**openEuler社区安全编程规范**

# 变量

## **规则 不要使用未初始化的变量**

 **【描述】**

这里的变量，指的是局部动态变量，并且还包括内存堆上申请的内存块。如malloc出来的内存没有被初始化为0，要确保这些内存被引用前已被初始化。

因为他们的初始值都是不可预料的，所以必须初始化以后才能访问。

void Foo(...)

{

 int data;

 Bar(data); // Bad: 未初始化就使用

 ...

}

如果有不同分支，要确保所有分支都得到初始化后才能使用：

void Foo(...)

{

 int data;

 if (...) {

 data = 100;

 }

 Bar(data); // Bad: 部分分支该值未初始化

 ...

}

## **规则** 指向资源句柄或描述符的变量，在资源释放后立即赋予新值

**【描述】**
悬空指针（Dangling pointers）会导致可利用的双重释放和访问已释放的内存漏洞。一种消除悬空指针并避免许多与内存相关的漏洞的简单而有效的方法是，将释放后的指针立即设置为NULL，或将其设置为另一个有效的对象。针对此类问题，本规范延伸要求为资源释放后，对应的变量应该立即赋予新值，防止后续又被重新引用。如果释放语句刚好在变量作用域的最后一句，可以不进行赋值

**【错误代码示例】**
下面错误示例代码中，消息的类型用于确定如何处理消息本身。假定messageType是一个整数，并且message是指向动态分配的字符数组的指针。如果messageType等于value1，则会处理相应消息。messageType等于value2时发生类似的操作。但是，如果messageType == value1的结果为true，并且messageType == value2的结果也为true，则message将被释放两次，从而导致双重释放漏洞。

char \*message;

int messageType;

/\* 初始化 message 和 messageType \*/

if (messageType == value1) {

 /\* 处理类型为value1的消息 \*/

 free(message);

}

/\* ... \*/

if (messageType == value2) {

 /\* 处理类型为value2的消息 \*/

 free(message);

}

**【正确代码示例】**
对一个NULL指针调用free时，free函数不会采取任何操作。释放后设置message为NULL可以消除message使用指针多次释放同一内存的可能性。

char \*message = NULL;

int messageType;

/\* 初始化 message 和 messageType \*/

if (messageType == value1) {

 /\* 处理类型为value1的消息 \*/

 free(message);

 message = NULL;

}

/\* ... \*/

if (messageType == value2) {

 /\* 处理类型为value2的消息 \*/

 free(message);

 message = NULL;

}

## 规则 类的成员变量必须通过构造函数初始化

**【描述】**

如果类中声明了变量，则必须在构造函数中对变量进行初始化。

**【正确代码示例】**

class CMsg {

public:

 CMsg();

 ~CMsg();

protected:

 int size;

 unsigned char \*msg;

};

CMsg::CMsg():size(0),msg(NULL)

{

}

## 规则 不要通过对指针变量进行sizeof操作来获取数组大小

**【描述】**

编码人员往往由于粗心，将指针当做数组进行sizeof操作，导致实际的执行结果与预期不符。

**【错误代码示例】**

下面的代码，buffer和path分别是指针和数组，编码人员想对这2个内存进行清0操作，但由于编码人员的疏忽，第5行代码，将内存大小误写成了sizeof，与预期不符。

char \*buffer = (char \*)malloc(size);

char path[MAX\_PATH];

...

memset(path, 0, sizeof(path));

memset(buffer, 0, sizeof(buffer));

**【正确代码示例】**

char \*buffer = (char \*)malloc(size);

char path[MAX\_PATH];

...

memset(path, 0, sizeof(path));

memset(buffer, 0, size);

函数的参数是数组时，数组参数在编译时会退化为指向类型的指针，对函数的数组参数使用sizeof运算，运算通常与预期不符。

**【错误代码示例】**

void Func(int array[LEN])函数的参数是数组时，参数array在编译时会退化为指向类型的指针，即退化为void Func(int \*array)。因此，在这个函数内使用sizeof(array)等同于sizeof(int \*)。

void Func(int array[LEN])

{

…

memset(array, 0, sizeof(array));

…

}

**【正确代码示例】**

void Func(int array[], size\_t len)

{

…

memset(array, 0, len);

…

}

## 规则 全局变量的访问如果涉及多个线程，需要考虑多线程竞争条件问题

**【描述】**

应该尽可能减少全局变量的使用，如果多个线程会访问到该全局变量，则访问过程必须加锁。

**【正确代码示例】**

以下代码中，g\_list是全局变量，对链表进行搜索操作时，在while循环语句的前后加锁，避免了多线程竞争问题。

ItemList \*g\_list = NULL;

ItemList \*SearchList(const char \*name)

