

10 | 弱随机数生成器：攻击者如何预测随机数？

2022-01-07 王昊天

《Web漏洞挖掘实战》



你好，我是王昊天。

上节课我们学习了密码算法的安全，有了设计优秀的密码算法，就像买了一扇牢固的防盗门，那么我们再也不用担心小偷了吗？

并不是这样，有了防盗门，我们还需要保管好钥匙才行。上节课我们一直在讨论防盗门的质量问题，而防盗门的钥匙，也就是密码算法所使用的密钥是如何生成的呢？这里就需要一个引入新的概念——随机数。

随机数的概念是很好理解的，但是实际操作起来却很难真的生成。

你一定玩过某一种棋牌类游戏，比如麻将、德州扑克这些，或者更简单的猜拳游戏也可以。考虑到每个人的游戏水平有高低，胜率一定会有些差异。在经常一起玩的朋友当中，一定有某个人在的时候你更容易赢，另外某个人在的时候你更容易输，而且这种输赢是具有统计意义上稳定性的，为什么呢？

因为在面对没有差异的选择时，你是有选择倾向性的，这种倾向性可能来自于你的回忆、你的幸运数字、你的生日等等。

所以，我们以为的随机数，往往没有那么随机。

随机数

我们来正式地认识一下随机数，这一概念在不同领域往往代表着不同的含义。我们一起来由浅入深地聊聊。

首先随机数最基本的概念是**统计学意义上的伪随机数**，对于给定的一个样本集，每个元素出现的概率是大概相似的，只要从人类的视角看上去一组数是随机的，就符合统计学意义上的伪随机数定义；因为统计学上的伪随机数，在给定随机样本和随机算法的情况下，能够有效地演算出随机样本的剩余部分，因此统计学上的伪随机数需要得到进一步地安全强化，**密码学安全的伪随机数**应运而生；而随机数的最终概念形态，则是**真随机数**，其定义是在满足前两个条件的基础上，再增加一个随机样本不可重现的条件。

然而，严格的真随机数是一种非常理想的形态，从真实情况来看，只要给定边界条件，真随机数其实并不存在。因为无论背景辐射、物理噪音还是抛掷硬币，只要经过非常精密的观察和测量，都是可以被预测的。但是在这些例子中，实际的边界条件非常复杂，而且是极难观测的，因此我们可以认为这些条件下产生的随机数是非常接近真随机数的伪随机数。

