GitHub 安全实验室 ctf4:CodeQL 和 Chill-Java 版

这个 CTF 现在关闭了！你仍然可以挑战自己的乐趣！

语言：Java-难度：
寻找寻找漏洞的挑战？那么这个 javactf 挑战就是为您准备的！您将磨练您的错误发现技能，并学习所有关于 CodeQL 的污点跟踪功能。
如果您选择接受它，您的任务是在一个流行的容器管理平台中寻找最近发现的漏洞。此漏洞使攻击者能够注入任意 Java EL 表达式，从而导致预授权远程代码执行（RCE）漏洞。
使用 CodeQL 跟踪从用户控制的 bean 属性到自定义错误消息的受污染数据，您将学习如何填补污点跟踪中的任何空白，从而为漏洞创建完整的数据流路径。
先决条件
为了完成这一挑战，参与者必须对 CodeQL 及其用于数据流分析的库有一定的了解。

*CodeQL 新手？别担心，这里有大量的 CodeQL 资源可以帮助您开始！试行*一个关于 CodeQL for C++的 GitHub 学习实验室课程使用 CodeQL for JavaScriptCodeQL Java 培训示例捕捉标志挑战，*以前的，或. 那就回来试试这个 CTF 吧。*

如何提交？

- 创建一个秘密的 GitHub 主旨或私有 GitHub 存储库。
- 提交你的文章，在你的要点或自述文件.md 或者在你的回购文件中。

- 您可以直接在主报告中添加响应，也可以在主报告中引用的单独文件中添加响应。
- 当你准备好提交时，只需发送电子邮件 ctf@github.com 带有指向主旨或回购的链接（如果您使用私有回购，请首先邀请用户 securitylab ctf 作为合作者）。

需要帮助吗？

- 你的第一站应该是文档。如果您需要更多关于 CodeQL 概念的帮助，请访问我们的论坛。小心，不要把你的解决方案泄露给其他竞争对手；-）
- 您可以联系我们 ctf@github.com
- 你也可以加入我们的 Slack 工作区。

介绍

许多应用程序，包括一些容器管理平台，都使用 javabean 验证来验证应用程序中的 javabean 对象是否满足开发人员设置的某些约束，如果这些约束不满足，它将呈现自定义错误消息。但是，在某些情况下，bean 是从用户控制的数据（如 HTTP 请求）中解组的。如果在将 bean 的属性用于自定义错误消息之前未正确清理，攻击者可以将任意 javael 表达式注入到 bean 属性中，这将导致在呈现错误消息时远程执行代码。在我们的咨询中了解更多关于这个问题的信息。

在这个挑战中，您将使用 CodeQL 跟踪从用户控制的 bean 属性到自定义错误消息的受污染数据流，并识别已知的注入漏洞。您还将学习如何为 Java 定制 CodeQL 数据流分析，以帮助您更好地探索源代码并找到更广泛的相关漏洞。

挑战性问题

该平台使用 javabean 验证（jsr380）自定义约束验证器，例如 com.netflix.titus 网站.api.jobmanager.model.job.sanitizer.SchedulingConstraintSetValidator.

阅读 BeanValidation2.0 规范，了解自定义验证器以及在呈现自定义约束错误消息时发生的插值类型。在构建自定义约束冲突错误消息时，必须了解它们支持不同类型的插值，包括 javael 表达式。因此，如果攻击者可以在传递给 Constr 的错误消息模板中插入任意数据 aintValidatorContext.buildConstraintViolationWithTemplate（）的第一个参数是，它们将能够运行任意 Java 代码。这些错误消息模板是用于注入的接收器漏洞。

不幸的是，经过验证的 bean 属性流到自定义错误消息中是很常见的。这些必须根据约束条件进行验证，因为它们很可能是用户控制的数据源。

作为一个例子，考虑 SchedulingCo 中的这段代码 nstraintSetValidator.java. 这里的 container 是一个正在验证的对象（因此很可能是不可信的），但是它的属性最终在 set common 中结束，后者用于创建错误消息模板而不进行清理。

```
@Override
public boolean isValid(Container container, ConstraintValidatorContext
context) {
```

```
    if (container == null) {
        return true;
    }
    Set<String> common = new
HashSet<>(container.getSoftConstraints().keySet());
    common.retainAll(container.getHardConstraints().keySet());
    if (common.isEmpty()) {
        return true;
    }
    context.buildConstraintViolationWithTemplate(
            "Soft and hard constraints not unique. Shared constraints: " +
common
    ).addConstraintViolation().disableDefaultConstraintViolation();
    return false;
}
```

