鹅厂(腾讯)代码安全指南

面向开发人员梳理的代码安全指南，旨在梳理API层面的风险点并提供详实可行的安全编码方案。基于DevSecOps理念，我们希望用开发者更易懂的方式阐述安全编码方案，引导从源头规避漏洞。

目　录

致谢

当前文档 《鹅厂(腾讯)代码安全指南》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2021-05-26。

内容来源：腾讯 https://github.com/Tencent/secguide

文档地址：http://www.bookstack.cn/books/Tencent-secguide

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

代码安全指南

面向开发人员梳理的代码安全指南，旨在梳理API层面的风险点并提供详实可行的安全编码方案。

理念

基于DevSecOps理念，我们希望用开发者更易懂的方式阐述安全编码方案，引导从源头规避漏洞。

索引

规范	最后修订日期
C/C++安全指南	2021-05-18
JavaScript安全指南	2021-05-18
Node安全指南	2021-05-18
Go安全指南	2021-05-18
Java安全指南	2021-05-18
Python安全指南	2021-05-18

实践

代码安全指引可用于以下场景：

- 开发人员日常参考
- 编写安全系统扫描策略
- 安全组件开发
- 漏洞修复指引

贡献

欢迎通过Issue或PR的方式提交修订建议，示例如下：

```
1. 标题：#JavaScript# 规范1.3.1条修订建议
2.
3. 内容：
4. 1、问题描述
5. JavaScript代码安全规范的【1.3.1条】赋值或更新HTML属性部分，需补充
6.
7. 2、解决建议
```

8. 应补充下列风险点：
9. area.href、input.formaction、button.formaction

授权许可

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）。

```
1.  （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
2.  standard compliant.
3.
    从Visual Studio 2015和Windows 10中的UCRT开始，snprintf不再与_snprintf相同。snprintf
4.  函数行为现在符合C99标准。
5.
    请参考：https://docs.microsoft.com/en-us/cpp/c-runtime-
    library/reference/snprintf-snprintf-snprintf-l-snwprintf-snwprintf-l?
6.  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。

```
1.  // Good
2.  char buf[101] = {0};
3.  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};
```

```
4.
5.  snprintf(buffer_send_to_remote_client, buf_size, "%d: %s", id, some_string);
6.
7.  buffer_send_to_remote_client[buf_size - 1] = '\0';
8.  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：

```
1.  // Bad
2.  // 如果这是暴露给客户的一个API：
3.  uint64_t GetUniqueObjectId(const Foo* pobject) {
4.    return (uint64_t)pobject;
5.  }
```

正确示例：

```
1.  // Good
2.  uint64_t g_object_id = 0;
3.
4.  void Foo::Foo() {
5.    this->object_id_ = g_object_id++;
6.  }
7.
8.  // 如果这是暴露给客户的一个API：
9.  uint64_t GetUniqueObjectId(const Foo* object) {
10.   if (object)
11.     return object->object_id_;
12.   else
13.     error(...);
14. }
```

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`）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.  // 1
2.  unsigned char a = 1;
3.  unsigned char b = 2;
4.
5.  if (a - b < 0)  // a - b = -1 (signed int)
6.    a = 6;
7.  else
8.    a = 8;
9.
10. // 2
11. unsigned char a = 1;
12. unsigned short b = 2;
13.
14. if (a - b < 0)  // a - b = -1 (signed int)
15.   a = 6;
16. else
17.   a = 8;
```

上述结果均为a=6

```
1.  // 3
2.  unsigned int a = 1;
3.  unsigned short b = 2;
4.
5.  if (a - b < 0)  // a - b = 0xffffffff (unsigned int)
6.    a = 6;
```

```
7.  else
8.    a = 8;
9.
10. // 4
11. unsigned int a = 1;
12. unsigned int b = 2;
13.
14. if (a - b < 0)  // a - b = 0xffffffff (unsigned int)
15.   a = 6;
16. else
17.   a = 8;
```

上述结果均为a=8

如果预期为8，则错误代码：

```
1.  // Bad
2.  unsigned short a = 1;
3.  unsigned short b = 2;
4.
5.  if (a - b < 0)  // a - b = -1 (signed int)
6.    a = 6;
7.  else
8.    a = 8;
```

正确代码：

```
1.  // Good
2.  unsigned short a = 1;
3.  unsigned short b = 2;
4.
5.  if ((unsigned int)a - (unsigned int)b < 0)  // a - b = 0xffff (unsigned short)
6.    a = 6;
7.  else
8.    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。

```
1.  // Bad
2.  int Foo(int bar) {
3.    switch (bar & 7) {
4.      case 0:
5.        return Foobar(bar);
6.        break;
7.      case 1:
8.        return Foobar(bar * 2);
9.        break;
10.   }
11. }
```

例如上述代码switch的取值可能从0～7，所以应当有default:

```
1.  // Good
2.  int Foo(int bar) {
3.    switch (bar & 7) {
4.      case 0:
5.        return Foobar(bar);
6.        break;
7.      case 1:
8.        return Foobar(bar * 2);
9.        break;
10.     default:
11.       return -1;
12.   }
13. }
```

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
```

```
2.  char* Foo(char* sz, int len) {
3.      char* a = new char[300];
4.      if (len > 100) {
5.          memcpy(a, sz, 100);
6.      }
7.      a[len] = '\0';
8.      return a;  // OK
9.  }
```

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。

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

应当确保有关联的nWeekdays和sWeekdays数据统一。

```
1.  // Good
2.  const int nWeekdays[] = {1, 2, 3, 4, 5, 6, 7};
3.  const char* sWeekdays[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
4.  assert(ARRAY_SIZE(nWeekdays) == ARRAY_SIZE(sWeekdays));
5.  for (int x = 0; x < ARRAY_SIZE(sWeekdays); x++) {
6.    if (strcmp(sWeekdays[x], input) == 0) {
7.      return nWeekdays[x];
8.    }
9.  }
```

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) {
```

```
3.    foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
4.    foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
5.  }
```

如上例，通常可能是：

```
1.  // Good
2.  void Foobar(SomeStruct& foobase, SomeStruct& foo1, SomeStruct& foo2) {
3.    foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
4.    foo2.bar = (foo2.bar & 0xffff) | (foobase.base & 0xffff0000);
5.  }
```

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。

```
1.  // Bad
2.  if (foo1.bar == foo1.bar) {
3.    …
4.  }
```

可能是：

```
1.  // Good
2.  if (foo1.bar == foo2.bar) {
3.    …
4.  }
```

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。

```
1.  // Bad
2.  void Foo() {
3.    char* bar = mmap(0, 0x800000, .....);
4.    *(bar + 0x400000) = '\x88'; // Wrong
5.  }
```

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。

```
1.  // Good
2.  void Foo() {
3.    char* bar = mmap(0, 0x800000, .....);
4.    if(bar == MAP_FAILED) {
5.      return;
6.    }
7.
8.    *(bar + 0x400000) = '\x88';
9.  }
```

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。

```
1.  // Bad
2.  void Foo() {
3.    if (bar = 0x99) ...
4.  }
```

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。

```
1.  // Bad
2.  char  g_somechar;
3.  void foo_thread1() {
4.    g_somechar += 3;
5.  }
6.
7.  void foo_thread2() {
8.    g_somechar += 1;
9.  }
```

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：

```
1.  // Good
2.  volatile char g_somechar;
3.  void foo_thread1() {
4.    __sync_fetch_and_add(&g_somechar, 3);
5.  }
6.
7.  void foo_thread2() {
8.    __sync_fetch_and_add(&g_somechar, 1);
9.  }
```

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。

```
1.  // Bad
2.  char *log_message;
3.
4.  void Handler(int signum) {
5.    syslog(LOG_NOTICE, "%s\n", log_m_essage);
6.    free(log_message);
7.    sleep(10);
8.    exit(0);
9.  }
10.
11. int main (int argc, char* argv[]) {
12.   log_message = strdup(argv[1]);
13.   signal(SIGHUP, Handler);
14.   signal(SIGTERM, Handler);
15.   sleep(10);
16. }
```

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。

```
1.  struct stat *st;
2.
3.  lstat("...", st);
4.
5.  printf("foo");
6.
7.  if (st->st_mtimespec == ...) {
8.    printf("Now updating things\n");
9.    UpdateThings();
10. }
```

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。

```
1. // Bad
2. int main() {
3.   int foo = rand();
4.   return 0;
5. }
```

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

```
1. // Good
2.
3. int main() {
4.   srand(time(0));
5.   int foo = rand();
6.   return 0;
7. }
```

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:

```
int a[5];
a[5] = 0;
```

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 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，例如从大段设备取出的值，要以大段进行处理，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：

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

可能错误：

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

可能是：

```
1.  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：

```
1.  int a[3];
2.  ...;
3.
4.  if (a > 0)
5.    ...;
```

该判断永远为真，等价于：

```
1.  int a[3];
2.  ...;
3.
4.  if (&a[0])
5.    ...;
```

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_msession;
  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();
```

```
 9. }
10.
11. // 输出：
12. // Obj is 000002C7BB550830, data = 1
13. // Obj is 000002C7BB557AF0, data = 2
```

关联漏洞：

高风险-内存破坏

- JavaScript页面类
- Node.js后台类

JavaScript页面类

I. 代码实现

1.1 原生DOM API的安全操作

1.1.1【必须】HTML标签操作，限定/过滤传入变量值

- 使用 `innerHTML=` 、 `outerHTML=` 、 `document.write()` 、 `document.writeln()` 时，如变量值外部可控，应对特殊字符（ `&, <, >, ", '` ）做编码转义，或使用安全的DOM API替代，包括： `innerText=`

```
// 假设 params 为用户输入， text 为 DOM 节点
// bad：将不可信内容带入HTML标签操作
const { user } = params;
// ...
text.innerHTML = `Follow @${user}`;

// good: innerHTML操作前，对特殊字符编码转义
function htmlEncode(iStr) {
    let sStr = iStr;
    sStr = sStr.replace(/&/g, "&amp;");
    sStr = sStr.replace(/>/g, "&gt;");
    sStr = sStr.replace(/</g, "&lt;");
    sStr = sStr.replace(/"/g, "&quot;");
    sStr = sStr.replace(/'/g, "&#39;");
    return sStr;
}

const { user } = params;
user = htmlEncode(user);
// ...
text.innerHTML = `Follow @${user}`;

// good: 使用安全的DOM API替代innerHTML
const { user } = params;
// ...
text.innerText = `Follow @${user}`;
```

1.1.2【必须】HTML属性操作，限定/过滤传入变量值

- 使用 `element.setAttribute(name, value);` 时，如第一个参数值 `name` 外部可控，应用白名单限定允许操作的属性范围。

- 使用 `element.setAttribute(name, value);` 时，操作 `a.href` 、 `ifame.src` 、 `form.action` 、 `embed.src` 、 `object.data` 、 `link.href` 、 `area.href` 、 `input.formaction` 、 `button.formaction` 属性时，如第二个参数值 `value` 外部可控，应参考*JavaScript*页面类规范*1.3.1*部分，限定页面重定向或引入资源的目标地址。

```
// good: setAttribute操作前，限定引入资源的目标地址
function addExternalCss(e) {
    const t = document.createElement('link');
    t.setAttribute('href', e),
    t.setAttribute('rel', 'stylesheet'),
    t.setAttribute('type', 'text/css'),
    document.head.appendChild(t)
}

function validURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) || (/^[/\\][^/\\]/i).test(sUrl));
}

let sUrl = "https://evil.com/1.css"
if (validURL(sUrl)) {
    addExternalCss(sUrl);
}
```

1.2 流行框架/库的安全操作

1.2.1【必须】限定/过滤传入jQuery不安全函数的变量值

- 使用 `.html()` 、 `.append()` 、 `.prepend()` 、 `.wrap()` 、 `.replaceWith()` 、 `.wrapAll()` 、 `.wrapInner()` 、 `.after()` 、 `.before()` 时，如变量值外部可控，应对特殊字符（ `&, <, >, ", '` ）做编码转义。
- 引入 `jQuery 1.x（等于或低于1.12）、jQuery2.x（等于或低于2.2）` ，且使用 `$()` 时，应优先考虑替换为最新版本。如一定需要使用，应对传入参数值中的特殊字符（ `&, <, >, ", '` ）做编码转义。

```
// bad：将不可信内容，带入jQuery不安全函数.after()操作
const { user } = params;
// ...
$("p").after(user);
```

```
5.
6.  // good: jQuery不安全函数.html()操作前，对特殊字符编码转义
7.  function htmlEncode(iStr) {
8.      let sStr = iStr;
9.      sStr = sStr.replace(/&/g, "&amp;");
10.     sStr = sStr.replace(/>/g, "&gt;");
11.     sStr = sStr.replace(/</g, "&lt;");
12.     sStr = sStr.replace(/"/g, "&quot;");
13.     sStr = sStr.replace(/'/g, "&#39;");
14.     return sStr;
15. }
16.
17. // const user = params.user;
18. user = htmlEncode(user);
19. // ...
20. $("p").html(user);
```

- 使用 `.attr()` 操作 `a.href` 、 `ifame.src` 、 `form.action` 、 `embed.src` 、 `object.data` 、 `link.href` 、 `area.href` 、 `input.formaction` 、 `button.formaction` 属性时，应参考*JavaScript*页面类规范*1.3.1*部分，限定重定向的资源目标地址。
- 使用 `.attr(attributeName, value)` 时，如第一个参数值 `attributeName` 外部可控，应用白名单限定允许操作的属性范围。
- 使用 `$.getScript(url [, success ])` 时，如第一个参数值 `url` 外部可控（如：从URL取值拼接，请求jsonp接口），应限定可控变量值的字符集范围为： `[a-zA-Z0-9_-]+` 。

