中软国际公司内部技术规范

C#语言安全编程规范

中软国际科技服务有限公司

修订声明

参考:华为 C#语言安全编码规范。

0.

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

目录

前言

背景

*《C#语言安全编程规范》*针对 C#语言编程中的输入校验、异常行为、I/O 操作、序列化、平台安全等方面，描述可能导致安全漏洞或风险的常见编码错误。基于业界最佳实践，参考业界安全编码规范相关著作，如 *MSDN Secure Coding Guideline*、*CWE/SANS TOP 25* 和 *OWASP Guide Project*，总结形成公司内部的安全编程规范。该规范旨在指导开发人员在编码过程中避免或减少 SQL 注入、敏感信息泄露、格式化字符串攻击、命令注入攻击、目录遍历等安全问题的发生。

使用对象

本规范读者及使用对象为使用 C#语言的开发人员和测试人员。

适用范围

本规范适用于基于 C#语言开发的产品和平台。
除非特别说明，所有的代码示例都是基于.NET 4.0。

术语定义

规则：编程时必须遵守的约定
说明：对此规则进行必要的解释
错误示例：对此规则从反面给出例子
正确示例：对此规则从正面给出例子
例外情况：此规则的例外场景
信任边界：程序数据或执行改变信任级别的边界。位于信任边界之内的所有组件都是被系统本身直接控制的。所有来自不受控外部系统的连接与数据，都是不可信的，如来自客户端与第三方系统的连接与数据，都需要在边界处对其进行校验，才能允许其进一步与本系统交互。
非信任代码：非产品包中的代码，如通过网络下载到本地 CLR 中加载并执行的代码。

1. 数据校验

规则 1.1 对跨信任边界的不可信数据进行校验

说明：程序可能会接收来自用户、网络连接或其他来源的不可信数据，并将这些数据跨信任边界传递到目标系统（如浏览器、数据库等）。由于目标系统可能无法区分处理畸形的不可信数据，未经校验的不可信数据可能会引起某种注入攻击，对系统造成严重影响，因此，必须对不可信数据进行校验，且数据校验必须在信任边界以内进行（如对于 Web 应用，需要在服务端做校验）。数据校验有输入校验和输出校验，对从信任边界外传入的数据进行校验的叫输入校验，对传出到信任边界外的数据进行校验的叫输出校验。

如下描述了四种数据校验策略（任何时候，尽可能使用接收已知合法数据的“白名单”策略）：

接受已知合法数据

这种策略被称为“白名单”或者“正向”校验。该策略检查数据是否属于一个严格约束的、已知的、可接受的合法数据集合。例如，下面的示例代码确保 name 参数只包含字母、数字以及下划线。

```
// ...

using  System.Text.RegularExpressions;

if (!Regex.Match(name, @"^[0-9A-Za-z_]+$").Success)

{

    throw new ArgumentOutOfRangeException("Invalid name");

}

// ...
```

拒绝已知非法数据

这种策略被称为“黑名单”或者“负向”校验，相对于“正向”校验，这是一种较弱的校验方式。由于潜在的不合法数据可能是一个未知的无限集合，这种策略意味着你必须一直维护一个已知不合法字符或模式的列表。如果不定期研究新的攻击方式并对校验的表达式进行日常更新，该校验方式就会很快过时。例如，下面代码使用正则表达式将包含“javascript”的输入替换成空字符串。

```
public static String removeJavascript(String input)

{

    Regex regex = new Regex(@"javascript",RegexOptions.IgnoreCase);

    Match m = regex.Match(input);

    return (!m.Success) ? input : "";
```

```
}
```

“白名单”方式净化

对任何不属于已验证合法字符数据中的字符进行净化，然后再使用净化后的数据，净化的方式包括删除、编码、替换。比如，如果你期望接收一个电话号码，那么你可以删除掉输入中所有的非数字字符，“(555)123-1234”，“555.123.1234”，与“555\";DROP TABLE USER;--123.1234”全部会被转换为“5551231234”，然后再对转换的结果进行校验。又比如，对于用户评论栏的文本输入，由于几乎所有的字符都可能被用到，确定一个合法的数据集合是非常困难的，一种解决方案是对所有非字母数字进行编码，如对“I like your web page!”使用 URL 编码，其净化后的输出为“I+like+your+web+page%21”。“白名单”方式净化不仅有利于安全，它也允许接收和使用更宽泛的有效用户输入。

“黑名单”方式净化

为了确保输入数据是“安全”的，可以剔除或者转换某些字符（例如，删除引号、转换成 HTML 实体）。跟“黑名单”校验类似，随着时间推移不合法字符的范围很可能不一样，需要对不合法字符进行日常维护。因此，执行一个单纯针对正确输入的“正向”校验更加简单、高效、安全。

```
public static string QuoteApostrophe(string input)
{
    if (input != null)
    {
        return input.Replace("\'", "&rsquo;");
    }
    else
    {
        return null;
    }
}
```

规则 1.2 禁止使用未经校验的不可信数据来拼接 SQL 语句

说明：SQL 注入是指，在运行时原始 SQL 语句被恶意动态参数更改成与程序预期不一致的 SQL 命令，执行更改后的 SQL 命令可能导致信息泄露或者数据被篡改。防范 SQL 注入的方式主要分为两类：

- 使用参数化查询
- 对不可信数据进行校验

参数化查询是一种简单有效的防止 SQL 注入的方式，应该优先考虑使用。同时，参数化查询还能改进数据库访问性能，如 SQL Server 与 Oracle 数据库会为其缓存一个查询计划，以便在重复执行相同的查询时提高性能。

错误示例（C#代码动态构建 SQL）：

```
try
{
    SqlConnection conn = new SqlConnection(***);
    conn.Open();
    string userName = ctx.GetAuthenticatedUserName(); //this is a constant
    string sqlString = @"SELECT * FROM t_item WHERE owner='" + userName + @"' AND
                    itemName='" + ItemName.Text + "'"; // ItemName.Text is user input
    var sda = new SqlDataAdapter(sqlString, conn);
    // ... result set handling
    DataTable dt = new DataTable();
    sda.Fill(dt);
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

示例中拼接查询字符串常量和用户输入来动态构建 SQL 查询命令，仅当 ItemName.Text 不包含单引号时，这条查询语句的行为与预期是一致的。如果攻击者以用户名 wiley 发起请求，并使用以下名称参数进行查询：

name' OR 'a' = 'a 那么查询语句将变成：

```
SELECT * FROM t_item WHERE owner = 'wiley' AND itemname = 'name'
OR 'a'='a';
```

`OR 'a'='a'`条件导致整个 WHERE 子句的执行结果总为真，整个查询便等价于：

```
SELECT * FROM t_item
```

此查询会绕过原有的条件限制，返回 t_item 表中的所有条目，而不管它们的所有者是谁（原本只返回属于当前已认证用户的条目）

正确示例（使用 SqlCommand 进行参数化查询，ADO.net）：

```
try
```

```
{
    SqlConnection conn = new SqlConnection(***);
    conn.Open();
    string userName = ctx.GetAuthenticatedUserName(); //this is a constant
    string itemName = ItemName.Text;
    // ...Ensure that the length of userName and itemName is legitimate
    // ...
    string sqlString = "SELECT * FROM t_item WHERE owner=@owner AND
                        itemName=@itemname";
    SqlCommand cmd = new SqlCommand(sqlString, conn);
    cmd.Parameters.AddWithValue("@owner", userName);
    cmd.Parameters.AddWithValue("@itemname", itemName);
    SqlDataReader reader = cmd.ExecuteReader();
    // ... result set handling
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

参数化查询在 SQL 语句中使用占位符表示需在运行时确定的参数值，使得 SQL 查询的语义逻辑预先被定义，实际的查询参数值则在程序运行时再确定。参数化查询使得数据库能够区分 SQL 语句中语义逻辑和数据参数，以确保用户输入无法改变预期的 SQL 查询语义逻辑。在 C#中，可以使用 SqlCommand.Parameters.AddWithValue 进行参数化查询。正确示例中，如果攻击者的 itemName 输入为 name' OR 'a' = 'a，参数化查询将会查找 itemName 匹配 name' OR 'a' = 'a 字符串的条目，而不是返回整个表中的所有条目。

错误示例（在存储过程中动态构建 SQL）：

C#代码：

```
try
{
    string userName = ctx.getAuthenticatedUserName(); //this is a constant
    String itemName = request.getParameter("itemName");
    SqlConnection conn = new SqlConnection(***);
```

```
    conn.Open();

    SqlCommand cmd = new SqlCommand(sqlString, conn);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Connection = conn;
    cmd.CommandText = "sp_queryItem"; // stored procedure name
    cmd.Parameters.AddWithValue("@owner", userName);
    cmd.Parameters.AddWithValue("@itemname", itemName);
    SqlDataReader reader = cmd.ExecuteReader();
    // ... result set handling
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

SQL Server 存储过程：

```
CREATE PROCEDURE sp_queryItem
    @userName varchar(50),
    @itemName varchar(50)
AS
BEGIN
    DECLARE @sql nvarchar(500);
    SET @sql = 'SELECT * FROM t_item
                WHERE owner = ''' + @userName + '''
                AND itemName = ''' + @itemName + '''';
    EXEC(@sql);
END
GO
```

在存储过程中，通过拼接参数值来构建查询字符串，和在应用程序代码中拼接参数一样，同样有 SQL 注入风险的。

正确示例（在存储过程中进行参数化查询）：

C#代码：

```
try

{

    string userName = ctx.getAuthenticatedUserName(); //this is a constant

    string itemName = request.getParameter("itemName");

    SqlConnection conn = new SqlConnection(***);

    conn.Open();

    SqlCommand cmd = new SqlCommand(sqlString, conn);

    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Connection = conn;

    cmd.CommandText = "sp_queryItem"; // stored procedure name

    cmd.Parameters.AddWithValue("@owner", userName);

    cmd.Parameters.AddWithValue("@itemname", itemName);

