot jail。
解释	应用程序经常使用 chroot()或 fchroot()将程序限制为只能查看文件系统中的特定目录树。但是，如果程序的工作目录在这个根目录限制之外，就有很多方法可以逃避 "chroot jail"。如果未显式将工作目录更改到 chroot，恶意程序或用户就能继续访问 chroot 之外的文件或目录。 大部分 Unix/Linux 系统不会自动更改当前的工作目录，调用 chroot 时，依靠调用应用程序（或系统管理员）显式调用 chdir()/fchdir() 将工作目录更改为新创建的 chroot jail。但是，应用程序在调用 chroot() 之前不会更改工作目录。 例如，如果当前工作目录在已进行 chroot 限制的目录树之外，则恶意用户可以使程序改变目录，例如，通过重复调用 chdir("..")，直至到达真正的 root 目录。完成此操作后，程序可以根据需要访问其他任意目录，不受 chroot 限制的影响。
建议	通用指南： 确保应用程序正确地使用 chroot 更改，包括显式更改工作目录。 建议在调用 chroot() 之前更改工作目录，以便最大程度减少在进行 chroot 约束前逃逸 chroot jail 约束的风险（例如，通过在调用之间抛出错误，或复杂的代码流）。 确保应用程序在调用 chroot() 后调用 setuid() 降低进程权限。 移动到 chroot 环境后关闭并释放所有打开的文件描述符。 不要依赖 chroot 提供安全边界，也不要依赖 chroot 提供完全安全的沙盒。可以考虑更现代化的机制，例如 BSD jail 或虚拟化容器。 注意还有一些其他问题可能使用户或进程逃出 chroot jail，包括硬链接，以及将工作目录移到 chroot 之外。 具体建议： 在调用 chroot() 前，先在要进行 chroot 的目录中调用 chdir()。
CWE	CWE ID 243
OWASP2017	A6-Security Misconfiguration

返回目录

漏洞名称	**Dangerous Function**
默认严重性	5.0
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	None
OWASP2017	None

返回目录

漏洞名称	**Dangerous Function:strcpy()**
默认严重性	2.0
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	**Dangerous Functions**
默认严重性	4
摘要	Y（文件） 文件第 M 行中发现使用了危险函数 YY（元素）。此类函数可能会泄露信息并使攻击者可以完全控制主机。
解释	使用危险函数可能导致与各个特定函数相关的各种风险，不正确地使用这些函数可能产生各种严重的潜在影响。存在这些函数说明代码维护策略和没有遵守安全编码实践，会将已知的危险代码引入应用程序。 代码中发现危险函数。函数被认为危险的原因有很多，这是因为与各种函数使用相关的漏洞集合是不同的。例如，某些字符串复制和拼接函数容易受到缓冲区溢出、内存泄露、拒绝服务等攻击。因此不建议使用这些函数。
建议	使用推荐的安全方式替代被认为有危险的函数。 如果没有安全的替代方案，可通过进一步的研究和测试确定当前用法能否成功地净化和验证这些值，从而成功避免确实有危险的函数用例定期审查使用的方法，确保所有外部库和内置函数都是最新的，并且其用法被纳入安全编码实践中。
CWE	CWE ID 242
OWASP2017	A9-Using Components with Known Vulnerabilities

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	**Dead Code**
默认严重性	2.0
摘要	永远不会执行 X(文件) 的第 N 行。它是一个 dead code。永远不会执行该指令。

解释	周围的代码使该指令永远不会被执行。
	示例：第二个 if 指令的条件不可能得到满足。这需要变量 s 为非 null 变量，且仅在可以将 s 指定为非 null 值的路径时，存在 return 指令。
	String s = null;
	if (b) {
	s = "Yes";
	return;
	}
	if (s != null) {
	Dead();
	}

建议	总之，您应该去修改或是删除未使用的代码。它不仅不能实现任何程序功能，还会带来额外的麻烦和维修负担。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Denial of Service**
默认严重性	3.0
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。

解释	攻击者可能通过对应用程序发送大量请求，而使它拒绝对合法用户的服务，但是这种攻击形式经常会在网络层就被排除掉了。更加严重的是那些只需要使用少量请求就可以使得攻击者让应用程序过载的 bug。这种 bug 允许攻击者去指定请求使用系统资源的数量，或者是持续使用这些系统资源的时间。
	示例 1：以下代码允许用户指定当前进程处于休眠状态的时长。通过指定一个较大的数字，攻击者便可以无限期地阻碍进程。
	unsigned int usrSleepTime = uatoi(usrInput);
	sleep(usrSleepTime);

建议	校验用户输入以确保它不会引起不适当的资源利用。
	例 2：以下代码允许用户指定线程进入休眠的时间，正如上面的例子一样，但仅当数值在合理的范围之内时才会有效。
	unsigned int usrSleepTime = uatoi(usrInput);
	if (usrSleepTime >= SLEEP_MIN &&
	usrSleepTime <= SLEEP_MAX) {
	sleep(usrSleepTime);
	} else {
	return -1; // Invalid sleep duration
	}

CWE	CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Deprecated CRT Functions VS2005**
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	**Directory Restriction**
默认严重性	3.0
摘要	对 chroot() 系统调用的使用不当会让攻击者从 chroot 监牢中逃脱出来。

解释	chroot() 系统调用允许程序修改其 file system 根目录的含义。适当地调用 chroot() 后，程序无法访问在新的根目录下定义的目录树之外的任何文件。这样的环境称为 chroot 监牢，通常用来防止攻击者破坏进程，继而访问未经授权的文件。例如，在 chroot 监牢环境中运行多个 FTP 服务器，可以防止攻击者发现新服务器漏洞后下载密码文件或者其他系统中的敏感文件。
	对 chroot() 的使用不当可能会让攻击者从 chroot 监牢中逃脱出来。因为 chroot() 函数的调用不会改变进程当前的工作目录，所以在调用 chroot() 之后，相对路径可能仍然会引用 chroot 监牢之外的 file system 资源。
	例 1：考虑以下这段来自（假设的）FTP 服务器的源代码：
	chroot("/var/ftproot");
	...
	fgets(filename, sizeof(filename), network);
	localfile = fopen(filename, "r");
	while ((len = fread(buf, 1, sizeof(buf), localfile)) != EOF) {
	fwrite(buf, 1, sizeof(buf), network);
	}
	fclose(localfile);
	这段代码负责从网络中读取文件名，然后将相应的文件在本地机器上打开，并通过网络传送内容。这段代码可用来执行 FTP GET 命令。FTP 服务器在其初始化例程中调用 chroot()，试图阻止对 /var/ftproot 之外的文件的访问。但因为服务器没有通过调用 chdir("/") 来更改当前的工作目录，所以攻击者可以请求文件的 "../../../../../etc/passwd"，并获取该系统密码文件的副本。

建议	确保每一个 chroot() 的调用后都紧跟着 chdir("/") 调用。验证 chroot() 和 chdir() 调用之间的处理错误的代码，不会打开任何文件，也不会返回对程序任何部分的控制权，关闭例程除外。chroot() 和 chdir() 命令应该尽可能一起执行，以便降低这种漏洞发生的可能性。同时，也可以首先通过调用 chdir() 创建一个合适的 chroot 监牢：
	chdir(newRoot);
	chroot(newRoot);
	但是我们更希望首先看到 chroot()。这样，调用 chdir() 就可以采用一个常量参数 ("/")，该参数既易于获取，又便于在代码复查中进行校验。

CWE	None

返回目录

OWASP2017	None

返回目录

漏洞名称	**Divide By Zero**
默认严重性	4
摘要	应用程序在 Y（文件） 中的 YYY（方法） 中执行了非法操作。在第 M 行中，程序尝试以 YY（元素） 相除，这可能在相除时等于 0（零）。此值可能是硬编码的零值，也可能是从 X（文件） 文件第 N 行的 XXX（方法） 中获得的外部、不可信任的输入 XX（元素）。
解释	程序将数字除以零时，会引发异常。如果应用程序未处理此异常，就可能会出现意外结果，包括应用程序崩溃。如果外部用户可以控制分母的值或者可以导致发生此错误，就可以将此视为 DoS（拒绝服务）攻击。 程序收到意外值，并且未进行过滤、验证或检查该值不为零便用于除法中。应用程序未显式处理此错误或避免发生除以零的情况。
建议	在除以未知值之前，先验证该数字并显式确认其不等于零。 验证来自所有来源的所有不可信任的输入，特别是在做除法之前要验证它不等于零。 验证方法、计算、字典查找等的输出，并在除结果之前确保它等于零。 确保除以零的错误能被捕捉到并正确处理。
CWE	CWE ID 369
OWASP2017	None

返回目录

漏洞名称	**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	**Double Free**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数会释放相同的内存地址两次。在同一个内存地址上两次调用 free()，会引发 buffer overflow。

解释	当同一内存地址被当作参数不止一次地调用 free() 时，会出现 Double free 错误。
	在这种情况下，该内存第一次在 XX（文件）中的第 M 行释放，而第二次在 X(文件) 中的第 N 行释放。
	针对同一个值两次调用 free()，会导致 buffer overflow。当程序使用同一参数两次调用 free() 时，程序中的内存管理数据结构会遭到破坏。这种破坏会导致程序崩溃。有时在某些情况下，还会导致两次调用 malloc() 延迟，而返回相同的指针。如果 malloc() 两次都返回同一个值，稍候程序便会允许攻击者控制整个已经写入双倍分配内存的数据，从而使程序更加容易受到 buffer overflow 的攻击。
	例 1：以下代码显示了一个关于 double free 漏洞的简单例子。
	char* ptr = (char*)malloc (SIZE);
	...
	if (abrt) {
	free(ptr);
	}
	...
	free(ptr);
	Double free 漏洞的产生有两个常见（有时候这两个原因会同时发生作用）原因：
	– 错误状况及其他异常情况。
	— 不清楚由程序的哪一部分负责释放内存。
	虽然某些 double free 漏洞并不比上一个例子复杂多少，但是大部分漏洞都分散在上百行代码中，甚至还会出现在不同的文件中。程序员似乎特别容易受到多次释放全程变量的影响。

建议	通过在调用 free() 后消除所有对内存的引用，您就能预防再次释放相同的内存了。通常情况下，这可以通过替换调用函数达到目的，例如，将 free() 替换为一个在其内存释放后立即将 NULL 分配到指针的宏： ... #define FREE(ptr) {free(ptr); ptr = NULL;} ... 如果您选择实施一个要求在指针被释放后为其分配 NULL 的策略，那么您可以通过以下方法检查是否违反了这一策略：编写一个自定义的控制流规则，用来标记没有紧接着执行此分配的任何 free() 调用。有关创建自定义规则的信息，请参阅《Fortify Static Code Analyzer 自定义规则指南》。

返回目录

CWE	CCI-002824
OWASP2017	None

返回目录

漏洞名称	**Download of Code Without Integrity Check**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法载入并执行了外部代码，但未运行完整性检查，使其容易受到攻击。
解释	最好的情况是，未通过完整性检查的代码也可能已损坏、被更改或者与预期代码不一致，从而导致意外行为。最坏的情况是，攻击者如果能影响加载的代码（例如在本地或通过中间人攻击），就可能在存储中或传输过程中更改加载的代码，这可能导致执行恶意代码和系统严重受损。 从数据集派生的完整性签名使接收者可以确定接收的数据集是接收者想要的数据集。这对于外部加载的代码特别重要，因为这些代码可能会因为本地存储的访问、中间人攻击等而受到损害。
建议	通过严格的完整性检查确保从外部来源获取的代码能得到验证，并有已知且受信任的签名。
CWE	CWE ID 494
OWASP2017	A1-Injection

返回目录

漏洞名称	**Encoding Used Instead of Encryption**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的敏感数据是使用不安全的机制保存在 Y（文件） 文件第 M 行的 YY（元素） 中
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 处理敏感数据时，一定要为其加密：无论是永久存储还是传输都应如此。 不加密个人身份信息可能会导致违反合规性要求。另外攻击者也会收集这些数据以泄露这些信息或将其用于恶意目的。 应用程序的加密相关功能有问题。 应用程序只是对敏感数据进行模糊处理（编码），而未使用加密来保护其存储或传输。虽然编码函数的输出表面上看是“随机的”，但是攻击者不需要任何密钥相关的知识即可解码。如果攻击者知道编码数据的算法，攻击者就可以使用其类似的解码函数来获取人类可读的版本。这也被称为 security-by-obscurity，是众所周知的弱安全形式。
建议	如果必须保存敏感数据（无论是在客户端还是服务器上），则应以加密的方式存储，而不能使用纯文本。 这些数据的加密密钥应以安全的方式使用用户密码派生，不应与数据保存到一起或硬编码到应用程序的代码中。
CWE	CWE ID 311
OWASP2017	None

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	**Exposure of System Data to Unauthorized Control Sphere**
默认严重性	3
摘要	X（文件） 文件第 N 行中的 XXX（方法） 读取的系统数据可能会被 Y（文件） 文件第 M 行中存在的 YYY（方法） 暴露。
解释	系统数据可以为攻击者提供与其设定为目标的系统和服务相关的重要分析数据——任意类型的系统数据（从服务版本到操作系统指纹）都可以帮助攻击者磨练他们的攻击、将数据与已知漏洞关联，或者集中精力开发针对特定技术的新攻击。 系统数据被读取后会暴露，然后可能被不可信任的实体读取。
建议	考虑暴露指定输入的影响，以及对指定输出的预期访问级别。如果不需要，可考虑删除此代码，或修改暴露的信息以排除潜在的敏感系统数据。
CWE	CWE ID 497
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Float Overflow**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的值未经验证便被用于 Y（文件） 文件第 M 行 YY（元素） 中的算术运算中，这可能导致算术溢流（通常称为“整数溢出”）。
解释	算术溢出会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断或丢失数据；使用溢出的数值执行内存操作时，甚至导致缓冲区溢出，从而导致执行任意代码。 所有数字数据类型都按位表示。如果在算术运算后，某个值超过了其按位表示的位数，那么被加的最重要的数字将被截断，此截断后剩余的值将是回绕操作的剩余值——这称为“算术溢出”，另常误称为——“整数溢出”，此误称是有误导的，因为此名称还适用于许多其他类型。如果可能溢出的数据类型低于其最小值，则它们将负向回绕，通常称为下溢。 例如，如果无符号的 32 位整数设置为 4,294,967,295，然后加 1，它就会溢出并回绕到 0。如果有符号的 32 位整数值为 2,147,483,647，然后加 1，则它会溢出并回绕到 -2,147,483,648。 为了保证代码正确，一定要检查值是否在预期范围内，确保算术运算结果不会溢出或下溢。
建议	对可能包含任意值的数据执行算术运算时，请考虑添加一个检查以保证数据在范围内，且这些操作的结果不会导致溢出或下溢。 考虑为所有算术运算创建封装器，以对特殊情况进行特定处理；例如，如果检查显示已经或将要发生溢出，则抛出异常。
CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**Format String**
默认严重性	4.0
摘要	攻击者可以控制 X(文件) 中第 N 行的 X(函数) 的 format string 参数，发起类似于缓冲区溢出的攻击。允许攻击者控制函数的 format string 会导致 buffer overflow。

解释	Format string 漏洞会在以下情况下发生：
	1. 数据从不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据作为 format string 参数传送到某个函数，如 sprintf()、FormatMessageW() 或 syslog()。
	在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。
	示例 1：以下代码使用 snprintf() 将一个命令行参数复制到缓冲区中。
	int main(int argc, char **argv){
	char buf[128];
	...
	snprintf(buf,128,argv[1]);
	}
	该代码允许攻击者查看堆栈的内容，并使用包含一连串的格式化指令的命令行参数对堆栈进行写入。攻击者能够通过提供更多的格式化指令（如 %x）来读取堆栈中的内容，然后函数会作为即将格式化的参数使用。（在本例中，函数没有采用任何即将格式化的参数。）通过使用 %n 格式化指令，攻击者能够对堆栈进行写入，进而使 snprintf() 记下迄今为止输出的字节数，并将其传送给指定的参数（而不是直接从参数中读取数值，这是程序员最初设计的行为）。对于这种攻击，更为的复杂的形式是使用四条交错的写入来完全控制堆栈中某个指针的值。
	示例 2：有些实现方式通过提供格式化指令来控制内存中的读写位置，这样可以简化较高级的攻击。以下代码就是这种指令的一个例子，它为 glibc 而写：
	printf("%d %d %1$d %1$d\n", 5, 9);
	该代码产生了如下输出：
	5 9 5 5
	此外，还可以使用 half-writes (%hn) 来准确地控制内存中的任意 DWORDS，这会大大地降低执行攻击的难度，否则此攻击将需要进行四次交错写入，如 Example 1 中所述。
	示例 3：简单的 format string 漏洞通常由表面上看似没有什么危险的快捷方式所导致。某些快捷方式的使用已经深入人心，以致于程序员可能都不会意识到他们所使用的函数需要一个 format string 参数。例如，syslog() 函数有时候可以这样使用：

返回目录

	...
	syslog(LOG_ERR, cmdBuf);
	...
	由于 syslog() 的第二个参数是格式字符串，因此 cmdBuf 中的任何格式化指令都会按照 Example 1 中所述进行解释。
	以下代码显示了 syslog() 的正确使用方式：
	...
	syslog(LOG_ERR, "%s", cmdBuf);
	...

建议	不管什么时候，只要可以，请将静态的 format string 传递到那些接受 format string 参数的函数。如果必须动态构造 format string，则可定义一组有效的 format string，并从这组安全的 format string 进行选择。最后，请始终要进行如下校验：在所选 format string 中的格式化指示的数量符合即将进行格式化的参数的数量。
CWE	CWE ID 134
OWASP2017	None

返回目录

漏洞名称	**Format String Attack**
默认严重性	5
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从用户输入接收 XX（元素） 值。然后该值被用于构建"格式字符串"YY（元素），并用作 Y（文件） 文件第 M 行 YYY（方法） 方法的字符串格式参数。
解释	在非托管内存环境中，允许攻击者控制格式字符串会使攻击者能够访问攻击者不该访问的内存区域，包括读取其他受限变量、歪曲数据，甚至覆写无访问权限的内存区域。在特定情况下，这甚至可能进一步导致缓冲区溢出和执行任意代码。 应用程序允许用户输入影响用于已格式化打印函数的字符串实参。这个函数族期望第一个实参指定动态构造的输出字符串的相对格式，包括如何表示其他各个实参。 允许外部用户或攻击者控制此字符串，允许他们控制打印函数的功能，从而访问不该访问的内存区域。
建议	通用指南： 不要让用户输入或任何其他外部数据影响格式字符串。 确保使用静态字符串作为格式参数调用所有字符串格式函数，并根据静态格式字符串将正确数量的实参传递给函数。 也可在格式字符串参数中使用所有用户输入来打印格式函数之前，对所有用户输入进行验证，并确保输入中不包含格式化 token。 具体建议： 不要直接在格式化函数的格式字符串参数（通常是第一个或第二个实参）中使用用户输入。 也可在格式字符串中使用从输入导出的已控信息，例如大小或长度——但不能使用输入本身的实际内容。
CWE	CWE ID 134
OWASP2017	A1-Injection

返回目录

漏洞名称	**Format String:Argument Number Mismatch**
默认严重性	3.0
摘要	X(文件) 第 N 行中 X(函数) 的 format string 参数使用了一个结构不良的 format string，其中包含的转换说明符的数量与函数具有的参数数量不一致。不正确的 format string 会导致该程序从分配的内存边界之外读取数据，这可以让攻击者访问敏感信息、引入错误行为，或造成程序崩溃。 该程序使用了一个结构不良的 format string，其中包含的转换说明符的数量与函数具有的参数数量不一致。不正确的 format string 会导致该程序从分配的内存边界之外读取数据，这

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这种情况下，结构不良的 format string 会导致程序访问分配的内存边界之外的值。
	示例：以下代码会从堆栈中读取任意值，因为格式说明符的数量与传递给该函数的参数数量不一致。
	void wrongNumberArgs(char *s, float f, int d) {
	char buf[1024];
	sprintf(buf, "Wrong number of %.512s");
	}

返回目录

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Format String:Argument Type Mismatch**
默认严重性	3.0
摘要	X(文件) 第 N 行中 X(函数) 的 format string 参数使用了一个结构不良的 format string，其中包含的转换说明符与传递给该函数的参数类型不一致。不正确的 format string 会导致该程序转换值时出错，还可能会在分配的内存边界之外读写数据，这会引入错误的行为或造成程序崩溃。 该程序使用了一个结构不良的 format string，其中包含的转换说明符与传递给该函数的参数类型不一致。不正确的 format string 会导致该程序转换值时出错，还可能会在分配的内存边界

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这里，一个不正确地构成的 format string 会导致程序错误地进行值的转换，或访问分配的内存边界之外的值。
	示例：以下代码使用 %d 格式说明符将一个浮点转换为 f。
	void ArgTypeMismatch(float f, int d, char *s, wchar *ws) {
	char buf[1024];
	sprintf(buf, "Wrong type of %d", f);
	...
	}

返回目录

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

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

返回目录

OWASP2017	None

返回目录

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

返回目录

漏洞名称	**Hashing Length Extension Attack**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的摘要容易受到长度扩展攻击。
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 Hash 函数在加密方面有很多种用途，但大部分都与密钥派生函数和签名有关。如果以不安全的方式使用 hash 函数，就会影响密码 hash 或签名的完整性和机密性。 创建消息身份验证代码 (MAC) 时，如果应用程序使用直接 hash 并且所选的 hash 原语容易受到长度扩展攻击，则攻击者可以在保持有效 MAC 的同时将数据追加到捕获的消息中，从而有效地伪造消息签名。 应用程序的加密相关功能有问题。 应用程序使用无法防御长度扩展攻击的直接 hash 创建消息身份验证代码或签名。
建议	如果应用程序需要签名或验证数据，方法之一是使用消息验证代码。为了避免长度扩展攻击和其他漏洞，建议使用 HMAC 算法。BLAKE2b 等一些现代 hash 算法可以安全地抵御长度扩展攻击，并且已有键控 hash 的本机支持，因此这些算法是 HMAC 的有效替代方案。
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Heap Inspection**
默认严重性	2.0
摘要	不要使用 realloc() 来调整储存敏感信息的缓冲区大小。因为该函数可能会将敏感信息的副本遗留在内存中，而您又无法对这部分内存进行覆盖。

解释	当敏感数据（如密码或加密密钥）因为没有从内存中删除而被泄漏给攻击者时，就会发生 Heap inspection 漏洞。
	realloc() 函数通常用来提高某个分配的内存块的大小。该操作经常要求将以前的内存块中的内容复制到新的且更大的内存块中。该操作会完整地保留原内存块中的内容，但不允许程序进行访问，这就使程序无法清除内存中的敏感数据。如果攻击者能够后期访问某个内存转储中的内容，敏感数据就会暴露。
	在这种情况下，X(文件)的第 N 行调用 X（函数）。
	示例：以下代码在一个包含敏感数据的缓冲区上调用了 realloc()：
	plaintext_buffer = get_secret();
	...
	plaintext_buffer = realloc(plaintext_buffer, 1024);
	...
	scrub_memory(plaintext_buffer, 1024);
	这段代码试图清除内存中敏感数据，但因为使用了 realloc()，所以在原来对 plaintext_buffer 分配的内存中仍会看到该数据的副本。

建议	如果敏感的信息储存在内存中，就会出现这样的风险：攻击者可以访问内存转储，将 realloc() 替换为对 malloc()、memcpy() 及 free() 的直接调用。该方法能够在原始缓冲区释放空间之前安全地清除其中的内容。
CWE	CWE ID 244
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Heap Inspection**
默认严重性	3
摘要	X（文件） 文件第 N 行上的 XXX（方法） 方法定义了 XX（元素），此元素被指定要包含用户密码。但是，虽然这个纯文本密码之后被分配给 XX（元素），但内存中的这个变量不会被清除。
解释	未授权的用户只要有机器的访问权限，就可以检索应用程序在未加密的内存中存储的所有变量。例如，有权限的攻击者可能将调试器附加到正在运行的进程，或者从交换文件或崩溃转储文件中检索进程的内存。攻击者找到内存中的密码后，用可以再次用其假冒用户访问系统。 字符串变量是不可变的——也就是说，字符串变量被赋值后，就无法再更改或删除其值。因此，这些字符串可能会一直留在内存中，而且可能保存在多个位置，直到垃圾回收器将其删除。敏感数据（如密码）会明文暴露在内存中，无法控制其生命周期。 虽然此时仍然可以检索内存中的数据，即使它使用了已清除的可变容器，或者从内存检索解密密钥并解密敏感数据——使用这些保护类型对敏感数据进行分层将大幅增加所需的工作量。通过为内存检索敏感数据设置高标准，减少内存中敏感数据的量和暴露时间，可以大幅降低攻击者成功获取有价值数据的可能性。
建议	在避免堆检查方面需要指出的是，如果提供了对内存或应用程序内存转储的任何读取权限，就有可能为攻击者提供一些敏感数据——在能成功获得此类内存读取权限的情况下，这些建议是一些用于保护敏感数据的深度防护原则。这些建议可大幅缩短敏感数据在内存中的暴露生命周期；但是，如果有足够的时间、努力和无限制的内存访问权限，这些建议也只能保护应用程序使用的敏感数据。处理堆检测问题的唯一方法是尽量减少数据暴露，并尽可能地将其隐藏在内存中。 不要将敏感数据（如密码或密钥）以明文形式存储在内存中，即使是短时间，也不要这样做。 建议使用在内存中存储加密数据的专用类，以确保不能随意在内存中检索。 当需要以原始形式使用敏感数据，将其临时存储在可变数据类型（如字节数组）中，以降低在内存中的可读性，然后立即将内存位置归零，以减少此数据在内存中的暴露时间。 确保不与不信任方交换内存转储，因为即使能保证以上所有内容——仍然可以对加密容器进行反向工程，或从内存中检索敏感数据字节并重建它。 具体建议——C++： 将密码保存在内存中时，将变量标记为易失性，以防止不必要的优化。
CWE	CWE ID 244

返回目录

OWASP2017	None

返回目录

漏洞名称	**Heap Inspection:Swappable Memory**
默认严重性	2.0
摘要	不要使用 VirtualLock 来锁定包含敏感数据的页面。因为该函数并不总会执行。
解释	当敏感数据（如密码或加密密钥）因为没有从内存中删除而被泄漏给攻击者时，就会发生 Heap inspection 漏洞。 VirtualLock 函数旨在锁定内存中的页面，以防这些页面在磁盘上被标注页码。然而，在 Windows 95/98/ME 操作系统上，这一函数仅作为桩函数使用，因而不会产生什么影响。 在这种情况下，X(文件)的第 N 行调用 X（函数）。
建议	如果将敏感信息储存于内存中，便会存在攻击者访问内存转储的风险，因此，请尽可能缩短数据存储在内存中的时间，并在不需要时删除这些数据。对于安全级别要求较高的系统，可以考虑拒绝运行某些旧版 Microsoft 平台，因为这些平台没有为储存在内存中的数据提供相应的安全机制。
CWE	CWE ID 591
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Heuristic 2nd Order Buffer Overflow malloc**
默认严重性	3
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Heuristic 2nd Order Buffer Overflow read**
默认严重性	3
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Heuristic 2nd Order SQL Injection**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取数据库数据。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	**Heuristic Buffer Improper Index Access**
默认严重性	3
摘要	X（文件） 文件第 N 行的数组索引 XX（元素） 被用于引用 Y（文件） 文件第 M 行的数组 YY（元素） 单元的索引。
解释	数组的元素通常会映射一个数值，表示元素在数组中的位置。如果无法保证用于从数组中读取元素的索引值在数组的范围内，会导致引用超出数组边界。
建议	将值用作数组索引之前，一定要检查值是否非负且小于数组大小。
CWE	CWE ID 129
OWASP2017	None

返回目录

漏洞名称	**Heuristic Buffer Overflow malloc**
默认严重性	3
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Heuristic Buffer Overflow read**
默认严重性	3
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Heuristic CGI Stored XSS**
默认严重性	3
摘要	X（文件） 文件第 N 行中发现未经验证的数据库输出。第 M 行的 YY（元素） 中发现可能有 XSS 漏洞。
解释	保存的恶意数据可能检索系统信息并通过 CGI（通用网关接口）利用系统。 CGI 规范提供了在服务器计算机及其连接的主机上读取文件、获取 shell 访问权限和损坏文件系统的可能性。 获取访问权限的方法有：利用脚本的假设，利用服务器环境中的弱点，以及利用其他程序和系统调用中的弱点。 CGI 脚本的主要弱点是输入验证不充分。
建议	不要提供不必要的文件权限。 验证并编码所有数据库输出。
CWE	CWE ID 79
OWASP2017	A7-Cross-Site Scripting (XSS)

返回目录

漏洞名称	**Heuristic DB Parameter Tampering**
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	**Heuristic NULL Pointer Dereference1**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 中声明的变量在被 Y（文件）文件第 M 行的 YY（元素） 使用时未初始化。
解释	Null 指针取消引用可能会导致 run-time 异常、崩溃或其他意外行为。 已声明但未赋值的变量将隐式保留一个 null 值，直到为变量赋一个值。也可显式为变量设置 null 值，以确保清除其内容。因为 null 并不是一个真正的值，它可能没有对象变量和方法，因此访问 null 对象的内容但未提前验证它的设置时会导致 null 指针取消引用异常。
建议	创建任何变量时，请确保声明和使用之间的所有逻辑流都先为变量分配非 null 值。 在取消引用之前对接收到的所有变量和对象强制执行 null 值检查，确保其不包含其他地方分配给的 null 值。 可考虑是否通过分配 null 值覆盖初始化的变量。可考虑重新分配或释放这些变量。
CWE	CWE ID 476
OWASP2017	None

返回目录

漏洞名称	**Heuristic NULL Pointer Dereference2**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 中声明的变量在被 Y（文件）文件第 M 行的 YY（元素） 使用时未初始化。
解释	Null 指针取消引用可能会导致 run-time 异常、崩溃或其他意外行为。 已声明但未赋值的变量将隐式保留一个 null 值，直到为变量赋一个值。也可显式为变量设置 null 值，以确保清除其内容。因为 null 并不是一个真正的值，它可能没有对象变量和方法，因此访问 null 对象的内容但未提前验证它的设置时会导致 null 指针取消引用异常。
建议	创建任何变量时，请确保声明和使用之间的所有逻辑流都先为变量分配非 null 值。 在取消引用之前对接收到的所有变量和对象强制执行 null 值检查，确保其不包含其他地方分配给的 null 值。 可考虑是否通过分配 null 值覆盖初始化的变量。可考虑重新分配或释放这些变量。
CWE	CWE ID 476
OWASP2017	None

