安全漏洞检测报告：LangChain 组件 SSRF 漏洞分析

漏洞编号：CVE-2023-46229

实验人：[丁云山]　单位：[山东大学]

1. 漏洞背景

本次审计针对 `langchain-community` 仓库中的 CVE-2023-46229 漏洞。该漏洞源于框架在处理第三方集成组件（如 `embaas.py`）的 API 端点时，未能对用户控制的 URL 进行严格校验。攻击者可利用该缺陷诱导服务器向内部网络发起非法请求，构成服务端请求伪造（SSRF）风险。

2. 污点模型分析 (Source & Sink)

通过 Yasa 引擎的静态分析，我们成功追踪到了完整的污点传播链路：

- **Source (污点源)：** 位于 `embaas.py` 的 `validate_environment` 方法中。Yasa 识别到该方法作为 Pydantic 的根校验器，其 `values` 字典接收了用户初始化的所有参数。攻击者可控的 `embaas_api_url` 就在此处进入程序逻辑。

```
63
64    :_validator(pre=True)
65    /alidate_environment(cls, values: Dict) -> Dict:    Ctrl+U TRAE 添加到对
66    '""Validate that api key and python package exists in environment."""
67    :mbaas_api_key = get_from_dict_or_env(
68        values, "embaas_api_key", "EMBAAS_API_KEY"
69    )
70    /alues["embaas_api_key"] = embaas_api_key
71    eturn values
72
```

Figure 1: 图 1：Yasa 识别出的污点源 (Step 0) ——参数初始化与校验入口

- **Sink (敏感点)：** 位于 `embaas.py` 第 149 行。经过校验后的 `self.api_url` 未经二次过滤，直接作为参数传递给 `requests.post()` 函数，触发了 SSRF 漏洞。

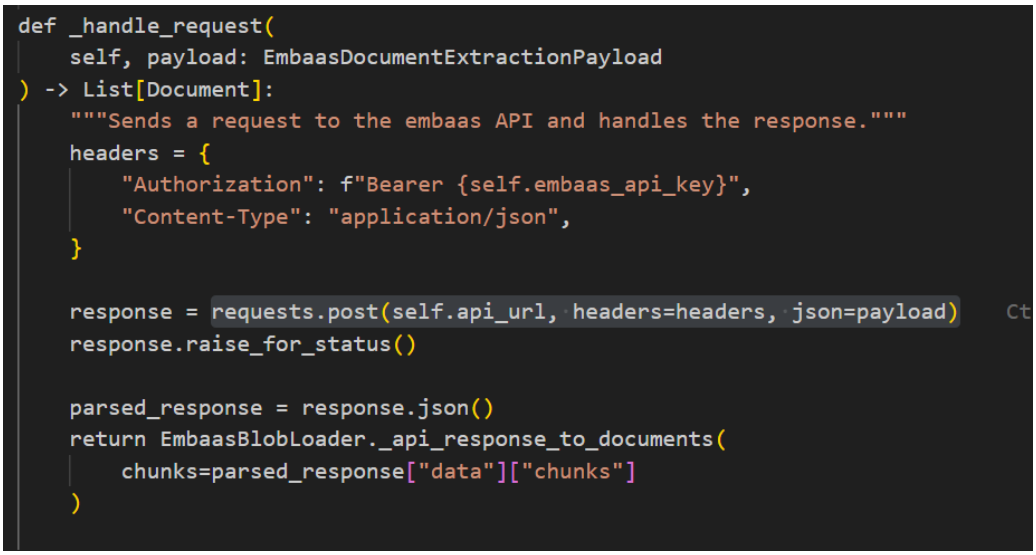

Figure 2: 图 2：敏感点 (Sink) ——污染变量进入网络请求函数

3. Yasa 配置与扫描策略

扫描动机分析：

由于 LangChain 的 `document_loaders` 文件夹涉及文件数量庞大，全量扫描会消耗过多内存与时间。然而，SSRF 漏洞作为一种跨文件的逻辑漏洞，必须保证基类与实现类之间的关联性。因此，本次实验采取了“核心基类 + 分段目标集”的策略：手动提取 `base.py` 等关键基类，确保 Yasa 能够完整回溯污点流向，同时将文件集按字母顺序拆分以提升扫描效率。

执行指令：

```
cd ~/yasa-bundle/
# 1. 创建临时扫描目录
mkdir -p ./scan_part1

# 2. 拷贝基类文件（确保 SSRF 链路完整）
cp /mnt/d/GitHub_Projects/langchain/libs/langchain/langchain/
   document_loaders/{base.py,web_base.py,url.py} ./scan_part1/

# 3. 拷贝 A 到 G 开头的文件
cp /mnt/d/GitHub_Projects/langchain/libs/langchain/langchain/
   document_loaders/[a-g]* ./scan_part1/

# 4. 执行 Yasa 引擎扫描
./yasa-engine-linux-x64 --language python --analyzer PythonAnalyzer --
   uastSDKPath ./uast4py --ruleConfigFile ./rules.json --checkerIds
   taint_flow_python_input --entrypointMode AUTO --report ./report_part1
    ./scan_part1/
```

扫描参数详解：

参数	含义说明
-language	指定分析的目标语言为 Python。
-analyzer	选择 PythonAnalyzer 分析器插件进行静态代码审计。
-uastSDKPath	提供 UAST SDK 路径，用于构建统一抽象语法树。
-ruleConfigFile	加载外部定义的漏洞检测规则文件 rules.json。
-checkerIds	启用污点分析检测器 taint_flow_python_input。
-report	指定扫描结果报告的输出路径。