在这个挑战中，我们希望识别这种模式的出现，其中用户控制的对象（可能包含 javael 表达式）将流入错误消息中。

安装说明

1. 安装 Visual Studio 代码 IDE。
2. 转到 CodeQL starter 工作区存储库，并按照该存储库自述文件中的说明进行操作。完成后，应该在 visualstudio 代码中安装 CodeQL 扩展并打开 vscode CodeQL starter 工作区。
3. 下载并解压缩这个 CodeQL 数据库，它对应于未修补的版本 8a8bd4c。
4. 将数据库导入 visualstudio 代码（请参阅文档）。

步骤 1：数据流和污点跟踪分析

正如导言中所解释的，报告这些问题将涉及跟踪通过应用程序的受污染数据流。在这一步中，我们将为 CodeQL 污染跟踪分析准备基本的构建块：源和汇。

步骤 1.1：来源

受污染数据的来源是经过约束验证的 bean 属性。在代码中，这些可以作为 ConstraintValidator.isValid(...).
编写一个 CodeQL 谓词来标识这些调用参数：

```
predicate isSource(DataFlow::Node source) { /* TODO describe source */ }
```

要测试谓词，请使用快速求值命令（右键单击>CodeQL:快速求值）。你应该得到 6 个结果。

提示：

- 确保只捕获在 ConstraintValidator 接口中定义的方法的实现。例如，不应将此案例视为来源。
- 有一个方便的类 RemoteFlowSource 告诉您何时从远程用户输入获取特定的数据流节点。
- 注意只获取与项目源代码相关的结果。

奖金

只有在区分质量和完整性非常接近的提交文件时，才会考虑这种可选改进。
您会注意到，这个实现会将每个经过验证的 bean 属性标记为污染源。但我们只想得到用户控制的资源。我们对攻击者无法控制的 bean 属性不感兴趣，例如当 bean 来自于对应用程序配置文件进行解组时。
考虑改进谓词，这样我们只考虑 bean 类型绑定到用户可控制数据（如 JAX-RS 端点）的情况。

步骤 1.2：sink

我们正在考虑的注入汇作为 Constr 调用的第一个参数出现 aintValidatorContext.buildConstraintViolationWithTemplate(...).
编写一个 CodeQL 谓词来标识这些接收器。

```
predicate isSink(DataFlow::Node sink) { /* TODO describe sink */ }
```

快速评估你的谓词应该会得到 5 个结果。

步骤 1.3：污点跟踪配置

在进一步讨论之前，我们建议您快速评估 isSource 和 ISink 谓词，以确保两者都与上述 SchedulingCo 中描述的问题匹配 nstraintSetValidator.java. 这个案子将是我们接下来的挑战的主要目标。
都做完了吗？好的，现在让我们通过跟踪受污染的数据来找到这个易受攻击的路径！
您需要按照 CodeQL 文档中的说明创建污点跟踪配置。在下面的模板中填写 isSource 和 ISink 的定义，以及一个更好的名称。谓词 hasFlowPath 将保存数据从源到接收器的任何路径。当您检查谓词是否为您提供了正确的源和汇时，我们将得到我们的漏洞。

```
/** @kind path-problem */
import java
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PathGraph

class MyTaintTrackingConfig extends TaintTracking::Configuration {
    MyTaintTrackingConfig() { this = "MyTaintTrackingConfig" }

    override predicate isSource(DataFlow::Node source) {
        // TODO
    }

    override predicate isSink(DataFlow::Node sink) {
        // TODO
    }
}

from MyTaintTrackingConfig cfg, DataFlow::PathNode source,
DataFlow::PathNode sink
```

```
where cfg.hasFlowPath(source, sink)
select sink, source, sink, "Custom constraint error message contains
unsanitized user data"
```

使用“运行”命令或“运行”命令。它应该给你。。。0 个结果！好吧，这太令人失望了！但现在不要放弃。

步骤 1.4：分流至救援

在开发污点跟踪查询时，我们可能经常会遇到这种情况。为什么我们没有被击中？

我们确定了源和汇，因此这表明我们的分析缺少了从源到汇的路径上的一个步骤。

CodeQL 的 Java 库可以帮助我们找到部分数据流调试机制缺失的空白。此功能允许您查找从给定源到任何可能的接收器的流，使接收器不受约束，同时限制要搜索的远离源的步骤数。因此，您可以使用此功能来跟踪从源到所有可能的接收器的受污染数据流，并查看进一步跟踪流的停止位置。

创建使用 hasPartialFlow 谓词的调试查询。您可以使用下面的模板。

```
/** @kind path-problem */
import java
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PartialPathGraph // this is different!

class MyTaintTrackingConfig extends TaintTracking::Configuration {
    MyTaintTrackingConfig() { ... } // same as before
    override predicate isSource(DataFlow::Node source) { ... } // same as
before
    override predicate isSink(DataFlow::Node sink) { ... } // same as
before
    override int explorationLimit() { result =  10} // this is different!
}
from MyTaintTrackingConfig cfg, DataFlow::PartialPathNode source,
DataFlow::PartialPathNode sink
where
  cfg.hasPartialFlow(source, sink, _) and
  source.getNode() = ... // TODO restrict to the one source we are
interested in, for ease of debugging
select sink, source, sink, "Partial flow from unsanitized user data"

predicate partial_flow(PartialPathNode n, Node src, int dist) {
  exists(MyTaintTrackingConfig conf, PartialPathNode source |
    conf.hasPartialFlow(source, n, dist) and
    src = source.getNode() and
    source =  // TODO - restrict to THE source we are interested in
  )
}
```