{

 Lock();

 ItemList \*p = g\_list;

 while (p != NULL) {

 if (strcmp(p->name, name) == 0) {

 break;

 }

 p = p->next;

 }

 UnLock();

 return p;

}

性能敏感的代码，请考虑采用原子操作或者无锁算法。

## 规则 避免大量栈分配

**【描述】**

程序在运行期间，函数内的局部变量存储在栈中，而栈的大小是有限的，如果申请过大的局部变量，可能导致出现运行出错。建议在定义函数局部变量时候，对其大小进行限制，例如，linux内核的默认构建配置文件中，限制函数帧的大小不超过1024字节，如果超出限制则出现编译告警。

程序在设计函数的局部变量规模时，需要充分考虑调用栈的开销(特别是递归调用时)，避免因占用过多的栈空间导致程序运行失败。

**【错误代码示例】**
如下代码中的 buff[MAX\_BUFF] 数组占用空间过大，可能导致栈空间不够，程序发生 stack overflow 异常：

#define MAX\_BUFF 0x1000000

void Func()

{

 char buff[MAX\_BUFF] = {0};

 ...

}

**【正确代码示例】**
如下正确代码示例中，通过在动态分配内存的方式，避免栈空间占用空间过大的问题：

#define MAX\_BUFF 0x1000000

Func()

{

 char \*buf = (char\*)malloc(MAX\_BUFF);

 if (buf == NULL) {

 /\* 错误处理 \*/

 }

 ...

}

如上代码中须检查动态分配函数的返回值，如果分配内存大小的来源不可信，则需要注意校验其值的范围。

**【错误代码示例】**
递归也会导致大量堆栈分配。递归函数必须确保它们不会由于过度递归而耗尽堆栈。如下递归函数：

unsigned long FooAdd1(unsigned int n)

{

 if (n == 0) {

 return 0;

 } else if (n == 1 || n == 2) {

 return 1;

 } else {

 return FooAdd1(n - 1) + FooAdd1(n - 2);

 }

}

所需的栈空间随着参数 n 呈线性增加，如果 n 较大，会导致程序异常终止。

**【正确代码示例】**
重构上面错误示例中递归函数，消除了递归的使用

unsigned long FooAdd2(unsigned int n)

{

 unsigned long prev = 1;

 unsigned long cur = 1;

 unsigned int i;

 if (n == 0) {

 return 0;

 } else if (n == 1 || n == 2) {

 return 1;

 }

 for (i = 3; i <= n; i++) {

 unsigned long tmp = cur;

 cur = cur + prev;

 prev = tmp;

 }

 return cur;

}

因为没有递归，所以所需的堆栈空间量不取决于参数 n，从而大大降低了栈溢出的风险。

# 断言

## 规则 断言不能用于校验程序在运行期间可能导致的错误，可能发生的错误要用错误处理代码来处理

**【描述】**

断言主要用于调试期间，在编译Release版本时将其关闭。因此，断言应该用于防止不正确的程序员假设，而不是用于运行时错误检查。

断言永远不应用于验证是否存在运行时（与逻辑相对）错误，例如

* 无效的用户输入（包括命令行参数和环境变量）
* 文件错误（例如，打开，读取或写入文件时出错）
* 网络错误（包括网络协议错误）
* 内存不足的情况（例如，malloc()类似的故障）
* 系统资源耗尽（例如，文件外描述符，进程，线程）
* 系统调用错误（例如，执行文件，锁定或解锁互斥锁时出错）
* 无效的权限（例如，文件，内存，用户）

例如，防止缓冲区溢出的代码不能使用断言实现，因为该代码必须编译到Release版本的可执行文件中。
特别是，断言通常不适合被编译到服务器程序或嵌入式系统。如果由恶意用户触发断言失败，会导致拒绝服务攻击。在这种情况下，更适合使用软故障模式，例如写入日志文件和拒绝请求。

**【错误代码示例】**
以下代码的所有ASSERT的用法是错误的。例如，错误的使用ASSERT宏来验证内存分配是否成功，因为内存的可用性取决于系统的整体状态，并且在过程生命周期中的任何时候都可能耗尽，所以必须准备一个健壮的程序来妥善处理并从其耗尽中恢复。因此，使用ASSERT宏来验证内存分配是否成功将是不合适的，因为这样做可能导致进程突然终止，从而开启了拒绝服务攻击的可能性。

