华为C&C++语言安全编程规范

Huawei C&C++ Secure Coding Standard

V3.1

目录

0 前言

目的

本规范旨在加强编程人员在编程过程中的安全意识，建立编程人员的攻击者思维，养成安全编码的习惯，编写出安全可靠的代码。

适用范围

C/C++语言编程人员都应遵循本规范所规定的内容。

攻击者思维

编程过程中应该时刻保持以下的假设：

1. 程序所处理的所有外部数据都是不可信的攻击数据
2. 攻击者时刻试图监听、篡改、破坏程序运行环境、外部数据

安全编码基本思想

基于以上的假设，得出安全编码基本思想：

1. 程序在处理外部数据时必须经过严格的合法性校验 编程人员在处理外部数据过程中必须时刻保持这种思维意识，不能做出任何外部数据符合预期的假设，外部数据必须经过严格判断后才能使用。编码人员必须在这种严酷的攻击环境下通过遵守这一原则保证程序的执行过程符合预期结果。
2. 尽量减少代码的攻击面 代码的实现应该尽量简单，避免与外部环境做多余的数据交互，过多的攻击面增加了被攻击的概率，尽量避免将程序内部的数据处理过程暴露到外部环境。

3. 通过防御性的编码策略来弥补潜在的编码人员的疏忽 粗心是人类的天性。由于外部环境的不确定性，以及编码人员的经验、习惯的差异，代码的执行过程很难达到完全符合预期设想的情况。因此在编码过程中必须采取防御性的策略，尽量缓解由于编码人员疏忽导致的缺陷。这些措施包括：

- 变量声明应该赋予初值
- 谨慎使用全局变量
- 禁用功能复杂、易用错的函数
- 禁用易用错的编译器/操作系统的机制
- 小心处理资源访问过程
- 不要改变操作系统的运行环境（创建临时文件、修改环境变量、创建进程等）
- 严格的错误处理
- 合理使用调试断言（ASSERT）

外部数据定义

- 文件（包括程序的配置文件）
- 注册表
- 网络
- 环境变量
- 命令行
- 用户输入（包括命令行、界面）
- 用户态数据（对于内核程序）
- 进程间通信（包括管道、消息、共享内存、socket、RPC等）
- 函数参数（对于API）
- 全局变量（在本函数内，其他线程会修改全局变量）

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。

例外：指规范不适用的某些特殊场景。"规则"的例外应该是极少的。

(**特别说明：为聚焦于每项规则及建议内容重点表达的内容，示例代码中通常省略了安全函数的返回值检查以及其他与重点表述内容无关的检查**)

1 基础要求

1.1 变量

规则1.1.1：指针变量、表示资源描述符的变量、BOOL变量声明必须赋予初值

变量声明赋予初值，可以避免由于编程人员的疏忽导致的变量未初始化引用。

示例：

```
SOCKET s = INVALID_SOCKET;
unsigned char *msg = NULL;
BOOL success = FALSE;
int fd = -1;
```

以下代码，由于变量声明未赋予初值，在最后free的时候出错。

```
char *message;  // 错误！必须声明为 char *message = NULL;
...
if (condition) {
    message = (char *)malloc(len);
    ...
}
...
if (message != NULL) {
    free(message);      //如果condition未满足，会造成free未初始化的内存。
}
```

例外1： 对全局变量，静态变量，在编译阶段自动初始化为0或者等于NULL，不用在定义时强制初始化。 例如：

```
OS_SEC_BSS TICK_ENTRY_FUNC    g_pfnTickTaskEntry;
OS_SEC_BSS volatile UINT64    g_ullSleepTime;
OS_SEC_BSS volatile UINT64    g_ullSleepBegin;
OS_SEC_BSS volatile UINT64    g_ullSleepEnd;
```

相关指南:

CERT.EXP33-C. Do not read uninitialized memory

MISRA.C.2004. Rule 9.1 (required): All automatic variables shall have been assigned a value before being used.

规则1.1.2：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

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。

示例：

```
SOCKET s = INVALID_SOCKET;
unsigned char *msg = NULL;
int fd = -1;
...
closesocket(s);
```

```
6.  s = INVALID_SOCKET;
7.  ...
8.  free(msg);
9.  msg = (unsigned char *)malloc(...);  //msg变量又被赋予新值
10. ...
11. close(fd);
12. fd = -1;
13. ...
```

相关指南：

CERT.MEM01-C. Store a new value in pointers immediately after free()

规则1.1.3：类的成员变量必须在构造函数中赋予初值

如果类中声明了变量，则必须在构造函数中对变量进行赋值。

示例：

```
1.  class CMsg {
2.  public:
3.      CMsg();
4.      ~CMsg();
5.  protected:
6.      int size;
7.      unsigned char *msg;
8.  };
9.
10. CMsg::CMsg()
11. {
12.     size = 0;
13.     msg = NULL;
14. }
```

规则1.1.4：严禁对指针变量进行sizeof操作

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。 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。

```
1.  char *buffer = (char *)malloc(size);
2.  char path[MAX_PATH] = {0};
3.  ...
4.  memset(path, 0, sizeof(path));
5.  memset(buffer, 0, sizeof(buffer));
```

如果要判断当前的指针类型大小，请使用sizeof(char *)的方式。

相关指南：

CERT.ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array

建议1.1.1：尽量使用const

在变量声明前加const关键字，表示该变量不可被修改，这样就可以利用编译器进行类型检查，将代码的权限降到更低。 例如下面是不好的定义：

```
1.  float pi = 3.14159f;
```

应当这样定义：

```
1.  const float PI = 3.14159f;
```

相关指南：

CERT.DCL00-C. Const-qualify immutable objects

建议1.1.2：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

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。 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。

```
1.  ItemList *g_list = NULL;
2.
3.  ItemList *SearchList(const char *name)
4.  {
5.      Lock();
6.
7.      ItemList *p = g_list;
8.
9.      while (p != NULL) {
10.         if (strcmp(p->name, name) == 0) {
11.             break;
12.         }
13.         p = p->next;
14.     }
15.
16.     UnLock();
17.
18.     return p;
19. }
```

性能敏感的代码，请考虑采用原子操作或者无锁算法。

相关指南：

CERT.CON43-C. Do not allow data races in multithreaded code

MITRE.CWE-366, Race condition within a thread

建议1.1.3：同一个函数内，局部变量所占用的空间不要过大

程序在运行期间，函数内的局部变量保存在栈中，栈的大小是有限的。如果申请过大的静态数组，可能导致出现运行出错。 建议在申请静态数组的时候，大小不超过0x1000。 下面的代码，buff申请过大，导致栈空间不够，程序发生stackoverflow异常。

```
#define MAX_BUFF 0x1000000
int Foo()
{
    char buff[MAX_BUFF] = {0};
    ...
}
```

相关指南：

CERT.MEM05-C. Avoid large stack allocations

1.2 断言(ASSERT)

断言是一种除错机制，用于验证代码是否符合编码人员的预期。编码人员在开发期间应该对函数的参数、代码中间执行结果合理地使用断言机制，确保程序的缺陷尽量在测试阶段被发现。 断言被触发后，说明程序出现了不应该出现的严重错误，程序会立即提示错误，并终止执行。 断言必须用宏进行定义，只在调试版本有效，最终发布版本不允许出现assert函数，例如：

```
#include <assert.h>
#ifdef DEBUG
#define ASSERT(f)  assert(f)
#else
#define ASSERT(f)  ((void)0)
#endif
```

下面的函数VerifyUser，上层调用者会保证传进来的参数是合法的字符串，不可能出现传递非法参数的情况。因此，在该函数的开头，加上4个ASSERT进行校验。

```
BOOL VerifyUser(const char *userName, const char *password)
{
    ASSERT(userName != NULL);
    ASSERT(strlen(userName) > 0);
    ASSERT(password != NULL);
    ASSERT(strlen(password) > 0);
    ...
}
```

以下的switch，由于不可能出现default的情况，所以在default处直接调用ASSERT：

```
enum {
    COLOR_RED = 1,
    COLOR_GREEN,
    COLOR_BLUE
};
...
switch (color) {
```

```
    case COLOR_RED:
    ...
    case COLOR_GREEN:
    ...
    case COLOR_BLUE:
    ...
    default: {
        ASSERT(0);
    }
}
```