那么为什么随机数的随机性如此重要呢？因为在我们前一节课程中探讨过的密码算法需要大量随机数的参与，一旦随机数的生成可以被预测，任何加密算法都将失去意义。

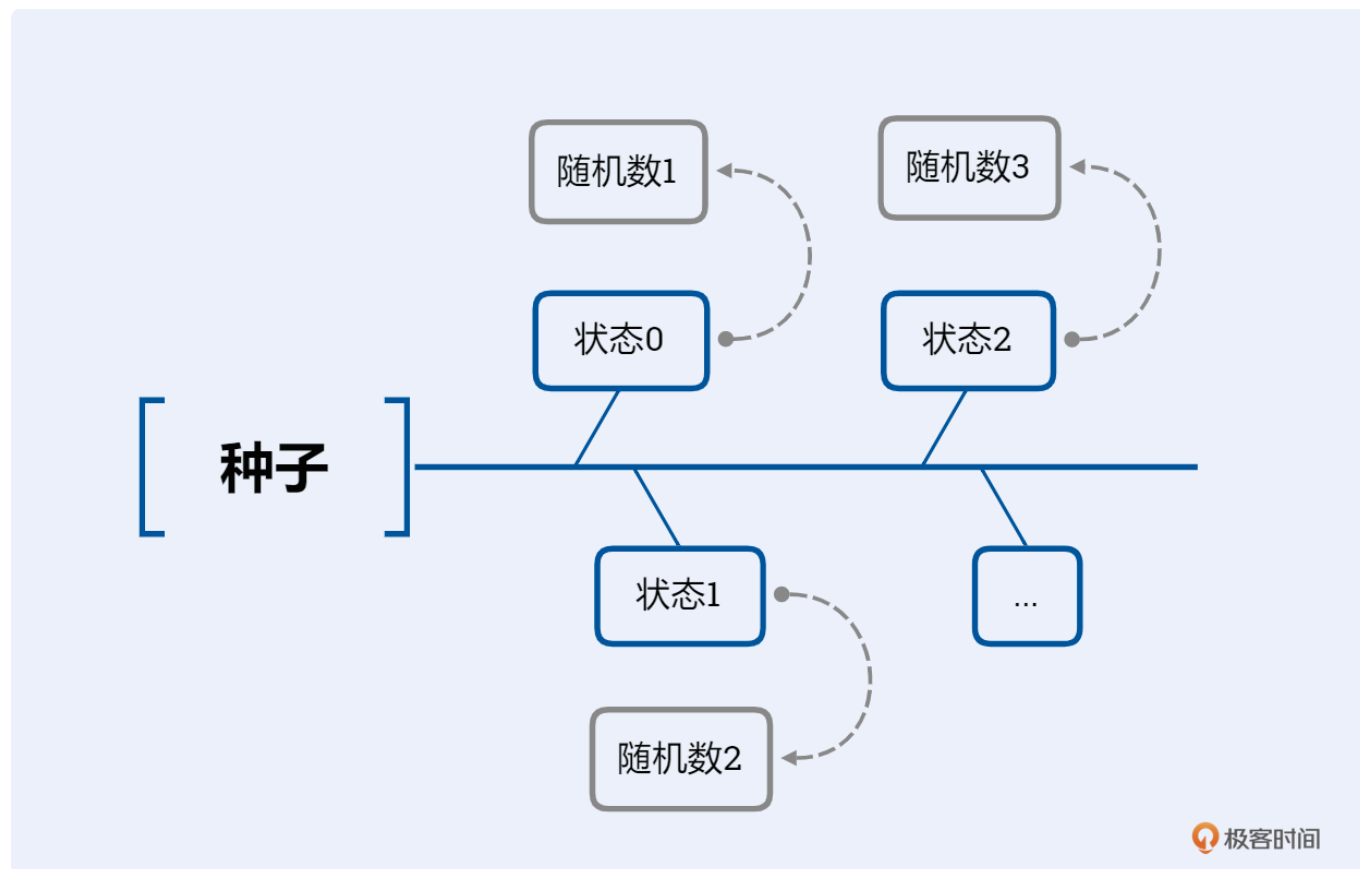
随机数的生成

产生随机数的方法被称为随机数生成器（RNG, random number generator）。

在实际应用中我们往往使用伪随机就足够了，这些随机数主要通过一个固定的、可重复的计算方法生成，这些计算方法经过特殊的设计，使得产生的结果具有类似真随机数的统计学特征。这种生成的伪随机数一般只是重复的周期比较大的数列，以算法和种子值共同作用生成。这种生成伪随机数的方法叫伪随机数生成器（PRNG, pseudo-random number

generator)，进一步能够生成密码学安全随机数的方法叫密码学伪随机数生成器（CPRNG, cryptographic pseudo-random number generator）。

从实现的角度来看，伪随机数生成器会在函数内部维护一个状态，每个随机数的诞生，时都是从这个状态计算出来的，这个状态随着下一个随机数的生成而改变，而第一个状态则是由种子初始化得到。



在一些密码学关键设施中，会使用到真正的随机数，这些随机数往往由噪音、辐射等物理现象生成，这个过程中所使用的生成器叫物理性随机数生成器。

在了解随机数概念及其生成方案后，接下来我们来了解一下开发过程中会面临哪些随机数方面的安全风险。

无效的随机数

当我们在开发应用的过程中，如果所使用的随机数算法不够安全，比如使用了 PRNG 而并没有使用 CPRNG，或者使用了不安全的种子值，就可能会使得攻击者能够猜测出下一个生成的随机数，进而凭借猜测的随机数发动攻击。至于 PRNG 究竟哪里不安全，我在后文的案例部分再向你详细介绍。