FILE \*fp = fopen(path, "r");

assert(fp != NULL); // 错误用法：文件有可能打开失败

char \*str = (char \*)malloc(MAX\_LINE);

assert(str != NULL); // 错误用法：内存有可能分配失败

ReadLine(fp, str);

char \*p = strstr(str, 'age=');

assert(p != NULL); // 错误用法：文件中不一定存在该字符串

int age = atoi(p + 4);

assert(age > 0); // 错误用法：文件内容不一定符合预期

**【正确代码示例】**
下面代码演示了如何重构上面的错误代码

FILE \*fp = fopen(path, "r");

if (fp == NULL) {

 /\* 错误处理 \*/

}

char \*str = (char \*)malloc(MAX\_LINE);

if (str == NULL) {

 /\* 错误处理 \*/

}

ReadLine(fp, str);

char \*p = strstr(str, 'age=');

if (p == NULL) {

 /\* 错误处理 \*/

}

int age = atoi(p + 4);

if (age <= 0) {

 /\* 错误处理 \*/

}

# 数组与字符串

## 规则 数组作为函数参数时，必须同时将其长度作为函数的参数

**【描述】**

通过函数参数传递数组或一块内存进行读写操作时，函数参数必须同时传递数组可容纳元素的个数或所传递的内存块大小，该参数是必需的，以确保函数不会访问数组范围之外的内存，否则函数在使用数组下标或访问内存偏移时，无法判断下标或偏移的合法范围，产生越界访问的漏洞。对于每个数组参数都应该存在一个相应的长度参数，代表数组中元素的最大数量。

请注意，在本规则中，”数组“用于表示数组、字符串或指向内存连续块的任何其他指针，在内存连续块中（潜在地）存储了一个或多个特定类型的元素。这些术语都是有效的同义词。另外，该规则要求附加的长度参数指定的是可以存储在数组中的最大元素数，而不是数组的最大大小（以字节为单位），因为

1. 例如，在整数数组的情况下，以字节为单位考虑数组大小是没有意义的。
2. 如果需要数组的字节大小，则可以从数组中的元素数得出。
3. 最好不要要求用户以数组字节为单位计算大小，以免增加功能用户的认知负担。

在大多数情况下，元素数和字节数之间的区别是没有意义的：两者之间存在清晰的映射，无论如何，从元素数的角度考虑都比较容易。不幸的是，当处理多字节字符串时，这个问题可能会变得混乱，因为被操作的逻辑实体与用于实现它的类型不同。这里，必须记住数组的类型是字符，而不是多字节字符。因此，数组中的元素数表示为字符数。

**【错误代码示例】**
以下代码中，函数ParseMsg不知道msg的范围，容易产生内存越界访问漏洞。

int ParseMsg(unsigned char \*msg)

{

 /\* ... \*/

}

/\* ... \*/

size\_t len = /\* ... \*/

unsigned char \*msg = (unsigned char \*)malloc(len);

/\* ... \*/

ParseMsg(msg);

/\* ... \*/

**【正确代码示例】**
正确的做法是将msg的大小作为参数传递到ParseMsg中，如下代码：

int ParseMsg(unsigned char \*msg, size\_t msgLen)

{

 ASSERT(msg != NULL);

 ASSERT(msgLen != 0);

 /\* ... \*/

}

/\* ... \*/

size\_t len = /\* ... \*/

unsigned char \*msg = (unsigned char \*)malloc(len);

/\* ... \*/

ParseMsg(msg, len);

/\* ... \*/

**【例外】**

可以通过运行时约束处理程序保证不发生越界读或写的函数可能会忽略最大元素参数。

## 规则 函数的指针参数如果不是用于修改所指向的对象就应该声明为指向const的指针

**【描述】**
const 指针参数，将限制函数通过该指针修改所指向对象，使代码更牢固、安全。

**【正确代码示例】**

int strncmp(const char \*s1, const char \*s2, size\_t n); //Good：不变参数声明为const

注意：

指针参数要不要加const取决于函数设计，而不是看函数实体内有没有发生“修改对象”的动作。

## 规则 确保字符串存储有足够的空间容纳字符数据和’\0’结束符

**【描述】**

将数据复制到不足以容纳数据的缓冲区，会导致缓冲区溢出。缓冲区溢出经常发生在字符串操作中[Seacord 2013b]。为了避免这种错误，截断拷贝的数据以限制字符串的字节长度是一种防御方法，但是最好的措施是确保目标缓冲区的大小足以容纳复制数据和'\0'结束符。当字符串存储在堆空间时， 确保分配内存时已分配了足够的空间。

部分字符串处理函数由于设计时安全考虑不足，或者存在一些隐含的目的缓冲区长度要求，容易被误用，导致缓冲区写溢出。此类典型函数包括不在C标准库函数中的itoa()，realpath()函数。

**【错误代码示例】**
如下错误代码示例中，试图将数字转为字符串，但是目标存储空间的长度不足：

int num = ...