以下代码，SendMsg是CMsg类的成员函数，socketID是成员变量，在调用SendMsg的时候必须保证socketID已经被初始化，因此在此处用ASSERT判断socketID的合法性。

```
CMsg::CMsg()
{
    socketID = INVALID_SOCKET;
}
int CMsg::SendMsg(const char *msg, int len)
{
    ASSERT(socketID != INVALID_SOCKET);
    ...
    ret = send(socketID, msg, len, 0);
    ...
}
```

在linux内核中定义ASSERT宏，可以采用如下方式：

```
#ifdef DEBUG
#define ASSERT(f)  BUG_ON(!(f))
#else
#define ASSERT(f)  ((void)0)
#endif
```

相关指南：

CERT.MSC11-C. Incorporate diagnostic tests using assertions

规则1.2.1：断言必须使用宏定义，禁止直接调用系统提供的assert()

断言只能在调试版使用，断言被触发后，程序会立即退出，因此严禁在正式发布版本使用断言，请通过编译选项进行控制。 错误用法如：

```
int Foo(int *array, int size)
{
    assert(array != NULL);
    ...
}
```

规则1.2.2：运行时可能会导致的错误，严禁使用断言

断言不能用于校验程序在运行期间可能导致的错误。 以下代码的所有ASSERT的用法是错误的。

```
FILE *fp = fopen(path, "r");
ASSERT(fp != NULL);     //文件有可能打开失败
char *str = (char *)malloc(MAX_LINE);
ASSERT(str != NULL);    //内存有可能分配失败
ReadLine(fp, str);
char *p = strstr(str, 'age=');
ASSERT(p != NULL);      //文件中不一定存在该字符串
int age = atoi(p+4);
ASSERT(age > 0);        //文件内容不一定符合预期
```

规则1.2.3：严禁在断言内改变运行环境

在程序正式发布阶段，断言不会被编译进去，为了确保调试版和正式版的功能一致性，严禁在断言中使用任何赋值、修改变量、资源操作、内存申请等操作。 例如，以下的断言方式是错误的：

```
ASSERT(p1 = p2);        //p1被修改
ASSERT(i++ > 1000);     //i被修改
ASSERT(close(fd) == 0);//fd被关闭
```

建议1.2.1：不要将多条语句放在同一个断言中

为了更加准确地发现错误的位置，每一条断言只校验一个条件。 下面的断言同时校验多个条件，在断言触发的时候，无法判断到底是哪一个条件导致的错误：

```
int Foo(int *array, int size)
{
    ASSERT(array != NULL && size > 0 && size < MAX_SIZE);
    ...
}
```

应该将每个条件分开：

```
int Foo(int *array, int size)
{
    ASSERT(array != NULL);
    ASSERT(size > 0);
    ASSERT(size < MAX_SIZE);
    ...
}
```

1.3 函数

规则1.3.1：数组作为函数参数时，必须同时将其长度作为函数的参数

通过函数参数传递数组或一块内存进行写操作时，函数参数必须同时传递数组元素个数或所传递的内存块大小，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。 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(BYTE *msg)
{
    ...
}
...
size_t len = ...
BYTE *msg = (BYTE *)malloc(len); //此处分配的内存块等同于字节数组
...
ParseMsg(msg);
...
```

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(BYTE *msg, size_t msgLen)
{
    ASSERT(msg != NULL);
    ASSERT(msgLen != 0);
    ...
}
...
size_t len = ...
BYTE *msg = (BYTE *)malloc(len);
...
ParseMsg(msg, len);
...
```

下面的代码，msg是固定长度的数组，也必须将数组大小作为函数的参数：

```
int ParseMsg(BYTE *msg, size_t msgLen)
{
    ASSERT(msg != NULL);
    ASSERT(msgLen != 0);
    ...
}
...
BYTE msg[MAX_MSG_LEN] = {0};
...
ParseMsg(msg, sizeof(msg));
...
```

对于const char *类型的参数，它的长度是通过'\0'的位置计算出来，不需要传长度参数。

```
int SearchName(const char *name)
{
    ...
}
...
char *name = getName(...);
...
int ret = SearchName(name);
...
```

如果参数是char *,且参数作为写内存的缓冲区，那么必须传入其缓冲区长度。如：

```
int SaveName(char *name, size_t len, const char *inputName)
{
    ...
    ret = strcpy_s(name, len, inputName);
    ...
}
...
char name[NAME_MAX] = {0};
...
int ret = SaveName(name, sizeof(name), inputName);
...
```

如果函数仅对字符串中的特定字符进行一对一替换，或者删除字符串中的特定字符，这时对字符数组的访问不会超过原字符串边界，因此这类函数不需要传待修改的字符串长度。

```
void FormatPathSeparator(char *path)
{
    unsigned int i = 0;

    if (path == NULL) {
        return;
    }

    while (path[i] != '\0') {
        if ('\\' == path[i]) {
            path[i] = '/';
        }
        i++;
    }
}
```

例外1： 对于const struct *类型的数组入参，如果它的长度可以通过特定元素值判断结尾，那么可以不传递结构体数组的长度。

```
struct DevType {
    int vendorID;
    int deviceID;
    int subDevice;
};
...
const struct DevType cardIds[] = {
    { CARD_ID, PCI_DEV_T1, PCI_ANY_ID },
    { CARD_ID, PCI_DEV_T2, PCI_ANY_ID },
    { CARD_ID, PCI_DEV_T3, PCI_ANY_ID },
    { 0, }
};
...
int BuildCardQueue(const struct DevType *cards)
{
```

```
    int index = 0;

    ASSERT(cards != NULL);
    while (cards[index]->vendorID != 0) {
        ...
    }
    ...
}
```

例外2： 对固定长度的数组，如果用数组的头地址作为子函数参数，由于性能原因，可以不用传递其长度。 下例中，EtherAddrCopy()函数仅用于MAC地址的赋值，不会用于其他地方，且长度是可保证的，其拷贝使用的下标（或偏移）没有外部数据的影响。由于性能高度敏感，因此这里没有传入目的缓冲区dst的长度。

```
#define ETH_ALEN 6

static const u8 ethReservedAddrBase[ETH_ALEN] = {...};
...
void EtherAddrCopy(unsigned char *dst, const unsigned char *src)
{
    dst[0] = src[0];
    dst[1] = src[1];
    dst[2] = src[2];
    dst[3] = src[3];
    dst[4] = src[4];
    dst[5] = src[5];
}

int AddDevice()
{
    unsigned char mac[ETH_ALEN];
    ...
    EtherAddrCopy(mac, ethReservedAddrBase);
    ...
}
```

相关指南：

CERT.ARR38-C. Guarantee that library functions do not form invalid pointers

CERT.API00-C. Functions should validate their parameters

MITRE. CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer

MITRE.CWE-121, Stack-based Buffer Overflow

MITRE.CWE-123, Write-what-where Condition

MITRE.CWE-125, Out-of-bounds Read

MITRE.CWE-805, Buffer Access with Incorrect Length Value

规则1.3.2：严禁对公共接口API函数的参数进行ASSERT操作

对于设计成API的函数，必须对参数进行合法性判断，严禁在API实现过程中产生CRASH。对API函数的参数进行ASSERT操作是没有意义的。 例如，对于提供应用服务器IP的平台公共API接口这样实现是错误的：

```
1.  int GetServerIP(char *ip, size_t ipSize)
2.  {
3.      ASSERT(ip != NULL);
4.      ...
5.  }
```

公共接口API应当对输入参数进行代码检查：

```
1.  int GetServerIP(char *ip, size_t ipSize)
2.  {
3.      if (ip == NULL) {
4.          ...
5.      }
6.      ...
7.  }
```

规则1.3.3：不对内容进行修改的指针型参数，定义为const

如果参数是指针型参数，且内容不会被修改，请定义为const类型。

示例:

```
1.  int Foo(const char *filePath)
2.  {
3.      ...
4.      int fd = open(filePath, ...);
5.      ...
6.  }
```

相关指南：

CERT.DCL13-C Declare function parameters that are pointers to values not changed by the function as const

MISRA.C.2004.Rule 16.7 (advisory): A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.