举个例子，如下代码尝试去为用户生成 session 值：

[复制代码](#)

```
1 function generateSessionValue( $user ) {  
2     // 注释：设置种子值  
3     srand( $user );  
4     return rand();  
5 }
```

由于代码中所使用的种子值每次都是不变的，因此该函数返回的 session 值也不会发生变化，攻击者可以利用该缺陷尝试劫持会话。

小空间种子选择

在上面描述的坏代码样本中，如果我们尝试去优化，那么方案是什么呢？

考虑到，这段代码存在的问题，是将种子值设定成了固定数值，那么为了解决这个问题，我们现在尝试将种子值，设置为随机数值：

[复制代码](#)

```
1 function generateSessionValue( $user ) {  
2     // 注释：设置种子值  
3     $random_val = rand(0,9);  
4     srand( $random_val );  
5     // ...  
6 }
```

这样的优化，是否会修复代码中不安全漏洞呢？答案是否定的，这种优化，只是小幅度提高了，漏洞利用的复杂程度。在上述代码中，由于 random_val 的取值空间过小，将会面临暴力破解攻击，攻击者只需要执行 10 次遍历，就可以找到被生成的随机数。

由于使用了取值空间很小的种子，这段代码将被暴露在暴力破解攻击中。要知道计算机是一种执行确定行为的机器，因此是无法生成真正的随机数的。虽然伪随机数生成器从算法设计层面满足了相似的随机性特征，但其一旦设定了种子值，其生成的随机数序列就是完全确定的。正因如此，我们要尽量确保种子值对于攻击者是不可预测的。

密码学安全的伪随机数

再进一步，我们将种子值的随机性放大，继续优化代码的安全性。

可以看到，如下代码使用`Random.nextInt()`函数来生成新的 URL 地址，其种子值由`(new Date()).getTime()`生成，该数值为 1970 年 1 月 1 日 00:00:00 GMT 至今的毫秒数，已经具备较强的随机性和不可预测性。那么，这是否能说明，这段代码已经安全了呢？

[复制代码](#)

```
1 String generateUrl( String baseUrl ) {
2     Random randomGen = new Random();
3     randomGen.setSeed((new Date()).getTime());
4     return (baseUrl + randomGen.nextInt(400000000) + ".html");
5 }
```

答案并非如此。`Random.nextInt()`函数是`java.util.Random`类的成员函数，而`java.util.Random`类是一个统计学意义上的伪随机数生成器，因此会更容易被攻击者猜到生成的数值，**对于安全性敏感的应用，建议使用密码学安全的随机数生成器 `java.security.SecureRandom`。**

案例实战

做了这么多年安全，有一句话我非常喜欢，“Talk is cheap, show me the code”。


很多高深的安全知识，讲出来似乎都十分有道理，但是实践起来往往不是那么回事。现在我们就来一起解答上面提到的问题：虽然推荐使用 CPRNG，但是 PRNG 究竟哪里不安全了呢？知其然更要知其所以然，接下来我们就上干货，带你实战攻击伪随机数生成器！

这次我们以漏洞 CVE-2019-10908 为例，这是一个在 Airsonic 10.2.1 版本存在的漏洞。Airsonic 是一个免费并且开源的产品，由社区驱动开发和维护，它是一个提供分享和访问多媒体流功能的 Web 应用。

该项目的 `RecoverController.java` 通过 `org.apache.commons.lang.RandomStringUtils` 来生成用户密码，而 `RandomStringUtils` 内部实现其实是使用了 `java.util.Random`。这里引入了两个潜在的安全隐患，一方面 `Random` 类是 PRNG，无法提供密码学安全的伪随机数生成；另一方面 `RandomStringUtils` 使用了 48bit 的种子，使其能够较容易地被攻击者爆破。