    SqlDataReader reader = cmd.ExecuteReader();

    // ... result set handling

}

catch (SQLException se)

{

    // ... logging and error handling

}
```

SQL Server 存储过程：

```
CREATE PROCEDURE sp_queryItem

    @userName varchar(50),

    @itemName varchar(50)

AS

BEGIN

    SELECT * FROM t_item

    WHERE userName = @userName

    AND itemName = @itemName;

END

GO
```

存储过程使用参数化查询，而不包含不安全的动态 SQL 构建。数据库编译此存储过程时，会生成一个 SELECT 查询的执行计划，只允许原始的 SQL 语义被执行，任何参数值，即使是被注入的 SQL 语句也不会被执行。

错误示例（NHibernate: 动态构建 SQL/HQL）：

HQL 查询：

```
string userName = ctx.GetAuthenticatedUserName(); //this is a constant

string itemName = ***; // user input

ISession session = _sessionFactory.OpenSession();

IQuery hqlQuery = session.CreateQuery("from Item as item where item.owner = '"

                    + userName + "' and item.itemName = '" + itemName + "'");

var hrs = hqlQuery.List<Item>();
```

使用 NHibernate，如果在动态构建 SQL/HQL 查询时包含了不可信输入，同样也会面临 SQL/HQL 注入问题。

正确示例（NHibernate: 参数化查询）：

HQL 中基于位置的参数化查询：

```
string userName = ctx.GetAuthenticatedUserName(); //this is a constant

string itemName = ***; // user input

IQuery hqlQuery = session.CreateQuery("from Item as item where item.owner = ?

                                        and item.itemName = ?");

hqlQuery.SetParameter<string>(1, userName);

hqlQuery.SetParameter<string>(2, itemName);

var rs = hqlQuery.List<Item>();
```

HQL 中基于名称的参数化查询：

```
string userName = ctx.GetAuthenticatedUserName(); //this is a constant

string itemName = ***; // user input

Query hqlQuery = session.CreateQuery("from Item as item where item.owner = :owner

                                      and item.itemName = :itemName");

hqlQuery.SetParameter<string>("owner", userName);

hqlQuery.SetParameter<string>("itemName", itemName);

var rs = hqlQuery.List<Item>();
```

NHibernate 支持 SQL/HQL 参数化查询。为了防止 SQL 注入以及改善性能，以上这些示例使用参数化绑定的方式来设置查询参数。

正确与错误示例（iBATIS SQL 映射）：

iBATIS SQL 映射允许在 SQL 语句中通过#字符指定动态参数，例如：

```
<select id="getItems" parameterClass="MyClass" resultClass="Item">

    SELECT * FROM t_item WHERE owner = #userName# AND itemName = #itemName#

</select>
```

#符号括起来的 userName 和 itemName 两个参数指示 iBATIS 在创建参数化查询时将它们替换成占位符：

```
string sqlString = "SELECT * FROM t_item WHERE owner=@owner AND itemName =

                    @itemname";

SqlCommand cmd = new SqlCommand(sqlString, conn);

cmd.Parameters.AddWithValue("@owner", userName);

cmd.Parameters.AddWithValue("@itemname", itemName);

SqlDataReader reader = cmd.ExecuteReader();

                                // ... convert results set to Item objects
```

然而，iBATIS 也允许使用$符号指示使用某个参数来直接拼接 SQL 语句，这种做法是有 SQL 注入漏洞的：

```
<select id="getItems" parameterClass="MyClass" resultClass="items">

    SELECT * FROM t_item WHERE owner = #userName# AND itemName = '$itemName$'

</select>
```

在这里，攻击者可以利用 itemName 参数发起 SQL 注入攻击。
因此，在需要指定动态参数时，应该使用”#”符号而不是”$”符号。

```
<select id="getItems" parameterClass="MyClass" resultClass="items">

    SELECT * FROM t_item WHERE owner = #userName# AND itemName = '#itemName#'

</select>
```

虽然参数化查询是防止 SQL 注入最便捷有效的一种方式，但不是 SQL 语句中任何部分在执行前都能够被占位符所替代，因此，参数化查询无法应用于所有场景（有关 SQL 语句执行前可被占位符替代的元素，请参考相应数据库的帮助文档）。当使用执行前不可被占位符替代的不可信数据来动态构建 SQL 语句时，必须对不可信数据进行校验。每种 DBMS 都有其特定的转义机制，通过这种机制来告诉数据库此输入应该被当作数据，而不应该是代码逻辑。因此，只要输入数据被适当转义，就不会发生 SQL 注入问题。对于常见数据库中需要注意的特殊字符，请参考附录 A。

规则 1.3 禁止使用未经校验的不可信数据来拼接 XML

说明：使用未经校验的不可信数据来构造 XML 可能会导致 XML 注入漏洞。如果用户被允许输入结构化的 XML 片段，则他可以在 XML 的数据域中注入 XML 标签来改写目标 XML 文档的结构和内容，XML 解析器会对注入的标签进行识别和解释，引起注入问题。

错误示例（未检查的输入）：

```
private void CreateXMLStream(StreamWriter outStream, User user)
{
    string xmlString;
    xmlString = "<user><role>operator</role><id>" + user.UserId
                + "</id><description>" + user.Description
                + "</description></user>";
    outStream.Write(xmlString.toArray());
    outStream.Flush();
}
```

恶意用户可能会使用下面的字符串作为用户 id：

```
"joe</id><role>administrator</role><id>joe"
```

并使用如下输入作为描述字段：

```
"I want to be an administrator"
```

最终，整个 XML 字符串将变成如下形式：

```
<user>
    <role>operator</role>
    <id>joe</id>
    <role>administrator</role>
    <id>joe</id>
    <description>I want to be an administrator</description>
</user>
```

由于解析器在解释 XML 文档时会将第二个 role 域的值覆盖前一个 role 域的值，导致此用户角色由操作员提升为了管理员，权限大大提升。

错误示例（XML Schema 或者 DTD 校验 ）：

```
private void CreateXMLStream(StreamWriter outStream, User user)
```

```
{
    string xmlString;
    xmlString = "<user><id>" + user.UserId
            + "</id><role>operator</role><description>"
            + user.Description + "</description></user>";

    XmlSchemaSet xmlSchemaSet = new XmlSchemaSet();
    xmlSchemaSet.Add(string.Empty, "schema.xsd");
    XmlReaderSettings settings = new XmlReaderSettings();
    settings.Schemas.Add(xmlSchemaSet);
    settings.ValidationType = ValidationType.Schema;
    settings.ValidationEventHandler += (sender, e) =>
    {
        if (e.Severity == XmlSeverityType.Error)
        {
            throw e.Exception;
        }
    };
    XmlReader reader = XmlReader.Create(xmlString, settings);
    try
    {
        while (reader.Read())
        {
        }
    }
    finally
    {
        try
        {
            reader.Close();
        }
        catch (Exception e)
```

```
        {
            // handle exception
        }
    }
    // the XML is valid, proceed
    outStream.Write(xmlString.ToArray());
    outStream.Flush();
}
```

如下是 schema.xsd 文件中的 schema 定义：

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="user">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="role" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

恶意用户可能会使用下面的字符串作为用户 id：

```
"joe</id><role>Administrator</role><!—"
```

并使用如下字符串作为描述字段：

```
"--><description>I want to be an administrator"
```

最终，整个 XML 字符串将变成如下形式：

```
<user>
    <id>joe</id>
    <role>Administrator</role><!--</id> <role>operator</role> <description>
-->
    <description>I want to be an administrator</description>
</user>
```

用户 id 结尾处的“<!--”和描述字段开头处的“-->”将会注释掉原本硬编码在 XML 字符串中的角色信息。虽然用户角色已经被攻击者篡改成管理员类型，但是整个 XML 字符串仍然可以通过 schema 的校验。XML schema 或者 DTD 校验仅能确保 XML 的格式是有效的，但是攻击者可以在不破坏原有 XML 格式的情况下，对 XML 内容进行篡改。

正确示例（白名单校验）：

```
private void CreateXMLStream(StreamWriter outStream, User user)
{
    /** Write XML string if userID contains alphanumeric and underscore characters only */
    if (!Regex.IsMatch(user.UserId, "[_a-zA-Z0-9]+"))
    {
        // Handle format violation
    }
    if (!Regex.IsMatch(user.Description, "[_a-zA-Z0-9]+"))
    {
        // Handle format violation
    }
    string xmlString = "<user><id>" + user.UserId
            + "</id><role>operator</role><description>"
            + user.Description + "</description></user>";
    outStream.Write(xmlString.ToArray());
    outStream.Flush();
}
```

使用白名单方式对输入进行校验，要求输入的 userId 字段中只能包含字母、数字或者下划线。

正确示例（使用安全的 XML 库）：

```
public static void BuidlXML(StreamWriter stream, User user)
{
    XmlDocument userDoc = new XmlDocument();
    XmlElement userElem = userDoc.CreateElement("user");
    userDoc.AppendChild(userElem);
```

```
    XmlElement idElem = userDoc.CreateElement("id");
    userElem.AppendChild(idElem);
    idElem.InnerText = user.UserId;
    XmlElement roleElem = userDoc.CreateElement("role");
    userElem.AppendChild(roleElem);
    roleElem.InnerText = "operator";
    XmlElement descrElem = userDoc.CreateElement("description");
    userElem.AppendChild(descrElem);
    descrElem.InnerText = user.Description;
    XmlWriter output = new XmlTextWriter(stream);
    try
    {
        userDoc.WriteTo(output);
    }
    finally
    {
        try
        {
            output.Close();
        }
        catch (Exception e)
        {
            // handle exception
        }
    }
}
```

正确示例中使用 XmlDocument 来构建 XML。XmlDocument 将会对文本数据域进行 XML 编码，从而使得 XML 的原始结构和格式免受破坏。若攻击者输入如下字符串作为用户 id：