char str[8] = {0};

itoa(num, str, 10); /\* 10进制整数的最大存储长度是12个字节 \*/

如下错误代码示例中，试图将路径标准化，但是目标存储空间的长度不足：

char resolvedPath[100] = {0};

realpath(path, resolvedPath); //realpath函数的存储缓冲区长度是由PATH\_MAX常量定义，或是由\_PC\_PATH\_MAX系统值配置的，通常都大于100字节

**【正确代码示例】**
如下正确代码示例中，在对外部数据进行解析并将内容保存到name中，考虑了name的大小:

char \*msg = GetMsg();

...

char name[MAX\_NAME] = {0};

int i=0;

/\* 必须考虑msg不包含预期的字符’\n’ \*/

while (\*msg != '\0' && \*msg != '\n' && i < sizeof(name) - 1) {

 name[i++] = \*msg++;

}

**【错误代码示例】**（差一错误）
如下错误代码示例中，展现了一个差一错误[Dowd 2006]。代码中的循环将数据从src复制到dest。但是因为循环没有考虑'\0'结束符，它可能错误的写入dest缓冲区结束位置之后的一个字节：

#include <stddef.h>

#define ARRAY\_SIZE 32

void Func(void) {

 size\_t i;

 dest[ARRAY\_SIZE];

 src[ARRAY\_SIZE];

 /\* ... \*/

 for (i = 0; src[i] != '\0' && i < ARRAY\_SIZE; ++i) {

 dest[i] = src[i];

 }

 dest[i] = '\0';

}

**【正确代码示例】**（差一错误）
如下正确代码示例中，修改了循环条件，避免差一错误导致的写越界：

#include <stddef.h>

#define ARRAY\_SIZE 32

void Func(void) {

 size\_t i;

 dest[ARRAY\_SIZE];

 src[ARRAY\_SIZE];

 /\* ... \*/

 for (i = 0; src[i] != '\0' && i < ARRAY\_SIZE - 1; ++i) {

 dest[i] = src[i];

 }

 dest[i] = '\0';

}

**【错误代码示例】**（gets函数）
gets函数在C99技术勘误3中被弃用，从C11中删除，它本是不安全的，绝不应该使用，因为它没有提供任何控制从stdin读取缓冲区数据量的办法。
如下错误代码示例中，假设gets函数不会读取超过(BUFFER\_SIZE - 1)个字符，但是这是个无效的假设，可能造成缓冲区溢出。
gets函数从stdin中读取字符写入目标数组，直到遇到文件结束符或换行符结束，并将换行符丢弃，在读入数组的最后一个字符后面写入'\0'结束符。

#include <stdio.h>

#define BUFFER\_SIZE 1024

void Func(void) {

 char buf[BUFFER\_SIZE];

 if (gets(buf) == NULL) {

 /\* 错误处理 \*/

 }

}

**【正确代码示例】**（fgets函数）
如下正确代码示例中，fgets函数从流中读取到数组的字符数最多不超过第二个参数减一，并保证目标字符串有'\0'结束符，因此不会造成缓冲区溢出。

#include <stdio.h>

#include <string.h>

#define BUFFER\_SIZE 32

void Func(void)

{

 char buf[BUFFER\_SIZE];

 int ch;

 if (fgets(buf, sizeof(buf), stdin) != NULL) {

 /\* fgets() 成功; 继续处理换行符 \*/

 } else {

 /\* fgets() 出错，处理错误 \*/

 }

}

fgets函数不是gets函数的严格替代品，应为fgets函数保留换行符（如果读到的话），也可能返回一个不完整的行。使用fgets函数安全地处理太长而无法保存在目标数组中的行是可能的，但是由于性能原因，不建议这么做。

## 规则 对字符串进行存储操作，确保字符串有’\0’结束符

**【描述】**

部分字符串处理函数操作字符串时，将截断超出指定限制的字符串，如strncpy函数最多复制n个字符到目的缓冲区，如果源字符串长度大于n，则不会写入'\0'到目的缓冲区，目的缓冲区的内容为n个被复制的字符。

使用这类函数时，可能会无意的截断会导致数据丢失，并在某些情况下会导致软件漏洞。因此，对字符串进行存储操作，必须确保字符串有’\0’结束符，否则在后续的调用strlen等操作中，可能会导致内存越界访问漏洞。

**【错误代码示例】**
在下面的错误代码示例中，使用strncpy函数复制字符串时可能会发生截断（发生条件为：strlen(name) > sizeof(fileName) - 1）。当发生截断时，fileName的内容是不完整的，并且缺少'\0'结束符，后续对fileName的操作可能会导致软件漏洞：

#include <stdio.h>

void Func(const char \*name)

{

 char fileName[128];

 strncpy(fileName, name, sizeof(fileName) - 1);

