中软国际公司内部技术规范

C语言安全编程规范

中软国际科技服务有限公司

修订声明

参考:华为 C&C++语言安全编码规范。

0.

日期	修订版本	修订描述	作者
2017-07-28	1.0	整理创建初稿	陈丽佳
2017-08-02	1.1	统一整理格式，并修正部分错误	陈丽佳

目录

概述

1 前言

随着公司业务发展，越来越多的产品被公众、互联网所熟知，并成为安全研究组织的研究对象、黑客的漏洞挖掘目标，容易引起安全问题。安全问题影响的不只是单个产品，甚至有可能影响到公司整体声誉。产品安全涉及需求、设计、实现、部署多个环节，实现的安全是产品安全的重要一环。为了帮助产品开发团队编写安全的代码，减少甚至规避由于编码错误引入安全风险，特制定本规范。

《C&C++语言安全编程规范》参考业界安全编码的研究成果，并结合产品编码实践的经验总结，针对 C/C++语言编程中的字符串操作、整数操作、内存管理、文件操作、内核操作、STL 库使用等方面，描述可能导致安全漏洞或潜在风险的常见错误。以期减少缓冲区溢出、整数溢出、格式化字符串攻击、命令注入攻击、目录遍历等典型安全问题。

2 使用对象

本规范的读者及使用对象主要为使用 C 语言的开发人员、测试人员等。

3 适用范围

本规范适合于公司基于 C 语言开发的产品。

4 术语定义

规则：编程时必须遵守的约定。

建议：编程时必须加以考虑的约定。

说明：对此规则/建议进行必要的解释。

错误示例：对此规则/建议从反面给出例子。

推荐做法：对此规则/建议从正面给出例子。

1 通用规则

规则 1：对外部输入进行校验

说明：软件最为普遍的缺陷就是对来自客户端或者外部环境的数据没有进行正确的合法性校验。这种缺陷可以导致几乎所有的程序弱点，例如 Dos、内存越界、命令注入、SQL 注入、缓冲区溢出、数据破坏、文件系统攻击等。这些不可信数据可能来自：

- 用户输入
- 外部调用的参数
- 进程间的通信数据
- 网络连接（甚至是一个安全的连接）
- 用户态输入（对于内核程序）
- 上层应用（业务）输入

当这些不可信输入用于如下场景时（包括但不局限于），需要校验其合法性：

- 作为循环条件

  将不可信数据作为循环限定条件，可能会引发缓冲区溢出、内存越界读/写、死循环等问题。

- 作为数组下标

  将不可信的数据作为数数组下标，可能导致超出数组上限，从而造成非法内存访问。

- 作为内存分配的尺寸参数

  请参考规则 C3.1、规则 C3.2 和规则 C4.5。

- 作为业务数据

  如作为命令执行参数、拼装 sql 语句、拼接格式化字符串等，这会导致命令注入、SQL 注入、格式化漏洞等问题。详细请参考规则 C2.1、C5.2 和 C6.3。

- 用于数据拷贝操作

  当作为拷贝长度时，极易造成目标缓冲区溢出。详细请参考规则 C1.1、C1.2 和 C1.3。

- 影响代码逻辑

  比如基于不可信输入做安全决策，影响代码逻辑走向。

- 会改变系统状

  比如未加校验直接打开不可信路径，可能会导致目录遍历攻击，操作了攻击者无权操作的文件，使得系统被攻击者所控制。

输入校验可能包括如下内容（包括但不局限于）：

- 校验数据长度
- 校验数据范围
- 校验数据类型和格式

- 校验输入只包含可接受的字符（可以采用“白名单”形式），尤其需要注意一些特殊情况下的特殊字符。了解更多关于特殊字符，可以参考附录 A 和附录 B。

规则 2：禁止在日志中保存口令、密钥

说明：在日志中不能保存口令和密钥，其中的口令包括明文口令和密文口令。对于敏感信息建议采取以下方法，

- 不打印在日志中；
- 若因为特殊原因必须要打印日志，则用“*”代替（不要显示出敏感信息的长度）。

规则 3：及时清除存储在可复用资源中的敏感信息

说明：存储在可复用资源中的敏感信息如果没有正确的清除则很有可能被低权限用户或者攻击者所获取和利用。因此敏感信息在可复用资源中保存应该遵循存储时间最短原则。可复用资源包括以下几个方面：

- 堆（heap）
- 栈（stack）
- 数据段（data segment）
- 数据库的映射缓存

存储口令、密钥的变量（包括加密后的变量）使用完后必须显式覆盖或清空。

规则 4：正确使用经过验证的安全的标准加密算法

说明：禁用私有算法或者弱加密算法（如 DES，SHA1 等），应该使用经过验证的、安全的、公开的加密算法。

加密算法分为对称加密算法和非对称加密算法。推荐使用的常用对称加密算法有：

- AES

推荐使用的常用非对称算法有：

- RSA

推荐使用的数字签名算法有：

- 数字签名算法（DSA）
- ECDSA

此外还有验证消息完整性的安全哈希算法（SHA256）等。基于哈希算法的口令安全存储必须加入盐值（参见规则 5：基于哈希算法的口令安全存储必须加入盐值）。

密钥长度符合最低安全要求：

- AES： 128 位

- RSA： 2048 位
- DSA： 2048 位

规则 5：基于哈希算法的口令安全存储必须加入盐值（salt）

说明：单向哈希是在一个方向上工作的哈希函数，从预映射的值很容易计算其哈希值，但要根据特定哈希值产生一个预映射的值却是非常困难的。单向哈希主要应用于加密、消息完整性校验、冗余校验等。假如没有加入盐值，则加密原理是：

密文= 哈希算法（明文）

此时，若攻击者获取到密文，同时知道哈希算法，则就可以通过字典攻击来探测和获取口令。加入盐值之后：

密文= 哈希算法（明文+盐值）

其中盐值可以随机设置，这样即使相同的口令，但盐值不同，密文也不同，从而增加了口令的破解难度、增强安全性。

规则 6：不要硬编码敏感信息

说明：硬编码口令、服务器 IP 地址以及加密密钥等敏感信息可能会将这些信息暴露给攻击者。任何人都可以反编译并发现这些敏感信息。因此，除了一些特殊情况（例如在 TPM 环境下）之外，程序中禁止硬编码任何敏感信息。

硬编码敏感信息还会增加维护管理成本，当修改代码时，需要额外管理并适配这些修改。例如，要更改一个已经部署了的程序的硬编码口令，可能需要下发一个补丁。

规则 7：不要在共享目录中创建临时文件

说明：程序员经常会在共享目录里创建临时文件。临时文件通常作为不需要或者不能驻留在内存中的数据的一种辅助存储方式，同时也可以作为与其它进程通过文件系统进行通信的一种方式。例如，一个进程会以一个公认的命名或者与合作进程协商好的名字在共享目录里创建临时文件，然后这些临时文件便可以在这些合作进程间共享信息。

但是，这是一个非常危险的操作。一个在共享目录里大家都知道名字的文件是很容被攻击者控制和操纵的。以下列出了几种可能的规避方法：

- 使用其它低级别进程间通信（IPC）机制，如使用sockes或者共享内存；
- 使用高级别IPC机制，如远程过程调用（remote procedure call）；
- 使用一个安全的目录或者设置一个只能被程序应用实例访问的jail（确保同一平台下的多个应用程序实例不会产生竞争）。

IPC 机制中有些需要使用临时文件，但是其它的不需要。例如，需要使用临时文件的 IPC 机制

有 POSIX 的 *mmap()*函数。而伯克利套接字（Berkeley Sockets）、POSIX 本地 IPC 套接字和 System V 共享内存却不需要临时文件。因为共享目录的多用户属性使得它具有与生俱来的危险，因此，利用共享临时文件来实现 IPC 是不推荐的。

当 2 个以上或者一组用户对目录具有写权限时，其危险和欺骗性比少量文件的共享访问更为严重。因此，当确实需要在共享目录中创建临时文件时，必须满足如下条件：

- 创建不可预测的文件名称；
- 创建唯一的文件名称；
- 原子打开；
- 独占打开；
- 使用合适的权限打开；
- 程序退出前必须删除。

规则 8：遵循最小权限原则

说明：程序在运行时可能需要不同的权限，但对于某一种权限不需要始终保留。例如，一个网络程序可能需要超级用户权限来捕获原始网络数据包，但是在执行数据报分析等其它任务时，则可能不需要相同的权限。因此程序在运行时只分配能完成其任务的最小权限。过高的权限可能会被攻击者利用并进行进一步的攻击。因此，权限在使用完毕后应该及时撤销。

在撤销权限时，应该尤其注意以下两点：

（1）撤销权限时应遵循正确的撤销顺序；

（2）完成权限撤销操作后，应确保权限撤销成功。

2 字符串操作安全

规则 C2.1：确保有足够的空间存储字符串的字符数据和'\0'结束符

说明：在分配内存或者在执行字符串复制操作时，除了要保证足够的空间可以容纳字符数据，还要预留'\0'结束符的空间，否则会造成缓冲区溢出。

错误示例 1：拷贝字符串时，源字符串长度可能大于目标数组空间。

```
enum { BUFFER_SIZE = 128 };
int main(int argc, char *argv[])
{
    char dst[BUFFER_SIZE] = {0x00};
    if (argc > 1)
    {
        strcpy(dst, argv[1]);  /*【错误】当源字符串长度大于目标数组 dst 时，会发生缓冲区溢出 */
    }
    /*...*/
    return 0;
}
```

推荐做法 1：根据源字符串长度来为目标字符串分配空间。

```
int main(int argc, char *argv[])
{
   char *dst = NULL;
   size_t length = strnlen(argv[1], MAX_LEN);
    if (argc > 1)
    {
        dst = (char *)malloc(length + 1); /* 【修改】 确保目标字符串可以存储源数据 */
        if(dst != NULL)
        {
            int ret = strcpy_s(dst, length + 1, argv[1]); /* 【修改】使用安全函数 strcpy_s 代替 strcpy 来拷贝字符串 */
            /* 校验 ret，确保安全函数执行成功 */
        }
    }
    /* 使用完后释放内存 */
    return 0;
}
```

错误示例 2：典型的差一错误。for 循环将 src 中的数据拷贝到 dst 中，然而未考虑'\0'结束符写入数组的位置，经过循环后，'\0'会写越界，超出数组 dst 一个字节，从而造成缓冲区溢出和内存改写。

```
enum { ARRAY_SIZE = 32 };
```

```
void func(void)
{
    char dst[ARRAY_SIZE + 1] = {0x00};
    char src[ARRAY_SIZE + 1] = {0x00};
    size_t i = 0;
    for (i = 0; src[i]!= '\0' && (i < (ARRAY_SIZE + 1)); ++i) /* 【错误】 结束符会写越界 dst 一个字节*/
    {
        dst[i] = src[i];
    }
    dst[i] = '\0'; /* 【错误】 结束符写越界*/
}
```

推荐做法 2：在赋值循环语句中考虑结束符的添加。

```
enum { ARRAY_SIZE = 32 };
void func(void)
{
    char dst[ARRAY_SIZE + 1] = {0x00};
    char src[ARRAY_SIZE + 1] = {0x00};
    size_t i = 0;
    for (i = 0; src[i] != '\0' && (i < ARRAY_SIZE); ++i) /* 【修改】 考虑结束符的写入位置，不会越界 */
    {
        dest[i] = src[i];
    }
    dst[i] = '\0';
}
```

错误示例 3：在下面的例子中，name 是可能来自用户输入、文件系统或者网络的字符串变量。代码中通过构造的一个文件名称来打开文件。然而，由于 sprintf()函数没有对输入的数据进行校验，因此当 name 是一个非常大的字符串变量时，就会产生缓冲区溢出。

```
void func(const char *name)
{
    char filename[NAME_SIZE + 1] = {0x00};
    sprintf(filename, "%s.txt", name); /* 【错误】 当 name 的长度超过目标数组 filename 大小时，会发生缓冲区溢出 */
}
```

推荐做法 3：一个比较好的方法就是使用安全版本函数 sprintf_s()。sprintf_s()会对入参进行校验，保证不会发生缓冲区溢出。

```
void func(const char *name)
{
    char filename[NAME_SIZE + 1] = {0x00};
    int ret = sprintf_s(filename, sizeof(filename), "%s.txt", name); /* 【修改】 使用 sprintf_s 来避免缓冲区溢出 */
    /* 校验 ret，确保安全函数执行成功 */
```

```
}
```

规则 C2.2：字符串操作过程中确保字符串有'\0'结束符

说明：字符串结束与否是以'\0'作为标志的。没有正确地使用'\0'结束字符串可能导致字符串操作时发生缓冲区溢出。因此对于字符串或字符数组的定义、设置、复制等操作，要给'\0'预留空间，并保证字符串有'\0'结束符。

错误示例：下列代码中，在调用 strncpy()之前 ntca 就被赋予了结束符。然而，接下来执行的 strncpy()可能会将这个结束符覆写。

```
...
char ntca[NTCA_SIZE + 1];
ntca[sizeof(ntca) - 1] = '\0';
strncpy(ntca, source, sizeof(ntca)); /*【错误】 strncpy()不能保证字符串结尾含有结束符，因此可能将已经赋予的结束符覆写 */
...
```

推荐做法：正确的方法是依照程序员的目的而来的。如果程序员想要截断一个字符串并且保证目标字符串结尾含有结束符，那么可以使用如下方法。

```
...
char ntca[NTCA_SIZE + 1];
int ret = strncpy_s(ntca, NTCA_SIZE + 1, source, NTCA_SIZE); /*【修改】 使用 strncpy_s()来代替 strncpy()，可以保证结尾含有结束符 */
/* 校验 ret，确保安全函数执行成功 */
...
```

规则 C2.3：把数据复制到固定长度的内存前必须检查边界

说明：将未知长度的数据复制到固定长度的内存空间可能会造成缓冲区溢出，因此在进行复制之前应首先获取并检查数据长度，并且在任何情况下都要在明确目标缓冲区的长度之后再进行复制操作。

错误示例：输入消息长度不可预测，不加检查的复制会造成缓冲区溢出。

```
void  MsgCopy()
{
    char dst[MAX_SIZE + 1] = {0x00};
    char *temp = getInputMsg();
    if(temp != NULL)
    {
        strcpy(dst,temp); /*【错误】当 temp 长度大于 dst 大小时，会产生缓冲区溢出 */
    }
}
```

推荐做法：

```
void  MsgCopy()
{
    char dst[MAX_SIZE + 1] = {0x00};
    char *temp = getInputMsg();
    size_t len = strlen(temp);
    if(len > MAX_SIZE || NULL == temp) /* 【修改】 校验 temp 的长度是合法的 */
    {
        return;
    }
    int ret = strcpy_s(dst, sizeof(dst), temp); /* 【修改】使用安全版本函数 strcpy_s()
来代替 strcpy()来进行操作 */
    /* 校验 ret，确保安全函数执行成功 */
}
```

规则 C2.4：避免字符串/内存操作函数的源指针和目标指针指向内存重叠区

说明：内存重叠区是指一段确定大小及地址的内存区，该内存区被多个地址指针指向或引用，这些指针介于首地址和尾地址之间。

使用函数拷贝内存重叠的对象可能导致未定义的行为，可被用来破坏数据的完整性。

错误示例 1：sprintf_s()函数不当使用

```
void LogMessageItem(int error_type , char * error_msg)
{
    size_t msg_length = strnlen(error_msg, MAX_LEN);
    int ret =  sprintf_s(error_msg, msg_length, "%d:%s",error_type,error_msg);
/* 【错误】 err_msg 变量既是源缓冲区，又是目标缓冲区 */
    /* 校验 ret，确保安全函数执行成功 */
    Log(error_msg);
    ...
}
```