```
"joe</id><role>Administrator</role><!—"
```

使用如下字符串作为描述字段：

```
"--><description>I want to be an administrator"
```

则最终生成的 XML 为：

```
<user>

  <id>joe&lt;/id&gt;&lt;role&gt;Administrator&lt;/role&gt;&lt;!—</id>

  <role>operator</role>

  <description>--&gt;&lt;description&gt;I want to be an
administrator</description>

</user>
```

可以看到，“<”与“>”经过 XML 编码被分别替换成了“&lt”与“&gt”，使攻击者不能将角色类型从操作员提升到管理员。

规则 1.4 禁止将未经校验的不可信数据直接记录到日志中

说明：如果在记录的日志中包含未经校验的不可信数据，则可能导致日志注入漏洞。恶意用户会插入伪造的日志数据，从而让系统管理员误以为这些日志数据是由系统记录的。例如，用户可能通过输入一个回车符或一个换行符（CRLF）来将一条合法日志拆分成两条日志，使得日志内容可能令人误解。将未经净化的用户输入写入日志可能会导致敏感数据泄露，若在日志中写入和存储了某些类型的敏感数据甚至可能违反当地法律法规。例如，如果用户要把一个未经加密的信用卡号插入到日志文件中，那么系统就会违反了 PCI DSS（Payment Card Industry Data Security Standard）标准。

可以对记录到日志中的任何不可信数据进行校验和净化来防止日志注入攻击。

错误示例：

```
if (loginSuccessful)

{

    Debug.Print ("User login succeeded for: " + username);

}

else

{

    Debug.Print ("User login failed for: " + username);

}
```

在接收到非法请求时，示例代码会记录用户的用户名，且没有进行任何净化处理，在这种情况下可能遭受日志注入攻击。

当 username 字段的值是 david 时，会生成一条标准的日志信息：

```
User login failed for: david
```

但是，如果记录日志时使用的 username 字段不是 david，而是如下所示的多行字符串：

```
david

User login succeeded for: administrator
```

那么日志中包含了以下可能引起误导的内容：

```
User login failed for: david

User login succeeded for: administrator
```

正确示例：

```
if (!Regex.IsMatch(username, "[A-Za-z0-9_]+"))
{
    // Unsanitized username
    Debug.Print ("User login failed for unauthorized user");
}
else if (loginSuccessful)
{
    Debug.Print ("User login succeeded for: " + username);
}
else
{
    Debug.Print ("User login failed for: " + username);
}
```

正确示例中会在登录之前对用户名输入进行净化，防止注入攻击。

规则 1.5 禁止直接使用不可信的数据来构造格式化字符串

说明：当转换参数与对应的格式符不匹配时，标准类库会抛出异常，这种方式可以减少被恶意攻击的机会。但如果代码中使用来源不可信的数据来构造格式化字符串时，仍有可能被攻击者利用，造成系统信息泄露或者拒绝服务。因此，不能直接将来自不可信源的字符串用于构造格式化字符串。

错误示例：

```
class Format
{
    static DateTime C = new DateTime(1995, 5, 23);
    public static void Print(params string[] args)
    {
        // args[0] is the credit card expiration date
        // args[0] may contain either {0:m} as malicious arguments
        // First argument prints 05 (May), second prints 23 (day)
        // and third prints 1995 (year)
        // Perform comparison with c, if it doesn't match print the following line
        Debug.Print(args[0] + " did not match! HINT: It was issued on {0:m} of some
            month", C);
    }
}
```

错误示例将信用卡的失效日期作为输入参数并将其用在格式字符串中，攻击者可以通过提供一段包含{0:m}的输入，来试探出用于与输入数据做对比验证的日期信息，引起信息泄露。

正确示例：

```
class Format
{
    static DateTime C = new DateTime(1995, 5, 23);
    public static void Print(params string[] args)
    {
        // args[0] is the credit card expiration date
        // Perform comparison with c,
        // if it doesn't match print the following line
        Debug.Print("{0} did not match! HINT: It was issued on {1:m} of some month",
            args[0], C);
    }
}
```

正确示例中不以用户输入来构造格式化字符串。

规则 1.6 禁止向 Process.Start() 方法传递不可信、未净化的数据

说明：使用了未经校验的不可信输入来执行系统命令或者外部程序时，就可能会导致命令和参数注入。

在 C#中，Process.Start()经常被用来调用新的进程，但只有在其中调用了命令解析器（例如，cmd.exe 或 /bin/sh）来执行命令或程序，才有可能导致注入。当通过 Process.Start()执行 bat 或者 sh 脚本时，将会自动调用命令解析器。例如 Process.Start("test.bat & notepad.exe")，bat 文件默认是由命令行解释器 cmd.exe 来解释执行的，后面的“&”会被当做命令分隔符处理，test.bat 和 notepad.exe 都会被执行。

而对于参数注入攻击，当参数中包含空格、双引号、以“-”或者“/”开头来指定开关参数时都有可能发生。

与命令和参数注入相关的特殊符号的详细信息，请参考附录 B。

错误示例：

```
class DirList
{
   public static void Main(string[] args)
   {
      if (args.Length == 0)
      {
         Console.WriteLine("No arguments");
         return;
      }
      try
      {
         Process.Start("cmd.exe /c dir " + args[0]);
         // ...
      }
      catch (Exception e)
      {
         // Handle errors
      }
   }
```

```
}
```

攻击者可以通过以下命令来利用这个漏洞程序：

```
DirList.exe "dummy & echo bad"
```

实际将会执行两个命令：

```
dir dummy

echo bad
```

前一条命令尝试罗列目录的内容，后一条命令打印信息到控制台。

正确示例（避免使用 Process.start()）：

```
class DirList

{

   public static void Main(string[] args)

   {

      if (args.Length == 0)

      {

         Console.WriteLine("No arguments");

         return;

      }

      try

      {

         DirectoryInfo dir = new DirectoryInfo(args[0]);

         if (!Validate(dir)) // the dir need to be validated

         {

            Console.WriteLine("An illegal directory");

         }

         else

         {

            foreach (FileInfo file in dir.GetFiles())

            {

               Console.WriteLine(file.Name);

            }

         }
```

```
        }
        catch (Exception e)
        {
            Console.WriteLine("An unexpected exception");
        }
    }
    // Other omitted code…-
}
```

如果可以使用标准的 API 替代运行系统命令来完成任务，则应该使用标准的 API，尽量避免使用 Process.start()。正确示例使用 DirectoryInfo.GetFiles()方法来列举目录下的内容，而不是调用 Process.Start()来运行一个外部进程，从而消除了发生命令注入与参数注入的可能。

正确示例（数据校验）：

```
//...
if (!Regex.IsMatch(dir, "[0-9A-Za-z@]+"))
{
    // Handle error
}
//...
```

如果无法避免使用 Process.Start()，则必须要对输入数据进行检查和净化，防止其中包含命令分隔符，管道，或者重定向操作符（“&&”，“&”，“||”，“|”，“>”，“>>”）等，以及参数开关符（“-”，“/”）。

规则 1.7 验证路径之前应该先对其进行标准化

说明：绝对路径或者相对路径中可能会包含如符号（软）链接（symbolic [soft] links）、硬链接（hard links）、快捷方式（shortcuts）、影子文件（shadows）、别名（aliases）和连接文件（junctions）等形式，在进行文件验证操作之前必须完整解析这些文件链接。路径中也可能会包含如下所示的文件名，使得验证变得困难：

“.”指目录本身。

在目录内，“..”指该目录的上一级目录。

除此之外，还有与特定操作系统和特定文件系统相关的命名约定，也会使验证变得困难。

同一个目录或者文件，可以通过多种路径名来引用它，对其进行标准化，可以使文件路径验证较为容易。

当试图限制用户只能访问某个特定目录中的文件时，或者当基于文件名或者路径名来做安全决策时，验证是必须的。攻击者可能会利用目录遍历（directory traversal）或者等价路径（path equivalence）漏洞的方式来绕过这些限制。目录遍历漏洞使得攻击者能够转移到一个特定目录进行 I/O 操作，等价路径漏洞使得攻击者可以使用与某个资源名不同但是等价的名称来绕过安全检查。

在程序获取一个文件标准路径与打开这个文件之间，会有一个固有的时间竞争窗口。在对标准化的文件路径进行验证时，文件系统可能被修改，标准化的路径名可能不再指向原始的有效文件。值得庆幸的是，可以使用标准化的路径名来判断引用的文件名是否在安全目录中，来消除条件竞争（更多信息请参考规则 3.3 避免在共享目录操作文件）。如果引用的文件是在一个安全目录之中，很明显攻击者无法篡改文件，也无法利用条件竞争。

错误示例：

```
static void Main(string[] args)
{
    if (args == null || args.Length < 1)
    {
       throw new ArgumentException();
    }
    string dirPath = System.AppDomain.CurrentDomain.BaseDirectory;
    string filePath = dirPath + args[0];
    if (!IsInSecureDir(filePath))
    {
        // Refer to Rule 3.3 for the details of IsInSecureDir()
        throw new ArgumentException();
    }
    if (!ValidatePath(filePath))
    {
         // Validation
         throw new ArgumentException();
    }
    /* …- */
```

```
}
```

直接使用用户输入的参数拼接文件路径，参数可能含有“../”序列，如果只检查 filePath 是否以 dirPath 作为路径前缀，存在目录遍历风险。

正确示例（Path.GetFullPath ()）：