返回目录

漏洞名称	**Heuristic Parameter Tampering**
默认严重性	3
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	**Heuristic SQL Injection**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行 YYY（方法） 方法中用于数据库查询。这可能导致 SQL 注入攻击。
解释	攻击者可能直接访问系统的所有数据。攻击者使用简单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。
建议	验证所有来源的输入。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。 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	**Heuristic Unchecked Return Value**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法调用 XX（元素） 函数。但是，代码未检查此函数的返回值，因此无法检测 runtime 错误或其他意外状态。
解释	不检查函数返回值的程序可能会导致应用程序进入未定义的状态。这可能导致意外行为和意外后果，包括数据不一致、系统崩溃或其他根据错误发起的攻击。 应用程序调用了一个函数，但未检查函数的返回值的结果。应用程序仅忽略了此结果值，首选认为此结果值是正确的和需要的并使用此结果值或传递此结果值。
建议	一定要检查所有返回值的被调用函数的结果，并验证结果是否为预期值。 确保调用函数响应所有可能的返回值。 预测 runtime 错误并正确地处理它们。显式定义处理意外错误的机制。
CWE	CWE ID 252
OWASP2017	None

返回目录

漏洞名称	**Illegal Pointer Value**
默认严重性	3.0
摘要	调用第 N 行的 X(函数) 会返回一个指向内存的指针，该内存不在搜索的缓冲区范围以内。该指针的并发操作可能会导致意料之外的结果。该函数会返回一个指向内存的指针，而该内存不在搜索的缓冲区范围以内。该指针的并发操作可能会导致意料之外的结果。
解释	
建议	
CWE	CWE ID 466
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 Null Termination**
默认严重性	5
摘要	X（文件） 文件第 N 行中的缓冲区 XX（元素） 没有 null 终止符。将此缓冲区视为 null 结尾字符串的任何后续操作都会导致意外或危险的行为。
解释	如果没有终止 null 字节，则对预期使用 null 字节结尾的字符串进行检查会导致大小估计错误。读取没有 null 终止符的字符串时，会读取字符串设置范围外的内存，从而可能会泄露敏感信息。将该字符串写入内存时，攻击者可能会尝试在字符串的初始分配之外写入任意数据，从而导致缓冲区溢出，如果这受到控制，可能会导致执行远程代码。 通常在 C 和 C++ 中，字符串是作为字符数组而不是更复杂的数据结构处理的。因此，为了确定 char 数组在堆栈或堆上的结束位置，使用了 null 字节作为参考点。例如，strlen 将返回字符数组中 null 字节的位置——如果没有空字节，该行为就是未定义的，就可能返回内存中的下一个 null 字节的位置或导致访问冲突。
建议	处理字符串时，可考虑使用比 char* 更现代的类型，例如 string，以避免 null 终止问题。 如果使用 char 数组包含字符串，一定要将字符串的最后一个字节设置为 null 字节，以确保字符串以 null 值终止。处理 null 终止的字符串时，一定要在设置此 null 字节之前确保缓冲区有足够的空间，方法是使目标缓冲区比源字符串长一个字节，并将最终字节设置为空 null (\0)。
CWE	CWE ID 170
OWASP2017	None

返回目录

漏洞名称	**Improper Resource Shutdown or Release**
默认严重性	3
摘要	X（文件） 中应用程序的 XXX（方法） 方法定义并初始化了 N 的 XX（元素） 对象。此对象封装了有限的计算资源，例如打开文件流、数据库连接或网络流。此资源不能在所有情况下都正确地关闭和释放。
解释	不释放资源可能导致系统资源耗尽，最终导致一般可靠性和可用性问题，例如性能下降、流程膨胀和系统不稳定。如果资源泄漏被攻击者有意利用，就会造成广泛的 DoS（拒绝服务）攻击。如果资源继续在后续分配中保留数据或用户 ID，这甚至可能会向无权限用户暴露敏感信息。 应用程序代码分配资源对象，但不能保证关闭和释放这些资源对象。这包括数据库连接、文件句柄、网络套接字或任何其他需要释放的资源。在某些情况下，这些可能会被自动释放——但前提是一切都按计划进行；如果在系统运行期间发生任何 runtime 异常，资源就会开始泄漏。 请注意，即使在 Java 等托管内存语言中，这些资源也必须显式释放。很多类型的资源即使在垃圾收集器运行时也不释放；即使对象最终释放资源，我们也无法控制垃圾收集器何时运行。
建议	一定要关闭并释放所有资源。 确保使用 finally{} 块释放资源（以及任何其他必要的清理）。不要使用 catch{} 块关闭资源，因为这是不能保证被调用的。 在实现 Closable 或 AutoClosable 接口的类的所有实例上显式调用 .close()。 还有一种更好的解决方案是使用 try-with-resources，以便自动关闭定义的所有 AutoClosable 实例。
CWE	CWE ID 404
OWASP2017	None

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	**Inadequate Encryption Strength**
默认严重性	4
摘要	应用程序使用 Y（文件） 文件第 M 行的弱加密算法 YY（元素） 保护 X（文件） 文件第 N 行的敏感信息 XX（元素）。
解释	使用弱的或过时的加密不能为敏感数据提供足够的保护。获得访问加密数据权限的攻击者可能会使用密码分析或强力攻击来破解加密。这样，攻击者可能会窃取用户密码和其他个人数据。这可能导致用户假冒身份或身份盗用。 应用程序使用了被认为过时的弱算法，因为相对容易破解。这些过时的算法容易受到多种类型的攻击，包括暴力破解。
建议	通用指南： 一定要使用强的现代算法进行加密、进行 hash 计算等。 不要使用弱的、过时的或淘汰的算法。 一定要根据具体要求选择正确的加密机制。 应使用专用密码保护方案保护密码，例如 bcrypt、scrypt、PBKDF2 或 Argon2。 具体建议： 不要使用 SHA-1、MD5 或任何其他弱的 hash 算法来保护密码或个人数据。相反，需要安全 hash 时，要使用较强的 hash，例如 SHA-256。 不要使用 DES、3DES、RC2 或任何其他弱加密算法来保护密码或个人数据。而要使用较强的加密算法来保护个人数据，例如 AES。 不要使用 ECB 等弱加密模式，也不要依赖不安全的默认值。要显式设置较强的加密模式，例如 GCM。 对于对称加密，请使用至少 256 位的密钥长度。
CWE	CWE ID 326
OWASP2017	A3-Sensitive Data Exposure

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	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Compiler Optimization**
默认严重性	2.0
摘要	如果调用 X(函数) 会从内存中删除敏感数据，则可能不会达到预期的效果。对内存中的敏感数据删除不当可能会危及数据的安全。

解释	Compiler optimization 错误会在以下情况中出现：
	1. 机密数据储存在内存中。
	2. 内存中的机密数据通过覆盖内容的方式进行清除。
	在这种情况下，该数据将由 X(文件)中第 N 行的 X（函数） 清除。
	3. 源代码使用一个优化编译器进行编译，从而标识和删除那些将相关内容作为死存储进行覆盖的函数，因为在随后的操作中不会再使用这一内存。
	在这种情况下，可以进行优化的函数为 X(文件)中第 N 行的 X（函数）。
	示例 1：以下代码将从用户处读取一个密码，使用该密码连接到后端大型机，然后尝试使用 memset() 来擦除内存中的密码。
	void GetData(char *MFAddr) {
	char pwd[64];
	if (GetPasswordFromUser(pwd, sizeof(pwd))) {
	if (ConnectToMainframe(MFAddr, pwd)) {
	// Interaction with mainframe
	}
	}
	memset(pwd, 0, sizeof(pwd));
	}
	如果例子中的代码被逐字执行，那么它就可以正确地运行，但是如果代码是使用优化的编译器进行编译的，如 Microsoft Visual C++(R) .NET 或者 GCC 3.x，那么 memset() 的调用会被当作一个死存储清除，因为 pwd 缓冲区在其数值被覆盖之后便不会再次使用了[2]。因为缓冲区 pwd 包含一个敏感值，所以如果数据长期驻留在内存中，应用程序会很容易受到攻击。如果攻击者能够访问正确的内存区域，那么他们就能使用复原后的密码来获取系统的控制权。
	为了防止攻击者获取系统机密，通常的做法是覆盖内存中操作的敏感数据，如密码或者用密码编写的密钥。然而，有了优化编译器，程序就不会一直仅按照源代码指示的那样运行。在本例中，编译器将 memset() 调用解析为一段 dead code，这是因为随后的操作不会再使用被写入的内存，然而显而易见的是这样做会引发安全问题。这里的问题是，很多编译器，实际上是很多编程语言在努力提高效率的同时没有考虑这个及其他安全问题。
	通常，攻击者会通过一个核心转储或运行时机制来利用这种漏洞，以便访问某个特定应用程序所使用的内存，并恢复机密信息。在攻击者

返回目录

	访问到机密信息之后，盗取更多的系统信息就相对简单了，并有可能危及到与该应用程序交互的其他资源。

建议	优化编译器对于性能来说是非常有利的，所以禁用优化并不是一种明智的选择。解决方法是准确地与编译器进行交流，以确定程序究竟该如何运行。因为对这种交流的支持还不完善，而且不同的平台之间各不相同，所以针对这一问题的解决方法目前也不是很完善。
	通常，可以通过读取从内存中清除的变量来强制使编译器保留对擦除函数的调用。另一个选项涉及可变指针，因为它们可以在应用程序之外进行修改，所以当前并未优化这些指针。您可以利用这一点来欺骗编译器，只要将指向敏感数据的指针强制转换为可变指针即可。可通过在 Example 1 中对 memset() 的调用之后添加以下行来实现这一点：
	(volatile char)pwd = *(volatile char*)pwd;
	虽然这两种解决方案都能阻止现有编译器优化对擦除函数（如 Example 1 中所示）的调用，但二者均采用当前的优化技术，这种技术将在未来不断发展。这也带来了另一个弊端：随着编译器技术的发展，像这样的安全漏洞仍可能会被再次引入程序，即便是应用程序的源代码并没有发生任何改变也是如此。
	在最新的 Windows(R) 平台上，可以考虑使用 SecureZeroMemory()，它可以安全地替代 ZeroMemory()，此函数会使用上述可变指针的方法来防止自己被优化 [2]。另外，在大多数 Microsoft Visual C++(R) 版本中，可以使用 #pragma 优化构造来防止编译器优化某个特定的代码块。例如：
	#pragma optimize("",off);
	memset(pwd, 0, sizeof(pwd));
	#pragma optimize("",on);

CWE	CWE ID 14
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Insecure Compiler Optimization:Pointer Arithmetic**
默认严重性	2.0
摘要	数组边界检查可能被错误地进行了优化。

解释	如果数组边界检查涉及计算非法指针，然后确定该指针超出边界，那么一些编译器将优化检查，假设程序员绝不会故意创建非法指针。
	示例：
	char *buf;
	int len;
	...
	len = 1<<30;
	if (buf+len < buf) //wrap check
	[handle overflow]
	操作 buf + len 大于 2^32，因此最终值小于 buf。但是由于指针上的运算溢出是未定义行为，一些编译器将假定 buf + len >= buf 并优化封装检查。由于该优化，后面的代码可能容易受缓冲区溢出的攻击。

建议	不要创建非法指针来执行边界检查。如果必须依赖于指针运算来进行边界检查，那么可将带符号的指针转换为
	其不带符号的等效指针。例如，“说明”部分中的代码可以被重写为：
	#include <stdint.h>
	char *buf;
	int len;
	...
	len = 1<<30;
	if ((uintptr_t)buf+len < (uintptr_t)buf) // wrap check
	[handle overflow]
	如果平台不支持 uintptr_t，您可以使用 size_t 作为一个替换选择。

CWE	CWE ID 733
OWASP2017	A3 Sensitive Data Exposure

返回目录

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

解释	在对安全性要求较高的环境中，使用能产生可预测数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，那么它就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，否则可能会导致严重的漏洞（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。
	char* CreateReceiptURL() {
	int num;
	time_t t1;
	char *URL = (char*) malloc(MAX_URL);
	if (URL) {
	(void) time(&t1);
	srand48((long) t1); /* use time to set seed */
	sprintf(URL, "%s%d%s", "http://test.com/", lrand48(), ".html");
	}
	return URL;
	}
	这段代码使用 lrand48() 函数为它生成的收据页面生成“唯一”的标识符。由于 lrand48() 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，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，

返回目录

	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。） 如今，有多种跨平台的解决方法都可以为 C 和 C++ 程序提供加密的安全 PRNG，例如 Yarrow [1]、CryptLib [2]、Crypt++ [3]、BeeCrypt [4] 以及 OpenSSL [5]。 在 Windows(R) 系统中，用 C 和 C++ 编写的程序可以在 CryptoAPI [6] 中使用 CryptGenRandom() 函数。为了避免牵扯到整个 CryptoAPI，直接访问其下级函数 RtlGenRandom() [7]。 在 Windows .NET 框架中，所有实现 System.Security.Cryptography.RandomNumberGenerator 的类都使用 GetBytes(函数，如 System.Security.Cryptography.RNGCryptoServiceProvider [8]。
CWE	CWE ID 338
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness:Hardcoded Seed**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数传递了种子的常数值。可生成随机或伪随机值的传递了种子的函数不应使用常量参数进行调用。可生成随机或伪随机值的传递了种子的函数不应使用常量参数进行调用。

解释	可生成随机或伪随机值的传递了种子的函数不应使用常量参数进行调用。如果伪随机数值生成器（如 rand()）使用特定值作为种子（使用类似 srand(unsigned int) 的函数），则通过 rand() 和通过可返回或分配值的类似方法返回的值对可以收集一定数量 PRNG 输出的攻击者来说是可预测的。
	示例 1：伪随机数生成器生成的值在前两个块中是可预测的，因为这两个块以同一种子开始。
	srand(2223333);
	float randomNum = (rand() % 100);
	syslog(LOG_INFO, "Random: %1.2f", randomNum);
	randomNum = (rand() % 100);
	syslog(LOG_INFO, "Random: %1.2f", randomNum);
	srand(2223333);
	float randomNum2 = (rand() % 100);
	syslog(LOG_INFO, "Random: %1.2f", randomNum2);
	randomNum2 = (rand() % 100);
	syslog(LOG_INFO, "Random: %1.2f", randomNum2);
	srand(1231234);
	float randomNum3 = (rand() % 100);
	syslog(LOG_INFO, "Random: %1.2f", randomNum3);
	randomNum3 = (rand() % 100);
	syslog(LOG_INFO, "Random: %1.2f", randomNum3);
	在此例子中，randomNum1 和 randomNum2 的结果设置相同的种子，因此在为伪随机数值生成器 srand(2223333) 设置种子的调用后，对 rand() 的每次调用都将会以相同的调用顺序产生相同的输出。例如，输出可能与以下内容相似：
	Random: 32.00
	Random: 73.00
	Random: 32.00
	Random: 73.00
	Random: 15.00
	Random: 75.00
	这些结果并不是随机的。

建议	使用硬件型随机性源设置种子的加密 PRNG，比如环形振子、磁盘驱动器定时、热噪声或放射性衰变。与设置常量种子相比，这样做会让使用 rand() 和类似方法生成的数据顺序更加难以预测得多。但应注

返回目录

	意，有比 rand() 效果好得多的伪随机数值生成器（例如，iOS 上的 arc4random()）。
CWE	CWE ID 336
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness:User-Controlled Seed**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数传递了种子的受污染值。生成了随机或伪随机值并传递了种子的函数不应使用受污染参数进行调用。生成了随机或伪随机值并传递了种子的函数不应使用受污染参数进行调用。
解释	对于生成了传递了种子（如 rand()）的随机或伪随机值（如 srand()）的函数，不应该使用受污染的参数进行调用。这样做可使攻击者控制作为伪随机数值生成器种子的值，因此能够预测由调用随机数值生成器产生的值（通常为整数）的顺序。
建议	使用通过硬件型随机性源（比如环形振子、磁盘驱动器定时、热噪声或放射性衰变）设置种子的加密 PRNG，例如，在 Unix 之类平台中，如果您需要一个高熵做种的伪随机数值生成器，则使用 /dev/random。这样做会让使用 rand() 和类似方法生成的数据顺序更加难以预测得多。
CWE	CWE ID 335
OWASP2017	None

返回目录

漏洞名称	**Insecure Randomness:Weak Entropy Source**
默认严重性	3.0
摘要	由 X(函数) 实施的随机或伪随机数值生成器依赖于一个弱熵源。随机或伪随机数值生成器依赖于一个弱熵源。

解释	缺少供随机或伪随机数值生成器使用的正确熵源可能会导致拒绝服务或生成可预测的数字序列。如果随机或伪随机数值生成器使用耗尽的熵源，程序可能会暂停或甚至崩溃，导致拒绝服务。或者，随机或伪随机数值生成器可能会生成可预测的数字。弱的随机或伪随机数字源可能会导致漏洞，比如容易被猜出的临时密码、可预测的加密密钥、会话劫持以及 DNS 欺骗。
	例 1: 以下代码使用系统时钟作为熵源:
	...
	srand (time(NULL));
	r = (rand() % 6) + 1;
	...
	因为系统时钟生成的是可预测的值，因此它不是理想的熵源。这点同样适用于其他非硬件型随机性源，包括系统/输入/输出缓冲区、用户/系统/硬件/网络序列号或地址，以及用户输入。

建议	避免使用非硬件型随机性源。尽可能使用硬件型随机性源，比如环形振子、磁盘驱动器定时、热噪声或放射性衰变。 在 Unix 之类的平台上，字符特殊文件 /dev/random 和 /dev/urandom（自 Linux 1.3.30 起开始出现）提供了一个与内核的随机数值生成器的接口。随机数值生成器将来自设备驱动程序和其他来源的环境噪声汇集成一个熵池。当熵池为空时，从 /dev/random 的数据读取将会阻塞，直到汇集了更多环境噪声。但是，从 /dev/urandom 的数据读取将不会阻塞，以等待更多熵。因此，如果熵池中没有足够的熵，返回的值理论上易受到针对驱动程序使用的算法的加密攻击。 总是首选使用 /dev/random，而非 /dev/urandom。
CWE	CWE ID 331
OWASP2017	None

返回目录

漏洞名称	**Insecure Scrypt Parameters**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不足的输出长度保存在 Y（文件） 文件第 M 行的 YY（元素） 中 X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不安全的机制保存在 Y（文件） 文件第 M 行的 YY（元素） 中
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 在这种情况下，密钥派生函数的输出长度太低。这可能导致冲突，使不同密码的 hash 计算结果相同，造成身份验证成功，而这是不应该发生的。 虽然 scrypt 专门用于阻止暴力攻击和其他快速攻击，但如果定义了错误的输入参数也会显著弱化算法的输出。 用于存储凭证的方法不安全。 密钥派生函数的输出长度太低。 参数 N、r 和 p 未设置安全值。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛 (PHC) 获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 然而，使用这些函数无法保证安全的凭证存储机制。为了避免冲突，密钥派生函数输出的安全长度应为 32 字节或更高。 在 scrypt 中，这些参数是 N、r 和 p。N 定义 CPU/内存成本参数，r 定义块大小参数，p 是并行化参数。N 和 r 定义执行一次尝试的难度，p 定义应该使用多少处理器来执行计算。确定为安全的当前值组定义是 N >= 2^15、r >= 8 和 p >= 2。
CWE	CWE ID 522
OWASP2017	None

返回目录

漏洞名称	**Insecure SSL:Server Identity Verification Disabled**
默认严重性	3.0
摘要	在进行 SSL 连接时，通过 X(文件) 中的 X(函数) 建立的连接不验证服务器证书。这使得应用程序易受到中间人攻击。当进行 SSL 连接时，服务器身份验证处于禁用状态。
解释	在一些使用 SSL 连接的库中，可以禁用服务器证书验证。这相当于信任所有证书。 例 1： 此应用程序显式禁用了证书验证： ... SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, verify_callback); ... 当尝试连接到有效主机时，此应用程序将随时接受颁发给“hackedserver.com”的证书。此时，当服务器被黑客攻击发生 SSL 连接中断时，应用程序可能会泄漏用户敏感信息。
建议	当进行 SSL 连接时，不要忘记服务器验证检查。根据所使用的库，一定要验证服务器身份并建立安全的 SSL 连接。 例 2：此应用程序明确地验证服务器证书。 ... SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|SSL_VERIFY_CLIENT_ONCE, verify_callback); ...
CWE	CWE ID 297
OWASP2017	A3 Sensitive Data Exposure

返回目录

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

解释	应用程序会非常频繁地使用临时文件，因此您可以使用多种不同的机制在 C 库和 Windows(R) API 中创建临时文件。而多数函数都很容易受到各种攻击。
	在这种情况下，用于创建临时文件的潜在 dangerous function 为 X(文件) 中第 N 行的 X（函数）。
	示例：以下代码使用一个临时文件，在它被处理之前用来存储从网络上收集到的中间数据。
	...
	if (tmpnam_r(filename)){
	FILE* tmp = fopen(filename,"wb+");
	while((recv(sock,recvbuf,DATA_SIZE, 0) >
	0)&&(amt!=0))
	amt = fwrite(recvbuf,1,DATA_SIZE,tmp);
	}
	...
	这种其他情况下无法标记的代码很容易受到多种不同形式的攻击，因为它靠一种不安全的方法来创建临时文件。以下部分主要描述了由该函数和其他函数引入的漏洞。大部分与临时文件创建有关的突出安全问题已经在基于 Unix 的操作系统上屡见不鲜，但是 Windows 应用程序同样存在着这样的风险。这部分对在 Unix 和 Windows 系统上创建临时文件的问题进行了讨论。
	不同系统之间使用的方法和行为可能各不不同，但是被引入的基本风险则相差不大。针对创建临时文件的安全方法，要了解有关安全的核心语言函数的信息及建议，请查看“建议”部分。
	对于旨在帮助创建临时文件的函数，可以根据它们是仅提供文件名还是实际打开新文件分成两个组。
	第 1 组 —“唯一的”文件名：
	在第一组中，C 库和 WinAPI 函数用来帮助创建一个临时文件。它们可为程序随后打开的新的临时文件生成一个唯一的文件名。这组包含了诸如 tmpnam()、tempnam() 和 mktemp() 等 C 库函数以及 C++ 中以 _（下划线）开头的相应函数和 Windows API 中的 GetTempFileName() 函数。这组函数在文件名的选择方面很可能会在底层碰到 race condition。虽然函数可以保证在选择文件时其文件名是唯一的，但是还无法防止其他进程或攻击者在选择文件后，而应用程序尚未尝试打开该文件前的这段时间内创建一个同名文件。不止是由其他程序调用相同函数所引发的合法冲突，攻击者还非常有可能创建

返回目录

	一个恶意的冲突，因为这些函数创建的文件名没有进行充分的随机化，使其难以被攻击者猜测。 如果使用选定的名称创建文件，那么根据打开方式的不同，文件现有的内容或访问权限可能会保持不变。如果文件的现有内容是恶意的，攻击者可能会在应用程序从临时文件中读取数据时向程序中注入危险数据。如果攻击者预先创建了一个能轻松获取访问权限的文件，那么可能会访问、修改或破坏应用程序存储在临时文件里的数据。在基于 Unix 的系统上，如果攻击者预先创建了一个作为另一个重要文件链接的文件，则可能会引发更加严重的攻击。然后，如果应用程序被截短或向文件中写入数据，那么它可能会在不知不觉中帮助攻击者，为其执行各种恶意操作。如果程序再使用提高了的权限运行，那会使问题变得更加严重。 最后，最好的情况就是通过调用 open() 函数并使用 O_CREAT 和 O_EXCL 标记来打开文件，或者通过调用 CreateFile() 函数并使用 CREATE_NEW 属性来打开文件，这样，如果文件已经存在，该操作就会失败，因此可以有效地防止上述攻击类型。然而，如果攻击者可以准确预测一系列临时文件名，那么就可以阻止应用程序打开必要的临时存储空间，从而导致拒绝服务 (DoS) 攻击。如果仅从一小部分随机数中选择由这些函数生成的文件名，那么会很容易发动这种类型的攻击。 第 2 组 —“唯一的”文件: 在第二组中，C 库函数通过生成唯一的文件名且打开这个文件，来解决一些与临时文件有关的安全问题。这部分包含了像 tmpfile() 这样的 C 库函数和与之对应的以 _（下划线）开头的 C++ 函数，以及表现更为出色的 C 库函数 mkstemp()。 tmpfile() 样式的函数可以构造唯一的文件名，并在传递了 "wb+" 标志的情况下能够按照与 fopen() 函数相同的方式（即，作为在读/写模式下的二进制文件）打开文件。如果文件已存在，tmpfile() 将把文件的大小缩小为 0，也许能缓解前面提到的安全问题（唯一文件名的选择与随后打开所选文件之间的 race condition）。然而，该操作显然不能解决函数的安全性问题。首先，攻击者可以预先创建一个能轻松获取访问权限的文件，该文件可能会被用 tmpfile() 函数打开的文件保留。其次，在基于 Unix 的系统上，如果攻击者预先创建了一个文件作为另一重要文件的链接，应用程序可能会使用提高了的权限去截短该文件，这样就能按照攻击者的意愿执行破坏。最后，如果 tmpfile() 创建了一个新文件，那么应用在该文件上的访问权限在不同的操作系统间是不同的，因此，应用程序的数据极易受到攻击，即便是攻击者无法预测要使用的文件名。 最后，mkstemp() 函数是一种创建临时文件的安全方法。它根据用户提供的模板（该模板由一系列随机生成的字符组成），尝试创建或打开一个唯一的文件。如果它无法创建一个这样的文件，则操作失败，并返回 -1。在最新的系统中，文件使用 0600 模式打开，这就意味着文件不会被篡改，除非用户直接更改其访问权限。然而，mkstemp() 仍然会受到使用可预测文件名的威胁，且如果攻击者通过猜测和预先

返回目录

	创建将要使用的文件名的文件，而导致 mkstemp() 函数失效，将使应用程序极易受到 denial of service 攻击。
建议	在基于 Unix 的系统上，只要可以使用 mkstemp() 函数，在提供的开箱即用的函数中，它是创建临时文件的最佳选择。在早期的系统中，mkstemp() 函数使用模式 0666 创建文件，致使所有用户都能访问文件，这样一来，应用程序数据极易受到攻击。正是由于早期的系统存在这样的问题，您应该要求所有新建的文件只能在创建临时文件之前调用 umask(077)，并且只有当前用户才能访问文件。 在基于 Windows 的系统上，唯一可行的开箱即用的解决方法是使用带有 CREATE_NEW 属性的 GetTempFileName() 和 CreateFile() 函数以及一个安全性足够高的描述符，这个描述符仅允许当前用户访问。 所有这些解决方案都强调由攻击者加载的 denial of service 攻击，此时攻击者能够预测出即将生成的文件名。如果您的环境中涉及到了此类攻击，唯一的解决方法是建立一个专有的临时文件生成框架，因为没有一个开箱即用的解决方案能够很好地处理这个问题。 如果您为创建临时文件编写了自己的代码，可以考虑把它放在一个不能被公共读写的目录下，以此防止与攻击者发生争用。如果确有需要，只要您小心些，便可以安全地在 /tmp 下创建这样一个目录。 如果您对生成难以猜测的临时文件名感兴趣，应该使用一个加密的安全伪随机数值生成器 (PRNG) 来创建一个在所有临时文件中使用的随机元素。 有很多种应用于 C 和 C++ 程序的跨平台解决方案，以提供加密的安全 PRNG，比如 Yarrow [1]、CryptLib [2]、Crypt++ [3]、BeeCrypt [4] 和 OpenSSL [5] 等。 在 Windows（r）系统上，C 和 C++ 程序可以利用 CryptoAPI [6] 中的 CryptGenRandom() 函数。如果您希望避免牵扯到其他 CryptoAPI，您可以直接访问其下级函数 RtlGenRandom() [7]。 最后，在 Windows .NET 框架中，您可以在任何实现了 System.Security.Cryptography.RandomNumberGenerator 接口的类（比如 System.Security.Cryptography.RNGCryptoServiceProvider [8]）中利用 GetBytes() 函数。
CWE	CWE ID 377
OWASP2017	None

返回目录

漏洞名称	**Insecure Transport:Weak SSL Protocol**
默认严重性	4.0
摘要	SSLv2、SSLv23 和 SSLv3 协议包含多个使它们变得不安全的缺陷，因此不应该使用它们来传输敏感数据。
解释	传输层安全 (TLS) 协议和安全套接字层 (SSL) 协议提供了一种保护机制，可以确保在客户端和 Web 服务器之间所传输数据的真实性、保密性和完整性。TLS 和 SSL 都进行了多次修订，因此需要定期进行版本更新。每次新的修订都旨在解决以往版本中发现的安全漏洞。使用不安全版本的 TLS/SSL 将削弱数据保护力度，并可能允许攻击者危害、窃取或修改敏感信息。 弱版本的 TLS/SSL 可能会呈现出以下其中一个或所有属性： - 没有针对中间人攻击的保护 - 身份验证和加密使用相同密钥 - 消息身份验证控制较弱 - 没有针对 TCP 连接关闭的保护 这些属性的存在可能会允许攻击者截取、修改或篡改敏感数据。
建议	强烈建议强制客户端仅使用最安全的协议。 例 1： c->sslContext = SSL_CTX_new (TLSv1_2_method()); Example 1 演示了如何强制实施基于 TLSv1.2 协议的通信。
CWE	CWE ID 327
OWASP2017	A6 Security Misconfiguration

返回目录

漏洞名称	**Insufficient BCrypt Cost**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不足的输出长度保存在 Y（文件） 文件第 M 行的 YY（元素） 中 Y（文件） 文件第 M 行 YY（元素） 中定义的成本参数过低。
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 在这种情况下，密钥派生函数的输出长度太低。这可能导致冲突，使不同密码的 hash 计算结果相同，造成身份验证成功，而这是不应该发生的。 将 bcrypt 成本参数设置为过低的值会显著降低生成的 hash 的安全性。 用于存储凭证的方法不安全。 密钥派生函数的输出长度太低。 bcrypt 函数的成本参数过低。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 然而，使用这些函数无法保证安全的凭证存储机制。为了避免冲突，密钥派生函数输出的安全长度应为 32 字节或更高。 Bcrypt 的成本参数应设置为默认值（10）或更高。减少此值会降低根据密码计算 hash 所需的计算工作量，从而削弱产生的 hash，使攻击者更容易破解密码。
CWE	CWE ID 522
OWASP2017	None