推荐做法 1：使用不同的源和目标缓冲区来实现复制功能。

```
void LogMessageItem(int error_type, char * error_msg)
{
    char tmp_msg[MAX_MESSAGE_SIZE + 1] = {0x00};
    int ret = sprintf_s(tmp_msg, sizeof(tmp_msg), "%d:%s",error_type,error_msg);
    /* 【修改】分配另一块内存来避免内存重叠 */
    /* 校验 ret，确保安全函数执行成功 */
    Log(tmp_msg);
    ...
}
```

错误示例 2：

```
...
unsigned char *p1 = GetCurrentMessage();
unsighed char *p2 = p1 + KEY_FIELD_OFFSET; /* 【错误】 p1 和 p2 存在重叠 */
int ret = memcpy_s(p2, MAX_SIZE, p1,KEY_FIELD_SIZE); /* 【错误】当 p1 和 p2 存在重叠时，memcpy_s()不能实现其功能 */
/* 校验 ret，确保安全函数执行成功 */
...
```

推荐做法 2：使用 memmove_s 函数，源字符串和目标字符串所指内存区域可以重叠，但复制后目标字符串内容会被更改，该函数将返回指向目标字符串的指针。

```
...
unsigned char * p1 = GetCurrentMessage();
unsigned char * p2 = p1 + KEY_FIELD_OFFSET;
int ret = memmove_s(p2, MAX_SIZE, p1, KEY_FIELD_SIZE); /* 【修改】使用 memmove_s 来代替 memcpy_s,来避免内存重叠带来的问题 */
/* 校验 ret，确保安全函数执行成功 */
...
```

memcpy_s 与 memmove_s 的目的都是将 N 个字节的源内存地址的内容拷贝到目标内存地址中。但当源内存和目标内存存在重叠时，memcpy_s 不会实现其功能，而 memmove_s 能正确地实施拷贝，但这也增加了一点点开销。

3 格式化输出安全

规则 C3.1：禁止以用户输入来构造格式化字符串

说明：调用格式化函数时，不要直接或者间接将用户输入作为格式化字符串的一部分或者全部。如果攻击者对一个格式化字符串可以部分或完全控制，将导致进程崩溃、查看栈的内容、改写内存、甚至执行任意代码等风险。

这些格式化函数有：

- 格式化输出函数：printf()，fprintf()，sprintf()，snprintf()，vprintf()，vfprintf()，vsprintf()，vsnprintf()，asprintf()（GNU 扩展函数），vasprintf()（GNU 扩展函数）及相应宽字节版本；
- 格式化输入函数：scanf()，fscanf()，sscanf()，vscanf()，vsscanf()，vfscanf()及相应宽字节版本；
- 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()。

备注：以上部分函数是禁用的，详细请参考规则 C5.1。

错误示例：代码中 incorrect_password_show()函数的功能是当某个用户的用户名没有找到或者口令不正确时，展示一条错误消息。这个函数接受来自用户的参数 user，而这个用户是未经过认证的，也就是不安全的外部输入。函数将 user 构造成一个告警展示信息，然后通过 fprintf()将该信息打印到 stderr 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void incorrect_password_show(const char *user)
{
    int ret = 0;
    /* 用户名称被限定在 256 个字节以内 */
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    if(len > MAX_LEN)
    {
        /* 处理错误 */
    }
```

```
    char *message = (char *)malloc(len);
    if (NULL == message)
    {
        /* 处理错误*/
    }
    ret = sprintf_s(message, len, msg_format, user);
    if (ret < 0)
    {
        /* 处理错误 */
    }
    else if (ret >= len)
    {
        /* 处理截断输出 */
    }
    fprintf(stderr, message); /* 【错误】格式化错误产生 */
    free(msg);
}
```

示例代码中首先计算了消息的长度，然后分配内存，接着利用 sprintf_s()函数构造消息内容。因此消息内容中包含了 msg_format 的内容和用户的内容，当用户输入大量的格式符（如%s，%p 等）后，fprintf()在执行时，会将 msg 作为一个格式化字符串来进行解析，而不是消息内容，也就是说 message 不会被打印到 stderr 中，反而会将一些未知的数据打印其中，会引发程序崩溃等未定义的行为。这是一个非常严重的格式化漏洞。

推荐做法 1：下列代码使用 fputs()来代替 fprintf()函数，fputs()会直接将 msg 的内容输出到 stderr 中，而不会去解析它。

```
void incorrect_password_show(const char *user)
{
    int ret = 0;
   /*用户名称被限定在 256 个字节以内*/
    static const char msg_format[] = "%s cannot be authenticated.\n";
    size_t len = strlen(user) + sizeof(msg_format);
    if(len > MAX_LEN)
    {
        /* 处理错误 */
    }
    char *message = (char *)malloc(len);
    if (NULL == message)
    {
        /* 处理错误 */
    }
    ret = sprintf_s(message, len, msg_format, user);
    if (ret < 0)
```

```
    {
        /* 处理错误 */
    }
    else if (ret >= len)
    {
        /* 处理截断错误 */
    }
    if (fputs(message, stderr) == EOF) /* 【修改】使用 fputs()函数代替 fprintf() */
    {
        /* 处理错误 */
    }
    free(message);
}
```

推荐做法 2：通过格式说明符“%s”将 user 以字符串的形式固定下来然后输出到 stderr 中。

```
void incorrect_password(const char *user)
{
    static const char msg_format[] = "%s cannot be authenticated.\n"; /* 【修改】使用“%s”来限定 msg 的格式，使 msg 不会被解析  */
    fprintf(stderr, msg_format, user);
}
```

4 整数安全

规则 C4.1：确保无符号整数运算时不会出现反转

说明：反转是指无法用无符号整数表示的运算结果将会根据该类型可以表示的最大值加 1 执行求模操作。以下表格说明了哪些操作符可能引起反转：

操作符	是否反转
+	是
-	是
*	是
++	是
--	是
+=	是
-=	是
*=	是
<<=	是
<<	是
un -	是

来自于系统外部或其它不可信数据参与到上述运算中的情形，只要将运算结果用于以下之一（包括但不局限于）的用途，都应该添加校验以防止反转：

- 作为数组索引
- 指针运算
- 作为对象的长度或者大小
- 作为数组的边界
- 作为内存分配函数的实参
- 作为循环终止判定条件
- 作为拷贝长度

在上述校验场景中，若代码执行前能确定运算结果不会反转的，可以不作校验，如：

- 两个静态常量（compile-time constants）操作；
- 与 0 进行运算（除数不能为 0）；
- 任意类型的最大值减法（如 UINT_MAX 减去任意无符号数都是安全的）；
- 任何变量乘 1 操作；
- 除法或者取余操作中，只要保证除数不为 0；
- 右移运算时，右移位数不超过该无符号整数类型的精度，如 UNIT_MAX >> x 中，只要 0 <= x < 32 就是合法的（假设 unsigned int 类型的精度是 32 位）。

错误示例：下列代码可能导致相加操作产生无符号数反转现象。

```
static int handlehdr_odc(aim_session_t * sess , aim_...)
{
    ...
    unsigned int payloadlength = aimbs_get32(bs);
    /* payloadlength is read from an untrusted source*/
    ...
    if(!(msg = calloc(1,payloadlength + 1)))  /*【错误】payloadlength + 1 未校验，可能反转为 0，导致内存申请参数为 0 */
    /* ...> potential overflow */
    {
        ...
    }
    while(payloadlength - recvd)
    {
        if(payloadlength - recvd >= 1024)
        {
            i = aim_recv(conn->fd,&msg[recvd],1024); /*msg 申请内存为 0，导致读取数据消息失败 */
        }
        else
          ...
    }
}
```

推荐做法：在运算之前添加校验，确保不会产生无符号数反转。

```
static int handlehdr_odc(aim_session_t * sess , aim_...)
{
    ...
    unsigned int payloadlength = aimbs_get32(bs);
    /* payloadlength is read from an untrusted source*/
    if(payloadlength == 0 || payloadlength > (MAX_SIZE - 1)) /*【修改】确保 payloadlength 合法 */
    {
        /* 错误处理 */
    }
    ...
    if(!(msg = calloc(1,payloadlength + 1)))
    /* ...> potential overflow */
    {
        ...
    }
```

```
    while(payloadlength - recvd)
    {
        if(payloadlength - recvd >= 1024)
        {
            i = aim_recv(conn->fd,&msg[recvd],1024);
        }
        else
          ...
    }
}
```

规则 C4.2：确保有符号整数运算时不会出现溢出

说明：整数溢出是是一种未定义的行为，意味着编译器在处理有符号整数溢出时具有很多选择。以下表格说明了哪些操作符可能引起整数溢出：

操作符	是否溢出	操作符	是否溢出
+	是	-=	是
-	是	*=	是
*	是	/=	是
/	是	%=	是
%	是	<<=	是
++	是	<<	是
--	是	un -	是
+=	是		

来自于系统外部或其它不可信数据参与到上述运算中的情形，只要将运算结果用于以下之一（包括但不局限于）的用途，都应该添加校验以防止溢出：

- 作为数组索引
- 指针运算
- 作为对象的长度或者大小
- 作为数组的边界
- 作为内存分配函数的实参
- 作为循环终止判定条件
- 作为拷贝长度

错误示例：下列代码中两个有符号整数相加可能会产生溢出。

```
static int handlehdr_odc(aim_session_t * sess , aim_...)
{
    ...
    char payloadlength = aimbs_get32(bs);
    /* payloadlength is read from an untrusted source*/
    ...
```

```
    if(!(msg = calloc(1,payloadlength + 1)))  /*【错误】payloadlength + 1 未校验，当 payloadlength 为 127 时，则溢出为-128，calloc()函数会将其转为非常大的正整数，可导致内存申请失败  */
    {
        ...
    }
    while(payloadlength - recvd)
    {
        if(payloadlength - recvd >= 1024)
        {
            i = aim_recv(conn->fd,&msg[recvd],1024);
        }
        else
          ...
    }
}
```

推荐做法：在运算之前添加校验，确保不会产生有符号溢出。

```
static int handlehdr_odc(aim_session_t * sess , aim_...)
{
    ...
    char payloadlength = aimbs_get32(bs);
    /* payloadlength is read from an untrusted source*/
    if(payloadlength <= 0 || payloadlength > (CHAR_MAX - 1)) /*【修改】确保 payloadlength 合法，其中 CHAR_MAX = 127 */
    {
        /* 错误处理 */
    }
    ...
    if(!(msg = calloc(1,payloadlength + 1)))
    {
        ...
    }
    while(payloadlength - recvd)
    {
        if(payloadlength - recvd >= 1024)
        {
            i = aim_recv(conn->fd,&msg[recvd],1024);
        }
        else
          ...
    }
}
```

规则 C4.3：确保整型转换时不会出现截断错误

说明：将一个较大整型转换为较小整型，并且该数的原值超出较小类型的表示范围，就会发生截断错误，原值的低位被保留而高位被丢弃。截断错误会引起数据丢失，甚至可能引发安全问题。特别是将运算结果用于以下用途：

- 作为数组索引
- 指针运算
- 作为对象的长度或者大小
- 作为数组的边界（如作为循环计数器）

错误示例：数据类型强制转化导致数据被截断。

```
void func(void)
{
    signed long int s_a = LONG_MAX;
    signed char sc = (signed char)s_a;  /* 【错误】不同类型强制转化会发生数据截断 */
    /* ... */
}
```

推荐做法：当不同数据类型强制转化时需要首先校验数据的范围，以确定是否会发生数据的丢失。

```
void func(void)
{
    signed long int s_a = LONG_MAX;
    signed char sc;
    if ((s_a < SCHAR_MIN) || (s_a > SCHAR_MAX))  /* 【修改】进行校验以确保在进行类型转化时不会产生截断 */
    {
        /* 处理错误 */
    }
    else
    {
        sc = (signed char)s_a; /* Use cast to eliminate warning */
    }
    /* ... */
}
```

规则 C4.4：确保有符号数和无符号数之间的转换符合预期

说明：有符号数和无符号数之间的转换包括：有符号数到无符号数的转换和无符号数到有符号数的转换。将转换结果用于敏感用途（如作为数组索引、指针运算、对象的长度或大小、数组边界、

内存分配函数实参等）时，一定要确保转换结果在自己的预期内，否则极易引发安全问题：

- 有符号数转换成无符号数：若有符号数为一个负数，那么转成无符号数时，将会是一个非常大的正整数。
- 无符号数转成有符号数：若无符号数为一个较大的数，那么转成有符号数时，可能会转换成一个负数。

错误示例 1：使用有符号数，且未完整校验就作为内存申请函数实参。

```
...
DataPacket *dataPacket = NULL;
int numHeaders = 0; /* numHeader 定义为有符号数 */
PacketHeader *headers = NULL;
...
sock = AcceptSocketConnection();
ReadPacket(dataPacket, sock);
numHeaders = dataPacket ->headers;
if (numHeaders > 100)  /* 只校验 numHeader 的上限值，未校验其下限值 */
{
    ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader)); /* 【错误】 当 numberHeader 负数时，malloc 申请的内存可能过大*/
ParsePacketHeaders(dataPacket, headers);
...
```

上述代码中，numHeaders 被定义为 signed int 类型，且来自外部输入，校验时只校验了上限值，却未校验下限值，当其为负数时，便会产生问题。比如，若 numHeaders 为-1，sizeof(PacketHeader)为 10，那么二者的运算为-10，当其作为 malloc()入参时，malloc()会将其转化为一个非常大的无符号数（即 4294967286），这样的入参，可能会让 malloc()执行失败，导致程序崩溃。也可能因此申请过大内存，导致资源耗尽。

推荐做法 1：添加校验，确保 numHeaders 不为负数，且 malloc()入参符合预期。

```
...
DataPacket * dataPacket = NULL;
int numHeaders = 0; /* numHeader 定义为有符号数 */
PacketHeader *headers = NULL;
...
sock = AcceptSocketConnection();
ReadPacket(dataPacket, sock);
numHeaders = dataPacket ->headers;
if (numHeaders > 100 || numHeaders < 0)  /* 【修改】添加校验，确保 numHeaders 不为负 */
{
    ExitError("too many headers!");
}
```

```
unsigned int mallocSize = numHeaders * sizeof(PacketHeader);
if (mallocSize == 0|| mallocSize >= MAX_SIZE) /* 【修改】添加 malloc 入参校验，确保不为 0，不超过最大申请值，即结果符合预期 */
{
    ExitError("malloc size error");
    return;
}
headers = malloc(numHeaders * sizeof(PacketHeader));
ParsePacketHeaders(dataPacket, headers);
...
```

错误示例 2：使用有符号数，且未完整校验即作为拷贝长度，导致缓冲区溢出。

```
void main (int argc, char **argv)
{
    char path[256] = {0x0};
    char *input = NULL;
    int length = 0;
    Length = GetUntrustedInt(); /* Length 来自外部不可信输入，可能为负  */
    if (length > 256)  /* 只校验上限，未校验下限值 */
    {
        DiePainfully("go away!\n");
        return;
    }
    input = GetUserInput("Enter pathname:");
    strncpy(path, input, length); /* 【错误】若 length 为负数，那么此处会被转换成一个非常大的正整数，导致 path 缓冲区溢出 */
    path[255] = '\0'; /* 添加结束符 */
    printf("Path is: %s\n", path);
}
```

推荐做法 2：添加校验，确保拷贝长度不为负数。