```
static void Main(string[] args)
{
  if (args == null || args.Length < 1)
    {
      throw new ArgumentException();
    }
    string dirPath = System.AppDomain.CurrentDomain.BaseDirectory;
    string filePath = dirPath + args[0];
    filePath = System.IO.Path.GetFullPath(filePath);
    if (!IsInSecureDir(filePath))
    {
      // Refer to Rule 3.3 for the details of IsInSecureDir()
      throw new ArgumentException();
    }
    if (!ValidatePath(filePath))
    {
      // Validation
      throw new ArgumentException();
    }
    /* …- */
}
```

正确示例使用了 System.IO.Path.GetFullPath()方法，它能对所有别名、快捷方式以及符号链接进行解析。如包含“../”的特殊文件名，在验证之前会被转换成标准形式，攻击者将无法使用../序列来突破特定目录。

规则 1.8 从压缩文件流中安全地提取文件

说明：在解压文件时需要小心谨慎。有两个特别的问题需要避免：一个是解压出的标准化路径文件在解压目标目录之外，另一个是解压的文件消耗过多的系统资源。前一种情况，攻击者可以从 zip 文件中往用户可访问的任何目录写入任意的数据。后一种情况，在与输入数据所使用资源相比严重不成比例时，就可能产生拒绝服务。由于 Zip 算法有极高的压缩率，即使在解压如 ZIP、GIF、gzip 编码 HTTP 的小文件时，也可能会导致过度的资源消耗，导致 zip 炸弹（zip bomb）。

Zip 算法有非常高的压缩比。例如，一个由字符 a 和字符 b 交替出现的行构成的文件，压缩比可以达到 200：1。使用针对目标压缩算法的输入数据，或者使用更多的输入数据（不针对目标压缩算法的），或者使用其他的压缩方法，甚至可以达到更高的压缩比。

任何被提取条目的目标路径不在程序预期目录之内时（必须先对文件名进行标准化，参照规则 1.7 验证路径之前应该先将其标准化），要么拒绝将其提取出来，要么将其提取到一个安全的位置。Zip 文件中任何被提取条目，若解压之后的文件大小超过一定的限制时，必须拒绝将其解压。具体大小限制由平台的处理性能来决定。

错误示例：

```
// <summary>
//
// </summary>
// <param name="zipPath"></param>
// <param name="extractPath"></param>
public void UnZip(string zipPath, string extractPath)
    {
        using (ZipArchive archive = ZipFile.OpenRead(zipPath))
        {
            foreach (ZipArchiveEntry entry in archive.Entries)
            {
                entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
            }
        }
}
```

错误示例中，未对解压的文件名做验证，直接将文件名传递给 ZipFile.OpenRead 函数。它也未检查解压文件的资源消耗情况，允许运行到解压操作完成或者本地资源被耗尽。

错误示例（Length）：

```
public const int MAXFILESIZE= 0x6400000; // 100MB
// <summary>
//
// </summary>
// <param name="zipPath"></param>
// <param name="extractPath"></param>
public void UnZip(string zipPath, string extractPath)
{
    using (ZipArchive archive = ZipFile.OpenRead(zipPath))
    {
        foreach (ZipArchiveEntry entry in archive.Entries)
        {
            if (entry.Length > MAXFILESIZE)
            {
                throw new AggregateException("File to be unzipped is huge.");
            }
            if (entry.Length <= 0)
            {
                throw new AggregateException("File to be unzipped might be huge.");
            }
            entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
        }
    }
}
```

错误示例调用 ZipArchiveEntry 的 Length 属性来判断某个解压条目的大小，以试图解决之前的问题。但不幸的是，恶意攻击者可以伪造 ZIP 文件中用来描述解压条目大小的字段，因此，Length 属性的返回值是不可靠的，本地资源仍可能被过度消耗。

正确示例：

```
public const int MAXFILESIZE = 0x6400000; // max size of unzipped data, 100M
public const int MAXFILENUMBER = 1024;// max number of files
// <summary>
//
// </summary>
// <param name="zipPath"></param>
// <param name="extractPath"></param>
public void UnZip(string zipPath, string extractPath)
{
    using (ZipArchive archive = ZipFile.OpenRead(zipPath))
    {
        int entries = 0;
        long total = 0;
        foreach (ZipArchiveEntry entry in archive.Entries)
        {
            entries++;
            total += entry.Length;
            if (entries > MAXFILENUMBER)
            {
                throw new AggregateException("Too many files to unzip.");
            }
            if (total > MAXFILESIZE)
            {
                throw new AggregateException("File to be unzipped is huge.");
            }
            if (entry.Length <= 0)
            {
                throw new AggregateException("File to be unzipped might be
                    huge.");
            }
            string fileName = Path.Combine(extractPath, entry.FullName);
```

```
            fileName = Path.GetFullPath(fileName);
            if (!fileName.StartsWith(extractPath))
            {
                throw new AggregateException("File is outside extraction target
                    directory.");
            }
            entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
        }
    }
}
```

正确示例中，代码会在解压每个条目之前对其文件名进行校验。如果某个条目校验失败，整个解压过程都将会被终止。实际上也可以忽略这个条目，继续后面的解压过程，甚至也可以将这个条目解压到某个安全位置。除了校验文件名，while 循环会检查从 zip 存档文件中解压出来的每个文件条目的大小。如果一个文件条目太大，此例中是 100MB，则会抛出异常。最后，代码会计算从存档文件中解压出来的文件条目总数，如果超过 1024 个，则会抛出异常。

2. 异常行为

规则 2.1 不要抑制或者忽略已检查异常

说明：程序员常常会通过一个空的或者无意义的 catch 块来抑制捕获的已检查异常。每一个 catch 块必须确保只会在继续有效的情况下程序才会继续运行下去。所以，catch 块要么从异常情况中恢复，要么重新抛出适合当前 catch 块上下文的另一个异常，以允许最邻近的外层 try-catch 语句块来进行恢复工作。异常会打断应用原本预期的控制流程，如 try 块中位于异常发生点之后的任何表达式和语句都不会被执行。因此，异常必须被适当处理。许多抑制异常的理由都是不合理的，如当对客户端从潜在问题恢复过来不抱期望时，一种好的做法是让异常被广播出来，而不是去捕获和抑制这个异常。

错误示例：

```
//...
try
{
    //...
}
catch (IOException ioe)
{
    Console.Write(ioe.toString());
}
//...
```

错误示例中，catch 块只是简单地将异常堆栈轨迹打印出来。虽然打印异常的堆栈轨迹对于定位问题是有帮助的，但是最终的程序运行逻辑与抑制异常时的情况是等同的。注意，即使在发生异常时错误示例会打印一个堆栈轨迹，但是程序会继续运行，如同异常从未被抛出过。换句话说，除了 try 块中位于异常发生点之后的表达式和语句不会被执行之外，发生的异常不会影响程序的其他行为。

正确示例（交互）：

```
//...
bool validFlag = false;
do
{
```

```
    try
    {
        // If requested file does not exist, throws FileNotFoundException
        // If requested file exists, sets validFlag to true
        validFlag = true;
    }
    catch (FileNotFoundException e)
    {
        // Ask the user for a different file name
    }
} while (validFlag != true);
// Use the file
```

正确示例要求用户指定另外一个文件名来处理 FileNotFoundException 异常。为了遵循规则 2.2 禁止在异常中暴露敏感信息，一个用户只允许访问特定目录中的文件，防止往循环外抛出其他 IOException 异常泄露敏感文件系统信息。

正确示例（Exception Reporter）：

```
public interface Reporter
{
    void Report(Exception t);
}

public class ReporterImpl: Reporter
{
    public void Report(Exception t)
    {
        Console.WriteLine(t.toString());
    }
}

public class ExceptionReporter
{
    // Exception reporter that prints the exception
```

```
    // to the console (used as default)
    private static readonly Reporter printException = new ReporterImpl();

    // Stores the default reporter.
    // The default reporter can be changed by the user.
    private static Reporter default = printException;

    // Helps change the default reporter back to
    // PrintException in the future
    public static Reporter GetPrintException()
    {
        return printException;
    }

    public static Reporter GetExceptionReporter()
    {
        return default;
    }

    // May throw a SecurityException (which is unchecked)
    public static void SetExceptionReporter(Reporter reporter)
    {
        // Custom permission
        Evidence evidence = new Evidence();
        evidence.AddHostEvidence(new Zone(SecurityZone.Intranet));
        // Check whether the caller has appropriate permissions
        SecurityManager.GetStandardSandbox(evidence).Demand();
        // Change the default exception reporter
        default = reporter;
    }
}
```

如何恰当的报告异常情况依赖于具体的上下文。例如，对于 GUI 应用应该以图形界面的方式报告异常，如弹出一个错误对话框。对于大部分库中的类应该能够客观的决定该如何报告一个异常来保持模块化；它们不能依赖于 System、特定的 logger，或者窗口环境的可用性。因此，对于会报告异常的库类，应该指定一个此类用来报告异常的 API。正确示例中指定了一个包含 Report()方法的用来报告异常的接口，以及一个默认的异常 Reporter 类供库类使用。可以定义子类来覆盖这个异常 Reporter 类。

库中的类后续便可以在 catch 子句中使用这个异常 Reptorter 类：

```
try
{
    // ...
}
catch (IOException warning)
{
    ExceptionReporter.GetExceptionReporter().Report(warning);
    // Recover from the exception...
}
```

任何使用这个异常 Reporter 类的客户代码，只要具备所需的权限许可，就能够覆写这个 ExceptionReporter，使用一个 logger 或者提供一个对话框来报告异常。例如，下面示例使用 GUI 客户代码，要求使用对话框来报告异常：

```
public class ReporterImpl: Reporter
{
    public void Report(Exception t)
    {
        MessageBox.Show(t.toString());
    }
}

ExceptionReporter.SetExceptionReporter(new ReporterImpl());
```

正确示例（继承 Exception Reporter 并过滤敏感异常）：

