

## 41 | 预案，代码的主动风险管理

2019-04-08 范学雷



上一次，我们聊了保持代码长治久安的基础——代码规范。这一次，我们接着聊第二个方面，代码的风险预案。

有些问题，并没有适用于各种场景的解决办法；有些设计，并不能适用于所有的用户；有些实现，并不能经受过去、现在和未来的检验。在你的日常工作中，有没有这样的情况出现？

做好预案，是我们管理风险的一个很重要的手段。代码的安全管理，也需要预案。

### 评审案例

让我们一起来看一段节选的Java代码变更。

```

public static String[] getDefaultCipherSuites() {
-   int ssl_ctx = SSL_CTX_new();
-   String[] supportedCiphers = SSL_CTX_get_ciphers(ssl_ctx);
-   SSL_CTX_free(ssl_ctx);
-   return supportedCiphers;
+   return new String[] {
+       "SSL_RSA_WITH_RC4_128_MD5",
+       "SSL_RSA_WITH_RC4_128_SHA",
+       "TLS_RSA_WITH_AES_128_CBC_SHA",
+       // snipped
+   };
}

```

对于这段代码，我先做一些说明。其中，“**Cipher Suites**”指的是我们在前面一篇文章中提到的**TLS**协议的密码算法族，“**SSL\_RSA\_WITH\_RC4\_128\_MD5**”是一种基于**RC4**加密技术的算法族，“**TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA**”是一种基于**CBC**（**Cipher Block Chaining**，链式加密）模式的算法族。

**getDefaultCipherSuites()**这个方法返回值的顺序，就是**TLS**协议使用这些算法的优先级别。比如，变更后的代码，“**SSL\_RSA\_WITH\_RC4\_128\_MD5**”算法族具有最高的优先级。相应地，“**SSL\_RSA\_WITH\_RC4\_128\_SHA**”具有第二优先级。在安全传输的连接中，优先级靠前的算法会获得优先考虑。一旦优先的算法被采用，其他的算法就不会被使用了。

这段代码是**Android**系统的一部分。这个修改发生在**2010年5月份**，这样做是为了使用**Android**偏爱的**RC4**加密算法。有了这样的变更，**Android**就能对算法的选择有更好的安排与控制。

想想上一篇文章中我们说到的**BEAST**攻击，这个修改是不是很有前瞻性？**BEAST**攻击技术是在**2011年9月份**公布的，有缺陷的算法是基于**CBC**模式的算法。**Android**提早了一年把涉及问题的**CBC**模式设为次优选择。**Chrome**浏览器可能更早做了类似的修改。所以，当**BEAST**攻击技术公开后，**Google**可以很自豪地说：“我们很早就优先使用更安全的**RC4**算法啦。”

可是，这个变更，还是有一点小问题的。

## 案例分析

要想看清楚这个问题，我们还需要讲述一段小插曲。

在**1999年**设计**TLS 1.0**的时候，有两种常用的加密算法类型。一个是**分组加密技术**，把原数据分成若干的小块，然后一小块一小块地分组加密。**3DES**是二十世纪九十年代最流行的分组加密算法。另一个是**流加密技术**，这种加密方式是把原数据一位一位地运算。**RC4**是二十世纪九十年

代最流行的流加密算法。对这两种算法，**TLS 1.0**都是支持的。其中的分组加密算法，**TLS 1.0**采用的是链式加密模式。

**2011年9月25日**，**BEAST**攻击技术公开发表。通过上一篇的介绍我们都知道，**BEAST**攻击技术针对的就是链式加密模式，链式加密模式不再安全了。你有没有惊喜地发现，**TLS 1.0**的设计真是周到，居然还有一个流加密技术可以使用，而且**RC4**算法被广泛支持。这真是一个可以救命的设计。

如果你回看**2011年**、**2012年**的安全分析文章，很多业界的专家都会推荐使用**RC4**来替代链式加密模式，很多产品也开始变更为优先使用**RC4**算法。毕竟，**BEAST**攻击是一个不可忽视的安全问题，而针对**BEAST**攻击的补救措施并不是一个完美的解决方案。在业界寻找链式加密模式的替代算法的同时，优先使用**RC4**算法似乎可以让大家喘口气。

这的确是一个救命的设计，但是，这是一个巧合的设计吗？如果身处**1999年**，我还没有足够的经验来判断这样的设计是有意为之，还是仅仅是一个巧合。但是，**20年**后的今天，如果我们的产品只支持一种模式的安全算法，我一定如坐针毡。因为我知道，**短则一两天，长则三五年**，一个算法的理论模型或者实现方式几乎一定会被破解。战战兢兢地等待着这个算法被破解，然后再去寻找补救的措施，显然不是一个可以让工程师心情愉悦、身心放松的好选择。