返回目录

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

返回目录

漏洞名称	**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	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Integer Overflow**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数没有对 Integer overflow 进行说明，这可能会导致逻辑错误或 buffer overflow。没有对 Integer overflow 进行说明可能会导致逻辑错误或 buffer overflow。

解释	程序没有对以下事实进行说明时，会产生 Integer overflow 错误：算术运算会导致数值大于数据类型的最大值或者小于数据类型的最小值。用户输入在与带符号的值和不带符号的值之间的隐式转换进行交互时会产生一些错误，而这些错误经常会导致内存分配函数出现问题。如果攻击者能够使程序分配的内存不足或是在进行内存操作时，将一个带符号的值作为不带符号的值来解析，将会使程序很容易出现 buffer overflow。
	例 1：以下代码摘自 OpenSSH 3.3，演示了一个经典的 integer overflow 案例：
	nresp = packet_get_int();
	if (nresp > 0) {
	response = xmalloc(nresp*sizeof(char*));
	for (i = 0; i < nresp; i++)
	response[i] = packet_get_string(NULL);
	}
	如果 nresp 拥有值 1073741824，sizeof(char*) 拥有典型值 4，那么 nresp*sizeof(char*) 的操作结果将溢出，xmalloc() 参数将变为 0。大多数 malloc() 实施将允许分配 0 字节缓冲区，导致随后的循环迭代次数超出堆缓冲区 response。
	例 2： 该示例处理的用户输入包含一系列可变长度结构。输入的前 2 个字节决定了要处理的结构大小。
	char* processNext(char* strm) {
	char buf[512];
	short len = *(short*) strm;
	strm += sizeof(len);
	if (len <= 512) {
	memcpy(buf, strm, len);
	process(buf);
	return strm + len;
	} else {
	return -1;
	}
	}
	程序员已为结构大小设置了一个上限：如果它大于 512，输入将不会被处理。问题是 len 为一个带符号的整数，因此针对最大结构长度的检查是用带符号的整数完成的，但是针对 memcpy() 调用，len 会转

返回目录

	化为一个不带符号的整数。如果 len 为负，那么结构将看起来有一个适当的尺寸（将执行 if 分支），但是通过 memcpy() 复制的内存数量将相当大，攻击者将能够使用 strm 中的数据溢出堆栈。

建议	虽然没有简单的原则能够使您避免 integer overflow 问题，但以下的建议还是可以帮助您避免一些最为恶劣的情况：
	— 要注意编译器提示的有关于带符号/不带符号的转换的警告。一些程序员可能认为这些警告没有用处，但是有些时候它们可能会指出潜在的 integer overflow 问题。
	— 对程序输入设定的上下限，请谨慎检查其合理性。即使程序只需要正整数，也必须进行检查以确保您所处理的值不小于零。（您可以通过使用不带符号的数据类型来取消对下限的界定。）
	— 谨慎限定您所允许的数值范围。
	- 当您调用函数，执行算术操作或者比较不同类型的数值时，了解程序执行过程中隐含的类型转换。
	示例 3： 以下代码实现了一个封装器函数，该函数可通过在调用 malloc() 之前执行适当的参数检查来为数组安全地分配内存空间。
	void* arrmalloc(uint sz, uint nelem) {
	void *p;
	if(sz > 0 && nelem >= UINT_MAX / sz)
	return 0;
	return malloc(sz * nelem);
	}

CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**Integer Overflow**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的值未经验证便被用于 Y（文件） 文件第 M 行 YY（元素） 中的算术运算中，这可能导致算术溢流（通常称为“整数溢出”）。
解释	算术溢出会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断或丢失数据；使用溢出的数值执行内存操作时，甚至导致缓冲区溢出，从而导致执行任意代码。 所有数字数据类型都按位表示。如果在算术运算后，某个值超过了其按位表示的位数，那么被加的最重要的数字将被截断，此截断后剩余的值将是回绕操作的剩余值——这称为“算术溢出”，另常误称为——“整数溢出”，此误称是有误导的，因为此名称还适用于许多其他类型。如果可能溢出的数据类型低于其最小值，则它们将负向回绕，通常称为下溢。 例如，如果无符号的 32 位整数设置为 4,294,967,295，然后加 1，它就会溢出并回绕到 0。如果有符号的 32 位整数值为 2,147,483,647，然后加 1，则它会溢出并回绕到 -2,147,483,648。 为了保证代码正确，一定要检查值是否在预期范围内，确保算术运算结果不会溢出或下溢。
建议	对可能包含任意值的数据执行算术运算时，请考虑添加一个检查以保证数据在范围内，且这些操作的结果不会导致溢出或下溢。 考虑为所有算术运算创建封装器，以对特殊情况进行特定处理；例如，如果检查显示已经或将要发生溢出，则抛出异常。
CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**Key Management:Empty Encryption Key**
默认严重性	4.0
摘要	位于 X(文件)中第 N 行的 X（函数） 的空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。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。 示例：以下代码使用空加密密钥: ... char encryptionKey[] = ""; ... 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（空或非空），会造成任何有足够权限的人均可读取和无意中误用加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Hardcoded Encryption Key**
默认严重性	4.0
摘要	硬编码加密密钥可能会削弱安全性，一旦出现安全问题将无法轻易修正。
解释	使用硬编码方式处理加密密钥绝非好方法。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，加密密钥位于 X(文件)中第 N 行的 X（函数）。 示例：以下代码使用硬编码加密密钥: ... char encryptionKey[] = "lakdsljkalkjlksdfkl"; ... 任何可访问该代码的人都能访问加密密钥。一旦程序发布，除非对程序进行修补，否则将无法更改加密密钥。雇员可以利用手中掌握的信息访问权限入侵系统。如果攻击者能够访问应用程序的可执行代码，他们就能对包含所使用的加密密钥值的代码进行反汇编。
建议	绝不能对加密密钥进行硬编码，通常应对其进行模糊化，并在外部源中进行管理。在系统中的任何位置采用明文的形式存储加密密钥，会造成任何有足够权限的人均可读取和无意中误用加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Key Management:Null Encryption Key**
默认严重性	4.0
摘要	位于 X(文件) 中第 N 行的 X（函数）处的 Null 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。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	使用 null 加密密钥绝非好方法。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。 示例：以下代码会使用 null 加密密钥： ... char encryptionKey[] = null; ... 不仅任何可以访问此代码的人能够确定它使用的是 null 加密密钥，而且任何掌握最基本破解技术的人都更有可能成功解密任何加密数据。一旦程序发布，要更改 null 加密密钥，就必须进行软件修补。雇员可以利用手中掌握的信息访问权限入侵系统。即使攻击者只能访问应用程序的可执行文件，他们也可以提取使用了 null 加密密钥的证据。
建议	加密密钥绝不能为 null，而通常应对其加以模糊化，并在外部源中进行管理。在系统中的任何位置采用明文的形式存储加密密钥（null 或非 null），会造成任何有足够权限的人均可读取和无意中误用此加密密钥。
CWE	CWE ID 321
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**LDAP Injection**
默认严重性	4.0
摘要	函数 XX（函数） 可以调用利用第 N 行中未经验证的输入生成的动态 LDAP 筛选器，这可以让攻击者借机修改指令的含义。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 X（文件）的第 N 行的 X（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 Y（文件）的第 M 行中的 Y（函数）。
	注意：即使在修复过后，仍然可能会报告 LDAP Injection 问题（优先级值有所降低）。如果 Fortify Static Code Analyzer 发现正用于构建 LDAP 语句的用户控制输入的明显数据流证据，将会报告高/重大优先级数据流问题。如果 Fortify SCA 无法确定数据源，但数据源可能被动态更改，则会报告低/中优先级语义问题。在少数几个漏洞类别（如 LDAP Injection）中会采用该策略，其中对漏洞加以利用的潜在影响大于审核误报问题的不便。
	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。该经理的名字是从网络套接字中读取的，因此不可信任。
	fgets(manager, sizeof(manager), socket);
	snprintf(filter, sizeof(filter, "(manager=%s)", manager);
	if ((rc = ldap_search_ext_s(ld, FIND_DN, LDAP_SCOPE_BASE,
	filter, NULL, 0, NULL, NULL, LDAP_NO_LIMIT,
	LDAP_NO_LIMIT, &result)) == LDAP_SUCCESS) {
	...
	}
	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 manager 不包含任何 LDAP 元字符时才能正常运行。如果攻击者为 manager 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。

返回目录

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。
	示例 2：上述例子可以进行重写，以便使用允许列表进行验证，从而构造一个保护指令命令结构的滤字符串。
	...
	fgets(manager, sizeof(manager), socket);
	char* regex = "^[a-zA-Z\-\.']$";
	re = pcre_compile(regex, 0, &err, &errOffset, NULL);
	int rc = pcre_exec(re, NULL, manager, strlen(manager), 0, 0, NULL, 0);
	if(rc == 1) {
	snprintf(filter, sizeof(filter), "(manager=%s)", manager);
	if ((rc = ldap_search_ext_s(ld, FIND_DN, LDAP_SCOPE_BASE,
	filter, NULL, 0, NULL, NULL, LDAP_NO_LIMIT,
	LDAP_NO_LIMIT, &result)) == LDAP_SUCCESS) {
	...
	}
	}

CWE	CCI-002754
OWASP2017	A1 Injection

返回目录

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

返回目录

OWASP2017	None

返回目录

漏洞名称	**LDAP Manipulation**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数会执行包含未经验证的输入的 LDAP 指令，这可能会让攻击者借机修改指令的意义或执行非法的 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 X（文件）的第 N 行的 X（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 Y（文件）中第 M 行的 Y（函数） 使用该数据。
	例 1：以下代码从套接字中读取 dn 字符串，然后使用该字符串来执行 LDAP 查询。
	...
	rc = ldap_simple_bind_s(ld, NULL, NULL);
	if (rc != LDAP_SUCCESS) {
	...
	}
	...
	fgets(dn, sizeof(dn), socket);
	if ((rc = ldap_search_ext_s(ld, dn, LDAP_SCOPE_BASE,
	filter, NULL, 0, NULL, NULL,
	LDAP_NO_LIMIT,
	LDAP_NO_LIMIT, &result)) !=
	LDAP_SUCCESS) {
	...
	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，但会验证从套接字中读取的值，以确保它对应于有效的 dn，并且该代码会使用 LDAP 简单身份验证来确保该查询只影响允许当前用户访问的记录。

返回目录

	check_dn 方法将在预定义的有效基 dn 值列表中查找 check_dn 字符串。
	...
	snprintf(username, sizeof(username), "(cn=%s)", user);
	rc = ldap_simple_bind_s(ld, username, password);
	if (rc != LDAP_SUCCESS) {
	...
	}
	fgets(dn, sizeof(dn), socket);
	if (check_dn(dn) == SUCCESS) {
	if ((rc = ldap_search_ext_s(ld, dn, LDAP_SCOPE_BASE,
	filter, NULL, 0, NULL, NULL,
	LDAP_NO_LIMIT,
	LDAP_NO_LIMIT,
	&result)) != LDAP_SUCCESS) {
	...
	}
	}

CWE	CWE ID 90
OWASP2017	None

返回目录

漏洞名称	**Least Privilege Violation**
默认严重性	3.0
摘要	程序应该在操作结束后立即丢弃执行诸如 chroot() 一类的操作所需要的提高了的权限。

解释	当程序调用一个受权限控制的函数时，比如 chroot()，它必须先获取 root 权限。当受权限控制的操作完成后，程序应该马上丢弃 root 权限并且返回调用它的用户权限等级。
	在这种情况下，没有随后执行权限丢弃操作的特权函数为 XX（文件）中第 M 行的 <Replace key="FirstTransitionFunction" link="FirstTraceLocation" />。
	示例：以下的代码调用了 chroot() 函数，将应用程序限制在 APP_HOME 下的 file system 的子集中，以防止攻击者通过程序访问位于其他地方的未经授权的文件。然后，代码打开一个由用户指定的文件并处理文件的内容。
	...
	chroot(APP_HOME);
	chdir("/");
	FILE* data = fopen(argv[1], "r+");
	...
	在打开文件之前，把应用程序的操作限制在主目录之内是一个相对安全的方法。然而，不使用一些非零值对函数 setuid() 进行调用，意味着应用程序仍在使用没有必要的 root 权限进行操作。任何由攻击者对应用程序实施的成功盗取都会导致发生权限扩大的攻击，因为所有的恶意操作都将以超级用户的权限执行。如果应用程序把权限等级降低到一个非 root 用户，则会显著减少很多潜在的破坏。

建议	应尽可能在执行特权操作（如 chroot()）后立即使用一个非零 uid 来调用 setuid() 函数以丢弃权限。 有时，让一个应用程序通过使用 setuid() 调用来丢弃权限是很难的，甚至是不可能的。这种情况通常发生在应用程序需要在运行中执行如 root 这样的操作，比如在一个 FTP 进程通过绑定 1-1024 端口运行时。在这种情况下，seteuid() 函数的使用应该暂时降低到一个较低权限等级，并且保留需要时返回为 root 的能力。请注意，攻击者同样能非常容易地把应用程序返回成 root 权限，因此只有在非常需要时再使用 seteuid() 函数。
CWE	APSC-DV-000500 CAT II, APSC-DV-000510 CAT I, APSC-DV-001500 CAT II
OWASP2017	None

返回目录

漏洞名称	**Leaving Temporary Files**
默认严重性	3
摘要	应用程序在 X（文件）:N 的 XX（元素） 方法中生成了一个临时文件 XXX（方法）。这个临时文件从未被删除，并且该文件将无限期地保留在 TEMP 文件夹中。
解释	应用程序通常会创建包含敏感业务数据或个人信息的临时文件，以便分步骤处理文件生成流程，甚至作为自动流程的输出。这些文件如果无确定期限地暴露在磁盘上，就可能会将秘密数据泄露给未经授权的用户。 应用程序经常会使用临时文件作为中间存储，或者帮助处理大量数据或长时间运行的计算。应用程序经常需要这样的文件，所以大部分操作系统都为临时文件分配了专用区域（例如 TEMP 目录），并且大部分平台都有多种创建此类文件的机制。但是，默认情况下，这些临时文件不会自动删除，并将无限期地保留在磁盘上。如果程序完成处理后未显式、主动地删除临时文件，则它们可能会被计算机的其他用户访问。
建议	一定要显式删除创建的所有临时文件。通过将临时文件封装在 finally {} 块中确保将其删除，或者调用 File.deleteOnExit() 保证它最终会被删除。 此外，为了确保所有临时文件最终都会被删除，可考虑使用其他功能定期删除所有未使用的、现有的临时文件。 尝试删除前先确认所有现有文件句柄或引用均已关闭。
CWE	CWE ID 376
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Leftover Debug Code**
默认严重性	3
摘要	应用程序源代码在 X（文件） 文件第 N 行包含源代码 XX（元素），这是开发和调试时留下的，不是指定应用程序功能的一部分。
解释	测试和调试代码不应部署到生产环境，它们可能会创建意外的入口点，从而增加应用程序的受攻击面。此外，这些代码通常未经过相应的测试或维护，并且可能还保留着已在代码库的其他部分中修复的历史漏洞。通常，调试代码会包含功能性“后门”，使程序员能够绕过运行安全机制，例如身份验证或访问控制。 在应用程序开发期间，程序员通常会使用专门的代码，以方便调试和测试。程序员甚至通常会使调试代码绕过安全机制，以便将测试集中在特定功能上，并将其与安全架构隔离开来。 此调试或测试代码未从代码库中删除，然后被包含在软件构建中并部署到了生产环境中。
建议	- 部署或构建应用程序之前删除所有调试代码。确保配置中未启用调试模式。 - 通过可以将测试用例代码与应用程序的其余部分隔离的专用测试框架实现所有测试代码。 - 避免在应用程序代码本身中实现特殊的“测试代码”、“调试时间”功能或“秘密”接口或参数。 - 使用专用的 CI / CD 工具定义和实施标准和自动的构建/部署过程，以自动配置部署的应用程序、排除所有临时代码、并仅包含指定的应用程序代码。
CWE	CWE ID 489
OWASP2017	None

返回目录

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

解释	在以下情况下会发生 Log Forging 的漏洞：
	1. 数据从一个不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据写入一个应用程序或系统日志文件。
	在这种情况下，数据通过 Y（文件）中第 M 行的 Y（函数） 记录下来。
	为了便于以后的审阅、统计数据收集或调试，应用程序通常使用日志文件来储存事件或事务的历史记录。根据应用程序自身的特性，审阅日志文件可在必要时手动执行，或者通过工具自动执行，以获取重要的数据点以及数据趋势。
	如果攻击者能给应用程序提供随后被逐字记录到日志文件的数据，可能会妨碍日志文件的检查，甚至基于日志数据的总结都可能是错误的。通过将日志条目分隔符所使用的字符包括到他们的数据中，攻击者可能会在日志文件中插入错误的条目信息。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]。在最糟糕是，攻击者可以在日志文件中注入代码或者其他的命令，充分利用日志处理实用程序 [2] 中的种种漏洞。
	示例：以下代码从一个 CGI 脚本中接受一个由用户提交的字符串，并试图将其转换为它所代表的长整数值。如果这个数值没能被解析成整数，那么数值会随同错误消息一起存入日志，提示发生的情况。
	long value = strtol(val, &endPtr, 10);
	if (*endPtr != '\0')
	syslog(LOG_INFO,"Illegal value = %s",val);
	...
	如果用户为 "val" 提交字符串 "twenty-one"，则日志中会记录以下条目：
	Illegal value=twenty-one
	然而，如果攻击者提交字符串 "twenty-one\n\nINFO: User logged out=evil"，则日志中会记录以下条目：
	INFO: Illegal value=twenty-one
	INFO: User logged out=evil

返回目录

	攻击者无疑能够使用同样的机制来插入任意日志条目。为了使这种类型的 log forging 攻击有效，攻击者必须首先定义有效的日志条目格式，但是这通常可以通过在目标应用程序中泄漏系统信息实现。
建议	防止 log forging 攻击的最好办法是使用一种间接方法：创建一组与不同事件一一对应的合法日志条目，这些事件必须记录在日志中，并且仅允许记录该组条目。要捕获动态内容，如用户注销系统，应经常使用受服务器控制的参数，而不是用户提供的数据。这就就可以确保用户提供的输入永远不会在日志条目中直接使用。 在某些情况下，这种方法有些不切实际，因为这一组合法的日志条目信息实在太大，或是难以跟踪。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。然而，任何不安全字符的列表很快就会变得不完善，而且可能会过时。更好的方法是创建一个字符列表，允许其中的字符出现在日志条目中，并且只接受完全由这些经认可的字符组成的输入。在大多数 Log Forging 攻击中，最关键的字符是“\n”换行符，这样的字符决不能出现在日志条目允许列表中。
CWE	CWE ID 117
OWASP2017	A1 Injection

返回目录

漏洞名称	**Log Forging**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于写审计日志。
解释	攻击者可以为安全敏感操作设计审计日志，然后设置错误的审计跟踪，并指向无辜的用户或隐藏事件。 应用程序会在出现安全敏感操作时写审计日志。因为审计日志中包含未经过数据类型验证或随后净化的用户输入，所以输入中可能包含表面上像合法审计日志数据的虚假信息，
建议	无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值 验证不能代替编码。完全编码所有来源的所有动态数据，然后再将其嵌入日志。 使用安全的日志记录机制。
CWE	CWE ID 117
OWASP2017	A1-Injection

返回目录

漏洞名称	**Long Overflow**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的值未经验证便被用于 Y（文件） 文件第 M 行 YY（元素） 中的算术运算中，这可能导致算术溢流（通常称为“整数溢出”）。
解释	算术溢出会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断或丢失数据；使用溢出的数值执行内存操作时，甚至导致缓冲区溢出，从而导致执行任意代码。 所有数字数据类型都按位表示。如果在算术运算后，某个值超过了其按位表示的位数，那么被加的最重要的数字将被截断，此截断后剩余的值将是回绕操作的剩余值——这称为“算术溢出”，另常误称为——“整数溢出”，此误称是有误导的，因为此名称还适用于许多其他类型。如果可能溢出的数据类型低于其最小值，则它们将负向回绕，通常称为下溢。 例如，如果无符号的 32 位整数设置为 4,294,967,295，然后加 1，它就会溢出并回绕到 0。如果有符号的 32 位整数值为 2,147,483,647，然后加 1，则它会溢出并回绕到 -2,147,483,648。 为了保证代码正确，一定要检查值是否在预期范围内，确保算术运算结果不会溢出或下溢。
建议	对可能包含任意值的数据执行算术运算时，请考虑添加一个检查以保证数据在范围内，且这些操作的结果不会导致溢出或下溢。 考虑为所有算术运算创建封装器，以对特殊情况进行特定处理；例如，如果检查显示已经或将要发生溢出，则抛出异常。
CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**Memory Leak**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数在第 M 行上分配内存，但无法释放。内存已分配，但永远不会释放。

解释	Memory leak 的产生有两个常见（有时候这两个原因会同时发生作用）原因：
	– 错误状况及其他异常情况。
	— 不清楚由程序的哪一部分负责释放内存。
	在这种情况下，XX（文件）中第 M 行分配的内存不总由该函数进行释放或返回。
	大多数 memory leak 会导致常规软件可靠性问题，但如果攻击者能够蓄意触发 memory leak，他就可能会通过引发程序崩溃来发起一个 denial of service 攻击，或者是利用因内存低的情况 [1] 所引发的其他意外的程序行为。
	例 1：如果调用 read() 后没有返回预期的字节数，以下 C 函数将会泄漏已分配的内存块的信息：
	char* getBlock(int fd) {
	char* buf = (char*) malloc(BLOCK_SIZE);
	if (!buf) {
	return NULL;
	}
	if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
	return NULL;
	}
	return buf;
	}

建议	因为 memory leak 难以跟踪，为此，您应该为您的软件建立一组内存管理模式和惯用语法。不要违反您制定的规则。
	在本例中，一个应对错误处理失当的良好解决方法便是使用向前式 goto 语句，使函数有一个明确定义的区域来处理错误，如下所示：
	char* getBlock(int fd) {
	char* buf = (char*) malloc(BLOCK_SIZE);
	if (!buf) {
	goto ERR;
	}
	if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
	goto ERR;
	}
	return buf;
	ERR:
	if (buf) {

返回目录

	free(buf);
	}
	return NULL;
	}

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Memory Leak**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的内存分配并不能保证在其声明范围内正确释放，可能会发生内存泄漏。
解释	如果代码中的任何地方以不可预测的方式发生内存泄漏，则应用程序可能会在一段时间内膨胀，直到耗尽所有可用内存，此时可能会发生速度变慢甚至崩溃。但是，如果攻击者可以引发内存泄漏，他们就能用它来故意造成拒绝服务攻击。 如果内存的分配方式需要释放，但使用后未正确释放，就会发生内存泄漏。实际上，如果错误的条件跳过内存释放阶段，或者代码中未明确说明在哪里释放内存，导致在所有地方都没有正确地释放，就会发生这种情况。
建议	确保内存在其声明的范围内释放，以避免忽视内存释放的情况。 处理错误时，一定要确保所有可能的代码流都正确地地丢弃了所有已分配的内存——即使发生错误时也是如此。
CWE	CWE ID 401
OWASP2017	None

返回目录

漏洞名称	**Memory Leak:Reallocation**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数会调整第 N 行分配的内存块的大小。如果调整大小时失败，初始块将会泄漏。该程序会调整分配的内存块的大小。如果调整大小时失败，初始块将会泄漏。

解释	Memory leak 的产生有两个常见（有时候这两个原因会同时发生作用）原因：
	– 错误状况及其他异常情况。
	— 不清楚由程序的哪一部分负责释放内存。
	在这种情况下，如果尝试调整大小时失败，则在 XX（文件）中第 M 行分配的内存可能会泄漏。
	大多数 memory leak 会导致常规软件可靠性问题，但如果攻击者能够蓄意触发 memory leak，他就可能会通过引发程序崩溃来发起一个 denial of service 攻击，或者是利用因内存低的情况 [1] 所引发的其他意外的程序行为。
	例 1：如果 realloc() 调用无法调整初始分配的大小，以下 C 函数会泄漏一块分配的内存。
	char* getBlocks(int fd) {
	int amt;
	int request = BLOCK_SIZE;
	char* buf = (char*) malloc(BLOCK_SIZE + 1);
	if (!buf) {
	goto ERR;
	}
	amt = read(fd, buf, request);
	while ((amt % BLOCK_SIZE) != 0) {
	if (amt < request) {
	goto ERR;
	}
	request = request + BLOCK_SIZE;
	buf = realloc(buf, request);
	if (!buf) {
	goto ERR;
	}
	amt = read(fd, buf, request);
	}
	return buf;
	ERR:
	if (buf) {
	free(buf);
	}

返回目录

	return NULL; }

建议	因为 memory leak 难以跟踪，为此，您应该为您的软件建立一组内存管理模式和惯用语法。不要违反您制定的规则。
	在本例中，一个应对错误处理失当的良好解决方法便是使用向前式 goto 语句，使函数有一个明确定义的区域来处理错误，如下所示：
	char* getBlocks(int fd) {
	int amt;
	int request = BLOCK_SIZE;
	char* newbuf;
	char* buf = (char*) malloc(BLOCK_SIZE + 1);
	if (!buf) {
	goto ERR;
	}
	amt = read(fd, buf, request);
	while ((amt % BLOCK_SIZE) != 0) {
	if (amt < request) {
	goto ERR;
	}
	request = request + BLOCK_SIZE;
	newbuf = realloc(buf, request);
	if (!newbuf) {
	goto ERR;
	}
	buf = newbuf;
	amt = read(fd, buf, request);
	}
	return buf;
	ERR:
	if (buf) {
	free(buf);
	}
	if (newbuf) {
	free(newbuf);
	}
	return NULL;
	}

CWE	APSC-DV-002400 CAT II
OWASP2017	Rule 18-4-1

返回目录

漏洞名称	**MemoryFree on StackVariable**
默认严重性	4
摘要	Y（文件） 文件第 M 行对 free() 的调用是在非动态分配的变量上执行的，但它是在 X（文件） 中的第 N 行上创建的，这会导致崩溃。
解释	崩溃可能会导致未定义的行为。崩溃可能给攻击者提供与系统和程序内部相关的有价值信息，并可能导致拒绝服务攻击。 在非动态分配的变量（例如 malloc）上调用 free() 将导致未定义的行为。
建议	仅在动态分析的变量（malloc、calloc、realloc 等）上调用 free()，以避免编译器出现意外行为。
CWE	CWE ID 590
OWASP2017	None

返回目录

漏洞名称	**Missing Check against Null**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数会间接引用第 <Replace key="LastTraceLocation.line" /> 行的 null 指针，因为它不会对 X(函数) 的返回值进行检查，而其返回值可能为 null。程序会间接引用 null 指针，因为它不会对函数的返回值进行检查，而该函数有可能返回 null。

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的两个可疑的假设是：一是这个函数调用不可能出错；二是即使出错了，也不会对系统造成什么重要影响。当程序员忽略函数返回值时，就暗示着自己是基于上述任一假设来执行操作。这种情况下，将使用未经 X(文件) 中第 N 行验证的一个之前的返回值。
	例 1：以下代码在使用由 malloc() 返回的指针之前，并没有检查内存是否分配成功。
	buf = (char*) malloc(req_size);
	strncpy(buf, xfer, req_size);
	对于这种编码错误的一贯辩解是：
	“如果我的程序耗尽了所有内存，则会失败。无论是在程序尝试间接引用 null 指针时处理相关错误，还是允许程序自行崩溃并出现分段故障，都无关紧要。”
	但是这个解释忽略了以下三个重要的因素：
	— 根据应用程序的类型和大小，可能会释放由其他程序使用的内存，从而使程序继续运行。
	- 程序不可能执行正常退出（如果需要）。如果程序执行原子操作，则会使系统处于不一致的状态。
	— 程序员失去了记下诊断信息的机会。对 malloc() 的调用失败是不是因为 req_size 太大，还是因为在同一时刻处理的请求太多。或者是由于已累计超时的 memory leak 引起的。如果不对错误进行处理，就不会知道是什么原因。

建议	如果函数返回错误代码或其他任何有关运行成功或失败的证据，请务必检查错误状况，即便没有任何明显迹象表明会发生这种错误。除了防止安全错误，许多乍看上去难以理解的 bug 最后都会归结为是由于忽略返回值而产生的。 在您的应用程序中创建一种便于使用的、标准的处理故障方法。如果错误处理过程简单直接，往往不容易被程序员忽略。一种标准化的错误处理方法是，将围绕检查和处理错误状况的常用函数写入封装器，而无需程序员的其他干预。实施并采用封装器后，就可以禁止使用未封装的函数，并利用自定义规则加以执行。

返回目录

	例 2：以下代码实现了对 malloc() 的封装，从而能检查 malloc() 的返回值是否为 NULL，而在内存分配失败的情况下自动退出。
	void *checked_malloc (size_t size) {
	void *ptr;
	ptr= malloc(size);
	if (ptr == NULL) {
	fprintf (stderr, "Out of memory: %s:%d",
	__FILE__,__LINE__);
	exit(-1);
	}
	return ptr;
	}
	Example 2 使用了内存较低情况下最为简单的错误处理机制：强制终止应用程序。根据发生错误的环境情况，其他处理行为可能会更合适。例如，可以考虑放弃相对来说不那么重要的操作，使其释放当前内存，从而使比较重要的应用程序能够成功获取所需的内存空间。根据应用程序以及运行它的系统，等待更多可用内存空间可能也是一个可行的选择。不管怎样，在大多数的情况下，低内存环境下错误处理的关键是日志条目，它有助于准确地诊断问题，并尽力避免将来再次出现这些问题。

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Missing Precision**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	None