```
class MyExceptionReporter : ExceptionReporter
{
```

```
    public static void Report(Exception t)
    {
        t = Filter(t);
        // Do any necessary user reporting (show dialog box or send to console)
    }
    public static Exception Filter(Exception t)
    {
        /* Sanitize sensitive data or replace sensitive exceptions with
           non-sensitive exceptions (whitelist)*/
        // Return non-sensitive exception
    }
}
```

出于安全原因（参考规则 2.2 禁止在异常中泄露敏感信息），有时候必须对用户隐藏异常。在这种情况下，一种可行的方式是继承 ExceptionReporter 类，并且在重写默认 Report()方法的基础上，增加一个 Filter()方法。

例外情况：

1) 在资源释放失败不会影响程序后续行为的情况下，释放资源时发生的异常可以被抑制。释放资源的例子包括关闭文件、网络套接字、线程等等。这些资源通常是在finally块内的try-catch块中被释放，并且在后续的程序运行中都不会再被使用。因此，除非资源被耗尽，否则不会有其他途径使得这些异常会影响程序后续的行为。在充分处理了资源耗尽问题的情况下，只需对异常进行净化和记录日志（以备日后改进）就足够了；在这种情况下没必要做其他额外的错误处理。
2) 如果在特定的抽象层次上不可能从异常情况中恢复过来，则在那个层级的代码就不用处理这个异常，而是应该抛出一个合适的异常，让更高层次的代码去捕获处理，并尝试恢复。对于这种情况，最通常的实现方法是省略掉catch语句块，允许异常被广播出去。

规则 2.2 禁止在异常中暴露敏感信息

说明：基于产品应用场景以及风险分析来确定敏感数据的范围。典型的敏感数据包括口令、银行账号、批量个人数据、用户通讯信息、密钥等。如果在传递异常时未对其中的敏感信息进行过滤常常会导致信息泄露，而这可能有助于攻击者尝试进一步的利用行为。攻击者可以构造恶意输入参数来暴露应用程序内部结构和机制。异常中的文本消息和异常类型都可能泄露敏感信息。例如，

FileNotFoundException 异常消息会揭示文件系统布局信息，而异常类型揭

示请求的文件不存在。因此，当异常会被传递到信任边界以外时，必须同时对敏感的异常消息和敏感的异常类型进行过滤。

错误示例（异常消息和类型中泄露敏感信息）：

```
class ExceptionExample : FileNotFoundException
{
    static void Main(string[] args)
    {
        //Windows stores a user's home directory path in %appdata%
        FileStream fs = File.Open(Environment.GetEnvironmentVariable("APPDATA")
                            + @"\a.txt", FileMode.Open);
    }
}
```

当请求的文件不存在时，FileStream 的构造器会抛出 FileNotFoundException 异常，使得攻击者可以重复传入伪造的路径名称来洞悉底层文件系统结构。

错误示例（封装后重新抛出敏感异常）：

```
try
{
    FileStream fs = File.Open(Environment.GetEnvironmentVariable("APPDATA") +
        args[0], FileMode.Open);
}
catch (FileNotFoundException e)
{
    //log the exception
    throw new IOException("unable to retrieve file",e);
}
```

即使用户无法访问到日志中记录的异常，但是原始异常仍然会有大量有用信息，攻击者可以借此来发现文件系统结构信息。

错误示例 （异常净化）：

```
class ExceptionExample
{
    static void Main(string[] args)
```

```
    {
        try
        {
            FileStream fs = File.Open(Environment.GetEnvironmentVariable
                                ("APPDATA") + args[0], FileMode.Open);
        }
        catch (FileNotFoundException e)
        {
            //log the exception
            throw new IOException();
        }
    }
}
```

相比前面几个错误示例，此示例抛出的异常中泄露的有用信息较少，但是它仍然会透露出指定的文件不可读。程序对于有效的文件路径和不存在的文件路径会有不同的反应，攻击者可以根据程序的行为推断出有关文件系统的敏感信息。未对用户输入做限制，使得系统面临暴力攻击的风险，攻击者可以多次传入所有可能的文件名进行查询来发现有效文件。如果传入一个文件名后程序返回一个净化后的异常，则表明该文件不存在，而如果不抛出异常则说明该文件是存在的。

正确示例（安全策略）：

```
class ExceptionExample
{
   public static void Main(string[] args)
    {
        string str = Environment.GetEnvironmentVariable("appdata") + args[0];
        try
        {
            if (!Path.GetFullPath(str).StartsWith("c:\\homepath"))
            {
                Console.WriteLine("Invalid file");
                Console.ReadKey();
                return;
```

```
            }
        }
        catch (IOException e)
        {
            Console.WriteLine("Invalid file");
            Console.ReadKey();
            return;
        }
        try
        {
            FileStream fs = new FileStream(str, FileMode.Open);
        }
        catch(FileNotFoundException e)
        {
            Console.WriteLine("Invalid file");
            Console.ReadKey();
            return;
        }
    }
}
```

正确示例中，规定用户只能打开 c:\homepath 目录下的文件，用户不能发现这个目录以外的任何信息。如果无法打开文件，或者文件不在合法的目录下，则会产生一条简洁的错误消息。任何 c:\homepath 目录以外的文件信息都被会隐蔽起来。

正确示例（限制输入）：

```
class ExceptionExample
{
   public static void Main(string[] args)
   {
        FileStream fs = null;
        try
        {
            switch (Int32.Parse(args[0]))
```

```
            {
                case 1:
                    fs = new FileStream("c:\\homepath\\file1", FileMode.Open);
                    break;
                case 2:
                    fs = new FileStream("c:\\homepath\\file2", FileMode.Open);
                    break;
                //...
                default:
                    Console.WriteLine("Invalid option");
                    break;
            }
        }
        catch(Exception t)
        {
            MyExceptionReporter.Report(t);// Sanitize any sensitive data
        }
        //...
    }
}
```

限制用户只能打开 c:\homepath\file1 与 c:\homepath\file2，同时它也会过滤在 catch 块中捕获的异常中的敏感信息。

规则 2.3 方法发生异常时要恢复到之前的对象状态

说明：通常对象应当（关键安全对象必须）维持一致的状态，即使在发生异常时。用来维持对象状态一致性的常用手段包括：

- 输入校验（如校验方法的调用参数）
- 调整逻辑顺序，使可能发生异常的代码在对象被修改之前执行
- 当业务操作失败时，进行回滚
- 对一个临时的副本对象执行必要操作，当成功完成这些操作后，再将更新提交到原始的对象
- 避免去修改对象状态

错误示例：

```
public class Dimensions
```

```
{
    private int length;
    private int width;
    private int height;
    public static readonly int PADDING = 2;
    public static readonly int MAX_DIMENSION = 10;
    public Dimensions(int length, int width, int height)
    {
        this.length = length;
        this.width = width;
        this.height = height;
    }
    public int GetVolumePackage(int weight)
    {
        length += PADDING;
        width += PADDING;
        height += PADDING;
        try
        {
            Validate(weight);
            int volume = length * width * height; // 12 * 12 * 12 = 1728
            // Revert
            length -= PADDING;
            width -= PADDING;
            height -= PADDING;
            return volume;
        }
        catch (Exception t)
        {
            MyExceptionReporter.Report(t); // Sanitize any sensitive data
            return -1; // Non-positive error code
        }
```

```
    }
    private void Validate(int weight)
    {
        // do some validation and may throw a exception
    }
    public static void Main(string[] args)
    {
        Dimensions d = new Dimensions(10, 10, 10);
        System.Console.WriteLine(d.GetVolumePackage(21)); // Prints -1 (error)
        // Prints 2744 instead of 1728
        System.Console.WriteLine(d.GetVolumePackage(19));
    }
}
```

在这个错误示例中，未有异常发生时，代码逻辑会恢复对象的原始状态。但是如果出现异常事件，则回滚代码不会被执行，从而导致后续的 GetVolumePackage() 调用不会返回正确的结果。

正确示例（回滚）：

```
// ...
}
catch (Exception t)
{
    MyExceptionReporter.Report(t); // Sanitize any sensitive data
    // Revert
    length -= PADDING;
    width -= PADDING;
    height -= PADDING;
    return -1;
}
```

在这个正确示例中，`GetVolumePackage()`方法的 catch 块会恢复对象状态。

正确示例（`finally` 子句）：

```
public int GetVolumePackage(int weight)
```

```
{
    length += PADDING;
    width += PADDING;
    height += PADDING;
    try
    {
        Validate(weight);
        int volume = length * width * height; // 12 * 12 * 12 = 1728
        return volume;
    }
    catch (Exception t)
    {
        MyExceptionReporter.Report(t); // Sanitize any sensitive data
        return -1; // Non-positive error code
    }
    finally
    {
        length -= PADDING;
        width -= PADDING;
        height -= PADDING; // Revert
    }
}
```

正确示例使用一个 finally 子句来执行回滚操作，以保证无论是否发生异常，都会进行回滚。

正确示例（输入校验）：

```
public int GetVolumePackage(int weight)
{
    try
    {
        Validate(weight); // Validate first
    }
    catch (Exception t)
```

```
    {
        MyExceptionReporter.Report(t); // Sanitize any sensitive data
        return -1;
    }
    length += PADDING;
    width += PADDING;
    height += PADDING;
    int volume = length * width * height;
    length -= PADDING;
    width -= PADDING;
    height -= PADDING;
    return volume;
}
```

正确示例在修改对象状态之前执行输入校验。注意，try 代码块中只包含可能会抛出异常的代码，而其他代码都被移到 try 块之外。

正确示例（未修改的对象）：