```
void main (int argc, char **argv)
{
    char path[256] = {0x0};
    char *input = NULL;
    int length = 0;
    Length = GetUntrustedInt(); /* Length 来自外部不可信输入，可能为负  */
    if (Length > 256 || length <= 0)  /* 【修改】添加下限校验，确保不为负数 */
    {
        DiePainfully("go away!\n");
        return;
    }
    input = GetUserInput("Enter pathname:");
    strncpy(path, input, length);
    path[255] = '\0'; /* 添加结束符 */
    printf("Path is: %s\n", path);
```

```
}
```

错误示例 3：无符号数隐式转换为有符号数，未经充分校验，导致非法内存访问。

```
char InitialChar[INITIAL_CHAR_SIZE] = {0x0};
/* 初始化数组 InitialChar*/
...
int findElement(unsigned short elementIndex)
{
    short index = 0;
    char tmpChar;
    ...
    if(elementIndex >= INITIAL_CHAR_SIZE) /* 【错误】校验输入下标，确保在数组范围内。但是未考虑到 index 小于 INITIAL_CHAR_SIZE 且足够大的情况，可能导致 index 为负，导致内存非法访问  */
    {
        return;
    }
    index = elementIndex;
    ...
    tmpChar = InitialChar[index]; /* 若 index 为负，则会造成内存非法访问 */
    ...
}
```

上述代码中，elementIndex 为无符号数，只校验了是否满足数组上限值的要求，但是并未考虑该值足够大，同时小于数组上限值。比如，若 elementIndex = 65533，而 INITIAL_CHAR_SIZE 为 66000，那么 elementIndex 便会绕过校验，直接对 index 进行赋值，隐式的转为有符号数，此时 index 就是-3，而引用作为数组的下标，造成内存的非法访问。

推荐做法 3：可以有多种修正方法，可以将 index 设置为 unsigned short 类型。也可以添加校验。

```
char InitialChar[INITIAL_CHAR_SIZE] = {0x0};
/* 初始化数组 InitialChar*/
...
int findElement(unsigned short elementIndex)
{
    short index = 0;
    char tmpChar;
    ...
    index = elementIndex; /* 赋值 */
    if(index >= INITIAL_CHAR_SIZE || index < 0) /* 【修改】校验输入下标，确保在数组范围内，且确保不为负  */
    {
        return 0;
    }
    ...
    tmpChar = InitialChar[index];
    ...
```

```
}
```

规则 C4.5：把整型表达式比较或赋值为一种更大类型之前必须用这种更大类型对它进行求值

说明：若一个整型表达式与一个很大长度的整数类型进行比较或者赋值为这种类型的变量，需要对该整型表达式的其中一个操作数类型显示转换为更大长度的整数类型，用这种更大的进行求值。这里所说的更大整数类型是相对整型表达式的操作数类型而言，譬如整型表达式的操作数类型是 unsigned int ,则该规则所说的更大类型是指 unsigned long long。

错误示例：下列代码为了防止无符号数反转，特意使用一个 unsigned long long 类型的变量 alloc 来存储 cBlocks * 16 得到的数据。

```
void *AllocBlocks(size_t cBlocks)

{

    if (0 == cBlocks)

    {

        return NULL;

    }

    unsigned long long alloc = cBlocks * 16;   /* 【错误】alloc < UINT_MAX 会永远成立 */

    return (alloc < UINT_MAX) ? malloc(cBlocks * 16) : NULL;

}
/*...申请的内存使用后 free...*/
```

这段代码中包含 2 个错误。首先代码认为 unsigned long long 数据类型至少应该比 size_t 类型多 4 个字节。其次假设 size_t 代表一个 32 位数，而 unsigned long long 代表一个 64 位数值，那么 2 个 32 位数相乘，得到的结果仍然是一个 32 位数，因此最后一行代码 alloc < UINT_MAX 是永远成立的，判断语句无效。

推荐做法：将 cBlocks 提型至 unsigned long long。

```
static_assert(
  CHAR_BIT *sizeof(unsigned long long) >=
  CHAR_BIT *sizeof(size_t) + 4,
  "Unable to detect wrapping after multiplication"
);
void *AllocBlocks(size_t cBlocks)
{
    if (0 == cBlocks)
    {
        return NULL;
    }
    unsigned long long alloc = (unsigned long long)cBlocks * 16; /*【修改】将 cBlocks 提型至 unsigned long long 类型 */
    return (alloc < UINT_MAX) ? malloc(cBlocks * 16) : NULL;
}
/*...申请的内存使用后 free...*/
```

需要注意的是，在该代码中只有当 unsigned long long 数据类型比 size_t 类型大于至少 4 个字节时，才能有效防止无符号反转。

建议 C4.1：避免对有符号整数进行位操作符运算

说明：位操作符（～、>>、<<、&、^、|）应该只用于无符号整型操作数，因为有符号整数上的有些位操作的结果是由编译器所决定的，可能会出现出乎意料的行为或编译器定义的行为。

错误示例：对有符号数作位操作运算。

```
enum { BUFFER_SIZE = 4 };
int rc = 0;
int stringify = 0x80000000;
char buf[BUFFER_SIZE] = {0x00};
rc = sprintf_s(buf, sizeof(buf), "%u", stringify >> 24); /* 【不推荐】避免使用有符号数作位操作符运算 */
if (rc == -1 || rc >= sizeof(buf))
{
    /* 处理错误 */
}
```

代码中，stringify >> 24 得到的结果为 0xFFFFFF80 或者写作 4294967168。当转换成字符串时，值 4294967168 是非常巨大的，是不能全部存储到 buf 中去的，因此会被 sprintf_s() 函数将数据截断。

推荐做法：下列代码将 stringify 声明为无符号数据，从而就避免了数据过大带来的截断问题。

```
enum { BUFFER_SIZE = 4 };
```

```
int rc = 0;
unsigned int stringify = 0x80000000;  /* 【修改】使用无符号数来执行位操作符运算 */
char buf[BUFFER_SIZE] = {0x00};
rc = sprintf_s(buf, sizeof(buf), "%u", stringify >> 24);
if (rc == -1 || rc >= sizeof(buf))
{
    /* 处理错误 */
}
```

5 内存管理安全

规则 C5.1：禁止引用未初始化的内存

说明：有些函数如 malloc()分配出来的内存是没有初始化的，可以使用 memset_s()进行清零，或者使用 calloc()进行内存分配，calloc()分配的内存是清零的（需要注意的是，calloc 函数会多一点开销）。当然，如果后面需要对申请的内存进行全部赋值，就不要清零了，但要确保内存被引用前是被初始化的。此外，分配内存初始化，可以消除之前可能存放在内存中的敏感信息，避免敏感信息的泄露。

错误示例：如下代码没有对 malloc()的 result 内存进行初始化，所以功能不正确。

```
int *CalcMetrixColomn( int **metrix ,int *param, size_t size )
{
    int *result = NULL;
    size_t bufsize = size * sizeof(int);
    ...
    /* 校验 bufsize 的合法性 */
    result = (int *)malloc(bufsize);
    if(NULL == result)
        return NULL;
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < size; j++)
            result[i] += metrix[i][j] * param[j];  /* 【错误】result 没有被初始化 */
    }
    ...
    return result;
}
/*...申请的内存使用后 free...*/
```

推荐做法 1：使用 memset_s()对分配出来的内存清零。

```
int *CalcMetrixColomn(int **metrix ,int *param, size_t size)
{
    int *result = NULL;
    size_t bufsize = size * sizeof(int);
    ...
    /* 校验 bufsize 的合法性 */
    result = (int *)malloc(bufsize);
    if(NULL == result)
        return NULL;
    int ret = memset_s(result, bufsize, 0, bufsize);  /*【修改】确保内存被初始化后才被引用*/
    /* 校验 ret，确保安全函数执行成功 */
```

```
    for(int i = 0; i < size; i++)
    {
        for(int j = 0; j < size; j++)
            result[i] += metrix[i][j] * param[j];
    }
    ...
    return result;
}
/*...申请的内存使用后 free...*/
```

推荐做法 2：使用 calloc()函数来申请内存。

```
int *CalcMetrixColomn(int **metrix ,int *param, size_t size)
{
    int *result = NULL;
    int i,j;
    ...
    result = (int *)calloc(size, sizeof(int)); /* 【修改】使用 calloc 函数来申请内存 */
    if(NULL == result)
        return NULL;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j++)
            result[i] += metrix[i][j] * param[j];
    }
    ...
    return result;
}

/*...申请的内存使用后 free...*/
```

规则 C5.2：禁止访问已经释放的内存

说明：访问已经释放的内存，是很危险的行为，主要分为两种情况：

（1）堆内存：一块内存释放了，归还内存池以后，就不应该再访问。因为这块内存可能已经被其他部分代码申请走，内容可能已经被修改；直接修改释放的内存，可能会导致其他使用该内存的功能不正常；读也不能保证数据就是释放之前写入的值。在一定的情况下，可以被利用执行恶意的代码。即使是对空指针的解引用，也可能导致任意代码执行漏洞。如果黑客事先对内存 0 地址内容进行恶意的构造，解引用后会指向黑客指定的地址，执行任意代码。

（2）栈内存：在函数执行时，函数内局部变量的存储单元都可以在栈上创建，函数执行完毕结束时这些存储单元自动释放。如果返回这些已释放的存储单元的地址（栈地址），可能导致程序崩溃或恶意代码被利用。

错误示例 1：下列代码中，ptr 在执行 ptr->next 语句前就已经被释放，解引用一个已经释放了内存的指针，会导致未定义的行为。

```
struct node
{
    int value;
    struct node *next;
};
void free_list(struct node *head)
{
    for (struct node *ptr = head; ptr != NULL; ptr = ptr ->next) /* 【错误】 解引用已经释放的内存 */
    {
        free(ptr);
    }
}
```

推荐做法 1：释放 ptr 之前，将 ptr->next 存储在 bptr 中。

```
struct node
{
    int value;
    struct node *next;
};
void free_list(struct node *head)
{
    struct node *;
    for (struct node *ptr = head; ptr != NULL; ptr = bptr)
    {
        bptr = ptr->next; /* 【修改】释放 ptr 前，见 ptr->next 进行保存 */
        free(ptr);
    }
}
```

错误示例 2：返回函数局部变量。当函数返回时，局部变量 name 会自动释放。因此当调用者强行访问这块内存时，会发生未定义的行为。

```
char* GetName()
{
    char name[STR_MAX] = {0x00};
    fillInName(name);
    return name;   /* 【错误】不能返回局部变量*/
}
```

因此程序员在任何时候都不能返回函数的局部变量。

规则 C5.3：禁止重复释放内存

说明：重复释放内存（double-free）会导致内存管理器出现问题。重复释放内存在一定情况下，有可能导致“堆溢出”漏洞，可以被用来执行恶意代码，具有很大的安全隐患。

错误示例：如下代码有可能重复释放内存 ptr。

```
char * ptr = (char*)malloc(SIZE);
...
if (abrt)
{
    free(ptr);
}
...
free(ptr);  /* 【错误】可能会产生双重释放 */
ptr = NULL;
...
```

推荐做法：确保申请的内存应该只释放一次。

```
char* ptr = (char*)malloc(SIZE);
...
if (abrt)
{
    free(ptr);
    ptr = NULL;
}
else  /* 【修改】确保不会发生双重释放  */
{
    ...
    free(ptr);
    ptr = NULL;
    ...
}
```

规则 C5.4：指针释放之后立即赋予新值

说明：悬挂指针可能会导致双重释放（double-free）以及访问已释放内存的危险。消除悬挂指针以及消除众多与内存相关危险的一个最为有效地方法就是当指针使用完后将其置 NULL 或者指向另一个合法对象。

需要注意的是，在一个特定的工程中，应当严格明确内存分配和释放分别是由调用者还是由被调用者负责。

错误示例：在如下代码中，message_type 为一个整数，message 为一个指向动态分配内存的指针。如果 message 在另外的代码片段中被引用，那么这段代码极有可能会发生双重释放的危险。

```
char *message = NULL;
message = (char *)malloc(MESS_SIZE);
if(!message)
{
    /* 处理错误 */
}
int message_type;
/* 初始化 message 和 message_type */
if (message_type == value)
{
    /* 处理 message_type */
    free(message);
}
/* ...*/
```

推荐做法：利用 free()去释放一个 NULL 指针不会产生任何行为。将 message 设置为 NULL 可以消除 message 被多重释放的可能。

```
char *message = NULL;
message = (char *)malloc(MESS_SIZE);
if(!message)
{
    /* 处理错误 */
}
int message_type;
/*初始化 message 和 message_type */
if (message_type == value)
{
    /*处理 message_type */
    free(message);
    message = NULL; /* 【修改】 message 释放后将其设置为 NULL*/
}
/* ...*/
```

如果一个指针释放后能够马上离开作用域（如下代码），因为它已经不能被再次访问，因此可以无需对其设置新值。

```
void foo(void)
{
```

```
    char *str;
    /* ... */
    free(str);
    return;
}
```

规则 C5.5：必须对指定申请内存大小的整数值进行合法性校验

说明：申请内存时没有对指定的内存大小整数作合法性校验，会导致未定义的行为，主要分为两种情况：

（1）使用 0 字节长度去申请内存的行为是没有定义的，在引用内存申请函数返回的地址时会引发不可预知或不能立即发现的问题。对于可能出现申请 0 地址的情况，需要增加必要的判断，避免出现这种情况

（2）使用负数长度去申请内存，负数会被当成一个很大的无符号整数，从而导致因申请内存过大而出现失败，造成拒绝服务。

错误示例：下列代码进行内存分配时，没有对内存大小整数作合法性校验。

```
int *GetRandomArray(size_t size)
{
    size_t *array = malloc(size * sizeof(size_t)); /* 【错误】未对 malloc 参数进行合法性校验 */
    if(NULL == array)
        return NULL;
    for(size_t i = 0; i < size; i++)
    {
        array[i] = CreatRand();
    }
    ...
}
/*...申请的内存使用后 free...*/
```

推荐做法：调用 malloc()之前，需要判断 malloc()的参数是否合法，以避免出现申请内存过大而导致拒绝服务。

```
int *GetRandomArray(size_t size )
{
    size_t *array = NULL;
```

```
    if((SIZE_T_MAX/sizeof(size_t) <= size )) /* 【修改】校验 malloc 参数，确保不会超出预计范围 */
        return NULL;
    array = malloc(size * sizeof(size_t));
    if(NULL == array)
        return NULL;
    for(size_t i = 0; i < size; i++)
    {
        array[i] = CreatRand();
    }
    ...
}
/*...申请的内存使用后 free...*/
```

规则 C5.6：禁止解引用空指针

说明：解引用空指针是一种未定义的行为。在很多平台上，解引用空指针可导致异常的程序终止。在一些特殊情况下，它可以被攻击者利用来执行任意代码。实际上，最为有效地防止空指针解引用的方法就是在指针使用前做非空校验（如下）：

```
str = malloc(size + 1);
if (NULL == str)
{
    /* 处理分配错误 */
}
```

错误示例：代码中，inputr_str 被拷贝到动态分配的内存 str 中。然而当 malloc()执行失败时会返回 NULL 给 str，当 str 在 memcpy_s()函数中被解引用时，程序会发生未定义的行为。