 /\* 函数实现其它部分 \*/

}

**【正确代码示例】**

#include <stdio.h>

void Func(const char \*name)

{

 char fileName[128];

strncpy(fileName, name, sizeof(fileName) - 1);

filename[sizeof(filename) - 1] =’\0’;

 /\* 函数实现其它部分 \*/

}

## 规则 外部数据作为数组索引时必须确保在数组大小范围内

**【描述】**

外部数据作为数组索引对内存进行访问时，必须对数据的大小进行严格的校验，否则会导致严重的错误。

**【描述】** 外部数据作为数组索引对内存进行访问时，必须对数据的大小进行严格的校验，否则会导致严重的错误。

**【错误代码示例】**（差一错误）
如下错误代码示例中的InsertInTable函数使用了无效索引，企图在刚刚超出数组结束位置的元素中存储一个数值。
首先，该函数错误地根据缓冲区的大小验证索引pos。当pos初始值等于size时，该函数企图将value存储在刚刚超出数组结束处的内存位置上：

#include <stdlib.h>
#define TABLE\_SIZE 100
static int table[TABLE\_SIZE];
int InsertInTable(size\_t pos, int value)
{
 if (pos > TABLE\_SIZE) {
 /\* 错误处理 \*/
 }
 table[pos] = value;
 return 0;
}

**【正确代码示例】**（差一错误）
在如下正确代码示例中，修改校验pos的条件，避免数组索引越界：

#include <stdlib.h>
#define TABLE\_SIZE 100
static int table[TABLE\_SIZE];
int InsertInTable(size\_t pos, int value)
{
 if (pos >= TABLE\_SIZE) {
 /\* 错误处理 \*/
 }
 table[pos] = value;
 return 0;
}

**【错误代码示例】**（使用程序外部数据作为数组下表）

int Foo(unsigned char \*buffer, size\_t size)

{

 ...

 size\_t offset = ReadIntFromMsg(); /\* 从程序外部的消息中读取数组索引值 \*/

 unsigned char c = buffer[offset]; /\* 使用前未校验数组下标，可能访问无效索引值 \*/

 ...

}

**【正确代码示例】**（使用程序外部数据作为数组下表）

int Foo(unsigned char \*buffer, size\_t size)

{

 ...

 size\_t offset = ReadIntFromMsg(); /\* 从程序外部的消息中读取数组索引值 \*/

 if (offset < size) {

 unsigned char c = buffer[offset];

 ...

 }

 ...

}

## 规则 函数的指针参数如果不是用于修改所指向的对象就应该声明为指向const的指针

**【描述】**

const 指针参数，将限制函数通过该指针修改所指向对象，使代码更牢固、安全。

**【正确代码示例】**

int Foo(const char \*filePath)

{

 ...

 int fd = open(filePath, ...);

 ...

}

## 规则 外部输入作为内存操作相关函数的复制长度时，需要校验其合法性

**【描述】**

将数据复制到容量不足以容纳该数据的内存中会导致缓冲区溢出。为了防止此类错误，必须根据目标容量的大小限制被复制的数据大小，或者必须确保目标容量足够大以容纳要复制的数据。 典型的内存操作相关的函数（例如memcpy*s、memmove*s等），如果复制长度外部可控，则必须校验其合法性，否则容易导致内存溢出。

**【错误代码示例】**

外部输入的数据不一定会直接作为内存复制长度使用，还可能会间接参与内存复制操作。 下例错误代码示例中，asnOcts->octetLen来自设备外部报文，虽然没有直接作为内存复制长度使用，而是作为for循环体的上限使用，间接参与了内存复制操作。由于没有校验其大小，可造成缓冲区溢出：

typedef struct BigIntType
{
    unsigned int length;
    char val[MAX\_INT\_DIGITS];
}BigInt;

BigInt \*AsnOctsToBigInt（const AsnOcts \*asnOcts)
{
    BigInt \*bigNumber = NULL;
    ...
    for (i = 0; i < asnOcts->octetLen; i++) {
        bigNumber->val[i] = asnOcts->octs[i];
    }
    ...
}

**【正确代码示例】**

如下正确代码中，对asnOcts->octetLen做了校验。

typedef struct BigIntType
{
    unsigned int length;
    char val[MAX\_INT\_DIGITS];
}BigInt;

BigInt \*AsnOctsToBigInt(const AsnOcts \*asnOcts)
{
    BigInt \*bigNumber = NULL;
    ...