```
public int GetVolumePackage(int weight)
{
    try
    {
        Validate(weight);
    }
    catch (Exception t)
    {
        MyExceptionReporter.Report(t); // Sanitize any sensitive data
        return -1;
    }
    int volume = (length + PADDING) * (width + PADDING) * (height + PADDING);
    return volume;
}
```

正确示例避免了修改对象状态，从而使得对象状态总是一致的，因此没有必要再进行回滚操作。相比之前的解决方案，更推荐使用这种方式。但是对于一些复杂的代码，这种方式可能无法实行。

3. I/O 操作

规则 3.1 临时文件使用完毕应及时删除

说明：程序员经常会在全局可写的目录中创建临时文件。例如，Windows 系统下的 C:\TEMP 目录，这类目录中的文件可能会被定期清理，如每天晚上或者重启时。然而，如果文件未被安全地创建或者用完后还是可访问的，具备本地文件系统访问权限的攻击者便可以对共享目录中的文件进行恶意操作。删除已经不再需要的临时文件有助于对文件名和其他资源（如二级存储）进行回收利用。每一个程序在正常运行过程中都有责任确保已使用完毕的临时文件被删除。

注意：下面的示例代码已假设文件在创建时指定了合适的访问权限（遵循规则 3.2 在多用户系统中创建文件时指定合适的访问权限）以及被创建在安全目录中（遵循规则 3.3 避免在共享目录操作文件）。

错误示例：

```
public class TempFile
{
   public static void Main(string[] args)
   {
      string tempFile = "tempnam.tmp";
      FileInfo f = new FileInfo(tempFile);
      if (f.Exists())
      {
         Console.WriteLine("This file already exists");
         return;
      }
      FileStream fop = null;
      try
      {
         fop = f.OpenWrite();
         fop.WriteByte(1);
         // … use temp file
      }
      finally
```

```
        {
            if (fop != null)
            {
                try
                {
                    fop.Close();
                }
                catch (IOException x)
                {
                    // handle error
                }
            }
        }
    }
}
```

错误示例代码在运行结束时未将临时文件删除。

正确示例（DELETE_ON_CLOSE）：

```
public class TempFile
{
    public static void Main(string[] args)
    {
        try
        {
            using (FileStream stream = File.Create(Path.GetTempFileName(), 1024,
                                        FileOptions.DeleteOnClose))
            {
                // write to the file and use it
            }
            //… write log
        }
        catch (IOException x)
        {
```

```
            // Some other sort of failure, such as permissions.
            //… write log
        }
    }
}
```

正确示例创建临时文件时用到了 System.IO 的几个方法。Path.GetTempFileName() 方法会新建一个随机的文件名（文件名的构造方式由具体的实现所定义），文件使用 using 构造块来打开，不管是否有异常发生都会自动关闭文件，并且在打开文件时用到了 DeleteOnClose 选项，使得文件在关闭时会被自动删除。

正确示例（手动删除临时文件）：

```
public class TempFile
{
    public static void Main(string[] args)
    {
        string f = Path.GetTempFileName();
        FileStream fop = null;
        try
        {
            fop = File.OpenWrite(f);
            // write to the file and use it
        }
        finally
        {
            if (fop != null)
            {
                try
                {
                    fop.Close();
                }
                catch (IOException x)
                {
                    // handle error
```

```
                }
                try
                {
                    File.Delete(f);
                }
                catch (Exception x)
                {
                    // handle error
                }
            }
        }
    }
}
```

规则 3.2 在多用户系统中创建文件时指定合适的访问权限

说明：多用户系统中的文件通常归属于一个特定的用户，文件的属主能够指定系统中其他哪些用户能够访问该文件的内容。文件系统使用权限和授权模型来保护文件访问。当一个文件被创建时，文件访问许可规定了哪些用户可以访问或者操作这个文件。当一个程序在创建文件时没有对文件的权限做足够的限制，攻击者可能在程序修改此文件的访问权限之前对其进行读取或者修改。因此，一定要在创建文件时就为其指定访问权限，以防止未授权访问。

错误示例：

```
var file = File.Create("");
```

默认情况下，将向所有用户授予对新文件的完全读/写权限。

正确示例：

```
var fSecurity = new FileSecurity();
// Add the FileSystemAccessRule to the security settings.
fSecurity.AddAccessRule(new FileSystemAccessRule(@"DomainName\AccountName"
    ,FileSystemRights.ReadData, AccessControlType.Allow));
// Create file follow the access rule of fSecurity.
var file = File.Create("", 1024, FileOptions.None,fSecurity);
```

正确示例（加密文件）：

如果文件是加密的，且只能用与加密相同的帐户来解密。

```
// Create an array of bytes.

byte[] messageByte = Encoding.ASCII.GetBytes("Here is some data.");

// Specify an access control list (ACL)

FileSecurity fs = new FileSecurity();

fs.AddAccessRule(new FileSystemAccessRule(@"DOMAINNAME\AccountName",
FileSystemRights.ReadData,
AccessControlType.Allow));

// Create a file using the FileStream class.

FileStream fWrite = new FileStream("test.txt", FileMode.Create,
FileSystemRights.Modify, FileShare.None, 8, FileOptions.Encrypted, fs);

// Write the number of bytes to the file.

fWrite.WriteByte((byte)messageByte.Length);

// Write the bytes to the file.

fWrite.Write(messageByte, 0, messageByte.Length);

// Close the stream.

fWrite.Close();
```

例外情况：

如果文件是创建在一个安全目录中，而且该目录对于非受信用户是不可读的，那么允许以默认权限创建文件。例如，如果整个文件系统是可信的或者只有可信用户可以访问，就属于这种情况。关于安全目录的定义请参考规则 3.3 避免在共享目录操作文件。

规则 3.3 避免在共享目录操作文件

说明：多用户系统允许多个具有不同权限的用户共享一个文件系统。攻击者可以利用许多文件系统的特性和功能，包括文件链接、设备文件和共享文件访问，在不具备权限的情况下对文件进行越权访问，特别是在多个用户可以创建、移动、删除文件的共享目录中操作文件。当程序以较高权限运行时，可能被攻击者利用来提升自己的权限。为了防止漏洞，程序必须仅在安全目录中操作文件。如果对于某个特定用户，只有该用户与系统管理员可以在其中创建、移动、删除文件，那么这个目录就是安全的。

错误示例：

```
public void Read(string fileName)

{
```

```
    FileStream stream = null;
    try
    {
        stream = new FileStream(fileName, FileMode.Open);
        // ...
    }
    finally
    {
        try
        {
            if (stream != null)
            {
                stream.Close();
            }
        }
        catch (IOException x)
        {
            // handle error
        }
    }
}
```

错误示例中，攻击者可以指定一个锁定设备或者一个先入先出（FIFO）文件名，导致在打开文件时程序挂起。

正确示例：

下面的解决方式确保传入的文件目录是属于当前用户或者管理员所有，任何其他的用户不能对其进行写操作。

```
// <summary>
//
// </summary>
// <param name="fileName"></param>
// <returns></returns>
public void Read(string fileName)
```

```
{
    FileStream stream = null;
    try
    {
        stream = new FileStream(fileName, FileMode.Open);
        // ...
    }
    finally
    {
        try
        {
            if (stream != null)
            {
                stream.Close();
            }
        }
        catch (IOException ex)
        {
            // handle error
        }
    }
}
public void Create()
{
    string dirName = "TestDirectory";
    // Add the access control entry to the directory for CurrrentAccount.
    AddDirectorySecurity(dirName, @"china\MyAccount",
        FileSystemRights. FullControl, AccessControlType.Allow);
    // Add the access control entry to the directory for SystemAccount.
    AddDirectorySecurity(dirName, @"MYDOMAIN\SystemAccount",
        FileSystemRights.FullControl, AccessControlType.Allow);
}
```

```
// Adds an ACL entry on the specified directory for the specified account.
public static void AddDirectorySecurity(string fileName, string Account,
FileSystemRights Rights, AccessControlType ControlType)
{
    // Create a new DirectoryInfo object.
    DirectoryInfo dInfo = new DirectoryInfo(fileName);
    /* Get a DirectorySecurity object that represents the current security
        settings.*/
    DirectorySecurity dSecurity = dInfo.GetAccessControl();
    // Add the FileSystemAccessRule to the security settings.
    dSecurity.AddAccessRule(new FileSystemAccessRule(Account, Rights,
        ControlType));
   // Set the new access settings.
   dInfo.SetAccessControl(dSecurity);
}
```

例外情况：

对于运行在单用户系统或者没有共享目录的系统，或者不可能有文件系统漏洞的系统上的程序，则无需在操作文件之前确保其在安全目录之中。

4. 序列化安全

规则 4.1 必须对 ISerializable 接口的实现进行保护

说明： 如果某个类想要控制其序列化进程，那么就需要实现 ISerializable 接口；并且，若该接口未受到保护，则可能会漏露敏感信息。因此，实现 ISerializable 接口来提供自定义的序列化操作时，需要应用如下保护措施：

1）通过要求SecurityPermission以及SerializationFormatter权限，

或者确保方法输出不会泄露机密信息来保护GetObjectData()方法的安全。

例如：

```
[SecurityPermissionAttribute(SecurityAction.Demand,SerializationFormatter
    =true)]
public override void GetObjectData(SerializationInfo sInfo, StreamingContext
    context)
{
}
```

2) 应在用于序列化的特殊构造函数中进行完备的输入校验，同时将其声明为受保护的(protected)或者私有的(private)以防被恶意代码滥用。这样，攻击者就不能通过反序列化来修改对象实例。除了在类的构造函数中，在通过任何其他方式获取一个类的实例时也应当进行同样的安全校验与权限检查，例如通过某种类型的工厂显式或间接创建类的实例。

规则 4.2 禁止序列化未加密敏感数据

说明：虽然序列化可以将对象的状态保存为一个字节序列，之后通过反序列化将字节序列又能重新构造出原来的对象，但是它并没有提供一种机制来保证序列化数据的安全性。因此，敏感数据序列化之后是潜在对外暴露的，可访问序列化数据的攻击者可以借此获取敏感信息并确定对象的实现细节。永远不应该被序列化的敏感信息包括：密钥、数字证书、以及那些在序列化时引用敏感数据的类，防止敏感数据被无意识的序列化导致敏感信息泄露。另外，声明了可序列化标识对象的所有字段在序列化时都会被输出为字节序列，能够解析这些字节序列的代码可以获取到这些数据的值，而不依赖于该字段在类中的可访问性。因此，若其中某些字段包含敏感信息，则会造成敏感信息泄露。

错误示例：