建议1.3.1：谨慎使用不可重入函数

不可重入函数在多线程环境下其执行结果不能达到预期效果，需谨慎使用。常见的不可重入函数包括：

rand, srand

getenv, getenv_s

strtok

strerror

asctime, ctime, localtime, gmtime

setlocale

atomic_init

tmpnam

mbrtoc16, c16rtomb, mbrtoc32, c32rtomb

gethostbyaddr

gethostbyname

inet_ntoa

建议1.3.2：字符串或指针作为函数参数时，请检查参数是否为NULL

如果字符串或者指针作为函数参数，为了防止空指针引用错误，在引用前必须确保该参数不为NULL，如果上层调用者已经保证了该参数不可能为NULL，在调用本函数时，在函数开始处可以加ASSERT进行校验。 例如下面的代码，因为BYTE *p有可能为NULL,因此在使用前需要进行判断。

```
1.  int Foo(int *p, int count)
2.  {
3.      if (p != NULL && count > 0) {
4.          int c = p[0];
5.      }
6.      ...
7.  }
8.
9.  int Foo2()
10. {
11.     int *arr = ...
12.     int count = ...
13.     Foo(arr, count);
14.     ...
15. }
```

下面的代码，由于p的合法性由调用者保证，对于Foo函数，不可能出现p为NULL的情况，因此加上ASSERT进行校验。

```
1.  int Foo(int *p, int count)
2.  {
3.      ASSERT(p != NULL); //ASSERT is added to verify p.
4.      ASSERT(count > 0);
5.      int c = p[0];
6.      ...
7.  }
8.
9.  int Foo2()
10. {
11.     int *arr = ...
12.     int count = ...
13.     ...
14.     if (arr != NULL && count > 0) {
15.         Foo(arr, count);
16.     }
17. ...
```

```
18. }
```

建议1.3.3：在函数的开始处对参数进行ASSERT操作（API除外）

ASSERT用于检测代码设计上的错误，如果ASSERT被触发，说明代码的设计不符合编码人员的预期。 在函数的开始处，对参数进行必要的ASSERT操作，可以在测试阶段有效地检验编码人员对代码设计上的预期。

1.4 循环

规则1.4.1：循环必须有退出条件

循环如果没有退出条件，那么程序无法安全退出。 以下代码，在一个大循环内接收外部数据并进行处理，但没有退出条件，会导致该程序无法正常退出。

```
1.  while (TRUE) {
2.      int size = 0;
3.      unsigned char *msg = ReceiveMsg(&size);
4.      if (msg != NULL) {
5.          ParseMsg(msg, size);
6.          FreeMsg(msg);
7.      }
8.  }
```

例外： 1、操作系统软件的IDLE线程，可能需要无限循环 2、操作系统在不可恢复的错误中，为避免更多错误发生，进入指令无限循环。例如：

```
1.  void Reboot(void)
2.  {
3.      reboot(reboot_cmd);
4.
5.      while (1) {
6.          CoreWait();
7.      }
8.  }
```

3、嵌入式设备的操作系统或主流程，可能使用无限循环。例如：

```
1.  void OsTask()
2.  {
3.      ...
4.      while (1)
5.      {
6.          msgHdl = RECEIVE(OS_WAIT_FOREVER, &msgId, &senderPid);
7.          if (msgHdl == 0)
8.          {
9.              continue;
10.         }
11.         switch (msgId)
12.         {
13.             ...
```

```
        }
        (void)FREE(msgHdl);
    }
}
```

相关指南：

MISRA.2004.Rule 13.5 (required): The three expressions of a for statement shall be concerned only with loop control.

MISRA.2004. Rule 14.6 (required): For any iteration statement there shall be at most one break statement used for loop termination.

JPL.2009.Rule 3: Use verifiable loop bounds for all loops meant to be terminating.

1.5 异常机制

规则1.5.1：禁用C++异常机制

严禁使用C++的异常机制，所有的错误都应该通过错误值在函数之间传递并做相应的判断, 而不应该通过异常机制进行错误处理。 编码人员必须完全掌控整个编码过程，建立攻击者思维，增强安全编码意识，主动把握有可能出错的环节。而使用C++异常机制进行错误处理，会削弱编码人员的安全意识。 异常机制会打乱程序的正常执行流程，使程序结构更加复杂，原先申请的资源可能会得不到有效清理。 异常机制导致代码的复用性降低，使用了异常机制的代码，不能直接给不使用异常机制的代码复用。 异常机制在实现上依赖于编译器、操作系统、处理器，使用异常机制，导致程序执行性能降低。 在二进制层面，程序被加载后，异常处理函数增加了程序的被攻击面，攻击者可以通过覆盖异常处理函数地址，达到攻击的效果。 例外： 在接管C++语言本身抛出的异常（例如new失败、STL）、第三方库（例如IDL）抛出的异常时，可以使用异常机制，例如：

```
int len = ...;
char *p = NULL;
try {
    p = new char[len];
}
catch (bad_alloc) {
    ...
    abort();
}
```

相关指南：

Google C++ Style Guide.Exceptions: We do not use C++ exceptions.

1.6 类

规则1.6.1：如果有构造函数，则必须有析构函数

析构函数的目的是为了在类释放的时候做相应的清理工作，以免编码人员忘记资源的清理。

规则1.6.2：构造函数内不能做任何有可能失败的操作

构造函数没有返回值，不能做错误判断，因此在构造函数内，不能做任何有可能失败的操作。 下面的代码中，open、new、ConnectServer都有可能失败，这些操作不应该放在构造函数内。

```
CFoo::CFoo()
{
    int fd = open(...);
    char *str = new char[...];
    BOOL b = ConnectServer(...);
    ...
}
```

规则1.6.3：严禁在构造函数中创建线程

构造函数内仅作成员变量的初始化工作，其他的操作通过成员函数完成。

规则1.6.4：严禁出现 delete this操作

资源的申请和释放必须在同一逻辑层，谁申请，谁释放。

规则1.6.5：如果类的公共接口中返回类的私有数据地址，则必须加const类型

示例：

```
class CMsg {
public:
    CMsg();
    ~CMsg();
    Const unsigned char *GetMsg();
protected:
    int size;
    unsigned char *msg;
};

CMsg::CMsg()
{
    size = 0;
    msg = NULL;
}

const unsigned char *CMsg::GetMsg()
{
    return msg;
}
```

建议1.6.1：尽量避免定义public成员

类成员进行定义的时候，需要考虑类的功能，尽量减少对外接口的暴露。

1.7 安全退出

规则1.7.1：禁用atexit函数

atexit函数注册若干个有限的函数，当exit被调用后，自动调用由atexit事先注册的函数。 当资源不再使用后，编码人员应该立即主动地进行清理，而不应该在最终程序退出后通过事先注册的例程被动地清理。

例外： 作为服务维测监控功能，为定位程序异常退出原因的模块，可以作为例外使用atexit()函数。

规则1.7.2：严禁调用kill、TerminateProcess函数终止其他进程

调用kill、TerminateProcess等函数强行终止其他进程(如kill -9)，会导致其他进程的资源得不到清理。 对于进程间通信，应该主动发送一个停止命令，通知对方进程安全退出。 当发送给对方进程退出信号后，在等待一定时间内如果对方进程仍然未退出，可以调用kill、TerminateProcess函数。

示例：

```
1.  if (WaitForRemoteProcessExit(...) == TIME_OUT) {
2.      kill(...);     //目标进程在限定时间内仍然未退出，强行结束目标进程
3.  }
```

规则1.7.3：禁用pthread_exit、ExitThread函数

严禁在线程内主动终止自身线程，线程函数在执行完毕后会自动、安全地退出。主动终止自身线程的操作，不仅导致代码复用性变差，同时容易导致资源泄漏错误。

建议1.7.1：禁用exit、ExitProcess函数（main函数除外）

程序应该安全退出，除了main函数以外，禁止任何地方调用exit、ExitProcess函数退出进程。直接退出进程会导致代码的复用性降低，资源得不到有效地清理。 程序应该通过错误值传递的机制进行错误处理。 以下代码加载文件，加载过程中如果出错，直接调用exit退出：

```
1.  void LoadFile(const char *filePath)
2.  {
3.      FILE* fp = fopen(filePath, "rt");
4.      if (fp == NULL) {
5.          exit(0);
6.      }
7.      ...
8.  }
```