虽然优先使用**RC4**可以让业界稍作喘息，但是，好景并不长。**2013年3月13日**，一个研究小组公开了一个关于**RC4**算法的严重的安全漏洞。不同寻常的是，这一次并没有合适的修改**RC4**算法的补救措施。该研究小组建议，停止使用**RC4**，**TLS 1.0**和**1.1**版本的用户应该转化到**CBC**模式的加密算法。这算是一个不小的玩笑，很多应用刚从**CBC**模式切换到**RC4**算法不久，就要重新调整，再切换回去。

这就类似于两个病例。**CBC**模式虽然是一场大病，可是，有成熟的救治方案。虽然那里或者这里或许会留个疤，可是手术一旦实施成功，**CBC**模式照样活蹦乱跳。这就好比以前的**100米**需要跑**9.8秒**，手术后也可以跑个十一二秒的。虽然离巅峰阶段有点差距，但是问题不算大。

而**RC4**的问题，就像是医生诊断后，直接重症监护，并时刻准备后事了。冷酷而又无奈！**2013年3月13日**，**RC4**算法宣告重病缠身，重症监护。

随后，业界开始重新转换回**CBC**模式，很多应用开始禁用**RC4**算法。**2013年8月**，**IETF**提出了在**TLS**协议中禁用**RC4**算法的议案。**2015年2月**，该议案获得通过。**RC4**算法这个因高效、安全而著名的算法，从**2013年3月**开始，慢慢淡出人们的视野。

有了上面的小插曲，你知道上面案例代码的问题了吗？这段代码写死了**TLS**协议算法的缺省优先级别。除非更改代码，否则这个缺省优先级别是无法更改的。一旦优先的算法出了问题，代码修改虽然简单，但是已部署产品的升级，有时候就是一件很复杂的事情。

世事无常，一个好的设计，需要有双引擎和降落伞。

## 双引擎，长远之计

现代的客机，一般采用双引擎甚至多引擎设计。如果其中一个引擎失灵，依靠其他的引擎依然可以延程飞行。有人戏称延程飞行是一个“要么多引擎，要么去游泳”的设计理念。但是，延程飞行时间也是有约束的，比如不得超过**90**分钟。为什么呢？因为延程飞行时，就只有一个发动机在工作了。单引擎运转，总是有更大的安全隐患，这实在是让人不安！

需要注意的是，**双引擎不是备份计划，不是应急计划，不是Plan B，两个引擎日常都要使用**。如果其中一个引擎闲置，那么当真正需要它的时候，我们就不知道它的状态如何，是否可以承担重任。

想一想，为什么**CBC**模式出事的时候，业界可以切换到**RC4**算法？**RC4**算法出事的时候，业界可以切换回**CBC**模式？其中有很重要的两点值得考虑。

1. 无论是**CBC**模式，还是**RC4**算法，都是实际投入使用的算法。
2. 无论是**CBC**模式，还是**RC4**算法，都是大部分应用同时支持的算法。

这两条，对于**CBC**模式和**RC4**算法之间的成功的切换，都是必不可少的隐性条件。

如果我们理一理**TLS**协议发展的脉络，就随时可以看到双引擎设计的理念的运用。

1999年，**TLS 1.0**提供了**CBC**模式和**RC4**算法两种加密算法。随后，2003年，发现了**CBC**模式的安全问题。2006年发布的**TLS 1.1**在协议设计层面修复了**CBC**模式的潜在问题，提供了**CBC**模式和**RC4**算法两种加密算法。2008年发布的**TLS 1.2**添加了**AEAD**加密算法，加上已被修复的**CBC**模式和**RC4**算法，这样就有三种加密算法可供选择。2018年8月发布的**TLS 1.3**，废弃了**CBC**模式和**RC4**算法，只保留了**AEAD**算法，但是**AEAD**算法有两个推荐选项，分组密码的**GCM**模式，和流密码的**Chacha20/Poly1305**模式。到2018年8月，**TLS**协议在这二十年里，逐步废弃了二十年前最流行的算法。但是在整个过程中，一直保持多算法并存的设计。

如果你熟悉**JDK**的安全规范和实现，可能会注意到，对于每一个类型的算法，我们总是尽可能地提供多种选择。如果一个算法面临问题，我们总是尽快地替换旧算法，并且补充新的算法。这样，尽快地结束单算法的延程飞行状态。所以，提供多种选择，不仅仅是为了提升丰富性，也是为了在面临关键风险的时候，有风险控制的办法。

对于生死攸关的风险点，我们要有双引擎设计的意识。然而，也有双引擎解决不了的问题。即便是多引擎飞机，也需要备用降落伞。

## 降落伞，权宜之计

在上述案例的代码中，算法的缺省优先级别是固定的。一旦优先的算法出了问题，该怎么办？如果等到出了问题、蒙受了损失，再去寻找解决方案，就太晚了。一般情况下，一个好的软件应该备好降落伞，提前设计部署好这些意外风险的应急办法。我们永远不希望使用降落伞，但是如