返回目录

漏洞名称	**MultiByte String Length**
默认严重性	4
摘要	Y（文件） 文件第 M 行中的函数 YY（元素） 为缓冲区分配了不正确的计算大小，导致写入值与写入的缓冲区大小不匹配。
解释	不正确的内存分配可能导致意外值覆写部分内存，从而发生意外行为。在某些情况下，如果攻击者可以控制内存分配和写入的值，就可能导致被执行恶意代码。 某些内存分配函数需要以参数的方式提供一个大小值。分配的大小应该使用提供的值导出，方法是使用提供的预期源的长度值乘以该长度的大小。如果未能通过正确的算法计算出准确的大小，就可能导致源溢出目标。
建议	一定要使用正确的算法计算大小。 对于内存分配来说，就是要根据分配源计算分配大小： 根据预期源的长度导出大小值，以确定要处理的单元数量。 一定要从编程的角度考虑各个单元的大小及其到内存单位的转换——例如，对单元类型使用 sizeof。 内存分配应该是所写单元数量的倍数，乘以每个单元的大小。
CWE	CWE ID 135
OWASP2017	A1-Injection

返回目录

漏洞名称	**Null Dereference**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数如果间接引用第 N 行上的 null 指针，将导致程序崩溃。该程序可能会间接引用一个 null 指针，从而造成分段故障。

解释	如果不符合程序员的一个或多个假设，则通常会出现 null 指针异常。此问题至少有三种类型：check-after-dereference、dereference-after-check 和 dereference-after-store。如果程序在检查可能为 null 的指针是否为 null 之前间接引用该指针，则会发生 check-after-dereference 错误。如果程序明确检查过 null，并确定该指针为 null，但仍继续间接引用该指针，则会出现 dereference-after-check 错误。此类错误通常是由于错别字或程序员疏忽造成的。如果程序明确将指针设置为 null，但稍后却间接引用该指针，则将出现 dereference-after-store 错误。此错误通常是因为程序员在声明变量时将该变量初始化为 null 所致。
	在这种情况下，在 N 行间接引用该变量时，该变量有可能为 null，从而引起分段故障。
	大多数 null 指针问题会导致一般软件可靠性问题，但如果攻击者可能有意触发 null 指针间接引用，他们可以使用生成的异常绕过安全逻辑以发动拒绝服务攻击，或使应用程序显示调试信息，这些信息在规划后续攻击时十分有用。
	示例 1：在下列代码中，程序员假设变量 ptr 不是 NULL。当程序员间接引用该指针时，这个假设就会清晰的体现出来。当程序员检查 ptr 是否为 NULL 时，就会与该假设发生矛盾。当在 if 语句中检查时，如果 ptr 可以为 NULL，则在其间接引用时也将为 NULL，并引起 segmentation fault。
	ptr->field = val;
	...
	if (ptr != NULL) {
	...
	}
	示例 2：在下列代码中，程序员会确认变量 ptr 为 NULL，然后错误地对其进行间接引用。如果在 if 语句中检查 ptr 时其为 NULL，则会发生 null dereference，从而导致分段故障。
	if (ptr == null) {
	ptr->field = val;
	...
	}
	示例 3：在下列代码中，程序员忘记了字符串 '\0' 实际上为 0 还是 NULL，从而间接引用 null 指针并引发分段故障。
	if (ptr == '\0') {

返回目录

	*ptr = val;
	...
	}
	示例 4：在下列代码中，程序员会将变量 ptr 明确设置为 NULL。之后，程序员会间接引用 ptr，而未检查对象是否为 null 值。
	*ptr = NULL;
	...
	ptr->field = val;
	...
	}

建议	由于间接引用 NULL 指针引发的安全问题大多数情况下会表现为拒绝服务攻击。如果攻击者可以持续触发 null 指针间接引用，那么其他用户可能就无法正常地访问应用程序。除了攻击者故意触发分段故障的情况之外，间接引用 null 指针可能会导致程序时不时地崩溃，且这种问题难以跟踪。 在间接引用可能为 null 的对象之前，请务必仔细检查。如有可能，在处理资源的代码的封装器中纳入 null 检查，以确保在所有情况下均会执行该检查，并最大限度地减少出错的位置。
CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**NULL Pointer Dereference**
默认严重性	3
摘要	X（文件） 文件第 N 行 XX（元素） 中声明的变量在被 Y（文件）文件第 M 行的 YY（元素） 使用时未初始化。
解释	Null 指针取消引用可能会导致 run-time 异常、崩溃或其他意外行为。 已声明但未赋值的变量将隐式保留一个 null 值，直到为变量赋一个值。也可显式为变量设置 null 值，以确保清除其内容。因为 null 并不是一个真正的值，它可能没有对象变量和方法，因此访问 null 对象的内容但未提前验证它的设置时会导致 null 指针取消引用异常。
建议	创建任何变量时，请确保声明和使用之间的所有逻辑流都先为变量分配非 null 值。 在取消引用之前对接收到的所有变量和对象强制执行 null 值检查，确保其不包含其他地方分配给的 null 值。 可考虑是否通过分配 null 值覆盖初始化的变量。可考虑重新分配或释放这些变量。
CWE	CWE ID 476
OWASP2017	A1-Injection

返回目录

漏洞名称	**Obsolete**
默认严重性	2.0
摘要	函数 XX（函数） 在第 N 行调用不推荐使用或过时的函数 X(函数)。使用 X(函数) 表明存在忽略的代码。使用不推荐的或过时的函数可能表示这是一段被忽略的代码。

解释	随着编程语言的发展，一些函数有时会被弃用，原因如下：
	— 为了改进该编程语言。
	— 对操作的有效性、安全性有更深一步的了解。
	—某些操作的管理规则发生了变化。
	在编程语中，函数会经常被删除或由新的替代函数所取代，因为新的函数能以我们所期望的方式从多种角度执行相同的任务。
	在这种情况下，X(文件)第 N 行的 X（函数） 函数被弃用或不推荐使用。请参阅 X（函数） 以确定该函数为什么被弃用或不推荐使用，并了解如何用其他可选方法实现相同的功能。本文余下的部分会讨论由于使用被弃用或不推荐使用的函数而带来的常见问题。
	示例：以下代码使用了不被推荐的函数 getpw() 来验证明文密码是否与用户加密密码相匹配。如果密码是有效的，则函数将 result 设为 1；如果无效，将其设为 0。
	...
	getpw(uid, pwdline);
	for (i=0; i<3; i++){
	cryptpw=strtok(pwdline, ":");
	pwdline=0;
	}
	result = strcmp(crypt(plainpw,cryptpw), cryptpw) == 0;
	...
	尽管代码经常正确地运行，使用 getpw() 函数从安全角度来说是有问题的，因为它可以溢出传递给它的第二个参数的缓冲区。因为这个漏洞，getpw() 已由 getpwuid() 替代，它与 getpw() 执行相同的查找，但返回一个指向静态分配结构的指针来降低风险。
	并非所有函数都会因为存在安全漏洞而被弃用或被取代。然而，出现被弃用的函数通常表示周围代码已经不起作用了，有可能处于不受维护的状况。在过去很长一段时间内，人们并没有将软件安全放在首位，甚至都未曾考虑过。如果程序使用了不推荐的或过时的函数，在其附近就会潜伏着安全问题。

建议	请不要使用不推荐的或过时的函数。不管是否会对安全产生直接影响，都要使用最新的函数来代替这些过时的函数。当您遇到过时的函数时，应意识到它的出现可能会给周围的代码带来安全隐患。请考虑进行应用程序开发时所依据的有关安全方面的各种假设。它们是否仍然有效？使用特定的过时函数是否会带来更大的维护问题？
CWE	CWE ID 477

返回目录

OWASP2017	None

返回目录

漏洞名称	**Obsolete:Inadequate Pointer Validation**
默认严重性	4.0
摘要	函数 X(函数) 已过时，不能保证指针是有效的，或引用的内存可以安全使用。函数已过时，不能保证指针是有效的，或引用的内存可以安全使用。

解释	不使用 IsBadXXXPtr() 类的函数有多种原因。这些函数是:
	1) 非线程安全的。
	2) 通常与由于其探测无效内存地址而导致的崩溃有关。
	3) 被错误地认为在异常条件中执行正确的错误处理。
	示例：以下代码使用 IsBadWritePtr() 尝试避免错误的内存写入。
	if (IsBadWritePtr(ptr, length))
	{
	[handle error]
	}
	程序员原本希望使用这些函数来检测异常情况，但它们通常导致更多无法修复的问题。

建议	不要使用 IsBadXXXPtr 类的函数。不要尝试处理无效内存访问或引用并继续操作，而是让程序以容易调试的方式失败。捕捉错误，并让该错误在以后任意时间自己暴露出来，这是解决问题的一种方法。
CWE	CWE ID 730
OWASP2017	None

返回目录

漏洞名称	**Off by One Error**
默认严重性	5
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	None

返回目录

漏洞名称	**Off by One Error in Arrays**
默认严重性	5
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	A1-Injection

返回目录

漏洞名称	**Off by One Error in Loops**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	A1-Injection

返回目录

漏洞名称	**Off by One Error in Methods**
默认严重性	4
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	A1-Injection

返回目录

漏洞名称	**Often Misused:Authentication**
默认严重性	2.0
摘要	调用 X(函数) 而返回的信息不值得信任。攻击者可以欺骗 DNS 条目。勿将 DNS 作为安全性的依据。攻击者可以欺骗 DNS 条目。勿将 DNS 名称作为安全性的依据。

解释	许多 DNS 服务器都很容易被攻击者欺骗，所以应考虑到某天软件有可能会在有问题的 DNS 服务器环境下运行。如果允许攻击者进行 DNS 更新（有时称为 DNS 缓存中毒），则他们会通过自己的机器路由您的网络流量，或者让他们的 IP 地址看上去就在您的域中。勿将系统安全寄托在 DNS 名称上。
	在这种情况下，DNS 信息通过 X(文件) 的第 N 行中的 X（函数）进入程序。
	例 1：以下代码使用 DNS 查找来确定输入请求是否来自可信赖的主机。如果攻击者可以攻击 DNS 缓存，那么他们就会获得信任。
	struct hostent *hp;
	struct in_addr myaddr;
	char* tHost = "trustme.trusty.com";
	myaddr.s_addr=inet_addr(ip_addr_string);
	hp = gethostbyaddr((char *) &myaddr,
	sizeof(struct in_addr), AF_INET);
	if (hp && !strncmp(hp->h_name, tHost, sizeof(tHost))) {
	trusted = true;
	} else {
	trusted = false;
	}
	IP 地址相比 DNS 名称而言更为可靠，但也还是可以被欺骗的。攻击者可以轻易修改要发送的数据包的源 IP 地址，但是响应数据包会返回到修改后的 IP 地址。为了看到响应的数据包，攻击者需要在受害者机器与修改的 IP 地址之间截取网络数据流。为实现这个目的，攻击者通常会尝试把自己的机器和受害者的机器部署在同一子网内。攻击者可能会巧妙地采取源地址路由的方法来回避这一要求，但是在今天的互联网上通常会禁止源地址路由。总而言之，核实 IP 地址是一种有用的 authentication 方式，但不应仅使用这一种方法进行 authentication。

建议	如果通过域名检查的方式可以确保主机接受和发送的 DNS 记录的一致性，您可以更加信任这一方式。攻击者如若不能控制目标域的域名服务器，就无法同时欺骗接受和发送的 DNS 记录。然而这种方法并不安全：攻击者也许可以说服域注册者把域移交给一个恶意的域名服务器。依赖于 DNS 记录的 authentication 是有风险的。 虽然没有十分简单的 authentication 机制，但是还有比基于主机的 authentication 更好的方法。密码系统提供了比较不错的安全性，但是这种安全性却易受密码选择不当、不安全的密码传送和 password

返回目录

	management 失误的影响。类似于 SSL 的方法值得考虑，但是通常这样的方法过于复杂，以至于使用时会有运行出错的风险，而关键资源也随时面临着被窃取的危险。在大多数情况下，包括一个物理标记的多重 authentication 可以在合理的代价范围内提供最大程度的安全保障。
CWE	CWE ID 247, CWE ID 292, CWE ID 558, CWE ID 807
OWASP2017	A2 Broken Authentication

返回目录

漏洞名称	**Often Misused:Exception Handling**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数会调用第 N 行的 _alloca()，这会抛出一个堆栈溢出异常，可能造成程序的崩溃。_alloca() 函数会抛出一个堆栈溢出异常，可能造成程序的崩溃。
解释	_alloca() 函数在堆栈上分配存储空间。如果分配请求对于可用的堆栈空间来说太大，_alloca() 将会抛出一个异常。如果未能捕获到异常，程序将会崩溃，并有可能引起 denial of service 攻击。 在这种情况下，X(文件)的第 N 行调用 _alloca()。 在 Microsoft Visual Studio 2005(R) 中，_alloca() 已经被淘汰。它已由更加安全的 _alloca_s() 所取代。
建议	应当避免使用 _alloca()。如果您必须使用 _alloca()，请务必使用一个异常处理器来封装调用，以捕获有可能出现的 overflow exception 异常。 在 Windows XP 中，如果在 try/catch 块中调用 _alloca()，您必须在 catch 块中调用 _resetstkoflw() 以便从堆栈溢出中恢复。 不要调用由 _alloca() 返回的指针上的 free() 函数。因为 _alloca() 是在堆栈上（而不是在堆上）分配内存空间，因此它并不在堆内存管理基础结构的管理之下。
CWE	CWE ID 248
OWASP2017	None

返回目录

漏洞名称	**Often Misused:File System**
默认严重性	2.0
摘要	将一个长度不合适的输出缓冲区传递到 X(函数) 会导致 buffer overflow。将一个长度不合适的输出缓冲区传递到一个 path manipulation 函数，会导致 buffer overflow。

解释	Windows 提供了大量的实用程序函数来处理包含文件名的缓冲区。在大多数情况下，结果会返回到一个作为输入传入的缓冲区中。（通常文件名会进行适当的修改。）大多数函数需要缓冲区至少为 MAX_PATH 字节的长度，但是您应该逐一检查每一个函数的文档。如果缓冲区大小不足以存储处理的结果，那么就会发生 buffer overflow。
	示例:
	char *createOutputDirectory(char *name) {
	char outputDirectoryName[128];
	if (getCurrentDirectory(128, outputDirectoryName) == 0) {
	return null;
	}
	if (!PathAppend(outputDirectoryName, "output")) {
	return null;
	}
	if (!PathAppend(outputDirectoryName, name)) {
	return null;
	}
	if (SHCreateDirectoryEx(NULL, outputDirectoryName, NULL)
	!= ERROR_SUCCESS) {
	return null;
	}
	return StrDup(outputDirectoryName);
	}
	在这个例子中，函数在当前目录中创建了名为 "output\<name>" 的目录，并且返回了一个同样名称的堆分配副本。对于大多数当前目录和名称参数的值来说，该函数都能够正常工作。但是，如果 name 参数特别长，那么第二个对 PathAppend() 的调用可能会溢出 outputDirectoryName 缓冲区，因为该缓冲区比 MAX_PATH 字节小。

建议	应当确保所有持有路径信息的缓冲区的长度至少是 MAX_PATH 字节。使用 MAX_PATH 常量来声明缓冲区。这样可以简化代码审计，并且能够防止因为不正确的值和环境的改变而引发的错误。如果将函数的参数直接传递给 path manipulation 函数，就需要一定长度的文档缓冲区。
CWE	CWE ID 249, CWE ID 560

返回目录

OWASP2017	None

返回目录

漏洞名称	**Often Misused:Privilege Management**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数没有遵守最低权限这一原则，会增加引发其他漏洞的风险。没有遵守最低权限原则会增加引发其他漏洞的风险。
解释	利用 root 权限运行的程序已经造成了无数的 Unix 安全灾难。仔细检查您的程序授权是否会引发安全问题十分必要，同样重要的是，对那些授予了权限的应用程序，应及时撤销其权限并返回至未授权状态，把因忽略漏洞而引发的破坏控制到最小。 在这种情况下，调用的 privilege management 函数是 X(文件) 中第 N 行的 X（函数）。 Privilege management 函数有时会以一些不明显的方式运行，并且它们在不同的平台上面会有较大差异。当您从一个非 root 用户切换到另外一个用户时，这种差异会尤其明显。 信号处理程序以及产生的进程会在其本身进程所具有的权限运行，所以当触发某一个信号或执行子进程时，一个进程以 root 权限运行，信号处理程序或者子进程将会以 Root 权限运行。因此，攻击者有可能利用这个提高的权限来进行更严重的破坏。

建议	如果程序经过重写后可不需要 root 访问权限，请进行重写。
	请在提高权限之前关闭信号，以避免信号处理代码会以意外的权限运行。在将权限降低回到用户权限后，可重新启用信号。
	请尽可能在完成需要特定权限的操作后，通过立即调用一个非零参数的 setuid() 来丢弃权限。
	在有些情况下，应用程序可能无法调用 setuid()/setgid() 来丢弃权限。尤其当应用程序需要不断地以 root 身份代表用户来执行某些操作时，会发生这种情况。比如，一个 FTP 进程需要绑定到 1 至 1024 之间的端口以便于为用户请求服务。在这样的情况下，seteuid() 和 getegid() 函数的使用应该暂时降低到一个较低权限等级，并且保留在需要时返回为 root 的能力。请注意，这同样会使攻击者可以很容易地把应用程序退回到 root 权限，所以当无法完全丢弃权限的时候，应该只使用 seteuid()/getegid()。
	以下程序大体勾勒出了构造较好的 setuid root 程序。
	
	int main(int argc, char** argv) {
	uid_t runner_uid = getuid();
	uid_t runner_gid = getgid();
	uid_t owner_uid = geteuid();
	uid_t owner_gid = getegid();
	int sigmask;

返回目录

	/* Drop privileges right up front, but
	we'll need them back in a little bit,
	so use effective id */
	if (setreuid(owner_uid, runner_uid) ||
	setregid(owner_gid, runner_gid)) {
	exit(-1);
	}
	/* privilege not necessary or desirable at this point */
	processCommandLine(argc, argv);
	/* disable signal handling */
	sigmask = sigprocmask(~0);
	/* Take privileges back */
	if (setreuid(runner_uid, owner_uid) ||
	setregid(runner_gid, owner_gid)) {
	exit(-1);
	}
	openSocket(88); /* requires root */
	/* Drop privileges for good */
	if (setuid(runner_uid) || setgid(runner_gid)) {
	exit(-1);
	}
	/* re-enable signals */
	sigprocmask(sigmask);
	doWork();
	}
	
	另外一个既能够保留权限又能最小化风险的办法是把程序分割成需要权限和不需要权限的部分。创建两个进程：一个进程会在没有权限的状态下完成大部分工作，另外一个进程持有权限，但仅在其他进程要求时才会执行十分有限的操作。
	Chen 和 Wagner 提供一个封装函数，该函数有一个一致且易于理解的接口来进行 privilege management [1]。

CWE	CWE ID 250
OWASP2017	None

返回目录

漏洞名称	**Often Misused:Strings**
默认严重性	2.0
摘要	X(文件) 文件中的函数 XX（函数） 会调用第 N 行的 X(函数)。_mbs 家族函数在处理不符合标准的多字节字符串时，很容易产生 buffer overflow。_mbs 家族函数在处理不符合标准的多字节字符串时，很容易产生 buffer overflow。
解释	Windows 提供了 _mbs 函数系列以对多字节字符串执行各种操作。如果为这些函数传递了一个格式不正确的多字节字符串（如包含一个有效首字节并后跟一个 null 字节的字符串），它们就可能读取或写入到该字符串缓冲区末尾之后的内容，从而造成缓冲区溢出。以下函数都存在发生缓冲区溢出的风险： _mbsinc _mbsdec _mbsncat _mbsncpy _mbsnextc _mbsnset _mbsrev _mbsset _mbsstr _mbstok _mbccpy _mbslen
建议	使用 _mbs 家族函数操作多字节字符串时，应当格外小心。避免使用这些函数来操作不规则的字符串。Microsoft 推荐使用 isleadbyte()、_ismbslead()、ismbstrail() 和 _mbbtype() 函数来测试开头和后续字节的有效性，并标识出无效的多字节字符串 [2]。
CWE	CWE ID 176, CWE ID 251
OWASP2017	None

返回目录

漏洞名称	**Out-of-Bounds Read**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数在第 N 行从 X（缓冲区） 边界之外读取数据。该程序从分配的内存边界之外读取数据。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这种情况下，该程序会从分配的内存边界之外进行读取，这会允许攻击者访问敏感信息、引入错误的行为，或者造成程序崩溃。
	例 1：在以下代码中，memcpy() 调用会从分配的 cArray 边界之外读取内存，其中包含 char 类型的 MAX 元素，而 iArray 包含 int 类型的 MAX 元素。
	void MemFuncs() {
	char array1[MAX];
	int array2[MAX];
	memcpy(array2, array1, sizeof(array2));
	}
	例 2：以下短程序在使用分析用的字元常数 memchr() 时，将一个不可信赖的命令行参数作为搜索缓冲区使用。
	int main(int argc, char** argv) {
	char* ret = memchr(argv[0], 'x', MAX_PATH);

返回目录

	printf("%s\n", ret);
	}
	该程序本来是要通过搜索 argv[0] 数据直至一个常数字节来输出 argv[0] 的一个子字符串。然而，因为（常数）字节数可能大于为 argv[0] 分配的数据，搜索可能在分配给 argv[0] 的数据之外进行。当在 argv[0] 中无法找到 x 时会发生这种情况。

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	CCI-002754, CCI-002824
OWASP2017	None

返回目录

漏洞名称	**Out-of-Bounds Read:Off-by-One**
默认严重性	2.5
摘要	X(文件) 中的 XX（函数） 函数在第 N 行从 X（缓冲区） 边界之外读取数据。程序从分配的内存边界之外读取数据。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的 off-by-one 错误在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆栈 buffer overflow 和堆 buffer overflow 等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这种情况下，该程序会从分配的内存边界之外进行读取，这会允许攻击者访问敏感信息、引入错误的行为，或者造成程序崩溃。
	示例：以下代码连续地间接引用 char 五元数组，最后一个引用会引入 off-by-one 错误。
	char Read() {
	char buf[5];
	return 0
	+ buf[0]
	+ buf[1]
	+ buf[2]
	+ buf[3]
	+ buf[4]
	+ buf[5];
	}

返回目录

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Out-of-Bounds Read:Signed Comparison**
默认严重性	2.5
摘要	X(文件) 中的函数 XX（函数） 在一个带符号的比较中使用变量，但该变量稍后被视为是不带符号。这会导致该程序在第 N 行的 X（缓冲区） 边界之外使用内存。该程序使用带符号的比较来检查稍后会被视为不带符号的值。这会导致程序从分配的内存边界之外读取数据。

解释	Buffer overflow 可能是人们最熟悉的一种软件安全漏洞。虽然绝大多数软件开发者都知道什么是 Buffer overflow 漏洞，但是无论是对继承下来的或是新开发的应用程序来说，Buffer overflow 攻击仍然是一种最常见的攻击形式。对于这个问题出现的原因，一方面是造成 buffer overflow 漏洞的方式有很多种，另一方面是用于防止 buffer overflow 的技术也容易出错。
	在一个典型的 buffer overflow 攻击中，攻击者将数据传送到某个程序，程序会将这些数据储存到一个较小的堆栈缓冲区内。结果，调用堆栈上的信息会被覆盖，其中包括函数的返回指针。数据会被用来设置返回指针的值，这样，当该函数返回时，函数的控制权便会转移给包含在攻击者数据中的恶意代码。
	虽然这种类型的堆栈 buffer overflow 在某些平台和开发组织中十分常见，但仍不乏存在其他各种类型的 buffer overflow，其中包括堆 buffer overflow 和 off-by-one 错误等。有关 buffer overflow 如何进行攻击的详细信息，许多优秀的著作都进行了相关介绍，如 Building Secure Software [1]、Writing Secure Code [2] 以及 The Shellcoder's Handbook [3]。
	在代码层上，buffer overflow 漏洞通常会违反程序员的各种假设。C 和 C++ 中的很多内存处理函数都没有执行边界检查，因而可轻易地超出缓冲区所操作的、已分配的边界。即使是边界函数（如 strncpy()），使用方式不正确也会引发漏洞。对内存的处理加之有关数据段大小和结构方面所存在种种错误假设，是导致大多数 buffer overflow 漏洞产生的根源。
	在这种情况下，该程序会从分配的内存边界之外进行读取，这会允许攻击者访问敏感信息、引入错误的行为，或者造成程序崩溃。
	示例：以下代码尝试通过检查从 getInputLength() 中读取的不可信的值，验证其是否小于目标缓冲区 output 的大小，来避免从边界之外进行读取的 buffer overflow。然而，因为 len 和 MAX 之间比较的是带符号的值，所以如果 len 为负值，在其转换为 memcpy() 不带符号的参数时，将会变成一个超级大的正数。
	void TypeConvert() {
	char input[MAX];
	char output[MAX];
	fillBuffer(input);
	int len = getInputLength();

返回目录

	if (len <= MAX) {
	memcpy(output, input, len);
	}
	...
	}

建议	虽然谨慎使用边界函数能够大大降低 buffer overflow 的风险，但这种移植也不能盲目地进行，而且依靠它自己来确保安全是远远不够的。 当您利用内存时（特别是字符串），切记 buffer overflow 漏洞通常会出现在以下代码中： — 依靠外部的数据来控制行为的代码。 — 受数据属性影响的代码，该数据在代码的临接范围之外执行。 — 代码过于复杂，以致于程序员无法准确预测它的行为。 另外，还要考虑到以下原则： — 永远不要相信外部资源会为内存操作提供正确的控制信息。 — 永远不要相信程序会在执行过程中维护正在处理中的数据所带有的属性。进行操作之前，对这些数据执行健全性检查。 — 限制内存处理和边界检查代码的复杂度。保持这些代码的简单性，并清楚地记录已执行的检查、已测试的假设以及在输入验证失败后，希望程序执行的操作。 — 如果输入的数据太大，截短数据要十分小心，然后再对这些数据继续处理。截短数据会改变输入的含义。 — 不要依赖诸如 StackGuard 之类的工具，或非可执行堆栈来阻止 buffer overflow 漏洞。这些方法不能应对堆 buffer overflow 以及更加敏感的堆栈溢出，该堆栈溢出可以改变控制程序的变量内容。另外，许多这样的方法都很容易破解，即使它们运行比较到位，也只能够找出问题的征兆，而不能发现问题的原因。
CWE	APSC-DV-002590 CAT I
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	3.0
摘要	X(文件) 中的 XX（函数） 函数在第 N 行中使用明文密码。采用明文的形式存储密码会危及系统安全。以明文形式存储密码可能会危及系统安全。
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数据存储中时，可能会发生 Password Management 漏洞。 在这种情况下，密码会通过 X（文件） 中第 N 行的 X（函数） 读取到程序中，并用于访问 Y（文件） 中第 M 行的 Y（函数） 资源。 示例：以下代码会从注册表中读取密码，并使用该密码来连接至数据库。 ... RegQueryValueEx(hkey,TEXT(.SQLPWD.),NULL, NULL,(LPBYTE)password, &size); rc = SQLConnect(*hdbc, server, SQL_NTS, uid, SQL_NTS, password, SQL_NTS); ... 该代码可以正常运行，但是任何对用于存储密码的注册表项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。大多数应用服务器通常会提供过时的、相对较弱的加密机制，而这在对安全性要求较高的环境中是远远不够的。 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 256
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Password Management:Empty Password**
默认严重性	4.0
摘要	Empty password 可能会危及系统安全，并且无法轻易修正出现的安全问题。

解释	为密码变量指定空字符串绝非一个好方法。如果使用 empty password 成功通过其他系统的验证，那么相应帐户的安全性很可能会被减弱，原因是其接受了 empty password。如果在为变量指定一个合法的值之前，empty password 仅仅是一个占位符，那么它将给任何不熟悉代码的人造成困惑，而且还可能导致出现意外控制流路径方面的问题。
	在这种情况下，在对 X(文件)第 N 行中的 X（函数） 的调用中发现空密码。
	示例 1：以下代码尝试使用空密码连接到数据库。
	...
	rc = SQLConnect(*hdbc, server, SQL_NTS, "scott", SQL_NTS, "", SQL_NTS);
	...
	如果 Example 1 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。
	示例 2：以下代码会将密码变量初始化为空字符串，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。
	...
	char *stored_password = "";
	readPassword(stored_password);
	if(safe_strcmp(stored_password, user_password))
	// Access protected resources
	...
	}
	...
	如果 readPassword() 因数据库错误或其他问题而未能取得存储的密码，攻击者只需向 user_password 提供一个空字符串，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:Hardcoded Password**
默认严重性	4.0
摘要	Hardcoded password 可能会危及系统安全性，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：以下代码使用硬编码密码来连接数据库： ... rc = SQLConnect(*hdbc, server, SQL_NTS, "scott", SQL_NTS, "tiger", SQL_NTS); ... 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。如果攻击者能够访问应用程序的可执行代码，他们就能对包含所用密码值的代码进行反汇编。
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。 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, 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	2.0
摘要	Null password 会损害安全性。

解释	最好不要为密码变量指定 null，因为这可能会使攻击者绕过密码验证，或是表明资源受空密码保护。
	在这种情况下，在对 X(文件) 第 N 行的 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。
	...
	char *stored_password = NULL;
	readPassword(stored_password);
	if(safe_strcmp(stored_password, user_password))
	// 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 user_password 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	2.0
摘要	以明文形式在系统或系统代码中存储密码或密码详细信息可能会以无法轻松修复的方式危及系统安全。

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

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

返回目录

漏洞名称	**Password Management:Weak Cryptography**
默认严重性	3.0
摘要	调用 X(函数) 给密码加密并不能提供任何意义上的保护。采用普通的编码方式给密码加密并不能有效地保护密码。
解释	当密码以明文形式存储在应用程序的配置文件或其他数据存储中时，会发生 password management 漏洞。程序员试图通过编码函数来遮蔽密码，以修补 password management 漏洞，例如使用 64 位基址编码方式，但都不能起到充分保护密码的作用。 在这种情况下，密码会通过 X（文件） 中第 N 行的 X（函数） 读取到程序中，并用于访问 Y（文件） 中第 M 行的 Y（函数） 资源。 示例：以下代码会从注册表中读取密码，使用一种简单的编码算法将密码解码，然后使用该密码连接至数据库。 ... RegQueryValueEx(hkey, TEXT(.SQLPWD.), NULL, NULL, (LPBYTE)password64, &size64); Base64Decode(password64, size64, (BYTE*)password, &size); rc = SQLConnect(*hdbc, server, SQL_NTS, uid, SQL_NTS, password, SQL_NTS); ... 该代码可以正常运行，但是任何对用于存储密码的注册表项具有访问权限的人都能读取 password64 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	决不能使用明文存储密码，也不能仅使用简单的编码方式。相反，应在系统启动时，由管理员输入密码。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。大多数应用服务器通常会提供过时的、相对较弱的加密机制，而这在对安全性要求较高的环境中是远远不够的。 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 261
OWASP2017	A3 Sensitive Data Exposure

