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 和 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 或 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。

在 C++代码中，除了要满足 C 安全编码要求外，还需要满足如下 C++安全编码要求。

2 禁用不安全函数和对象

规则 2.1:禁止使用 std::ostrstream,推荐使用 std::ostringstream

说明： std::ostrstream 的使用上需要特别注意几点：

（1）str() 会调用成员函数 freeze(),它会冻结字符序列，当缓冲区不够大以至于需要分配新缓冲区时，这么做可以避免事情变得复杂。

（2）str()不会附加字符串终止符号（‘\0’）。

（3）data()返回所有字符串，没有附带‘\0’结尾字符（目前有些编译器自动调用 c_str 方法了）。

上面如果不注意，就可能会导致内存访问越界、缓冲区溢出等问题，所以建议不要使用 ostrstream。[C++03]标准将 std::strstream 标明为 deprecated，替代方案是 std::stringstream。ostringstream 没有上述问题。

错误示例：下列代码使用了 std::ostrstream，可能会导致内存访问越界等问题。

```
...
ostrstream mystr;  /* 【错误】不要使用 std::ostrstream */
mystr << "Information is here.";
cout << mystr.str() << std::endl;
...
```

推荐做法：

```
...
ostringstream mystr;  /* 【修改】使用 ostringstream 来代替
ostrstream */
mystr << "hello world";
cout << mystr.str() << std::endl;
...
```

建议 2.2：在 C++代码中优先使用 C++类库函数

说明：C++是面向对象语言而 C 是面向过程的编程语言，正是由于这种特性，C++语言相比于 C 语言具有更多的面向对象特性（封装，继承，多态）和可维护性。总之 C++类库函数比 C 函数更加安全。举例如下：

C 标准的系列字符串处理函数 strcpy/strcat/sprintf/scanf/gets，不检查目标缓冲区的大小，容易引入缓冲区溢出的安全漏洞。C++标准库提供了字符串类抽象的一个公共实现 std::string，支持字符串的常规操作：

- 字符串拷贝

- 读写访问单个字符
- 字符串比较
- 字符串连接
- 字符串长度查询
- 字符串是否为空的判断。

因此，在C++程序中，尽可能使用std::string、std::ostringstream等替代不安全的C字符串操作函数。

错误示例：使用了C风格的字符串操作函数。

```
...
const char filename[]="some_file.txt";
char cmd[BUFSIZE]="Usage:file=";
strcat(cmd,filename);  /* 【不推荐】在C++中不要使用C风格字符粗操作函数 */
...
```

推荐做法：

```
...
std::string cmd = "Usage:file=";
cmd += "some_file.txt";  /* 【修改】使用C++标准库来代替C字符粗函数 */
...
```

对于输入/输出函数，C++类中的析构函数可以确保资源不会泄露。但是在C中需要手工进行资源释放，很容易造成资源泄露危险。

此外，混用C和C++函数是很危险的容易造成异步问题（例如规则C8.3：多线程环境下只使用可重入函数所展示的printf()和std::out的混用）。因此在C++代码中应该尽可能的使用C++类库函数。

3 C++类和对象安全

规则 3.1：禁止切分多态的类对象

说明：当一个基类有继承类时，禁止从继承类对象到基类对象实例的拷贝，也不能在多个继承类的对象之间相互拷贝，这样会导致信息的丢失，程序运行异常，从而引发 DOS（denial-of-service）。

错误示例：下列代码中切分了类对象，会导致数据丢失。

```
class Employee {
public:
   Employee(string theName): name(theName){};
   string getName() const {return name;}
   virtual void print() const
   {
      cout << "Employee: " << getName() << endl;
   }
virtual ~Employee()
private:
   string name;
};
class Manager: public Employee {
public:
   Manager(string theName, Employee
theEmployee):Employee(theName), assistant(theEmployee) {};
   Employee getAssistant() const {return assistant;}
   virtual void print() const
   {
      cout << "Manager: " << getName() << endl;
      cout << "Assistant: " << assistant.getName() << endl;
   }
private:
   Employee assistant;
};
int main ()
{
   Employee coder("Joe Smith");
   Employee typist("Bill Jones");
   Manager designer("Jane Doe", typist);

   coder = designer; /* 【错误】切分了对象 designer:Jane Doe */

   coder.print();
}
```