 // 根据应用场景，对来自外部报文的循环长度asnOcts->octetLen与bigNumber->val数组大小做校验，避免造成缓冲区溢出
 if (asnOcts->octetLen > sizeof(bigNumber->val) {
 return NULL;
 }

    for (i = 0; i < asnOcts->octetLen; i++) {
        bigNumber->val[i] = asnOcts->octs[i];
    }

    ...
}

## 规则 调用格式化输入/输出函数时，禁止format参数由外部可控

**【描述】**

调用格式化函数时，如果format参数由外部可控，会造成字符串格式化漏洞。

攻击者如果能够完全或者部分控制格式字符串内容，可以使被攻击的进程崩溃、查看栈内容、查看内存内容或者在任意内存位置写入数据。结果是，攻击者能够以被攻击进程的权限执行任意代码。

这些格式化函数有： 格式化输出函数: xxxprintf; 格式化输入函数: xxxscanf; 格式化错误消息函数: err(), verr(), errx(), verrx(), warn(), vwarn(), warnx(), vwarnx(), error(), error*at*line(); 格式化日志函数: syslog(), vsyslog().

**【错误代码示例】**

char \*msg = GetMsg();

...

printf(msg);

**【正确代码示例】**

char \*msg = GetMsg();

...

printf("%s\n", msg);

# 整数

## 规则 整数之间运算时必须严格检查，确保不会出现溢出、反转、除0

**【描述】**

在计算机中，整数存储的长度是固定的（例如32位或64位），当整数之间进行运算时，可能会超过这个大固定长 度，导致整数溢出或反转，使得实际计算结果与预期结果不符。

如果涉及到除法或者求余操作，必须确保除数不为0。

**【错误代码示例】**

size\_t width = ReadSize();

size\_t height = ReadSize();

size\_t total = width \* height; //可能整数溢出

void \*bitmaps = malloc(total);

**【正确代码示例】**

size\_t width = ReadSize();

size\_t height = ReadSize();

if (width == 0 || height == 0 || width > MAX\_WIDTH || height > MAX\_HEIGHT) {

 /\* 处理错误 \*/

 ...

}

size\_t total = width \* height; // MAX\_WIDTH \* MAX\_HEIGHT 不会溢出

void \*bitmaps = malloc(total);

**【错误代码示例】**

size\_t a = ReadSize();

size\_t b = 1000 / a; //a可能是0

size\_t c = 1000 % a; //a可能是0

...

**【正确代码示例】**

size\_t a = ReadSize();

if (a == 0) {

 /\* 处理错误 \*/

 ...

}

size\_t b = 1000 / a; //a不可能是0

size\_t c = 1000 % a; //a不可能是0

...

# 内存

## 规则 内存分配后必须判断是否成功

【描述】

内存分配失败后，那么后续的操作存在未定义的行为风险。比如malloc申请失败返回了空指针，对空指针的解引用是一种未定义行为。

**【错误代码示例】**

以下错误代码示例中，调用malloc分配内存之后，没有判断是否成功，直接引用了p。如果malloc失败，它将返回一个空指针并传递给p。当如下代码在内存拷贝中解引用了该空指针p时，程序会出现未定义行为。

/\* inSize是pIn指向内存所占的字节数 \*/
void Foo(const char \*pIn, const size\_t inSize)
{
 // 本函数内，size使用前做合法性检查
 if ((pIn == NULL) || (inSize == 0) || (inSize >= FOO\_MAX\_LEN)) { /\* FOO\_MAX\_LEN被定义为符合程序设计预期的最大内存空间 \*/
 /\* 错误处理 \*/
 }

 ...

 char \*p = (char \*)malloc(inSize); // 这里违反本规则：内存申请之后未判断是否成功。
 memcpy(p, pIn, inSize);
 ...
 free(p); // 这里p可能是NULL，也存在解引用空指针问题。

 p = NULL;

 ...
}

**【正确代码示例】**

下面的正确示例代码中，在malloc分配内存之后，立即判断其是否成功，消除了上述的风险。

/\* inSize是pIn指向内存所占的字节数 \*/
void Foo(const char \*pIn, const size\_t inSize)
{
 // 本函数内，size使用前做合法性检查
 if ((pIn == NULL) || (inSize == 0) || (inSize >= FOO\_MAX\_LEN)) { /\* FOO\_MAX\_LEN被定义为符合程序设计预期的最大内存空间 \*/
 /\* 错误处理 \*/
 }

 ...

 char \*p = (char \*)malloc(inSize);
 if (p == NULL) { /\* 修改：内存申请之后立即判断是否成功。\*/
 /\* 错误处理 \*/
 }

 memcpy\_s(p, pIn, inSize);
 ...
 free(p);

 P = NULL;

 ...
}

## 规则 不要使用realloc()函数

【描述】

realloc()原型如下：

void \*realloc(void \*ptr, size\_t size);