```
size_t size = strlen(input_str) + 1;
str = (char *)malloc(size); /* 【错误】 没有检查 malloc()的返回值 */
int ret = memcpy_s(str, size, input_str, size);
/* 校验 ret，确保安全函数执行成功 */
/* ... */
free(str);
str = NULL;
```

推荐方法：确保 malloc()函数返回值不为 NULL。

```
size_t size = strnlen(input_str, MAX_LEN) + 1;
str = (char *)malloc(size);
if (NULL == str) /* 【修改】确保 malloc()的返回值不为 NULL */
{
    /* 处理错误 */
}
errno_t rc = memcpy_s(str, size, input_str, size);
```

```
/* 校验 rc，确保安全函数执行成功 */
/* ... */
free(str);
str = NULL;
```

规则 C5.7：禁止使用 realloc()函数

说明：realloc()是一个非常特殊的函数，其原型如下：

```
void *realloc(void *ptr, size_t size)
```

随着参数的不同，其行为也是不同。

1） 当ptr与size均不为NULL时，该函数会重新调整内存大小，并将新的内存指针返回，并保证最小的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。

错误示例 1：不当使用导致内存泄露。

```
void *ptr = realloc(ptr, NEW_SIZE); /* 【错误】当 realloc()执行失败时会发生错误 */
if (!ptr)
{
    /* 错误处理 */
}
```

这里就引出了一个内存泄露的问题，当 realloc()分配失败的时候，会返回 NULL。但是参数中的 ptr 的内存是没有被释放的，如果直接将 realloc()的返回值赋给 ptr，那么 ptr 原来指向的内存就会丢失，造成内存泄露。

错误示例 2：大家都知道 malloc(0)是合法的语句，会返还一个合法的指针，且该指针可以通过 free()去释放。这就造成了很多人对 realloc()的错误理解，认为当 NEW_SIZE 为 0 时，实际上 realloc()也会返回一个合法的指针，后面依然需要使用 free()去释放该内存。

```
void *new_ptr = realloc(old_ptr, NEW_SIZE); /* 【错误】不要使用 realloc()函数 */
/* 其它代码 */
free(new_ptr);
```

由于错误的认识，不去检验 NEW_SIZE 是否为 0，如果 NEW_SIZE 等于 0，old_ptr 就会被释放掉，并且会返回 NULL。因此 new_ptr 就被置为 NULL，如果 new_ptr 在别处被调用，就可能会发生未定义的行为（如程序崩溃）。

正确示例 2：使用 malloc()函数代替 realloc()函数，并校验 malloc()函数是否执行成功。

```
void *new_ptr = malloc(NEW_SIZE); /* 【修改】 使用 malloc()函数代替 realloc()函数 */
if(NULL == new_ptr )
{
    /* 错误处理 */
```

```
}
int ret = memcpy_s(new_ptr, new_size, old_ptr, old_size);
/* 校验 ret，确保安全函数执行成功 */
/* 申请的内存使用后 free */
```

建议 C5.1：避免使用 alloca()函数申请内存

说明：POSIX 和 C99 均未定义 alloca()的行为，在不支持的平台上运行会有未定义的后果，且该函数在栈帧里申请内存，申请的大小可能越过栈的边界而无法预知。

错误示例：使用了 alloca()从堆栈分配内存。

```
char *UnfixedMessage = NULL;
UnfixedMessage = (char *)alloca(SINGLE_UNIT_SIZE); /* 【不推荐】不用使用 alloc()函数 */
if(NULL == UnfixedMessage)
    DoExit();
...
```

推荐做法：改用 malloc()从堆分配内存。

```
char *UnfixedMessage = NULL;
UnfixedMessage = (char *)malloc(SINGLE_UNIT_SIZE); /* 【修改】 使用 malloc()函数来代替 alloca()函数 */
if(NULL == UnfixedMessage)
    DoExit();
...
```

6 禁用不安全函数

规则 C6.1：禁止使用危险函数

说明：当有更加安全的函数存在时，应当禁止使用危险函数，而改用其安全版本。C 标准的许多函数，没有检查目标缓冲区的大小，很容易引入缓冲区溢出的安全漏洞。并且，这些函数没有处理一些特殊情况（例如内存重叠），因此是不安全的，可能会导致意想不到的问题。

以下列出了这些危险函数：

- 内存拷贝函数：memcpy(), wmemcpy(), memmove(), wmemmove()
- 内存初始化函数：memset()
- 字符串拷贝函数：strcpy(), wcscpy(),strncpy(), wcsncpy()
- 字符串拼接函数：strcat(), wcscat(),strncat(), wcsncat()
- 字符串格式化输出函数：sprintf(), swprintf(), vsprintf(), vswprintf(), snprintf(), vsnprintf()
- 字符串格式化输入函数：scanf(), wscanf(), vscanf(), vwscanf(), fscanf(), fwscanf(),vfscanf(),vfwscanf(),sscanf(), swscanf(), vsscanf(), vswscanf()
- stdin 流输入函数：gets()

这类函数是公认的危险函数，应禁止使用此类函数（微软从 Windows Vista 的开发开始就全面禁用了危险 API）。除此之外，对于这些危险系统函数的简单封装、或者自定义的与这些函数功能类似的函数却没有进行足够输入校验的，也是禁用的。

最优选择：使用 ISO/IEC TR 24731-1 定义的字符串操作函数的安全版本，如 strcpy_s()、strcat_s()、sprintf_s()、scanf_s()、gets_s() 等。这个版本的函数增加了以下安全检查：

- 检查源指针和目标指针是否为 NULL；
- 检查目标缓冲区的最大长度是否小于源字符串的长度；
- 检查复制的源和目的对象是否重叠。

了解更多关于危险函数替换，可以参考附录 C。

错误示例：使用不安全的函数。

```
int ProcessMessage(char * message)
{
    ...
    char temp[MSG_MAX_SIZE + 1]= {0x00};
    strcpy(temp,message);  /* 【错误】不用使用 strcpy()函数 */
    ...
}
```

示例代码中，temp 长度是固定的 MSG_MAX_SIZE + 1，而 message 的长度是不确定的，在 message 太大时就会发生缓冲区溢出。

推荐做法：使用带长度参数版本的函数或者自行实现安全版本，向目标缓冲区中复制指定长度的字符，截断超出限制的字符。

```
int ProcessMessage(char * message)
{
    ...
    char temp[MSG_MAX_SIZE + 1] = {0x00};
    int ret = strcpy_s(temp, sizeof(tem), message)   /* 【修改】使用安全版本函数 strcpy_s 来代替不安全函数 */
    /* 校验 ret，确保安全函数执行成功 */
    ...
}
```

规则 C6.2：禁止调用 OS 命令解析器执行命令或运行程序，防止命令注入

说明：命令解析器（如 UNIX/Linux 的 bash、sh，Windows 的 CMD.exe）支持命令分隔符（请参考附录 B），用于连续执行多个命令/程序。这是产生命令注入漏洞的根本原因。

C99 函数 system()/popen()的实现正是通过调用命令解析器来执行入参指定的程序/命令。如果 system()/popen()的参数由用户的输入组成，恶意用户可以通过构造恶意输入，改变函数调用的行为。

除非入参是硬编码的，否则禁止使用 system()和 popen()。

替代方案可以参考如下：

在 POSIX 下可以使用 exec 系列函数。但是参数中必须禁止再次调用命令解析器（如/bin/sh）来执行命令。

```
#define CMD any_cmd
...
execl("/bin/sh", "sh", "-c", CMD, (VOS_CHAR *) 0);
...
```

上述代码中 execl()函数调用“/bin/sh”来执行命令 CMD，这样做与直接使用 system()没有任何分别，同样会引起命令注入。

对于 Windows 系统，建议使用 Win32 API CreateProcess()等与命令解释器无关的进程创建函数来替代。同样需要注意的是不要创建再次调用命令解析器的进程。

对于确实需要使用 system()/popen()的场景，则必须做好输入校验，编写过滤函数，将命令字符串中的所有命令分隔符和特殊字符（参考附录 B）进行过滤。

错误示例：直接调用 system()函数执行用户命令。

```
void ProcessDirectory(char * input_dir )
```

```
{
    char command[COMMAND_SIZE + PATH_MAX + 1] = {0x00};
    size_t len = strnlen(input_dir, PATH_MAX);
    int ret = strncpy_s(command, sizeof(command), "any_command", COMMAND_SIZE);
    /* 校验 ret，确保安全函数执行成功 */
    ret = strncat_s(command, sizeof(command), input_dir, len);
    /* 校验 ret，确保安全函数执行成功 */
    system(command); /* 【错误】 因为参数不是硬编码的，所以不能使用 system()函数 */
    ...
}
```

上述代码将用户输入作为参数，在这种情况下，一旦用户输入类似下面的恶意参数：

```
anyExe; useradd attacker
```

shell 会将字符串“anyExe; useradd attacker”解释为两条独立的命令连续执行：

```
any_command anyExe
useradd attacker
```

这样攻击者通过注入了一条命令“useradd attacker”创建了一个新用户。这明显不是程序所希望的。

推荐做法 1：使用不会调用命令解析器的函数来执行，在 window 下可使用 CreateProcess()函数 。

```
void ProcessDirectory(char * input_dir )
{
    ...
    CreateProcess(  "any_cmd",input_dir,NULL,NULL,
                FALSE,CREATE_DEFAULT_ERROR_MODE,NULL,
                    NULL,NULL,NULL); /* 【修改】在 windows 下使用 CreatProcess()函数来代替 system() */
    ...
}
```

推荐做法 2：使用不会调用命令解析器的函数来执行，在 Linux 下可使用 exec 系列函数。

```
void ProcessDirectory(char * input_dir )
{
    ...
    char *args[] = {"",input_dir,NULL};
    char *envs[] = {NULL};
    ...
    execve("any_cmd", args, envs)  /* 【修改】在 Linux 下使用 exec 系列函数来代替 system() */
    ...
}
```

7 文件输入/输出安全

规则 C7.1：必须使用 int 类型来接收字符输入/输出函数的返回值

说明：字符输入/输出函数 fgetc()、getc()和 getchar()都从一个流读取一个字符，并把它以 int 值的形式返回。如果这个流到达了文件尾或者发生读取错误，函数返回 EOF。fputc()、putc()、putchar()和 ungetc()也返回一个字符或 EOF。

如果这些 I/O 函数的返回值需要与 EOF 进行比较，不要将返回值转换为 char 类型。因为 char 是有符号 8 位的值，int 是 32 位的值。如果 getchar()返回的字符的 ASCII 值为 0xFF，转换为 char 类型后将被解释为 EOF。0xFF 这个值被有符号扩展后是 0xFFFFFFFF，刚好等于 EOF 的值。

错误示例：下列代码使用 char 类型来接收字符 I/O 的返回值，可能会导致返回值错误。

```
char buf[BUFFSIZE + 1] = {0x00};
char c;  /* 【错误】不用使用 char 类型来接收字符 I/O 的返回值 */
size_t i = 0;
c = getchar();
while (c != '\n' && c != EOF && i < BUFSIZE)
{
    buf[i++] = c;
    c = getchar();
}
buf[i] = '\0'; /* terminate NTBS */
```

推荐做法：

```
char buf[BUFSIZE + 1] = {0x00};
int c;  /* 【修改】使用 int 类型来接收字符 I/O 的返回值 */
int i = 0;
c = getchar();
while (c != '\n' && c != EOF && i < BUFSIZE)
{
    buf[i++] = c;
    c = getchar();
}
buf[i] = '\0'; /* terminate NTBS */
```

注意：对于 sizeof(int) == sizeof(char)的平台，用 int 接收返回值也可能无法与 EOF 区分，这时要用 feof()和 ferror()检测文件尾和文件错误。

规则 C7.2：创建文件时必须显式指定合适的文件访问权限

说明：创建文件时，如果不显式指定合适访问权限，可能会让未经授权的用户访问该文件。访问权限依赖于文件系统，但一般文件系统都会提供控制访问权限的功能。

错误示例：下列代码没有显式配置文件的访问权限。

```
void  SaveConfigDataToFile(char * file_name)
{
    int fd = open(file_name, O_CREAT | O_WRONLY); /*【错误】 缺少访问权限设置*/
    if (-1 == fd)
    {
        /* 处理错误 */
    }
    DoSaveData();
    ...
}
```

推荐做法：为 open 函数设置文件访问权限。

```
void  SaveConfigDataToFile(char * file_name)
{
    /* 初始化 file_name 和 file_access_permissions */
    int fd = open(file_name, O_CREAT | O_WRONLY, S_IRUSR|S_IWUSR);
    /* 【修改】显式配置文件访问权限 */
    if (-1 == fd)
    {
        /* 处理错误 */
    }
    DoSaveData();
    ...
}
```

对于 open()函数，第三个参数用于设置文件访问权限的初始值，其仅当创建新文件时（即第二个参数使用了 O_CREAT 时）才使用。需要注意的是，真正创建文件时的权限还会受到 umask 值所影响，因此文件的权限应该为(mode&(~umask))。

对于 fopen()函数，其函数本身没有用于设置访问权限的参数，所有 fopen()创建的文件都默认具有权限 S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH (0666)，因此创建完后，可以使用 fchmod()和 umask()函数来显式配置文件的访问权限。

规则 C7.3：文件路径验证前，必须对其进行标准化

说明：当文件路径来自非信任域时，需要先将文件路径规范化再做校验。路径在验证时会有很多干扰因素，如相对路径与绝对路径，如文件的符号链接、硬链接、快捷路径、别名等。

所以在验证路径时需要对路径进行标准化，使得路径表达唯一化、无歧义。

如果没有作标准化处理，攻击者就有机会：

（1）构造一个跨越目录限制的文件路径，例如“../../../etc/passwd”或“../../../boot.ini”

（2）构造指向系统关键文件的链接文件，例如 symlink("/etc/shadow","/tmp/log")

通过上述两种方式之一可以实现读取或修改系统重要数据文件，威胁系统安全。

错误示例：直接将用户输入作为参数并执行。

```
void  SaveAllData(char * input_path)
{
    SaveConfigDataToFile(input_path); /* 【错误】不能直接将用户输入作为参数并执行 */
    ...
}
```

推荐做法（Linux）：Linux 下对文件进行标准化，可以防止黑客通过构造指向系统关键文件的链接文件。realpath()函数返回绝对路径，删除了所有符号链接：

```
void  SaveAllData(char * input_path)
{
    char path[PATH_MAX + 1] = {0x00};
    if( strlen(input_path) > PATH_MAX || NULL == realpath(input_path,path))  /*【修改】使用 realpath()函数来规范化文件路径 */
        return;
    ValidatePath(path);  /* 校验路径符合预期 */
    SaveConfigDataToFile(path);
    ...
}
```

推荐做法（Windows）：Windows 下可以使用 PathCanonicalize()函数对文件路径进行标准化。

```
void  SaveAllData(char * input_path)
{
    char path[PATH_MAX + 1] = {0x00};
    char *lppath = path;
    if(strlen(input_path) > PATH_MAX || PathCanonicalize (lppath,input_path) == FALSE)  /*【修改】使用 PathCanonicalize()函数来规范化文件路径 */
        return;
    ValidatePath(path);  /* 校验路径符合预期 */
    SaveConfigDataToFile(path);
    ...
```