```
[Serializable()]
```

```
public class TestSimpleObject
{
    public int member1;
    public string member2;
    public string member3;
    public double member4;
    // sensitive field.
    public string member5;
    public TestSimpleObject()
    {
        member1 = 11;
        member2 = "hello";
        member3 = "hello";
        member4 = 3.14159265;
        member5 = "hello world!";
    }
    public void Print()
    {
        Console.WriteLine("member1 = '{0}'", member1);
        Console.WriteLine("member2 = '{0}'", member2);
        Console.WriteLine("member3 = '{0}'", member3);
        Console.WriteLine("member4 = '{0}'", member4);
        Console.WriteLine("member5 = '{0}'", member5);
    }
}
public class Test
{
    public static void Main()
    {
        //Creates a new TestSimpleObject object.
        TestSimpleObject obj = new TestSimpleObject();
        Console.WriteLine("Before serialization the object contains: ");
```

```
        obj.Print();
        //Opens a file and serializes the object into it in binary format.
        Stream stream = null;
        SoapFormatter formatter = new SoapFormatter();
        //BinaryFormatter formatter = new BinaryFormatter();
        try
        {
            stream = File.Open("data.xml", FileMode.Create);
            formatter.Serialize(stream, obj);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Serialize error:" + ex.Message);
        }
        finally
        {
            try
            {
                if (stream != null)
                {
                    stream.Close();
                }
            }
            catch (Exception e)
            {
                //Handle Exception
            }
        }
        //Empties obj.
        obj = null;
        formatter = new SoapFormatter();
        //formatter = new BinaryFormatter();
```

```
        try
        {
            //Opens file "data.xml" and deserializes the object from it.
            stream = File.Open("data.xml", FileMode.Open);
            obj = (TestSimpleObject)formatter.Deserialize(stream);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Serialize error:" + ex.Message);
        }
        finally
        {
            try
            {
                if (stream != null)
                {
                    stream.Close();
                }
            }
            catch (Exception e)
            {
                //Handle Exception
            }
        }
        Console.WriteLine("");
        Console.WriteLine("After deserialization the object contains: ");
        obj.Print();
    }
}
```

示例代码中，假定某成员信息是敏感的，那么将其序列化到数据流中，使之面临敏感信息泄露以及被恶意篡改的风险。

正确示例（NonSerialized）：

```
[Serializable()]
public class TestSimpleObject
{
    public int member1;
    public string member2;
    public string member3;
    public double member4;
   // A field that is not serialized.
   [NonSerialized()]
    public string member5;
    public TestSimpleObject()
    {
        member1 = 11;
        member2 = "hello";
        member3 = "hello";
        member4 = 3.14159265;
        member5 = "hello world!";
    }
    public void Print() {
        Console.WriteLine("member1 = '{0}'", member1);
        Console.WriteLine("member2 = '{0}'", member2);
        Console.WriteLine("member3 = '{0}'", member3);
        Console.WriteLine("member4 = '{0}'", member4);
        Console.WriteLine("member5 = '{0}'", member5);
    }
}
```

在对某个包含敏感数据的类进行序列化时，程序必须确保敏感数据不被序列化，包括阻止包含敏感信息的数据成员被序列化，以及不可序列化或者敏感对象的引用被序列化。该示例将相关字段声明为 NonSerialized，从而使它们不包括在按照默认序列化机制应该被序列化的字段列表中，这样既避免了错误的序列化，又防止了敏感数据被意外序列化。

例外情况：

对于任何可能包含敏感数据的对象，尽可能使该对象不可序列化，如果它必须为可序列化的，请正确加密敏感数据。如果无法实现这一点，则应注意该数据会被公开给任何拥有序列化权限的代码，需确保不让任何恶意代码获得带有指定 SerializationFormatter 标志的 SecurityPermission 权限。在默认策略下，通过 Internet 下载的代码或 Intranet 代码不会授予该权限，只有本地计算机上的代码才被授予该权限。

5. 平台安全

规则 5.1 持有安全状态的对象不能传递给不受信任的代码

说明：某些对象自己保存有安全状态。不可信代码若能访问这些对象，将会获得超越自身权限的安全授权。

例如创建 userInfo.dat 文件的 FileStream 对象，在创建时要求具有 FileIOPermission 权限，如果将对象传递给没有 userInfo.dat 文件访问权限的恶意代码，则恶意代码就拥有了读写 userInfo.dat 文件的权限。因此，对于任何试图访问 userInfo.dat 文件的代码，需检查是否拥有 userInfo.dat 文件的 FileIOPermission 权限。

错误示例：

```
public class PersistentObjectProvider
{
    private FileStream userInfoFileStream;
    public FileStream UserInfoFileStream
    {
        get
        {
            if (userInfoFileStream == null)
            {
                userInfoFileStream = File.Open("userInfo.dat",
                    FileMode.OpenOrCreate);
            }
            return userInfoFileStream;
        }
    }
}
```

能够读写 userInfo.dat 文件的 FileStream 对象创建后未做任何安全检查，若被传递给不具有 userInfo.dat 文件权限的恶意代码，恶意代码可以任意读写 userInfo.dat 文件。

正确示例：

```
public FileStream UserInfoFileStream
{
```

```
    get
    {
        //demand the file permission
        FileIOPermission filePermission = new
            FileIOPermission(FileIOPermissionAccess.AllAccess, "userInfo.dat");
        filePermission.Demand();
        if (userInfoFileStream == null)
        {
            userInfoFileStream = File.Open("userInfo.dat",
                FileMode.OpenOrCreate);
        }
        return userInfoFileStream;
    }
}
```

在正确示例中，任何试图获取该 FileStream 对象的代码都需拥有 userInfo.dat 文件的 FileIOPermissionAccess.AllAccess 权限，防止不可信代码越权。

规则 5.2 禁止给仅执行非特权操作的代码签名

说明：在 C#中代码签名可以让代码获得更高的权限。许多安全策略允许被签名的代码执行更高权限的操作。代码签名被用来对代码的来源做认证以及验证代码的完整性。它依赖于认证机构（CA）来确认签名者的身份。用户通常将数字签名与代码的安全执行相关联，相信签名的代码不会带来危害。若签名的代码出现漏洞则会产生问题，因为许多系统被配置为固定的信任某些签名机构，如果系统下载了由这些机构签名的包含漏洞的代码，但是这些系统不会警告和通知其用户。攻击者可以向用户输送带有合法签名的漏洞代码，并利用这些漏洞代码。由于上述问题，对于那些不需要执行特权操作的代码不要对其进行签名，让它们运行在受限的沙箱（sandbox）里面。

规则 5.3 使用强名称签名对敏感操作进行访问控制

说明：敏感代码不适合被不受信任代码调用。然而在某些情况下，敏感代码必须为程序集外的特定程序提供接口，这就需要对这些敏感操作进行访问限制。例如，存在需要被所有的 DLL 调用的接口，它必须是公开的，但是又不希望它被随意调用而被客户访问和恶意代码利用，因此敏感操作不能声明为 Public（强名称签名

对 Public 操作无效）；这时可以使用强名称签名指定可访问的 DLL，以达到保护敏感信息的目的。

对敏感操作使用强名称签名的目的：

强名称程序集只能被强名称程序集访问。
强名称程序集可以确保名称的唯一性
强名称保护程序集的版本沿袭
强名称提供可靠的完整性检查

正确示例（强名称签名配置）：

```
[assembly: InternalsVisibleTo("StrongNameAccess,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100cb
e636861dfd3bf400d6e89d701c8d77411a1a53bbc4cc647c4a22bbdcc86fdf64425f9d3085d2
2876edbcfddb5d8c9cc0b78f64714497877d47b732a601b2886482ef9d2b049403b0926802ea
4a12b5a037ccbeec450ccc5afacc62e4329a700cdaca2ca079ee021e5f835ae81b45675696db
0384788371a2f5d7810b1f0af4")]
```

在强名称签名程序的配置文件中添加上述代码，来指定可访问强名称签名程序的应用程序。其中 StrongNameAccess 是命名空间名称，publicKey 是强名称签名程序使用的加密密钥。例如：在 VS 命令行输入 sn -k keyPair.snk 生成 keyPair.snk 密钥文件；然后设置程序属性->签名，在程序集签名上打钩添加 keyPair.snk 密钥文件即可完成强名称签名；最后通过 sn.exe 工具提取 PublicKey，配置上述代码，即可指定特定的访问程序。

正确示例（强名称签名访问）：

```
namespace StrongNameAccess
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Class1.GetName());
            Console.ReadKey();
        }
    }
}
```

上述代码是一个简单的强名称程序访问实现。其中，Class1 是不同名称空间的强名称类，其它程序不可访问；由于强名称程序在配置文件中指定 StrongNameAccess 可访问，因此，除了 StrongNameAccess，其它程序依旧不可访问。

规则 5.4 避免使用 SuppressUnmanagedCodeSecurityAttribute 特性

说明：当托管代码调入非托管代码（通过 PInvoke 或者 COM 互操作进入本机代码）时，均要求非托管代码权限以确保所有调用方都有允许调入的必要权限。通过应用 SuppressUnmanagedCodeSecurityAttribute 特性标记，可以在运行时取消这种要求，以提升调用性能。

但是，增加 SuppressUnmanagedCodeSecurityAttribute 特性标记，只会在实时编译时检查权限，后续将不进行权限检查。例如，如果函数 A 调用函数 B 并且函数 B 用 SuppressUnmanagedCodeSecurityAttribute 进行了标记，则在实时编译时将检查函数 A 的非托管代码权限，但随后在运行时不进行此检查。

例外情况：

非托管入口成为内部的或者以其他方式禁止其他访问非托管入口，确保代码不会避开安全检查调用非托管入口，非托管代码不涉及安全问题，则可以使用 SuppressUnmanagedCodeSecurityAttribute 来提升性能。

6. 其他

规则 6.1 不要使用公共只读数组来定义程序的边界行为或安全性

说明：某些 .NET Framework 类中提供包含平台特定边界参数的只读公共字段。例如，InvalidPathChars 字段是一个数组，该数组描述文件路径字符串中不允许使用的字符。在整个.NET Framework 中存在许多类似的字段。可以通过您的代码修改公共只读字段（如 InvalidPathChars）的值。不应使用这样的公共只读数组字段来定义应用程序的边界行为，否则，恶意代码可能会更改边界定义并以出乎意料的方式使用您的代码。在.NET Framework 2.0 版和更高版本中，应使用可返回新数组的方法，而不应使用公共数组字段。例如，您不应使用 InvalidPathChars 字段，而应使用 GetInvalidPathChars 方法。

错误示例：