1.2.2【必须】限定/过滤传入Vue.js不安全函数的变量值

- 使用 `v-html` 时，不允许对用户提供的内容使用HTML插值。如业务需要，应先对不可信内容做富文本过滤。

```
1.  // bad：直接渲染外部传入的不可信内容
2.  <div v-html="userProvidedHtml"></div>
3.
4.  // good：使用富文本过滤库处理不可信内容后渲染
5.  <!-- 使用 -->
6.  <div v-xss-html="{'mode': 'whitelist', dirty: html, options: options}" ></div>
7.
8.  <!-- 配置 -->
9.  <script>
10.     new Vue({
```

```
    el: "#app",
    data: {
        options: {
            whiteList: {
                a: ["href", "title", "target", "class", "id"],
                div: ["class", "id"],
                span: ["class", "id"],
                img: ["src", "alt"],
            },
        },
    },
});
</script>
```

- 使用 `v-bind` 操作 `a.href` 、 `ifame.src` 、 `form.action` 、 `embed.src` 、 `object.data` 、 `link.href` 、 `area.href` 、 `input.formaction` 、 `button.formaction` 时，应确保后端已参考*JavaScript*页面类规范*1.3.1*部分，限定了供前端调用的重定向目标地址。
- 使用 `v-bind` 操作 `style` 属性时，应只允许外部控制特定、可控的CSS属性值

```
// bad：v-bind允许外部可控值，自定义CSS属性及数值
<a v-bind:href="sanitizedUrl" v-bind:style="userProvidedStyles">
click me
</a>

// good：v-bind只允许外部提供特性、可控的CSS属性值
<a v-bind:href="sanitizedUrl" v-bind:style="{
color: userProvidedColor,
background: userProvidedBackground
}" >
click me
</a>
```

1.3 页面重定向

1.3.1【必须】限定跳转目标地址

- 使用白名单，限定重定向地址的协议前缀（默认只允许HTTP、HTTPS）、域名（默认只允许公司根域），或指定为固定值；
- 适用场景包括，使用函数方

法：`location.href`、`window.open()`、`location.assign()`、`location.replace()`；赋值或更新HTML属性：`a.href`、`ifame.src`、`form.action`、`embed.src`、`object.data`、`link.href`、`area.href`、`input.formaction`、`button.formaction`；

```
// bad: 跳转至外部可控的不可信地址
const sTargetUrl = getURLParam("target");
location.replace(sTargetUrl);

// good: 白名单限定重定向地址
function validURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) || (/^[/\\][^/\\]/i).test(sUrl));
}

const sTargetUrl = getURLParam("target");
if (validURL(sTargetUrl)) {
    location.replace(sTargetUrl);
}

// good: 制定重定向地址为固定值
const sTargetUrl = "http://www.qq.com";
location.replace(sTargetUrl);
```

1.4 JSON解析/动态执行

1.4.1【必须】使用安全的JSON解析方式

- 应使用`JSON.parse()`解析JSON字符串。低版本浏览器，应使用安全的Polyfill封装

```
// bad: 直接调用eval解析json
const sUserInput = getURLParam("json_val");
const jsonstr1 = `{"name":"a","company":"b","value":"${sUserInput}"}`;
const json1 = eval(`(${jsonstr1})`);

// good: 使用JSON.parse解析
const sUserInput = getURLParam("json_val");
JSON.parse(sUserInput, (k, v) => {
    if (k === "") return v;
    return v * 2;
});
```

```
12.
13. // good: 低版本浏览器，使用安全的Polyfill封装（基于eval）
    <script src="https://github.com/douglascrockford/JSON-js/blob/master/json2.js">
14. </script>;
15. const sUserInput = getURLParam("json_val");
16. JSON.parse(sUserInput);
```

1.5 跨域通讯

1.5.1【必须】使用安全的前端跨域通信方式

- 具有隔离登录态（如：p_skey）、涉及用户高敏感信息的业务（如：微信网页版、QQ空间、QQ邮箱、公众平台），禁止通过 `document.domain` 降域，实现前端跨域通讯，应使用postMessage替代。

1.5.2【必须】使用postMessage应限定Origin

- 在message事件监听回调中，应先使用 `event.origin` 校验来源，再执行具体操作。
- 校验来源时，应使用 `===` 判断，禁止使用 `indexOf()`

```
1.  // bad: 使用indexOf校验Origin值
2.  window.addEventListener("message", (e) => {
3.      if (~e.origin.indexOf("https://a.qq.com")) {
4.      // ...
5.      } else {
6.      // ...
7.      }
8.  });
9.
10. // good: 使用postMessage时，限定Origin，且使用===判断
11. window.addEventListener("message", (e) => {
12.     if (e.origin === "https://a.qq.com") {
13.     // ...
14.     }
15. });
```

II. 配置&环境

2.1 敏感/配置信息

2.1.1【必须】禁止明文硬编码AK/SK

- 禁止前端页面的JS明文硬编码AK/SK类密钥，应封装成后台接口，AK/SK保存在后端配置中心或密钥管理系统

2.2 第三方组件/资源

2.2.1【必须】使用可信范围内的统计组件

2.2.2 【必须】禁止引入非可信来源的第三方JS

2.3 纵深安全防护

2.3.1【推荐】部署CSP，并启用严格模式

Node.js后台类

I. 代码实现

1.1 输入验证

1.1.1【必须】按类型进行数据校验

- 所有程序外部输入的参数值，应进行数据校验。校验内容包括但不限于：数据长度、数据范围、数据类型与格式。校验不通过，应拒绝。

```
// bad：未进行输入验证
Router.get("/vulxss", (req, res) => {
    const { txt } = req.query;
    res.set("Content-Type", "text/html");
    res.send({
        data: txt,
    });
});

// good：按数据类型，进行输入验证
const Router = require("express").Router();
const validator = require("validator");

Router.get("/email_with_validator", (req, res) => {
    const txt = req.query.txt || "";
    if (validator.isEmail(txt)) {
        res.send({
            data: txt,
        });
    } else {
        res.send({ err: 1 });
    }
});
```

关联漏洞：纵深防护措施 - 安全性增强特性

1.2 执行命令

1.2.1 【必须】使用child_process执行系统命令，应限定或校验命令和参数的内容

- 适用场景包括：`child_process.exec`，`child_process.execSync`，`child_process.spawn`，`child_process.spawnSync`，`child_process.execFile`，`child_process.execFileSync`

- 调用上述函数，应首先考虑限定范围，供用户选择。
- 使用 `child_process.exec` 或 `child_process.execSync` 时，如果可枚举输入的参数内容或者格式，则应限定白名单。如果无法枚举命令或参数，则必须过滤或者转义指定符号，包括： `|;&$()><`!`
- 使用 `child_process.spawn` 或 `child_process.execFile` 时，应校验传入的命令和参数在可控列表内。

```
const Router = require("express").Router();
const validator = require("validator");
const { exec } = require('child_process');

// bad：未限定或过滤，直接执行命令
Router.get("/vul_cmd_inject", (req, res) => {
    const txt = req.query.txt || "echo 1";
    exec(txt, (err, stdout, stderr) => {
        if (err) { res.send({ err: 1 }) }
        res.send({stdout, stderr});
    });
});

// good：通过白名单，限定外部可执行命令范围
Router.get("/not_vul_cmd_inject", (req, res) => {
    const txt = req.query.txt || "echo 1";
  const phone = req.query.phone || "";
    const cmdList = {
        sendmsg: "./sendmsg "
    };
    if (txt in cmdList && validator.isMobilePhone(phone)) {
        exec(cmdList[txt] + phone, (err, stdout, stderr) => {
          if (err) { res.send({ err: 1 }) };
          res.send({stdout, stderr});
        });
    } else {
        res.send({
            err: 1,
            tips: `you can use '${Object.keys(cmdList)}'`,
        });
    }
});

// good：执行命令前，过滤/转义指定符号
```

```
35. Router.get("/not_vul_cmd_inject", (req, res) => {
36.     const txt = req.query.txt || "echo 1";
37.   let phone = req.query.phone || "";
38.     const cmdList = {
39.         sendmsg: "./sendmsg "
40.     };
41.     phone = phone.replace(/(\||;|&|\$\(|\(|\)|>|<|\`|!)/gi,"");
42.     if (txt in cmdList) {
43.         exec(cmdList[txt] + phone, (err, stdout, stderr) => {
44.           if (err) { res.send({ err: 1 }) };
45.           res.send({stdout, stderr});
46.         });
47.     } else {
48.         res.send({
49.             err: 1,
50.             tips: `you can use '${Object.keys(cmdList)}'`,
51.         });
52.     }
53. });
```

关联漏洞：高风险 - 任意命令执行

1.3 文件操作

1.3.1 【必须】限定文件操作的后缀范围

- 按业务需求，使用白名单限定后缀范围。

1.3.2 【必须】校验并限定文件路径范围

- 应固定上传、访问文件的路径。若需要拼接外部可控变量值，检查是否包含 `..` 、 `.` 路径穿越字符。如存在，应拒绝。
- 使用 `fs` 模块下的函数方法时，应对第一个参数即路径部分做校验，检查是否包含路径穿越字符 `.` 或 `..` 。涉及方法包括但不限于： `fs.truncate` 、 `fs.truncateSync` 、 `fs.chown` 、 `fs.chownSync` 、 `fs.lchown` 、 `fs.lchownSync` 、 `fs.stat` 、 `fs.lchmodSync` 、 `fs.lstat` 、 `fs.statSync` 、 `fs.lstatSync` 、 `fs.readlink` 、 `fs.unlink` 、 `fs.unlinkSync` 、 `fs.rmdir` 、 `fs.rmdirSync` 、 `fs.mkdir` 、 `fs.mkdirSync` 、 `fs.readdir` 、 `fs.readdirSync` 、 `fs.openSync` 、 `fs.open` 、 `fs.createReadStream` 、 `fs.createWriteStream`
- 使用express框架的 `sendFile` 方法时，应对第一个参数即路径部分做校验，检查是否包含路径穿越字符 `.` 或 `..`
- 校验时，应使用 `path` 模块处理前的路径参数值，或判断处理过后的路径是否穿越出了当前工作

目录。涉及方法包括但不限于：`path.resolve`、`path.join`、`path.normalize`等

```
const fs = require("fs");
const path = require("path");
let filename = req.query.ufile;
let root = '/data/ufile';

// bad：未检查文件名/路径
fs.readFile(root + filename, (err, data) => {
    if (err) {
        return console.error(err);
    }
    console.log(`异步读取: ${data.toString()}`);
});

// bad：使用path处理过后的路径参数值做校验，仍可能有路径穿越风险
filename = path.join(root, filename);
if (filename.indexOf("..") < 0) {
    fs.readFile(filename, (err, data) => {
        if (err) {
            return console.error(err);
        }
        console.log(data.toString());
    });
};

// good：检查了文件名/路径，是否包含路径穿越字符
if (filename.indexOf("..") < 0) {
    filename = path.join(root, filename);
    fs.readFile(filename, (err, data) => {
        if (err) {
            return console.error(err);
        }
        console.log(data.toString());
    });
};
```

1.3.3 【必须】安全地处理上传文件名

- 将上传文件重命名为16位以上的随机字符串保存。
- 如需原样保留文件名，应检查是否包含`..`、`.`路径穿越字符。如存在，应拒绝。

1.3.4 【必须】敏感资源文件，应有加密、鉴权和水印等加固措施

- 用户上传的 `身份证` 、 `银行卡` 等图片，属敏感资源文件，应采取安全加固。
- 指向此类文件的URL，应保证不可预测性；同时，确保无接口会批量展示此类资源的URL。
- 访问敏感资源文件时，应进行权限控制。默认情况下，仅用户可查看、操作自身敏感资源文件。
- 图片类文件应添加业务水印，表明该图片仅可用于当前业务使用。

1.4 网络请求

1.4.1 【必须】限定访问网络资源地址范围

- 应固定程序访问网络资源地址的 `协议` 、 `域名` 、 `路径` 范围。
- 若业务需要，外部可指定访问网络资源地址，应禁止访问内网私有地址段及域名。

```
// 以RFC定义的专有网络为例，如有自定义私有网段亦应加入禁止访问列表。
10.0.0.0/8
172.16.0.0/12
192.168.0.0/16
127.0.0.0/8
```

1.4.2 【推荐】请求网络资源，应加密传输

- 应优先选用https协议请求网络资源

关联漏洞：高风险 - *SSRF*，高风险 - *HTTP*劫持

1.5 数据输出

1.5.1 【必须】高敏感信息禁止存储、展示

- 口令、密保答案、生理标识等鉴权信息禁止展示
- 非金融类业务，信用卡cvv码及日志禁止存储

1.5.2【必须】一般敏感信息脱敏展示

- 身份证只显示第一位和最后一位字符，如：`3**********1`
- 移动电话号码隐藏中间6位字符，如：`134**********48`
- 工作地址/家庭地址最多显示到 `区` 一级
- 银行卡号仅显示最后4位字符，如：`**********8639`

1.5.3 【推荐】返回的字段按业务需要输出

- 按需输出，避免不必要的用户信息泄露
- 用户敏感数据应在服务器后台处理后输出，不可以先输出到客户端，再通过客户端代码来处理展示

关联漏洞：高风险 - 用户敏感信息泄露

1.6 响应输出

1.6.1 【必须】设置正确的HTTP响应包类型

- 响应头Content-Type与实际响应内容，应保持一致。如：API响应数据类型是json，则响应头使用 `application/json` ；若为xml，则设置为 `text/xml` 。

1.6.2 【必须】添加安全响应头

- 所有接口、页面，添加响应头 `X-Content-Type-Options: nosniff` 。
- 所有接口、页面，添加响应头 `X-Frame-Options` 。按需合理设置其允许范围，包括： `DENY` 、 `SAMEORIGIN` 、 `ALLOW-FROM origin` 。用法参考：MDN文档
- 推荐使用组件： `helmet`