```
}
```

需要注意的是：PathCanonicalize()函数没有对入参进行校验，使用不当极易引起缓冲区溢出，因此在使用时，一定要校验确保入参的大小不能超过目标缓冲区的大小。

同时，如果有条件可以使用微软推荐的 PathCchCanonicalize()、PathCchCanonicalizeEx()或 chenhPathAllocCanonicalize()函数来进行路径的规范化。

建议 C7.1：访问文件时尽量使用文件描述符代替文件名作为输入，以避免竞争条件问题

说明：该建议应用场景如下，当对文件的元信息进行操作时（比如修改它的所有者、对文件进行统计，或者修改它的权限位），首先要打开该文件，然后对打开的文件进行操作。只要有可能，应尽量避免使用获取文件名的操作，而是使用获取文件描述符的操作。这样做将避免文件在程序运行时被替换（一种可能的竞争条件）。

例如，当 access()和 open()两者都利用一个字符串参数而不是一个文件句柄来进行相关操作时，攻击者就可以通过在 access()和 open()之间的间隙替换掉原来的文件，如下所示：

行式打印	攻击者
access(”/tmp/attack”)	
	unlink(”/tmp/attack”)
	symlink(”/etc/shadow”, ”/tmp/attack”)
open(”/tmp/attack”)	

错误示例：下列代码使用 access()函数，可能引发竞争条件问题。

```
if(!access(file_name, W_OK))  /* 【不推荐】不要使用 access()函数，易引发条件竞争 */
{
    f = fopen(file_name, "w+");
    operate(f);
    /*...*/
    /* fclose f after operate(f)*/
}
else
{
    fprintf(stderr,"Unable to open file %s.\n",file);
}
```

推荐方法：在 Linux 下可以使用如下代码。

```
fd = creat(file_name, 0644);  /* 【修改】使用文件描述符来操作文件 */
if (fd == -1)
    return;
operate(fd);
```

```
/* 关闭文件操作符 */
```

8 信号

规则 C8.1：不要在信号处理程序中访问共享对象

说明：在信号处理程序中访问或者修改共享对象（主要是指结构体、数组或者字符串等）可能会引发条件竞争。有两种情况例外：读写无锁原子对象（lock-free atomic objects）和读写 volatile sig_atomic_t 类型变量。除此之外，在信号处理程序中访问任何其它类型对象都有可能导致未定义行为。

错误示例：下列代码中，err_msg 用于表明收到一个中断（SIGINT）信号。而 err_msg 变量是一个字符串指针而不是一个 volatile sig_atomic_t 类型变量。

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
enum { MAX_MSG_SIZE = 24 };
char *err_msg; /*【错误】字符串指针类型 err_msg 有可能会产生条件竞争 */
void handler(int signum)
{
    int ret = strcpy_s(err_msg, MAX_MSG_SIZE + 1, "SIGINT encountered.");
   /* 校验 ret，确保安全函数执行成功 */
}
int main(void)
{
    signal(SIGINT, handler);
    err_msg = (char *)malloc(MAX_MSG_SIZE + 1);
    if(NULL == err_msg)
    {
        /* 处理错误 */
    }
   int ret = strcpy_s(err_msg, MAX_MSG_SIZE + 1, "No errors yet.");
   /* 校验 ret，确保安全函数执行成功 */
    /* Main code loop */
    return 0;
}
```

推荐做法 1（写 volatile sig_atomic_t 类型）：为了获得最好的可移植性，信号处理程序应该无条件的将变量类型设置为 volatile sig_atomic_t 类型并返回。

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
enum { MAX_MSG_SIZE = 24 };
volatile sig_atomic_t e_flag = 0; /*【修改】使用一个 valatile sig_atomic_t 变量代替之 */
```

```
void handler(int signum)
{
  e_flag = 1;
}
int main(void)
{
    char *err_msg = (char *)malloc(MAX_MSG_SIZE + 1);
    if (NULL == err_msg)
    {
        /* 处理错误 */
    }
    signal(SIGINT, handler);
    int ret = strcpy_s(err_msg, MAX_MSG_SIZE + 1, "No errors yet.");
   /* 校验 ret，确保安全函数执行成功 */
     /* Main code loop */
    if (e_flag)
    {
        ret = strcpy_s(err_msg, MAX_MSG_SIZE + 1, "SIGINT received.");
       /* 校验 ret，确保安全函数执行成功 */
    }
    return 0;
}
```

推荐做法 2（无锁原子访问）：可以使用无锁原子访问对象。

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <stdatomic.h>

#if __STDC_NO_ATOMICS__ == 1
#error "Atomics is not supported"
#elif ATOMIC_INT_LOCK_FREE == 0
#error "int is never lock-free"
#endif

atomic_int e_flag = ATOMIC_VAR_INIT(0);
void handler(int signum)
{
    eflag = 1;
}
int main(void)
{
    enum { MAX_MSG_SIZE = 24 };
    char err_msg[MAX_MSG_SIZE + 1];
#if ATOMIC_INT_LOCK_FREE == 1
```

```
    if(!atomic_is_lock_free(&e_flag))
    {
        return EXIT_FAILURE;
    }
#endif
    if(signal(SIGINT, handler) == SIG_ERR)
    {
        return EXIT_FAILURE;
    }
    int ret = strcpy_s(err_msg, MAX_MSG_SIZE + 1, "No errors yet.");
   /* 校验 ret，确保安全函数执行成功 */
    /* Main code loop */
    if(e_flag)
    {
        ret = strcpy_s(err_msg, MAX_MSG_SIZE + 1, "SIGINT received.");
       /* 校验 ret，确保安全函数执行成功 */
    }
    return EXIT_SUCCESS;
}
```

规则 C8.2：在信号处理程序中只调用异步安全函数

说明：在信号处理程序中只能调用异步安全函数。对于要求比较严格的程序，只有 C 标准库中的 abort()， _Exit()，quick_exit()和 signal()函数可以在信号处理程序中安全的调用。使用异步不安全函数所导致的行为是未定义的。产品可能也会自定义一些异步安全函数，这些函数也可以在信号处理程序中使用。这条规则适用于库函数和程序自定义函数。

一般情况下，在信号处理程序中不要调用输入/输出（I/O）函数。程序员在任何时候，都要确保他们在信号处理程序中调用的函数都是异步安全的。

错误示例：下列代码中的信号处理程序通过 log_message()函数调用了 C 标准库函数 fprintf()和 free()函数，然而这两个函数不是异步安全的。

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
enum { MAXLINE = 1024 };
char *info = NULL;
void log_message(void)
{
    fprintf(stderr, info); /*【错误】fprintf()不是异步安全的 */
}
void handler(int signum)
{
    log_message();
```

```
    free(info); /*【错误】free()不是异步安全的 */
    info = NULL;
}
int main(void)
{
    if(signal(SIGINT, handler) == SIG_ERR)
    {
        /* 处理错误 */
    }
    info = (char *)malloc(MAXLINE);
    if (NULL == info)
    {
        /* 处理错误 */
    }
    while(1)
    {
        /* Main loop program code */
        log_message();
        /* More program code */
    }
    return 0;
}
```

推荐做法：信号处理程序应尽可能的简洁，理想状态下只设置一个标志（flag）并返回。下列代码中就是只设置了一个 volatile sig_atomic_t 类型 flag 并返回。main()函数中直接调用 log_message()和 free()函数。

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;
char *info = NULL;
void log_message(void)
{
    fprintf(stderr, info);
}
void handler(int signum)
{
    eflag = 1;
}
int main(void)
{
    if (signal(SIGINT, handler) == SIG_ERR)
    {
        /* 处理错误 */
```

```
    }
    info = (char *)malloc(MAXLINE);
    if(NULL == info)
    {
        /* 处理错误  */
    }
    while(!eflag)
    {
        /* Main loop program code */
        log_message();
        /* More program code */
    }
    log_message();
    free(info);
    info = NULL;
    return 0;
}
```

错误示例（longjmp()）：在信号处理程序中调用 longjmp()函数可能导致未定义的行为。因此无论是 longjmp()还是 POSIX 的 siglongjmp()函数都不能在信号处理程序中调用。

下列代码执行 main()中的循环来记录一些数据。一旦收到一个 SIGINT 信号，程序会退出循环，记录错误然后终止。

然而，攻击者可以在 log_message()函数中第二个 if 语句之前产生一个 SIGINT 信号来利用这个程序。导致的结果就是，longjmp()函数将控制权返回给 main()，然后 log_message()函数又一次被调用。但是此时第一个 if 语句将不会被执行（因为之前的中断，buf 没有被设置为 NULL），因此程序将会写入 buf0 所指向的无效内存地址。

```
#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>
enum { MAXLINE = 1024 };
static jmp_buf env;
void handler(int signum)
{
    longjmp(env, 1);
}
void log_message(char *info1, char *info2)
{
    static char *buf = NULL;
    static size_t bufsize;
    char buf0[MAXLINE] = {0x00};
    if(buf == NULL)
    {
        buf = buf0;
```

```
        bufsize = sizeof(buf0);
    }
    /*
    * Try to fit a message into buf, else reallocate
    * it on the heap and then log the message.
    */
    /* Program is vulnerable if SIGINT is raised here */
    if (buf == buf0)
    {
        buf = NULL;
    }
}
int main(void)
{
    if (signal(SIGINT, handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info1 = NULL;
    char *info2 = NULL;
    /* info1 and info2 are set by user input here */
    if(0 == setjmp(env))
    {
        while (1)
        {
            /* Main loop program code */
            log_message(info1, info2);
            /* More program code */
        }
    }
    else
    {
        log_message(info1, info2);
    }
    return 0;
}
```

推荐做法：删除 longjmp()的调用，信号处理程序只是设置了一个错误标志。

```
#include <signal.h>
#include <stdlib.h>
enum { MAXLINE = 1024 };
volatile sig_atomic_t eflag = 0;
void handler(int signum)
{
    eflag = 1;
```

```
}

void log_message(char *info1, char *info2)
{
    static char *buf = NULL;
    static size_t bufsize;
    char buf0[MAXLINE] = {0x00};
    if (buf == NULL)
    {
        buf = buf0;
        bufsize = sizeof(buf0);
    }
    /*
    * Try to fit a message into buf, else reallocate
    * it on the heap and then log the message.
    */
    if (buf == buf0)
    {
        buf = NULL;
    }
}
int main(void)
{
    if (signal(SIGINT, handler) == SIG_ERR)
    {
        /* Handle error */
    }
    char *info1 = NULL;
    char *info2 = NULL;
    /* info1 and info2 are set by user input here */
    while(!eflag)
    {
        /* Main loop program code */
        log_message(info1, info2);
        /* More program code */
    }
    log_message(info1, info2);
    return 0;
}
```

错误示例（raise()）：下列代码中嵌套调用 raise()函数，会导致未定义的行为。

```
#include <signal.h>
#include <stdlib.h>
void term_handler(int signum)
{
```

```
    /* SIGTERM handler */
}
void int_handler(int signum)
{
    /* SIGINT handler */
    if(raise(SIGTERM) != 0)
    {
        /* Handle error */
    }
}
int main(void)
{
    if (signal(SIGTERM, term_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    if(signal(SIGINT, int_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if (raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return EXIT_SUCCESS;
}
```

推荐做法：int_handler()调用了 term_handler()函数来代替 raise(SIGTERM)。

```
#include <signal.h>
#include <stdlib.h>
void term_handler(int signum)
{
    /* SIGTERM handler */
}
void int_handler(int signum)
{
    /* SIGINT handler */
    /* Pass control to the SIGTERM handler */
    term_handler(SIGTERM);
}
int main(void)
{
    if (signal(SIGTERM, term_handler) == SIG_ERR)
```

```
    {
        /* Handle error */
    }
    if(signal(SIGINT, int_handler) == SIG_ERR)
    {
        /* Handle error */
    }
    /* Program code */
    if(raise(SIGINT) != 0)
    {
        /* Handle error */
    }
    /* More code */
    return EXIT_SUCCESS;
}
```

想了解更多关于的异步安全函数，请参考附录 D。

9 内核操作安全

规则 C9.1：内核 mmap 接口实现中，确保对映射起始地址和大小进行合法性校验

说明：Linux 内核 mmap 接口中，经常使用 remap_pfn_range()函数将设备物理内存映射到用户进程空间。如果映射起始地址等参数由用户态控制并缺少合法性校验，将导致用户态可通过映射读写任意内核地址。如果攻击者精心构造传入参数，甚至可在内核中执行任意代码。

错误示例：如下代码在使用 remap_pfn_range()进行内存映射时，未对用户可控的映射起始地址和空间大小进行合法性校验，可导致内核崩溃或任意代码执行。

```
static int incorrect_mmap(struct file *file, struct vm_area_struct *vma)
{
    unsigned long size;
    size = vma->vm_end - vma->vm_start;
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
    if    (remap_pfn_range(vma,    vma->vm_start,    vma->vm_pgoff,    size,
vma->vm_page_prot)) /*【错误】未对映射起始地址、空间大小做合法性校验*/
    {
        err_log("%s, remap_pfn_range fail", __func__);
        return EFAULT;
    }
    else
    {
        vma->vm_flags &= ~VM_IO;
    }
    return EOK;
}
```

推荐做法：增加对映射起始地址等参数的合法性校验。

```
static int correct_mmap(struct file *file, struct vm_area_struct *vma)
{
    unsigned long size;
    size = vma->vm_end - vma->vm_start;
    if (!valid_mmap_phys_addr_range(vma->vm_pgoff, size)) /* 【修改】添加校验函数，
验证映射起始地址、空间大小是否合法 */
    {
        return EINVAL;
    }
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
    if    (remap_pfn_range(vma,    vma->vm_start,    vma->vm_pgoff,    size,
vma->vm_page_prot))
```

```
    {
        err_log("%s, remap_pfn_range fail", __func__);
        return EFAULT;
    }
    else
    {
        vma->vm_flags &= ~VM_IO;
    }
    return EOK;
}
```

规则 C9.2：内核程序中必须使用内核专用函数读写用户态缓冲区

说明：用户态与内核态之间进行数据交换时，如果在内核中不加任何校验（如校验地址范围、空指针）而直接引用用户态传入指针，当用户态传入非法指针时，可导致内核崩溃、任意地址读写等问题。因此，应当禁止使用 memcpy()、sprintf()等危险函数，而是使用内核提供的专用函数：copy_from_user()、copy_to_user()、put_user()和 get_user()来读写用户态缓冲区，这些函数内部添加了入参校验功能。

所有禁用函数列表为:memcpy()、bcopy()、memmove()、strcpy()、strncpy()、strcat()、strncat()、sprintf()、vsprintf()、snprintf()、vsnprintf()、sscanf()、vsscanf()。

错误示例 1：内核态直接使用用户态传入的 buf 指针作为 snprintf()的参数，当 buf 为 NULL 时，可导致内核崩溃。

```
ssize_t incorrect_show(struct file *file, char __user *buf, size_t size, loff_t *data)
{
        return snprintf(buf, size, "%ld\n", debug_level); /* 【错误】直接引用用户态传入指针，如果 buf 为 NULL，则空指针异常导致内核崩溃 */
}
```

推荐做法 1：使用 copy_to_user()函数代替 snprintf()。

```
ssize_t correct_show (struct file *file, char __user *buf, size_t size, loff_t *data)
{
    int ret = 0;
    char level_str[MAX_STR_LEN] = {0};
    snprintf(level_str, MAX_STR_LEN, "%ld\n", debug_level);
    if(strlen(level_str) >= size)
        return EFAULT;
    ret = copy_to_user(buf, level_str, strlen(level_str)+1); /* 【修改】使用专用函数 copy_to_user()将数据写入到用户态 buf，并注意防止缓冲区溢出 */
    return ret;
}
```