随着参数的不同，其行为也是不同。

1） 当ptr不为NULL，且size不为0时，该函数会重新调整内存大小，并将新的内存指针返回，并保证小的size的内 容不变；

2） 参数ptr为NULL，但size不为0，那么行为等同于malloc(size)；

3） 参数size为0，则realloc的行为等同于free(ptr)。

由此可见，一个简单的C函数，却被赋予了3种行为，这不是一个设计良好的函数。虽然在编码中提供了一些便利 性，但是却极易引发各种bug。

## 规则 不要使用alloca()函数申请栈上内存

【描述】

POSIX和C99均未定义alloca()的行为，在有些平台下不支持该函数，使用alloca会降低程序的兼容性和可移植性，该 函数在栈帧里申请内存，申请的大小很可能超过栈的边界，影响后续的代码执行。

请使用malloc或new，从堆中动态分配内存。

# 不安全函数

## 规则 不要将程序外部可控数据作为system、popen、execl, execlp, execle, execv, execvp等进程启动函数的参数

**【描述】**

这些函数会创建一个新的进程，如果外部可控数据作为这些函数的参数，会导致注入漏洞。

即使参数经过拼接，也可能由于命令分隔符（请参考"附录B 命令注入相关的特殊字符"）机制，导致注入漏洞。

如果需要使用system()、popen()、WinExec()、ShellExecute()，请使用白名单机制校验其参数，确保这些函数的参 数不受任何外来数据的命令注入影响。

**【错误代码示例】**

以下代码从外部获取数据后直接作为system函数的参数，具有注入漏洞。

char \*msg = GetMsgFromRemote(); /\* msg来自网络报文 \*/

system(msg);

**【错误代码示例】**

以下代码从外部获取数据后进行拼接，未考虑到命令分隔符，仍然具有注入漏洞：

char \*msg = GetMsgFromRemote(); /\* msg来自网络报文 \*/

sprintf\_s(cmd, sizeof(cmd), "dir %s", msg);

system(cmd);

建议Linux/Unix下使用exec系列函数来避免命令注入。exec系列函数中的path，file参数禁止使用命令解析器 (如/bin/sh)。

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg,..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

**【错误代码示例】**

例如，不要如下使用方式：

execl("/bin/sh", "sh", "-c", CMD, (VOS\_CHAR \*) 0);

## 规则 禁止在信号处理例程中调用非异步安全函数

**【描述】**

信号处理例程应尽可能简化。在信号处理例程中如果调用非异步安全函数，可能会导致函数的执行不符合预期的结果。 下列代码中的信号处理程序通过调用fprintf()写日志，但该函数不是异步安全函数。

void Handler(int sigNum)

{

 ...

fprintf(stderr, "%s\n", info);

}

## 规则 不要使用setjmp/longjmp

**【描述】**

setjmp/longjmp函数允许C语言跨函数跳转。调用setjmp函数保存当前的执行环境，后续调用longjmp跳转到之前 setjmp位置处。

使用setjmp/longjmp函数使程序变得特别复杂，不易理解，资源不能得到有效清理，并且无法在多线程环境下使用。

**【错误代码示例】**

以下函数中main处调用setjmp，在Foo处调用longjmp后，又直接跳转到了main函数：

jmp\_buf buff;

void Foo()

{

 printf("Foo1\n");

 longjmp(buff, 100);

 printf("Foo2\n");

}

int main()

{

 int a = setjmp(buff);

 printf("a = %d\n", a);

 if (a == 0) {

 Foo();

 }

 else {

 printf("main\n");

 }

 return 0;

}

输出： a = 0 Foo1 a = 100 main

# 文件输入/输出

## 规则 必须对文件路径进行规范化后再使用

【描述】

当文件路径来自外部数据时，需要先将文件路径规范化，如果没有作规范化处理，攻击者就有机会通过恶意构造文件 路径进行文件的越权访问：

例如，攻击者可以构造“../../../etc/passwd”的方式进行任意文件访问。

在linux下，使用realpath函数，在windows下，使用PathCanonicalize函数进行文件路径的规范化。

**【错误代码示例】**

以下代码从外部获取到文件名称，拼接成文件路径后，直接对文件内容进行读取，导致攻击者可以读取到任意文件的 内容：

char \*fileName = GetMsgFromRemote();

...

sprintf\_s(untrustPath, sizeof(untrustPath), "/tmp/%s", fileName);

char \*text = ReadFileContent(untrustPath);

**【正确代码示例】**

正确的做法是，对路径进行规范化后，再判断路径是否是本程序所认为的合法的路径：

char \*fileName = GetMsgFromRemote();

...

sprintf\_s(untrustPath, sizeof(untrustPath), "/tmp/%s", fileName);

char path[PATH\_MAX] = {0};

if (realpath(untrustPath, path) == NULL) {

 //error

 ...