1.6.3 【必须】外部输入拼接到响应页面前，进行编码处理

场景	编码规则
输出点在HTML标签之间	需要对以下6个特殊字符进行HTML实体编码(&, <, >, “, ‘,/)。 示例: & –> & < –> < >–> > “ –> " ‘ –> ' / –> /
输出点在HTML标签普通属性内（如href、src、style等，on事件除外）	要对数据进行HTML属性编码。 编码规则：除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为&#xHH;(以&#x开头，HH则是指该字符对应的十六进制数字，分号作为结束符)
输出点在JS内的数据中	需要进行js编码 编码规则： 除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为 \xHH （以 \x 开头，HH则是指该字符对应的十六进制数字） Tips：这种场景仅限于外部数据拼接在js里被引号括起来的变量值中。除此之外禁止直接将代码拼接在js代码中。
输出点在CSS中（Style属性）	需要进行CSS编码 编码规则： 除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为 \HH （以 \ 开头，HH则是指该字符对应的十六进制数字）
输出点在URL属性中	对这些数据进行URL编码 Tips：除此之外，所有链接类属性应该校验其协议。禁止JavaScript、data和Vb伪协议。

1.6.4 【必须】响应禁止展示物理资源、程序内部代码逻辑等敏感信息

- 业务生产（正式）环境，应用异常时，响应内容禁止展示敏感信息。包括但不限于： `物理路径` 、 `程序内部源代码` 、 `调试日志` 、 `内部账号名` 、 `内网ip地址` 等。

```
// bad
Access denied for user 'xxx'@'xx.xxx.xxx.162' (using password: NO)"
```

1.6.5 【推荐】添加安全纵深防御措施

- 部署CSP，规则中应引入最新的严格模式特性 `nonce-`

```
// good：使用helmet组件安全地配置响应头
const express = require("express");
const helmet = require("helmet");
const app = express();
app.use(helmet());

// good：正确配置Content-Type、添加了安全响应头，引入了CSP
Router.get("/", (req, res) => {
    res.header("Content-Type", "application/json");
    res.header("X-Content-Type-Options", "nosniff");
    res.header("X-Frame-Options", "SAMEORIGIN");
    res.header("Content-Security-Policy", "script-src 'self'");
});
```

关联漏洞：中风险 - *XSS*、中风险 - 跳转漏洞

1.7 执行代码

1.7.1 【必须】安全的代码执行方式

- 禁止使用 `eval` 函数
- 禁止使用 `new Function("input")()` 来创建函数
- 使用 `setInteval` ， `setTimeout` ，应校验传入的参数

关联漏洞：高风险 - 代码执行漏洞

1.8 Web跨域

1.8.1 【必须】限定JSONP接口的callback字符集范围

- JSONP接口的callback函数名为固定白名单。如callback函数名可用户自定义，应限制函数名仅包含 字母、数字和下划线。如： `[a-zA-Z0-9_-]+`

1.8.2 【必须】安全的CORS配置

- 使用CORS，应对请求头Origin值做严格过滤、校验。具体来说，可以使用“全等于”判断，或使用严格的正则进行判断。如： `^https://domain\.qq\.com$`

```
// good：使用全等于，校验请求的Origin
if (req.headers.origin === 'https://domain.qq.com') {
    res.setHeader('Access-Control-Allow-Origin', req.headers.origin);
    res.setHeader('Access-Control-Allow-Credentials', true);
}
```

关联漏洞：中风险 - *XSS*，中风险 - *CSRF*，中风险 - *CORS*配置不当

1.9 SQL操作

1.9.1 【必须】SQL语句默认使用预编译并绑定变量

- 应使用预编译绑定变量的形式编写sql语句，保持查询语句和数据相分离

```
// bad：拼接SQL语句查询，存在安全风险
const mysql  = require("mysql");
const connection = mysql.createConnection(options);
connection.connect();

const sql = util.format("SELECT * from some_table WHERE Id = %s and Name = %s", req.body.id, req.body.name);
connection.query(sql, (err, result) => {
    // handle err..
});

// good：使用预编译绑定变量构造SQL语句
const mysql  = require("mysql");
const connection = mysql.createConnection(options);
connection.connect();

const sql = "SELECT * from some_table WHERE Id = ? and Name = ?";
const sqlParams = [req.body.id, req.body.name];
connection.query(sql, sqlParams, (err, result) => {
    // handle err..
});
```

- 对于表名、列名等无法进行预编译的场景，如：`__user_input__` 拼接到比如 `limit`，`order by`，`group by`，`from tablename` 语句中。请使用以下方法：

  方案*1*：使用白名单校验表名/列名

```
// good
```

```
const tableSuffix = req.body.type;
if (["expected1", "expected2"].indexOf(tableSuffix) < 0) {
    // 不在表名白名单中，拒绝请求
    return ;
}
const sql = `SELECT * from t_business_${tableSuffix}`;
connection.query(sql, (err, result) => {
    // handle err..
});
```

方案2：使用反引号包裹表名/列名，并过滤 `__user_input__` 中的反引号

```
// good
let { orderType } = req.body;
// 过滤掉__user_input__中的反引号
orderType = orderType.replace("`", "");
const sql = util.format("SELECT * from t_business_feeds order by `%s`",
orderType);
connection.query(sql, (err, result) => {
    // handle err..
});
```

方案3：将 `__user_input__` 转换为整数

```
// good
let { orderType } = req.body;
// 强制转换为整数
orderType = parseInt(orderType, 10);
const sql = `SELECT * from t_business_feeds order by ${orderType}`;
connection.query(sql, (err, result) => {
    // handle err..
});
```

1.9.2 【必须】安全的ORM操作

- 使用安全的ORM组件进行数据库操作。如 `sequelize` 等
- 禁止 `__user_input__` 以拼接的方式直接传入ORM的各类raw方法

```
//bad: adonisjs ORM
//参考：https://adonisjs.com/docs/3.2/security-introduction#_sql_injection
const username = request.param("username");
```

```
const users = yield Database
  .table("users")
  .where(Database.raw(`username = ${username}`));

//good: adonisjs ORM
const username = request.param("username");
const users = yield Database
  .table('users')
  .where(Database.raw("username = ?", [username]));
```

- 使用ORM进行Update/Insert操作时，应限制操作字段范围

```
/*
good
假设该api用于插入用户的基本信息，使用传入的req.body通过Sequelize的create方法实现
假设User包含字段：username,email,isAdmin,
其中,isAdmin将会用于是否系统管理员的鉴权，默认值为false
*/
// Sequelize: 只允许变更username、email字段值
User.create(req.body, { fields: ["username", "email"] }).then((user) => {
    // handle the rest..
});
```

> 为什么要这么做？ 在上述案例中，若不限定fields值，攻击者将可传入 `{"username":"boo","email":"foo@boo.com","isAdmin":true}` 将自己变为 `Admin` ，产生垂直越权漏洞。

关联漏洞：高风险 - *SQL*注入，中风险 - *Mass Assignment* 逻辑漏洞

1.10 NoSQL操作

1.10.1 【必须】校验参数值类型

- 将HTTP参数值代入NoSQL操作前，应校验类型。如非功能需要，禁止对象（Object）类型传入。

```
// bad：执行NOSQL操作前，未作任何判断
app.post("/", (req, res) => {
    db.users.find({ username: req.body.username, password: req.body.password },
(err, users) => {
    // **TODO:** handle the rest
    });
});
```

```
7.
8.  // good：在进入nosql前先判断`__USER_INPUT__`是否为字符串。
9.  app.post("/", (req, res) => {
10.     if (req.body.username && typeof req.body.username !== "string") {
11.         return new Error("username must be a string");
12.     }
13.     if (req.body.password && typeof req.body.password !== "string") {
14.         return new Error("password must be a string");
15.     }
        db.users.find({ username: req.body.username, password: req.body.password },
16. (err, users) => {
17.         // **TODO:** handle the rest
18.     });
19. });
```

> 为什么要这么做？
>
> JavaScript中，从http或socket接收的数据可能不是单纯的字符串，而是被黑客精心构造的对象(Object)。在本例中：
>
> - 期望接收的POST数据：`username=foo&password=bar`
> - 期望的等价条件查询sql语句：`select * from users where username = 'foo' and password = 'bar'`
> - 黑客的精心构造的攻击POST数据：`username[$ne]=null&password[$ne]=null` 或JSON格式：`{"username": {"$ne": null},"password": {"$ne": null}}`
> - 黑客篡改后的等价条件查询sql语句：`select * from users where username != null & password != null`
> - 黑客攻击结果：绕过正常逻辑，在不知道他人的username/password的情况登录他人账号。

1.10.2 【必须】NoSQL操作前，应校验权限/角色

- 执行NoSQL增、删、改、查逻辑前，应校验权限

```
1.  // 使用express、mongodb(mongoose)实现的删除文章demo
2.  // bad：在删除文章前未做权限校验
3.  app.post("/deleteArticle", (req, res) => {
        db.articles.deleteOne({ article_id: req.body.article_id }, (err, users) =>
4.  {
5.      // TODO: handle the rest
6.  });
7.  });
8.
9.  // good：进入nosql语句前先进行权限校验
10. app.post("/deleteArticle", (req, res) => {
```

```
11.     checkPriviledge(ctx.uin, req.body.article_id);
        db.articles.deleteOne({ article_id: req.body.article_id }, (err, users) =>
12. {
13.         // TODO: handle the rest
14.     });
15. });
```

关联漏洞：高风险 - 越权操作，高风险 - *NoSQL*注入

1.11 服务器端渲染（SSR）

1.11.1 【必须】安全的Vue服务器端渲染(Vue SSR)

- 禁止直接将不受信的外部内容传入 `{{{ data }}}` 表达式中
- 模板内容禁止被污染

```
1.  // bad: 将用户输入替换进模板
2.  const app = new Vue({
3.      template: appTemplate.replace("word", __USER_INPUT__),
4.  });
5.  renderer.renderToString(app);
```

- 对已渲染的HTML文本内容（renderToString后的html内容）。如需再拼不受信的外部输入，应先进行安全过滤，具体请参考**1.6.3**

```
1.  // bad: 渲染后的html再拼接不受信的外部输入
2.  return new Promise(((resolve) => {
3.      renderer.renderToString(component, (err, html) => {
4.          let htmlOutput = html;
5.          htmlOutput += `${__USER_INPUT__}`;
6.          resolve(htmlOutput);
7.      });
8.  }));
```

1.11.2 【必须】安全地使用EJS、LoDash、UnderScore进行服务器端渲染

- 使用render函数时，模板内容禁止被污染

  lodash.Template:

  ```
  1.  // bad: 将用户输入送进模板
  2.  const compiled = _.template(`<b>${__USER_INPUT__}<%- value %></b>`);
  ```

```
3.  compiled({ value: "hello" });
```

ejs:

```
1.  // bad: 将用户输入送进模板
2.  const ejs = require("ejs");
3.  const people = ["geddy", "neil", "alex"];
    const html = ejs.render(`<%= people.join(", "); %>${__USER_INPUT__}`, {
4.  people });
```

- Ejs、LoDash、UnderScore提供的HTML插值模板默认形似 `<%= data %>` ，尽管在默认情况下 `<%= data %>` 存在过滤，在编写HTML插值模板时需注意：

  i. 用户输入流入html属性值时，必须使用双引号包裹： `<base data-id = "<%= __USER_INPUT__ %>">`

  ii. 用户输入流入 `<script></script>` 标签或on*的html属性中时，如 `<script>var id = <%= __USER_INPUT__ %></script>` ，须按照1.6.3中的做法或白名单方法进行过滤，框架/组件的过滤在此处不起作用

1.11.3 【必须】在自行实现状态存储容器并将其JSON.Stringify序列化后注入到HTML时，必须进行安全过滤

1.12 URL跳转

1.12.1【必须】限定跳转目标地址

- 适用场景包括:

1. 使用30x返回码并在Header中设置Location进行跳转
2. 在返回页面中打印 `<script>location.href=__Redirection_URL__</script>`

- 使用白名单，限定重定向地址的协议前缀（默认只允许HTTP、HTTPS）、域名（默认只允许公司根域），或指定为固定值;

```
1.  // 使用express实现的登录成功后的回调跳转页面
2.
3.  // bad: 未校验页面重定向地址
4.  app.get("/login", (req, res) => {
5.      // 若未登录用户访问其他页面，则让用户导向到该处理函数进行登录
      // 使用参数loginCallbackUrl记录先前尝试访问的url，在登录成功后跳转回
6.  loginCallbackUrl:
7.      const { loginCallbackUrl } = req.query;
8.      if (loginCallbackUrl) {
```

```
        res.redirect(loginCallbackUrl);
    }
});

// good: 白名单限定重定向地址
function isValidURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) || (/^[/\\][^/\\]/i).test(sUrl));
}
app.get("/login", (req, res) => {
    // 若未登录用户访问其他页面，则让用户导向到该处理函数进行登录
    // 使用参数loginCallbackUrl记录先前尝试访问的url，在登录成功后跳转回
loginCallbackUrl:
    const { loginCallbackUrl } = req.query;
    if (loginCallbackUrl && isValidUrl(loginCallbackUrl)) {
        res.redirect(loginCallbackUrl);
    }
});

// good: 白名单限定重定向地址，通过返回html实现
function isValidURL(sUrl) {
    return !!((/^(https?:\/\/)?[\w\-.]+\.(qq|tencent)\.com($|\/|\\)/i).test(sUrl) || (/^[\w][\w/.\-_%]+$/i).test(sUrl) || (/^[/\\][^/\\]/i).test(sUrl));
}
app.get("/login", (req, res) => {
    // 若未登录用户访问其他页面，则让用户导向到该处理函数进行登录
    // 使用参数loginCallbackUrl记录先前尝试访问的url，在登录成功后跳转回
loginCallbackUrl:
    const { loginCallbackUrl } = req.query;
    if (loginCallbackUrl && isValidUrl(loginCallbackUrl)) {
        // 使用encodeURI，过滤左右尖括号与双引号，防止逃逸出包裹的双引号
        const redirectHtml = `<script>location.href = "${encodeURI(loginCallbackUrl)}";</script>`;
        res.end(redirectHtml);
    }
});
```