返回目录

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

解释	当满足以下两个条件时，就会产生 path manipulation 错误：
	1.攻击者能够指定某一文件系统操作中所使用的路径。
	2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。
	例如，在某一程序中，攻击者可以获得特定的权限，以重写指定的文件或是在其控制的配置环境下运行程序。
	在这种情况下，攻击者可以指定通过 X（文件） 中第 N 行的 X（函数） 进入程序的值，这一数值可以通过 Y（文件） 中第 M 行的 Y（函数） 访问文件系统资源。
	示例 1：以下代码利用来自 CGI 请求的输入生成一个文件名。程序员没有考虑到攻击者可能使用像“../../apache/conf/httpd.conf”一样的文件名，从而导致应用程序删除特定的配置文件。
	char* rName = getenv("reportName");
	...
	unlink(rName);
	示例 2：以下代码使用来自于命令行的输入来决定该打开哪个文件，并返回到用户。如果程序以足够的权限运行，并且恶意用户能够创建指向文件的软链接，那么他们可以使用程序来读取系统中任何文件的开始部分。
	ifstream ifs(argv[0]);
	string s;
	ifs >> s;
	cout << s;

建议	防止 Path Manipulation 的最佳方法是采用一些间接手段：创建一个必须由用户选择的合法值的列表。通过这种方法，就不能直接使用用户提供的输入来指定资源名称。 但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大，维护难度过大。因此，在这种情况下，程序员通常会采用执行拒绝列表的办法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。但是，任何这样一个列表都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一个字符列表，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	CWE ID 22, CWE ID 73
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Path 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	**PBKDF2 Insufficient Iteration Count**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的凭证是使用不足的输出长度保存在 Y（文件） 文件第 M 行的 YY（元素） 中 Y（文件） 文件第 M 行 YY（元素） 中定义的迭代参数过低。
解释	使用不安全的凭证存储方式可能会暴露用户和/或管理员的凭证，从而受到暴力破解、彩虹表和其他类型的攻击。 即使应用程序使用安全的凭证存储机制，仍然有可能会以不安全的方式实施解决方案，从而降低所存储凭证的安全性。 在这种情况下，密钥派生函数的输出长度太低。这可能导致冲突，使不同密码的 hash 计算结果相同，造成身份验证成功，而这是不应该发生的。 将 PBKDF2 迭代参数设置为过低的值会显著降低生成的 hash 的安全性。 用于存储凭证的方法不安全。 密钥派生函数的输出长度太低。 PBKDF2 函数的迭代参数过低。
建议	应使用密码 hash 函数小心地完成凭证（即密码）的存储。大多数 hash 函数都是以速度为前提创建的，用于为文件完整性和其他目的创建加密的安全 hash，使它们不适合直接用于密码；而其他函数则在过去因冲突或其他安全问题已被破坏（MD5、SHA-1 等…）。 我们推荐的安全凭证存储函数有： Argon2（最近的密码 Hash 竞赛（PHC）获奖者） scrypt bcrypt PBKDF2（仅为提供旧版兼容性） 然而，使用这些函数无法保证安全的凭证存储机制。为了避免冲突，密钥派生函数输出的安全长度应为 32 字节或更高。 PBKDF2 的迭代参数至少应设置为 10.000 或更高的值。降低此值会降低根据密码计算 hash 所需的计算工作量，从而削弱产生的 hash，使攻击者更容易破解密码。
CWE	CWE ID 522
OWASP2017	None

返回目录

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

返回目录

漏洞名称	**Personal Information Without Encryption**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的敏感数据是使用不安全的机制保存在 Y（文件） 文件第 M 行的 YY（元素） 中。
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 处理敏感数据时，一定要为其加密：无论是永久存储还是传输都应如此。 不加密个人身份信息可能会导致违反合规性要求。另外攻击者也会收集这些数据以泄露这些信息或将其用于恶意目的。 应用程序的加密相关功能有问题。 在这种情况下，不会对储存的敏感信息进行加密。
建议	如果必须保存敏感数据（无论是在客户端还是服务器上），则应以加密的方式存储，而不能使用纯文本。 这些数据的加密密钥应以安全的方式使用用户密码派生，不应与数据保存到一起或硬编码到应用程序的代码中。
CWE	CWE ID 311
OWASP2017	None

返回目录

漏洞名称	**Plaintext Storage Of A Password**
默认严重性	4
摘要	YYY（方法） 中用于验证身份时进行对比的密码以纯文本的方式保存在 Y（文件） 中。
解释	以纯文本的方式保存在数据库中的密码会轻易被有访问权限的攻击者获取到。 敏感信息被以纯文本的方式保存在数据库中。
建议	通用指南: 请不要使用纯文本格式存储任何敏感信息，例如数据库密码。 使用合适的加密方法安全地加密应用程序密码。如果原始密码必须是可检索的，例如要连接数据库，请使用有强密钥和随机 IV 的 AES-CBC 或 GCM。 使用合适的密钥管理，包括动态生成随机密钥、保护密钥、并根据需要更换密钥。 也可使用平台机制或硬件设备加密密码。
CWE	CWE ID 256
OWASP2017	None

返回目录

漏洞名称	**Poor Style:Redundant Initialization**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数从不使用在第 N 行为变量 Y(变量) 指定的初始值。变量赋值后并不使用，而变成一个死存储。

解释	没有使用该变量的初始值。初始化之后，变量或者被重新赋值，或者转向作用域之外。
	示例：以下摘录的代码为变量 r 赋值，并在没有使用所赋数值的情况下，对其加以重写。
	int r = getNum();
	r = getNewNum(buf);

建议	为了使代码易于理解和维护，删除不必要的赋值。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Poor Style:Value Never Read**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数从来不使用第 N 行中为变量 Y(变量) 指定的值。变量赋值后并不使用，而变成一个死存储。

解释	没有使用该变量的值。赋值之后，变量或者被重新赋值，或者超出范围之外。
	示例：以下摘录的代码为变量 r 赋值，并在没有使用所赋数值的情况下，对其加以重写。
	r = getName();
	r = getNewBuffer(buf);

建议	为了使代码易于理解和维护，删除不必要的赋值。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Poor Style:Variable Never Used**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数声明了变量 X（变量），但从未使用。从未使用过该变量。

解释	从未使用过该变量。有可能该变量只是一个残留的无用代码，但也可能该未使用的变量会指出一个 bug。
	示例：在以下代码中，复制粘贴错误导致同一个循环迭代器 (i) 使用了两次。而变量 j 却从未使用。
	int i,j;
	for (i=0; i < outer; i++) {
	for (i=0; i < inner; i++) {
	...

建议	通常，您应该删除那些未使用的变量，使代码更容易理解和维护。
CWE	None
OWASP2017	None

返回目录

漏洞名称	**Portability Flaw**
默认严重性	2.0
摘要	调用第 N 行的 X(函数) 可能会导致可移植性问题，因为它在不同的操作系统和操作系统版本下具有不同的语义。在多个操作系统和操作系统版本之间，执行语义不一致的函数会导致可移植性问题。
解释	这一类函数的行为会因操作系统而异，甚至有时还会受操作系统版本的影响。函数的实现差异主要包括: — 参数解析方式的细微差异会导致结果不一致。 — 一些函数的实现本身就包含了很大的安全风险。 — 并非所有平台上都定义了函数。 在这种情况下，不一致的函数为 X(文件)中第 N 行的 X（函数）。
建议	为了能编写可移植代码，要避免使用实现方式不一致的函数。 某些情况下能够减轻与函数相关的问题。例如，如果您执行了足够的输入校验，就可以在调用 getopt() 时防止内部的 buffer overflow。 然而通常情况下，使用某些函数时您是无法避免这些问题的。这样的例子包括: - vfork() 的实现会因不同的平台而异。 - strcmpi() 并没有在某些 Unix 系统上定义。 - memmem() 会因版本的变化（参数的顺序被颠倒）而出现问题。 一般来说，可以把实现不一致的函数调用替换成较为安全的相应函数调用。
CWE	CWE ID 474
OWASP2017	None

返回目录

漏洞名称	**Potential Off by One Error in Loops**
默认严重性	3
摘要	Y（文件） 文件第 M 行 YY（元素） 的大小与缓冲区的实际大小不一致，导致大小差一错误访问。
解释	差一错误可能导致覆写或过读意外的内存；在大多数情况下，这可能会导致意外行为甚至应用程序崩溃。此外，如果攻击者可以控制分配，就能通过结合变量分配和差一错误导致执行恶意代码。 为内存指定变量时，在确定差一错误的大小或长度时可能发生计算错误。 例如在循环中，分配大小为 2 的数组时，其单元会计为 0,1——因此，如果数组上的 For 循环迭代器被错误地设置为启动条件 i=0 且连续条件 i<=2 ，则会访问三个单元而不是 2，并会尝试写入或读取最初未分配的单元 [2]，可能导致最初分配的数组边界之外的存储器被破坏。 另一个例子是，复制字符数组形式的 null 字节结尾字符串时不会复制结尾的 null 字节。如果没有 null 字节，字符串的表示就没有终止，这会导致某些函数过量读取内存，因为这些函数需要缺少的 null 结尾。
建议	一定要确保给定的迭代边界是正确的： 对于数组迭代，大小为 n 的数组可考虑以单元 0 开头并以单元 n-1 结尾。 对于字符数组和 null 字节结尾的字符串，请考虑 null 字节是必需的，不应覆写或忽略；确保使用中的函数不会发生差一错误，特别是缓存末尾自动附加 null 字节，而非代替最后一个字符的情况。 尽量使用不容易出现差一错误的安全函数管理内存。
CWE	CWE ID 193
OWASP2017	A1-Injection

返回目录

漏洞名称	**Potential Path Traversal**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取动态数据。然后，此元素的值将传递到代码，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于本地磁盘访问的文件路径中。这可能会导致路径遍历漏洞。
解释	攻击者可能为要使用的应用程序定义任意文件路径，可能导致： 窃取敏感文件，例如配置或系统文件 覆写文件，例如程序二进制文件、配置文件或系统文件 删除关键文件，导致拒绝服务 (DoS) 攻击。 应用程序使用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	**Potential Precision Problem**
默认严重性	3
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Privacy Violation**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数错误地处理了第 N 行的机密信息。程序可能危及用户的个人隐私。对机密信息（如客户密码或社会保障号码）处理不当会危及用户的个人隐私，这是一种非法行为。

解释	Privacy Violation 会在以下情况下发生：
	1. 用户私人信息进入了程序。
	在这种情况下，数据来自于 X（文件） 中第 N 行的 X（函数）。
	2. 数据被写到了一个外部介质，例如控制台、file system 或网络。
	这种情况下，数据被传递给 Y（文件） 的第 M 行中的 Y（函数）。
	示例 1：以下代码包含了一个日志记录语句，该语句通过在日志文件中存储记录信息跟踪添加到数据库中的各条记录信息。get_password() 函数可以从存储的其他值中返回一个由用户提供的、与该用户帐户相关的明文密码。
	pass = get_password();
	...
	fprintf(dbms_log, "%d:%s:%s:%s", id, pass, type, tstamp);
	Example 1 中的代码会将明文密码记录到文件系统中。虽然许多开发人员认为文件系统是存储所有数据的安全位置，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，那么这时学号就应被视为私人信息。
	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，这种做法就会带来额外的风险。
	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。然而，尽管某些特定资源已经被限制访问，但仍无法保证所有能够访问该资源的个体都是可以信赖的。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**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法向应用程序外发送用户信息。这可能造成“侵犯隐私”。
解释	用户的个人信息可能被恶意程序员或拦截数据的攻击者窃取。 应用程序将用户信息（例如密码、帐户信息或信用卡号）发送到应用程序外，例如将其写入本地文本或日志文件或将其发送到外部 Web 服务。
建议	应该在将个人数据写入日志或其他文件之前将其删除。 检查将个人数据发送到远程 Web 服务的必要性和理由。
CWE	CWE ID 359
OWASP2017	A3-Sensitive Data Exposure

返回目录

漏洞名称	**Process Control**
默认严重性	4.0
摘要	X(文件) 文件中的函数 XX（函数） 会调用第 N 行的 X(函数)。该调用会从一个不可信赖的数据源或不可信赖的环境中加载库。这种调用会导致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不可信赖的环境中加载库，会导致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：
	– 攻击者能够篡改程序执行的库：攻击者直接控制了库的名称。
	— 攻击者可以篡改库加载的环境：攻击者间接控制库名称的含义。
	在这种情况下，我们着重关注第一种情况，即攻击者有可能控制加载的库的名称。这种类型的 Process Control 漏洞会在以下情况下出现：
	1. 数据从不可信赖的数据源进入应用程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据作为字符串的一部分，代表一个由应用程序加载的库名。
	在这种情况下，库由 Y（文件）中第 M 行的 Y（函数） 加载。
	3. 通过在库中执行代码，应用程序授予攻击者在一般情况下无法获得的权限或能力。
	示例 1：以下代码来自于一个特权应用程序，使用注册表项来决定安装的目录，然后用基于指定目录的相对路径，加载一个库文件。
	...
	RegQueryValueEx(hkey, "APPHOME",
	0, 0, (BYTE*)home, &size);
	char* lib=(char*)malloc(strlen(home)+strlen(INITLIB));
	if (lib) {
	strcpy(lib,home);
	strcat(lib,INITCMD);
	LoadLibrary(lib);
	}
	...
	在这个例子中，代码通过篡改注册表主键来指定一个包含 INITLIB 恶意版本的其他路径，进而允许攻击者加载一个任意库，以提高的应用程序权限去执行库中的任意代码。因为程序不会校验从环境中读取的值，如果攻击者能够控制 APPHOME 的值，就能欺骗应用程序去运行恶意的代码。
	示例 2：以下代码来自于一个基于 web 的管理实用程序，它允许用户访问一个接口，通过这个接口更新用户在系统上的配置文件。该实用程序使用一个名为 liberty.dll 的库，该库通常可在一个标准的系统目录中找到。
	LoadLibrary("liberty.dll");

返回目录

	但是，程序并没有指定 liberty.dll 的绝对路径。在搜索顺序上，如果攻击者将一个名为 liberty.dll 的恶意库放在原本想要的库的前面，并且攻击者能够让程序在他们的环境（而不是 web 服务器环境）中运行，那么应用程序就会加载该恶意库，而不是原本想要的可信赖的库。由于这种类型的应用程序会以提高了的权限运行，因此攻击者的 liberty.dll 中的内容也将以提高的权限运行，这可能会使攻击者完全控制系统。 这种类型的攻击可能是由于在没有指明绝对路径的情况下，LoadLibrary() 所使用的搜索顺序造成的。如果当前目录比系统目录先搜索到，就像现在大多数最新的 Windows 版本，那么如果攻击者可在本地执行程序，这种类型的攻击就会变得十分简单。搜索顺序取决于操作系统的版本，在比较新的操作系统中，这一顺序由注册表主键控制： HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode 该键值没有在 Windows 2000/NT 及 Windows Me/98/95 中定义。在存在此键的操作系统上，LoadLibrary() 按以下方式运行： 如果 SafeDllSearchMode 为 1，搜索顺序如下所示： （这是 Windows XP-SP1 及更高版本，包括 Windows Server 2003 的默认设置。） 1. 应用程序被加载的目录。 2. 系统目录。 3. 16 位系统目录（如果存在）。 4. Windows 目录。 5. 当前目录。 6. 在 PATH 环境变量中列出来的目录。 如果 SafeDllSearchMode 为 0，搜索顺序如下所示： 1. 应用程序被加载的目录。 2. 当前目录。 3. 系统目录。 4. 16 位系统目录（如果存在）。 5. Windows 目录。 6. 在 PATH 环境变量中列出来的目录。
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加载这些库的环境，间接地控制某个程序所使用的库。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。应尽可能由应用程序来控制各个库，并使用绝对路径来进行加载。如果在编译的过程中不知道具体的

返回目录

	路径，如跨平台应用程序，则应该在执行过程中，从已知值中构造一个绝对路径。 因为 Windows API 使用的是一个特定的搜索顺序，这个顺序不仅仅基于一系列目录，而且还基于在没有特别指定时自动添加的文件扩展名列表，攻击者可能会注入一个指定名称的恶意库，该指定名称的扩展名在搜索顺序中更加靠前。因此，同样应该在 Windows 系统上指定文件扩展名的绝对路径。 从配置文件或者环境中读取的库名和路径都应该根据一组定义有效值的常量进行健全性检查。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，以校验库的有效性。如果某个库总是被一些特定用户拥有，或者总是被分配了一些特定的访问权限，那么程序会在加载之前对这些属性进行验证。
CWE	CWE ID 114, CWE ID 494
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Process Control**
默认严重性	5
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 中动态载入外部库；如果攻击者能影响此库，或者将调用更改到另一个库，攻击者可能能够执行任何代码。
解释	如果攻击者可以通过篡改调用或通过操纵动态加载的库来直接影响动态加载哪个库，则攻击者可能会引入自己的代码，从而接管执行过程，甚至接管正在运行的系统。 从代码动态加载库且满足以下情况之一时会发生流程控制漏洞： 攻击者可以替换正在加载的库，从而在应用程序中引入攻击者自己的代码，可能将其破解。 攻击者可以影响正在加载哪个库，从而使攻击者能够加载不安全的库
建议	加载的库应来自受信任的来源。如果库不是来自可靠的来源，请检查其源代码。 应该对所有本机库进行验证，以确定应用程序是否需要使用该库。 使用最小权限原则运行应用程序，以减少受到成功攻击时的影响。 净化影响库调用的所有输入，以避免恶意内容。 调用库时使用绝对路径，以避免调用不需要的库。 载入使用本地调用的库时，验证所有输入以避免缓冲区溢出。
CWE	CWE ID 114
OWASP2017	A1-Injection

返回目录

漏洞名称	**Race Condition:File System Access**
默认严重性	3.0
摘要	利用调用 XXX（函数） 和 YYY（函数） 之间的这段时间间隙，可以发起一次扩大权限的攻击。从检查到文件资源到使用文件的这段时间间隙，可以用来发动一次扩大权限的攻击。

解释	文件访问的 race condition（如已知的 time-of-check、time-of-use (TOCTOU) race condition）在以下情况中出现：
	1. 程序检查某个文件的属性时，根据名称来引用文件。
	在这种情况下，检查由 XX（文件）中第 M 行的 <Replace key="FirstTransitionFunction" link="FirstTraceLocation" /> 执行。
	2. 稍后该程序会使用相同的文件名来执行 file system 操作，并假设之前检查的属性仍然有效。
	然后，该文件会在 X(文件) 中第 N 行的 <Replace key="PrimaryTransitionFunction" link="PrimaryLocation" /> 中进行使用。
	示例 1：以下代码来自一个安装了 setuid root 的程序。该程序以无权限用户的名义执行了特定的文件操作，并使用访问检查以确保程序没有使用其根权限来执行操作，而这些操作对于当前用户来说是不可行的。程序在打开文件和执行必要的操作之前，使用 access() 系统调用来检查运行程序的用户是否有权访问这些指定的文件。
	`if (!access(file,W_OK)) {`
	`  f = fopen(file,"w+");`
	`  operate(f);`
	`  ...`
	`}`
	`else {`
	`  fprintf(stderr,"Unable to open file %s.\n",file);`
	`}`
	对 access() 的调用按照预期计划的那样执行，而且如果运行程序的用户具有编辑文件的必要权限，则会返回 0，反之则会返回 -1。然而，因为 access() 和 fopen() 都是对文件名进行操作，而不是文件句柄，所以当 file 变量传递到 fopen() 时，就无法保证该变量仍能像传递到 access() 时那样引用磁盘上相同的文件。如果攻击者在调用 access() 之后，用指向其他文件的象征性链接来取代 file，程序就会使用根权限对文件进行操作，即便这个文件是攻击者在其他情况下无法进行更改的。通过欺骗程序去运行其他情况下不允许执行的操作，从而使攻击者获得更高的权限。
	这种形式的漏洞不限于具有 root 权限的程序。如果应用程序能够执行本不允许攻击者执行的操作，那么有可能就会成为攻击目标。
	导致这种攻击漏洞的原因是从开始测试文件属性到使用文件这段时间的间隙。即使检查后立即使用文件，现今的操作系统也无法确保程序

返回目录

	在让出 CPU 之前能够执行的代码数量。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。
	示例 2：以下代码将生成一个文件，然后更改该文件的所有者。
	fd = creat(FILE, 0644); /* Create file */
	if (fd == -1)
	return;
	if (chown(FILE, UID, -1) < 0) { /* Change file owner */
	...
	}
	该代码假定通过调用 chown() 操作的文件与通过调用 creat() 创建的文件是一样的，但事实未必如此。因为 chown() 根据文件名而不是文件句柄操作，攻击者可以将该文件替换为指向不归他所有的文件的链接。调用 chown() 将给予攻击者链接文件的所有权。

建议	为防止文件访问发生 race condition，必须确保程序对文件开始了一系列操作后，便无法覆盖或修改这一文件。避免使用那些对文件名进行操作的函数，因为它们无法保证在一个单独的函数调用范围之外仍会引用磁盘上的同一文件。先打开文件，然后再使用对文件句柄而不是对文件名进行操作的函数。 检查文件访问权限最有效的方法是降低当前用户的权限，然后尝试使用这个降低的权限去打开文件。如果文件打开成功，则程序会使用得出的文件句柄自动执行其他访问检查。如果文件打开失败，则用户就会无权访问文件，而执行的操作也会中止。在尝试一系列文件操作之前先降低用户的权限，程序就不会因为底层 file system 的改变而轻易受骗。
CWE	APSC-DV-001995 CAT II
OWASP2017	None

返回目录

漏洞名称	**Race Condition:Signal Handling**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数为多个函数安装了相同的信号处理函数，当在短时间内连续收到不同的信号时，会导致 race condition。为多重信号安装相同的信号处理函数的话，当在短时间内连续收到不同的信号时，会导致 race condition。

解释	当安装作为信号处理函数的函数属于非可重入函数时，信号处理就会发生 race condition。非可重入函数会保留一些内部状态，或调用其他同样如此的函数。当安装同一函数去处理多重信号时，很有可能发生这种 race condition。
	信号处理 race condition 很有可能在以下情况中出现：
	1. 程序为多重信号仅安装一个信号处理函数。
	2. 在短时间内，信号处理函数收到两种不同的信号，导致该信号处理函数中出现 race condition。
	示例：以下代码为两个不同的信号仅安装了一个简单的、非折返信号处理函数。如果攻击者使信号在适当的时间内一起发送，信号处理函数就会遭受 double free 漏洞威胁。针对同一个值两次调用 free()，会导致 buffer overflow。当程序使用同一参数两次调用 free() 时，程序中的内存管理数据结构会遭到破坏。这种破坏会导致程序崩溃。有时在某些情况下，还会导致两次调用 malloc() 延迟，而返回相同的指针。如果 malloc() 两次都返回同一个值，稍候程序便会允许攻击者控制整个已经写入双倍分配内存的数据，从而使程序更加容易受到 buffer overflow 的攻击。
	void sh(int dummy) {
	...
	free(global2);
	free(global1);
	...
	}
	int main(int argc,char* argv[]) {
	...
	signal(SIGHUP,sh);
	signal(SIGTERM,sh);
	...
	}

建议	为了防止信号处理发生 race condition，作为信号处理函数安装的函数必须是完全可折返的：这类函数无法保留任何内部状态或调用其他同样如此的函数。编写信号处理函数时，简单化是最好的指导原则。在一个信号处理函数中仅执行最基本的功能，并且对调用的函数要十分小心。可以由信号处理函数安全调用的折返函数列表已经编译完成。POSIX 规范将以下常用函数列为可以从信号处理函数中安全调用： _Exit()

返回目录

	_Exit() _exit() abort() accept() access() aio_error() aio_return() aio_suspend() alarm() bind() cfgetispeed() cfgetospeed() cfsetispeed() cfsetospeed() chdir() chmod() chown() clock_gettime() close() connect() creat() dup() dup2() execle() execve() fchmod() fchown() fcntl() fdatasync() fork() fpathconf() fstat() fsync() ftruncate() getegid() geteuid() getgid() getgroups() getpeername() getpgrp() getpid() getppid() getsockname() getsockopt() getuid() kill() link() listen() lseek() lstat() mkdir() mkfifo() open() pathconf() pause() pipe() poll() posix_trace_event() pselect() raise() read() readlink() recv() recvfrom() recvmsg() rename() rmdir() select() sem_post() send() sendmsg() sendto() setgid() setpgid() setsid() setsockopt() setuid() shutdown() sigaction() sigaddset() sigdelset() sigemptyset() sigfillset() sigismember() signal() sigpause() sigpending() sigprocmask() sigqueue() sigset() sigsuspend() sleep() socket() socketpair() stat() symlink() sysconf() tcdrain() tcflow() tcflush() tcgetattr() tcgetpgrp() tcsendbreak() tcsetattr() tcsetpgrp() time() timer_getoverrun() timer_gettime() timer_settime() times() umask() uname() unlink() utime() wait() waitpid() write() 信号处理函数很容易出错，且没有避免出错的相关良策。当您不确定某个函数是否为折返函数及能否在信号处理函数中安全调用时，请采取保守的方法，即不要调用该函数。同理，不要为多重信号安装相同的信号处理函数，因为这会增加发生 race condition 的可能性。
CWE	CCI-003178
OWASP2017	None

返回目录

漏洞名称	**Redundant Null Check**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数如果间接引用第 N 行上的 null 指针，将导致程序崩溃。该程序可能会间接引用一个 null 指针，从而造成分段故障。

解释	如果不符合程序员的一个或多个假设，则通常会出现 null 指针异常。此问题至少有三种类型：check-after-dereference、dereference-after-check 和 dereference-after-store。如果程序在检查可能为 null 的指针是否为 null 之前间接引用该指针，则会发生 check-after-dereference 错误。如果程序明确检查过 null，并确定该指针为 null，但仍继续间接引用该指针，则会出现 dereference-after-check 错误。此类错误通常是由于错别字或程序员疏忽造成的。如果程序明确将指针设置为 null，但稍后却间接引用该指针，则将出现 dereference-after-store 错误。此错误通常是因为程序员在声明变量时将该变量初始化为 null 所致。
	在这种情况下，在 N 行间接引用该变量时，该变量有可能为 null，从而引起分段故障。
	大多数 null 指针问题会导致一般软件可靠性问题，但如果攻击者可能有意触发 null 指针间接引用，他们可以使用生成的异常绕过安全逻辑以发动拒绝服务攻击，或使应用程序显示调试信息，这些信息在规划后续攻击时十分有用。
	示例 1：在下列代码中，程序员假设变量 ptr 不是 NULL。当程序员间接引用该指针时，这个假设就会清晰的体现出来。当程序员检查 ptr 是否为 NULL 时，就会与该假设发生矛盾。当在 if 语句中检查时，如果 ptr 可以为 NULL，则在其间接引用时也将为 NULL，并引起 segmentation fault。
	ptr->field = val;
	...
	if (ptr != NULL) {
	...
	}
	示例 2：在下列代码中，程序员会确认变量 ptr 为 NULL，然后错误地对其进行间接引用。如果在 if 语句中检查 ptr 时其为 NULL，则会发生 null dereference，从而导致分段故障。
	if (ptr == null) {
	ptr->field = val;
	...
	}
	示例 3：在下列代码中，程序员忘记了字符串 '\0' 实际上为 0 还是 NULL，从而间接引用 null 指针并引发分段故障。
	if (ptr == '\0') {

返回目录

	*ptr = val;
	...
	}
	示例 4：在下列代码中，程序员会将变量 ptr 明确设置为 NULL。之后，程序员会间接引用 ptr，而未检查对象是否为 null 值。
	*ptr = NULL;
	...
	ptr->field = val;
	...
	}

建议	由于间接引用 NULL 指针引发的安全问题大多数情况下会表现为拒绝服务攻击。如果攻击者可以持续触发 null 指针间接引用，那么其他用户可能就无法正常地访问应用程序。除了攻击者故意触发分段故障的情况之外，间接引用 null 指针可能会导致程序时不时地崩溃，且这种问题难以跟踪。 在间接引用可能为 null 的对象之前，请务必仔细检查。如有可能，在处理资源的代码的封装器中纳入 null 检查，以确保在所有情况下均会执行该检查，并最大限度地减少出错的位置。
CWE	CCI-001094
OWASP2017	None

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	**Resource Injection**
默认严重性	3.0
摘要	攻击者可以控制 X(文件) 中第 N 行的 X(函数) 的资源标识符参数，借此访问或修改其他受保护的系统资源。使用用户输入控制资源标识符，借此攻击者可以访问或修改其他受保护的系统资源。

解释	当满足以下两个条件时，就会发生 resource injection：
	1. 攻击者可以指定已使用的标识符来访问系统资源。
	例如，攻击者可能可以指定用来连接到网络资源的端口号。
	2. 攻击者可以通过指定特定资源来获取某种权限，而这种权限在一般情况下是不可能获得的。
	例如，程序可能会允许攻击者把敏感信息传输到第三方服务器。
	在这种情况下，攻击者可以指定通过 X（文件） 中第 N 行的 X（函数） 进入程序的值，这一数值可以通过 Y（文件） 中第 M 行的 Y（函数） 访问系统资源。
	注意：如果资源注入涉及存储在文件系统中的资源，则可以将其报告为名为路径篡改的不同类别。有关这一漏洞的详细信息，请参见 path manipulation 的描述。
	示例：以下代码使用读取自 CGI 请求的端口号来建立一个套接字。
	...
	char* rPort = getenv("rPort");
	...
	serv_addr.sin_port = htons(atoi(rPort));
	if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
	error("ERROR connecting");
	...
	这种受用户输入影响的资源表明其中的内容可能存在危险。例如，包含如句点、斜杠和反斜杠等特殊字符的数据在与 file system 相作用的方法中使用时，具有很大风险。类似的，对于创建远程结点的函数来说，包含 URL 和 URI 的数据也具有很大风险。

