Tencent/secguide

https://github.com/Tencent/secguide/blob/main/C,C++安全指南.md

None Mon Jun, 28 14:26

通用安全指南

1 C/C++使用错误

1.1 【必须】不得直接使用无长度限制的字符拷贝函数

不应直接使用legacy的字符串拷贝、输入函数，如strcpy、strcat、sprintf、wcscpy、mbscpy等，这些函数的特征是：可以输出一长串字符串，而不限制长度。如果环境允许，应当使用其_s安全版本替代，或者使用n版本函数（如：snprintf，vsnprintf）。

若使用形如sscanf之类的函数时，在处理字符串输入时应当通过%10s这样的方式来严格限制字符串长度，同时确保字符串末尾有\0。如果环境允许，应当使用_s安全版本。

但是注意，虽然MSVC 2015时默认引入结尾为0版本的 `snprintf` （行为等同于C99定义的 `snprintf` ）。但更早期的版本中，MSVC的 `snprintf` 可能是 `_snprintf` 的宏。而 `_snprintf` 是不保证\0结尾的（见本节后半部分）。

```
(MSVC)
Beginning with the UCRT in Visual Studio 2015 and Windows 10, snprintf is no longer identical
to _snprintf. The snprintf function behavior is now C99 standard compliant.

从Visual Studio 2015和Windows 10中的UCRT开始，snprintf不再与_snprintf相同。snprintf函数行为现在符合C99
标准。

请参考：https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/snprintf-snprintf-
snprintf-l-snwprintf-snwprintf-l?redirectedfrom=MSDN&view=vs-2019
```

因此，在使用n系列拷贝函数时，要确保正确计算缓冲区长度，同时，如果你不确定是否代码在各个编译器下都能确保末尾有0时，建议可以适当增加1字节输入缓冲区，并将其置为\0，以保证输出的字符串结尾一定有\0。

```
// Good
char buf[101] = {0};
snprintf(buf, sizeof(buf) - 1, 'foobar ...', ...);
```

一些需要注意的函数，例如 `strncpy` 和 `_snprintf` 是不安全的。 `strncpy` 不应当被视为 `strcpy` 的n系列函数，它只是恰巧与其他n系列函数名字很像而已。 `strncpy` 在复制时，如果复制的长度超过n，不会在结尾补\0。

同样，MSVC `_snprintf` 系列函数在超过或等于n时也不会以0结尾。如果后续使用非0结尾的字符串，可能泄露相邻的内容或者导致程序崩溃。

```
// Bad
char a[4] = {0};
_snprintf(a, 4, '%s', 'AAAA');
foo = strlen(a);
```

上述代码在MSVC中执行后， a[4] == 'A'，因此字符串未以0结尾。a的内容是'AAAA'，调用 `strlen(a)` 则会越界访问。因此，正确的操作举例如下：

```
// Good
char a[4] = {0};
_snprintf(a, sizeof(a), '%s', 'AAAA');
a[sizeof(a) - 1] = '\0';
foo = strlen(a);
```

在 C++ 中，强烈建议用 `string` 、 `vector` 等更高封装层次的基础组件代替原始指针和动态数组，对提高代码的可读性和安全性都有很大的帮助。

关联漏洞:

`中风险-信息泄露`

`低风险-拒绝服务`

`高风险-缓冲区溢出`

1.2 【必须】创建进程类的函数的安全规范

system、WinExec、CreateProcess、ShellExecute等启动进程类的函数，需要严格检查其参数。

启动进程需要加上双引号，错误例子:

```
// Bad
WinExec('D:\\program files\\my folder\\foobar.exe', SW_SHOW);
```

当存在 `D:\program files\my.exe` 的时候，my.exe会被启动。而foobar.exe不会启动。

```
// Good
WinExec('\'D:\\program files\\my folder\\foobar.exe\'', SW_SHOW);
```

另外，如果启动时从用户输入、环境变量读取组合命令行时，还需要注意是否可能存在命令注入。

```
// Bad
std::string cmdline = 'calc ';
cmdline += user_input;
system(cmdline.c_str());
```

比如，当用户输入 `1+1 && ls` 时，执行的实际上是calc 1+1和ls 两个命令，导致命令注入。

需要检查用户输入是否含有非法数据。

```
// Good
std::string cmdline = 'ls ';
cmdline += user_input;

if(cmdline.find_first_not_of('1234567890.+-*/e ') == std::string::npos)
  system(cmdline.c_str());
else
  warning(...);
```

关联漏洞:

`高风险-代码执行`

高风险–权限提升

1.3 【必须】尽量减少使用 _alloca 和可变长度数组

_alloca 和可变长度数组使用的内存量在编译期间不可知。尤其是在循环中使用时，根据编译器的实现不同，可能会导致：（1）栈溢出，即拒绝服务；（2）缺少栈内存测试的编译器实现可能导致申请到非栈内存，并导致内存损坏。这在栈比较小的程序上，例如IoT设备固件上影响尤为大。对于 C++，可变长度数组也属于非标准扩展，在代码规范中禁止使用。

错误示例：

```
// Bad
for (int i = 0; i < 100000; i++) {
  char* foo = (char *)_alloca(0x10000);
  ..do something with foo ..;
}

void Foo(int size) {
  char msg[size]; // 不可控的栈溢出风险！
}
```

正确示例：

```
// Good
// 改用动态分配的堆内存
for (int i = 0; i < 100000; i++) {
  char * foo = (char *)malloc(0x10000);
  ..do something with foo ..;
  if (foo_is_no_longer_needed) {
    free(foo);
    foo = NULL;
  }
}

void Foo(int size) {
  std::string msg(size, '\0');  // C++
  char* msg = malloc(size);  // C
}
```