关联漏洞：中风险 - 任意*URL*跳转漏洞

1.13 Cookie与登录态

1.13.1【推荐】为Cookies中存储的关键登录态信息添加http-only保护

关联漏洞：纵深防护措施 - 安全性增强特性

II. 配置&环境

2.1 依赖库

2.1.1【必须】使用安全的依赖库

- 使用自动工具，检查依赖库是否存在后门/漏洞，保持最新版本

2.2 运行环境

2.2.1 【必须】使用非root用户运行Node.js

2.3 配置信息

2.3.1【必须】禁止硬编码认证凭证

- 禁止在源码中硬编码 `AK/SK` 、 `数据库账密` 、 `私钥证书` 等配置信息
- 应使用配置系统或KMS密钥管理系统。

2.3.2【必须】禁止硬编码IP配置

- 禁止在源码中硬编码 `IP` 信息

> 为什么要这么做？
>
> 硬编码IP可能会导致后续机器裁撤或变更时产生额外的工作量，影响系统的可靠性。

2.3.3【必须】禁止硬编码员工敏感信息

- 禁止在源代码中含员工敏感信息，包括但不限于： `员工ID` 、 `手机号` 、 `微信/QQ号` 等。

- 通用类
- 后台类

通用类

1. 代码实现类

1.1 内存管理

1.1.1【必须】切片长度校验

- 在对slice进行操作时，必须判断长度是否合法，防止程序panic

```
// bad: 未判断data的长度，可导致 index out of range
func decode(data [] byte) bool {
    if data[0] == 'F' && data[1] == 'U' && data[2] == 'Z' && data[3] == 'Z' &&
data[4] == 'E' && data[5] == 'R' {
        fmt.Println("Bad")
        return true
    }

    return false
}

// bad: slice bounds out of range
func foo() {
    var slice = []int{0, 1, 2, 3, 4, 5, 6}
    fmt.Println(slice[:10])
}

// good: 使用data前应判断长度是否合法
func decode(data [] byte) bool {
    if len(data) == 6 {
        if data[0] == 'F' && data[1] == 'U' && data[2] == 'Z' && data[3] == 'Z'
&& data[4] == 'E' && data[5] == 'R' {
            fmt.Println("Good")
            return true
        }
    }

    return false
}
```

1.1.2【必须】nil指针判断

- 进行指针操作时，必须判断该指针是否为nil，防止程序panic，尤其在进行结构体Unmarshal时

```
type Packet struct {
    PackeyType        uint8
    PackeyVersion     uint8
    Data              *Data
}

type Data struct {
    Stat    uint8
    Len     uint8
    Buf     [8]byte
}

func (p *Packet) UnmarshalBinary(b []byte) error {
    if len(b) < 2 {
       return io.EOF
    }

    p.PackeyType = b[0]
    p.PackeyVersion = b[1]

    // 若长度等于2, 那么不会new Data
    if len(b) > 2 {
        p.Data = new(Data)
        // Unmarshal(b[i:], p.Data)
    }

    return nil
}

// bad: 未判断指针是否为nil
func main() {
    packet := new(Packet)
    data := make([]byte, 2)
    if err := packet.UnmarshalBinary(data); err != nil {
        fmt.Println("Failed to unmarshal packet")
        return
    }

    fmt.Printf("Stat: %v\n", packet.Data.Stat)
}

// good: 判断Data指针是否未nil
```

```
func main() {

    packet := new(Packet)
    data := make([]byte, 2)

    if err := packet.UnmarshalBinary(data); err != nil {
        fmt.Println("Failed to unmarshal packet")
        return
    }

    if packet.Data == nil {
        return
    }

    fmt.Printf("Stat: %v\n", packet.Data.Stat)
}
```

1.1.3【必须】整数安全

- 在进行数字运算操作时，需要做好长度限制，防止外部输入运算导致异常：
    - 确保无符号整数运算时不会反转
    - 确保有符号整数运算时不会出现溢出
    - 确保整型转换时不会出现截断错误
    - 确保整型转换时不会出现符号错误
- 以下场景必须严格进行长度限制：
    - 作为数组索引
    - 作为对象的长度或者大小
    - 作为数组的边界（如作为循环计数器）

```
// bad: 未限制长度，导致整数溢出
func overflow(numControlByUser int32) {
    var numInt int32 = 0
    numInt = numControlByUser + 1
    //对长度限制不当，导致整数溢出
    fmt.Printf("%d\n", numInt)
    //使用numInt，可能导致其他错误
}

func main() {
    overflow(2147483647)
```

```
}

// good:
func overflow(numControlByUser int32) {
    var numInt int32 = 0
    numInt = numControlByUser + 1
    if numInt < 0 {
        fmt.Println("integer overflow")
        return;
    }
    fmt.Println("integer ok")
}

func main() {
    overflow(2147483647)
}
```

1.1.4【必须】make分配长度验证

- 在进行make分配内存时，需要对外部可控的长度进行校验，防止程序panic。

```
// bad
func parse(lenControlByUser int, data[] byte) {
    size := lenControlByUser
    //对外部传入的size，进行长度判断以免导致panic
    buffer := make([]byte, size)
    copy(buffer, data)
}

// good
func parse(lenControlByUser int, data[] byte) ([]byte, error){
    size := lenControlByUser
    //限制外部可控的长度大小范围
    if size > 64*1024*1024 {
        return nil, errors.New("value too large")
    }
    buffer := make([]byte, size)
    copy(buffer, data)
    return buffer, nil
}
```

1.1.5【必须】禁止SetFinalizer和指针循环引用同时使用

- 当一个对象从被GC选中到移除内存之前，runtime.SetFinalizer()都不会执行，即使程序正常结束或者发生错误。由指针构成的“循环引用”虽然能被GC正确处理，但由于无法确定Finalizer依赖顺序，从而无法调用runtime.SetFinalizer()，导致目标对象无法变成可达状态，从而造成内存无法被回收。

```
// bad
func foo() {
    var a, b Data
    a.o = &b
    b.o = &a

    //指针循环引用, SetFinalizer()无法正常调用
    runtime.SetFinalizer(&a, func(d *Data) {
        fmt.Printf("a %p final.\n", d)
    })
    runtime.SetFinalizer(&b, func(d *Data) {
        fmt.Printf("b %p final.\n", d)
    })
}

func main() {
    for {
        foo()
        time.Sleep(time.Millisecond)
    }
}
```

1.1.6【必须】禁止重复释放channel

- 重复释放一般存在于异常流程判断中，如果恶意攻击者构造出异常条件使程序重复释放channel，则会触发运行时恐慌，从而造成DoS攻击。

```
// bad
func foo(c chan int) {
    defer close(c)
    err := processBusiness()
    if err != nil {
        c <- 0
        close(c) // 重复释放channel
        return
    }
    c <- 1
```

```
}

// good
func foo(c chan int) {
    defer close(c) // 使用defer延迟关闭channel
    err := processBusiness()
    if err != nil {
        c <- 0
        return
    }
    c <- 1
}
```

1.1.7【必须】确保每个协程都能退出

- 启动一个协程就会做一个入栈操作，在系统不退出的情况下，协程也没有设置退出条件，则相当于协程失去了控制，它占用的资源无法回收，可能会导致内存泄露。

```
// bad: 协程没有设置退出条件
func doWaiter(name string, second int) {
    for {
        time.Sleep(time.Duration(second) * time.Second)
        fmt.Println(name, " is ready!")
    }
}
```

1.1.8【推荐】不使用unsafe包

- 由于unsafe包绕过了 Golang 的内存安全原则，一般来说使用该库是不安全的，可导致内存破坏，尽量避免使用该包。若必须要使用unsafe操作指针，必须做好安全校验。

```
// bad: 通过unsafe操作原始指针
func unsafePointer() {
    b := make([]byte, 1)
    foo := (*int)(unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) +
uintptr(0xfffffffe)))
    fmt.Print(*foo + 1)
}

// [signal SIGSEGV: segmentation violation code=0x1 addr=0xc100068f55
pc=0x49142b]
```

1.1.9【推荐】不使用slice作为函数入参

- slice是引用类型，在作为函数入参时采用的是地址传递，对slice的修改也会影响原始数据

```
// bad
// slice作为函数入参时是地址传递
func modify(array []int) {
    array[0] = 10 // 对入参slice的元素修改会影响原始数据
}

func main() {
    array := []int{1, 2, 3, 4, 5}

    modify(array)
    fmt.Println(array) // output：[10 2 3 4 5]
}

// good
// 数组作为函数入参时，而不是slice
func modify(array [5]int) {
  array[0] = 10
}

func main() {
    // 传入数组，注意数组与slice的区别
    array := [5]int{1, 2, 3, 4, 5}

    modify(array)
    fmt.Println(array)
}
```

1.2 文件操作

1.2.1【必须】 路径穿越检查

- 在进行文件操作时，如果对外部传入的文件名未做限制，可能导致任意文件读取或者任意文件写入，严重可能导致代码执行。

```
// bad: 任意文件读取
func handler(w http.ResponseWriter, r *http.Request) {
    path := r.URL.Query()["path"][0]

    // 未过滤文件路径，可能导致任意文件读取
    data, _ := ioutil.ReadFile(path)
    w.Write(data)

    // 对外部传入的文件名变量，还需要验证是否存在../等路径穿越的文件名
    data, _ = ioutil.ReadFile(filepath.Join("/home/user/", path))
    w.Write(data)
}

// bad: 任意文件写入
func unzip(f string) {
    r, _ := zip.OpenReader(f)
    for _, f := range r.File {
        p, _ := filepath.Abs(f.Name)
        // 未验证压缩文件名，可能导致../等路径穿越，任意文件路径写入
        ioutil.WriteFile(p, []byte("present"), 0640)
    }
}

// good: 检查压缩的文件名是否包含..路径穿越特征字符，防止任意写入
func unzipGood(f string) bool {
    r, err := zip.OpenReader(f)
    if err != nil {
        fmt.Println("read zip file fail")
        return false
    }
    for _, f := range r.File {
        p, _ := filepath.Abs(f.Name)
        if !strings.Contains(p, "..") {
            ioutil.WriteFile(p, []byte("present"), 0640)
```

```
        }
    }
    return true
}
```

1.2.2【必须】 文件访问权限

- 根据创建文件的敏感性设置不同级别的访问权限，以防止敏感数据被任意权限用户读取。例如，设置文件权限为：`-rw-r-----`

```
ioutil.WriteFile(p, []byte("present"), 0640)
```

1.3 系统接口

1.3.1【必须】命令执行检查

- 使用 `exec.Command`、`exec.CommandContext`、`syscall.StartProcess`、`os.StartProcess` 等函数时，第一个参数（path）直接取外部输入值时，应使用白名单限定可执行的命令范围，不允许传入 `bash`、`cmd`、`sh` 等命令；
- 使用 `exec.Command`、`exec.CommandContext` 等函数时，通过 `bash`、`cmd`、`sh` 等创建shell，-c后的参数（arg）拼接外部输入，应过滤\n $ & ; | ‘ “ ( ) `等潜在恶意字符；

```
// bad
func foo() {
    userInputedVal := "&& echo 'hello'" // 假设外部传入该变量值
    cmdName := "ping " + userInputedVal

    //未判断外部输入是否存在命令注入字符，结合sh可造成命令注入
    cmd := exec.Command("sh", "-c", cmdName)
    output, _ := cmd.CombinedOutput()
    fmt.Println(string(output))

    cmdName := "ls"
    //未判断外部输入是否是预期命令
    cmd := exec.Command(cmdName)
    output, _ := cmd.CombinedOutput()
    fmt.Println(string(output))
}

// good
func checkIllegal(cmdName string) bool {
    if strings.Contains(cmdName, "&") || strings.Contains(cmdName, "|") ||
strings.Contains(cmdName, ";") ||
        strings.Contains(cmdName, "$") || strings.Contains(cmdName, "'") ||
strings.Contains(cmdName, "`") ||
        strings.Contains(cmdName, "(") || strings.Contains(cmdName, ")") ||
strings.Contains(cmdName, "\"") {
        return true
    }
    return false
}
```

```
27.
28. func main() {
29.     userInputedVal := "&& echo 'hello'"
30.     cmdName := "ping " + userInputedVal
31.
32.     if checkIllegal(cmdName) { // 检查传给sh的命令是否有特殊字符
33.         return // 存在特殊字符直接return
34.     }
35.
36.     cmd := exec.Command("sh", "-c", cmdName)
37.     output, _ := cmd.CombinedOutput()
38.     fmt.Println(string(output))
39. }
```

1.4 通信安全

1.4.1【必须】网络通信采用TLS方式

- 明文传输的通信协议目前已被验证存在较大安全风险，被中间人劫持后可能导致许多安全风险，因此必须采用至少TLS的安全通信方式保证通信安全，例如gRPC/Websocket都使用TLS1.3。

```
// good
func main() {
  http.HandleFunc("/", func (w http.ResponseWriter, req *http.Request) {
    w.Header().Add("Strict-Transport-Security", "max-age=63072000;
includeSubDomains")
    w.Write([]byte("This is an example server.\n"))
  })

  //服务器配置证书与私钥
  log.Fatal(http.ListenAndServeTLS(":443", "yourCert.pem", "yourKey.pem", nil))
}
```

1.4.2【推荐】TLS启用证书验证

- TLS证书应当是有效的、未过期的，且配置正确的域名，生产环境的服务端应启用证书验证。

```
// bad
import (
    "crypto/tls"
    "net/http"
)

func doAuthReq(authReq *http.Request) *http.Response {
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    }
    client := &http.Client{Transport: tr}
    res, _ := client.Do(authReq)
    return res
}

// good
import (
    "crypto/tls"
```

```
19.     "net/http"
20. )
21.
22. func doAuthReq(authReq *http.Request) *http.Response {
23.     tr := &http.Transport{
24.         TLSClientConfig: &tls.Config{InsecureSkipVerify: false},
25.     }
26.     client := &http.Client{Transport: tr}
27.     res, _ := client.Do(authReq)
28.     return res
29. }
```