运行结果：Employee: Jane Doe

示例代码中基类 Employee，继承类 Manager（增加了属性 assistant），如果将 Manager 类的对象数据拷贝给 Employee 类的对象，则将发生对象切分，Manager 类的 assistant 属性数据将丢失。

推荐做法 1（引用）：

```
int main ()
{
   Employee coder("Joe Smith");
   Employee typist("Bill Jones");
   Manager designer("Jane Doe", typist);
   Employee &toPrint = designer; /* Jane remains entire */
   toPrint.print();
}
```

推荐做法 2（使用指针）：

```
int main ()
{
   Employee *coder = new Employee("Joe Smith");
   Employee *typist = new Employee("Bill Jones");
   Manager *designer = new Manager("Jane Doe", *typist);
   coder = designer;
   coder->print();
}
```

推荐做法 3（使用智能指针）：

```
int main ()
{
   auto_ptr<Employee> coder(new Employee("Joe Smith"));
   auto_ptr<Employee> typist(new Employee("Bill Jones"));
   auto_ptr<Manager> designer(new Manager("Jane Doe",
*typist));
   coder = designer; /* Smith deleted, Doe xferred */
   coder->print();
   /* everyone deleted */
}
```

运行结果：

```
Manager: Jane Doe
Assistant: Bill Jones
```

规则 3.2：禁止定义基类析构函数为非虚函数，所有可能被继承类的析构函数都必须定义为 virtual

说明：基类的析构函数如果不是 virtual 的，那么在对一个 Base 类型的指针进行 delete 时，就不会调用到派生类 Derived 的析构函数。而派生类里的析构函数一般会用于析构其内部的子对象，这样就可能会造成内存泄漏。

错误示例：代码中的析构函数没有被定义成虚函数。

```
class Base
{
public:
   ~Base(){}; /*【错误】禁止定义基类析构函数为非虚函数 */
};
class Derived : public Base
{
private:
   char *pc;
public:
   Derived()
   {
      pc=new char[100];
   };
   ~ Derived()
   {
      delete [] pc;
   };
};
void main()
{
   Base *obj = new Derived();
   delete obj;
}
```

以上示例代码基类 Base 的析构函数不是 virtual 的。因为不是 virtual，所以在对 Base 类型的指针 obj 进行 delete 时，不会调用到派生类 Derived 的析构函数，这样就造成内存泄漏。

推荐做法：基类 Base 的析构函数定义为 virtual，这样确保在对 Base 类型的指针 obj 进行 delete 时调用派生类 Derived 的析构函数。

```
class Base {
public:
   virtual ~Base(){};/*【修改】定义基类析构函数为虚函数 */
```

```
};
class Derived : public Base {
private:
   char *pc;
public:
   Derived()
   {
      pc=new char[100];
   };
   ~ Derived()
   {
      delete [] pc;
   };
};
void main()
{
   Base *obj = new Derived();
   delete obj;
}
```

规则 3.3：避免出现 delete this 操作

说明：对象指针应避免使用 delete this 语句硬删除，除非能保证 this 指针删除后不再被引用，并且保证对象是通过 new 操作符在堆上创建的。

原因有两个：

（1）类的对象既可能是栈对象，也可能是堆对象。如果对栈对象的指针进行 delete，即删除非动态分配的内存，会导致未定义行为；

（2）二是 delete this 容易产生悬挂指针（dangling pointer），悬挂指针是个严重的安全漏洞，可以被攻击者利用执行任意代码。

错误示例：错误的删除 this 指针

```
class SomeClass {
public:
   SomeClass();
   ~SomeClass();
   void doSomething();
   void destroy();
   /* ... */
};
void SomeClass::destroy()
{
```

```
    /* ...do something... */
    delete this; /*【错误】删除 this 指针会导致出现悬挂指针 */
}

void main()
{
    /* ...do something... */
    SomeClass sc; // 声明栈对象
    /* ...do something... */
    sc.destroy(); /* 释放非动态分配的内存*/
}
```