建议	阻止 resource injection 的最佳做法是采用一些间接手段。例如创建一份合法资源名的列表，并且规定用户只能选择其中的文件名。通过这种方法，用户就不能直接由自己来指定资源的名称了。 但在某些情况下，这种方法并不可行，因为这样一份合法资源名的列表过于庞大，维护难度过大。因此，在这种情况下，程序员通常会采用执行拒绝列表的办法。在输入之前，拒绝列表会有选择地拒绝或避免潜在的危险字符。但是，任何这样一个列表都不可能是完整的，而且将随着时间的推移而过时。更好的方法是创建一个字符列表，允许其中的字符出现在资源名称中，且只接受完全由这些被认可的字符组成的输入。
CWE	CWE ID 99

返回目录

OWASP2017	A5 Broken Access Control

返回目录

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

返回目录

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

返回目录

漏洞名称	**Second Order SQL Injection**
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	**Setting Manipulation**
默认严重性	3.0
摘要	攻击者可以控制 X(文件) 中第 N 行的 X(函数) 的一个参数，从而导致服务中断或意外的应用程序行为。允许对系统设置进行外部控制可以导致服务中断或意外的应用程序行为。
解释	当攻击者能够通过控制某些值来监控系统的行为、管理特定的资源、或在某个方面影响应用程序的功能时，即表示发生了 Setting Manipulation 漏洞。 这种情况下，潜在的恶意的数据会进入程序并传递给 Y（文件） 的第 M 行中的 Y（函数）。 由于 Setting Manipulation 漏洞影响到许多功能，因此，对它的任何说明都必然是不完整的。与其在 Setting Manipulation 这一类中寻找各个功能之间的紧密关系，不如往后退一步，考虑有哪些系统数值类型不能由攻击者来控制。 例 1：以下 C 代码接受数字作为其命令行参数，并将其设为当前机器的主机 ID。 ... sethostid(argv[1]); ... 虽然进程必须在赋予权限后才能成功调用 sethostid()，但是未被赋予权限的用户也可能会调用这个程序。这个例子中的代码允许用户输入直接控制系统设置的值。如果攻击者为主机 ID 提供一个恶意值，攻击者会错误地识别网络上受影响的机器或者引发其他一些意料之外的行为。 总之，应禁止使用用户提供的数据或通过其他途径获取不可信任的数据，以防止攻击者控制某些敏感的数值。虽然攻击者控制这些数值的影响不会总能立刻显现，但是不要低估了攻击者的攻击力。
建议	禁止由不可信赖的数据来控制敏感数值。在发生此种错误的诸多情况中，应用程序预期通过某种特定的输入，仅得到某一区间内的数值。如果可能的话，应用程序应仅通过输入从预定的安全数值集合中选择数据，而不是依靠输入得到期望的数值，从而确保应用程序行为得当。针对恶意输入，传递给敏感函数的数值应当是该集合中的某些安全选项的默认设置。即使无法事先了解安全数值集合，通常也可以检验输入是否在某个安全的数值区间内。若上述两种验证机制均不可行，则必须重新设计应用程序，以避免应用程序接受由用户提供的潜在危险数值。
CWE	CWE ID 15
OWASP2017	None

返回目录

漏洞名称	**Setting Manipulation**
默认严重性	4
摘要	应用程序使用从 X（文件） 文件第 N 行的用户输入 @Source 元素中收到的值设置 Y（文件） 文件第 M 行中的环境配置设置 YY（元素）。
解释	如果外部用户可以控制应用程序的配置或环境设置，外部用户就可以中断应用程序服务，或导致应用程序出现意外行为，从而可能在应用程序中创建其他漏洞，甚至让应用程序执行恶意操作。 应用程序接收来自用户的未经验证的输入，并直接将该值设置到应用程序配置或环境设置中，而未正确地净化或约束输入。
建议	不要让用户输入或不可信任的数据控制敏感值，特别是配置或环境设置。 在将从用户收到的数据用于任何内部用途之前，一定要进行验证。
CWE	CWE ID 15
OWASP2017	A2-Broken Authentication

返回目录

漏洞名称	**Short Overflow**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中的值未经验证便被用于 Y（文件） 文件第 M 行 YY（元素） 中的算术运算中，这可能导致算术溢流（通常称为“整数溢出”）。
解释	算术溢出会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断或丢失数据；使用溢出的数值执行内存操作时，甚至导致缓冲区溢出，从而导致执行任意代码。 所有数字数据类型都按位表示。如果在算术运算后，某个值超过了其按位表示的位数，那么被加的最重要的数字将被截断，此截断后剩余的值将是回绕操作的剩余值——这称为“算术溢出”，另常误称为——“整数溢出”，此误称是有误导的，因为此名称还适用于许多其他类型。如果可能溢出的数据类型低于其最小值，则它们将负向回绕，通常称为下溢。 例如，如果无符号的 32 位整数设置为 4,294,967,295，然后加 1，它就会溢出并回绕到 0。如果有符号的 32 位整数值为 2,147,483,647，然后加 1，则它会溢出并回绕到 -2,147,483,648。 为了保证代码正确，一定要检查值是否在预期范围内，确保算术运算结果不会溢出或下溢。
建议	对可能包含任意值的数据执行算术运算时，请考虑添加一个检查以保证数据在范围内，且这些操作的结果不会导致溢出或下溢。 考虑为所有算术运算创建封装器，以对特殊情况进行特定处理；例如，如果检查显示已经或将要发生溢出，则抛出异常。
CWE	CWE ID 190
OWASP2017	None

返回目录

漏洞名称	**SQL Injection**
默认严重性	4.0
摘要	X(文件)的 XX（函数） 函数调用通过不可信赖的数据源输入构建的 SQL 查询，这会允许攻击者修改语句的含义或执行任意 SQL 命令。通过不可信赖的数据源输入构建动态 SQL 语句，攻击者就能够修改语句的含义或者执行任意 SQL 命令。

解释	SQL injection 错误在以下情况下发生：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 数据用于动态地构造一个 SQL 查询。
	在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。
	例 1：以下代码动态地构造并执行了一个 SQL 查询，该查询可以搜索与指定名称相匹配的项。该查询仅会显示条目所有者与被授予权限的当前用户一致的条目。
	...
	ctx.getAuthUserName(&userName); {
	CString query = "SELECT * FROM items WHERE owner = '"
	+ userName + "' AND itemname = '"
	+ request.Lookup("item") + "'";
	dbms.ExecuteSQL(query);
	...
	例 2：此外，SQLite 使用以下代码可以获得类似的结果：
	...
	sprintf (sql, "SELECT * FROM items WHERE owner='%s' AND itemname='%s'", username, request.Lookup("item"));
	printf("SQL to execute is: \n\t\t %s\n", sql);
	rc = sqlite3_exec(db,sql, NULL,0, &err);
	...
	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 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';

返回目录

	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 3：此示例说明了将不同的恶意值传递给 Example 1.中构造和执行的查询所带来的影响。如果一个用户名为 wiley 的攻击者为 itemName 输入字符串“name'); DELETE FROM items; --”，则该查询就会变为以下两个查询： SELECT * FROM items WHERE owner = 'wiley' AND itemname = 'name'; DELETE FROM items; --' 众多数据库服务器，其中包括 Microsoft(R) SQL Server 2000，都可以一次性执行多条用分号分隔的 SQL 指令。对于那些不允许运行用分号分隔的批量指令的数据库服务器，比如 Oracle 和其他数据库服务器，攻击者输入的这个字符串只会导致错误；但是在那些支持这种操作的数据库服务器上，攻击者可能会通过执行多条指令而在数据库上执行任意命令。 注意末尾的一对连字符 (--)；这在大多数数据库服务器上都表示该语句剩余部分将视为注释，不会加以执行 [4]。在这种情况下，注释字符对修改的查询中遗留的结尾单引号起作用。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。

返回目录

	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。

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，它们就会创建捆绑参数，这些捆绑参数是一些占位符，用来存放随后插入的数据。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，而不会将数据解析成命令。
	C++ 并没有提供内置的、类似于 Java、C# 或者其他编程语言的参数化 SQL 指令执行机制。如果您正在使用 C++，那么您可以有几个不同选择来使用专门的解决方案或开发您自己的解决方案。
	第一个选择是扩展 CRecordset 和 CDatabase 类来处理参数化的 SQL 指令。Microsoft 提供了一个可以生成这些实例 [5] 的实用程序，并且在创建通用解决方案方面做了一些工作 [6]。虽然在 C++ 中执行参数化的 SQL 指令机制并不在本文档的讨论范围内，但是下面的例子可以说明如何正确地执行通用解决方案：
	ctx.getAuthUserName(&userName);
	CMyRecordset rs(&dbms);
	rs.PrepareSQL(
	"SELECT * FROM items WHERE itemname=? AND owner=?");
	rs.SetParam_String(0,request.Lookup("item"));
	rs.SetParam_String(1,userName);
	rs.SafeExecuteSQL();
	第二个选择是直接使用 ODBC 方法，如 SQLNumParams() [7] 和 SQLBindParameters() [8] 或者是 OLE DB 的 ICommandWithParameters 接口 [9]。使用其中的任意一种方法，都可以得到与使用参数化高级别对象一样的安全保证，但是这些方法只需要低级别的实现方式。
	例 4： 以下 Objective-C 的例子说明了参数化的 sqlite 语句的使用方法：
	...
	sqlite3 *db;
	sqlite3_stmt *dbps;
	...

返回目录

	ctx.getAuthUserName(&userName);
	int dbc = sqlite3_open([dbFile UTF8String], &db);
	...
	dbc = sqlite3_prepare_v2(db, "SELECT * FROM items WHERE itemname=? AND owner=?", -1, SQLITE_TRANSIENT);
	sqlite3_bind_text(dbps, 1, [request lookup:@"item"], -1, SQLITE_TRANSIENT);
	sqlite3_bind_text(dbps, 2, username, -1, SQLITE_TRANSIENT);
	while(sqlite3_step(dbps) == SQLITE_ROW) {
	...
	更加复杂的情况总是出现在生成报表的代码中，用户的输入会改变 SQL 指令的结构，比如在 WHERE 子句中加入动态的约束条件。不要因为这一需求，就无条件地让查询字符串接受连续的用户输入。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**
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 注入攻击。
解释	攻击者可能直接访问系统的所有数据。攻击者使用简单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。
建议	验证所有来源的输入。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。 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	**Stored 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	**Stored Buffer Overflow boundcpy**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Buffer Overflow cpycat**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Buffer Overflow fgets**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Buffer Overflow fscanf**
默认严重性	4
摘要	在向缓冲区写入数据前，未正确地对 Y（文件） 文件第 M 行 YY（元素） 中的 YYY（方法） 方法使用的缓冲区大小进行验证。这可能导致使用 XXX（方法） 传递给 X（文件） 文件第 N 行的 XX（元素） 的源缓冲区覆盖目标缓冲区，导致缓冲区溢出攻击。
解释	各种缓冲区溢出攻击都可能使攻击者控制某些内存区域。通常，这些区域会被用于覆写程序正常运行所必需的堆栈数据，例如代码和内存地址，当然也有其他形式的此类攻击。利用此漏洞一般会导致系统崩溃、无限循环甚至执行任意代码。 缓冲区溢出有很多不同的变种。最基本的形式就是，攻击者控制缓冲区，然后不经大小验证即将其复制到较小的缓冲区。因为攻击者的源缓冲区大于程序的目标缓冲区，所以攻击者的数据会覆写堆栈中的下一个数据，从而使攻击者控制程序结构。 此外，也可能是由于未正确地检查边界导致漏洞；将内存地址暴露在其有效范围之外；允许攻击者控制目标缓冲区的大小；或其他各种原因。
建议	复制缓冲区或字符串之前一定要执行适当的边界检查。 建议使用更安全的函数和结构，例如 char* 的安全字符串类，strcpy 的 strncpy 等。 始终检查缓冲区的大小。 不要返回变量范围之外的变量地址。
CWE	CWE ID 120
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

	如果确实需要使用不可信任的数据调用系统命令或执行外部程序，请不要将数据直接与命令拼接。而要使用支持相关功能的平台函数将参数与命令分离。 不要调用 system() 或其变体，因为这不支持将数据参数与系统命令分离。 而要使用独立接收命令的参数并进行验证的函数之一。这包括 ShellExecute()、execve() 或其变体之一。 一定要将用户控制的数据作为 lpParameters 或 argN 参数（或等效值）传递给函数，并正确地进行引用。切勿将用户控制的数据作为 cmdname 或 filePath 的第一个参数传递。 不要使用用户控制的输入直接执行任何 shell 或命令解释器，例如 bash、cmd 或 make。
CWE	CWE ID 77
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Connection String Injection**
默认严重性	4
摘要	应用程序的 YYY（方法） 方法收到不可信任的、用户控制的数据，并使用这些数据通过 Y（文件） 文件第 M 行的 YY（元素） 元素连接数据库。这可能导致“连接字符串注入”攻击。 攻击者可以将格式错误的数据插入数据库或本地文件中的标准文本字段来注入连接字符串。此文本被 XXX（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 获取，然后被驱动程序用于连接数据库服务器。
解释	如果攻击者可以操作应用程序到数据库服务器的连接字符串，则攻击者可能也能够执行以下任何操作: 破坏应用程序性能（通过增加 MIN POOL SIZE） 篡改网络连接（例如，通过受信任连接） 将应用程序指向攻击者的伪装数据库 （通过暴力攻击）获得数据库中的系统帐户密码。 为了与数据库或其他外部服务器（例如 Active Directory）通信，应用程序会动态构造连接字符串。此连接字符串中包含不可信任的数据，此漏洞可能被恶意用户利用。因为此不可信任的数据未经过限制或适当净化，可能被恶意利用来操纵连接字符串。
建议	无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项: 数据类型 大小 范围 格式 预期值 不允许用户控制数据库连接字符串。避免根据不可信任的数据（特别是用户输入）来动态创建连接字符串。 以合适的配置机制来保存所有的连接字符串。如果 runtime 时需要动态构建连接字符串，请不要直接在连接字符串中添加不可信任的数据；而要让用户从预定义好的连接字符串中选择。
CWE	CWE ID 99
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored DB Parameter Tampering**
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	**Stored Format String Attack**
默认严重性	4
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法从用户输入接收 XX（元素） 值。然后该值被用于构建"格式字符串"YY（元素），并用作 Y（文件） 文件第 M 行 YYY（方法） 方法的字符串格式参数。
解释	在非托管内存环境中，允许攻击者控制格式字符串会使攻击者能够访问攻击者不该访问的内存区域，包括读取其他受限变量、歪曲数据，甚至覆写无访问权限的内存区域。在特定情况下，这甚至可能进一步导致缓冲区溢出和执行任意代码。 应用程序允许用户输入影响用于已格式化打印函数的字符串实参。这个函数族期望第一个实参指定动态构造的输出字符串的相对格式，包括如何表示其他各个实参。 允许外部用户或攻击者控制此字符串，允许他们控制打印函数的功能，从而访问不该访问的内存区域。
建议	通用指南： 不要让用户输入或任何其他外部数据影响格式字符串。 确保使用静态字符串作为格式参数调用所有字符串格式函数，并根据静态格式字符串将正确数量的实参传递给函数。 也可在格式字符串参数中使用所有用户输入来打印格式函数之前，对所有用户输入进行验证，并确保输入中不包含格式化 token。 具体建议： 不要直接在格式化函数的格式字符串参数（通常是第一个或第二个实参）中使用用户输入。 也可在格式字符串中使用从输入导出的已控信息，例如大小或长度——但不能使用输入本身的实际内容。
CWE	CWE ID 134
OWASP2017	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	None

返回目录

漏洞名称	**Stored Log Forging**
默认严重性	3
摘要	X（文件） 文件第 N 行 XXX（方法） 方法从 XX（元素） 元素获取用户输入。然后，此元素的值将传递到代码，而不会进行合适的净化或验证，并最终在 Y（文件） 文件第 M 行的 YYY（方法） 中用于写审计日志。
解释	攻击者可以为安全敏感操作设计审计日志，然后设置错误的审计跟踪，并指向无辜的用户或隐藏事件。 应用程序会在出现安全敏感操作时写审计日志。因为审计日志中包含未经过数据类型验证或随后净化的用户输入，所以输入中可能包含表面上像合法审计日志数据的虚假信息，
建议	无论来源如何，一概验证所有输入。验证应基于白名单：仅接受符合指定结构的数据，而不是排除不符合要求的模式。检查以下项： 数据类型 大小 范围 格式 预期值 验证不能代替编码。完全编码所有来源的所有动态数据，然后再将其嵌入日志。 使用安全的日志记录机制。
CWE	CWE ID 117
OWASP2017	A1-Injection

返回目录

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

返回目录

漏洞名称	**Stored Process Control**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 中动态载入外部库；如果攻击者能影响此库，或者将调用更改到另一个库，攻击者可能能够执行任何代码。
解释	如果攻击者可以通过篡改调用或通过操纵动态加载的库来直接影响动态加载哪个库，则攻击者可能会引入自己的代码，从而接管执行过程，甚至接管正在运行的系统。 从代码动态加载库且满足以下情况之一时会发生流程控制漏洞： 攻击者可以替换正在加载的库，从而在应用程序中引入攻击者自己的代码，可能将其破解。 攻击者可以影响正在加载哪个库，从而使攻击者能够加载不安全的库
建议	加载的库应来自受信任的来源。如果库不是来自可靠的来源，请检查其源代码。 应该对所有本机库进行验证，以确定应用程序是否需要使用该库。 使用最小权限原则运行应用程序，以减少受到成功攻击时的影响。 净化影响库调用的所有输入，以避免恶意内容。 调用库时使用绝对路径，以避免调用不需要的库。 载入使用本地调用的库时，验证所有输入以避免缓冲区溢出。
CWE	CWE ID 114
OWASP2017	A1-Injection

返回目录

漏洞名称	**Stored Resource Injection**
默认严重性	4
摘要	应用程序在 Y（文件） 文件第 M 行的 YYY（方法） 方法使用 X（文件） 文件第 N 行的 XX（元素） 提供的可能受污染的值打开了一个资源。 这可能使攻击者能够控制应用程序的 I/O 资源，或者导致应用程序耗尽其可用资源。 攻击者可将任意数据写入数据库，然后被应用程序使用 X（文件） 文件第 N 行 XXX（方法） 方法中的 XX（元素） 获取。然后该值会被直接用于打开并连接套接字到远程地址和套接字端口。
解释	资源注入产生的影响与实现方式关系很大。如果允许攻击者控制服务器侧资源 I/O，例如网络、存储或内存，攻击者可能会更改这些资源的路由使其暴露、生成多个实例来耗尽资源，或以能够屏蔽其他 I/O 操作的方式创建一个资源。 此外，这个漏洞会被利用来绕过防火墙或其他访问控制机制，或使用应用程序作为扫描内部网络端口或直接访问本地系统的代理。 应用程序根据用户的输入创建了一个能够使攻击者可以控制的资源。
建议	禁止用户或不可信任的数据定义 I/O 参数，例如网络套接字、存储访问权限或其他资源配置。 同样，不要让用户控制的输入或不可信任的数据定义环境变量或文件位置。
CWE	CWE ID 99
OWASP2017	A1-Injection

返回目录

漏洞名称	**String Termination Error**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数在调用第 N 行的 X(函数) 时需要适当的字符串终止，但源缓冲区可能不包含 null 终止符。可能会发生缓冲区溢出。需要适当的字符串终止可能会导致缓冲区溢出。

解释	string termination error 在以下情况下出现：
	1.数据通过某一函数进入程序，而该函数不会以“null”结束输出。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2.数据被传递到某个函数，而该函数的输入需要以“null”结尾。
	在这种情况下，数据将传递到 Y（文件）的第 M 行中的 Y（函数）。
	示例 1：以下代码会从 cfgfile 进行读取，并使用 strcpy() 将输入复制到 inputbuf 中。但是，该代码会错误的假定 inputbuf 始终包含 null 终止符。
	#define MAXLEN 1024
	...
	char *pathbuf[MAXLEN];
	...
	read(cfgfile,inputbuf,MAXLEN); //不以“null”结尾
	strcpy(pathbuf,inputbuf); //需要以“null”结尾的输入
	...
	如果从 cfgfile 中读取的数据在磁盘上按预期以“null”结尾，则 Example 1 中的代码可正常运行。但是，如果攻击者能够篡改此输入，使其不包含所需的 null 字符，则对 strcpy() 的调用将连续从内存进行复制，直到遇到任意 null 字符为止。因此，可能会溢出目标缓冲区，更有甚者，如果攻击者能够控制紧随 inputbuf 之后的内存内容，那么就会使应用程序遭受 Buffer Overflow 攻击。
	示例 2：在以下代码中，readlink() 对存储在缓冲区 path 上的某个符号链接名进行了扩展，以使缓冲区 buf 包含可通过该符号链接而引用的文件的绝对路径。而最终的长度值将通过 strlen() 来计算。
	...
	char buf[MAXPATH];
	...
	readlink(path, buf, MAXPATH);
	int length = strlen(buf);
	...
	由于通过 readlink() 读入 buf 的值不会以“null”结尾，Example 2 中的代码将无法正确运行。在测试过程中，类似于这样的漏洞可能不会被捕捉到，因为 buf 中未使用的内容或紧随其后的内存可能都为 null，因此会显示 strlen()，而这看上去似乎运行正常一样。然而，在默认情况下，strlen() 将持续遍历内存，直到在堆栈中遇到任意 null 字符为

返回目录

	止，这会导致 length 的值远远大于 buf 的大小，从而在随后使用该值的操作中可能会造成缓冲区溢出。 一般来说，字符串表示为一个内存区域，其中的数据以 null 字符结尾。先前的字符串处理方法往往会通过此 null 字符来确定字符串的长度。如果某个不包含 null 终止符的缓冲区被传递给其中一个函数，那么该函数就会读取到该缓冲区末尾之后的内容。 恶意用户通常会通过对应用程序注入意外大小的数据或内容，从而达到利用这类漏洞的目的。为了达到这个目的，他们可能会直接向程序提供恶意输入，或者间接地修改应用程序的资源，如配置文件等。如果攻击者能够使应用程序读取超出缓冲区边界的信息，那么他就可以利用 buffer overflow 漏洞在系统中注入或是执行任意代码。
建议	请尽可能不要调用那些要求字符串以 null 结尾的字符串操作函数，而是通过长度参数（该参数应始终限定在目标缓冲区大小边界内）手动限制字符串长度。只有当不存在带边界的等效函数时，才要求以 null 结尾。在这种情况下，请在使用之前手动终止所有字符串，确保随后使用该字符串时可以按预期正常运行。 有些字符串操作函数是非常危险的，因此应该完全将其禁用。如果您的安全策略禁止使用这些函数，您可以通过编写自定义规则来在应用程序源代码分析过程中无条件标记这些函数来强制实施此策略。有关详细信息，请参阅《Fortify Static Code Analyzer 自定义规则指南》。 此外，您还可以通过重写违反安全策略的函数在头文件中的定义来禁用这些函数。这样做可以将所有使用此类函数的情况都标记为编译器错误。 例如，要禁用 strcat()，可定义以下宏： #define strcat unsafe_strcat
CWE	CWE ID 170, CWE ID 665
OWASP2017	None

返回目录

漏洞名称	**Symmetric Encryption Insecure Cipher Mode**
默认严重性	4
摘要	X（文件） 文件第 N 行在 XX（元素） 的对称密码使用了 Y（文件） 文件第 M 行中 YY（元素） 的不安全密码模式。
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 不安全的密码模式可能会给系统带来很多漏洞。例如，CBC 这样的未经验证的密码模式不能保证解密时密文的完整性。这意味着攻击者可以修改密文使解密失败或者滥用解密终端并用其帮助解密密文。另一个相关案例是 ECB，它会泄漏与明文数据模式相关的信息。 应用程序的加密相关功能有问题。 应用程序使用不安全的密码模式。发现某些操作模式会泄漏与纯文本相关的信息或缺少验证。例如，在 ECB 模式中，执行加密/解密操作时缺乏扩散会导致所产生的密文泄漏与明文数据相关的模式信息。
建议	对称加密用于保护需要安全保存的敏感数据。为避免受到各种攻击，请遵循以下规则： 密钥必须使用安全的源以安全的方式导出 初始化向量（或随机数）必须使用加密安全的伪随机数生成器生成 为避免篡改，请对密文进行身份验证（建议使用经过身份验证的加密密码模式） 使用 GCM 模式下的 AES 或 ChaCha20-Poly1305，因为这些都是推荐的对称密码。
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Symmetric Encryption Insecure Predictable IV**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的加密函数使用了 Y（文件） 文件第 M 行中的可预测初始化向量 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 在对称密码上使用静态或可预测的初始化向量可能会泄漏与纯文本相关的信息。 应用程序的加密相关功能有问题。 初始化向量是某些密码使用的固定大小的随机数（使用一次的数字），它们需要根据每次操作的情况使用加密安全的随机源生成。如果该数字在密钥相同的多个加密操作中保持静态，则应用程序可能会泄漏与数据相关的信息，这是因为如果两个加密操作的明文相同且 IV 和密钥也相同，则产生的密文也是相同的。
建议	对称加密用于保护需要安全保存的敏感数据。为避免受到各种攻击，请遵循以下规则： 密钥必须使用安全的源以安全的方式导出 初始化向量（或随机数）必须使用加密安全的伪随机数生成器生成 为避免篡改，请对密文进行身份验证（建议使用经过身份验证的加密密码模式）
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Symmetric Encryption Insecure Predictable Key**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的加密函数使用了 Y（文件） 文件第 M 行中的可预测密钥 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 应用程序的加密相关功能有问题。 应用程序使用不安全的密钥加密敏感数据。应用程序中硬编码的静态密钥会被通过二进制逆向工程获得。如果静态加密密钥被第三方泄露，攻击者就可以解密使用此密钥加密的所有过去的消息。对于这种情况，供应商通常很难在部署后进行补救，因为硬编码的密钥意味着开发人员没有在应用程序上构建密钥管理层，也就无法轮换受损/过期的密钥。容易预测或暴力破解的密钥也可能导致被攻击者泄漏敏感数据。
建议	对称加密用于保护需要安全保存的敏感数据。为避免受到各种攻击，请遵循以下规则: 密钥必须使用安全的源以安全的方式导出 初始化向量（或随机数）必须使用加密安全的伪随机数生成器生成 为避免篡改，请对密文进行身份验证（建议使用经过身份验证的加密密码模式）
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Symmetric Encryption Insecure Static IV**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的加密函数使用了 Y（文件） 文件第 M 行中的静态初始化向量 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 在对称密码上使用静态或可预测的初始化向量可能会泄漏与纯文本相关的信息。 应用程序的加密相关功能有问题。 初始化向量是某些密码使用的固定大小的随机数（使用一次的数字），它们需要根据每次操作的情况使用加密安全的随机源生成。如果该数字在密钥相同的多个加密操作中保持静态，则应用程序可能会泄漏与数据相关的信息，这是因为如果两个加密操作的明文相同且 IV 和密钥也相同，则产生的密文也是相同的。
建议	对称加密用于保护需要安全保存的敏感数据。为避免受到各种攻击，请遵循以下规则： 密钥必须使用安全的源以安全的方式导出 初始化向量（或随机数）必须使用加密安全的伪随机数生成器生成 为避免篡改，请对密文进行身份验证（建议使用经过身份验证的加密密码模式）
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Symmetric Encryption Insecure Static Key**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的加密函数使用了 Y（文件） 文件第 M 行中的静态密钥 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 如果应用程序使用静态或可预测的密钥进行加密，则攻击者可以利用此信息来解密捕获到的密文。 应用程序的加密相关功能有问题。 应用程序使用不安全的密钥加密敏感数据。应用程序中硬编码的静态密钥会被通过二进制逆向工程获得。如果静态加密密钥被第三方泄露，攻击者就可以解密使用此密钥加密的所有过去的消息。对于这种情况，供应商通常很难在部署后进行补救，因为硬编码的密钥意味着开发人员没有在应用程序上构建密钥管理层，也就无法轮换受损/过期的密钥。容易预测或暴力破解的密钥也可能导致被攻击者泄漏敏感数据。
建议	对称加密用于保护需要安全保存的敏感数据。为避免受到各种攻击，请遵循以下规则: 密钥必须使用安全的源以安全的方式导出 初始化向量（或随机数）必须使用加密安全的伪随机数生成器生成 为避免篡改，请对密文进行身份验证（建议使用经过身份验证的加密密码模式）
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**System Information Leak**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数通过调用第 N 行的 X(函数) 来揭示系统数据或调试信息。由 X(函数) 揭示的信息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，X（文件）的第 N 行的 X（函数） 中的数据会通过 Y（文件） 的第 M 行的 Y（函数） 流出程序。 示例 1：以下代码会将路径环境变量输出到标准错误流： char* path = getenv("PATH"); ... fprintf(stderr, "cannot find exe on path %s\n", path); 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，该错误消息可以准确地告诉攻击者系统容易遭受哪种类型的攻击。例如，一则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。在 Example 1 中，搜索路径可能会暗示操作系统的类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施。
建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。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	3.0
摘要	X(文件) 中的 XX（函数） 函数通过调用第 N 行的 X(函数) 来揭示系统数据或调试信息。由 X(函数) 揭示的信息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，X（文件）的第 N 行的 X（函数） 中的数据会通过 Y（文件） 的第 M 行的 Y（函数） 流出程序。
	示例 1：以下代码会通过套接字泄露系统信息：
	int sockfd;
	int flags;
	char hostname[1024];
	hostname[1023] = '\0';
	gethostname(hostname, 1023);
	...
	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	flags = 0;
	send(sockfd, hostname, strlen(hostname), flags);
	该信息可能会显示给远程用户。在某些情况下，该错误消息可以准确地告诉攻击者系统容易遭受哪种类型的攻击。例如，一则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。在 Example 1 中，搜索路径可能会暗示操作系统的类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施。