1.5 敏感数据保护

1.5.1【必须】敏感信息访问

- 禁止将敏感信息硬编码在程序中，既可能会将敏感信息暴露给攻击者，也会增加代码管理和维护的难度
- 使用配置中心系统统一托管密钥等敏感信息

1.5.2【必须】敏感数据输出

- 只输出必要的最小数据集，避免多余字段暴露引起敏感信息泄露
- 不能在日志保存密码（包括明文密码和密文密码）、密钥和其它敏感信息
- 对于必须输出的敏感信息，必须进行合理脱敏展示

```
// bad
func serve() {
    http.HandleFunc("/register", func(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()
        user := r.Form.Get("user")
        pw := r.Form.Get("password")

        log.Printf("Registering new user %s with password %s.\n", user, pw)
    })
    http.ListenAndServe(":80", nil)
}

// good
func serve1() {
    http.HandleFunc("/register", func(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()
        user := r.Form.Get("user")
        pw := r.Form.Get("password")

        log.Printf("Registering new user %s.\n", user)

        // ...
        use(pw)
    })
    http.ListenAndServe(":80", nil)
}
```

- 避免通过GET方法、代码注释、自动填充、缓存等方式泄露敏感信息

1.5.3【必须】敏感数据存储

- 敏感数据应使用SHA2、RSA等算法进行加密存储
- 敏感数据应使用独立的存储层，并在访问层开启访问控制
- 包含敏感信息的临时文件或缓存一旦不再需要应立刻删除

1.5.4【必须】异常处理和日志记录

- 应合理使用panic、recover、defer处理系统异常，避免出错信息输出到前端

```
1.  defer func () {
2.          if r := recover(); r != nil {
3.              fmt.Println("Recovered in start()")
4.          }
5.      }()
```

- 对外环境禁止开启debug模式，或将程序运行日志输出到前端

错误例子:

```
1.  dlv --listen=:2345 --headless=true --api-version=2 debug test.go
```

正确例子:

```
1.  dlv debug test.go
```

1.6 加密解密

1.6.1【必须】不得硬编码密码/密钥

- 在进行用户登陆，加解密算法等操作时，不得在代码里硬编码密钥或密码，可通过变换算法或者配置等方式设置密码或者密钥。

```
// bad
const (
    user     = "dbuser"
    password = "s3cretp4ssword"
)

func connect() *sql.DB {
    connStr := fmt.Sprintf("postgres://%s:%s@localhost/pqgotest", user,
password)
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        return nil
    }
    return db
}

// bad
var (
    commonkey = []byte("0123456789abcdef")
)

func AesEncrypt(plaintext string) (string, error) {
    block, err := aes.NewCipher(commonkey)
    if err != nil {
        return "", err
    }
}
```

1.6.2【必须】密钥存储安全

- 在使用对称密码算法时，需要保护好加密密钥。当算法涉及敏感、业务数据时，可通过非对称算法协商加密密钥。其他较为不敏感的数据加密，可以通过变换算法等方式保护密钥。

1.6.3【推荐】不使用弱密码算法

- 在使用加密算法时，不建议使用加密强度较弱的算法。

错误例子：

```
1.  crypto/des, crypto/md5, crypto/sha1, crypto/rc4等。
```

1.7 正则表达式

1.7.1【推荐】使用regexp进行正则表达式匹配

- 正则表达式编写不恰当可被用于DoS攻击，造成服务不可用，推荐使用regexp包进行正则表达式匹配。regexp保证了线性时间性能和优雅的失败：对解析器、编译器和执行引擎都进行了内存限制。但regexp不支持以下正则表达式特性，如业务依赖这些特性，则regexp不适合使用。
    - 回溯引用Backreferences和查看Lookaround

```
// good
matched, err := regexp.MatchString(`a.b`, "aaxbb")
fmt.Println(matched) // true
fmt.Println(err)     // nil (regexp is valid)
```

后台类

1 代码实现类

1.1 输入校验

1.1.1【必须】按类型进行数据校验

- 所有外部输入的参数，应使用 `validator` 进行白名单校验，校验内容包括但不限于数据长度、数据范围、数据类型与格式，校验不通过的应当拒绝

```
// good
import (
    "fmt"
    "github.com/go-playground/validator/v10"
)

var validate *validator.Validate
validate = validator.New()
func validateVariable() {
    myEmail := "abc@tencent.com"
    errs := validate.Var(myEmail, "required,email")
    if errs != nil {
        fmt.Println(errs)
        return
        //停止执行
    }
    // 验证通过，继续执行
    ...
}
```

- 无法通过白名单校验的应使用 `html.EscapeString` 、 `text/template` 或 `bluemonday` 对 `<, >, &, ','` 等字符进行过滤或编码

```
  import(
      "text/template"
  )

  // TestHTMLEscapeString HTML特殊字符转义
  func main(inputValue string) string{
      escapedResult := template.HTMLEscapeString(inputValue)
      return escapedResult
  }
```

1.2 SQL操作

1.2.1【必须】SQL语句默认使用预编译并绑定变量

- 使用 `database/sql` 的prepare、Query或使用GORM等ORM执行SQL操作

```
import (
  "github.com/jinzhu/gorm"
  _ "github.com/jinzhu/gorm/dialects/sqlite"
)

type Product struct {
  gorm.Model
  Code string
  Price uint
}
...
var product Product
db.First(&product, 1)
```

- 使用参数化查询，禁止拼接SQL语句，另外对于传入参数用于order by或表名的需要通过校验

```
// bad
  import (
      "database/sql"
      "fmt"
      "net/http"
  )

  func handler(db *sql.DB, req *http.Request) {
      q := fmt.Sprintf("SELECT ITEM,PRICE FROM PRODUCT WHERE ITEM_CATEGORY='%s'
ORDER BY PRICE",
          req.URL.Query()["category"])
      db.Query(q)
  }

// good
func handlerGood(db *sql.DB, req *http.Request) {
    //使用?占位符
      q := "SELECT ITEM,PRICE FROM PRODUCT WHERE ITEM_CATEGORY='?' ORDER BY
PRICE"
```

```
18.         db.Query(q, req.URL.Query()["category"])
19.  }
```

1.3 网络请求

1.3.1【必须】资源请求过滤验证

- 使用 `"net/http"` 下的方法 `http.Get(url)` 、 `http.Post(url, contentType, body)` 、 `http.Head(url )` 、 `http.PostForm(url, data)` 、 `http.Do(req)` 时，如变量值外部可控（指从参数中动态获取），应对请求目标进行严格的安全校验。
- 如请求资源域名归属固定的范围，如只允许 `a.qq.com` 和 `b.qq.com` ，应做白名单限制。如不适用白名单，则推荐的校验逻辑步骤是：
    - 第 1 步、只允许HTTP或HTTPS协议
    - 第 2 步、解析目标URL，获取其HOST
    - 第 3 步、解析HOST，获取HOST指向的IP地址转换成Long型
    - 第 4 步、检查IP地址是否为内网IP，网段有：

```
// 以RFC定义的专有网络为例，如有自定义私有网段亦应加入禁止访问列表。
10.0.0.0/8
172.16.0.0/12
192.168.0.0/16
127.0.0.0/8
```

    - 第 5 步、请求URL
    - 第 6 步、如有跳转，跳转后执行1，否则绑定经校验的ip和域名，对URL发起请求
- 官方库 `encoding/xml` 不支持外部实体引用，使用该库可避免xxe漏洞

```
import (
    "encoding/xml"
    "fmt"
    "os"
)

func main() {
    type Person struct {
        XMLName   xml.Name `xml:"person"`
        Id        int      `xml:"id,attr"`
        UserName string   `xml:"name>first"`
        Comment string `xml:",comment"`
```

```
13.         }
14.
15.         v := &Person{Id: 13, UserName: "John"}
16.         v.Comment = " Need more details. "
17.
18.         enc := xml.NewEncoder(os.Stdout)
19.         enc.Indent("  ", "    ")
20.         if err := enc.Encode(v); err != nil {
21.             fmt.Printf("error: %v\n", err)
22.         }
23.
24.     }
```

1.4 服务器端渲染

1.4.1【必须】模板渲染过滤验证

- 使用 `text/template` 或者 `html/template` 渲染模板时禁止将外部输入参数引入模板，或仅允许引入白名单内字符。

```
    // bad
     func handler(w http.ResponseWriter, r *http.Request) {
       r.ParseForm()
       x := r.Form.Get("name")

       var tmpl = `<!DOCTYPE html><html><body>
     <form action="/" method="post">
         First name:<br>
     <input type="text" name="name" value="">
     <input type="submit" value="Submit">
     </form><p>` + x + ` </p></body></html>`

       t := template.New("main")
       t, _ = t.Parse(tmpl)
       t.Execute(w, "Hello")
     }

// good
     import (
         "fmt"
         "github.com/go-playground/validator/v10"
     )

     var validate *validator.Validate
     validate = validator.New()
     func validateVariable(val) {
         errs := validate.Var(val, "gte=1,lte=100")//限制必须是1-100的正整数
         if errs != nil {
             fmt.Println(errs)
             return False
         }
         return True
     }

```

```
35.     func handler(w http.ResponseWriter, r *http.Request) {
36.         r.ParseForm()
37.         x := r.Form.Get("name")
38.
39.         if validateVariable(x):
40.             var tmpl = `<!DOCTYPE html><html><body>
41.             <form action="/" method="post">
42.             First name:<br>
43.             <input type="text" name="name" value="">
44.             <input type="submit" value="Submit">
45.             </form><p>` + x + ` </p></body></html>`
46.             t := template.New("main")
47.             t, _ = t.Parse(tmpl)
48.             t.Execute(w, "Hello")
49.         else:
50.             ...
51.     }
```

1.5 Web跨域

1.5.1【必须】跨域资源共享CORS限制请求来源

- CORS请求保护不当可导致敏感信息泄漏，因此应当严格设置Access-Control-Allow-Origin使用同源策略进行保护。

```
1.  // good
2.   c := cors.New(cors.Options{
3.       AllowedOrigins: []string{"http://qq.com", "https://qq.com"},
4.       AllowCredentials: true,
5.       Debug: false,
6.   })
7.
8.   //引入中间件
9.   handler = c.Handler(handler)
```

1.6 响应输出

1.6.1 【必须】设置正确的HTTP响应包类型

- 响应头Content-Type与实际响应内容，应保持一致。如：API响应数据类型是json，则响应头使用 `application/json` ；若为xml，则设置为 `text/xml` 。

1.6.2 【必须】添加安全响应头

- 所有接口、页面，添加响应头 `X-Content-Type-Options: nosniff` 。
- 所有接口、页面，添加响应头 `X-Frame-Options` 。按需合理设置其允许范围，包括： `DENY` 、 `SAMEORIGIN` 、 `ALLOW-FROM origin` 。用法参考：MDN文档

1.6.3【必须】外部输入拼接到HTTP响应头中需进行过滤

- 应尽量避免外部可控参数拼接到HTTP响应头中，如业务需要则需要过滤掉 `\r` 、 `\n` 等换行符，或者拒绝携带换行符号的外部输入。

1.6.4【必须】外部输入拼接到response页面前进行编码处理

- 直出html页面或使用模板生成html页面的，推荐使用 `text/template` 自动编码，或者使用 `html.EscapeString` 或 `text/template` 对 `<, >, &, ',"` 等字符进行编码。

```
1.  import(
2.      "html/template"
3.  )
4.
5.  func outtemplate(w http.ResponseWriter,r *http.Request) {
6.      param1 := r.URL.Query().Get("param1")
7.      tmpl := template.New("hello")
8.      tmpl, _ = tmpl.Parse(`{{define "T"}}{{.}}{{end}}`)
9.      tmpl.ExecuteTemplate(w, "T", param1)
10. }
```

1.7 会话管理

1.7.1【必须】安全维护session信息

- 用户登录时应重新生成session，退出登录后应清理session。 ```go import (“net/http” “github.com/gorilla/mux” “github.com/gorilla/handlers”)

//创建cookie func setToken(res http.ResponseWriter, req *http.Request) { expireToken := time.Now().Add(time.Minute* 30).Unix() expireCookie := time.Now().Add(time.Minute * 30) … cookie := http.Cookie{ Name: “Auth”, Value: signedToken, Expires: expireCookie, // 过期失效 HttpOnly: true, Path: “/“, Domain: “127.0.0.1”, Secure: true }

```
http.SetCookie(res, &cookie)
http.Redirect(res, req, "/profile", 307)
```

} // 删除cookie func logout(res http.ResponseWriter, req *http.Request) { deleteCookie := http.Cookie{ Name: “Auth”, Value: “none”, Expires: time.Now() } http.SetCookie(res, &deleteCookie) return }

```

#### 1.7.2【必须】CSRF防护

- 涉及系统敏感操作或可读取敏感信息的接口应校验`Referer`或添加`csrf_token`。

```go
// good
import (
    "net/http"
    "github.com/gorilla/csrf"
    "github.com/gorilla/mux"
)

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/signup", ShowSignupForm)
    r.HandleFunc("/signup/post", SubmitSignupForm)
    //使用csrf_token验证
    http.ListenAndServe(":8000",
        csrf.Protect([]byte("32-byte-long-auth-key"))(r))
}
```
```

1.8 访问控制

1.8.1【必须】默认鉴权

- 除非资源完全可对外开放，否则系统默认进行身份认证，使用白名单的方式放开不需要认证的接口或页面。
- 根据资源的机密程度和用户角色，以最小权限原则，设置不同级别的权限，如完全公开、登录可读、登录可写、特定用户可读、特定用户可写等
- 涉及用户自身相关的数据的读写必须验证登录态用户身份及其权限，避免越权操作

```
-- 伪代码
select id from table where id=:id and userid=session.userid
```