正确的做法应该通过错误值传递机制，例如：

```
1.  BOOL LoadFile(const char *filePath)
2.  {
3.      BOOL ret = FALSE;
4.      FILE* fp = fopen(filePath, "rt");
5.      if (fp != NULL) {
6.          ...
7.      }
8.      ...
9.      return ret;
10. }
```

建议1.7.2：禁用abort函数

abort会导致程序立即退出，资源得不到清理。 例外： 只有发生致命错误，程序无法继续执行的时候，在错误处理函数中使用abort退出程序，例如：

```
1.  void FatalError(int sig)
2.  {
3.      abort();
4.  }
5.  int main(int argc, char *argv[])
6.  {
7.      signal(SIGSEGV, FatalError);
8.      ...
9.  }
```

2 字符串/数组操作

规则2.1：确保有足够的存储空间

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。典型函数如itoa，realpath。 以下的代码，试图将数字转为字符串，但是目标存储空间的长度不足。

```
1.  int num = ...
2.  char str[8] = {0};
3.  itoa(num, str, 10);  // 10进制整数的最大存储长度是12个字节
```

以下的代码，试图将路径标准化，但是目标存储空间的长度不足。

```
1.  char resolvedPath[100] = {0};
2.  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
```

以下的代码，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，是正确的做法。

```
1.  char *msg = GetMsg();
2.  ...
3.  char name[MAX_NAME] = {0};
4.  int i=0;
5.  //必须考虑msg不包含预期的字符'\n'
6.  while (*msg != '\0' && *msg != '\n' && i < sizeof(name) - 1) {
7.      name[i++] = *msg++;
8.  }
9.  name[i] = '\0';    //保证最后有'\0'
```

相关指南：

CERT.STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

CERT.STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator

规则2.2：对字符串进行存储操作，确保字符串有'\0'结束符

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。

相关指南:

CERT.STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

CERT.STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator

规则2.3：外部数据作为数组索引时必须确保在数组大小范围内

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，否则为导致严重的错误。 下面的代码，通过if语句判断offset的合法性：

```
1.  int Foo(BYTE *buffer, int size)
2.  {
3.      ...
4.      int offset = ReadIntFromMsg();
5.      if (offset >= 0 && offset < size) {
6.          BYTE c =  buffer[offset];
7.          ...
8.      }
9.      ...
10. }
```

相关指南：

MITRE.CWE-129: Improper Validation of Array Index

规则2.4：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

在调用内存操作相关的函数时（例如memcpy、memmove、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。 下例中，循环长度来自设备外部报文，由于没有校验大小，可造成缓冲区溢出：

```
1.  typedef struct BigIntType {
2.      unsigned int length;
3.      char val[MAX_INT_DIGITS];
4.  }BigInt;
5.
6.  BigInt *AsnOctsToBigInt(const AsnOcts *asnOcts)
7.  {
8.      BigInt *bigNumber = NULL;
```

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

规则2.5：调用格式化函数时，禁止format参数由外部可控

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。 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()。

错误示例:

```
char *msg = GetMsg();
...
printf(msg);
```

推荐做法：

```
char *msg = GetMsg();
...
printf("%s\n", msg);
```

相关指南：

CERT.FIO47-C. Use valid format strings

MITRE.CWE-134: Use of Externally-Controlled Format String

规则2.6：调用格式化函数时，format中参数的类型与个数必须与实际参数类型一致

格式化函数中，Format中的参数类型与个数与实际参数不一致，会导致读写的数据与期望不一致，造成输入输出数据错误，访问非法内存，写越界等问题。 例外： 在格式化输出函数中，signed char，signed short可以用有符号整型格式%d输出，unsigned char, unsigned short类型参数可以用无符号整型格式如%u输出。

3 正确使用安全函数

规则 3.1：正确设置安全函数中的destMax参数

安全函数的destMax参数设置应当准确，有效，并满足良好的编码风格要求。详细指导可参考《安全函数正确使用指导规范》中”destMax参数设置原则”部分。

相关指南：

CERT.EXP09-C. Use sizeof to determine the size of a type or variable

CSEC Reports

规则 3.2：禁止不正确地重定义或封装安全函数

禁止直接使用宏的方式重定义安全函数时，或以函数封装的形式重新包装安全函数或危险函数时，忽略安全函数的destMax参数，或用count参数直接代替destMax参数。

错误示例1：使用类似危险函数的接口封装安全函数，destMax与count参数使用相同参数

```
1.  IPC_VOID *XXX_memcpy(void *dest, const void *src, size_t count)
2.  {
3.      ...
4.
5.      memcpy_s(dest,  count,  src, count);
6.      ...
7.  }
```

错误示例2：使用类似安全函数的接口封装安全函数，调用安全函数时，忽略了destMax入参，调用安全函数时destMax与count参数使用相同参数

```
1.  errno_t XXX_memcpy_s(void *dest, size_t destMax, const void *src, size_t count)
2.  {
3.      ...
4.      memcpy_s(dest,  count,  src, count);
5.      ...
6.  }
```

错误示例3：宏定义伪安全函数，名称为安全函数，对应的函数实际不是安全函数

```
1.  #define my_snprintf_s snprintf
```

错误示例4：使用类似危险函数接口的宏，重定义安全函数，destMax与count参数使用相同参数

```
1.  #define XXX_memcpy(dest, src, count) memcpy_s(dest, count, src, count)
```

错误示例5：使用类似安全函数接口的宏，重定义安全函数，destMax与count参数使用相同参数

```
1.  #define XXX_memcpy_s(dest, destMax, src, count) memcpy_s(dest, count, src,count)
```

错误示例6：使用类安全函数接口的宏，重定义危险函数，忽略了destMax参数

```
#define XXX_memcpy(dest, destMax, src, count) memcpy(dest, src, count)
```

错误示例7：使用类似安全函数的名字的宏或者函数，但是参数与安全函数不同

```
1.  #define XXX_memcpy_s(dest, src, count) ...
2.  errno_t XXX_memcpy_s(void *dest, const void *src, size_t count)
3.  {
4.      ...
5.  }
```

错误示例8：使用自定义危险函数宏或者函数

```
#define XXX_strncpy(dest,destMax,src) ...
errno_t XXX_strncpy(char *dest, size_t destMax, const char *src)
{
    ...
}
```

错误示例9：使用类似安全函数的接口封装危险函数，忽略了destMax参数

```
errno_t XXX_memcpy_s(void *dest, size_t destmax, const void *src, size_t count)
{
    ...
    memcpy(dest, src, count);
    ...
}
```

相关指南：

CSEC Reports

规则 3.3：禁止用宏重命名安全函数

使用宏重命名安全函数不利于静态代码扫描工具（非编译型）定制针对安全函数误用的规则，同时，由于命名风格多样，也不利于提示代码开发者函数的真实用途，容易造成对代码的误解及重命名安全函数的误用。重命名安全函数不会改变安全函数本身的检查能力。

错误示例1：代码中未直接调用安全函数，而是以宏的方式调用安全函数

```
#define XXX_memcpy_s  memcpy_s
#define SEC_MEM_CPY   memcpy_s
#define XX_memset_s(dst, dstMax, val, n) memset_s((dst), (dstMax), (val), (n))
```

相关指南：

CSEC Reports

规则 3.4：禁止自定义安全函数

Huwei Secure C库中提供的安全函数符合C11中已有的定义，并结合产品诉求，在平台兼容性和执行效率上有其独特优势。使用类似名称创建的函数难以与C11中的函数定义相适配，同时也容易与Huwei Secure C的实现产生混淆，造成误用，并引入安全风险。因此不应当实现私有的安全函数。

错误示例1：在下面的拷贝安全函数实现中，由于不恰当的进行了类型转换，因此缺乏在64位系统上的可移植性，同时，该函数的返回值也不兼容C11中的安全拷贝函数memcpy_s。

```
#define SECUREC_MEM_MAX_LEN (0x7fffffffUL)
void MemcpySafe(void *dest, unsigned int destMax, const void *src, unsigned int
count)
{
    if (destMax == 0 || destMax > SECUREC_MEM_MAX_LEN ) {
```

```
        return;
    }
    if (dest == NULL || src == NULL) {
        return;
    }
    if (count > destMax) {
        return;
    }
    if (dest == src) {
        return;
    }
    if (((uint32_t)dest >(uint32_t)src &&(uint32_t) dest <(uint32_t) (void *)
((uint8_t *)src + count)) ||
        ((uint32_t)src > (uint32_t)dest && (uint32_t)src <(uint32_t) (void *)((
uint8_t *)dest + count))) {
        return;
    }
    memcpy(dest, src, count);
    return;
}
```