推荐做法：不 delete this，让栈对象离开作用域后自动析构。

```
class SomeClass{
public:
    SomeClass();
    ~SomeClass();
    void doSomething();
    void destroy();
    /* ... */
};
void SomeClass::destroy()
{
    /* ...do something... */
}
/* ... */

void main()
{
    SomeClass sc; // 声明栈对象
    /* ...do something... */
} /* 离开作用域，自动调用 sc.~SomeClass() */
```

如果不得不使用 delete this，保证类对象是堆对象，且 this 指针 delete 后置 NULL，可参考如下示例代码：

```
class SomeClass {
public:
    SomeClass();
    void doSomething();
    void destroy();
    /* ... */
```

```
protected:
   ~SomeClass();
};
void SomeClass::destroy()
{
   /* ...do something... */
   delete this;
}
/* ... */
{
   SomeClass* sc = new SomeClass();
   /* ...do something... */
   sc->destroy();
   sc = NULL;
}
```

这个示例代码中，将析构函数声明为 protected，可以保证类 SomeClass 的对象不会在栈上生成。同时，在显示调用 destory()来 delete this 指针后，再将指针置 NULL，防止指针解引用。

规则 3.4：禁止在类的公共接口中返回类的私有数据地址

说明：如果一个类私有成员数据的引用或者其指针，被类的公有函数作为返回值 return，则此私有数据可能会遭受到非可信代码的修改，导致引入不安全因素。

错误示例：下列代码类中的私有成员变量被公共成员函数所引用。

```
class Widget {
public:
   Widget (): total(0) {}
   /* ... */
   void add (someType someParameters)
   {
      /* ... */
      total ++;
      /* ... */
   }
   void remove (someType someParameters)
   {
      /* ... */
      total --;
      /* ... */
   }
   /* ... */
```

```
    int& getTotal() {return total;} /*【错误】禁止返回类的私有
```

数据成员地址 */

```
    /* ... */
private:
    int total;
    /* ... */
};
```

示例代码中，total 作为类的私有成员，维护着对类方法 add 与 remove 的调用计数，但是其实际值却被类的公共成员函数 getTotal 对外提供了可引用的接口。

推荐做法：

```
class Widget {
public:
    Widget(): total(0) {}
    /* ... */
    void add(someType someParameters)
    {
        /* ... */
        total ++;
        /* ... */
    }
    void remove(someType someParameters)
    {
        /* ... */
        total --;
        /* ... */
    }
    /* ... */
    int getTotal() const {return total;}
    /* ... */
private:
    int total;
    /* ... */
};
```

建议 3.1：重载后缀操作符应返回 const 类型

说明：C++标准中，列出了自增和自减操作符的特点：

```
class X
{
public:
  X& operator++();    // prefix ++a
```

```
  X operator++(int); // postfix a++
};
class Y { };
Y& operator++(Y&);      // prefix ++b
Y operator++(Y&, int); // postfix b++
```

需要注意的是，前缀操作符返回的结果是 non-const 引用，而后缀操作符返回的可能是临时变量或者一个地址。传统重载自增和自减操作符的实现如下：

```
class C
{
  void Increment();
public:
  C(const C&);
  C& operator=(const C&);

  C operator++(int) {
    C R(*this);
    Increment();
    return R;
  }
};
```

代码中返回的对象只是调用 Increment()函数之前该对象的一个快照，因此，所有调用 operator++(int)所操作的仅仅是一个临时变量，并不会对原始对象产生任何影响。这就要求当重载后缀操作符时，建议重载函数返回值类型为 const。