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

返回目录

漏洞名称	**System Information Leak:Internal**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数通过调用第 N 行的 X(函数) 来揭示系统数据或调试信息。由 X(函数) 揭示的信息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	通过日志或打印功能将系统数据或调试信息发送到本地文件、控制台或屏幕时，就会发生内部信息泄露。 在这种情况下，X（文件）的第 N 行的 X（函数） 中的数据会通过 Y（文件） 的第 M 行的 Y（函数） 流出程序。 示例 1：以下代码会将路径环境变量输出到标准错误流： char* path = getenv("PATH"); ... fprintf(stderr, "cannot find exe on path %s\n", path); 根据这一系统配置，该信息可能会转储到控制台、写入日志文件或公开给用户。在某些情况下，该错误消息可以准确地告诉攻击者系统容易遭受哪种类型的攻击。例如，一则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。在 Example 1 中，搜索路径可能会暗示操作系统的类型、系统上安装了哪些应用程序以及管理员在配置程序时采取了哪些保护措施。
建议	编写错误消息时，始终要牢记安全性。在编码的过程中，尽量避免使用繁复的消息，提倡使用简短的错误消息。限制生成与存储繁复的输出数据可帮助管理员和程序员诊断问题。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	**TOCTOU**
默认严重性	3
摘要	Y（文件） 文件中的 YYY（方法） 方法使用了 YY（元素），会被其他并发功能以对线程不安全的方式访问，从而导致此资源产生竞争条件。
解释	最好的情况是，竞争条件可能导致准确性问题、值被覆盖或可能导致拒绝服务攻击的意外行为。在最坏的情况下，它可能会使攻击者通过重用可控制的竞争条件获得有利条件，从而检索数据或绕过安全机制。 条件竞争就是一个公共资源实例被多个并发逻辑进程使用。如果这些逻辑进程都尝试检索和更新资源，但没有及时的管理系统（例如锁定），就会发生条件竞争。 例如如果一个资源可以返回特定的值给进程以进行进一步的编辑，但值又被第二个进程更新，导致原始进程的数据不再有效，这就是一种条件竞争。原始进程编辑不正确的值并将其更新回资源后，第二个进程的更新就会被覆盖并丢失。
建议	在应用程序的各个并发进程之间共享资源时，一定要保证这些资源的线程安全，或者实现锁定机制以保证预期的并发活动。
CWE	CWE ID 367
OWASP2017	None

返回目录

漏洞名称	**Type Conversion Error**
默认严重性	4
摘要	Y（文件） 文件第 M 行将较大的数据类型变量 YY（元素） 分配给了较小的数据类型。这将导致数据丢失，而且通常是数值的重要位或符号位。
解释	未经检查和显式转换便将大数据类型分配给较小的数据类型会导致未定义的行为和意外的影响，例如数据损坏（例如值回绕，即最大值变为最小值）；系统崩溃；无限循环；逻辑错误，例如绕过安全机制；截断数据；甚至会缓冲区溢出，从而导致执行任意代码。 隐式将较大的数值数据类型转换为较小数据类型的变量时，就可能产生该缺陷。这会迫使程序丢弃数字中的一些信息。 根据数值数据类型在内存中的存储方式，这通常会是值最高的位，从而导致存储型数字受到严重破坏。也可能丢失有符号整数的符号位，完全颠倒数字正负。
建议	避免将较大的数据类型转换为较小的类型。 建议将目标变量升级为足够大的数据类型。 如果需要向下转换，一定要在转换前检查值是否有效且在目标类型的范围内
CWE	CWE ID 681
OWASP2017	None

返回目录

漏洞名称	**Type Mismatch:Integer to Character**
默认严重性	3.0
摘要	函数 X(函数) 将返回转换为 int 的 unsigned char，但返回值将赋给 char 类型。函数将返回转换为 int 的 unsigned char，但返回值将赋给 char 类型。

解释	当转换为整数的不带符号的字符赋给带符号的字符时，可能无法从 EOF 区别其值。
	示例 1：以下代码会读取一个字符，并将其与 EOF 进行比较。
	char c;
	while ((c = getchar()) != '\n' && c != EOF) {
	…
	}
	在这种情况下，来自 getchar() 的返回值将转换为 char 并与 EOF（一个 int）进行比较。假设 c 是一个带符号的 8 位值，EOF 是一个带符号的 32 位值，那么如果 getchar() 返回由 0xFF 表示的字符，则与 EOF 相比，c 的值将是 0xFFFFFFFF 的符号扩展。由于 EOF 通常定义为 -1 (0xFFFFFFFF)，因此该循环将错误地终止。

建议	不要将返回值从字符输入和输出函数转换为 char。相反，应将它们存储在类型为 int 的变量中。要避免发生这种类型转换错误，可考虑使用 feof() 和 ferror() 函数，而不是直接与 EOF 进行比较。T
	示例 2：以下代码会使用此方法更正 Example 1 中显示的错误。
	char c;
	while ((c = getchar()) != '\n' && !feof()
	&& !ferror()) {
	…
	}

CWE	CWE ID 192
OWASP2017	None

返回目录

漏洞名称	**Type Mismatch:Negative to Unsigned**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数声明为返回一个不带符号的值，但在第 N 行上，它返回一个负值。该函数声明为返回一个不带符号的值，但在有些情况下却返回一个负值。

解释	依赖在带符号和不带符号的数字之间进行隐式转换是很危险的，因为转换的结果可能是一个超出预料的值，并且会违反程序员在程序中所作的其他假设。
	示例 1：在此示例中，变量 amount 在返回时可能包含负值。由于函数已声明为返回不带符号的整数，因此 amount 将隐式转换为无符号的值。
	unsigned int readdata () {
	int amount = 0;
	…
	if (result == ERROR)
	amount = -1;
	…
	return amount;
	}
	如果满足 Example 1 中的错误条件，则 readdata() 的返回值在使用 32 位整型的系统上将为 4,294,967,295。
	在带符号和不带符号的值之间进行转换会引发各种错误，但从安全性角度来说，最常见的是引发 integer overflow 和 buffer overflow 漏洞。

建议	尽管在带符号和不带符号的数量之间出现意外转换只会引起一般的质量问题，但根据转换违背的具体假设不同，可能会导致严重的安全性风险。请注意编译器关于 signed/unsigned 转换的警告。有些程序员可能认为这些警告无关紧要，但是在某些情况下，它们可以指出潜在的 integer overflow 问题。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Type Mismatch:Signed to Unsigned**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数声明为返回一个不带符号的值，但在第 N 行上，它返回一个带符号的值。该函数声明为返回一个不带符号的值，但却返回一个带符号的值。

解释	依赖在带符号和不带符号的数字之间进行隐式转换是很危险的，因为转换的结果可能是一个超出预料的值，并且会违反程序员在程序中所作的其他假设。
	示例 2：在此示例中，根据 accecssmainframe() 的返回值，变量 amount 在返回时可能包含负值。由于函数已声明为返回不带符号的值，因此 amount 将隐式强制转换为无符号的数字。
	unsigned int readdata () {
	int amount = 0;
	…
	amount = accessmainframe();
	…
	return amount;
	}
	如果 accessmainframe() 的返回值为 -1，则 readdata() 的返回值在一个 32 位整型系统上将为 4,294,967,295。
	在带符号和不带符号的值之间进行转换会引发各种错误，但从安全性角度来说，最常见的是引发 integer overflow 和 buffer overflow 漏洞。

建议	尽管在带符号和不带符号的数量之间出现意外转换只会引起一般的质量问题，但根据转换违背的具体假设不同，可能会导致严重的安全性风险。请注意编译器关于 signed/unsigned 转换的警告。有些程序员可能认为这些警告无关紧要，但是在某些情况下，它们可以指出潜在的 integer overflow 问题。
CWE	APSC-DV-002590 CAT I
OWASP2017	None

返回目录

漏洞名称	**Unchecked Return Value**
默认严重性	2.0
摘要	X(文件) 中的 XX（函数） 函数忽略了第 N 行的 X(函数) 返回的值，这可能会导致程序无法发现意外状况和情况。忽略方法的返回值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的两个可疑的假设是：一是这个函数调用不可能出错；二是即使出错了，也不会对系统造成什么重要影响。当程序员忽略函数返回值时，就暗示着自己是基于上述任一假设来执行操作。在这种情况下，返回代码不会在 X(文件) 中第 N 行进行检查。
	示例：请考虑以下代码：
	char buf[10], cp_buf[10];
	fgets(buf, 10, stdin);
	strcpy(cp_buf, buf);
	程序员希望在 fgets() 返回时，buf 包含一个以“null”结尾的字符串，并且该字符串的长度小于或等于 9。但是，如果发生了 I/O 错误，fgets() 将不会返回以“null”结尾的 buf。此外，如果在读取任何字符之前，已经到达文件的结尾处，那么 fgets() 在返回时就不会在 buf 中写入任何内容。在这两种情况下，fgets() 会通过返回 NULL 来表示已发生异常情况，但在该代码中，不会发出警告。如果 buf 中缺少一个 null 终止符，则可能会导致在随后调用 strcpy() 时出现缓冲区溢出。

建议	如果函数返回错误代码或其他任何有关运行成功或失败的证据，请务必检查错误状况，即便没有任何明显迹象表明会发生这种错误。除了防止安全错误，许多乍看上去难以理解的 bug 最后都会归结为是由于忽略返回值而产生的。 在您的应用程序中创建一种便于使用的、标准的处理故障方法。如果错误处理过程简单直接，往往不容易被程序员忽略。一种标准化的错误处理方法是，将围绕检查和处理错误状况的常用函数写入封装器，而无需程序员的其他干预。实施并采用封装器后，就可以禁止使用未封装的函数，并利用自定义规则加以执行。
CWE	CCI-001314, CCI-003272
OWASP2017	None

返回目录

漏洞名称	**Unchecked Return Value**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法调用 XX（元素） 函数。但是，代码未检查此函数的返回值，因此无法检测 runtime 错误或其他意外状态。
解释	不检查函数返回值的程序可能会导致应用程序进入未定义的状态。这可能导致意外行为和意外后果，包括数据不一致、系统崩溃或其他根据错误发起的攻击。 应用程序调用了一个函数，但未检查函数的返回值的结果。应用程序仅忽略了此结果值，首选认为此结果值是正确的和需要的并使用此结果值或传递此结果值。
建议	一定要检查所有返回值的被调用函数的结果，并验证结果是否为预期值。 确保调用函数响应所有可能的返回值。 预测 runtime 错误并正确地处理它们。显式定义处理意外错误的机制。
CWE	CWE ID 252
OWASP2017	None

返回目录

漏洞名称	**Undefined Behavior**
默认严重性	2.0
摘要	除非 X(函数) 的控制参数设置为一个特定值，否则无法定义它的行为。除非该函数的控制参数设置为一个特定值，否则无法定义的行为。
解释	The Linux Standard Base Specification 2.0.1 for libc 对一些内部函数的参数进行了限制 [1]。如果函数的参数不满足这些约束条件，将无法定义函数的行为。 在这种情况下，X（函数）的行为未在 X(文件) 中的第 N 行进行定义。 通常不应直接调用 X（函数）。大部分情况下，都会通过一个定义于系统头文件中的宏来进行调用，且该宏必须确保满足以下约束条件： 在下列 file system 函数中，值 1 必须传递给第一个参数（版本号）： __xmknod 在下列宽字符字符串函数中，数值 2 必须传递给第三个参数（组参数）： __wcstod_internal __wcstof_internal _wcstol_internal __wcstold_internal __wcstoul_internal 在下列 file system 函数中，数值 3 必须作为第一参数（版本号）进行传递： __xstat __lxstat __fxstat __xstat64 __lxstat64 __fxstat64
建议	请利用在标准系统头文件中定义的相应的宏来使用函数，而不是对函数进行直接调用。
CWE	CWE ID 475
OWASP2017	None

返回目录

漏洞名称	**Undefined Behavior:Redundant Delete**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数将显式删除托管指针。当托管指针类稍后间接引用此指针时，程序有可能会崩溃或无法正常运行。显式删除托管指针将导致程序崩溃或无法正常运行。

解释	删除托管指针后，如果指针管理代码假设该指针仍然有效，将引发程序崩溃或执行错误的操作。以下示例可说明这一错误。
	std::auto_ptr<foo> p(new foo);
	foo* rawFoo = p.get();
	delete rawFoo;
	仅当托管指针类支持“分离”操作，允许程序员控制特定指针的内存管理时，此规则不适用。如果程序在调用 delete 之前将指针从管理类中分离出来，则管理类知道以后不再使用该指针。

建议	引发这种错误最常见的原因是：程序员并没有意识到指针受托管指针类的控制，当不再引用指针时将自动释放指针。如果程序确实要脱离指针管理类的控制并释放内存，并且如果托管指针类支持“分离”操作，那么可在调用 delete 之前通过分离指针来解决此问题。
	std::auto_ptr<foo> p(new foo);
	foo* rawFoo = p.get();
	p.release();
	delete rawFoo;

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Uninitialized Variable**
默认严重性	2.0
摘要	X(文件) 中的函数 XX（函数） 在初始化之前使用变量 X（变量）。 程序可能会在初始化之前使用一个变量。

解释	默认情况下，C 语言和 C++ 语言中的堆栈变量是未经初始化的。它们的初始值取决于调用函数时，它们所在的栈发生了什么情况。程序应该永不使用 uninitialized variable。
	在这里，在 XX（文件） 中第 M 行声明的变量可在未经事先设置的情况下用于第 N 行。
	对于程序员来说，他们通常会使用代码中 uninitialized variable 来处理错误或是一些特殊和异常的情况。uninitialized variable 警告有时能够指出代码中存在的排字错误。
	例 1：以下切换语句企图为变量 aN 和 bN 赋值，但在默认情况下，程序员会一不小心为 aN 赋两次值。
	switch (ctl) {
	case -1:
	aN = 0; bN = 0;
	break;
	case 0:
	aN = i; bN = -i;
	break;
	case 1:
	aN = i + NEXT_SZ; bN = i - NEXT_SZ;
	break;
	default:
	aN = -1; aN = -1;
	break;
	}
	多数 uninitialized variable 问题都是由软件的可靠性造成的，但如果攻击者能够故意触发某个未经初始化的变量的使用，那么它就能通过引发程序崩溃来发动一个 denial of service 攻击。在适当的情况下，攻击者通过调用函数之前影响堆栈中的数值，就可以控制某个 uninitialized variable 的值。

建议	使用变量前，请先对其初始化。
CWE	APSC-DV-002400 CAT II
OWASP2017	None

返回目录

漏洞名称	**Unreleased Resource**
默认严重性	3.0
摘要	文件 X(文件) 中的函数 XX（函数） 有时无法成功释放由第 M 行的 XXX（函数） 函数分配的系统资源。程序可能无法成功释放某一项系统资源。

解释	程序可能无法成功释放某一项系统资源。
	这种情况下，尽管程序并不总会释放 XX（文件） 文件第 M 行所分配的资源，但执行这一操作程序的路径依然存在。
	资源泄露至少有两种常见的原因：
	- 错误状况及其他异常情况。
	- 未明确程序的哪一部份负责释放资源。
	大部分 Unreleased Resource 问题只会导致常规软件可靠性问题，但如果攻击者能够故意触发资源泄漏，该攻击者就有可能通过耗尽资源池的方式发起 Denial of Service 攻击。
	示例： 如果产生错误，以下函数不会关闭它所打开的文件句柄。如果进程长时间存在，它会耗尽文件句柄。
	int decodeFile(char* fName)
	{
	char buf[BUF_SZ];
	FILE* f = fopen(fName, "r");
	if (!f) {
	printf("cannot open %s\n", fName);
	return DECODE_FAIL;
	} else {
	while (fgets(buf, BUF_SZ, f)) {
	if (!checkChecksum(buf)) {
	return DECODE_FAIL;
	} else {
	decodeBlock(buf);
	}
	}
	}
	fclose(f);
	return DECODE_SUCCESS;
	}

建议	因为资源泄漏很难追踪，所以要建立一系列资源管理模式和软件指令，并且不违反您所定制的这些规则。 在本例中，一个应对错误处理失当的良好解决方法便是使用向前式 goto 语句，使函数有一个可以处理错误的明确定义的区域，如下所示： int decodeFile(char* fName)

返回目录

	{
	char buf[BUF_SZ];
	FILE* f = fopen(fName, "r");
	if (!f) {
	goto ERR;
	} else {
	while (fgets(buf, BUF_SZ, f)) {
	if (!checkChecksum(buf)) {
	goto ERR;
	} else {
	decodeBlock(buf);
	}
	}
	}
	fclose(f);
	return DECODE_SUCCESS;
	ERR:
	if (!f) {
	printf("cannot open %s\n", fName);
	} else {
	fclose(f);
	}
	return DECODE_FAIL;
	}

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Unreleased Resource:Database**
默认严重性	3.0
摘要	X(文件)中的函数 XX（函数） 有时无法释放由第 M 行的 XXX（函数） 分配的数据库资源。程序可能无法释放某个数据库资源。

解释	资源泄露至少有两种常见的原因：
	- 错误状况及其他异常情况。
	- 未明确程序的哪一部份负责释放资源。
	大部分 Unreleased Resource 问题只会导致常规软件可靠性问题，但如果攻击者能够故意触发资源泄漏，该攻击者就有可能通过耗尽资源池的方式发起 Denial of Service 攻击。
	示例 1：以下代码执行数据库查询，但是不释放语句或连接资源。
	- void insertUser:(NSString *)name {
	...
	sqlite3_stmt *insertStatement = nil;
	NSString *insertSQL = [NSString stringWithFormat:@INSERT INTO users (name, age) VALUES (?, ?)];
	const char *insert_stmt = [insertSQL UTF8String];
	...
	if ((result = sqlite3_prepare_v2(database, insert_stmt,-1, &insertStatement, NULL)) != SQLITE_OK) {
	MyLog(@"%s: sqlite3_prepare error: %s (%d)", __FUNCTION__, sqlite3_errmsg(database), result);
	return;
	}
	if ((result = sqlite3_step(insertStatement)) != SQLITE_DONE) {
	MyLog(@"%s: step error: %s (%d)", __FUNCTION__, sqlite3_errmsg(database), result);
	return;
	}
	...
	}

建议	始终释放资源以防止出现 Memory Leak。
	
	示例 2：以下代码执行数据库查询，并且释放语句和连接资源。
	- void insertUser:(NSString *)name {
	...
	sqlite3_stmt *insertStatement = nil;
	NSString *insertSQL = [NSString stringWithFormat:@INSERT INTO users (name, age) VALUES (?, ?)];
	const char *insert_stmt = [insertSQL UTF8String];
	...

返回目录

	if ((result = sqlite3_prepare_v2(database, insert_stmt,-1, &insertStatement, NULL)) != SQLITE_OK) {
	MyLog(@"%s: sqlite3_prepare error: %s (%d)", __FUNCTION__, sqlite3_errmsg(database), result);
	sqlite3_finalize(insert_stmt);
	sqlite3_close(database);
	return;
	}
	if ((result = sqlite3_step(insertStatement)) != SQLITE_DONE) {
	MyLog(@"%s: step error: %s (%d)", __FUNCTION__, sqlite3_errmsg(database), result);
	sqlite3_finalize(insert_stmt);
	sqlite3_close(database);
	return;
	}
	...
	sqlite3_finalize(insert_stmt);
	sqlite3_close(database);
	}

CWE	CWE ID 619, CWE ID 772
OWASP2017	None

返回目录

漏洞名称	**Unreleased Resource:Synchronization**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数无法释放它在第 M 行获取的锁，这将导致死锁。程序无法释放其持有的锁，这可能会导致死锁。

解释	程序可能无法成功释放某一项系统资源。
	这种情况下，尽管程序并不总会释放 XX（文件） 文件第 M 行所分配的资源，但执行这一操作程序的路径依然存在。
	资源泄露至少有两种常见的原因：
	- 错误状况及其他异常情况。
	- 未明确程序的哪一部份负责释放资源。
	大部分 Unreleased Resource 问题只会导致常规软件可靠性问题，但如果攻击者能够故意触发资源泄漏，该攻击者就有可能通过耗尽资源池的方式发起 Denial of Service 攻击。
	示例：如果发生错误，以下函数会破坏它分配的条件变量。如果进程长时间存在，它会耗尽文件句柄。
	int helper(char* fName)
	{
	int status;
	...
	pthread_cond_init (&count_threshold_cv, NULL);
	pthread_mutex_init(&count_mutex, NULL);
	status = perform_operation();
	if (status) {
	printf("%s", "cannot perform operation");
	return OPERATION_FAIL;
	}
	pthread_mutex_destroy(&count_mutex);
	pthread_cond_destroy(&count_threshold_cv);
	return OPERATION_SUCCESS;
	}

建议	因为资源泄漏很难追踪，所以要建立一系列资源管理模式和软件指令，并且不违反您所定制的这些规则。
	在本例中，一个应对错误处理失当的良好解决方法便是使用向前式 goto 语句，使函数有一个可以处理错误的明确定义的区域，如下所示：
	int helper(char* fName)
	{
	int status;
	...
	pthread_cond_init (&count_threshold_cv, NULL);
	pthread_mutex_init(&count_mutex, NULL);

返回目录

	status = perform_operation();
	if (status) {
	goto ERR;
	}
	pthread_mutex_destroy(&count_mutex);
	pthread_cond_destroy(&count_threshold_cv);
	return OPERATION_SUCCESS;
	ERR:
	printf("%s", "cannot perform operation");
	pthread_mutex_destroy(&count_mutex);
	pthread_cond_destroy(&count_threshold_cv);
	return OPERATION_FAIL;
	}

CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Unsafe Reflection**
默认严重性	3.0
摘要	攻击者可以控制 X(文件) 中第 N 行的反射方法 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。
	如果攻击者能够将文件上传到应用程序的路径或者库路径上出现的位置，那么对应用程序来说，情况会非常糟糕。无论处于上面哪种情况，攻击者都能通过反射将新的行为引入应用程序，而这一行为往往可能是恶意的。
	在这种情况下，不可信赖的数据通过 X（文件）的第 N 行进入 X（函数） 的程序。传递到 Y（文件） 的第 M 行中的 Y（函数） 的反射 API。
	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。下面的例子显示了一个 JNI 命令发送器，它使用反射来执行 Java 方法，该方法由读取自 CGI 请求中的数值进行验证。执行该代码允许攻击者调用在 clazz 中定义的所有函数。
	char* ctl = getenv("ctl");
	...
	jmethodID mid = GetMethodID(clazz, ctl, sig);
	status = CallIntMethod(env, clazz, mid, JAVA_ARGS);
	...

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。
CWE	CWE ID 470, CWE ID 494
OWASP2017	A5 Broken Access Control

返回目录

漏洞名称	**Use After Free**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数会引用第 N 行中一个已释放的内存地址。在释放内存后对其进行引用会导致程序崩溃。

解释	当程序持续使用某个已释放的指针时，就会发生 use after free 错误。就像 double free 和 memory leak 错误一样，use after free 错误有以下两种情况，有时这两种情况会同时发生：
	– 错误状况及其他异常情况。
	— 不清楚由程序的哪一部分负责释放内存
	在这种情况下，该数据释放在 XX（文件）中第 M 行，并在 X(文件) 中的第 N 行再次使用。
	Use after free 错误有时对程序来说是没有任何影响的，但有时会造成程序的崩溃。目前技术上可以对释放的内存空间进行重新分配，而攻击者可以利用这一点，发动 buffer overflow 攻击，同时我们对此类攻击并不会有任何察觉。
	示例：以下代码举例说明了一个 use after free 错误：
	char* ptr = (char*)malloc (SIZE);
	…
	if (err) {
	abrt = 1;
	free(ptr);
	}
	…
	if (abrt) {
	logError("operation aborted before commit", ptr);
	}

建议	请不要在内存空间释放后使用内存，即一旦调用 free()，立即放弃对该内存的所有引用。这是很容易实现的，我们只要使用一个宏来替换对 free() 的调用，并在宏中定义当指针被释放后立即分配 NULL： #define FREE(ptr) {free(ptr); ptr = NULL;} 如果您不清楚应该何时释放内存空间，可以使用这项技术，虽然该技术能够防止已释放的内存被再次使用，但为指针分配 NULL 会导致间接引用 null 指针。对于多数情况而言，这种技术可能会改进系统，原因是在测试期间就可能会捕捉到该错误，且不易造成可供利用的漏洞。其实，该方法是将一个不可预测的错误转换成了一个易于调试的错误。
CWE	CCI-001094
OWASP2017	None

返回目录

漏洞名称	**Use After Free**
默认严重性	4
摘要	Y（文件） 文件第 M 行的指针 YY（元素） 在释放后又被使用。
解释	内存释放后使用错误会导致代码使用之前已分配特定值的内存区域，但该区域已被释放并且可能已被另一个值覆写。此错误可能导致意外行为、内存损坏和崩溃错误。在某些情况下，释放和使用的内存部分会被用于确定执行流程，而攻击者可以引发错误，导致执行恶意代码。 指向变量的指针使代码可以有设定大小的到动态分配变量的地址。最终，指针的目标会被释放——这可能是在代码中显式完成的，例如以编程的方式释放此变量时；或者隐式完成，例如在返回局部变量时——返回后，变量的范围也被释放。释放后，内存将被应用程序重用，并被新数据覆写。此时，取消引用此指针可能会解析新写入的和意外的数据。
建议	不要返回局部变量或指针 检查代码以确保显式释放指针后没有流程可以使用指针
CWE	CWE ID 416
OWASP2017	A1-Injection

返回目录

漏洞名称	**Use of a One Way Hash without a Salt**
默认严重性	4
摘要	应用程序使用 Y（文件） 文件第 M 行 YYY（方法） 中的 YY（元素） 保护密码，使用加密 hash XX（元素）。但是，代码不使用不可预测的随机值来 salt hash 值，使攻击者能够反转 hash 值。
解释	如果攻击者能够获得 hash 密码，攻击者就能使用这个弱点反转 hash，并获得原始密码。获得密码后，攻击者就可以冒充用户，充分利用用户的权限并访问用户的个人数据。此外，这还很难被发现，因为攻击者使用的都是受害者的凭证。 典型的加密 hash 值都非常快，例如 SHA-1 和 MD5。结合预先计算的彩虹表等攻击技术，攻击者可以比较轻松地反转 hash 值并发现原始密码。如果不向密码添加独特的随机 salt，会使暴力破解变得更加简单。
建议	通用指南： - 一定要使用强力的现代算法进行加密、hash 计算等。 - 不要使用弱的、过时的或淘汰的算法。 - 一定要根据具体要求选择正确的加密机制。 具体建议： - 应使用密码 hash 算法保护密码，而不要使用通用加密 hash。这包括 bcrypt、scrypt、PBKDF2 和 Argon2 等自适应 hash。 - 根据指定的环境和风险情况调整自适应 hash 函数的工作因子或成本。 - 不要使用常规的加密 hash 值保护密码，例如 SHA-1 或 MD5，因为这些太快了。 - 如果确实需要使用普通 hash 来保护密码，请在密码之前加几个字节的唯一随机数据（"salt"）再进行 hash 计算。将 salt 和 hash 密码保存在一起，并且不要为多个密码使用相同的 salt。
CWE	CWE ID 759
OWASP2017	None

返回目录

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

返回目录

漏洞名称	**Use of 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	**Use of Insufficiently Random Values**
默认严重性	3
摘要	X（文件） 文件第 N 行的 XXX（方法） 方法使用弱方法 XX（元素） 生成随机值。这些值可能被用作个人身份标识、会话 Token 或密码输入；但是，因为其随机性不足，攻击者可能推导出值。
解释	随机值经常被用作一种机制，用于防止恶意用户知道或预测给定的值，例如密码、密钥、或会话标识。根据这个随机值的用途，攻击者可以根据通常用于生成特定随机性的源预测下一次生成的数，或者之前生成的值；但是，虽然它们看起来是随机的，但大量统计样本可能显示它们并没有足够的随机性，可能的"随机"值空间比真正的随机样本小得多。这使得攻击都能够推导或猜出这个值，从而劫持其他用户的会话、假冒其他用户，或破解加密密钥（根据伪随机值的用途）。 应用程序使用弱方法生成伪随机值，因此可能使用相对小的样本大小确定其他数字。因为所使用的伪随机数发生器被设计为使用统计上分布均匀的值，所以它几乎就是确定性的。因此，收集了一些生成的值之后，攻击者就可能计算出过去的或未来的值。 具体而言，如果在安全环境中使用此伪随机值，例如一次性密码、密钥、secret 标识符或 salt，则攻击者将能够预测生成的下一个数字并窃取它，或猜到先前生成的值并破坏其原来的用途。
建议	总是使用密码安全的伪随机数生成器，不要使用基本的随机方法，特别是在安全环境下 使用您的语言或平台上内置的加密随机生成器，并确保其种子安全。不要为生成器提供非随机的弱种子。（在大多数情况下，默认是有足够的随机安全性）。 确保使用足够长的随机值，提高暴力破解的难度。
CWE	CWE ID 330
OWASP2017	A3-Sensitive Data Exposure

返回目录

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

返回目录

漏洞名称	**Use of Uninitialized Pointer**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中声明的变量在被 Y（文件）文件第 M 行的 YY（元素） 使用时未初始化。
解释	Null 指针取消引用可能会导致 run-time 异常、崩溃或其他意外行为。 已声明但未赋值的变量将隐式保留一个 null 值，直到为变量赋一个值。也可显式为变量设置 null 值，以确保清除其内容。因为 null 并不是一个真正的值，它可能没有对象变量和方法，因此访问 null 对象的内容但未提前验证它的设置时会导致 null 指针取消引用异常。
建议	创建任何变量时，请确保声明和使用之间的所有逻辑流都先为变量分配非 null 值。 在取消引用之前对接收到的所有变量和对象强制执行 null 值检查，确保其不包含其他地方分配给的 null 值。 可考虑是否通过分配 null 值覆盖初始化的变量。可考虑重新分配或释放这些变量。
CWE	CWE ID 457
OWASP2017	None

返回目录

漏洞名称	**Use Of Weak Hashing Primitive**
默认严重性	4
摘要	应用程序使用了 Y（文件） 文件第 M 行中的弱 hash 原语 YY（元素）
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 Hash 函数在加密方面有很多种用途，但大部分都与密钥派生函数和签名有关。如果以不安全的方式使用 hash 函数，就会影响密码 hash 或签名的完整性和机密性。 如果应用程序使用弱的或已被破解的 hash 函数来执行加密操作以提供完整性或身份验证功能，攻击者就可以利用已知的针对性攻击来破解签名或密码 hash 值。这可能导致数据的机密性、完整性和真实性被破坏。 应用程序的加密相关功能有问题。 应用程序使用弱 hash 原语。MD4、MD5、SHA-1 等已被发现存在冲突和其他弱点，使得这些方法不适合现在的部署。
建议	将 hash 函数更新为更安全的替代方法，例如： BLAKE2B（现代，软件速度快，可防止长度扩展攻击） SHA-2 系列散列 (SHA-256, SHA-384, SHA-512)
CWE	CWE ID 326
OWASP2017	None