- 没有独立账号体系的外网服务使用 `QQ` 或 `微信` 登录，内网服务使用 `统一登录服务` 登录，其他使用账号密码登录的服务需要增加验证码等二次验证

1.9 并发保护

1.9.1【必须】禁止在闭包中直接调用循环变量

- 在循环中启动协程，当协程中使用到了循环的索引值，由于多个协程同时使用同一个变量会产生数据竞争，造成执行结果异常。

```
1.  // bad
2.  func main() {
3.      runtime.GOMAXPROCS(runtime.NumCPU())
4.      var group sync.WaitGroup
5.
6.      for i := 0; i < 5; i++ {
7.          group.Add(1)
8.          go func() {
9.              defer group.Done()
10.             fmt.Printf("%-2d", i) //这里打印的i不是所期望的
11.         }()
12.     }
13.     group.Wait()
14. }
15.
16. // good
17. func main() {
18.     runtime.GOMAXPROCS(runtime.NumCPU())
19.     var group sync.WaitGroup
20.
21.     for i := 0; i < 5; i++ {
22.         group.Add(1)
23.         go func(j int) {
24.             defer func() {
25.                 if r := recover(); r != nil {
26.                     fmt.Println("Recovered in start()")
27.                 }
28.                 group.Done()
29.             }()
30.         fmt.Printf("%-2d", j) // 闭包内部使用局部变量
31.         }(i)  // 把循环变量显式地传给协程
32.     }
33.     group.Wait()
34. }
```

1.9.2【必须】禁止并发写map

- 并发写map容易造成程序崩溃并异常退出，建议加锁保护

```
// bad
func main() {
  m := make(map[int]int)
  //并发读写
  go func() {
      for {
          _ = m[1]
      }
  }()
  go func() {
      for {
          m[2] = 1
      }
  }()
  select {}
}
```

1.9.3【必须】确保并发安全

敏感操作如果未作并发安全限制，可导致数据读写异常，造成业务逻辑限制被绕过。可通过同步锁或者原子操作进行防护。

通过同步锁共享内存

```
// good
var count int
func Count(lock *sync.Mutex) {
    lock.Lock()// 加写锁
    count++
    fmt.Println(count)
    lock.Unlock()// 解写锁，任何一个Lock()或RLock()均需要保证对应有Unlock()或RUnlock()
}

func main() {
    lock := &sync.Mutex{}
    for i := 0; i < 10; i++ {
        go Count(lock) //传递指针是为了防止函数内的锁和调用锁不一致
    }
```

```
    for {
        lock.Lock()
        c := count
        lock.Unlock()
        runtime.Gosched()//交出时间片给协程
        if c > 10 {
            break
        }
    }
}
```

- 使用 `sync/atomic` 执行原子操作

```
// good
import (
    "sync"
    "sync/atomic"
)

func main() {
    type Map map[string]string
    var m atomic.Value
    m.Store(make(Map))
    var mu sync.Mutex // used only by writers
    read := func(key string) (val string) {
        m1 := m.Load().(Map)
        return m1[key]
    }
    insert := func(key, val string) {
        mu.Lock() // 与潜在写入同步
        defer mu.Unlock()
        m1 := m.Load().(Map) // 导入struct当前数据
        m2 := make(Map)      // 创建新值
        for k, v := range m1 {
            m2[k] = v
        }
        m2[key] = val
        m.Store(m2)   // 用新的替代当前对象
    }
    _, _ = read, insert
}
```

- 安卓类
- 后台类

安卓类

I. 代码实现

1.1 异常捕获处理

1.1.1 【必须】序列化异常捕获

对于通过导出组件 intent 传递的序列化对象，必须进行 try…catch 处理，以避免数据非法导致应用崩溃。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        try {
            Intent mIntent = getIntent();
            //String msg = intent.getStringExtra("data");
            Person mPerson =
(Person)mIntent.getSerializableExtra(ObjectDemo.SER_KEY)
            //textView.setText(msg);
        } catch (ClassNotFoundException exp) {
            // ......
        }
    }
}
```

1.1.2 【必须】NullPointerException 异常捕获

对于通过 intent getAction 方法获取数据时，必须进行 try…catch 处理，以避免空指针异常导致应用崩溃。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        try {
            Intent mIntent = getIntent();
            if mIntent.getAction().equals("StartNewWorld") {
                // ......
            }
            // ......
        } catch (NullPointerException exp) {
            // ......
```

```
        }
    }
}
```

1.1.3 【必须】ClassCastException 异常捕获

对于通过 intent getSerializableExtra 方法获取数据时，必须进行 try…catch 处理，以避免类型转换异常导致应用崩溃。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        try {
            Intent mIntent = getIntent();
            Person mPerson =
(Person)mIntent.getSerializableExtra(ObjectDemo.SER_KEY)
            // ......
        } catch (ClassCastException exp) {
            // ......
        }
    }
}
```

1.1.4 【必须】ClassNotFoundException 异常捕获

同 1.1.3

1.2 数据泄露

1.2.1 【必须】logcat 输出限制

release 版本禁止在 logcat 输出信息。

```
public class MainActivity extends Activity {
    String DEBUG = "debug_version";

    protected void onCreate(Bundle savedInstanceState) {
        // ......
        if (DEBUG == "debug_version") {
            Log.d("writelog", "start activity");
        }
        // ......
```

```
    }
}
```

1.3 webview 组件安全

1.3.1 【必须】addJavaScriptInterface 方法调用

对于设置 minsdk <= 18 的应用，禁止调用 addJavaScriptInterface 方法。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        // ......
        mWebView = new WebView(this);
        if (Build.VERSION.SDK_INT > 18) {
            mWebView.addJavascriptInterface(new
wPayActivity.InJavaScriptLocalObj(this), "local_obj");
        }
        // ......
    }
}
```

1.3.2 【建议】setJavaScriptEnabled 方法调用

如非必要，setJavaScriptEnabled 应设置为 false 。加载本地 html ，应校验 html 页面完整性，以避免 xss 攻击。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        // ......
        mWebView = new WebView(this);
        mWebView.getSettings().setJavaScriptEnabled(false);
        // ......
    }
}
```

1.3.3 【建议】setAllowFileAccess 方法调用

建议禁止使用 File 域协议，以避免过滤不当导致敏感信息泄露。

```
public class MainActivity extends Activity {
```

```
2.
3.      protected void onCreate(Bundle savedInstanceState) {
4.          // ......
5.          mWebView = new WebView(this);
6.          mWebView.getSettings().setAllowFileAccess(false);
7.          // ......
8.      }
9.  }
```

1.3.4 【建议】setSavePassword 方法调用

建议 setSavePassword 的设置为 false ，避免明文保存网站密码。 建议禁止使用 File 域协议，以避免过滤不当导致敏感信息泄露。

```
1.  public class MainActivity extends Activity {
2.
3.      protected void onCreate(Bundle savedInstanceState) {
4.          // ......
5.          mWebView = new WebView(this);
6.          mWebView.getSettings().setSavePassword(false);
7.          // ......
8.      }
9.  }
```

1.3.5 【必须】onReceivedSslError 方法调用

webview 组件加载网页发生证书认证错误时，不能直接调用 handler.proceed() 忽略错误，应当处理当前场景是否符合业务预期，以避免中间人攻击劫持。

```
1.  public class MainActivity extends Activity {
2.
3.      protected void onCreate(Bundle savedInstanceState) {
4.          // ......
5.          mWebView = new WebView(this);
6.          mWebView.setWebViewClient(new WebViewClient() {
7.              @Override
                public void onReceivedSslError(WebView view, SslErrorHandler
8.  handler, SslError error) {
9.                  // must check error
10.                 check_error();
11.                 handler.proceed();
12.             }
```

```
        }
        // ......
    }
}
```

1.4 传输安全

1.4.1 【必须】自定义 HostnameVerifier 类

自定义 HostnameVerifier 类后，必须实现 verify 方法校验域名，以避免中间人攻击劫持。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        // ......
        HostnameVerifier hnv = new HostnameVerifier() {
            @Override
            public boolean verify(String hostname, SSLSession session) {
                // must to do
                isValid = checkHostName(hostname);
                return isValid;
            }
        };
        // ......
    }
}
```

1.4.2 【必须】自定义 X509TrustManager 类

自定义 X509TrustManager 类后，必须实现 checkServerTrusted 方法校验服务器证书，以避免中间人攻击劫持。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        // ......
        TrustManager tm = new X509TrustManager() {
            public void checkServerTrusted(X509Certificate[] chain, String
authType)
                    throws CertificateException {
                // must to do
                check_server_valid();
```

```
            }
        };
        // ......
    }
}
```

1.4.3 【必须】setHostnameVerifier 方法调用

禁止调用 setHostnameVerifier 方法设置 ALLOW_ALL_HOSTNAME_VERIFIER 属性，以避免中间人攻击劫持。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        // ......
        SchemeRegistry schemeregistry = new SchemeRegistry();
        SSLSocketFactory sslsocketfactory =
SSLSocketFactory.getSocketFactory();
        // set STRICT_HOSTNAME_VERIFIER

sslsocketfactory.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
        // ......
    }
}
```

II. 配置&环境

2.1 AndroidManifest.xml 配置

2.1.1 【必须】PermissionGroup 属性设置

禁止设置 PermissionGroup 属性为空。

2.1.2 【必须】protectionLevel 属性设置

对于自定义权限的 protectionLevel 属性设置，建议设置为 signature 或 signatureOrSystem。

2.1.3 【建议】sharedUserId 权限设置

最小范围和最小权限使用 sharedUserId 设置。

2.1.4 【建议】allowBackup 备份设置

如非产品功能需要，建议设置 allowBackup 为 false。

```
1.  <application android:allowBackup="false">
2.  </application>
```

2.1.5 【必须】debuggable 调试设置

release 版本禁止设置 debuggable 为 true。

```
1.  <application android:debuggable="false">
2.  </application>
```

后台类

I. 代码实现

1.1 数据持久化

1.1.1【必须】SQL语句默认使用预编译并绑定变量

Web后台系统应默认使用预编译绑定变量的形式创建sql语句，保持查询语句和数据相分离。以从本质上避免SQL注入风险。

如使用Mybatis作为持久层框架，应通过#{}语法进行参数绑定，MyBatis 会创建 `PreparedStatement` 参数占位符，并通过占位符安全地设置参数。

示例：JDBC

```
1.  String custname = request.getParameter("name");
2.  String query = "SELECT * FROM user_data WHERE user_name = ? ";
3.  PreparedStatement pstmt = connection.prepareStatement( query );
4.  pstmt.setString( 1, custname);
5.  ResultSet results = pstmt.executeQuery( );
```

Mybatis

```
    <select id="queryRuleIdByApplicationId" parameterType="java.lang.String"
1.  resultType="java.lang.String">
          select rule_id from scan_rule_sqlmap_tab where application_id=#
2.  {applicationId}
3.  </select>
```

应避免外部输入未经过滤直接拼接到SQL语句中，或者通过Mybatis中的${}传入SQL语句（即使使用PreparedStatement，SQL语句直接拼接外部输入也同样有风险。例如Mybatis中部分参数通过${}传入SQL语句后实际执行时调用的是PreparedStatement.execute()，同样存在注入风险）。

1.1.2【必须】白名单过滤

对于表名、列名等无法进行预编译的场景，比如外部数据拼接到order by, group by语句中，需通过白名单的形式对数据进行校验，例如判断传入列名是否存在、升降序仅允许输入“ASC”和“DESC”、表明列名仅允许输入字符、数字、下划线等。参考示例：

```
1.  public String someMethod(boolean sortOrder) {
     String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" :
2.  "DESC");`
3.   ...
```

1.10 操作业务

1.10.1【必须】部署CSRF防御机制

CSRF是指跨站请求伪造（Cross-site request forgery），是web常见的攻击之一。对于可重放的敏感操作请求，需部署CSRF防御机制。可参考以下两种常见的CSRF防御方式

- 设置CSRF Token

  服务端给合法的客户颁发CSRF Token，客户端在发送请求时携带该token供服务端校验，服务端拒绝token验证不通过的请求。以此来防止第三方构造合法的恶意操作链接。Token的作用域可以是Request级或者Session级。下面以Session级CSRF Token进行示例

  i. 登录成功后颁发Token，并同时存储在服务端Session中

```
String uuidToken = UUID.randomUUID().toString();
map.put("token", uuidToken);
request.getSession().setAttribute("token",uuidToken );
return map;
```

  ii. 创建Filter

```
public class CsrfFilter implements Filter {
  ...
   HttpSession session = req.getSession();
   Object token = session.getAttribute("token");
   String requestToken = req.getParameter("token");
   if(StringUtils.isBlank(requestToken) ||
!requestToken.equals(token)){
         AjaxResponseWriter.write(req, resp,
ServiceStatusEnum.ILLEGAL_TOKEN, "非法的token");
            return;
        }
   ...
```

  CSRF Token应具备随机性，保证其不可预测和枚举。另外由于浏览器会自动对表单所访问的域名添加相应的cookie信息，所以CSRF Token不应该通过Cookie传输。

- 校验Referer头

  通过检查HTTP请求的Referer字段是否属于本站域名，非本站域名的请求进行拒绝。

  这种校验方式需要注意两点：

i. 要需要处理Referer为空的情况，当Referer为空则拒绝请求
ii. 注意避免例如qq.com.evil.com 部分匹配的情况。

1.10.2【必须】权限校验

对于非公共操作，应当校验当前访问账号进行操作权限（常见于CMS）和数据权限校验。

1. 验证当前用户的登录态
2. 从可信结构中获取经过校验的当前请求账号的身份信息（如：session）。禁止从用户请求参数或Cookie中获取外部传入不可信用户身份直接进行查询。
3. 校验当前用户是否具备该操作权限
4. 校验当前用户是否具备所操作数据的权限。避免越权。