错误示例 2：内核态直接使用用户态传入的指针 user_buf 作为数据源进行 memcpy()操作，当 user_buf 为 NULL 时，可导致内核崩溃。

```
size_t incorrect_write(struct file *file, const char __user *user_buf, size_t count, loff_t *ppos)
{
    ...
    char buf[128] = {0};
    int buf_size = 0;
    buf_size = min(count, (sizeof(buf)-1));
    memcpy(buf, user_buf, buf_size); /* 【错误】直接引用用户态传入指针，如果 user_buf 为 NULL，则可导致内核崩溃 */
    ...
}
```

推荐做法 2：使用 copy_from_user()函数代替 memcpy()。

```
ssize_t correct_write (struct file *file, const char __user *user_buf, size_t count, loff_t *ppos)
{
    ...
    char buf[128] = {0};
    int buf_size = 0;
    buf_size = min(count, (sizeof(buf)-1));
    if (copy_from_user(buf, user_buf, buf_size)) /* 【修改】使用专用函数 copy_from_user()将数据写入到内核态 buf，并注意防止缓冲区溢出 */
        return EFAULT;
    ...
}
```

规则 C9.3：必须对 copy_from_user()拷贝长度进行校验，防止缓冲区溢出

说明：内核态从用户态拷贝数据时通常使用 copy_from_user()函数，如果未对拷贝长度做校验或者校验不当，会造成内核缓冲区溢出，导致内核 panic 或提权。

错误示例：未校验拷贝长度。

```
static long gser_ioctl(struct file *fp, unsigned cmd, unsigned long arg)
{
   char smd_write_buf[GSERIAL_BUF_LEN];
   switch (cmd)
   {
      case GSERIAL_SMD_WRITE:
      if (copy_from_user(&smd_write_arg, argp, sizeof(smd_write_arg))) {…}
      copy_from_user(smd_write_buf, smd_write_arg.buf, smd_write_arg.size);
      /* 【错误】拷贝长度参数 smd_write_arg.size 由用户输入，未校验 */
```

```
        …
    }
}
```

推荐做法：添加长度校验。

```
static long gser_ioctl(struct file *fp, unsigned cmd, unsigned long arg)
{
    char smd_write_buf[GSERIAL_BUF_LEN];
    switch (cmd)
    {
        case GSERIAL_SMD_WRITE:
        if (copy_from_user(&smd_write_arg, argp, sizeof(smd_write_arg))){…}
        if (smd_write_arg.size >= GSERIAL_BUF_LEN) {……} /*【修改】添加校验 */
        copy_from_user(smd_write_buf, smd_write_arg.buf, smd_write_arg.size);
        …
    }
}
```

规则 C9.4：必须对 copy_to_user()拷贝的数据进行初始化，防止信息泄漏

说明：内核态使用 copy_to_user()向用户态拷贝数据时，当数据未完全初始化（如结构体成员未赋值、字节对齐引起的内存空洞等），会导致栈上指针等敏感信息泄漏。攻击者可利用绕过 kaslr 等安全机制。

错误示例：未完全初始化数据结构成员。

```
static long rmnet_ctrl_ioctl(struct file *fp, unsigned cmd, unsigned long arg)
{
    struct ep_info info;
    switch (cmd) {
    case FRMNET_CTRL_EP_LOOKUP:
        info.ph_ep_info.ep_type = DATA_EP_TYPE_HSUSB;
        info.ipa_ep_pair.cons_pipe_num = port->ipa_cons_idx;
        info.ipa_ep_pair.prod_pipe_num = port->ipa_prod_idx;
        ret = copy_to_user((void __user *)arg, &info, sizeof(info));  /*【错误】
info 结构体有 4 个成员，未全部赋值 */
        …
    }
}
```

推荐做法：全部进行初始化。

```
static long rmnet_ctrl_ioctl(struct file *fp, unsigned cmd, unsigned long arg)
{
    struct ep_info info;
```

```
    int ret = memset_s(&info, sizeof(ep_info), '\0', sizeof(ep_info));   /* 【修改】使用 memset 初始化缓冲区，保证不存在因字节对齐或未赋值导致的内存空洞 */
    /* 校验 ret，确保安全函数执行成功 */
    switch (cmd) {
    case FRMNET_CTRL_EP_LOOKUP:
        info.ph_ep_info.ep_type = DATA_EP_TYPE_HSUSB;
        info.ipa_ep_pair.cons_pipe_num = port->ipa_cons_idx;
        info.ipa_ep_pair.prod_pipe_num = port->ipa_prod_idx;
        ret = copy_to_user((void __user *)arg, &info, sizeof(info));
        …
    }
}
```

规则 C9.5：禁止在异常处理中使用 BUG_ON 宏，避免造成内核 panic

说明：BUG_ON 宏会调用内核的 panic()函数，打印错误信息并主动崩溃系统，在正常逻辑处理中（如 ioctl 接口的 cmd 参数不识别）不应当使系统崩溃，禁止在此类异常处理场景中使用 BUG_ON 宏，推荐使用 WARN_ON 宏。

错误代码：正常流程中使用了 BUG_ON 宏

```
/* 判断 Q6 侧设置定时器是否繁忙，1-忙，0-不忙 */
static unsigned int is_modem_set_timer_busy(special_timer* smem_ptr)
{
    int i = 0;
    if(NULL == smem_ptr)
    {
        printk(KERN_EMERG "%s:smem_ptr NULL!\n", __FUNCTION__);
        BUG_ON(1);  /*【错误】系统 BUG_ON 宏打印调用栈后调用 panic()，导致内核拒绝服务，不应在正常流程中使用 */
        return 1;
    }
    …
}
```

推荐做法：去掉 BUG_ON 宏。

```
/* 判断 Q6 侧设置定时器是否繁忙，1-忙，0-不忙 */
static unsigned int is_modem_set_timer_busy(special_timer* smem_ptr)
{
    int i = 0;
    if(NULL == smem_ptr)
    {
        printk(KERN_EMERG "%s:smem_ptr NULL!\n", __FUNCTION__);
        /* 【修改】去掉 BUG_ON 调用，或使用 WARN_ON */
        return 1;
```

```
    }
    …
}
```

规则 C9.6：在中断处理程序或持有自旋锁的进程上下文代码中，禁止使用会引起进程休眠的函数

说明：Linux 以进程为调度单位，在 Linux 中断上下文中，只有更高优先级的中断才能将其打断，系统在中断处理的时候不能进行进程调度。如果中断处理程序处于休眠状态，就会导致内核无法唤醒，从而使得内核处于瘫痪。

自旋锁在使用时，抢占是失效的。若自旋锁在锁住以后进入睡眠，由于不能进行处理器抢占，其它进程都将因为不能获得 CPU（单核 CPU）而停止运行，对外表现为系统将不作任何响应，出现挂死。

因此，在中断处理程序或持有自旋锁的进程上下文代码中，应该禁止使用可能会引起休眠的函数（如 printk()，vmalloc()等）。

规则 C9.7：合理使用内核栈，防止内核栈溢出

说明：Linux 的内核栈大小是固定的（一般 32 位系统为 8K，64 位系统为 16K），因此资源非常宝贵。不合理的使用内核栈，可能会导致栈溢出，造成系统挂死。因此需要做到以下几点：

- 在栈上申请内存空间不要超过内核栈大小；
- 注意内核函数的嵌套使用次数；
- 不要定义过多的变量。

错误代码：以下代码中定义的变量过大，导致栈溢出。

```
...
struct result
{
    char name[4];
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};  /* 结构体 result 的大小为 20 字节 */

int func()
{
    struct result temp[512];
    /*【错误】temp 数组含有 512 个元素，总大小为 10K，远超内核栈大小 */
```

```
    int ret = memset_s(temp, sizeof(result) * 512, 0, sizeof(result) * 512);
    /* 校验 ret，确保安全函数执行成功 */
    /*use temp do something...*/
    return 0;
}
...
```

代码中数组 temp 有 512 个元素，总共 10K 大小，远超内核的 8K，明显的栈溢出。

推荐做法：使用 kmalloc()代替之。

```
...
struct result
{
    char name[4];
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};   /* 结构体 result 的大小为 20 字节 */

int func()
{
    struct result *temp = NULL;
    temp = (result *)kmalloc(sizeof(result) * 512, GFP_KERNEL); /*【修改】使用
kmalloc()申请内存 */
    /* check temp is not NULL*/
    (void)memset_s(temp, sizeof(result) * 512, 0, sizeof(result) * 512);
    /*use temp do something...*/
    /* free temp*/
    return 0;
}
...
```

规则 C9.8：临时关闭地址校验机制后，在操作完成后必须及时恢复

说明：SMEP 安全机制是指禁止内核执行用户空间的代码（PXN 是 ARM 版本的 SMEP）。系统调用（如 open()，write()等）本来是提供给用户空间程序访问的。默认情况下，这些函数会对传入的参数地址进行校验，如果入参是非用户空间地址则报错。因此，要在内核程序中使用这些系统调用，就必须使参数地址校验功能失效。set_fs()/get_fs()就用来解决该问题。详细说明见如下代码：

```
...
mm_segment_t old_fs;
printk("Hello, I'm the module that intends to write message to file.\n");
```

```
if(file == NULL)
    file =filp_open(MY_FILE, O_RDWR | O_APPEND | O_CREAT, 0664);
if(IS_ERR(file))
{
    printk("Error occured while opening file %s, exiting...\n", MY_FILE);
    return 0;
}
sprintf_s(buf, BUF_SIZE, "%s", "The Message.");
old_fs = get_fs(); /*get_fs()的作用是获取用户空间地址上限值 #define get_fs()
(current->addr_limit */
set_fs(KERNEL_DS); /* set_fs 的作用是将地址空间上限扩大到 KERNEL_DS，这样内核代码可以
调用系统函数 */
file->f_op->write(file, (char *)buf, sizeof(buf), &file->f_pos); /* 内核代码可
以调用 write()函数 */
set_fs(old_fs); /* 使用完后及时回复原来用户空间地址限制值 */
...
```

通过上述代码，可以了解到最为关键的就是操作完成后，要及时恢复地址校验功能。否则 SMEP/PXN 安全机制就会失效，使得许多漏洞的利用变得很容易。

错误代码：在程序错误处理分支，未通过 set_fs()恢复地址校验功能。

```
...
oldfs = get_fs();
set_fs(KERNEL_DS);
/* 在时间戳目录下面创建 done 文件 */
fd = sys_open(path, O_CREAT | O_WRONLY, FILE_LIMIT);
if(fd < 0)
{
    BB_PRINT_ERR("sys_mkdir [%s] error, fd is [%d]\n", path, fd);
    return; /* 【错误】在错误处理程序分支未恢复地址校验机制 */
}

sys_close(fd);
set_fs(oldfs);
...
```

推荐做法：在错误处理程序中恢复地址校验功能。

```
...
oldfs = get_fs();
set_fs(KERNEL_DS);
/* 在时间戳目录下面创建 done 文件 */
fd = sys_open(path, O_CREAT | O_WRONLY, FILE_LIMIT);
if(fd < 0)
{
    BB_PRINT_ERR("sys_mkdir [%s] error, fd is [%d]\n", path, fd);
    set_fs(oldfs); /* 【修改】在错误处理程序分支中恢复地址校验机制 */
```

```
    return;
}
sys_close(fd);
set_fs(oldfs);
```

10 其它

规则 C10.1：禁止使用不安全的 C 标准库函数产生用于安全用途的伪随机数

说明：C 标准库函数 rand()和 random()产生的随机数随机性很不好，其产生的随机数序列存在一个较短的循环周期，因此它的随机数是可预测的，禁止用于安全用途。

安全用途的场景包括但不限于以下几种：

- 重要SessionID的生成；
- 挑战算法中的随机数生成；
- 验证码的随机数生成；
- 生成重要随机文件（例如存有系统信息的文件等）的随机文件名；
- 用于密码算法用途（例如用于生成IV、盐值、密钥等）的随机数生成。

错误示例：下列代码中使用不安全的随机数生成函数 rand()来生成随机数。

```
void  GenerateRandomNumber()
{
    enum {LEN = 12};
    char SessionID[LEN + 1] = {0x00}; /* SessionID will hold the ID, starting with
                    * the characters "ID" followed by a
                    * random integer */
    int r = 0;
    int num = 0;
    /* ...do something... */
    r = rand(); /* 【错误】rand()产生的随机数是可以被预测的 */
    num = sprintf_s(SessionID, sizeof(SessionID), "ID-%d", r); /* generate the
ID */
   /* 校验 num，确保安全函数执行成功 */
    /* ...do something... */
}
```

以上代码利用 rand()产生一个 ID 的数字部分，因此这些 ID 是可预测的并且随机性有很大限制。

推荐做法：Unix/Linux 下推荐读取/dev/random 文件来获取真随机数。

```
void  GenerateRandomNumber()
{
    enum {LEN = 12};
    char SessionID[LEN + 1] = {0x00}; /* SessionID will hold the ID, starting with
                     * the characters "ID" followed by a
                     * random integer */
    int r = 0;
```

```
    int num = 0;
    /* ...do something... */
    int fd;
    fd = open("/dev/random", O_RDONLY); /* 通过读取/dev/random 来获取随机数 */
    if (fd > 0)
    {
        read (fd,&r,sizeof (int));
    }
    close (fd);
    num = sprintf_s(SessionID, sizeof(SessionID), "ID-%d", r);  /* generate the
ID */
   /* 校验 num，确保安全函数执行成功 */
    /* ...do something... */
}
```

Windows 推荐使用随机数生成函数 CryptGenRandom()：

```
#include "Wincrypt.h"
void  GenerateRandomNumber()
{
    HCRYPTPROV hCryptProv;
    union
    {
        BYTE bs[sizeof(long int)];
        long int li;
    } rand_buf;
    if (!CryptGenRandom(hCryptProv, sizeof(rand_buf), &rand_buf))
    {
        /* Handle error */
    }
    else
    {
        printf("Random number: %ld\n", rand_buf.li);
    }
}
```

由于以上推荐的 2 种做法并不能保证主流编译环境下满足可靠性要求，对于可靠性要求很严格的产品可以使用开源组件 openssl：

OpenSSL 示例：

```
#include <openssl/rand.h>
#include <stdio.h>
#pragma comment(lib, "libeay32.lib")
#define BUF_MAX 100
void main()
{
    unsigned char buf[BUF_MAX + 1] = {0x00};
```

```
    RAND_screen();  /* load screen data as seed */
    if(RAND_status() == 1)
    {
        RAND_bytes(buf, sizeof(buf)); /*  use RAND_bytes to generate random
number */
        for (int i = 0; i < sizeof(buf); i++)
        {
            printf("%02X", buf[i]);/* print random data */
        }
    }
    RAND_cleanup();
}
```

规则 C10.2：禁止存储某些特殊函数返回的字符串指针

说明： 某些特殊函数返回的指针指向的是静态分配的内存，其值很有可能会被随后的类似函数调用而改写，存储该返回值可能会导致一个危险的指针或者引用错误的数据，所以不要存储这些函数返回的字符串指针。这些返回的字符串应该被马上引用然后丢弃，如果需要后续使用该字符串，应该把该字符串拷贝到动态分配的内存里，然后在需要的时候引用该份拷贝。

例如 getenv()返回指针的指向值可能会被后续的 getenv()，putenv()，setenv()，unsetenv()调用所改写，或因其它操作修改了环境变量而变得无效。