相关指南：

CSEC Reports

规则 3.5：必须检查安全函数返回值，并进行正确的处理

原则上，如果使用了安全函数，需要进行返回值检查。如果返回值!=EOK, 那么本函数应该立即返回，不能继续执行。 安全函数有多个错误返回值，如果安全函数返回失败，在本函数返回前，根据产品具体场景，可以做如下操作：（1）记录日志（2）返回错误（3）调用abort立即退出程序 例如：

```
BOOL ParseBuff(BYTE *destBuff, size_t destMax)
{
    BYTE *src = ...
    size_t srcLen = ...
    errno_t err = EOK;
    if (destBuff == NULL || destMax == 0) {
        return FALSE;
    }
    err = memcpy_s(destBuff, destMax, src, srcLen);
    if (err != EOK) {
        Log("memcpy_s failed, err = %d\n", err);
        return FALSE;
    }
    ...
    return TRUE;
}
```

例外1： 规则6.6 禁止使用内存操作类危险函数中的例外场景中描述了允许继续使用不安全函数的场景，这些场景对应的代码如果使用了安全函数，可以不进行返回值检查。

例外2： 在安全函数返回值检查错误处理代码中，如果又调用了安全函数，可以不进行返回值检查。 例如，在以下代码中，调用strcpy_s失败后，错误处理代码试图记录日志，在错误处理代码内又调用了sprintf_s安全函数，此时不需要进行返回值检查。（编码人员需要仔细检查该语句不会产生安全问题）在实际的代码中，如果要进行安全函数返回值错误处理，可以将第4-6行进行宏定义封装。

```
1.  errno_t e = strcpy_s(...);
2.  if (e != EOK) {
3.      char buff[MAX_BUFF];
4.      sprintf_s(buff, sizeof(buff), ...);
5.      Log(buff);
6.      return -1;
7.  }
8.  ...
```

相关指南：

CERT.EXP12-C. Do not ignore values returned by functions

CSEC Reports

4 整数

规则4.1：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

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，可能会超过这个最大固定长度，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。

错误示例1：

```
1.  size_t width = ReadByte();
2.  size_t height = ReadByte();
3.  size_t total = width * height;      //可能整数溢出
4.  void *bitmaps = malloc(total);
```

推荐做法1：

```
1.  size_t width = ReadByte();
2.  size_t height = ReadByte();
3.  if (width == 0 || height == 0 || width > MAX_WIDTH || height > MAX_HEIGHT) {
4.      //error
5.      ...
6.  }
7.  size_t total = width * height;  // MAX_WIDTH * MAX_HEIGHT 不会溢出
8.  void *bitmaps = malloc(total);
```

错误示例2：

```
size_t a = ReadByte();
size_t b = 1000 / a;        //a可能是0
size_t c = 1000 % a;        //a可能是0
...
```

推荐做法2：

```
size_t a = ReadByte();
if (a == 0) {
    //error
    ...
}
size_t b = 1000 / a;        //a不可能是0
size_t c = 1000 % a;        //a不可能是0
...
```

相关指南:

CERT.INT30-C. Ensure that unsigned integer operations do not wrap

CERT.INT32-C. Ensure that operations on signed integers do not result in overflow

CERT.INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors

CERT.INT02-C. Understand integer conversion rules

MITRE.CWE-190, Integer Overflow or Wraparound

规则4.2：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

由于整数在运算过程中可能溢出，当运算结果赋值给比他更大类型，或者和比他更大类型进行比较时，会导致实际结果与预期结果不符。 请观察以下二个代码及其输出：

```
int main(int argc, char *argv[])
{
    unsigned int a = 0x10000000;
    unsigned long long b = a * 0xab;
    printf("b = %llX\n", b);
    return 0;
}
```

输出： b = B0000000

```
int main(int argc, char *argv[])
{
    unsigned int a = 0x10000000;
    unsigned long long b = (unsigned long long )a * 0xab;
    printf("b = %llX\n", b);
    return 0;
}
```

输出： b = AB0000000

规则4.3：禁止对有符号整数进行位操作符运算

位操作符（ ~、>>、<<、&、^、|)应该只用于无符号整型操作数。 错误示例：

```
1.  int data = ReadByte();
2.  int a = data >> 24;
```

推荐做法：(为简化示例代码，此处假设ReadByte函数实际不存在返回值小于0的情况)

```
1.  unsigned int data = (unsigned int)ReadByte();
2.  unsigned int a = data >> 24;
```

相关指南：

CERT.INT13-C. Use bitwise operators only on unsigned operands

MISRA.C.2004.Rule 12.7 (required): Bitwise operators shall not be applied to operands whose underlying type is signed.

规则4.4：禁止整数与指针间的互相转化

指针的大小随着平台的不同而不同，强行进行整数与指针间的互相转化，降低了程序的兼容性，在转换过程中可能引起指针高位信息的丢失。

错误示例：

```
1.  char *ptr = ...;
2.  unsigned int number = (unsigned int)ptr;
```

推荐做法：

```
1.  char *ptr = ...;
2.  uintptr_t number = (uintptr_t)ptr;
```

相关指南：

CERT.INT36-C. Converting a pointer to integer or integer to pointer

MISRA.C.2004.Rule 11.3 (advisory): A cast should not be performed between a pointer type and an integral type.

规则4.5：禁止对指针进行逻辑或位运算（&&、||、!、 ~、>>、<<、&、^、|）

对指针进行逻辑运算，会导致指针的性质改变，可能产生内存非法访问的问题。 下面是错误的用法：

```
BOOL dealName(const char *nameA, const char *nameB)
{
    ...
    if (nameA)
    ...
    if (!nameB)
    ...
}
```

下面是正确的用法：

```
BOOL dealName(const char *nameA, const char *nameB)
{
    ...
    if (nameA != NULL)
    ...
    if (nameB == NULL)
    ...
}
```

例外： 为检查地址对齐而对地址指针进行的位运算可以作为例外。

相关指南：

MISRA.C.2004.Rule 12.6 (advisory): The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !, =, ==, != and ?:).

规则4.6：循环次数如果受外部数据控制，需要校验其合法性

如下示例中，由于循环条件受外部输入的报文内容控制，可进入死循环：

```
unsigned char *FindAttr(unsigned char type, unsigned char *msg, size_t inputMsgLen)

{
    ...
    msgLength = ntohs(*(unsigned short *)&msg[RD_LEA_PKT_LENGTH]);
    ...
    while (msgLength != 0) {
        attrType = msg[0];
        attrLength = msg[RD_LEA_PKT_LENGTH];
        ...
        msgLength -= attrLength;
        msg += attrLength;
    }
    ...
}
```

此例中，需要检查报文的实际可读长度，报文内容提供的循环增量（避免为0），以防止缓冲区溢出。

5 内存

规则5.1：内存申请前，必须对申请内存大小进行合法性校验

内存申请的大小可能来自于外部数据，必须检查其合法性，防止过多地、非法地申请内存。不能申请0长度的内存。 例如：

```
1.  int Foo(int size)
2.  {
3.      if (size <= 0) {
4.          //error
5.          ...
6.      }
7.      ...
8.      char *msg = (char *)malloc(size);
9.      ...
10. }
```

相关指南：

CERT.MEM04-C. Beware of zero-length allocations

MITRE.CWE-789: Uncontrolled Memory Allocation

规则5.2：内存分配后必须判断是否成功

```
1.  char *msg = (char *)malloc(size);
2.  if (msg != NULL) {
3.      ...
4.  }
```