错误示例：下列代码的目的是希望 c 自增两次得到结果 2，但是由于 c 只是返回了一个预先值，所以 c.I 只自增了一次。c++返回的对象又自增了一次，但是却没有影响到原始的 c 对象。

```
#include <iostream>

class C
{
  int I;
public:
  C() : I(0) {}
  C(const C &RHS) : I(RHS.I) {}

  C& operator=(const C &RHS) {
    I = RHS.I;
    return *this;
  }

  C operator++(int)
  {
    C R(*this);
    I++;
```

```
    return R;
  }

  int getI() const { return I; }
};

int main()
{
  C c;
  c++++;
  std::cout << c.getI();
}
```

代码的执行结果是 1 而不是 2。

推荐做法：重载后缀操作符为 const 类型。

```
#include <iostream>

class  C
{
    int  I;

public:
    C() : I(0) {}
    C(const  C &RHS) : I(RHS.I) {}

    C& operator=(const  C &RHS)
    {
        I = RHS.I;
        return  *this;
    }

    const  C operator++(int)  /* 【修改】重载++a 时返回 const 类型 */
    {
        C R(*this);
        I++;
        return  R;
    }

    int  getI()  const  {  return  I; }
};
int  main()
{
    C c;
    c++;
    c++; /* 【修改】换一种自增方式 */
```

```
    std::cout << c.getI();
}
```

将自加和自减后缀操作符的返回值定义为 const 型后，若执行 C++++这样的代码，在编译时编译器就会告警，从而避免之前错误的发生。

建议 3.2：显式声明的模板类应进行类型特化

说明：编译器不会严格地验证模板的参数，容易被破解者利用，并造成攻击。

错误示例：模板类使用错误。

```
template <typename T>
class Sample
{
public:
    void funcA() {/* ... */}
    void funcB()
    {
        T t;
        t.x = 50;
    }
};
int main()
{
    Sample<int> a;    /* 【错误】Sample<int>::funcB 有问题，int
并不是 class 且没有成员变量 x */
    a. funcA();
}
```

示例代码 Sample <int>:: funcB 明显是有问题的，因为类型 int 并不是 class，并且也没有成员变量 x。很明显，模板 A 的设计者并不是将该模板应用于类型 int。然而编译器并不会捕捉到这个错误，因此代码会被成功编译，却引入了缺陷。

推荐做法：

```
template <typename T>
class Sample
{
public:
    void funcA() {/* ... */}
    void funcB()
    {
        T t;
        t.x = 50;
    }
```

```
};
template class Sample<int>;  /*【修改】显示声明模板类特化*/
int main()
{
   Sample<int> a;
   a.funcA();
}
```

添加如上代码后，编译器会捕获到样例代码中的错误，因为模板的声明会强制编译器初始化类的所有成员，包括 Sample<int>:: funcB()，此时就会捕获到编译错误。

4 STL 库安全

规则 4.1：引用容器前后元素时要确保容器元素存在

说明：没有判断是否为空就直接通过引用 STL 容器首尾元素，这在容器为空时会导致程序异常。

错误示例：

```
bool NoCompliant(const NodeKeyList &srcList, const
NodeKeyList &snkList)
{
   NodeKey srcNode = srcList.front();
   NodeKey snkNode = snkList.back();
   /* ...do something... */
}
```

示例代码对函数的入参 srcList 没有判断长度直接通过 front()和 back()方法取了第一个和最后一个元素，在容器列表为空的情况下，会导致程序异常，与 front()类似的还有通过 begin()数据下标获取对应元素，比如 *srcList.begin()，或者 srcList.begin()->GetID()，或者是 srcList[0](在为 vector 时)。

推荐做法：

```
bool CombineList(const NodeKeyList &srcList, const
NodeKeyList &snkList)
{
   if (srcList.empty() || snkList.empty()) /* 【修改】确保 STL
容器内有元素存在 */
   {
      return false;
   }
   NodeKey srcNode = srcList.front();
   NodeKey snkNode = snkList.back();
   /* ...do something... */
}
```