接下来我们从攻击种子的角度尝试进行漏洞利用，不过在判断对不对之前，要先判断是不是，因此在我们进行漏洞利用之前要先分析清楚是否真的存在这个漏洞。

首先来看一下 Airsonic 10.2.1 版本的源码：

 复制代码

```
1 package org.airsonic.player.controller;
2
3 // ...
4 import org.apache.commons.lang.RandomStringUtils;
5 // ...
6
7 // ...
8 public ModelAndView recover(HttpServletRequest request, HttpServletResponse re
9     Map<String, Object> map = new HashMap<String, Object>();
10     String usernameOrEmail = StringUtils.trimToNull(request.getParameter("user
11
12     if (usernameOrEmail != null) {
13         map.put("usernameOrEmail", usernameOrEmail);
14         User user = getUserByUsernameOrEmail(usernameOrEmail);
15
16         boolean captchaOk;
17         if (settingsService.isCaptchaEnabled()) {
18             String recaptchaResponse = request.getParameter("g-recaptcha-respo
19             ReCaptcha captcha = new ReCaptcha(settingsService.getRecaptchaSecr
20             captchaOk = recaptchaResponse != null && captcha.isValid(recaptcha
21         } else {
22             captchaOk = true;
23         }
24
25         if (!captchaOk) {
26             map.put("error", "recover.error.invalidcaptcha");
27         } else if (user == null) {
28             map.put("error", "recover.error.usernotfound");
29         } else if (user.getEmail() == null) {
30             map.put("error", "recover.error.noemail");
31         } else {
32             // 注释
33             // 这行代码引入了潜在的安全风险
34             String password = RandomStringUtils.randomAlphanumeric(8);
35 // ...
```

根据代码段中我添加的注释，我们继续分析 RandomStringUtils：

 复制代码

```
1 package org.apache.commons.lang;
2
3
4 import java.util.Random;
5
6 public class RandomStringUtils {
7     // 注释1
8     private static final Random RANDOM = new Random();
9
10    public RandomStringUtils() {
11    }
12    // ...
13    // 注释2
14    public static String randomAlphanumeric(int count) {
15        return random(count, true, true);
16    }
17    // ...
18    // 老师加的注释3
19    public static String random(int count, int start, int end, boolean letters
20        // ...
```

在注释 1 部分，可以看到 RandomStringUtils 类在调用过程中会创建一个静态的成员变量 RANDOM，在应用运行过程中，这个变量会一直存在。

在注释 2 部分，是 randomAlphanumeric 的函数实现，不断追踪函数调用栈可以发现最终调用了注释 3 部分的函数，而调用过程最后一个函数参数即是注释 1 部分创建的 RANDOM 变量，这一调用过程意味 RANDOM 是唯一一组随机数序列，只要我们判断出 RANDOM 序列即可执行攻击。

那么顺着思路，我们继续分析关键点——RANDOM。RANDOM 是通过 Random() 创建的 java.util.Random 对象。如下是直接调用 Random() 生成新对象时的代码：

[复制代码](#)

```
1 public Random() {
2     this(seedUniquifier() ^ System.nanoTime());
3 }
4
5 private static long seedUniquifier() {
6     // L'Ecuyer, "Tables of Linear Congruential Generators of
7     // Different Sizes and Good Lattice Structure", 1999
8     for (;;) {
9         long current = seedUniquifier.get();
10        long next = current * 1181783497276652981L;
11        if (seedUniquifier.compareAndSet(current, next))
12            return next;
13    }
```

```
14 }
```