1.10.3【建议】加锁操作

对于有次数限制的操作，比如抽奖。如果操作的过程中资源访问未正确加锁。在高并发的情况下可能造成条件竞争，导致实际操作成功次数多于用户实际操作资格次数。此类操作应加锁处理。

1.2 文件操作

1.2.1【必须】文件类型限制

须在服务器端采用白名单方式对上传或下载的文件类型、大小进行严格的限制。仅允许业务所需文件类型上传，避免上传.jsp、.jspx、.class、.java等可执行文件。参考示例：

```
        String file_name = file.getOriginalFilename();
         String[] parts = file_name.split("\\.");
         String suffix = parts[parts.length - 1];
         switch (suffix){
             case "jpeg":
                 suffix = ".jpeg";
                 break;
             case "jpg":
                 suffix = ".jpg";
                 break;
             case "bmp":
                 suffix = ".bmp";
                 break;
             case "png":
                 suffix = ".png";
                 break;
             default:
                 //handle error
                 return "error";
         }
```

1.2.2【必须】禁止外部文件存储于可执行目录

禁止外部文件存储于WEB容器的可执行目录（appBase）。建议保存在专门的文件服务器中。

1.2.3【建议】避免路径拼接

文件目录避免外部参数拼接。保存文件目录建议后台写死并对文件名进行校验（字符类型、长度）。建议文件保存时，将文件名替换为随机字符串。

1.2.4【必须】避免路径穿越

如因业务需要不能满足1.2.3的要求，文件路径、文件命中拼接了不可行数据，需判断请求文件名和文件路径参数中是否存在../或..\(仅windows)， 如存在应判定路径非法并拒绝请求。

1.3 网络访问

1.3.1【必须】避免直接访问不可信地址

服务器访问不可信地址时，禁止访问私有地址段及内网域名。

```
// 以RFC定义的专有网络为例，如有自定义私有网段亦应加入禁止访问列表。
10.0.0.0/8
172.16.0.0/12
192.168.0.0/16
127.0.0.0/8
```

建议通过URL解析函数进行解析，获取host或者domain后通过DNS获取其IP，然后和内网地址进行比较。

对已校验通过地址进行访问时，应关闭跟进跳转功能。

参考示例：

```
        httpConnection = (HttpURLConnection) Url.openConnection();

        httpConnection.setFollowRedirects(false);
```

1.4 XML读写

1.4.1【必须】XML解析器关闭DTD解析

读取外部传入XML文件时，XML解析器初始化过程中设置关闭DTD解析。

参考示例：

javax.xml.parsers.DocumentBuilderFactory

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
try {
    dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
true);
    dbf.setFeature("http://xml.org/sax/features/external-general-entities",
false);
    dbf.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
    dbf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-
dtd", false);
    dbf.setXIncludeAware(false);
    dbf.setExpandEntityReferences(false);
    ……
}
```

org.dom4j.io.SAXReader

```
saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
true);
saxReader.setFeature("http://xml.org/sax/features/external-general-entities",
false);
saxReader.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
```

org.jdom2.input.SAXBuilder

```
SAXBuilder builder = new SAXBuilder();
builder.setFeature("http://apache.org/xml/features/disallow-doctype-
decl",true);
builder.setFeature("http://xml.org/sax/features/external-general-entities",
false);
builder.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
```

```
5.  Document doc = builder.build(new File(fileName));
```

org.xml.sax.XMLReader

```
1.  XMLReader reader = XMLReaderFactory.createXMLReader();
    reader.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
2.  true);
    reader.setFeature("http://apache.org/xml/features/nonvalidating/load-external-
3.  dtd", false);
    reader.setFeature("http://xml.org/sax/features/external-general-entities",
4.  false);
    reader.setFeature("http://xml.org/sax/features/external-parameter-entities",
5.  false);
```

1.5 响应输出

1.5.1【必须】设置正确的HTTP响应包类型

响应包的HTTP头“Content-Type”必须正确配置响应包的类型，禁止非HTML类型的响应包设置为“text/html”。此举会使浏览器在直接访问链接时，将非HTML格式的返回报文当做HTML解析，增加反射型XSS的触发几率。

1.5.2【建议】设置安全的HTTP响应头

- X-Content-Type-Options:

建议添加“X-Content-Type-Options”响应头并将其值设置为“nosniff”，可避免部分浏览器根据其“Content-Sniff”特性，将一些非“text/html”类型的响应作为HTML解析，增加反射型XSS的触发几率。

- HttpOnly:

控制用户登录鉴权的Cookie字段 应当设置HttpOnly属性以防止被XSS漏洞/JavaScript操纵泄漏。

- X-Frame-Options:

设置X-Frame-Options响应头，并根据需求合理设置其允许范围。该头用于指示浏览器禁止当前页面在frame、iframe、embed等标签中展现。从而避免点击劫持问题。它有三个可选的值： DENY： 浏览器会拒绝当前页面加载任何frame页面； SAMEORIGIN: 则frame页面的地址只能为同源域名下的页面 ALLOW-FROM origin: 可以定义允许frame加载的页面地址。

- Access-Control-Allow-Origin

  当需要配置CORS跨域时，应对请求头的Origin值做严格过滤。

```
...
String currentOrigin = request.getHeader("Origin");
if (currentOrigin.equals("https://domain.qq.com")) {
        response.setHeader("Access-Control-Allow-Origin", currentOrigin);
            }
 ...
```

1.5.3【必须】外部输入拼接到response页面前进行编码处理

当响应“content-type”为“html”类型时，外部输入拼接到响应包中，需根据输出位置进行编码处理。编码规则：

场景	编码规则
	需要对以下6个特殊字符进行HTML实体编码(&, <, >, “, ‘,/)。

输出点在HTML标签之间	示例: & –> & < –> < >–> > “ –> " ‘ –> ' / –> /
输出点在HTML标签普通属性内（如href、src、style等，on事件除外）	要对数据进行HTML属性编码。 编码规则：除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为&#xHH;(以&#x开头，HH则是指该字符对应的十六进制数字，分号作为结束符)
输出点在JS内的数据中	需要进行js编码 编码规则： 除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为 \xHH （以 \x 开头，HH则是指该字符对应的十六进制数字） Tips：这种场景仅限于外部数据拼接在js里被引号括起来的变量值中。除此之外禁止直接将代码拼接在js代码中。
输出点在CSS中（Style属性）	需要进行CSS编码 编码规则： 除了阿拉伯数字和字母，对其他所有的字符进行编码，只要该字符的ASCII码小于256。编码后输出的格式为 \HH （以 \ 开头，HH则是指该字符对应的十六进制数字）
输出点在URL属性中	对这些数据进行URL编码 Tips：除此之外，所有链接类属性应该校验其协议。禁止JavaScript、data和Vb伪协议。

以上编码规则相对较为繁琐，可参考或直接使用业界已有成熟第三方库如ESAPI.其提供一下函数对象上表中的编码规则：

```
1. ESAPI.encoder().encodeForHTML();
2. ESAPI.encoder().encodeForHTMLAttribute();
3. ESAPI.encoder().encodeForJavaScript();
4. ESAPI.encoder().encodeForCSS();
5. ESAPI.encoder().encodeForURL();
```

1.5.4【必须】外部输入拼接到HTTP响应头中需进行过滤

应尽量避免外部可控参数拼接到HTTP响应头中，如业务需要则需要过滤掉“\r”、”\n”等换行符，或者拒绝携带换行符号的外部输入。

1.5.5【必须】避免不可信域名的302跳转

如果对外部传入域名进行302跳转，必须设置可信域名列表并对传入域名进行校验。

为避免校验被绕过，应避免直接对URL进行字符串匹配。应通过通过URL解析函数进行解析，获取host或者domain后和白名单进行比较。

需要注意的是，由于浏览器的容错机制，域名 `https://www.qq.com\www.bbb.com` 中的 `\` 会被替换成 `/` ，最终跳转到 `www.qq.com` 。而Java的域名解析函数则无此特性。为避免解析不一致导致绕过，建议对host中的 `/` 和 `#` 进行替换。

参考代码：

```
String host="";
        try {
            url = url.replaceAll("[\\\\#]","/"); //替换掉反斜线和井号
            host = new URL(url).getHost();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        if (host.endsWith(".qq.com")){
            //跳转操作
        }else{
            return;
        }
```

1.5.6【必须】避免通过Jsonp传输非公开敏感信息

jsonp请求再被CSRF攻击时，其响应包可被攻击方劫持导致信息泄露。应避免通过jsonp传输非公开的敏感信息，例如用户隐私信息、身份凭证等。

1.5.7【必须】限定JSONP接口的callback字符集范围

JSONP接口的callback函数名为固定白名单。如callback函数名可用户自定义，应限制函数名仅包含 字母、数字和下划线。如：`[a-zA-Z0-9_-]+`

1.5.8【必须】屏蔽异常栈

应用程序出现异常时，禁止将数据库版本、数据库结构、操作系统版本、堆栈跟踪、文件名和路径信息、SQL 查询字符串等对攻击者有用的信息返回给客户端。建议重定向到一个统一、默认的错误提示页面，进行信息过滤。

1.5.9【必须】模板&表达式

web view层通常通过模板技术或者表达式引擎来实现界面与业务数据分离，比如jsp中的EL表达式。这些引擎通常可执行敏感操作，如果外部不可信数据未经过滤拼接到表达式中进行解析。则可能造成严重漏洞。

下列是基于EL表达式注入漏洞的演示demo：

```
    @RequestMapping("/ELdemo")
    @ResponseBody
    public String ELdemo(RepeatDTO repeat) {
        ExpressionFactory expressionFactory = new ExpressionFactoryImpl();
        SimpleContext simpleContext = new SimpleContext();
        String exp = "${"+repeat.getel()+"}";
```

```
            ValueExpression valueExpression =
7.  expressionFactory.createValueExpression(simpleContext, exp, String.class);
8.          return valueExpression.getValue(simpleContext).toString();
9.      }
```

外部可通过el参数，将不可信输入拼接到EL表达式中并解析。

此时外部访问：x.x.x.x/ELdemo?el=”’.getClass().forName(‘java.lang.Runtime’).getMethod(‘exec’,’’.getClass()).invoke(‘’.getClass().forName(‘java.lang.Runtime’).getMethod(‘getRuntime’).invoke(null),’open /Applications/Calculator.app’)“ 可执行操作系统命令调出计算器。

基于以上风险：

- 应避免外部输入的内容拼接到EL表达式或其他表达式引起、模板引擎进行解析。
- 白名单过滤外部输入，仅允许字符、数字、下划线等。

1.6 OS命令执行

1.6.1【建议】避免不可信数据拼接操作系统命令

当不可信数据存在时，应尽量避免外部数据拼接到操作系统命令使用 `Runtime` 和 `ProcessBuilder` 来执行。优先使用其他同类操作进行代替，比如通过文件系统API进行文件操作而非直接调用操作系统命令。

1.6.2【必须】避免创建SHELL操作

如无法避免直接访问操作系统命令，需要严格管理外部传入参数，使不可信数据仅作为执行命令的参数而非命令。

- 禁止外部数据直接直接作为操作系统命令执行。
- 避免通过”cmd”、“bash”、“sh”等命令创建shell后拼接外部数据来执行操作系统命令。
- 对外部传入数据进行过滤。可通过白名单限制字符类型，仅允许字符、数字、下划线；或过滤转义以下符号：|;&$><`（反引号）！

  白名单示例：