相关指南：

CERT.MEM11-C. Do not assume infinite heap space

CERT.ERR33-C. Detect and handle standard library errors

CERT.MEM52-CPP. Detect and handle memory allocation errors

MITRE.CWE 252, Unchecked Return Value

MITRE.CWE 391, Unchecked Error Condition

MITRE.CWE 476, NULL Pointer Dereference

MITRE.CWE 690, Unchecked Return Value to NULL Pointer Dereference

MITRE.CWE 703, Improper Check or Handling of Exceptional Conditions

MITRE.CWE 754, Improper Check for Unusual or Exceptional Conditions

规则5.3：禁止引用未初始化的内存

malloc、new分配出来的内存没有被初始化为0，要确保内存被引用前是被初始化的。 以下代码使用malloc申请内存，在使用前没有初始化：

```
1.  int *CalcMetrixColomn( int **metrix ,int *param, size_t size )
2.  {
3.      int *result = NULL;
4.      ...
5.      size_t bufSize = size * sizeof(int);
6.      ...
7.      result = (int *)malloc(bufSize);
8.      ...
9.      result[0] += metrix[0][0] * param[0];
10.     ...
11.     return result;
12. }
```

以下代码使用memset_s()对分配出来的内存清零。

```
1.  int *CalcMetrixColomn(int **metrix ,int *param, size_t size)
2.  {
3.      int *result = NULL;
4.      ...
5.      size_t bufSize = size * sizeof(int);
6.      ...
7.      result = (int *)malloc(bufSize);
8.      ...
9.      int ret = memset_s(result, bufSize, 0, bufSize);  //【修改】确保内存被初始化后才被引
用
10.     ...
11.     result[0] += metrix[0][0] * param[0];
12.     ...
13.     return result;
14. }
```

相关指南：

CERT.EXP33-C. Do not read uninitialized memory

CERT.EXP53-CPP. Do not read uninitialized memory

规则5.4：内存释放之后立即赋予新值

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新值。 如果一个指针释放后能够马上离开作用域，因为它已经不能被再次访问，因此可以无需对其赋予新值。

示例：

```
char *message = NULL;
...
message = (char *)malloc(len);
...
if (...) {
    free(message);      //在这个分支内对内存进行了释放
    message = NULL;     //释放后将指针赋值为NULL
}
...
if (message != NULL) {
    free(message);
    message = NULL;
}
```

相关指南:

CERT.MEM01-C. Store a new value in pointers immediately after free()

CERT.MEM50-CPP. Do not access freed memory

规则5.5：禁止使用realloc()函数

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。

相关指南:

CERT.MEM36-C. Do not modify the alignment of objects by calling realloc()

规则5.6：禁止使用alloca()函数申请栈上内存

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。

6 不安全函数

规则6.1：禁止外部可控数据作为system、popen、WinExec、ShellExecute、execl, execlp, execle, execv, execvp、CreateProcess等进程启动函数的参数

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。 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。

```
1.  char *msg = GetMsgFromRemote();
2.  system(msg);
```

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：

```
1.  char *msg = GetMsgFromRemote();
2.  sprintf_s(cmd, sizeof(cmd), "dir %s", msg);
3.  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)。

```
1.  int execl(const char *path, const char *arg, ...);
2.  int execlp(const char *file, const char *arg, ...);
3.  int execle(const char *path, const char *arg,..., char * const envp[]);
4.  int execv(const char *path, char *const argv[]);
5.  int execvp(const char *file, char *const argv[]);
```

例如，禁止如下使用方式：

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

windows下可用如下方式使用CreateProcess以避免命令注入:

```
1.  void ProcessDirectory(const char *input_dir )
2.  {
3.      ...
4.      CreateProcess("any_cmd", input_dir, NULL, NULL,
5.                    FALSE, CREATE_DEFAULT_ERROR_MODE, NULL,
6.                    NULL, NULL, NULL);
7.      ...
8.  }
```

使用CreateProcess应注意lpApplicationName参数中的空格问题，需要使用8dot3格式，或者对空格进行转义，同时参数中禁止使用命令解析器(cmd, powershell)。

相关指南:

CERT.ENV33-C. Do not call system()

规则6.2：禁止外部可控数据作为dlopen/LoadLibrary等模块加载函数的参数

这些函数会加载外部模块，如果外部可控数据作为这些函数的参数，有可能会加载攻击者事先预制的模块。 如果要使用这些函数，请使用白名单机制，确保这些函数的参数不受任何外来数据的影响。 以下代码从外部获取数据后直接作为LoadLibrary函数的参数，有可能导致程序被植入木马。

```
1.  char *msg = GetMsgFromRemote();
2.  LoadLibrary(msg);
```

例外： 对于使用了密钥，签名机制保护的动态模块，由于其完整性有充分保证，可以例外。

规则6.3：禁止使用外部数据拼接SQL命令

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 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（对于每个引入的不可信数据推荐“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 SQL注入相关的特殊字符"）。 下列代码拼接用户输入，没有进行输入检查，存在SQL注入风险：

```
1.  char name[NAME_MAX];
2.  char sqlStatements[SQL_CMD_MAX];
3.  int ret = GetUserInput(name, NAME_MAX);
4.  ...
5.  ret = sprintf_s(sqlStatements, SQL_CMD_MAX, "SELECT childinfo FROM children WHERE
name= '%s'", name );
6.  ...
7.  ret = mysql_query(&myConnection, sqlStatements);
8.  ...
```

使用预处理语句进行参数化查询可以防御SQL注入攻击：

```
1.  char name[NAME_MAX];
2.  ...
3.  MYSQL_STMT *stmt = mysql_stmt_init(myConnection);
4.  char *query = "SELECT childinfo FROM children WHERE name= ?";
5.  if (mysql_stmt_prepare(stmt, query, strlen(query))) {
6.      ...
7.  }
8.  ret = GetUserInput(name, NAME_MAX);
9.  ...
10. MYSQL_BIND params[1];
11. res = memset_s(params, sizeof(params), 0, sizeof(params));
12. ...
13. params[0].bufferType = MYSQL_TYPE_STRING;
14. params[0].buffer = (char *)name;
15. params[0].bufferLength = strlen(name);
```

```
params[0].isNull= 0;

ret = mysql_stmt_bind_param(stmt, params);
...
ret = mysql_stmt_execute(stmt);
...
```

规则6.4：禁止在信号处理例程中调用非异步安全函数

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);
}
```

更多关于的异步安全函数，请参考"附录D 异步安全的函数列表"。

规则6.5：禁用setjmp/longjmp

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。 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

相关指南:

CERT.ERR52-CPP. Do not use setjmp() or longjmp()

规则6.6：禁止使用内存操作类危险函数

C标准的许多函数，没有将目标缓冲区的大小作为参数，并且未考虑到内存重叠、非法指针的情况，在使用中很容易引入缓冲区溢出等安全漏洞。 基于历史缓冲区溢出漏洞触发的情况统计，有很大一部分是因为调用了这些内存操作类函数但未考虑目标缓冲区大小而导致。 以下列出了内存操作类危险函数： □ 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() 请使用对应的安全函数(请参考"附录C 危险函数及替换的安全函数列表")。

例外： 在下列情况下，由于未涉及到外部数据处理，不存在被攻击的场景，内存操作完全在本函数内完成，不存在失败的可能性。如果使用安全函数反而造成代码的冗余，可以留用危险函数： （1）对固定长度的数组进行初始化：

```
1.  BYTE g_array[ARRAY_SIZE];
2.
3.  void Foo()
4.  {
5.      char destBuff[BUFF_SIZE];
6.      ...
7.      memset(g_array, c1, sizeof(g_array));       //对全局固定长度的数据赋值
8.      ...
9.      memset(destBuff, c2, sizeof(destBuff));     //对局部固定长度的数据赋值
10.     ...
11. }
```

（2）函数参数中有表示内存的参数，对该内存进行初始化：

```
1.  void Foo(BYTE *buff1, size_t len1, BYTE *buff2, size_t len2)
2.  {
3.      ...
4.      memset(buff1, 0, len1);     //对buff1清0
5.      memset(buff2, 0, len2);     //对buff2清0
6.      ...
7.  }
```

（3）从堆中分配内存后，赋予初值：

```
1.  size_t len = ...
2.  char *str = (char *)malloc(len);
3.  if (str != NULL) {
4.      memset(str, 0, len);
5.      ...
6.  }
```

（4）根据源内存的大小进行同等大小的内存复制： 以下代码基于srcSize分配了一块相同大小的内存，并复制过去：

```
BYTE *src = ...
size_t srcSize = ...
BYTE *destBuff = new BYTE[srcSize];
memcpy(destBuff, src, srcSize);
```

以下代码根据源字符串的大小分配一块相同的内存，并复制过去：

```
char *src = ...
size_t len = strlen(src);
if (len > BUFF_SIZE) {
    ...
}
char *destBuff = new char[len + 1];
strcpy(destBuff, src);
```