运行修改后的查询，以探索数据流并检测路径停止的位置。
提示：

- 还是没有结果？因此，请务必仔细阅读 particle traint 配置，并相应地阅读 partialtaint 配置。
- 可以使用 exists 关键字添加分析所需的其他变量，或者作为 from 子句的参数。例如，可以限制在特定的封闭函数或文件中仅限于源节点和汇节点位置。这个将帮助您快速筛选出您感兴趣的结果。

步骤 1.5：识别缺失的污染步骤

您一定已经发现 CodeQL 不会通过像 container.getHardConstraints 以及 container.getSoftConstraints. 你能猜出为什么要实现这种默认行为吗？

步骤 1.6：添加其他污染步骤

现在你知道有些污点的步骤不见了。这是因为默认情况下，分析是谨慎的，并且尽量不给您额外的流，这可能导致误报。现在您需要告诉您的污点跟踪配置，受污染的数据可以通过某些代码模式传播。
CodeQL 允许您在特定的污点跟踪配置中声明额外的污染步骤，如本例所示。
但是，我们将使用一种更通用的方法，它允许我们全局地添加污点步骤，以便可以通过几个污点跟踪配置来获取这些步骤（并可能在许多查询中重用）。为此，您只需扩展 TaintTracking:：AdditionalTaintStep 类并实现步骤谓词。
当受污染的数据从 node1 流向 node2 时，step 谓词应该为 true。
添加污点步骤后再次运行原始查询。你得到预期的结果了吗？还是不行。
再次运行分部流查询，以查找这次丢失了受污染数据跟踪的位置。
提示：在步骤谓词中，您应该指出这两个节点是 MethodAccess 的两个元素：一个是它的限定符，一个是在调用站点找到的返回值。

步骤 1.7：通过构造函数添加污染步骤

对所有中断污点跟踪的方法重复上述过程，直到您的部分流谓词最终将您带到构造函数 HashSet 的调用。
现在您可以看到 CodeQL 没有通过 HashSet 构造函数传播。为此编写一个额外的污点步骤，然后重新运行查询。

第 1.8 步：第一期结束

重复上面的过程，根据需要添加更多的污点步骤，直到污染的数据流到 buildConstraintViolationWithTemplate 调用的参数中。运行查询。
万岁！现在应该报告这个问题！

第二步：第二期

Schedulin 中也有类似的问题 gConstraintValidator.java. 按照上面相同的过程，找出为什么查询没有报告它，并编写必要的污染步骤来报告它。
提示：我们不喜欢重复代码。;-)

步骤 3：错误和异常

由于此接收器与生成错误消息相关联，因此在许多情况下，它们将从流中的异常消息生成，例如：

```
try {
    parse(tainted);
} catch (Exception e) {
    sink(e.getMessage())
}
```

我们目前的查询不包括这个案子。准确的污染步骤需要分析抛出方法的实现，以确定受污染的输入是否实际反映在异常消息中。
不幸的是，我们的 CodeQL 数据库识别对库方法的调用及其签名，但是没有这些方法实现的源代码。所以我们需要模仿他们的行为。为这些情况写一个额外的污点步骤。
注：为了测试附加的污点步骤，对其步骤谓词使用 quickevaluation 来检查它是否如您所期望的那样检测到上述模式。我们当前的代码库不包含流向这些异常消息模式的用户控制 bean 的情况，因此您无法通过运行整个查询来测试新的污染步骤。您的查询应继续只找到相同的 2 个结果。
提示：

- 阅读 codeql java 库中 TryStmt 和 CatchClause 类的文档。使用跳转到定义或在 IDE 中悬停来查看它们的定义。
- 您将不得不限制到通过调用特定方法编写异常的 catchClause。
- 使用启发式方法来决定哪些方法编写错误消息。

第 4 步：利用和补救

步骤 4.1:PoC

为它写一个工作 PoC。你可以使用官方的 Docker 图片。

步骤 4.2：补救

下载修补代码的数据库，将其导入到 VS 代码中，然后运行查询以验证它不再报告该问题。
我们的建议包含其他补救技术。修改您的查询，使其更精确，或捕捉到漏洞的更多变体。例如，考虑处理禁用 javael 插值并且只使用 ParameterMessageInterpolator 的情况。

官方规则

阅读官方比赛规则。

以前的 CTF 比赛

你可以试试以前的 ctf。