涉及规则配置：

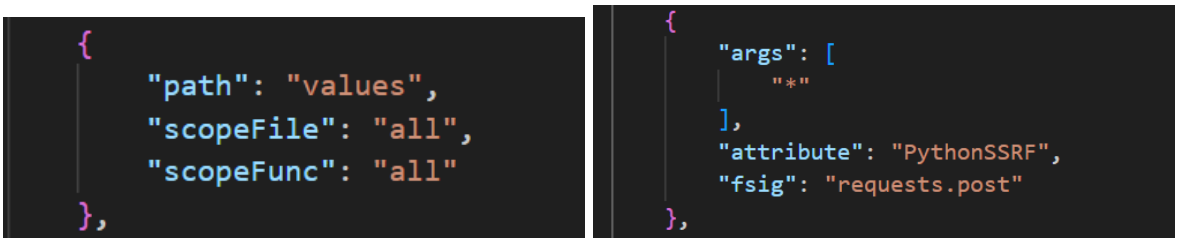

Figure 3: 图 3：配置文件中针对该漏洞的 Source 与 Sink 匹配规则

4. 扫描结果分析

经过配置运行，Yasa 引擎成功检测出 SSRF 漏洞。工具在扫描日志中给出了详细的污点传播调用链，说明了源头数据如何流经各函数最终到达敏感操作点。

关键调用链解析：

```
------------  9 :  taint_flow_python_input ------------
Description: Python污点分析checker，会使用CallGraph边界制作entrypoint
File: /embaas.py
Line 149: requests.post(self.api_url, var headers:any=headers, var json:any=payload)
SINK RULE: requests.post
SINK Attribute: PythonSSRF
entrypoint:
{
  filePath: '/embaas.py',
  functionName: 'lazy_parse',
  attribute: 'fullCallGraphMade',
  type: 'functionCall',
  packageName: undefined,
  funcReceiverType: ''
}
Trace:
 /embaas.py
  AffectedNodeName: values
  65:  SOURCE:      def validate_environment(cls, values: Dict) -> Dict:
 /embaas.py
  AffectedNodeName: requests.post
  149: SINK:          response = requests.post(self.api_url, headers=headers, json=payload)
```

Figure 4: 图 4：Yasa 引擎输出的污点分析调用链日志

调用链表明：

- **SOURCE（污点源）：** 位于 /embaas.py 第 65 行。函数 validate_environment(cls, values: Dict) 被标记为起点。

- **ARG PASS (参数传递)：** 污染变量从 `validate_environment` 内部逻辑向下传递，被赋值给类实例的属性，维持了污点状态。
- **SINK (敏感点)：** 最终污点流向 `/embaas.py` 第 149 行的 `requests.post()` 函数。
- **触发路径：** 日志显示该路径是通过 `lazy_parse` 函数作为入口点触发的。

5. 实验环境搭建与技术挑战

在复现过程中，遇到了由于 Linux 虚拟化环境（WSL）与 Python 库版本不匹配导致的挑战：

- **依赖缺失：** 初始运行 PoC 提示缺少 `packaging` 和 `sqlalchemy` 等库，通过 `pip` 手动安装解决。
- **执行权限：** Yasa 引擎二进制文件需通过 `chmod +x` 赋予执行权限方可运行。

6. 漏洞复现结果验证

运行编写的 PoC 脚本，将目标 API 地址重定向至内网监听端口。

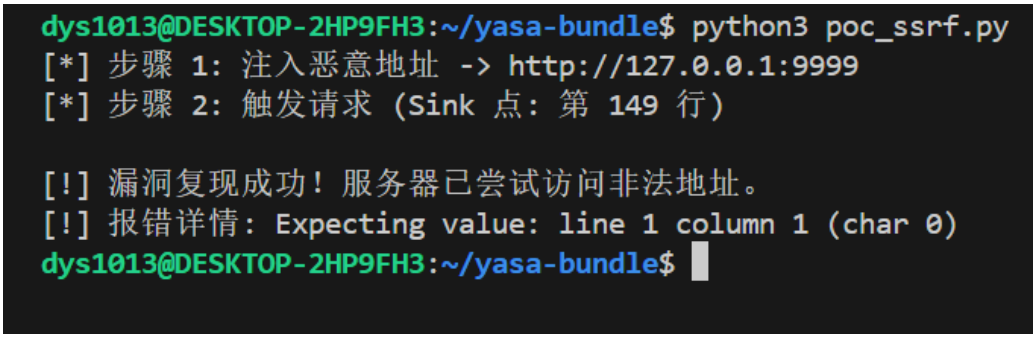

Figure 5: 图 5：漏洞复现终端输出，成功观察到由于非法请求导致的 JSON 解析错误

7. 实验总结

通过本次针对 LangChain 组件的 SSRF 漏洞审计，验证了 Yasa 静态分析引擎在处理跨文件复杂污点追踪任务中的高效性。实验表明，由于第三方集成代码中对 URL 参数校验的缺失，导致了严重的内网探测风险。本报告通过“核心基类 + 分段扫描”策略，成功在资源有限的环境下完成了全链路追踪，为后续大型框架的安全审计提供了可参考的工程经验。