关联漏洞：

低风险–拒绝服务

高风险–内存破坏

1.4 【必须】printf系列参数必须对应

所有printf系列函数，如sprintf，snprintf，vprintf等必须对应控制符号和参数。

错误示例：

```
// Bad
const int buf_size = 1000;
char buffer_send_to_remote_client[buf_size] = {0};

snprintf(buffer_send_to_remote_client, buf_size, '%d: %p', id, some_string);  // %p 应为 %s

buffer_send_to_remote_client[buf_size - 1] = '\0';
send_to_remote(buffer_send_to_remote_client);
```

正确示例：

```
// Good
const int buf_size = 1000;
char buffer_send_to_remote_client[buf_size] = {0};

snprintf(buffer_send_to_remote_client, buf_size, '%d: %s', id, some_string);

buffer_send_to_remote_client[buf_size - 1] = '\0';
send_to_remote(buffer_send_to_remote_client);
```

前者可能会让client的攻击者获取部分服务器的原始指针地址，可以用于破坏ASLR保护。

关联漏洞:

中风险-信息泄露

1.5 【必须】防止泄露指针（包括%p）的值

所有printf系列函数，要防止格式化完的字符串泄露程序布局信息。例如，如果将带有%p的字符串泄露给程序，则可能会破坏ASLR的防护效果。使得攻击者更容易攻破程序。

%p的值只应当在程序内使用，而不应当输出到外部或被外部以某种方式获取。

错误示例：

```
// Bad
// 如果这是暴露给客户的一个API：
uint64_t GetUniqueObjectId(const Foo* pobject) {
  return (uint64_t)pobject;
}
```

正确示例：

```
// Good
uint64_t g_object_id = 0;

void Foo::Foo() {
  this->object_id_ = g_object_id++;
}

// 如果这是暴露给客户的一个API：
uint64_t GetUniqueObjectId(const Foo* object) {
  if (object)
    return object->object_id_;
  else
    error(...);
}
```

关联漏洞：

中风险-信息泄露

1.6 【必须】不应当把用户可修改的字符串作为printf系列函数的“format”参数

如果用户可以控制字符串，则通过 %n %p 等内容，最坏情况下可以直接执行任意恶意代码。

在以下情况尤其需要注意： WIFI名，设备名……

错误：

```
snprintf(buf, sizeof(buf), wifi_name);
```

正确：

```
snprinf(buf, sizeof(buf), '%s', wifi_name);
```

关联漏洞：

高风险-代码执行

高风险-内存破坏

中风险-信息泄露

低风险-拒绝服务

1.7 【必须】对数组delete时需要使用delete[]

delete []操作符用于删除数组。delete操作符用于删除非数组对象。它们分别调用operator delete[]和operator delete。

```
// Bad
Foo* b = new Foo[5];
delete b;  // trigger assert in DEBUG mode
```

在new[]返回的指针上调用delete将是取决于编译器的未定义行为。代码中存在对未定义行为的依赖是错误的。

```
// Good
Foo* b = new Foo[5];
delete[] b;
```

在 C++ 代码中，使用 `string`、`vector`、智能指针（比如std::unique_ptr<T[]>）等可以消除绝大多数 `delete[]` 的使用场景，并且代码更清晰。

关联漏洞:

高风险–内存破坏

中风险–逻辑漏洞

低风险–内存泄漏

低风险–拒绝服务

1.8 【必须】注意隐式符号转换

两个无符号数相减为负数时，结果应当为一个很大的无符号数，但是小于int的无符号数在运算时可能会有预期外的隐式符号转换。

```
// 1
unsigned char a = 1;
unsigned char b = 2;

if (a - b < 0)  // a - b = -1 (signed int)
  a = 6;
else
  a = 8;

// 2
unsigned char a = 1;
unsigned short b = 2;

if (a - b < 0)  // a - b = -1 (signed int)
  a = 6;
else
  a = 8;
```

上述结果均为a=6

```
// 3
unsigned int a = 1;
unsigned short b = 2;

if (a - b < 0)  // a - b = 0xffffffff (unsigned int)
  a = 6;
else
  a = 8;

// 4
unsigned int a = 1;
unsigned int b = 2;

if (a - b < 0)  // a - b = 0xffffffff (unsigned int)
  a = 6;
else
  a = 8;
```

上述结果均为a=8

如果预期为8，则错误代码：

```
// Bad
unsigned short a = 1;
unsigned short b = 2;

if (a - b < 0)  // a - b = -1 (signed int)
  a = 6;
else
  a = 8;
```

正确代码:

```
// Good
unsigned short a = 1;
unsigned short b = 2;

if ((unsigned int)a - (unsigned int)b < 0)  // a - b = 0xffff (unsigned short)
  a = 6;
else
  a = 8;
```

关联漏洞:

中风险-逻辑漏洞

1.9【必须】注意八进制问题

代码对齐时应当使用空格或者编辑器自带的对齐功能，谨慎在数字前使用0来对齐代码，以免不当将某些内容转换为八进制。

例如，如果预期为20字节长度的缓冲区，则下列代码存在错误。buf2为020（OCT）长度，实际只有16（DEC）长度，在memcpy后越界:

```
// Bad
char buf1[1024] = {0};
char buf2[0020] = {0};

memcpy(buf2, somebuf, 19);
```

应当在使用8进制时明确注明这是八进制。

```
// Good
int access_mask = 0777;  // oct, rwxrwxrwx
```

关联漏洞:

中风险-逻辑漏洞

2 不推荐的编程习惯

2.1 【必须】switch中应有default

switch中应该有default，以处理各种预期外的情况。这可以确保switch接受用户输入，或者后期在其他开发者修改函数后确保switch仍可以覆盖到所有情况，并确保逻辑正常运行。

```
// Bad
int Foo(int bar) {
  switch (bar & 7) {
    case 0:
      return Foobar(bar);
      break;
    case 1:
      return Foobar(bar * 2);
      break;
  }
}
```