（5）源内存全部是静态字符串常量（编码时需要检查目标内存是否足够的存储空间）：以下代码直接将字符串常量“hello”复制到数组中：

```
char destBuff[BUFF_SIZE];
strcpy(destBuff, "hello");
```

以下代码对静态字符串常量进行拼接：

```
const char *g_list[] = {"red","green","blue"};
char destBuff[BUFF_SIZE];
sprintf(destBuff, "hello %s", g_list[i]);
```

（6）使用Visual Studio2005(及更高版本)开发windows程序时，使用了微软提供的安全函数，那么不必单独开发并使用memset_s安全函数。

7 文件输入/输出

规则7.1：创建文件时必须显式指定合适的文件访问权限

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。 下列代码没有显式配置文件的访问权限。

```
    int fd = open(fileName, O_CREAT | O_WRONLY); //【错误】缺少访问权限设置
```

推荐做法：

```
    int fd = open(fileName, O_CREAT | O_WRONLY, S_IRUSR|S_IWUSR);
```

规则7.2：必须对文件路径进行规范化后再使用

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。 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：

```
1.  char *fileName = GetMsgFromRemote();
2.  ...
3.  sprintf_s(untrustPath, sizeof(untrustPath), "/tmp/%s", fileName);
4.  char *text = ReadFileContent(untrustPath);
```

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：

```
1.  char *fileName = GetMsgFromRemote();
2.  ...
3.  sprintf_s(untrustPath, sizeof(untrustPath), "/tmp/%s", fileName);
4.  char path[PATH_MAX] = {0};
5.  if (realpath(untrustPath, path) == NULL) {
6.      //error
7.      ...
8.  }
9.  if (!IsValidPath(path)) {    //检查文件是否的位置是否正确
10.     //error
11.     ...
12. }
13. 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。 例如：

```
1.  ...
2.  int main(int argc, char **argv)
3.  {
4.      int fd = -1;
5.
6.      if (argc == 2) {
7.          fd = open(argv[1], O_RDONLY);
8.          ...
9.      }
10.     ...
11. }
```

规则7.3：不要在共享目录中创建临时文件

建议7.1：在进行文件操作时避免引起竞争条件

在对文件操作时，在判断文件属性和进行文件的实际操作的时间差内，会受到攻击者的攻击，导致在文件的操作过程不符合预期。 例如，下面的代码通过调用access函数先判断文件是否可写，如果可写，就创建文件。但是在调用access和fopen的微小的时间差内，攻击者可能改变文件的属性，导致实际执行fopen的时候，文件属性变为不可写。

```
if (access(fileName, W_OK) == 0) {
    fp = fopen(fileName, "w+");
    //operate fp;
}
```

相关指南： CERT. FIO45-C. Avoid TOCTOU race conditions while accessing files

8 敏感信息处理

规则8.1：禁用rand函数产生用于安全用途的伪随机数

C标准库rand()函数生成的是伪随机数，请使用/dev/random生成随机数。

相关指南:

CERT.MSC30-C. Do not use the rand() function for generating pseudorandom numbers

CERT.MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers

MITRE.CWE-327, Use of a Broken or Risky Cryptographic Algorithm

MITRE.CWE-330, Use of Insufficiently Random Values

规则8.2：内存中的敏感信息使用完毕后立即清0

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。 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_s对password清0。

```
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_s(password, sizeof(password), 0, sizeof(password));
    ...
}
```

对敏感信息清理的时候要同时防止因编译器优化而使清理代码无效。 例如，下列代码使用了可能被编译器优化掉的语句。

```
1.  void  SecureLogin()
2.  {
3.      char pwd[PWD_SIZE] = {0x00};
4.      if (retrievePassword(pwd, sizeof(pwd))) {
5.          // 口令检查及其他处理
6.      }
7.      memset(pwd, 0, sizeof(pwd));     // 编译器优化有可能会使该语句失效
8.      ...
9.  }
```

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。 可以使用华为安全函数库中的memset_s()函数来避免因编译器优化导致敏感信息清理失败。

```
1.  void  SecureLogin()
2.  {
3.      char pwd[PWD_SIZE] = {0x00};
4.      if (RetrievePassword(pwd, sizeof(pwd))) {
5.          // 口令检查，其他安全处理
6.      }
7.      (void)memset_s(pwd, sizeof(pwd), 0, sizeof(pwd));
8.      ...
9.  }
```

在windows系统可以使用SecureZeroMemory()函数。

```
1.  void  SecureLogin()
2.  {
3.      char pwd[PWD_SIZE] = {0x00};
4.      if (RetrievePassword(pwd, sizeof(pwd))) {
5.         // 口令检查，其他安全处理
6.      }
7.      SecureZeroMemory(pwd, sizeof(pwd));
8.      ...
9.  }
```

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。

```
1.  void  SecureLogin()
2.  {
3.      char pwd[PWD_SIZE] = {0x00};
4.      if (RetrievePassword(pwd, sizeof(pwd))) {
5.           // checking of password, secure operations, etc
6.      }
7.      #pragma optimize("", off)
8.      // 清除内存
9.      ...
10.     #pragma optimize("", on)
11.     ...
12. }
```

规则8.3：严禁使用string类存储敏感信息

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。

```
1.  int VerifyPassword(string password)
2.  {
3.      //...
4.  }
5.  int Foo()
6.  {
7.      string password = GetPassword();
8.      VerifyPassword(password);
9.      ...
10. }
```

应该使用char或unsigned char保存敏感信息，如下代码：