除此之外，getenv()不是线程安全的，要确保处理使用该函数可能导致的竞争情况。

存在类似问题的函数还有：asctime()，localeconv()，setlocale()和 strerror()。自定义函数若具有上述特性，同样需要注意。

错误示例：下列代码错误的保存了 getenv()返回的字符串指针。

```
void  InitProgramEnvironment()
{
    char *tmpvar = getenv("TMP");
    if (!tmpvar)
        return -1;
    char *tempvar = getenv("TEMP"); /* 【错误】不要存储 getenv()返回的字符串指针*/
    if (!tempvar)
        return -1;
    if (0 == CompareValue(tmpvar, tempvar)) /* The two values may compare equal
*/
    {
        /* … */
```

```
    }
    /* … */
}
```

示例代码比较环境变量 TMP 和 TEMP 的值是否相同。在示例代码中，tmpvar 指向的内容可能会因为第二次调用 getenv()而被改写，从而导致 tmpvar 和 tempvar 指向相同的内容，即使环境变量 TMP 和 TEMP 的值并不相同。

推荐做法：使用 malloc()和 strcpy_s()来拷贝存储 getenv()的返回值。

```
char *GetEnvStr(const char* env)
{
    const char *temp = getenv(env);
    if (temp != NULL)
    {
        int len = strnlen(temp, MAX_LEN);
        tmpvar = (char *)malloc(len + 1);
        if (tmpvar != NULL)
        {
            int ret = strcpy_s(tmpvar, len + 1, temp);
            /* 校验 ret，确保安全函数执行成功 */
            return tmpvar;
        }
    }
    return NULL;
}
void  InitProgramEnvironment()
{
    char *tmpvar = GetEnvStr("TMP");
    if (NULL == tmpvar)
    {
        /* Handle error */
    }
    char *tempvar = GetEnvStr("TEMP");  /* 【修改】将 getenv()的字符串存储下来*/
    if (NULL == tempvar)
    {
        /* Handle error */
    }
    if (0 == CompareValue(tmpvar, tempvar))
    {
        /* ...do something... */
    }
    /* ...do something... */
    /* ... free after tempvar... */
}
```

规则 C10.3：多线程环境下只使用可重入函数

（1）多线程环境下，禁止 std::cout 与 printf 混用

说明：printf 与 std::cout 分别为标准 c 语言与 C++中的函数，两者的缓冲区机制不同（printf 无缓冲区，而 std::cout 有），而且对于标准输出的加锁时机也略有不同：

- printf：在对标准输出作任何处理前先加锁；
- std::cout：在实际向标准输出打印时方才加锁；

二者存在微弱的时序差别，而多线程环境下，很多问题就是由于微弱的时序差别造成的。所以两者的混用很容易带来不可预知的错误，常见的错误有打印输出的结果不符合预期，而严重错误时甚至会导致内部缓存区溢出，导致 crash。

错误示例：cout 和 printf 混用可能导致 crash。

```
void OutputDataToConsole()
{
    int j = 0;
    for(j = 0; j < 5; ++j)
    {
        cout << "j="; /*【错误】混用 cout 和 printf 在多线程中会导致 crash */
        printf("%d\n", j);
    }
}
```

上面代码的输出结果很可能为：

```
1

2

3

4

j=j=j=j=j=
```

这很明显不符合程序员的预期。造成这样错误的原因就是 std::cout 的标准流输出是带有缓冲区的，如果没有及时清理缓冲区而在期间采用了其它系统的输出函数，可能会暴露两种输出函数的不兼容性，从而出现非预期错误。所以建议在代码中检查对于系统标准打印输出的兼容性，一

定要使用统一的打印输出方法,而对于 C++程序，更推荐统一使用流输出方法，而不推荐使用 C 风格的代码。

推荐做法：

```
void OutputDataToConsole()
{
    int j = 0;
    for(j = 0; j < 5; ++j)
    {
        printf("j = %d\n", j); /* 【修改】只使用 printf()函数 */
    }
}
```

或者

```
void OutputDataToConsole()
{
    int j = 0;
    for(j = 0; j < 5; ++j)
    {
        cout << "j=" << j << endl; /*【修改】只使用 cout()函数 */
    }
}
```

（2）多线程环境下，禁止使用 strtok 函数

说明：strtok 是一个线程不安全的函数，因为它使用了静态分配的空间来存储被分割的字符串位置。初次调用 strtok 时传递一个字串的地址，比如”aaa.bbb.dddd”,将字串的地址保存在自己的静态变量中，当再次调用 strtok 并传递 NULL 时（strtok 的特殊用法，第二次调用时字串传 NULL 表示对第一次传进去的字串继续分隔，所以要先保存字串地址），该函数就会引用保存好的字串地址。在多线程环境下，另一个线程也可能调用 strtok，在这种环境下，另一个线程会在第一个线程不知道的情况下替换静态变量中的字串地址,这就会导致各种难以排除的错误出现。

错误示例：下面这个程序是一个用来确定一个文本中的每行单词个数的平均次数的错误算法。wordaverage 函数用来确定每一行，不幸的是，wordcount 函数也使用了 strtok，这一次是用它来解析本行中的字，这时，strtok 保持的内部状态信息被改变了。

```
#include <string.h>
#define LINE_DELIMITERS "\n"
#define WORD_DELIMITERS " "
static int WordCount(char *s)
{
```

```
    int count = 1;
    if(NULL == strtok(s,WORD_DELIMITERS))
    /* 【错误】在多线程环境中使用 strtok 会导致 crash */
        return 0;
    while(strtok(NULL, WORD_DELIMITERS) != NULL)
        count++;
    return count;
}
double WordAveraget(char *s)
{
    int linecount = 1;
    char *nextline = NULL;
    int words = 0;
    nextline = strtok(s, LINE_DELIMITERS);
    if(NULL == nextline)
        return 0.0;
    words = WordCount(nextline);
    while((nextline = strtok(NULL, LINE_DELIMITERS)) != NULL)
    {
        words += WordCount(nextline);
        linecount++;
    }
    return (double)words/linecount;
}
```

推荐做法：带有_r 的函数主要来自于 UNIX 平台。所有的带有_r 和不带_r 的函数的区别的是：带_r 的函数是线程安全的，r 的意思是 reentrant，可重入的。

```
#include <string.h>
#define LINE_DELIMITERS "\n"
#define WORD_DELIMITERS " "
static int WordCount(char* s)
{
    int count = 1;
    char *lasts = NULL;
    if(NULL == strtok_r(s, WORD_DELIMITERS, &lasts))
    /* [Modification] Use the strtok_r() function instead of strtok() */
        return 0;
    while(strtok_r(NULL ,WORD_DELIMITERS, &lasts) != NULL)
        count++;
    return count;
}
double WordAverage (char *s)
{
```

```
    int linecount = 1;
    char *nextline = NULL;
    int words = 0;
    char *lasts = NULL;
    nextline = strtok_r(s, LINE_DELIMITERS, &lasts);
    if(NULL == nextline)
        return 0.0;
    words = WordCount(nextline);
    while((nextline = strtok_r(NULL, LINE_DELIMITERS, &lasts)) != NULL)
    {
        words += WordCount(nextline);
        linecount++;
    }
    return (double)words/linecount;
}
```

规则 C10.4：检查返回值

说明：一些函数，如输入/输出函数和内存分配函数等，在执行完成后，要么会返回一个合法值，要么会返回一个指示错误类型的数据（例如-1 或者空指针 NULL）。如果都想当然的认为它们都会执行成功不会失败，那么当错误发生时，往往会出现意想不到或未定义的行为。因此，程序员必须严格的校验并合适的处理函数返回的错误值。

错误示例：代码中函数 utf8_to_ucs()的作用是将 UTF-8 字符串转成 UCS 格式。函数首先调用 setlocale()设置全球语言环境为"en_US.UTF-8"，但是却没有校验函数执行成功与否。setlocale()函数执行失败（比如资源缺失、语言环境没有安装等情况下）后会返回 NULL 指针。这就导致接下来调用的 mbstowcs()函数要么执行失败，要么会将一些无法预知的宽字节字符写入到 usc 中。

```
size_t utf8_to_ucs(wchar_t *ucs, size_t n, const char *utf8)
{
    setlocale(LC_CTYPE, "en_US.UTF-8"); /* 【错误】没有校验函数返回值*/
    return mbstowcs(ucs, utf8, n);
}
```

推荐做法：校验 setlocale()的返回值，避免在该函数执行失败后调用 mbstowcs()。同时，函数需要考虑在返回给调用者之前，恢复语言环境的初始化设置。

```
size_t utf8_to_ucs(wchar_t *ucs, size_t n, const char *utf8)
{
    const char *save = NULL;
    save = setlocale(LC_CTYPE, "en_US.UTF-8");
    if (NULL == save)  /*【修改】调用函数返回值*/
    {
```

```
        /* Propagate error to caller */
        return (size_t)-1;
    }
    n = mbstowcs(ucs, utf8, n);
    if (NULL == setlocale(LC_CTYPE, save))
        n = -1;
    return n;
}
```

规则 C10.5：禁止使用不可信数据拼接 SQL 命令

说明：SQL 注入是指原始 SQL 查询被恶意动态更改成一个与程序预期完全不同的查询。执行更改后的查询可能会导致信息泄露或者数据被篡改。而 SQL 注入的根源就是使用不可信的数据来拼接 SQL 语句。C/C++语言中常见的使用不可信数据拼接 SQL 语句的底层场景有（包括但不局限于）：

- 连接MySQL时调用mysql_query(),Execute()时的入参
- 连接SQL Server时调用db-library驱动的dbsqlexec()的入参
- 调用ODBC驱动的SQLprepare()连接数据库时的SQL语句参数
- C++程序调用OTL类库中的otl_stream()，otl_column_desc()时的入参
- C++程序连接Oracle数据库时调用ExecuteWithResSQL()的入参

因此，这些场景中切不可直接使用不可信数据来拼接 SQL 语句。同时，封装的 SQL 命令执行接口，也必须遵守同样的原则。

而防止 SQL 注入的方法主要有以下几种：

- 参数化查询（通常也叫作预处理语句）：参数化查询是一种简单有效的防止SQL注入的查询方式，应该被优先考虑使用。支持的数据库有MySQL，Oracle（OCI）。
- 参数化查询(通过ODBC驱动)：支持ODBC驱动参数化查询的数据库有Oracle、SQLServer、PostgreSQL和GaussDB。
- 对不可信数据进行校验（推荐“白名单”校验）。
- 对不可信数据中的SQL特殊字符进行转义（参见附录A）。

错误示例：下列操作 MySQL 的代码中使用来自用户的不可信数据 name 来拼接 SQL 语句，并直接执行，可能会造成 SQL 注入。

```
...
MYSQL my_connection = NULL;
int res = 0;
char name[20] = {0x00};
char sqlstatements[100] = {0x00};

my_connection = mysql_init(NULL);
```

```
/* validate my_connection */
if (mysql_real_connect(&my_connection, "localhost", "username", "passwd",
"newdatabase", 0, NULL, 0))
{
    printf("Connection success\n");
    name = getuserinput();  /* name 来自外部用户输入，且保证 < 20 字节 */
    res = sprintf_s(sqlstatements, 100, "SELECT childinfo FROM children WHERE
name= '%s'", name ); /* 【错误】使用不可信数据拼接 SQL 语句*/
    /* validate res */
    res = mysql_query(&my_connection, sqlstatements);
    /* validate res */
    ...
}
mysql_close(my_connection);
...
```

推荐做法：使用参数化查询。

```
...
MYSQL my_connection = NULL;
int res = 0;
char name[20] = {0x00};
char sqlstatements[100] = {0x00};

my_connection = mysql_init(NULL);
/* validate my_connection */
if (mysql_real_connect(&my_connection, "localhost", "username", "passwd",
"newdatabase", 0, NULL, 0))
{
    /* 【修改】 使用预处理语句进行参数化查询 */
    MYSQL_STMT *stmt = mysql_stmt_init(my_connection); /* 创建 MYSQL_STMT 句柄 */
    char *query = "SELECT childinfo FROM children WHERE name= ?";
    if(mysql_stmt_prepare(stmt, query, strlen(query)))
    {
       /* handle error */
    }
    name = getuserinput(); /* name 来自外部用户输入，且保证 < 20 字节 */
    MYSQL_BIND params[1];
    res = memset_s(params, sizeof(params), 0, sizeof(params));
    /* validate res */
    params[0].buffer_type = MYSQL_TYPE_STRING;
    params[0].buffer = (char *)name;
    params[0].buffer_length = strlen(name);
    params[0].is_null= 0;

    res = mysql_stmt_bind_param(stmt, params); /* 绑定参数 */
```

```
    /* validate res */
    res = mysql_stmt_execute(stmt);    /* 执行与语句句柄相关的预处理 */
    /* validate res */
    ...
    mysql_stmt_close(stmt);
    ...
}
...
mysql_close(my_connection);
...
```

建议 C10.1：编译时应当使用编译器的最高警告等级

说明：程序员应当使用编译器的最高警告等级。在编译过程中，应该修改程序中的错误，直到警告解除。在此同时，应当使用静态和动态的分析工具来检测和清除安全缺陷。

另外，开启一些和安全相关的编译选项，可以使编译出来的程序具有更好的安全特性。

对于 GCC 编译器，建议开启以下安全选项：

（1）使用 PIE 选项（gcc -fPIE / ld -pie），可以将源代码编译成和位置无关的可执行程序。

（2）使用-fstack-protector-all 或-fstack-protector 选项，通过栈保护，来防止程序出现缓冲区溢出错误。

其它类似的选项还有如下这些：

- Werror：强制将所有的告警标记为错误，因此可以强制程序员定位错误；
- Wconversion：在隐式类型转换时告警；
- -Wformat-security：对于printf()族格式化函数进行特殊检查；
- -Wextra：在可能发生安全危险时，产生更多告警；
- -D_FORTIFY_SOURCE=2：检测部分缓冲区溢出风险；
- -Wl, -z relro, -z now：加固ELF内部数据段；
- -Wstack-protector：当函数没有防范栈溢出时进行告警；
- -param ssp-buffer-size=4：控制栈溢出保护（SSP）的最小缓冲区值。

对于 VC（VS2005 及以上的版本）编译器，可以开启如下和安全相关的编译选项：

（1）GS（栈保护）：通过在栈中加入校验单元来防止出现缓冲区溢出错误。

（2）NXCOMPAT（与数据执行保护兼容）：DEP，也就是数据执行保护，可以有效降低堆或栈上的缓存溢出漏洞的危害性。采用 NXCOMPAT 选项后，应用程序的运行被 DEP 机制保护。在考虑兼容性的前提下，建议开发人员采用 NXCOMPAT 链接选项。

（3）ASLR（地址空间分布随机）：是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

（4）SafeSEH：SafeSEH 会增加缓冲区利用的难度。一旦开启/SAFESEH，那么编译器生成二进制 IMAGE 的时候，会把所有合法的 SEH 函数的地址解析出来，在 IMAGE 里生成一张合法的 SEH 函数表，用于异常处理时候进行严格的匹配检查。

建议 C10.2：防止处理敏感数据的代码因被编译器优化而失效

说明：有时候编译器在优化时会删除一些它认为不必要的代码，但实际这些看似多余的代码是存在安全考虑。一个典型的例子就是函数返回前清除栈上敏感数据的操作，如果这些操作被删除掉，攻击者就有机会访问栈上的敏感数据。因此必须确保这些安全操作在编译器优化的场景下仍然得以执行。