规则 4.2：迭代子使用前必须保证迭代子有效

说明：STL 算法 std::find()、std::find_if()和 std::set::find()等有可能返回容器的 end()位置，迭代子定义时可以不初始化，或者初始化指向容 find()等方法返回的位置，与指针类似地，若未判断迭代子有效性，直接引用迭代子有可能导致程序崩溃。

错误示例：

```
void STLIterTest::IterReference_NoCompliant(int CmdCode,
MAP_GENKEY_VALUE& allResult)
{
   TEGenKey tmpKey;
   tmpKey.attrID = DWDMTL1_ATTRPORT_CLIENTPRO;
   tmpKey.objectID = it->first.objectID;
   MAP_GENKEY_VALUE::iterator iter =
allResult.find( tmpKey );

   if ("FC100" != iter->second.sValue) /* 非 FC100 设置为无效
*/
   {
      it->second.access = TEGenVar::ACCESS_INVALID;
   }
}
```

示例代码通过 map 的 find 函数返回的迭代子 iter，if 语句直接通过 iter->second 来引用变量，如果迭代子 iter 指向为 allResult 的 end()位置，则程序会崩溃。

推荐做法：

```
void STLIterTest::IterReference_Compliant(int CmdCode,
MAP_GENKEY_VALUE& allResult)
{
   TEGenKey tmpKey;
   tmpKey.attrID = DWDMTL1_ATTRPORT_CLIENTPRO;
   tmpKey.objectID = it->first.objectID;
   MAP_GENKEY_VALUE::iterator iter =
allResult.find(tmpKey);

   if (iter != allResult.end()) /*【修改】确保迭代子有效后再进
行操作 */
   {
      if ( "FC100" != iter->second.sValue)
      {
         it->second.access = TEGenVar::ACCESS_INVALID;
      }
   }
}
```

规则 4.3：必须确保迭代子指向的内容有效

说明：在理解上迭代子可以视为 C 指针，迭代子只有在指向了容器中一个存在的对象时，访问才是安全有效的，其他情况的访问都可能存在风险。典型问题：

对连续内存容器来说(如 std::vector)会分配一块固定内存来保存连续对象，在插入新元素后（成员函数包括：reserve()，resize()， push_back()，insert()等)，可能会引起容器重新分配内存和数据迁移，如果在插入元素之前使用迭代子保存了迭代子位置，那么插入新元素之后，前面保存的迭代子就可能是无效的。

错误示例 1：下列代码的迭代子在操作过程中失效。

```
void ProcessMessageQueue()
{
   deque<double> d;
   double data[5] = { 2.3, 3.7, 1.4, 0.8, 9.6 };
   deque<double>::iterator pos = d.begin();
   for (size_t i = 0; i < 5; ++i)
   {
      d.insert(pos++, data[i] + 41); /*【错误】insert 操作后，pos 已失效 */
   }
}
```

Insert 操作后，迭代子 pos 已经失效，执行自增操作导致异常。

推荐做法：

```
void ProcessMessageQueue()
{
   double data[5] = { 2.3, 3.7, 1.4, 0.8, 9.6 };
   deque<double> d;
   deque<double>::iterator pos = d.begin();
   for (size_t i = 0; i < 5; ++i)
   {
      pos = d.insert(pos, data[i] + 41); /*【修改】通过返回值获得新的有效的迭代子 */    ++pos;
   }
}
```

std::remove 和 std::remove_if 仅会将被删除元素后移并返回该被删除元素位置的迭代子，并没有真正从容器中删除对象，需要另配合 erase 函数才能删除，所以一般建议配合一起使用。