```
1.  int VerifyPassword(const char *password)
2.  {
3.      //...
4.  }
5.  int Foo()
6.  {
7.      char password[MAX_PASSWORD] = {0};
8.      GetPassword(password, sizeof(password));
9.      VerifyPassword(password);
10.     ...
11. }
```

附录A SQL注入相关的特殊字符

表1 常用数据库中与SQL注入攻击相关的特殊字符

数据库	特殊字符	描述	转义序列
Oracle	'	单引号	''(两个单引号)
MySql	'	单引号	\'
MySql	"	双引号	\"
DB2	'	单引号	''(两个单引号)
DB2	;	分号	.
SQL Server	'	单引号	''(两个单引号)

表2 like条件中的通配符及转义方式

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

附录B 命令注入相关的特殊字符

表3 shell脚本中常用的与命令注入相关的特殊字符

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。
内联命令	&	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。
表达式	$	可用在${}中作为变量的正规表达式。
重定向操作	>	将命令输出写入到目标文件中。
重定向操作	<	将目标文件的内容发送到命令当中。
反引号	`	可在``之间构造命令内容并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。
感叹号	!	事件提示符(Event Designators),可以引用历史命令。

换行符	\n	可以用在一行命令的结束，用于分隔不同的命令行。

上述字符也可能以组合方式影响命令拼接，如管道符“||”，“>>”，“<<”，逻辑操作符“&&”等，由于基于单个危险字符的检测可以识别这部分组合字符，因此不再列出。另外可以表示账户的home目录“~”，可以表示上层目录的符号“..”，以及文件名通配符“?”(匹配文件名中除null外的单个字元)，“*”(匹配文件名的任意字元)由于只影响命令本身的语义，不会引入额外的命令，因此未列入命令注入涉及的特殊字符，需根据业务本身的逻辑进行处理。

附录C 危险函数及替换的安全函数列表

不同平台下危险函数及替代函数请参考配套文档《不同平台危险函数以及对应的安全函数.xlsx》。

附录D 异步安全的函数列表

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）。

参考资料

1.CERT C安全编码规范: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

2.CERT C++安全编码规范：https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

3.CWE软件开发中的常见弱点：http://cwe.mitre.org/data/definitions/699.html

4.MISRA-C:2004 Guidelines for the use of the C language in critical systems

贡献者

感谢所有参与规则制订、检视、评审的专家、同事！

感谢所有提 issue / MR 参与贡献的同事！

规范号	主要起草部门专家	主要评审部门专家	版本
DKBA6914-2018.12	研发能力中心 俞科技 00258636	2012 中软 杨胤博444939 中硬 施文超303178 研发能力中心 吴海翔286643,刘进283514 海思 刘炜刚358808、孙习波246516 CBG 终端 黄雷172019 终端云 舒超302625 终端芯片 彭代兵163789 CloudBU CloudBU 傅宇宏178233 P&S 无线 周文水274925 鼎桥 张旭WX10698 公共开发部 樊思龙385867 IT 李蔚189256 网络 杨爱晶184831 云核心网 刘巍290580 网络能源 陈小霞241217 GTS 软件 桂莉莉 00266347	V3.1 规则1.1.1：增加关于全局变量和静态变量的初始化例外 规则1.3.1： 1、增加对const char *的说明 2、增加对字符替换、字符裁剪的说明 3、增加结构体数组的特定例外场景 4、增加固定长度数组的特定例外场景 规则1.4.1： 1、增加对操作系统软件idle线程的例外 2、增加操作系统本身出错处理的例外 3、增加嵌入式系统主任务处理流程的例外 规则1.7.1：增加维测功能的例外。 规则1.7.2：增加对kill -9的说明，明确其他信号机制仍然是可以使用的 规则3.5：抛出异常不是推荐的错误处理方式 规则4.5：禁止对指针进行逻辑或位运算（&&、||、!、 ~、>>、<<、&、^、|） 1、补充描述，使语义更完整 2、增加地址对齐操作的例外 规则6.1：规则6.1：禁止外部可控数据作为system、execl, execlp, execle, execv, execvp、WinExec、CreateProcess、ShellExecute等进程启动函数的参数 1、修改规则表达方式，提升可读性 2、增加了部分示例代码，提升可读性 规则6.2：增加了对模块有完整性保护机制时的例外 规则6.6：禁止使用内存操作类危险函数
DKBA6914-2018.09	研发能力中心 俞科技 00258636	P&S 无线 周文水00274925、孙钢00449514、舒芳 00333269、李文 00197797、何涛00468248、林戎 00302555、谯洋 00468139	V3.0 1、 概述增加了攻击者思维、安全编码基本思想内容 2、 增加了断言、安全退出的用法要求 3、 增加了C++异常、类的用法要

		鼎桥 张旭WX10698 公共开发部 樊思龙00385867 IT 李蔚00189256、黄全伟00442111 网络 杨爱晶00184831、洪冬梅 00453918、张益波00386600、肖华山00317303、黄维东00265762、申力华00247159、阮强胜00342606、赵广 00176778 云核心网 刘巍00290580、王宏磊 00435402 网络能源 陈小霞00241217 2012 中软 朱其刚405975、倪小明202754、王磊 00176598 中硬 施文超303178 研发能力中心 吴海翔00286643、陆林 00179153、宋鑫 00473779、赵志健00297261、刘进 00283514、张杰 00316590 海思 刘炜刚00358808、孙习波00246516 CBG 终端 黄雷00474221 终端云 舒超00302625 终端芯片 彭代兵00163789 CloudBU CloudBU 傅宇宏00178233 GTS 软件 桂莉莉 00266347、王会灵 00286595 特战队 范一鸣 00342532、方超00189446 美研所 Bin Liang 00452313 、Xi Kang 00385799、Yan Chen 00754567、Wenzhe Zhou 00725915、Chun Liu 00415003、Wei Wei 00904415、Xuejun Yang 00759278、Haichuan Wang 00759152	求 4、 增加了安全函数使用要求 5、 敏感信息规则集中单列一章 6、 优化了原规范中安全原则有关的规则条目，在基本思想及各章具体条款中体现 7、 优化了原C++规则与安全关联较弱的部分 8、 优化了部分应用范围非常狭窄的规范条目，提升阅读效率

DKBA6914-2017.12	CSPL：于鹏 90006799、刘进 00283514、赵志健 00297261	安全办公室：李怀林 00255143、李杰 00340536 CSPL：宋鑫 00168120、陈宇 00291422、吴敏 00326311、章可镌 00326344 无线：周文水 00274925 鼎桥：张旭 WX10698 软件：古晶 00180691、瞿孝干 00341965、郭亚奇 00291956 IT：傅宇宏 00178233、李蔚 00189256 网络：杨爱晶 00184831、郑飞 00399154 云核心网：刘巍 00290580、蔚鸽 00345693 网络能源：詹松烈 00173034、陈小霞 00241217 中软：朱其刚 00405975 中硬：施文超 00303178 研发能力中心：吴海翔 00286643、周志东00290381 终端：黄雷 00172019 终端云：舒超 00302625 终端芯片：彭代兵 00163789 P&S研发流程质量部：卫涛 00329558	V2.3 修改规则1，规则C1.3，规则C4.5，规则C4.6，规则C4.7，规则C6.3，规则C8.6，规则C8.7，规则C9.5，附录A，附录B。 V2.31 附录3危险函数及其替换函数不再于此规范中重复展示，以独立文档《不同平台危险函数以及对应的安全函数》为唯一数据来源。
DKBA6914-2016.06	网络安全能力中心：于鹏 90006799	研发能力中心：郭曙光 00121837、施勇 00134637 安全能力中心：朱喜红 00210657 固定网络研发管理部：张伟 00118807 无线网络研发管理部：陶永祥 00120482 IP开发部：江巍 00223927、杨磊 00234388 电信软件与核心网：赵玉锡 00232229、陈辉军 00123456 IT开发管理部：林国仁 00145836 高斯3部：侯朋飞 00173757	V2.2 1. 新增“内核操作安全”章节，该章节中包含8条规则（2条来自原“其它”章节，6条为新增）； 2. 修改规则1、C3.1、C3.2，建议CPP2.1； 3. 删除原规则C3.4，并新增规则C3.4。 V2.21 新增规则C9.5，增加SQL注入相关内容。

DKBA6914-2015.10	网络安全能力中心：于鹏 90006799、臧志远 00294031、程鹤 00312162	研发能力中心：郭曙光 00121837、施勇 00134637 安全能力中心：朱喜红 00210657 固定网络研发管理部：张伟 00118807 无线网络研发管理部：陶永祥 00120482 IP开发部：江巍 00223927、杨磊 00234388 电信软件与核心网：赵玉锡 00232229、陈辉军 00123456 IT开发管理部：林国仁 00145836 内部网络安全实验室：方亮 00211245	V2.1 1. 新增C8.5、C8.6两条内核规则； 2. 重点修改了规则C2.1、C5.1、C5.2、C6.2、C8.1、C8.2和建议C8.2； 3. 进一步完善附录B和附录C； 4. 修改了部分描述和代码笔误。
DKBA6914-2014.07		研发能力中心：郭曙光 00121837、施勇 00134637 安全能力中心：朱喜红 00210657 固定网络研发管理部：张伟 00118807 无线网络研发管理部：陶永祥 00120482 IP开发部：江巍 00223927、杨磊 00234388 电信软件与核心网：赵玉锡 00232229、陈辉军 00123456 IT开发管理部：林国仁 00145836	V2.0 1. 修改了文章结构。分为C安全编程部分和C++编程部分； 2. 刷新了示例； 3. 删除了6条规则； 4. 增加了10条规则； 5. 重点修改了6条规则。

DKBA6914-2013.05	网络安全能力中心： 罗东 67107、 于鹏 90006799、苗宏 90006736、朱喜红 00210657 电信软件与核心网：陈辉军 00190784 无线产品线：肖飞龙 00051938 网络产品线：魏建雄 00222905 IT产品线：熊华梁 00106214 中央软件院：朱楚毅00217543、林水平 00109837、周强 00048368、辛威 00176185、鞠章蕾 00040951、谢青 00101378 中央硬件院：刘永合 00222758 终端公司：杨棋斌 00060469 企业网络：黄凯进 00040281、 企业SecoSpace：王瑾 90003828	中央软件院：黄茂青 00057072、卢峰 00210300 网络产品线：李强 00203020、罗天 00062283、廖永强 00111217、任志清 00048956、李海蛟 00040826、陈璟 00222879、勾国凯 00048893、范佳甲 00109753 中央硬件院：刘崇山 00159994、施文超 00109740 企业网络：李有永 90002701 IT产品线：李显才 00044635、何昌军 00061280 能力中心：郭曙光 00121837 网络安全实验室：林结斌 00206214 电信软件与核心网：朱刚 00192988 无线产品线：李瀛 00130531、王爱成 00223009、杨彬 00065941、于继万 00052142、解然 00234688 IT B部：罗晓星00102517	V1.0