错误示例：下列代码使用了可能被编译器优化掉的语句。

```
void  SecureLogin()
{
    char pwd[PWD_SIZE + 1] = {0x00};
    if (retrievePassword(pwd, sizeof(pwd)))
    {
        /* checking of password, secure operations, etc */
    }
    memset(pwd, 0, sizeof(pwd));    /* 【错误】编译器优化有可能会使该语句失效 */
    …
}
```

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()操作会被优化掉。

以下列出了几种可能的解决方法，其中的某些方法不具有普适性，因此需要结合实际选择相应的方法。

推荐做法 1：使用安全函数库中的 memset_s()函数。

```
void  SecureLogin()
{
    char pwd[PWD_SIZE + 1] = {0x00};
    if (retrievePassword(pwd, sizeof(pwd)))
    {
        /* checking of password, secure operations, etc */
    }
    int ret = memset_s(pwd, sizeof(pwd), 0, sizeof(pwd));    /*【修改】使用 memset_s 函数代替之 */
   /* 校验 ret，确保安全函数执行成功 */
    …
}
```

推荐方法 2：使用 SecureZeroMemory()函数代替 memset。

```
void  SecureLogin()
```

```
{
    char pwd[PWD_SIZE + 1] = {0x00};
    if (retrievePassword(pwd, sizeof(pwd)))
    {
        /*checking of password, secure operations, etc */
    }
    SecureZeroMemory(pwd, sizeof(pwd));
   /* 【修改】使用 SecureZeroMemory()代替 memset */
    …
}
```

该函数可以确保不被优化，但是只适用于 Windows 系统。

推荐做法 3：

```
void  SecureLogin()
{
    char pwd[PWD_SIZE + 1] = {0x00};
    if (retrievePassword(pwd, sizeof(pwd)))
    {
        /* checking of password, secure operations, etc */
    }
    #pragma optimize("", off)   /* [【修改】禁用部分优化编译选项，确保 pwd 被处理*/
    /* 清除内存 */
    #pragma optimize("", on)
    …
}
```

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要优化此处的操作。

推荐方法 4：编写自定义的安全的内存清零函数。

推荐方法 5：如果确实需要使用 memset，可以使用如下方法。

```
void  SecureLogin()
{
    char pwd[PWD_SIZE + 1] = {0x00};
    if (retrievePassword(pwd, sizeof(pwd)))
    {
        /* checking of password, secure operations, etc */
    }
    memset(pwd, 0, sizeof(pwd));
    (volatile char*)pwd = *(volatile char*)pwd;   /* 【修改】使用 volatile 自赋值语句，确保 pwd 被处理 */
    …
}
```

但是需要注意的是，某些编译器（如 MIPSpro，GCC3.0 及以上版本）虽然会执行 memset 语句，但是却会智能的只将目标 buffer 的首字节置零而其余部分却仍保持完好。此种情况下，则不可采取此种方法。

建议 C10.3：函数参数定义尽量使用 const

说明：如果想要避免函数（方法）中的参数被意外改写，应该使用 const。

尤其是当为其它应用程序开发接口或者与其他团队合作开发时，正确使用 const 可以有效地避免数据覆写和误解。为了正确的使用 const，应仔细检查所使用的库和头文件。即使当程序员认为确实不会修改数据时，也应该使用 const 加以确保。

错误示例：下列代码中，程序员忽视了一个问题，strtok_s()函数不是工作在一份拷贝上的，它会修改要分割的字符串。

```
char *get_second_token(char *input)
{
    /* do dome strtok_s() that overwrites input's content! */
    /* 【错误】strtok_s()会修改函数入参 */
    return tok;
}
```

推荐做法：将函数的入参加上 const，这样编译器就会产生一个告警提示程序员在操作前应该做一份拷贝。

```
char *get_second_token(const char *input) /* 【修改】在函数入参加入 const */
{
    /* do dome strtok_s() that overwrites input's content! */
    return tok;
}
```

参考资料

1. Robert C.Seacord. The Cert C Secure Coding Standard. Pearson Education, 2009

2. CERT C++ Secure Coding Standard. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637

3. Robert C.Seacord. Secure Coding in C and C++. Addison Wesley Professional, 2005

附录 A

下表中列出了几种常用数据库中可能导致 SQL 注入的特殊字符以其转义序列：

数据库	特殊字符	描述	转义序列
Oracle	%	百分号：任意字符（>=0）	/% escape '/'
	_	下划线：任何单字节字符	/_ escape '/'
	/	斜划线：转义字符	// escape '/'
	'	单引号	''
MySQL	'	单引号	\'
	"	双引号	\"
	\	反斜杠	\\
	%	百分号：任意字符（>=0）	\%
	_	下划线：任意单字节字符	_
DB2	'	单引号	''
	;	分号	.
SQL Server	'	单引号	''
	[	左方括号：转义字符	[[]
	_	下划线:任意字符	[_]
	%	百分号：任意字符（>=0）	[%]
	^	插入符号：排除下列字符	[^]

附录 B

下表中列出了 shell 脚本中可能导致命令注入的特殊字符：

分类	符号	功能描述
管道	|	连结上个指令的标准输出，作为下个指令的标准输入。
内联命令	;	连续指令符号
	&	单一个& 符号，且放在完整指令列的最后端，即表示将该指令列放入后台中工作。
逻辑操作符	$	变量替换(Variable Substitution)的代表符号。
	&&	代表与（and）逻辑的符号。
	||	代表或（or）逻辑的符号。
重定向操作	>	将命令输出写入到目标文件中。
	>>	将命令输出附加到目标文件中。
	<	将目标文件的内容发送到命令当中。
表达式	${}	变量的正规表达式。
反引号	` `	返回当前执行命令的结果。
倒斜线	\	在交互模式下的 escape 字元，有几个作用；放在指令前，有取消 aliases 的作用；放在特殊符号前，则该特殊符号的作用消失；放在指令的最末端，表示指令连接下一行。
引号	“”	被双引号括住的内容将被视为单一字符串。但是对$，\，`和”不起作用。
	‘’	被单引号括住的内容，将被视为单一字符串。
括号	()	用括号将一串连续指令括起来。
	[]	常出现在流程控制中，扮演括住判断式的作用。
双分号	;;	case 语句中担任结束符。
文件目录	~	账户的 home 目录。
	.	一个 dot 代表当前目录，两个 dot 代码上层目录。
文件名扩展	?	在文件名扩展(Filename expansion)上扮演的角色是匹配一个任意的字元，但不包含 null 字元。
	*	在文件名扩展(Filename expansion)上，她用来代表任何字元，包含 null 字元。

附录 C

不同平台下危险函数及替代函数。

分类	Linux		Dopra/VRA		Windows	
	危险函数	安全替代（使用安全函数库）	危险函数	安全替代	危险函数	Safe CRT 替代 (VS2005 以上版本支持)
内存拷贝	*memcpy*	*memcpy_s*	*VOS_MemCpy* *VOS_Mem_Copy* *VOS_ChkMemCpy* *_VOS_MemCpy*	*VOS_memcpy_s*	*memcpy*, *RtlCopyMemory*, *CopyMemory*	*memcpy_s*
	wmemcpy	*wmemcpy_s*			*wmemcpy*	*wmemcpy_s*
	memmove	*memmove_s*	*VOS_ChkMemMove* *_VOS_MemMove*	*VOS_memmove_s*	*memmove*	*memmove_s*
	wmemmove	*wmemmove_s*			*wmemmove*	*wmemove_s*
内存初始化	*memset*	*memset_s*	*VOS_MemSet* *VOS_ChkMemSet* *VOS_Mem_Set* *VOS_Mem_Zero* *_VOS_MemSet* *_VOS_Bzero*	*VOS_memset_s*	*memset*	-
字符串复制	*strcpy*	*strcpy_s*	*VOS_StrCpy* *VOS_strcpy* *VOS_ChkStrCpy* *_VOS_StrCpy*	*VOS_strcpy_s*	*strcpy*, *strcpyA*, *strcpyW*, *_tcscpy*, *_mbscpy*, *StrCpy*, *StrCpyA*, *StrCpyW*, *lstrcpy*, *lstrcpyA*, *lstrcpyW*, *_tccpy*, *_mbccpy*, *_ftcscpy*,	*strcpy_s*
	wcscpy	*wcscpy_s*			*wcscpy*	*wcscpy_s*
	strncpy	*strncpy_s*	*VOS_StrNCpy* *VOS_strncpy* *VOS_ChkStrNC*	*VOS_strncpy_s*	*strncpy*, *_tcsncpy*, *_mbsncpy*,	*strncpy_s*

			py *_VOS_StrNCpy*		*_mbsnbcpy* *, StrCpyN,* *StrCpyNA,* *StrCpyNW,* *StrNCpy,* *strcpynA,* *StrNCpyA,* *StrNCpyW,* *lstrcpyn,* *lstrcpynA* *,* *lstrcpynW* *,* *_fstrncpy*	
	wcsncpy	*wcsncpy_* *s*			*wcsncpy*	*wcsncpy_s*
字符串连接	*strcat*	*strcat_s*	*VOS_StrCat* *VOS_strcat* *VOS_ChkStrCa* *t* *_VOS_StrCat*	*VOS_strcat_s*	*strcat,* *strcatA,* *strcatW,* *_tcscat,* *_mbscat,* *StrCat,* *StrCatA,* *StrCatW,* *lstrcat,* *lstrcatA,* *lstrcatW,* *StrCatBuf* *f,* *StrCatBuf* *fA,* *StrCatBuf* *fW,* *StrCatCha* *inW,* *_tccat,* *_mbccat,* *_ftcscat*	*strcat_s*
	wcscat	*wcscat_s*			*wcscat*	*wcscat_s*
	strncat	*strncat_* *s*	*VOS_StrNCat* *VOS_ChkStrNC* *at* *_VOS_StrNCat*	*VOS_strncat_* *s*	*strncat,_* *tcsncat,* *_mbsncat,* *_mbsnbcat* *, StrCatN,*	*strncat_s*

					StrCatNA, StrCatNW, StrNCat, StrNCatA, StrNCatW, lstrncat, lstrcatnA, lstrcatnW, lstrcatn, _fstrncat	
	wcsncat	wcsncat_s			wcsncat	wcsncat_s
格式化输出	sprintf	sprintf_s	VOS_sprintf VOS_sprintf_Safe	VOS_sprintf_s	sprintfW, sprintfA, sprintf, _stprintf, wsprintf, wsprintfW, wsprintfA, wvsprintf, wvsprintfA, wvsprintfW	sprintf_s
	swprintf	swprintf_s			swprintf	swprintf_s
	vsprintf	vsprintf_s	VOS_vsprintf VOS_nvsprintf	VOS_vsprintf_s	vsprintf, _vstprintf	vsprintf_s
	vswprintf	vswprintf_s			vswprintf	vswprintf_s
	snprintf	snprintf_s	VOS_snprintf VOS_nsprintf	VOS_snprintf_s	_snprintf, sntprintf, _sntprintf, nsprintf	_snprintf_s _snwprintf_s sprintf_s

					_snwprintf wnsprintf, wnsprintfA, wnsprintfW	
	vsnprintf	vsnprintf_s			_vsnprintf, _vsnwprintf, _vsntprintf, wvnsprintf, wvnsprintfA, wvnsprintfW	vsnprintf_s
格式化输入	scanf	scanf_s		VOS_scanf_s	scanf, _tscanf, _stscanf snscanf, snwscanf, _sntscanf	scanf_s _snscanf_s
	wscanf	wscanf_s			wscanf	wscanf_s
	vscanf	vscanf_s		VOS_vscanf_s	vscanf	vscanf_s
	vwscanf	vwscanf_s			vwscanf	vwscanf_s
	fscanf	fscanf_s		VOS_fscanf_s	fscanf	fscanf_s
	fwscanf	fwscanf_s			fwscanf	fwscanf_s
	vfscanf	vfscanf_s		VOS_vfscanf_s	vfscanf	vfscanf_s
	vfwscanf	vfwscanf_s			vfwscanf	vfwscanf_s
	sscanf	sscanf_s	VOS_sscanf	VOS_sscanf_s	sscanf	sscanf_s
	swscanf	swscanf_s			swscanf	swscanf_s

	vsscanf	*vsscanf_s*	*vos_vsscanf*	*vos_vsscanf_s*	*vsscanf*	*vsscanf_s*
	vswscanf	*vswscanf_s*			*vswscanf*	*vswscanf_s*
标准输入	*gets*	*gets_s*		*VOS_gets_s*	*gets*, *_getts*, *_gettws*	*gets_s*

附录 D

POSIX

下表中所列的均为异步信号安全函数，来自 POSIX 标准。应用程序可以在信号处理程序中调用这些异步安全函数。

_Exit()	*fexecve()*	*posix_trace_event()*	*sigprocmask()*
_exit()	*fork()*	*pselect()*	*sigqueue()*
abort()	*fstat()*	*pthread_kill()*	*sigset()*
accept()	*fstatat()*	*pthread_self()*	*sigsuspend()*
access()	*fsync()*	*pthread_sigmask()*	*sleep()*
aio_error()	*ftruncate()*	*raise()*	*sockatmark()*
aio_return()	*futimens()*	*read()*	*socket()*
aio_suspend()	*getegid()*	*readlink()*	*socketpair()*
alarm()	*geteuid()*	*readlinkat()*	*stat()*
bind()	*getgid()*	*recv()*	*symlink()*
cfgetispeed()	*getgroups()*	*recvfrom()*	*symlinkat()*
cfgetospeed()	*getpeername()*	*recvmsg()*	*tcdrain()*
cfsetispeed()	*getpgrp()*	*rename()*	*tcflow()*
cfsetospeed()	*getpid()*	*renameat()*	*tcflush()*
chdir()	*getppid()*	*rmdir()*	*tcgetattr()*
chmod()	*getsockname()*	*select()*	*tcgetpgrp()*
chown()	*getsockopt()*	*sem_post()*	*tcsendbreak()*
clock_gettime()	*getuid()*	*send()*	*tcsetattr()*
close()	*kill()*	*sendmsg()*	*tcsetpgrp()*
connect()	*link()*	*sendto()*	*time()*
creat()	*linkat()*	*setgid()*	*timer_getoverrun()*
dup()	*listen()*	*setpgid()*	*timer_gettime()*
dup2()	*lseek()*	*setsid()*	*timer_settime()*
execl()	*lstat()*	*setsockopt()*	*times()*
execle()	*mkdir()*	*setuid()*	*umask()*
execv()	*mkdirat()*	*shutdown()*	*uname()*
execve()	*mkfifo()*	*sigaction()*	*unlink()*
faccessat()	*mkfifoat()*	*sigaddset()*	*unlinkat()*
fchdir()	*mknod()*	*sigdelset()*	*utime()*

fchmod()	*mknodat()*	*sigemptyset()*	*utimensat()*
fchmodat()	*open()*	*sigfillset()*	*utimes()*
fchown()	*openat()*	*sigismember()*	*wait()*
fchownat()	*pause()*	*signal()*	*waitpid()*
fcntl()	*pipe()*	*sigpause()*	*write()*
fdatasync()	*poll()*	*sigpending()*	

OpenBSD

OpenBSD singal()手册列出了少量异步安全函数，但是这些函数其它平台下可能不是安全的。这些函数包括：snprintf()， vsnprintf()和 syslog_r()函数（只有当 syslog_data 结构体初始化为本地变量的情况下才可以）。