果有意外发生，降落伞的存在就非常必要了。随时需要，随时就可以拿来使用。

以JDK为例，对于TLS协议的密码算法，一旦一个算法出现问题，修改源代码，替换掉出问题的算法是JDK提供的常规解决方案。另外，JDK还提供了多样的应急方案：

1. 修改JVM系统的安全参数（Security Property），降低出问题算法的优先级；
2. 修改JVM系统的安全参数（Security Property），废弃出问题算法；
3. 修改JVM系统的安全参数（Security Property），升级到没有问题TLS版本；
4. 修改应用的系统属性（System Property），使用指定的算法；
5. 修改应用的系统属性（System Property），升级到没有问题的TLS版本。

JVM系统的安全参数可以控制运营在JVM上的所有应用程序，而应用的系统属性一般只影响使用它的应用程序。

在JDK中，可以通过修改<java-home>/conf/security/java.security文件设置JVM系统的安全参数。比如，“jdk.tls.legacyAlgorithms”是一个设置TLS历史遗留算法的安全参数。一旦一个算法被设置为历史遗留算法，这个算法就不会被优先使用，除非不存在其他可替换的算法。如果我们把RC4算法设置为历史遗留算法，它的优先级就被降到最低，即使它的缺省优先级别是最高的。

```
jdk.tls.legacyAlgorithms = RC4_128
```

一旦在java.security文件中设置了这个参数，所有使用这个JDK配置的应用程序都会受到影响。

一个应用程序运行时，可以指定系统属性，比如：

```
$ java -Djdk.tls.client.protocols="TLSv1.3" myApp
```

那么这个应用程序就使用TLS 1.3版本的客户端。另外一个运行的程序，也可以使用TLS 1.2。两个运行程序的设置互不影响。

```
$ java -Djdk.tls.client.protocols="TLSv1.2" myApp
```

通过上面的例子，你可以看到，这些应急方案采用了配置参数的方式，使用非常简单，不需要运营代码的更改。简单、易用、快速上手，这是我们设计应急降落伞的一个思路。

一旦一个系统采纳了双引擎和降落伞的设计，系统的可靠性和抗风险能力往往会有大幅度提高。

可是，这并不是白白得来的。它同时也意味着软件研发的巨大投入，和软件复杂度的显著提升。

我们总是尽最大的可能使得软件程序简化、简化再简化。可是对于生死攸关的风险点，我们有时需要选择相反的方向，强化、强化再强化。不是所有的复杂都是必要的，也不是所有的复杂都是不必要的。软件的设计，是一个需要反复权衡、反复妥协的艺术。

## 小结

通过对这个评审案例的讨论，我想和你分享下面两点个人看法。

1. 尽管我们无法预料未来可能出现的风险，但是软件的设计和实现依然要考虑抗风险的能力。
2. 对于生死攸关的风险点，我们既要有长期的双引擎设计的意识，也要有权宜的应急预案；

如果深入到软件的架构和设计里，双引擎和降落伞的使用随处可见，你愿意分享你见到过的双引擎和降落伞的案例吗？欢迎在留言区留言。

## 一起来动手

这不算是个练习，而是一个请求。如果你有时间，你能够研究下你使用的语言、架构或者应用，找找其中的风险防范设计吗？代码安全和风险控制，是一个需要超大范围合作的技术领域。我们也需要共同创作这一话题，共同学习其中的经验。

比如说，我有个疑问就是，很多业务需要手机验证码，当我的手机不能使用时，我还有没有办法操作我的银行账户？

如果你觉得这篇文章有所帮助，欢迎点击“请朋友读”，把它分享给你的朋友或者同事。

# 代码精进之路

你写的每一行代码都是你的名片

范学雷

Oracle 首席软件工程师  
Java SE 安全组成员  
OpenJDK 评审成员



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言



我来也

👍 0

像我公司的业务——休闲网络游戏，就有设计备用入口。

因为现在域名劫持，用户所在的小运营商网络，某个服务器被攻击，都会导致部分用户无法正常连上服务器。

我们会在客户端中预留一些连接入口，在极端情况下尝试去实用备用线路。

以前也会有选择最优线路的方案。

2019-04-09

作者回复

谢谢提供这么好的一个用例。

2019-04-10



hua168

👍 0

如果是我们写代码，我们都基本用框架（如spring）去实现的，不用双引擎和降落伞吧，像我们日常web开发，什么情况下才会用这些。

2019-04-08

作者回复

框架当然不能解决所有的安全问题，什么样预案，依赖具体的应用。这没有什么统一的答案。

虽然我知道有一个统一的答案该有多好，但是就是没有统一的答案，因为没有统一的场景。

2019-04-09