例如上述代码switch的取值可能从0～7，所以应当有default：

```
// Good
int Foo(int bar) {
  switch (bar & 7) {
    case 0:
      return Foobar(bar);
      break;
    case 1:
      return Foobar(bar * 2);
      break;
    default:
      return -1;
  }
}
```

关联漏洞：

中风险-逻辑漏洞

中风险-内存泄漏

2.2 【必须】不应当在Debug或错误信息中提供过多内容

包含过多信息的Debug消息不应当被用户获取到。Debug信息可能会泄露一些值，例如内存数据、内存地址等内容，这些内容可以帮助攻击者在初步控制程序后，更容易地攻击程序。

```
// Bad
int Foo(int* bar) {
  if (bar && *bar == 5) {
    OutputDebugInfoToUser('Wrong value for bar %p = %d\n', bar, *bar);
  }
}
```

而应该：

```
// Good
int foo(int* bar) {

#ifdef DEBUG
  if (bar && *bar == 5) {
    OutputDebugInfo('Wrong value for bar.\n', bar, *bar);
  }
#endif

}
```

关联漏洞：

中风险-信息泄漏

2.3 【必须】不应该在客户端代码中硬编码对称加密秘钥

不应该在客户端代码中硬编码对称加密秘钥。例如：不应在客户端代码使用硬编码的 AES/ChaCha20-Poly1305/SM1 密钥，使用固定密钥的程序基本和没有加密一样。

如果业务需求是认证加密数据传输，应优先考虑直接用 HTTPS 协议。

如果是其它业务需求，可考虑由服务器端生成对称秘钥，客户端通过 HTTPS 等认证加密通信渠道从服务器拉取。

或者根据用户特定的会话信息，比如登录认证过程可以根据用户名用户密码业务上下文等信息，使用 HKDF 等算法衍生出对称秘钥。

又或者使用 RSA/ECDSA + ECDHE 等进行认证秘钥协商，生成对称秘钥。

```
// Bad
char g_aes_key[] = {...};

void Foo() {
  ....
  AES_func(g_aes_key, input_data, output_data);
}
```

可以考虑在线为每个用户获取不同的密钥:

```
// Good
char* g_aes_key;

void Foo() {
  ....
  AES_encrypt(g_aes_key, input_data, output_data);
}

void Init() {
  g_aes_key = get_key_from_https(user_id, ...);
}
```

关联漏洞:

中风险-信息泄露

2.4 【必须】返回栈上变量的地址

函数不可以返回栈上的变量的地址，其内容在函数返回后就会失效。

```
// Bad
char* Foo(char* sz, int len){
  char a[300] = {0};
  if (len > 100) {
    memcpy(a, sz, 100);
  }
  a[len] = '\0';
  return a;  // WRONG
}
```

而应当使用堆来传递非简单类型变量。

```
// Good
char* Foo(char* sz, int len) {
    char* a = new char[300];
    if (len > 100) {
        memcpy(a, sz, 100);
    }
    a[len] = '\0';
    return a;  // OK
}
```

对于 C++ 程序来说，强烈建议返回 `string`、`vector` 等类型，会让代码更加简单和安全。

关联漏洞:

高风险–内存破坏

2.5 【必须】有逻辑联系的数组必须仔细检查

例如下列程序将字符串转换为week day，但是两个数组并不一样长，导致程序可能会越界读一个int。

```
// Bad
int nWeekdays[] = {1, 2, 3, 4, 5, 6};
const char* sWeekdays[] = {'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'};
for (int x = 0; x < ARRAY_SIZE(sWeekdays); x++) {
  if (strcmp(sWeekdays[x], input) == 0)
    return nWeekdays[x];
}
```

应当确保有关联的nWeekdays和sWeekdays数据统一。

```
// Good
const int nWeekdays[] = {1, 2, 3, 4, 5, 6, 7};
const char* sWeekdays[] = {'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'};
assert(ARRAY_SIZE(nWeekdays) == ARRAY_SIZE(sWeekdays));
for (int x = 0; x < ARRAY_SIZE(sWeekdays); x++) {
  if (strcmp(sWeekdays[x], input) == 0) {
    return nWeekdays[x];
  }
}
```

关联漏洞:

高风险–内存破坏

2.6 【必须】避免函数的声明和实现不同

在头文件、源代码、文档中列举的函数声明应当一致，不应当出现定义内容错位的情况。

错误:

foo.h

```
int CalcArea(int width, int height);
```

foo.cc

```
int CalcArea(int height, int width) {  // Different from foo.h
  if (height > real_height) {
    return 0;
  }
  return height * width;
}
```

正确： foo.h

```
int CalcArea(int height, int width);
```

foo.cc

```
int CalcArea (int height, int width) {
  if (height > real_height) {
    return 0;
  }
  return height * width;
}
```

关联漏洞:

中风险-逻辑问题

2.7 【必须】检查复制粘贴的重复代码（相同代码通常代表错误）

当开发中遇到较长的句子时，如果你选择了复制粘贴语句，请记得检查每一行代码，不要出现上下两句一模一样的情况，这通常代表代码哪里出现了错误:

```
// Bad
void Foobar(SomeStruct& foobase, SomeStruct& foo1, SomeStruct& foo2) {
  foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
  foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
}
```

如上例，通常可能是:

```
// Good
void Foobar(SomeStruct& foobase, SomeStruct& foo1, SomeStruct& foo2) {
  foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
  foo2.bar = (foo2.bar & 0xffff) | (foobase.base & 0xffff0000);
}
```