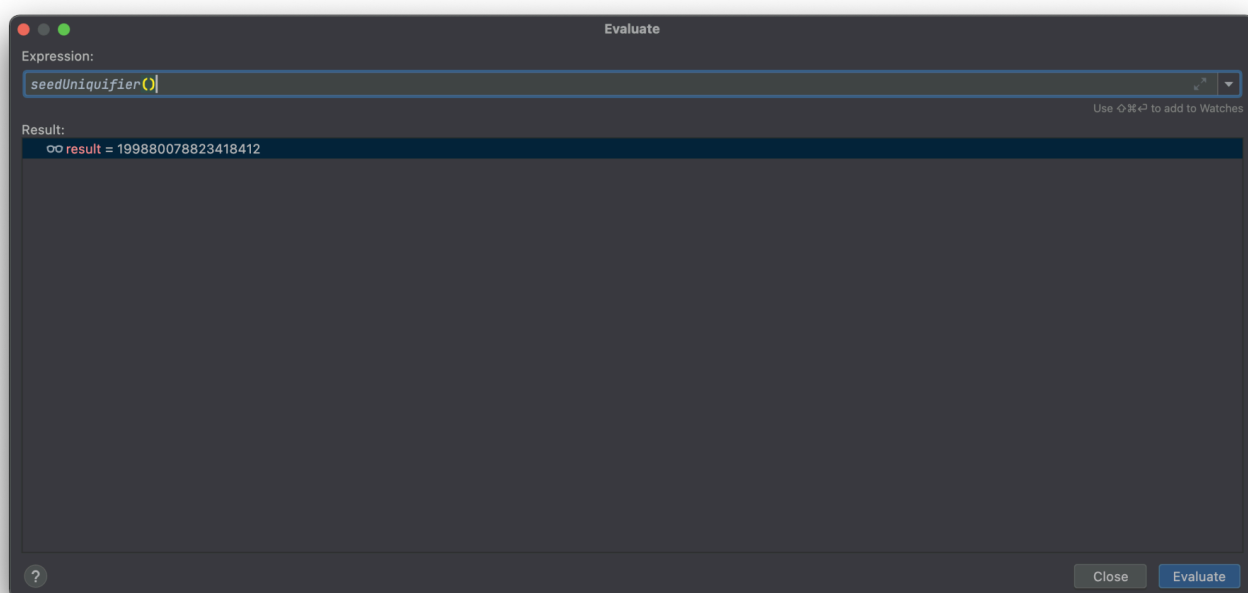
可以看到在 `Random()` 函数内部执行了种子值的设置，而具体种子值的计算则是由一段数学运算得出。

接下来我们尝试写一个 Demo 程序，来搭建一个最简单的环境进行安全性分析：

[复制代码](#)

```
1 import org.apache.commons.lang.RandomStringUtils;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.println("Hello World!");
7
8         String password = RandomStringUtils.randomAlphanumeric(8);
9
10        System.out.println(password);
11    }
12 }
```

开启调试模式，然后断点下在 `Random()` 函数内部，通过 IDEA 集成的 Expression 查看功能，我们来看一下 `Random()` 所设置的种子值大概长什么样子：