```
public enum UserType
{
    Guest = 0, User = 1, Admin = 2
}
public enum Permission
{
    Read = 1, Write = 2, Execute = 4
}
class PermissionManager
{
    public readonly Permission[] Permissions; //public read-only array field
    public PermissionManager()
    {
        Permissions = new Permission[3];
        Permissions[(int)UserType.Guest] = Permission.Read;
        Permissions[(int)UserType.User] = Permission.Read | Permission.Write;
        Permissions[(int)UserType.Admin] = Permission.Read | Permission.Write |
            Permission.Execute;
    }
}
```

```
class IOManager
{
    private PermissionManager m_manager;
    public IOManager()
    {
        m_manager = new PermissionManager();
    }
    public void WriteFile(UserType userType)
    {
        if ((m_manager.Permissions[(int)userType] & Permission.Write) !=
            Permission.Write)
        {
            Console.WriteLine("No permission");
            return;
        }
        Console.WriteLine("Write file completed");
    }
}
static void Main(string[] args)
{
    var mgr = new IOManager();
    mgr.WriteFile(UserType.Guest);
    mgr.WriteFile(UserType.Admin);
}
```

上面的例子中使用了公用只读数组 Permissions 来储存不同角色的权限，恶意代码能通过修改该数组的内容来欺骗系统，例如把 Admin 的权限赋给 Guest：

```
manager.Permissions[(int)UserType.Guest]=m_manager.Permissions[(int)UserType
.Admin];
```

正确示例：

```
class PermissionManager
{
    private readonly Permission[] m_permissions;
```

```
    public Permission[] GetPermissions()
    {
        return m_permissions.ToArray();
    }
}

class IOManager
{
    //...
    public void WriteFile(UserType userType)
    {
        if ((m_manager.GetPermissions()[(int)userType] & Permission.Write)
            != Permission.Write)
        {
            //...
        }
    }
}
```

使用 GetPermissions()方法替换原来的公共只读数组，方法每次返回一个新数组，从而避免外部代码修改原始数组的内容。

规则 6.2 禁止在日志中保存口令、密钥和其他敏感数据

说明：在日志中不能输出口令、密钥和其他敏感信息，口令包括明文口令和密文口令。对于敏感信息建议采取以下方法：

- 不在日志中打印敏感信息。
- 若因为特殊原因必须要打印日志，则用固定长度的星号（*）代替输出的敏感信息。

规则 6.3 禁止将敏感信息硬编码在程序中

说明：如果将敏感信息（包括口令和加密密钥）硬编码在程序中，可能会将敏感信息暴露给攻击者。任何能够访问到编译文件的人都可以反编译这些文件并发现这些敏感信息。因此，不能将信息硬编码在程序中。同时，硬编码敏感信息会增

加代码管理和维护的难度。例如，在一个已经部署的程序中修改一个硬编码的口令需要发布一个补丁才能实现。

错误示例：

```
class IPAddress
{
   private string ipAddress = "172.16.254.1";
   public static void Main(string[] args)
   {
      //...
   }
}
```

恶意用户可以使用反编译工具反编译 dll 文件来发现其中硬编码的服务器 IP 地址。反编译器的输出信息透露了服务器的明文 IP 地址：172.16.254.1。

正确示例：

```
class IPAddress
{
   public static void Main(string[] args)
   {
      char[] ipAddress = new char[30];
      StreamReader sr = new StreamReader("serveripaddress.txt");
      // Reads the server IP address into the char array,
      // returns the number of bytes read
      int n = sr.Read(ipAddress,0,30);
      // Validate server IP address
      // Manually clear out the server IP address
      // immediately after
      for (int i = n - 1; i >= 0; i--)
      {
          ipAddress[i] = '0';
      }
      sr.Close();
```

```
    }
}
```

正确示例从一个安全目录下的外部文件获取服务器 IP 地址。并在其使用完后立即从内存中将其清除，防止后续的信息泄露。

规则 6.4 使用强随机数

说明：伪随机数生成器（PRNG）使用确定性数学算法来产生具有良好统计属性的数字序列。但是这种数字序列并不具有真正的随机特性。伪随机数生成器通常以一个算术种子值为起始。算法使用该种子值生成一个输出以及一个新的种子，这个种子又被用来生成下一个随机值，以此类推。

.Net 提供了伪随机数生成器（PRNG）—— System.Random 类。这个伪随机数生成器具有可移植性和可重复性。因此，如果两个 Random 类的实例创建时使用的是相同的种子值，那么对于所有的实现，它们将生成相同的数字序列。在系统重启或应用程序初始化时，Seed 值总是被重复使用。在一些其他情况下，seed 值来自系统时钟的当前时间。攻击者可以在系统的一些安全脆弱点上监听，并构建相应的查询表预测将要使用的 seed 值。

因此，Random 类不能用于安全敏感应用或者敏感数据保护。应使用更加安全的随机数生成器，例如

`System.Security.Cryptography.RandomNumberGenerator` 类。

正确示例：

```
public byte[] GenRandBytes(int len)
{
    byte[] bytes = null;
    if (len > 0 && len < 1024)
    {
        bytes = new byte[len];
        RandomNumberGenerator random = RandomNumberGenerator.Create();
        random.GetBytes(bytes);
    }
    return bytes;
}
```

规则 6.5 生产环境中部署的代码不能包含任何调试入口点

说明：一种常见的做法就是由于调试或者测试目的在代码中添加特定的后门代码，这些代码并没有打算与应用一起交付或者部署。当这类的调试代码不小心被留在了应用中，这个应用对某些无意的交互就是开放的。这些后门入口点可以导致安全风险，因为在设计和测试的时候并没有考虑到而且处于应用预期的运行情况之外。

被忘记的调试代码最常见的例子比如一个 web 应用中出现的 Main()方法。虽然这在产品生产的过程中也是可以接受的，但是在生产环境下，ASP.NET 应用中的类是不应该定义有 Main()的。

错误示例：

```
class Stuff
{
    // other fields and methods
    public static void Main(string[] args)
    {
        Stuff stuff = new Stuff();
        // Test stuff
    }
}
```

错误示例中，Stuff 类使用了一个 Main()函数来测试其方法。尽管对于调试是很有用的，如果这个函数被留在了生产代码中（例如，一个 Web 应用），那么攻击者就可能直接调用 Stuff.Main()来访问 Stuff 类的测试方法。

正确示例：

正确的代码示例中将 Main()方法从 Stuff 类中移除，这样攻击者就不能利用这个入口点了。

参考资料

1. MSDN Secure Coding Guideline,
http://msdn.microsoft.com/en-us/library/d55zzx87(v=vs.90).aspx
2. CWE&SANS TOP 25, http://www.sans.org/top25-software-errors/
3. OWASP Guide Project, https://www.owasp.org/index.php/OWASP_Guide_Project

附录 A

下表中总结了常用数据库中与 SQL 注入攻击相关的特殊字符：

数据库	特殊字符	描述	转义序列
Oracle	%	百分比：任何包括 0 或更多字符的字符串	/% escape '/'
	_	下划线：任意单个字符	/_ escape '/'
	/	斜线：转义字符	// escape '/'
	'	单引号	''
MySQL	'	单引号	\'
	"	双引号	\"
	\	反斜杠	\\
	%	百分比：任何包括 0 或更多字符的字符串	\%
	_	下划线：任意单个字符	_
DB2	'	单引号	''
	;	冒号	.
SQL Server	'	单引号	''
	[	左中括号：转义字符	[[]
	_	下划线：任意单个字符	[_]
	%	百分比：任何包括 0 或更多字符的字符串	[%]

	^	插入符号：不包括以下字符	[^]

附录 B

下表中总结了 shell 脚本中常用的与命令注入相关的特殊字符：

类型	举例	常见注入模式和结果
管道	|	| shell_command -执行命令并返回命令输出信息
内联	;	; shell_command -执行命令并返回命令输出信息
	&	& shell_command -执行命令并返回命令输出信息
逻辑运算符	$	$(shell_command) -执行命令
	&&	&& shell_command -执行命令并返回命令输出信息
	||	|| shell_command -执行命令并返回命令输出信息
重定向运算符	>	> target_file -使用前面命令的输出信息写入目标文件
	>>	>> target_file -将前面命令的输出信息附加到目标文件
	<	< target_file -将目标文件的内容发送到前面的命令

附录 C

下表列举了一些常见的敏感异常：

异常名称	信息泄露或威胁描述
System.IO.FileNotFoundException	泄露文件系统结构和文件名列举。
System.ContextMarshalException	敏感封装内容泄露
System.BadImageFormatException	图像资源列举
System.MemberAccessException	类成员列举
System.NotImplementedException	类方法列举
System.Threading.SynchronizationLockException	不安全线程
System.Xml.Schema.XmlSchemaException	泄露 XML 构架信息
System.OutOfMemoryException	DoS

System.StackOverflowException	DoS
System.Data.SqlClient.SqlException	数据库结构，用户名列举