最好是把重复的代码片段提取成函数，如果函数比较短，可以考虑定义为 `inline` 函数，在减少冗余的同时也能确保不会影响性能。

关联漏洞:

中风险-逻辑问题

2.8 【必须】左右一致的重复判断/永远为真或假的判断（通常代表错误）

这通常是由于自动完成或例如Visual Assistant X之类的补全插件导致的问题。

```
// Bad
if (foo1.bar == foo1.bar) {
  …
}
```

可能是:

```
// Good
if (foo1.bar == foo2.bar) {
  …
}
```

关联漏洞:

中风险-逻辑问题

2.9 【必须】函数每个分支都应有返回值

函数的每个分支都应该有返回值，否则如果函数走到无返回值的分支，其结果是未知的。

```
// Bad
int Foo(int bar) {
  if (bar > 100) {
    return 10;
  } else if (bar > 10) {
    return 1;
  }
}
```

上述例子当bar<10时，其结果是未知的值。

```
// Good
int Foo(int bar) {
  if (bar > 100) {
    return 10;
  } else if (bar > 10) {
    return 1;
  }
  return 0;
}
```

开启适当级别的警告（GCC 中为 `-Wreturn-type` 并已包含在 `-Wall` 中）并设置为错误，可以在编译阶段发现这类错误。

关联漏洞:

中风险-逻辑问题

中风险-信息泄漏

2.10 【必须】不得使用栈上未初始化的变量

在栈上声明的变量要注意是否在使用它之前已经初始化了

```
// Bad
void Foo() {
  int foo;
  if (Bar()) {
    foo = 1;
  }
  Foobar(foo); // foo可能没有初始化
}
```

最好在声明的时候就立刻初始化变量，或者确保每个分支都初始化它。开启相应的编译器警告（GCC 中为 `-Wuninitialized`），并把设置为错误级别，可以在编译阶段发现这类错误。

```
// Good
void Foo() {
  int foo = 0;
  if (Bar()) {
    foo = 1;
  }
  Foobar(foo);
}
```

关联漏洞:

中风险-逻辑问题

中风险-信息泄漏

2.11 【建议】不得直接使用刚分配的未初始化的内存（如realloc）

一些刚申请的内存通常是直接从堆上分配的，可能包含有旧数据的，直接使用它们而不初始化，可能会导致安全问题。例如，CVE-2019-13751。应确保初始化变量，或者确保未初始化的值不会泄露给用户。

```
// Bad
char* Foo() {
  char* a = new char[100];
  a[99] = '\0';
  memcpy(a, 'char', 4);
  return a;
}
```

```
// Good
char* Foo() {
  char* a = new char[100];
  memcpy(a, 'char', 4);
  a[4] = '\0';
  return a;
}
```

在 C++ 中，再次强烈推荐用 `string`、`vector` 代替手动内存分配。

关联漏洞:

中风险-逻辑问题

中风险-信息泄漏

2.12 【必须】校验内存相关函数的返回值

与内存分配相关的函数需要检查其返回值是否正确，以防导致程序崩溃或逻辑错误。

```
// Bad
void Foo() {
  char* bar = mmap(0, 0x800000, .....);
  *(bar + 0x400000) = '\x88'; // Wrong
}
```

如上例mmap如果失败，bar的值将是0xffffffff (ffffffff)，第二行将会往0x3ffffff写入字符，导致越界写。

```
// Good
void Foo() {
  char* bar = mmap(0, 0x800000, .....);
  if(bar == MAP_FAILED) {
    return;
  }

  *(bar + 0x400000) = '\x88';
}
```

关联漏洞:

中风险-逻辑问题

高风险-越界操作

2.13 【必须】不要在if里面赋值

if里赋值通常代表代码存在错误。

```
// Bad
void Foo() {
  if (bar = 0x99) ...
}
```

通常应该是:

```
// Good
void Foo() {
  if (bar == 0x99) ...
}
```

建议在构建系统中开启足够的编译器警告（GCC 中为 `-Wparentheses` 并已包含在 `-Wall` 中），并把该警告设置为错误。

关联漏洞:

中风险-逻辑问题

2.14 【建议】确认if里面的按位操作

if里，非bool类型和非bool类型的按位操作可能代表代码存在错误。

```
// Bad
void Foo() {
  int bar = 0x1;     // binary 01
  int foobar = 0x2;    // binary 10

  if (foobar & bar)     // result = 00, false
    ...
}
```

上述代码可能应该是:

```
// Good
void foo() {
  int   bar = 0x1;
  int foobar = 0x2;

  if (foobar && bar)  // result : true
    ...
}
```

关联漏洞:

中风险-逻辑问题

3 多线程

3.1 【必须】变量应确保线程安全性

当一个变量可能被多个线程使用时，应当使用原子操作或加锁操作。

```
// Bad
char  g_somechar;
void foo_thread1() {
  g_somechar += 3;
}

void foo_thread2() {
  g_somechar += 1;
}
```

对于可以使用原子操作的，应当使用一些可以确保内存安全的操作，如:

```
// Good
volatile char g_somechar;
void foo_thread1() {
  __sync_fetch_and_add(&g_somechar, 3);
}

void foo_thread2() {
  __sync_fetch_and_add(&g_somechar, 1);
}
```

对于 C 代码， `C11` 后推荐使用 atomic 标准库。 对于 C++代码， `C++11` 后，推荐使用 `std::atomic`。

关联漏洞:

高风险-内存破坏

中风险–逻辑问题

3.2 【必须】注意signal handler导致的条件竞争

竞争条件经常出现在信号处理程序中，因为信号处理程序支持异步操作。攻击者能够利用信号处理程序争用条件导致软件状态损坏，从而可能导致拒绝服务甚至代码执行。

1. 当信号处理程序中发生不可重入函数或状态敏感操作时，就会出现这些问题。因为信号处理程序中随时可以被调用。比如，当在信号处理程序中调用 `free` 时，通常会出现另一个信号争用条件，从而导致双重释放。即使给定指针在释放后设置为 `NULL`，在释放内存和将指针设置为 `NULL` 之间仍然存在竞争的可能。
2. 为多个信号设置了相同的信号处理程序，这尤其有问题——因为这意味着信号处理程序本身可能会重新进入。例如，malloc()和free()是不可重入的，因为它们可能使用全局或静态数据结构来管理内存，并且它们被syslog()等看似无害的函数间接使用；这些函数可能会导致内存损坏和代码执行。

```
// Bad
char *log_message;

void Handler(int signum) {
  syslog(LOG_NOTICE, '%s\n', log_m_essage);
  free(log_message);
  sleep(10);
  exit(0);
}

int main (int argc, char* argv[]) {
  log_message = strdup(argv[1]);
  signal(SIGHUP, Handler);
  signal(SIGTERM, Handler);
  sleep(10);
}
```

可以借由下列操作规避问题：

1. 避免在多个处理函数中共享某些变量。
2. 在信号处理程序中使用同步操作。
3. 屏蔽不相关的信号，从而提供原子性。
4. 避免在信号处理函数中调用不满足异步信号安全的函数。

关联漏洞：

高风险–内存破坏

中风险–逻辑问题

3.3 【建议】注意Time-of-check Time-of-use (TOCTOU) 条件竞争

TOCTOU: 软件在使用某个资源之前检查该资源的状态，但是该资源的状态可以在检查和使用之间更改，从而使检查结果无效。当资源处于这种意外状态时，这可能会导致软件执行错误操作。

当攻击者可以影响检查和使用之间的资源状态时，此问题可能与安全相关。这可能发生在共享资源(如**文件、内存**，甚至多线程程序中的**变量**)上。在编程时需要注意避免出现TOCTOU问题。

例如，下面的例子中，该文件可能已经在检查和lstat之间进行了更新，特别是因为printf有延迟。

```
struct stat *st;

lstat('...', st);

printf('foo');

if (st->st_mtimespec == ...) {
  printf('Now updating things\n');
  UpdateThings();
}
```

TOCTOU难以修复，但是有以下缓解方案:

1. 限制对来自多个进程的文件的交叉操作。
2. 如果必须在多个进程或线程之间共享对资源的访问，那么请尝试限制”检查“（CHECK）和”使用“（USE）资源之间的时间量，使他们相距尽量不要太远。这不会从根本上解决问题，但可能会使攻击更难成功。
3. 在Use调用之后重新检查资源，以验证是否正确执行了操作。
4. 确保一些环境锁定机制能够被用来有效保护资源。但要确保锁定是检查之前进行的，而不是在检查之后进行的，以便检查时的资源与使用时的资源相同。

关联漏洞:

高风险–内存破坏

中风险–逻辑问题

4 加密解密

4.1 【必须】不得明文存储用户密码等敏感数据

用户密码应该使用 Argon2, scrypt, bcrypt, pbkdf2 等算法做哈希之后再存入存储系统, https://password-hashing.net/

https://libsodium.gitbook.io/doc/password_hashing/default_phf#example-2-password-storage

用户敏感数据，应该做到传输过程中加密，存储状态下加密 传输过程中加密，可以使用 HTTPS 等认证加密通信协议

存储状态下加密，可以使用 SQLCipher 等类似方案。

4.2 【必须】内存中的用户密码等敏感数据应该安全抹除

例如用户密码等，即使是临时使用，也应在使用完成后应当将内容彻底清空。

错误:

```
#include <openssl/crypto.h>
#include <unistd.h>

   {
       ...
       string user_password(100, '\0');
       snprintf(&user_password, 'password: %s', user_password.size(), password_from_input);
       ...
   }
```

正确:

```
   {
       ...
       string user_password(100, '\0');
       snprintf(&user_password, 'password: %s', user_password.size(), password_from_input);
       ...
       OPENSSL_cleanse(&user_password[0], user_password.size());
   }
```

关联漏洞:

高风险–敏感信息泄露

4.3 【必须】rand() 类函数应正确初始化

rand类函数的随机性并不高。而且在使用前需要使用srand()来初始化。未初始化的随机数可能导致某些内容可预测。

```
// Bad
int main() {
  int foo = rand();
  return 0;
}
```

上述代码执行完成后，foo的值是固定的。它等效于 `srand(1); rand();` 。

```
// Good

int main() {
  srand(time(0));
  int foo = rand();
  return 0;
}
```

关联漏洞:

高风险-逻辑漏洞

4.4 【必须】在需要高强度安全加密时不应使用弱PRNG函数

在需要生成 AES/SM1/HMAC 等算法的密钥/IV/Nonce， RSA/ECDSA/ECDH 等算法的私钥，这类需要高安全性的业务场景，必须使用密码学安全的随机数生成器 (Cryptographically Secure PseudoRandom Number Generator (CSPRNG))，不得使用 `rand()` 等无密码学安全性保证的普通随机数生成器。

推荐使用的 CSPRNG 有:

1. OpenSSL 中的 `RAND_bytes()` 函数, https://www.openssl.org/docs/man1.1.1/man3/RAND_bytes.html
2. libsodium 中的 `randombytes_buf()` 函数
3. Linux kernel 的 `getrandom()` 系统调用, https://man7.org/linux/man-pages/man2/getrandom.2.html . 或者读 /dev/urandom 文件, 或者 /dev/random 文件。
4. Apple IOS 的 `SecRandomCopyBytes()` , https://developer.apple.com/documentation/security/1399291-secrandomcopybytes