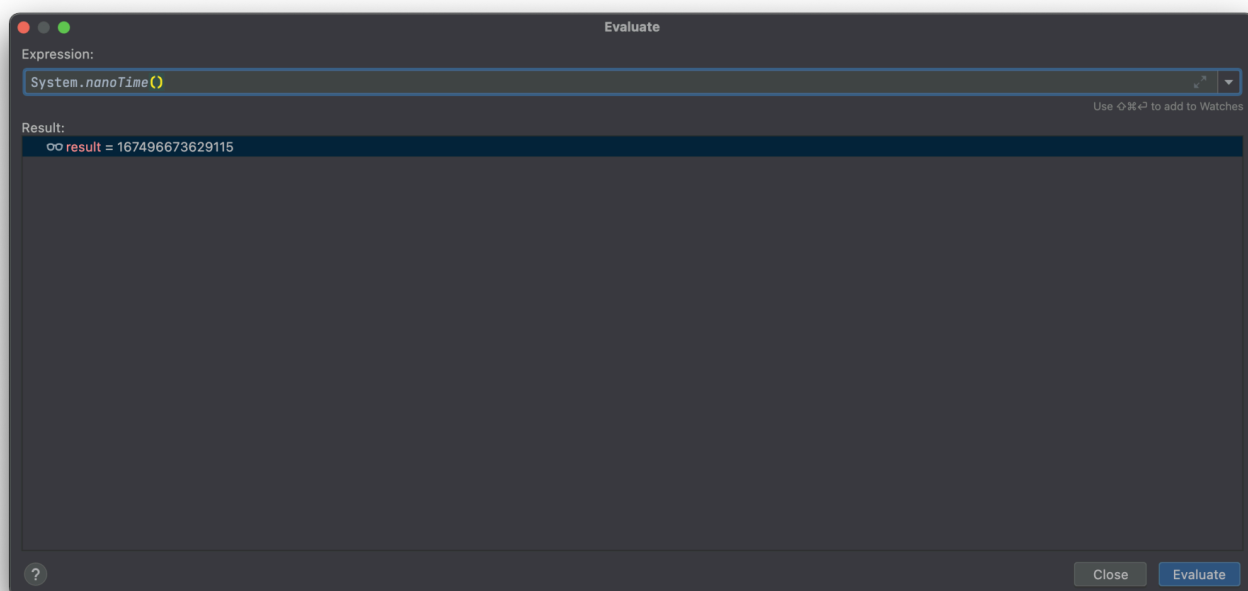


通过多次执行 Demo 程序，我们可以发现 `seedUniquifier()` 的取值序列每次都是相同的，以下是我通过调试模式取出的数值：

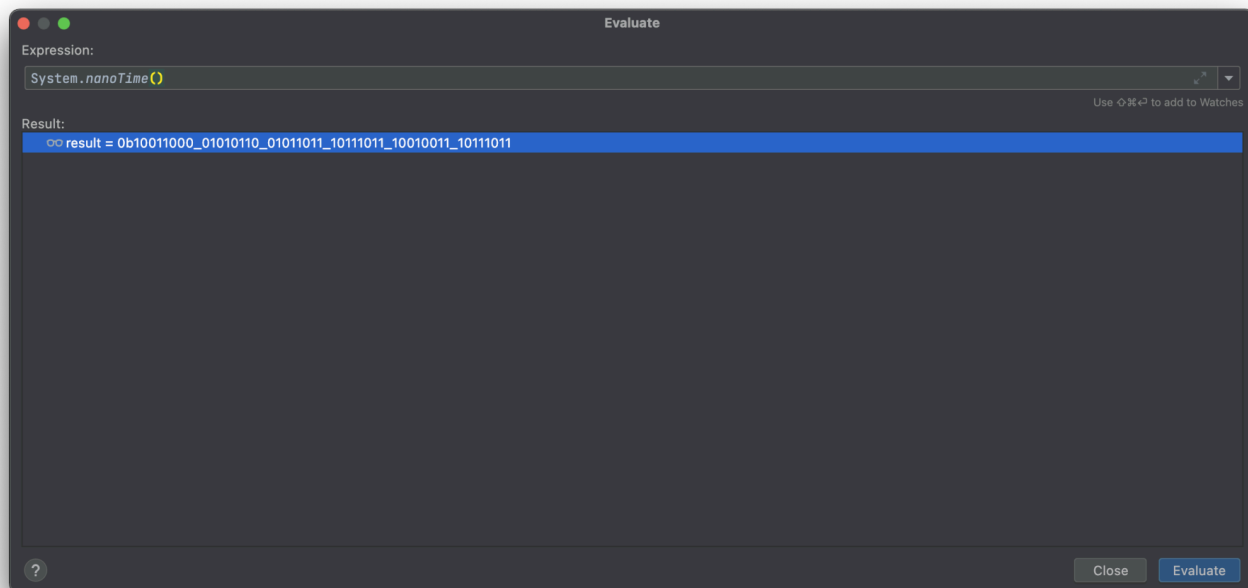
[复制代码](#)

```
1 8006678197202707420
2 -3282039941672302964
3 3620162808252824828
```

考虑到种子值是由 `seedUniquifier() ^ System.nanoTime()` 计算得出，而 `seedUniquifier()` 的取值序列固定，因此种子值将取决于 `System.nanoTime()` 函数。继续通过 Expression 查看功能：



启用二进制模式查看：




可以发现`System.nanoTime()`的取值空间为 48bit，取值范围为 $2^{48}=281474976710656 \approx 2.8 \times 10^{14}$ ，这看起来并不是一个很大的数字，直观上存在爆破的可能性。那么我们再简单修改一下 Demo 程序，看一下我们电脑的算力如何：