}

if (!IsValidPath(path)) { //检查文件是否的位置是否正确

 //error

 ...

}

char \*text = ReadFileContent(untrustPath);

**【例外】**

运行于控制台的命令行程序，通过控制台手工输入文件路径，可以作为本规则例外。

## 规则 不要在共享目录中创建临时文件

**【描述】**

程序的临时文件应当是程序自身独享的，任何将自身临时文件置于共享目录的做法，将导致其他共享用户获得该程序的额外信息，产生信息泄露。因此，不要在任何共享目录创建仅由程序自身使用的临时文件。

如Linux下的/tmp目录是一个所有用户都可以访问的共享目录，不应在该目录下创建仅由程序自身使用的临时文件。

# 敏感信息处理

## 规则 不要用rand函数产生用于安全用途的伪随机数

【描述】

C语言标准库rand()函数生成的是伪随机数，所以不能保证其产生的随机数序列质量。不要使用rand()函数产生的随机数用于安全用途，必须使用安全的随机数产生方式，如：类Unix平台的/dev/random文件。

**【错误代码示例】**

如下场景：程序员期望生成一个唯一的不可被猜测的ID，但该ID是通过调用rand()函数产生的数字随机数，它的ID是可猜测的，并且随机性有限。

#include <stdio.h>
#include "securec.h"

#define LEN 32

void func(void)
{
 char id[LEN] = {0};
 int r;
 int iRet = -1;

 /\* ... \*/

 r = rand(); /\* 使用了rand产生了一个伪随机数 \*/
 iRet = snprintf(id, LEN, "%d", r); /\* 伪随机数用作ID，可以被猜测到 \*/
 if (iRet == -1) {
 /\* 错误处理 \*/
 }

 /\* ... \*/
}

**【正确代码示例】(POSIX)**

在类Unix平台上，使用/dev/random文件得到随机数。

#include <stdio.h>
include "securec.h"

#define LEN 32

void func(void)
{
 char \*id = (char \*) malloc(LEN);
 if (id == NULL) {
 /\* 错误处理 \*/
 }
 (void)memset(id, 0, LEN);

 FILE \*fp = NULL;
 fp = fopen("/dev/random", "r");
 /\* ... 检查fp，并做错处处理 \*/

 size\_t count = fread(id, 1, LEN, fp);
 /\* ... \*/

 /\* 使用随机数id \*/

 /\* ... \*/
 fclose(fp);
}

## 规则 内存中的敏感信息使用完毕后立即清0

**【描述】**

口令、密钥等敏感信息使用完毕后立即清0，避免被攻击者获取。

**【错误代码示例】**

以下代码获取到密码后，将密码保存到password中，进行密码验证，使用完毕后，通过memset对password清0。由于某些编译器在优化时候不会执行它认为不会改变程序执行结果的代码，因此memset调用可能被编译器优化，清理可能失效。

int Foo()

{

 char password[MAX\_PWD\_LEN] = {0};

 if (!GetPassword(password, sizeof(password))) {

 //错误处理

 ...

 }

 if (!VerifyPassword(password)) {

 //错误处理

 ...

 }

 ...

 memset(password, sizeof(password), 0, sizeof(password));

 ...

}

**【正确代码示例】**

如果编译器支持#pragma指令，那么可以使用该指令指示编译器不作优化。

int Foo()

{

 char password[MAX\_PWD\_LEN] = {0};

 if (!GetPassword(password, sizeof(password))) {

 //错误处理

 ...

 }

 if (!VerifyPassword(password)) {

 //错误处理

 ...

 }

...

#pragma optimize("", off)

memset(password, sizeof(password), 0, sizeof(password));

…

#pragma optimize("", on)

 ...

}

## 规则 不要使用string类存储敏感信息

**【描述】**

string类是C++内部定义的字符串管理类，如果口令等敏感信息通过string进行操作，在程序运行过程中，敏感信息可 能会散落到内存的各个地方，并且无法清0。

以下代码，Foo函数中获取密码，保存到string变量password中，随后传递给VerifyPassword函数，在这个过程中， password实际上在内存中出现了2份。

**【错误代码示例】**

int VerifyPassword(string password)

{

 //...

}

int Foo()

{

 string password = GetPassword();

 VerifyPassword(password);

 ...

}

**【正确代码示例】**

应该使用char或unsigned char保存敏感信息，如下代码：

int VerifyPassword(const char \*password)

{

 /\* ... \*/

}

int Foo()

{

 char password[MAX\_PASSWORD] = {0};

 GetPassword(password, sizeof(password));

 VerifyPassword(password);

 ...

}