5. Windows 下的 `BCryptGenRandom()`，`CryptGenRandom()`，`RtlGenRandom()`

```
#include <openssl/aes.h>
#include <openssl/crypto.h>
#include <openssl/rand.h>
#include <unistd.h>

    {
        unsigned char key[16];
        if (1 != RAND_bytes(&key[0], sizeof(key))) {  //... 错误处理
            return -1;
        }

        AES_KEY aes_key;
        if (0 != AES_set_encrypt_key(&key[0], sizeof(key) * 8, &aes_key)) {
            // ... 错误处理
            return -1;
        }

        ...

        OPENSSL_cleanse(&key[0], sizeof(key));
    }
```

`rand()` 类函数的随机性并不高。敏感操作时，如设计加密算法时，不得使用rand()或者类似的简单线性同余伪随机数生成器来作为随机数发生器。符合该定义的比特序列的特点是，序列中“1”的数量约等于“0”的数量；同理，“01”、“00”、“10”、“11”的数量大致相同，以此类推。

例如 C 标准库中的 `rand()` 的实现只是简单的线性同余算法，生成的伪随机数具有较强的可预测性。

当需要实现高强度加密，例如涉及通信安全时，不应当使用 `rand()` 作为随机数发生器。

实际应用中，C++11 标准提供的`random_device`保证加密的安全性和随机性 但是 C++ 标准并不保证这一点。跨平台的代码可以考虑用 OpenSSL 等保证密码学安全的库里的随机数发生器。

关联漏洞:

高风险-敏感数据泄露

4.5 【必须】自己实现的rand范围不应过小

如果在弱安全场景相关的算法中自己实现了PRNG，请确保rand出来的随机数不会很小或可预测。

```
// Bad
int32_t val = ((state[0] * 1103515245U) + 12345U) & 999999;
```

上述例子可能想生成0~999999共100万种可能的随机数，但是999999的二进制是11110100001000111111，与&运算后，0位一直是0，所以生成出的范围明显会小于100万种。

```
// Good
int32_t val = ((state[0] * 1103515245U) + 12345U) % 1000000;

// Good
int32_t val = ((state[0] * 1103515245U) + 12345U) & 0x7fffffff;
```

关联漏洞:

高风险–逻辑漏洞

5 文件操作

5.1 【必须】避免路径穿越问题

在进行文件操作时，需要判断外部传入的文件名是否合法，如果文件名中包含 `../` 等特殊字符，则会造成路径穿越，导致任意文件的读写。

错误:

```
void Foo() {
  char file_path[PATH_MAX] = '/home/user/code/';
  // 如果传入的文件名包含../可导致路径穿越
  // 例如'../file.txt'，则可以读取到上层目录的file.txt文件
  char name[20] = '../file.txt';
  memcpy(file_path + strlen(file_path), name, sizeof(name));
  int fd = open(file_path, O_RDONLY);
  if (fd != -1) {
    char data[100] = {0};
    int num = 0;
    memset(data, 0, sizeof(data));
    num = read(fd, data, sizeof(data));
    if (num > 0) {
      write(STDOUT_FILENO, data, num);
    }
    close(fd);
  }
}
```

正确:

```
void Foo() {
  char file_path[PATH_MAX] = '/home/user/code/';
  char name[20] = '../file.txt';
  // 判断传入的文件名是否非法，例如'../file.txt'中包含非法字符../，直接返回
  if (strstr(name, '..') != NULL){
    // 包含非法字符
    return;
  }
  memcpy(file_path + strlen(file_path), name, sizeof(name));
  int fd = open(file_path, O_RDONLY);
  if (fd != -1) {
    char data[100] = {0};
    int num = 0;
    memset(data, 0, sizeof(data));
    num = read(fd, data, sizeof(data));
    if (num > 0) {
      write(STDOUT_FILENO, data, num);
    }
    close(fd);
   }
}
```

关联漏洞:

高风险-逻辑漏洞

5.2 【必须】避免相对路径导致的安全问题（DLL、EXE劫持等问题）

在程序中，使用相对路径可能导致一些安全风险，例如DLL、EXE劫持等问题。

例如以下代码，可能存在劫持问题:

```
int Foo() {
  // 传入的是dll文件名，如果当前目录下被写入了恶意的同名dll，则可能导致dll劫持
  HINSTANCE hinst = ::LoadLibrary('dll_nolib.dll');
  if (hinst != NULL) {
    cout<<'dll loaded!' << endl;
  }
  return 0;
}
```

针对DLL劫持的安全编码的规范:

1）调用LoadLibrary，LoadLibraryEx，CreateProcess，ShellExecute等进行模块加载的函数时，指明模块的完整（全）路径，禁止使用相对路径，这样就可避免从其它目录加载DLL。 2）在应用程序的开头调用SetDllDirectory(TEXT(''));从而将当前目录从DLL的搜索列表中删除。结合SetDefaultDllDirectories，AddDllDirectory，RemoveDllDirectory这几个API配合使用，可以有效的规避DLL劫持问题。这些API只能在打了KB2533623补丁的Windows7，2008上使用。

关联漏洞:

中风险-逻辑漏洞

5.3 【必须】文件权限控制

在创建文件时，需要根据文件的敏感级别设置不同的访问权限，以防止敏感数据被其他恶意程序读取或写入。

错误:

```
int Foo() {
  // 不要设置为777权限，以防止被其他恶意程序操作
  if (creat('file.txt', 0777) < 0) {
    printf('文件创建失败！\n');
  } else {
    printf('文件创建成功！\n');
  }
  return 0;
}
```

关联漏洞:

中风险-逻辑漏洞

6 内存操作

6.1 【必须】防止各种越界写（向前/向后）

错误1:

错误2:

```
int a[5];
int b = user_controlled_value;
a[b] = 3;
```

关联漏洞:

高风险-内存破坏

6.2 【必须】防止任意地址写

任意地址写会导致严重的安全隐患，可能导致代码执行。因此，在编码时必须校验写入的地址。

错误：

```
void Write(MyStruct dst_struct) {
  char payload[10] = { 0 };
  memcpy(dst_struct.buf, payload, sizeof(payload));
}

int main() {
  MyStruct dst_stuct;
  dst_stuct.buf = (char*)user_controlled_value;
  Write(dst_stuct);
  return 0;
}
```

关联漏洞：

高风险-内存破坏

7 数字操作

7.1 【必须】防止整数溢出

在计算时需要考虑整数溢出的可能，尤其在进行内存操作时，需要对分配、拷贝等大小进行合法校验，防止整数溢出导致的漏洞。

错误（该例子在计算时产生整数溢出）

```
const int kMicLen = 4;
// 整数溢出
void Foo() {
  int len = 1;
  char payload[10] = { 0 };
  char dst[10] = { 0 };
  // Bad, 由于len小于4字节，导致计算拷贝长度时，整数溢出
  // len - MIC_LEN == 0xfffffffd
  memcpy(dst, payload, len - kMicLen);
}
```

正确例子

```
void Foo() {
  int len = 1;
  char payload[10] = { 0 };
  char dst[10] = { 0 };
  int size = len - kMicLen;
  // 拷贝前对长度进行判断
  if (size > 0 && size < 10) {
    memcpy(dst, payload, size);
    printf('memcpy good\n');
  }
}
```

关联漏洞:

高风险-内存破坏

7.2 【必须】防止Off-By-One

在进行计算或者操作时，如果使用的最大值或最小值不正确，使得该值比正确值多1或少1，可能导致安全风险。

错误:

```
char firstname[20];
char lastname[20];
char fullname[40];

fullname[0] = '\0';

strncat(fullname, firstname, 20);
// 第二次调用strncat()可能会追加另外20个字符。如果这20个字符没有终止空字符，则存在安全问题
strncat(fullname, lastname, 20);
```

正确:

```
char firstname[20];
char lastname[20];
char fullname[40];

fullname[0] = '\0';

// 当使用像strncat()函数时，必须在缓冲区的末尾为终止空字符留下一个空字节，避免off-by-one
strncat(fullname, firstname, sizeof(fullname) - strlen(fullname) - 1);
strncat(fullname, lastname, sizeof(fullname) - strlen(fullname) - 1);
```

对于 C++ 代码，再次强烈建议使用 `string`、`vector` 等组件代替原始指针和数组操作。

关联漏洞:

高风险-内存破坏

7.3 【必须】避免大小端错误

在一些涉及大小端数据处理的场景，需要进行大小端判断，例如从大端设备取出的值，要以大端进行处理，避免端序错误使用。

关联漏洞:

中风险-逻辑漏洞

7.4 【必须】检查除以零异常

在进行除法运算时，需要判断被除数是否为零，以防导致程序不符合预期或者崩溃。

错误:

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

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

正确:

```
double divide(double x, double y) {
  if (y == 0) {
    throw DivideByZero;
  }
  return x / y;
}
```

关联漏洞:

低风险-拒绝服务

7.5 【必须】防止数字类型的错误强转

在有符号和无符号数字参与的运算中，需要注意类型强转可能导致的逻辑错误，建议指定参与计算时数字的类型或者统一类型参与计算。

错误例子

```
int Foo() {
  int len = 1;
  unsigned int size = 9;
  // 1 < 9 - 10 ? 由于运算中无符号和有符号混用，导致计算结果以无符号计算
  if (len < size - 10) {
    printf('Bad\n');
  } else {
    printf('Good\n');
  }
}
```

正确例子

```
void Foo() {
  // 统一两者计算类型为有符号
  int len = 1;
  int size = 9;
  if (len < size - 10) {
    printf('Bad\n');
  } else {
    printf('Good\n');
  }
}
```

关联漏洞:

高风险-内存破坏

中风险-逻辑漏洞

7.6 【必须】比较数据大小时加上最小/最大值的校验

在进行数据大小比较时，要合理地校验数据的区间范围，建议根据数字类型，对其进行最大和最小值的判断，以防止非预期错误。

错误:

```
void Foo(int index) {
  int a[30] = {0};
  // 此处index是int型，只考虑了index小于数组大小，但是并未判断是否大于0
  if (index < 30) {
    // 如果index为负数，则越界
    a[index] = 1;
  }
}
```

正确:

```
void Foo(int index) {
  int a[30] = {0};
  // 判断index的最大最小值
  if (index >=0 && index < 30) {
    a[index] = 1;
  }
}
```

关联漏洞:

高风险-内存破坏

8 指针操作

8.1 【建议】检查在pointer上使用sizeof

除了测试当前指针长度，否则一般不会在pointer上使用sizeof。

正确:

```
size_t pointer_length = sizeof(void*);
```

可能错误:

```
size_t structure_length = sizeof(Foo*);
```

可能是:

```
size_t structure_length = sizeof(Foo);
```

关联漏洞:

中风险-逻辑漏洞

8.2 【必须】检查直接将数组和0比较的代码

错误:

```
int a[3];
...;

if (a > 0)
  ...;
```

该判断永远为真，等价于:

```
int a[3];
...;

if (&a[0])
  ...;
```

可能是:

```
int a[3];
...;

if(a[0] > 0)
  ...;
```

开启足够的编译器警告（GCC 中为 `-Waddress`，并已包含在 `-Wall` 中），并设置为错误，可以在编译期间发现该问题。

关联漏洞:

中风险-逻辑漏洞

8.3 【必须】不应当向指针赋予写死的地址

特殊情况需要特殊对待（比如开发硬件固件时可能需要写死）

但是如果是系统驱动开发之类的，写死可能会导致后续的问题。

关联漏洞:

高风险-内存破坏

8.4 【必须】检查空指针

错误:

```
*foo = 100;

if (!foo) {
  ERROR('foobar');
}
```

正确:

```
if (!foo) {
  ERROR('foobar');
}

*foo = 100;
```

错误：

```
void Foo(char* bar) {
  *bar = '\0';
}
```

正确：

```
void Foo(char* bar) {
  if(bar)
    *bar = '\0';
  else
    ...;
}
```

关联漏洞：

低风险-拒绝服务

8.5 【必须】释放完后置空指针

在对指针进行释放后，需要将该指针设置为NULL，以防止后续free指针的误用，导致UAF等其他内存破坏问题。尤其是在结构体、类里面存储的原始指针。

错误：

```
void foo() {
  char* p = (char*)malloc(100);
  memcpy(p, 'hello', 6);
  // 此时p所指向的内存已被释放，但是p所指的地址仍然不变
  printf('%s\n', p);
  free(p);
  // 未设置为NULL，可能导致UAF等内存错误

  if (p != NULL) {  // 没有起到防错作用
    printf('%s\n', p); // 错误使用已经释放的内存
  }
}
```

正确：

```
void foo() {
  char* p = (char*)malloc(100);
  memcpy(p, 'hello', 6);
  // 此时p所指向的内存已被释放，但是p所指的地址仍然不变
  printf('%s\n', p);
  free(p);
  //释放后将指针赋值为空
  p = NULL;
  if (p != NULL)  { // 没有起到防错作用
    printf('%s\n', p); // 错误使用已经释放的内存
  }
}
```

对于 C++ 代码，使用 string、vector、智能指针等代替原始内存管理机制，可以大量减少这类错误。

关联漏洞:

高风险–内存破坏

8.6 【必须】防止错误的类型转换（type confusion）

在对指针、对象或变量进行操作时，需要能够正确判断所操作对象的原始类型。如果使用了与原始类型不兼容的类型进行访问，则存在安全隐患。

错误:

```
const int NAME_TYPE = 1;
const int ID_TYPE = 2;

// 该类型根据 msg_type 进行区分，如果在对MessageBuffer进行操作时没有判断目标对象，则存在类型混淆
struct MessageBuffer {
  int msg_type;
  union {
    const char *name;
    int name_id;
  };
};

void Foo() {
  struct MessageBuffer buf;
  const char* default_message = 'Hello World';
  // 设置该消息类型为 NAME_TYPE，因此buf预期的类型为 msg_type + name
  buf.msg_type = NAME_TYPE;
  buf.name = default_message;
  printf('Pointer of buf.name is %p\n', buf.name);

  // 没有判断目标消息类型是否为ID_TYPE，直接修改nameID，导致类型混淆
  buf.name_id = user_controlled_value;

  if (buf.msg_type == NAME_TYPE) {
    printf('Pointer of buf.name is now %p\n', buf.name);
    // 以NAME_TYPE作为类型操作，可能导致非法内存读写
    printf('Message: %s\n', buf.name);
  } else {
    printf('Message: Use ID %d\n', buf.name_id);
  }
}
```

正确（判断操作的目标是否是预期类型）：

```
void Foo() {
  struct MessageBuffer buf;
  const char* default_message = 'Hello World';
  // 设置该消息类型为 NAME_TYPE, 因此buf预期的类型为 msg_type + name
  buf.msg_type = NAME_TYPE;
  buf.name = default_msessage;
  printf('Pointer of buf.name is %p\n', buf.name);

  // 判断目标消息类型是否为 ID_TYPE, 不是预期类型则做对应操作
  if (buf.msg_type == ID_TYPE)
    buf.name_id = user_controlled_value;

  if (buf.msg_type == NAME_TYPE) {
    printf('Pointer of buf.name is now %p\n', buf.name);
    printf('Message: %s\n', buf.name);
  } else {
    printf('Message: Use ID %d\n', buf.name_id);
  }
}
```

关联漏洞:

高风险-内存破坏

8.7 【必须】智能指针使用安全

在使用智能指针时，防止其和原始指针的混用，否则可能导致对象生命周期问题，例如 UAF 等安全风险。

错误例子:

```
class Foo {
 public:
  explicit Foo(int num) { data_ = num; };
  void Function() { printf('Obj is %p, data = %d\n', this, data_); };
 private:
  int data_;
};

std::unique_ptr<Foo> fool_u_ptr = nullptr;
Foo* pfool_raw_ptr = nullptr;

void Risk() {
  fool_u_ptr = make_unique<Foo>(1);

  // 从独占智能指针中获取原始指针,<Foo>(1)
  pfool_raw_ptr = fool_u_ptr.get();
  // 调用<Foo>(1)的函数
  pfool_raw_ptr->Function();

  // 独占智能指针重新赋值后会释放内存
  fool_u_ptr = make_unique<Foo>(2);
  // 通过原始指针操作会导致UAF, pfool_raw_ptr指向的对象已经释放
  pfool_raw_ptr->Function();
}


// 输出:
// Obj is 0000027943087B80, data = 1
// Obj is 0000027943087B80, data = -572662307
```

正确, 通过智能指针操作:

```
void Safe() {
  fool_u_ptr = make_unique<Foo>(1);
  // 调用<Foo>(1)的函数
  fool_u_ptr->function();

  fool_u_ptr = make_unique<Foo>(2);
  // 调用<Foo>(2)的函数
  fool_u_ptr->function();
}

// 输出:
// Obj is 000002C7BB550830, data = 1
// Obj is 000002C7BB557AF0, data = 2
```

关联漏洞:

高风险-内存破坏