错误示例 2：下列代码中错误的仅使用 remove()函数来删除容器中元素。

```
void CleanContainer()
{
   vector<int> container;
   int value = 42;
  iterator end = remove( container.begin(), container.end(), value);
```

```
    for (iterator i = container.begin(); i != container.end();
++i)
    {
        cout << "Container element: " << *i << endl;
    }
}
```

remove() 删除任一个成员后返回值将指向任一个成员，值将不可预知。所以被删除后需要立即调用 erase()抹去，防止不可预知的数据访问。

推荐做法：

```
void CleanContainer()
{
    vector<int> container;
    int value = 42;
    container.erase(remove(container.begin(),container.end
(),value), container.end()); /*【修改】remove 删除成员后立即调
用 erase，确保迭代子指向的内容是有效的*/
    for (iterator i = container.begin(); i != container.end();
++i)
    {
        cout << "Container element: " << *i << endl;
    }
}
```

规则 4.4：正确处理容器 erase()方法与迭代子的关系

说明：调用容器的 erase(iter)方法后，迭代子指向的对象被析构，迭代子已经失效，如果再对迭代子执行递增递减或者引用操作会导致程序崩溃。

错误示例：下列代码中的迭代子在执行删除操作过程中已失效。

```
void STLIterTest::IterVisitContainer()
{
    std::map<oid,NE>::iterator it = m_mapID2NE.begin();
    for (; it != m_mapID2NE.end(); )
    {
        if (pNEInfo->GetNEState(ulNEID) == NESTATE_LOGIN)
        {
            m_mapID2NE.erase(iter);
            iter++;  /*【错误】erase()后，iter 指向的对象可能已失效
*/
        }
```

```
        else {++iter;}
    }
}
```

推荐做法：将迭代子后置递增作为 erase()的参数。

```
void STLIterTest::IterVisitContainer()
{
    std::map<oid,NE>::iterator it = m_mapID2NE.begin();
    for (; it != m_mapID2NE.end(); )
    {
        if ( pNEInfo->GetNEState(ulNEID) == NESTATE_LOGIN)
        {
            m_mapID2NE.erase(iter++); /*【修改】将迭代子后置递增作为 erase()参数 */
        }
        else {++iter;}
    }
}
```

也可以使用 earse 方法的返回值来保存迭代子，因为返回的是被删除元素迭代子指向的下一个元素位置：

```
iter = erase(iter)。
```

注意这种用法可以用于 list 和 vector 的 erase()，但不适用于 map。因为 std::map::erase()的返回值在不同 STL 实现版本是有差异的，有的有返回值，有的没有返回值，所以对 map 只能使用推荐做法。

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_ChkStrNCpy _VOS_StrNCpy	VOS_strncpy_s	strncpy, _tcsncpy, _mbsncpy, _mbsnbcpy, StrCpyN, StrCpyNA, StrCpyNW, StrNCpy, strcpynA, StrNCpyA, StrNCpyW, lstrcpyn, lstrcpynA, lstrcpynW, _fstrncpy	strncpy_s

	wcsncpy	wcsncpy_s			wcsncpy	wcsncpy_s
字符串连接	strcat	strcat_s	VOS_StrCat VOS_strcat VOS_ChkStrCat _VOS_StrCat	VOS_strcat_s	strcat, strcatA, strcatW, _tcscat, _mbscat, StrCat, StrCatA, StrCatW, lstrcat, lstrcatA, lstrcatW, StrCatBuff, StrCatBuffA, StrCatBuffW, StrCatChainW, _tccat, _mbccat, _ftcscat	strcat_s
	wcscat	wcscat_s			wcscat	wcscat_s
	strncat	strncat_s	VOS_StrNCat VOS_ChkStrNCat _VOS_StrNCat	VOS_strncat_s	strncat,_tcsncat, _mbsncat, _mbsnbcat,StrCatN, StrCatNA, StrCatNW, StrNCat, StrNCatA, StrNCatW, lstrncat, lstrcatnA, lstrcatnW, lstrcatn, _fstrncat	strncat_s
	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 _snwprintf wnsprintf, wnsprintfA, wnsprintfW	_snprintf_s _snwprintf_s sprintf_s
	vsnprintf	vsnprintf_s			_vsnprintf, _vsnwprintf,	vsnprintf_s

					_vsntprintf, wvnsprintf, wvnsprintfA, wvnsprintfW	
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）。