[复制代码](#)

```
1 import org.apache.commons.lang.RandomStringUtils;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // write your code here
7         System.out.println("Hello World!");
8
9         long startTime = System.nanoTime();
10
11         for (int i = 0; i < 1000000000; i++) {
12             String password = RandomStringUtils.randomAlphanumeric(8);
13         }
14
15         long duringTime = System.nanoTime() - startTime;
16
17         System.out.println(duringTime);
18         System.out.println(duringTime / 1000000000.0);
19     }
20 }
```


如下是执行结果：

 复制代码

```
1 Hello World!  
2 13950490348  
3 13.950490348  
4  
5 Process finished with exit code 0
```

简单总结一下就是按照这个程序的执行速度，进行 100000000 也就是 10^8 次运算需要 13.95 秒。

有了这些基本数据，就很好进行分析了，简单的除法运算就能够分析清楚爆破所需时间：

 复制代码

```
1  $2.8 \times 10^{14} / 10^8 * 13.95 = 3.9 \times 10^7$  秒 = 10833小时 = 451天
```

虽然找到了攻击方案，并且根据算力情况评估出了攻击成本，但很明显这个耗时有些过长了，真实场景很难实施，因此**我们需要想办法降低时间消耗**。

既然考虑到了效率优化，就需要关注一下我们上面所构建的 Demo 程序是否已经实现效率最大化。从代码逻辑上来看，很明显 Demo 程序是一个单进程单线程应用，可是我的电脑的 CPU 核心应该是 8 核 16 线程，因此我们可以从这个角度入手去提升它的效率优化空间。从 Demo 程序执行时的 CPU 占用率也可以看得出，在运行期间事实上仅有一个核心达到了 100% 占用：



于是我决定优化一下程序执行逻辑，并预期可以取得 16 倍的性能提升：

复制代码

```
1 package org.example;
2
3 import org.apache.commons.lang.RandomStringUtils;
4
5 import java.util.Random;
6 import java.util.stream.IntStream;
7 import java.time.Instant;
8 import java.time.Duration;
9 import java.security.SecureRandom;
10
11 public class Main {
12
13     public static void main(String[] args) {
14         if (args.length != 2) {
15             System.out.println("must be 2 arguments");
16             return;
17         }
18         int from = Integer.parseInt(args[0]);
19         int to = Integer.parseInt(args[1]);
20         System.out.println("from " + from + " to " + to);
21         System.out.println(Runtime.getRuntime().availableProcessors());
22     }
23 }
```

```
23     eval("sequential", () -> {
24         for (int i = 0; i <= 100000000; i++) {
25             new RandomStringUtilsTest(i).randomAlphanumeric(8);
26         }
27     });
28
29     eval("parallel", () -> {
30         IntStream.range(0, 100000000).parallel().forEach(i -> new RandomSt
31     });
32 }
33
34 public static void eval(String task, Runnable runnable) {
35     Instant start = Instant.now();
36     runnable.run();
37     Instant end = Instant.now();
38     System.out.printf("%s spend %s%n", task, Duration.between(start, end))
39 }
40 }
```

从输出结果可以发现并行计算下的效率大约是串行计算的 16 倍：

[复制代码](#)

```
1 from 1 to 100000
2 16
3 sequential spend PT15.235S
4 parallel spend PT1.826S
5
6 Process finished with exit code 0
```

再观察一下执行期间的 CPU 占用，可以发现 CPU 确实是全核心在运行的。



这里仅以我的个人计算机为例进行模拟攻击的时间测算，在实际应用场景中，我们完全可以通过调度云端计算机使上百核心的并发计算，我们以 128 核心为例：

复制代码

1 451天 / 128 ≈ 3.5天

可以发现在上面这种情况下，我们只需要 3.5 天即可完成针对该漏洞的攻击。

值得展开讨论的是，虽然在上述情况中我们可以成功进行攻击，但是实际情况往往会更加复杂，比如我们已知的随机数已经是生成的第几百个随机数，这种情况下由于需要产生较长的随机数序列进行匹配，将会导致更大的计算量。但是这种计算量增长都是线性的，我们仍然可以根据需求选择能够负载的时间和金钱成本，顺利完成攻击。