返回目录

漏洞名称	**Use of Zero Initialized Pointer**
默认严重性	4
摘要	X（文件） 文件第 N 行 XX（元素） 中声明的变量在被 Y（文件）文件第 M 行的 YY（元素） 使用时未初始化。
解释	Null 指针取消引用可能会导致 run-time 异常、崩溃或其他意外行为。 已声明但未赋值的变量将隐式保留一个 null 值，直到为变量赋一个值。也可显式为变量设置 null 值，以确保清除其内容。因为 null 并不是一个真正的值，它可能没有对象变量和方法，因此访问 null 对象的内容但未提前验证它的设置时会导致 null 指针取消引用异常。
建议	创建任何变量时，请确保声明和使用之间的所有逻辑流都先为变量分配非 null 值。 在取消引用之前对接收到的所有变量和对象强制执行 null 值检查，确保其不包含其他地方分配给的 null 值。 可考虑是否通过分配 null 值覆盖初始化的变量。可考虑重新分配或释放这些变量。
CWE	CWE ID 457
OWASP2017	None

返回目录

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

返回目录

漏洞名称	**Weak Cryptographic Hash:Hardcoded PBE Salt**
默认严重性	3.0
摘要	Hardcoded salt 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用硬编码方式处理 salt 绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的 salt，而且还会使解决这一问题变得极其困难。在代码投入使用之后，无法轻易更改该 salt。如果攻击者知道该 salt 的值，他们就可以计算出该应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 X(文件)第 N 行中的 X（函数） 的调用中发现硬编码 salt。 例 1：下列代码使用了 hardcoded salt： … PKCS5_PBKDF2_HMAC(pass, strlen(pass), "2!@$(5#@532@%#$253l5#@$", 2, ITERATION, EVP_sha512(), outputBytes, digest); … 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： … PKCS5_PBKDF2_HMAC(pass, strlen(pass), salt, sizeof(salt), ITERATION, EVP_sha512(), outputBytes, digest); …
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	2.0
摘要	Hardcoded salt 可能会削弱系统安全性，一旦出现安全问题将无法轻易修正。
解释	使用硬编码方式处理 salt 绝非好方法。这不仅是因为所有项目开发人员都可以使用通过硬编码方式处理的 salt，而且还会使解决这一问题变得极其困难。在代码投入使用之后，无法轻易更改该 salt。如果攻击者知道该 salt 的值，他们就可以计算出该应用程序的“彩虹表”，并更轻松地确定散列值。 在这种情况下，在对 X(文件)第 N 行中的 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	3.0
摘要	基于密码的密钥派生函数所使用的迭代计数过低。
解释	密钥派生函数用来从基本密钥和其他参数派生出密钥。在基于密码的密钥派生函数中，基本密钥是一个密码，其他参数则是一个 salt 值和一个迭代计数。迭代计数传统上用于提高从一个密码生成密钥的代价。如果迭代计数过低，攻击的可行性就会提高，因为攻击者可以计算出应用程序的“彩虹表”，并更轻易地确定散列密码值。 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。 例 1： 以下代码使用 50 的迭代计数: ... #define ITERATION 50 ... PKCS5_PBKDF2_HMAC(pass, sizeof(pass), salt, sizeof(salt), ITERATION, EVP_sha512(), outputBytes, digest); ... 对基于密码的加密使用一个较低迭代计数的应用程序容易受到琐碎的字典式攻击 - 而这种类型的攻击正是基于密码的加密方案设计来进行防范的。
建议	使用一个基于密码的密钥派生函数时，迭代计数应至少为 1000，理想情况下为 100,000 或以上。迭代计数为 1000 将大幅增加穷尽式密码搜索的代价，而对派生各个密钥的代价不会产生显著影响。NIST SP 800-132 建议为关键密钥或非常强大的系统使用高达 10,000,000 的迭代计数。 当使用的迭代计数小于 1000 时，Fortify 安全编码规则将报告较严重的问题，当使用的迭代计数为 1000 到 100,000 之间时，将报告较低严重性的问题。如果源代码使用的迭代为 100,000 或以上，将不报告问题。 例 2： 以下代码使用 100,000 的迭代计数: ... #define ITERATION 100000 ... PKCS5_PBKDF2_HMAC(pass, sizeof(pass), salt, sizeof(salt), ITERATION, EVP_sha512(), outputBytes, digest); ...
CWE	CWE ID 916
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:Missing Required Step**
默认严重性	4.0
摘要	在 X(文件) 的第 N 行，在调用必需步骤之前以及调用第 M 行的 XXX（函数） 之后，代码将调用方法 X(函数)。在生成加密散列过程中，代码错过了对一个必需步骤的调用。

解释	加密散列的生成涉及多个步骤，错过对任何必需步骤的调用都会削弱所生成散列的强度。
	例 1： 以下代码跳过了对方法 CryptCreateHash 的调用，这将会导致创建不基于任何数据的散列：
	...
	if(!CryptAcquireContext(&hCryptProv, NULL,
	MS_ENH_RSA_AES_PROV, PROV_RSA_AES, 0)) {
	break;
	}
	if(!CryptHashData(hHash, (BYTE*)hashData, strlen(hashData), 0)) {
	break;
	}
	...

建议	在生成加密散列过程中实施所有必需步骤。尽可能明确地指定所用参数，以确保不会削弱散列的强度。
	示例 2：Example 1 中的代码可以按如下方式修复：
	...
	if(!CryptAcquireContext(&hCryptProv, NULL,
	MS_ENH_RSA_AES_PROV, PROV_RSA_AES, 0)) {
	break;
	}
	if(!CryptCreateHash(hCryptProv, CALG_SHA_256, 0, 0, &hHash)) {
	break;
	}
	if(!CryptHashData(hHash, (BYTE*)hashData, strlen(hashData), 0)) {
	break;
	}
	...

CWE	CWE ID 325
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:User-Controlled PBE Salt**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数包括的用户输入在第 N 行的基于密码的密钥派生函数 (PBKDF) 中使用的加密 salt 值范围内。这使得攻击者可以指定一个空 salt，既能更容易地确定散列值，又能泄露有关程序如何执行加密散列的信息。可能受污染的用户输入不应该作为 salt 参数传递到基于密码的密钥派生函数 (PBKDF)。
解释	在以下情况下会发生 Weak Cryptographic Hash:用户控制 PBE 的 Salt 问题将在以下情况下出现： 1. 数据通过一个不可信赖的数据源进入程序 在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。 2. 用户控制的数据包括在 salt 中，或完全作为基于密码的密钥派生函数中的 salt 使用。 在这种情况下，在 Y（文件）中第 M 行的 Y（函数） 使用该数据。 如同许多软件安全漏洞一样，Weak Cryptographic Hash:用户控制 PBE 的 Salt 是到达终点的一个途径，其本身并不是终点。从本质上看，这些漏洞是显而易见的：攻击者可将恶意数据传递到应用程序，然后这些数据被用作 PSKDF 中的全部或部分 salt。 用户控制 salt 的问题在于，它可以实现几个不同的攻击： 1. 攻击者可以利用这一漏洞，指定一个空 salt 作为自己的密码。由此，可以轻易地使用许多不同的散列快速地对其密码执行派生，以泄露有关您的应用程序中使用的 PBKDF 实现的信息。这样，通过限制所用散列的特定变体，可以更轻松地“破解”其他密码。 2. 如果攻击者能够操纵其他用户的 salt，或者诱骗其他用户使用空 salt，这将使他们能够计算应用程序的“彩虹表”，并更轻松地确定派生值。 例 1： 以下代码使用用户控制 salt： ... salt = getenv("SALT"); PKCS5_PBKDF2_HMAC(pass, sizeof(pass), salt, sizeof(salt), ITERATION, EVP_sha512(), outputBytes, digest); ... 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。
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

返回目录

	理。在系统中采用明文的形式存储 salt，会造成任何有充分权限的人读取和无意中误用 salt。
CWE	CWE ID 328, CWE ID 760
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Hash:User-Controlled Salt**
默认严重性	3.0
摘要	X(文件) 中的 XX（函数） 函数包括的用户输入在第 N 行的加密散列中使用的 salt 值范围内。这使得攻击者可以指定一个空 salt，既能更容易地确定散列值，又能泄露有关程序如何执行加密散列的信息。对于会生成作为 salt 传递的加密散列的函数，不应使用被污染的 salt 参数进行调用。

解释	在以下情况下会发生 Weak Cryptographic Hash:用户控制 salt 问题将在以下情况下出现：
	1. 数据通过一个不可信赖的数据源进入程序
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	2. 用户控制的数据包括在 salt 中，或完全用作加密散列函数中的 salt。
	在这种情况下，在 Y（文件）中第 M 行的 Y（函数） 使用该数据。
	如同许多软件安全漏洞一样，Weak Cryptographic Hash:用户控制 Salt 是到达终点的一个途径，其本身并不是终点。从本质上看，这些漏洞是显而易见的：攻击者可将恶意数据传递到应用程序，然后这些数据被用作加密散列中的全部或部分 salt。
	用户控制 salt 的问题在于，它可以实现几个不同的攻击：
	1. 攻击者可以利用这一漏洞，为被散列的数据指定一个空 salt。由此，可以轻易地使用许多不同的散列算法控制被散列的数据，以泄露有关您的应用程序中使用的散列实现的信息。这样，通过限制所用散列的特定变体，可以更轻松地“破解”其他密码。
	2. 如果攻击者能够操纵其他用户的 salt，或者诱骗其他用户使用空 salt，这将使他们能够计算应用程序的“彩虹表”，并更轻松地确定哈希值。
	例 1： 以下代码使用用户控制 salt 进行密码散列：
	...
	salt = getenv("SALT");
	password = crypt(getpass("Password:"), salt);
	...
	Example 1 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 来操纵用于对密码执行散列的 salt。此外，此代码还使用了 crypt() 函数，该函数不应用于对密码执行加密散列。一旦程序发布，撤消与用户控制的 salt 相关的问题就会非常困难，因为很难知道恶意用户是否确定了密码散列的 salt。

建议	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

返回目录

OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Signature:Insufficient Key Size**
默认严重性	4.0
摘要	X(文件) 文件中的 XX（函数） 方法使用了强大的加密签名算法，但密钥长度不够，从而导致签名更容易受到强力攻击。原本强大的加密签名算法如果使用的密钥长度不够，就可能会更加容易受到强力攻击。
解释	当前的密码指南建议，RSA 和 DSA 算法使用的密钥长度至少应为 2048 位。但是，计算能力和因子分解技术方面的持续进步 [1] 意味着未来将不可避免地需要提高建议的密钥大小。 例 1： 以下代码生成 1024 位 DSA 签名密钥。 ... DSA_generate_parameters_ex(dsa, 1024, NULL, 0, NULL, NULL, NULL); ...
建议	确保 RSA 和 DSA 签名密钥的长度不少于 2048 位。 例 2： 以下代码生成 2048 位 DSA 签名密钥。 ... DSA_generate_parameters_ex(dsa, 2048, NULL, 0, NULL, NULL, NULL); ...
CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Cryptographic Signature:Missing Required Step**
默认严重性	4.0
摘要	在 X(文件) 的第 N 行，在调用必需步骤之前以及调用第 M 行的 XXX（函数） 之后，代码将调用方法 X(函数)。在生成加密签名过程中，代码错过了对一个必需步骤的调用。

解释	加密签名的生成涉及多个步骤，错过对任何必需步骤的调用都会削弱所生成签名的强度。
	例 1： 以下代码跳过了对方法 EVP_SignUpdate 的调用，这将会导致创建不基于任何数据的签名：
	...
	rv = EVP_SignInit(ctx, EVP_sha512());
	...
	rv = EVP_SignFinal(ctx, sig, &sig_len, key);
	...

建议	在生成加密签名过程中实施所有必需步骤。尽可能明确地指定所用参数，以确保不会削弱签名的强度。
	示例 2：Example 1 中的代码可以按如下方式修复：
	...
	rv = EVP_SignInit(ctx, EVP_sha512());
	...
	rv = EVP_SignUpdate(ctx, data, data_len);
	...
	rv = EVP_SignFinal(ctx, sig, &sig_len, key);
	...

CWE	CWE ID 325
OWASP2017	A3 Sensitive Data Exposure

返回目录

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

返回目录

漏洞名称	**Weak Encryption:Inadequate RSA Padding**
默认严重性	4.0
摘要	X(文件) 中的 XX（函数） 方法执行不带 OAEP 填充模式的公钥 RSA 加密，因此加密机制比较脆弱。公钥 RSA 加密在不使用 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：
	void encrypt_with_rsa(BIGNUM *out, BIGNUM *in, RSA *key) {
	u_char *inbuf, *outbuf;
	int ilen;
	...
	ilen = BN_num_bytes(in);
	inbuf = xmalloc(ilen);
	BN_bn2bin(in, inbuf);
	if ((len = RSA_public_encrypt(ilen, inbuf, outbuf, key,
	RSA_NO_PADDING)) <= 0) {
	fatal("encrypt_with_rsa() failed");
	}
	...
	}

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：
	void encrypt_with_rsa(BIGNUM *out, BIGNUM *in, RSA *key) {
	u_char *inbuf, *outbuf;
	int ilen;
	...
	ilen = BN_num_bytes(in);
	inbuf = xmalloc(ilen);
	BN_bn2bin(in, inbuf);
	if ((len = RSA_public_encrypt(ilen, inbuf, outbuf, key,
	RSA_PKCS1_OAEP_PADDING)) <= 0) {
	fatal("encrypt_with_rsa() failed");
	}
	...
	}

CWE	CWE ID 780
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insecure Initialization Vector**
默认严重性	4.0
摘要	初始化矢量应该使用加密伪随机数值生成器进行创建。
解释	初始化矢量 (IV) 应该使用加密伪随机数值生成器进行创建。如果不使用随机 IV，则结果密码文本可预测性会高得多，容易受到字典式攻击。 例 1： 以下代码使用硬编码字符串创建非随机 IV。 unsigned char * iv = "12345678"; EVP_EncryptInit_ex(&ctx, EVP_idea_gcm(), NULL, key, iv);

建议	将足够长度的初始化矢量 (IV) 与适当的随机数据源中的字节结合使用。
	例 2： 以下代码使用 /dev/random 作为熵源，创建了完全随机的 IV：
	unsigned char * iv;
	int fd = open("/dev/random", O_RDONLY);
	if (fd != -1) {
	(void) read(fd, (void *)&iv, ivlength);
	(void) close(fd);
	}
	EVP_EncryptInit_ex(&ctx, EVP_idea_gcm(), NULL, key, iv);

CWE	CWE ID 329
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insecure Mode of Operation**
默认严重性	4.0
摘要	X(文件) 中的函数 XX（函数） 在第 N 行上将密码加密算法用于不安全的操作模式。请勿将密码加密算法用于不安全的操作模式。
解释	块密码操作模式是一种算法，用来描述如何重复地应用密码的单块操作，以安全地转换大于块的数据量。一些操作模式包括电子代码本 (ECB)、密码块链 (CBC)、密码反馈 (CFB) 和计数器 (CTR)。 ECB 模式本质上较弱，因为它会对相同的明文块生成一样的密文。 CBC 模式容易受到密文填塞攻击。CTR 模式由于没有这些缺陷，使之成为一个更好的选择。 示例 1：以下代码将 AES 密码用于 ECB 模式： EVP_EncryptInit_ex(&ctx, EVP_aes_256_ecb(), NULL, key, iv);
建议	加密大于块的数据时，避免使用 ECB 和 CBC 操作模式。CBC 模式效率较低，并且在和 SSL 一起使用时会造成严重风险 [1]。请改用 CCM (Counter with CBC-MAC) 模式，或者如果更注重性能，则使用 GCM （Galois/Counter Mode）模式。 示例 2：以下代码将 AES 密码用于 GCM 模式： EVP_EncryptInit_ex(&ctx, EVP_aes_256_gcm(), NULL, key, iv);
CWE	CWE ID 327
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Insufficient Key Size**
默认严重性	4.0
摘要	X(文件) 中的 XX（函数） 方法使用了密钥长度不够的加密算法，导致加密数据容易受到强力攻击。另外，当使用的密钥长度不够时，强大的加密算法便容易受到强力攻击。

解释	当前的密码指南建议，RSA 算法使用的密钥长度至少应为 2048 位。但是，计算能力和因子分解技术方面的持续进步 [1] 意味着未来将不可避免地需要提高建议的密钥大小。
	例 1： 以下代码可生成 512 位 RSA 加密密钥。
	EVP_PKEY * get_RSA_key() {
	unsigned long err;
	EVP_PKEY * pkey;
	RSA * rsa;
	rsa = RSA_generate_key(512, 35, NULL, NULL);
	if (rsa == NULL) {
	err = ERR_get_error();
	printf("Error = %s\n",ERR_reason_error_string(err));
	return NULL;
	}
	pkey = EVP_PKEY_new();
	EVP_PKEY_assign_RSA(pkey, rsa);
	return pkey;
	}
	对于对称加密，密钥长度必须至少为 128 位。

建议	最低限度下，确保 RSA 密钥长度不少于 2048 位。未来几年需要较强加密的应用程序的密码长度应至少为 4096 位。
	
	如果使用 RSA 算法，请确保特定密钥的长度至少为 2048 位。
	例 2： 以下代码可生成 2048 位 RSA 加密密钥。
	EVP_PKEY * get_good_RS_key() {
	unsigned long err;
	EVP_PKEY * pkey;
	RSA * rsa;
	rsa = RSA_generate_key(2048, 35, NULL, NULL);
	if (rsa == NULL) {
	err = ERR_get_error();
	printf("Error = %s\n",ERR_reason_error_string(err));
	return NULL;
	}
	pkey = EVP_PKEY_new();
	EVP_PKEY_assign_RSA(pkey, rsa);

返回目录

	return pkey;
	}
	同样，如果使用对称加密，请确保特定密钥的长度至少为 128 位（适用于 AES）和 168 位（适用于 Triple DES）。
	例 3: 以下代码使用 128 位 AES 加密密钥。
	int do_crypt(int do_encrypt, unsigned char key. unsigned char iv) {
	EVP_CIPHER_CTX ctx;
	...
	EVP_CIPHER_CTX_init(&ctx);
	EVP_CipherInit_ex(&ctx, EVP_aes_128_ecb(), NULL, NULL, NULL, do_encrypt);
	EVP_CIPHER_CTX_set_key_length(&ctx, 128);
	EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, do_encrypt);
	...
	}

CWE	CWE ID 326
OWASP2017	A3 Sensitive Data Exposure

返回目录

漏洞名称	**Weak Encryption:Missing Required Step**
默认严重性	4.0
摘要	在 X(文件) 的第 N 行，在调用必需步骤之前以及调用第 M 行的 XXX（函数） 之后，代码将调用方法 X(函数)。在对称密钥生成、加密或解密过程中，代码错过了对一个必需步骤的调用。
解释	对称密钥的生成以及加密和解密涉及多个步骤，错过任何必需步骤都可能会削弱所生成对称密钥或密文的强度，或导致对现有密文的不正确解密。 例 1： 以下代码跳过了对 EVP_DecryptUpdate 的调用，这将导致密文解密失败。 ... EVP_DecryptInit_ex(&ctx, EVP_aes_256_gcm(), NULL, key, iv); ... if(!EVP_DecryptFinal_ex(&ctx, outBuf+outBytes, &tmpOutBytes)) prtErrAndExit(1, "ERROR: EVP_DecryptFinal_ex did not work...\n"); ...

建议	在生成对称密钥或密文，或解密现有密文的过程中，实施所有必需步骤。尽可能明确地指定所用参数，以确保不会削弱加密的强度。
	示例 2：Example 1 中的代码可以按如下方式修复：
	...
	EVP_DecryptInit_ex(&ctx, EVP_aes_256_gcm(), NULL, key, iv);
	...
	if(!EVP_DecryptUpdate(&ctx, outBuf + outBytes, &tmpOutBytes, buf2crypt, bytesInBuf))
	prtErrAndExit(1, "ERROR: EVP_DecryptUpdate did not work...\n");
	...
	if(!EVP_DecryptFinal_ex(&ctx, outBuf+outBytes, &tmpOutBytes))
	prtErrAndExit(1, "ERROR: EVP_DecryptFinal_ex did not work...\n");
	....

CWE	CWE ID 325
OWASP2017	A3 Sensitive Data Exposure

返回目录

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

返回目录

漏洞名称	**Weak Randomness Biased Random Sample**
默认严重性	4
摘要	X（文件） 文件第 N 行中 XX（元素） 的可信任数字生成流程在 Y（文件） 文件第 M 行中的 YY（元素） 生成偏置值
解释	应用程序通过加密保护机密和其他敏感数据或个人身份识别数据。如果加密实现中有缺陷，就可能影响应用程序数据的完整性、真实性或机密性。 保证密码系统安全性的要求之一就是随机性。它们被用于加密中的各种参数，例如密钥、初始化向量，salt 等。 如果加密时使用的随机数偏向某些值，则攻击者可以利用此信息在执行暴力破解时纠正此偏差以提高猜到正确值的几率。 应用程序的加密相关功能有问题。 影响加密的随机数生成函数的输出中存在偏差。这种偏差是滥用模运算提取低于给定阈值的随机数造成的。
建议	一般来说，对于与加密相关的目的，应尽量使用底层操作系统的随机源来利用内核的熵池。 不应从用户空间为这些源提供种子，所以使用这些源的时候不要为这些对象提供您自己的种子。 在生成特定阈值下的随机数时，应用程序必须确保提取的值分布均匀。这样做的初级方法是持续调用随机数生成器，直到生成一个低于阈值的数字，但这对于较低的阈值其效率不是很高。安全代码示例中有更有效率的方法。
CWE	CWE ID 330
OWASP2017	None

返回目录

漏洞名称	**Wrong Memory Allocation**
默认严重性	4
摘要	Y（文件） 文件第 M 行中的函数 YY（元素） 为缓冲区分配了不正确的计算大小，导致写入值与写入的缓冲区大小不匹配。
解释	不正确的内存分配可能导致意外值覆写部分内存，从而发生意外行为。在某些情况下，如果攻击者可以控制内存分配和写入的值，就可能导致被执行恶意代码。 某些内存分配函数需要以参数的方式提供一个大小值。分配的大小应该使用提供的值导出，方法是使用提供的预期源的长度值乘以该长度的大小。如果未能通过正确的算法计算出准确的大小，就可能导致源溢出目标。
建议	一定要使用正确的算法计算大小。 对于内存分配来说，就是要根据分配源计算分配大小： 根据预期源的长度导出大小值，以确定要处理的单元数量。 一定要从编程的角度考虑各个单元的大小及其到内存单位的转换——例如，对单元类型使用 sizeof。 内存分配应该是所写单元数量的倍数，乘以每个单元的大小。
CWE	CWE ID 131
OWASP2017	None

返回目录

漏洞名称	**Wrong Size t Allocation**
默认严重性	4
摘要	Y（文件） 文件第 M 行中的函数 YY（元素） 为缓冲区分配了不正确的计算大小，导致写入值与写入的缓冲区大小不匹配。
解释	不正确的内存分配可能导致意外值覆写部分内存，从而发生意外行为。在某些情况下，如果攻击者可以控制内存分配和写入的值，就可能导致被执行恶意代码。 某些内存分配函数需要以参数的方式提供一个大小值。分配的大小应该使用提供的值导出，方法是使用提供的预期源的长度值乘以该长度的大小。如果未能通过正确的算法计算出准确的大小，就可能导致源溢出目标。
建议	一定要使用正确的算法计算大小。 对于内存分配来说，就是要根据分配源计算分配大小： 根据预期源的长度导出大小值，以确定要处理的单元数量。 一定要从编程的角度考虑各个单元的大小及其到内存单位的转换——例如，对单元类型使用 sizeof。 内存分配应该是所写单元数量的倍数，乘以每个单元的大小。
CWE	CWE ID 789
OWASP2017	None

返回目录

漏洞名称	**XML External Entity Injection**
默认严重性	4.0
摘要	在 X(文件) 的第 N 行中，XX（函数） 方法允许外部实体引用。攻击者可以利用此调用将 XML 外部实体注入 XML 文档以显示文件内容或内部网络实体。标识的方法允许外部实体引用。攻击者可以利用此调用将 XML 外部实体注入 XML 文档以显示文件内容或内部网络资源。

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。
	2. 数据写入到 XML 文档的 DTD（文档类型定义）<ENTITY> 的元素中。
	应用程序通常使用 XML 来存储数据或发送消息。当 XML 用于存储数据时，XML 文档通常会像数据库一样进行处理，而且可能会包含敏感信息。XML 消息通常在 web 服务中使用，也可用于传输敏感信息。XML 消息甚至还可用于发送身份验证凭据。
	如果攻击者能够写入原始 XML，则可以更改 XML 文档和消息的语义。在危害最轻的情况下，攻击者可能会插入嵌套的实体引用，导致 XML 解析器不断消耗越来越多的 CPU 资源。在发生 XML 外部实体注入这种危害更大的攻击的情况下，攻击者可以添加 XML 元素，从而暴露本地文件系统资源的内容或显示是否存在内部网络资源。
	例 1：下面是一些易受 XXE 攻击的 Objective-C 代码：
	- (void) parseSomeXML: (NSString *) rawXml {
	BOOL success;
	NSData *rawXmlConvToData = [rawXml dataUsingEncoding:NSUTF8StringEncoding];
	NSXMLParser *myParser = [[NSXMLParser alloc] initWithData:rawXmlConvToData];
	[myParser setShouldResolveExternalEntities:YES];
	[myParser setDelegate:self];
	}
	假设攻击者能够控制 rawXml，该 XML 的形式如下所示：
	<?xml version="1.0" encoding="ISO-8859-1"?>
	<!DOCTYPE foo [
	<!ELEMENT foo ANY >
	<!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>
	当服务器对该 XML 进行评估时，<foo> 元素将包含 boot.ini 文件的内容。

建议	将用户提供的数据写入 XML 时，应该遵守以下准则： 1. 禁用外部实体引用。 2.不要使用从用户输入派生的名称创建标签或属性。 3.写入到 XML 之前，先对用户输入进行 XML 实体编码。

返回目录

	3. 将用户输入包含在 CDATA 标签中。
CWE	CWE ID 611
OWASP2017	A4 XML External Entities (XXE)

返回目录

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

返回目录

	当使用 SAX 解析器时，第二个 <price> 标签中的值将会覆盖第一个 <price> 标签中的值。这样，攻击者就可以只花 1 美元购买一双价值 100 美元的鞋。
建议	将用户提供的数据写入 XML 时，应该遵守以下准则: 1.不要使用从用户输入派生的名称创建标签或属性。 2.写入到 XML 之前，先对用户输入进行 XML 实体编码。 3. 将用户输入包含在 CDATA 标签中。
CWE	CWE ID 91
OWASP2017	A1 Injection

返回目录

漏洞名称	**XPath Injection**
默认严重性	4.0
摘要	在 X(文件) 的第 N 行，XX（函数） 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。标识方法会调用通过未经验证的输入构建的 XPath 查询。此调用使攻击者可修改语句的含义或者执行任意 XPath 查询。

解释	XPath injection 会在以下情况中出现：
	1. 数据从一个不可信赖的数据源进入程序。
	在这种情况下，数据进入 X（文件）的第 N 行的 X（函数） 中。
	<IfDef var="ConditionalDescriptions">
	
	<ConditionalText condition="taint:number">
	在这种情况下，即使数据为数字类型，由于其未经验证仍会被视为恶意内容，因此程序仍将报告漏洞，但是优先级值会有所降低。
	</ConditionalText>
	
	</IfDef>
	2. 数据用于动态构造一个 XPath 查询。
	在这种情况下，查询将传递到 Y（文件）的第 M 行中的 Y（函数）。
	示例 1：以下 Objective-C 代码会调用 C API，它会动态地构造并执行一个 XPath 查询，用于检索指定帐户 ID 的电子邮件地址。由于该帐户 ID 是从 HTTP 请求中读取的，因此不受信任。
	...
	NSString *accountStr = account.text;
	xmlXPathContextPtr xpathCtx;
	NSString *query = @"/accounts/account[actId='" + accountStr + @"']/email/text()";
	xpathCtx = xmlXPathNewContext(doc);
	/* Evaluate XPath expression */
	xmlChar *queryString =
	(xmlChar *)[query cStringUsingEncoding:NSUTF8StringEncoding];
	xpathObj = xmlXPathEvalExpression(queryString, xpathCtx);
	...
	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 acctID 不包含单引号字符

返回目录

	时，才会正确执行这一查询。如果攻击者为 acctID 输入字符串 1' or '1' = '1，则该查询会变成： /accounts/account[acctID='1' or '1' = '1']/email/text() 附加条件 1' or '1' = '1 会使 where 从句永远评估为 true，因此该查询在逻辑上将等同于一个更为简化的查询： //email/text() 通常，查询必须仅返回已通过身份验证的用户所拥有的条目，而通过以这种方式简化查询，攻击者就可以规避这一要求。现在，查询会返回存储在文档中的所有电子邮件地址，而不论其指定所有者是谁。

建议	造成 XPath injection 漏洞的根本原因在于攻击者能够改变 XPath 查询的上下文，导致程序员期望解释为数据的某个数值最终被解释为命令。构建 XPath 查询后，程序员知道哪些数值应解释为命令的一部分，哪些数值应解释为数据。
	为了防止攻击者侵犯程序员的各种预设情况，可以使用允许列表的方法，确保 XPath 查询中由用户控制的数值完全来自于预定的字符集合，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。
	示例 2：以下 Objective-C 代码会调用 C API，它会显示一个基于 Example 1 进行改进的方法。
	...
	NSString *accountStr = account.text;
	if (accountStr != NULL) {
	NSInteger acct = [accountStr integerValue];
	if (acct != 0) {
	xmlXPathContextPtr xpathCtx;
	NSString *query = @"/accounts/account[actId='" + accountStr + @"']/email/text()";
	xpathCtx = xmlXPathNewContext(doc);
	/* Evaluate XPath expression */
	xmlChar *queryString = (xmlChar *)[query cStringUsingEncoding:NSUTF8StringEncoding];
	xpathObj = xmlXPathEvalExpression(queryString, xpathCtx);
	...

CWE	CWE ID 643
OWASP2017	A1 Injection

返回目录