```
    private static final Pattern FILTER_PATTERN = Pattern.compile("[0-9A-Za-
1.  z_]+");
2.  if (!FILTER_PATTERN.matcher(input).matches()) {
3.    // 终止当前请求的处理
4.  }
```

1.7 会话管理

1.7.1【必须】非一次有效身份凭证禁止在URL中传输

身份凭证禁止在URL中传输，一次有效的身份凭证除外（如CAS中的st）。

1.7.2【必须】避免未经校验的数据直接给会话赋值

防止会话信息被篡改，如恶意用户通过URL篡改手机号码等。

1.8 加解密

1.8.1【建议】对称加密

建议使用AES，秘钥长度128位以上。禁止使用DES算法，由于秘钥太短，其为目前已知不安全加密算法。使用AES加密算法请参考以下注意事项：

- AES算法如果采用CBC模式：每次加密时IV必须采用密码学安全的伪随机发生器（如/dev/urandom），禁止填充全0等固定值。
- AES算法如采用GCM模式，nonce须采用密码学安全的伪随机数
- AES算法避免使用ECB模式，推荐使用GCM模式。

1.8.2【建议】非对称加密

建议使用RSA算法，秘钥2048及以上。

1.8.3【建议】哈希算法

哈希算法推荐使用SHA-2及以上。对于签名场景，应使用HMAC算法。如果采用字符串拼接盐值后哈希的方式，禁止将盐值置于字符串开头，以避免哈希长度拓展攻击。

1.8.4【建议】密码存储策略

建议采用随机盐+明文密码进行多轮哈希后存储密码。

1.9 查询业务

1.9.1【必须】返回信息最小化

返回用户信息应遵循最小化原则，避免将业务需求之外的用户信息返回到前端。

1.9.2【必须】个人敏感信息脱敏展示

在满足业务需求的情况下，个人敏感信息需脱敏展示,如:

- 鉴权信息（如口令、密保答案、生理标识等）不允许展示
- 身份证只显示第一位和最后一位字符，如3**1。
- 移动电话号码隐藏中间6位字符，如134**48。
- 工作地址/家庭地址最多显示到“区”一级。
- 银行卡号仅显示最后4位字符，如**8639

1.9.3【必须】数据权限校验

查询个人非公开信息时，需要对当前访问账号进行数据权限校验。

1. 验证当前用户的登录态
2. 从可信结构中获取经过校验的当前请求账号的身份信息（如：session）。禁止从用户请求参数或Cookie中获取外部传入不可信用户身份直接进行查询。
3. 验当前用户是否具备访问数据的权限

- 通用类
- 后台类

通用类

I. 代码实现

1.1 加密算法

1.1.1【必须】避免使用不安全的哈希算法

- DES和3DES已经不再现代应用程序，应改为使用AES。

1.2 程序日志

1.2.1 【建议】对每个重要行为都记录日志

- 确保重要行为都记录日志，且可靠保存6个月以上。

1.2.2 【建议】禁止将未经验证的用户输入直接记录日志

- 当日志条目包含未经净化的用户输入时会引发记录注入漏洞。恶意用户会插入伪造的日志数据，从而让系统管理员以为是系统行为。

1.2.3 【建议】避免在日志中保存敏感信息

- 不能在日志保存密码（包括明文密码和密文密码）、密钥和其它敏感信息

1.3 系统口令

1.3.1【必须】禁止使用空口令、弱口令、已泄露口令

1.3.2 【必须】口令强度要求

> 口令强度须同时满足：
>
> 1. 密码长度大于14位
> 2. 必须包含下列元素：大小写英文字母、数字、特殊字符
> 3. 不得使用各系统、程序的默认初始密码
> 4. 不能与最近6次使用过的密码重复
> 5. 不得与其他外部系统使用相同的密码

1.3.3 【必须】口令存储安全

- 禁止明文存储口令
- 禁止使用弱密码学算法（如DES和3DES）加密存储口令
- 使用不可逆算法和随机salt对口令进行加密存储

1.3.4【必须】禁止传递明文口令

1.3.5 【必须】禁止在不安全的信道中传输口令

II. 配置&环境

2.1 Python版本选择

2.1.1【建议】使用Python 3.6+的版本

- 新增的项目应使用 Python 3.6+

> 为什么要这么做？ 由于 Python 2 在 2020 年停止维护，相关组件的漏洞不能得到及时修复与维护

2.2 第三方包安全

2.2.2 【必须】禁止使用不安全的组件

2.3 配置信息

2.3.1 【必须】密钥存储安全

- 在使用对称密码算法时，需要保护好加密密钥。当算法涉及敏感、业务数据时，可通过非对称算法协商加密密钥。其他较为不敏感的数据加密，可以通过变换算法等方式保护密钥。

2.3.2【必须】禁止硬编码敏感配置

- 禁止在源码中硬编码AK/SK、IP、数据库账密等配置信息
- 应使用配置系统或KMS密钥管理系统。

后台类

I. 代码实现

1.1 输入验证

1.1.1【必须】按类型进行数据校验

- 所有程序外部输入的参数值，应进行数据校验。校验内容包括但不限于：数据长度、数据范围、数据类型与格式。校验不通过，应拒绝。
- 推荐使用组件：Cerberus、jsonschema、Django-Validators

```
# Cerberus示例
v = Validator({'name': {'type': 'string'}})
v.validate({'name': 'john doe'})

# jsonschema示例
schema = {
     "type" : "object",
     "properties" : {
         "price" : {"type" : "number"},
         "name" : {"type" : "string"},
     },
}

validate(instance={"name" : "Eggs", "price" : 34.99}, schema=schema)
```

1.10 异常处理

1.10.1【必须】不向对外错误提示

- 应合理使用`try/except/finally`处理系统异常，避免出错信息输出到前端。
- 对外环境禁止开启debug模式，或将程序运行日志输出到前端。

1.10.2 【必须】禁止异常抛出敏感信息

1.11 Flask安全

1.11.1【必须】生产环境关闭调试模式

1.11.2【建议】遵循Flask安全规范

- 参考Flask文档中的安全注意事项
  https://flask.palletsprojects.com/en/master/security/

1.12 Django安全

1.12.1【必须】生产环境关闭调试模式

1.12.2【建议】保持Django自带的安全特性开启

- 保持Django自带的安全特性开启
  https://docs.djangoproject.com/en/3.0/topics/security/
- 在默认配置下，Django自带的安全特性对XSS、CSRF、SQL注入、点击劫持等类型漏洞可以起到较好防护效果。应尽量避免关闭这些安全特性。

1.2 SQL操作

1.2.1 【必须】使用参数化查询

- 使用参数化SQL语句，强制区分数据和命令，避免产生SQL注入漏洞。

```
# 错误示例
import mysql.connector

mydb = mysql.connector.connect(
... ...
)

cur = mydb.cursor()
userid = get_id_from_user()
# 使用%直接格式化字符串拼接SQL语句
cur.execute("SELECT `id`, `password` FROM `auth_user` WHERE `id`=%s " %
(userid,))
myresult = cur.fetchall()
```

```
# 安全示例
import mysql.connector

mydb = mysql.connector.connect(
... ...
)
cur = mydb.cursor()
userid = get_id_from_user()
# 将元组以参数的形式传入
cur.execute("SELECT `id`, `password` FROM `auth_user` WHERE `id`=%s " ,
(userid,))
myresult = cur.fetchall()
```

- 推荐使用ORM框架来操作数据库，如：使用 `SQLAlchemy` 。

```
# 安装sqlalchemy并初始化数据库连接
# pip install sqlalchemy
from sqlalchemy import create_engine
# 初始化数据库连接，修改为你的数据库用户名和密码
engine =
create_engine('mysql+mysqlconnector://user:password@host:port/DATABASE')
```

```python
# 引用数据类型
from sqlalchemy import Column, String, Integer, Float
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
# 定义 Player 对象:
class Player(Base):
    # 表的名字:
    __tablename__ = 'player'

    # 表的结构:
    player_id = Column(Integer, primary_key=True, autoincrement=True)
    team_id = Column(Integer)
    player_name = Column(String(255))
    height = Column(Float(3, 2))
```

```python
# 增删改查
from sqlalchemy.orm import sessionmaker
# 创建 DBSession 类型:
DBSession = sessionmaker(bind=engine)
# 创建 session 对象:
session = DBSession()

# 增:
new_player = Player(team_id=101, player_name="Tom", height=1.98)
session.add(new_player)
# 删:
row = session.query(Player).filter(Player.player_name=="Tom").first()
session.delete(row)
# 改:
row = session.query(Player).filter(Player.player_name=="Tom").first()
row.height = 1.99
# 查:
rows = session.query(Player).filter(Player.height >= 1.88).all()

# 提交即保存到数据库:
session.commit()
# 关闭 session:
session.close()
```

1.2.2 【必须】对参数进行过滤

- 将接受到的外部参数动态拼接到SQL语句时，必须对参数进行安全过滤。

```
def sql_filter(sql, max_length=20):
    dirty_stuff = ["\"", "\\", "/", "*", "'", "=", "-", "#", ";", "<", ">",
"+",
                   "&", "$", "(", ")", "%", "@", ","]
    for stuff in dirty_stuff:
        sql = sql.replace(stuff, "x")
    return sql[:max_length]
```

1.3 执行命令

1.3.1【建议】避免直接调用函数执行系统命令

- 相关功能的实现应避免直接调用系统命令（如 `os.system()` 、 `os.popen()` 、 `subprocess.call()` 等），优先使用其他同类操作进行代替，比如：通过文件系统API进行文件操作而非直接调用操作系统命令
- 如评估无法避免，执行命令应避免拼接外部数据，同时进行执行命令的白名单限制。

1.3.2【必须】过滤传入命令执行函数的字符

- 程序调用各类函数执行系统命令时，如果涉及的命令由外部传入，过滤传入命令执行函数的字符。

```
1.  import os
2.  import sys
3.  import shlex
4.
5.  domain = sys.argv[1]
6.  # 替换可以用来注入命令的字符为空
7.  badchars = "\n&;|'\"$()`-"
8.  for char in badchars:
9.      domain = domain.replace(char, " ")
10.
11. result = os.system("nslookup " + shlex.quote(domain))
```

1.4 XML读写

1.4.1 【必须】禁用外部实体的方法

- 禁用外部实体的方法，来预防XXE攻击。

```
1.  from lxml import etree
2.
3.  xmlData = etree.parse(xmlSource,etree.XMLParser(resolve_entities=False))
```

1.5 文件操作

1.5.1【必须】文件类型限制

- 通过白名单对上传或者下载的文件类型、大小进行严格校验。仅允许业务所需文件类型上传，避免上传木马、WebShell等文件。

```
import os

ALLOWED_EXTENSIONS = ['txt','jpg','png']

def allowed_file(filename):
    if ('.' in filename and
        '..' not in filename and
        os.path.splitext(filename)[1].lower() in ALLOWED_EXTENSIONS):

        return filename
    return None
```

1.5.2 【必须】禁止外部文件存储于可执行目录

- 禁止外部文件存储于WEB容器的可执行目录（appBase）。建议使用 `tempfile` 库处理临时文件和临时目录。

1.5.3 【必须】避免路径穿越

- 保存在本地文件系统时，必须对路径进行合法校验，避免目录穿越漏洞

```
import os

upload_dir = '/tmp/upload/' # 预期的上传目录
file_name = '../../etc/hosts' # 用户传入的文件名
absolute_path = os.path.join(upload_dir, file_name) #
/tmp/upload/../../etc/hosts
normalized_path = os.path.normpath(absolute_path) # /etc/hosts
if not normalized_path.startswith(upload_dir): # 检查最终路径是否在预期的上传目录中
    raise IOError()
```

1.5.4 【建议】避免路径拼接

- 文件目录避免外部参数拼接。保存文件目录建议后台写死并对文件名进行校验（字符类型、长

度）。

1.5.5 【建议】文件名hash化处理

- 建议文件保存时，将文件名替换为随机字符串。

```
1.  import uuid
2.
3.  def random_filename(filename):
4.      ext = os.path.splitext(filename)[1]
5.      new_filename = uuid.uuid4().hex + ext
6.      return new_filename
```

1.6 网络请求

1.6.1 【必须】限定访问网络资源地址范围

当程序需要从用户指定的 `URL地址获取网页文本内容` 、 `加载指定地址的图片` 、 `进行下载` 等操作时，需要对URL地址进行安全校验：

1. 只允许HTTP或HTTPS协议
2. 解析目标URL，获取其host
3. 解析host，获取host指向的IP地址转换成long型
4. 检查IP地址是否为内网IP

```
# 以RFC定义的专有网络为例，如有自定义私有网段亦应加入禁止访问列表。
10.0.0.0/8
172.16.0.0/12
192.168.0.0/16
127.0.0.0/8
```

1. 请求URL
2. 如果有跳转，跳转后执行1，否则对URL发起请求

1.7 响应输出

1.7.1【必须】设置正确的HTTP响应包类型

响应包的HTTP头“Content-Type”必须正确配置响应包的类型，禁止非HTML类型的响应包设置为“text/html”。

1.7.2【必须】设置安全的HTTP响应头

- X-Content-Type-Options

  添加“X-Content-Type-Options”响应头并将其值设置为“nosniff ”

- HttpOnly 控制用户登鉴权的Cookie字段 应当设置HttpOnly属性以防止被XSS漏洞/JavaScript操纵泄漏。

- X-Frame-Options

  设置X-Frame-Options响应头，并根据需求合理设置其允许范围。该头用于指示浏览器禁止当前页面在frame、 iframe、embed等标签中展现。从而避免点击劫持问题。它有三个可选的值：DENY: 浏览器会拒绝当前页面加 载任何frame页面; SAMEORIGIN:则frame页面的地址只能为同源域名下的页面 ALLOW-FROM origin:可以定 义允许frame加载的页面地址。

1.7.3【必须】对外输出页面包含第三方数据时须进行编码处理

- 当响应“Content-Type”为“text/html”类型时，需要对响应体进行编码处理

```
# 推荐使用mozilla维护的bleach库来进行过滤
import bleach
bleach.clean('an <script>evil()</script> example')
# u'an &lt;script&gt;evil()&lt;/script&gt; example'
```

1.8 数据输出

1.8.1【必须】敏感数据加密存储

- 敏感数据应使用SHA2、RSA等算法进行加密存储
- 敏感数据应使用独立的存储层，并在访问层开启访问控制
- 包含敏感信息的临时文件或缓存一旦不再需要应立刻删除

1.8.2【必须】敏感信息必须由后台进行脱敏处理

- 敏感信息须再后台进行脱敏后返回，禁止接口返回敏感信息交由前端/客户端进行脱敏处理。

1.8.3【必须】高敏感信息禁止存储、展示

- 口令、密保答案、生理标识等鉴权信息禁止展示
- 非金融类业务，信用卡cvv码及日志禁止存储

1.8.4【必须】个人敏感信息脱敏展示

在满足业务需求的情况下，个人敏感信息需脱敏展示，如:

- 身份证只显示第一位和最后一位字符，如3**1。
- 移动电话号码隐藏中间6位字符，如134**48。
- 工作地址/家庭地址最多显示到“区”一级。
- 银行卡号仅显示最后4位字符，如**8639

1.8.5【必须】隐藏后台地址

- 若程序对外提供了登录后台地址，应使用随机字符串隐藏地址。

```
# 不要采取这种方式
admin_login_url = "xxxx/login"
```

```
# 安全示例
admin_login_url = "xxxx/ranD0Str"
```

1.9 权限管理

1.9.1【必须】默认鉴权

- 除非资源完全可对外开放，否则系统默认进行身份认证（使用白名单的方式放开不需要认证的接口或页面）。

1.9.2【必须】授权遵循最小权限原则

- 程序默认用户应不具备任何操作权限。

1.9.3【必须】避免越权访问

- 对于非公共操作，应当校验当前访问账号进行操作权限（常见于CMS）和数据权限校验。

1. 验证当前用户的登录态；
2. 从可信结构中获取经过校验的当前请求账号的身份信息（如：session），禁止从用户请求参数或Cookie中获取外部传入不可信用户身份直接进行查询；
3. 校验当前用户是否具备该操作权限；
4. 校验当前用户是否具备所操作数据的权限；
5. 校验当前操作是否账户是否预期账户。

1.9.4【建议】及时清理不需要的权限

- 程序应定期清理非必需用户的权限。