安全实践

以 CVE-2019-10908 为例，我们来看一下开源项目的修复方案。

首先从包的使用方面，取消了org.apache.commons.lang.RandomStringUtils的使用，并且替换为java.security.SecureRandom，根据名字很容易判断新替换的包是一个密码学安全的伪随机数生成器，该随机数生成器从算法设计角度上是密码学安全的，同时内部所使用的种子值强度也会更高。

然后在代码实现层面，放弃了原有的RandomStringUtils.randomAlphanumeric函数，转为使用新的代码实现：

[复制代码](#)

```
1     private static final String SYMBOLS = "abcdefghi jklmnopqrstuvwxyzABCDEFGH
2     private final SecureRandom random = new SecureRandom();
3     private static final int PASSWORD_LENGTH = 32;
4     // ...
5     StringBuilder sb = new StringBuilder(PASSWORD_LENGTH);
6     for(int i=0; i<PASSWORD_LENGTH; i++) {
7         int index = random.nextInt(SYMBOLS.length());
8         sb.append(SYMBOLS.charAt(index));
9     }
10    String password = sb.toString();
```

简单来说，就是在开发过程中要通过安全的加密库来使用 CPRNG。在不同语言中涉及的模块会有一些差异，比如 Linux 或者 MacOS 中，大部分加密库内部都依赖于/dev/random或/dev/urandom这两个随机源；在 Windows 中可以使用 Crypto API 或者 BCryptGenRandom；在C#中可以使用 System.Security.Cryptography.RandomNumberGenerator；在 Python 中可以使用 os.urandom 或 secrets 库；在 Java 中则可以使用我们本节课介绍的 java.security.SecureRandom。

总结

这节课我们学习了随机数相关的知识。

虽然物理世界充满了不确定性和随机性，但是计算机世界并非如此。计算机是一种执行确定计算过程的机器，我们在开发过程中使用的随机数基本都是由软件算法生成的伪随机数。但即使是伪随机数，也有安全和不安全之分。

常规的伪随机数生成器又称 PRNG，是基于概率设计的；而为了保证安全则需要使用密码学意义上伪随机数生成器，这种生成器又叫 CPRNG。这些经过 PRNG 或 CPRNG 生成的数字，可以理解为重复周期非常大的序列，因此能够满足随机性需求。而既然是序列，就一定会有一个开始，这个开始值的产生由种子值确定，这个种子我们又称之为 seed。

在考虑到 PRNG 以及 CPRNG 算法安全的情况下，种子值的安全成为了关键要素，因为一旦种子值可以被预测则生成的所有随机数都将可以被预测。由此引发的一系列安全风险包括无效的随机数、小空间种子值以及非密码学安全的随机数生成器等。

虽然围绕密码学和随机数展开的攻击案例并不多见，但是本节课我们以 CVE-2019-10908 为例进行了真实的随机数层面的攻击，从实战过程可以发现这类攻击要求攻击者具备足够深厚的密码学和随机数相关的知识，同时需要具备一定的开发功底。正因如此，许多开发者并未重视这类安全问题，与 CVE-2019-10908 相似的安全风险仍普遍存在于许多应用中。

关于随机数方面的安全建议，仍然可以参考 CVE-2019-10908 的开源项目更新方案，从安全库的使用及随机数生成过程两个方面共同优化。

也许你会觉得本节课程中涉及的漏洞利用过程稍有复杂，不过别担心，随着课程的不断更新，在后续课程中你将学会搭建和使用属于自己的个性化智能攻防对抗系统，这将大幅度降低类似攻击过程的复杂度及难度，让你的安全能力持续积累、快速提高。

思考

本节课我们以 CVE-2019-10908 为例研究了随机数攻击中的种子爆破攻击，你可以尝试从 PRNG v.s. CPRNG 的角度分析其安全缺陷吗？

欢迎你在评论区留下自己的思考，我们下节课再见。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

下一篇 11 | 忘记加“盐”：加密结果强度不够吗？