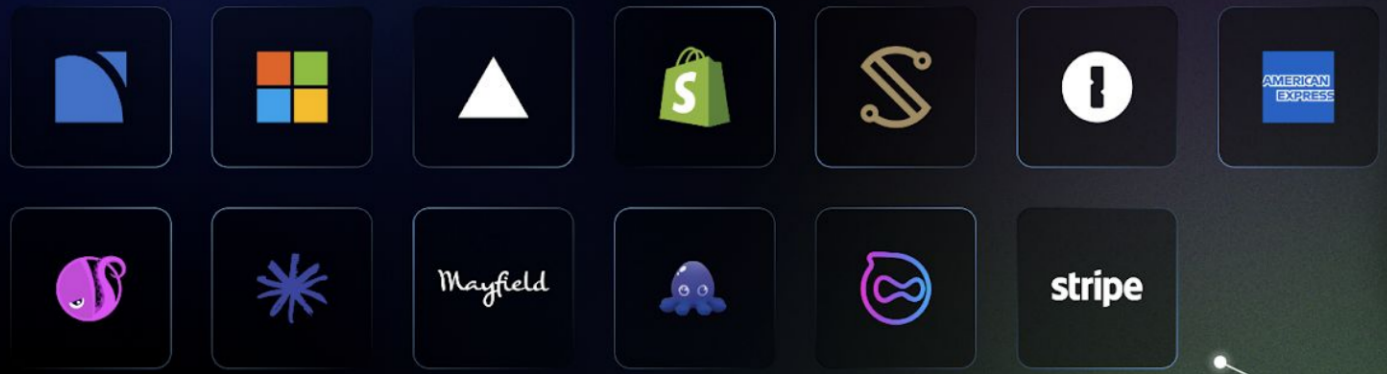
GitHub Secure Open Source Fund

powered by GitHub Sponsors



GitHub Secure Open Source Fund
powered by GitHub Sponsors

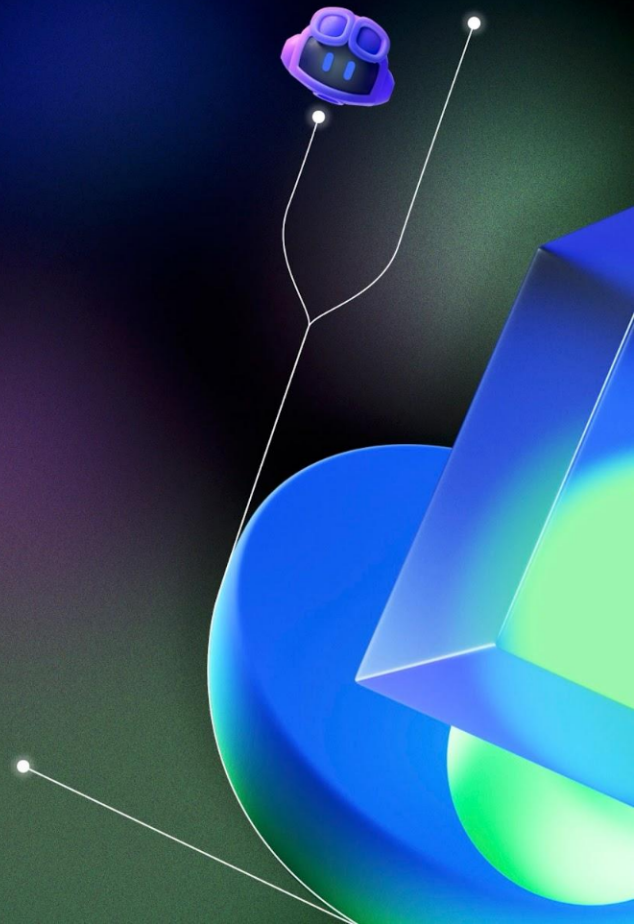
Module: Finding vulnerabilities with CodeQL

GitHub Secure Open Source Fund
powered by GitHub Sponsors

About Me

Sylwia Budzynska
GitHub Security Lab
Security Researcher

GitHub: @sylwia-budzynska



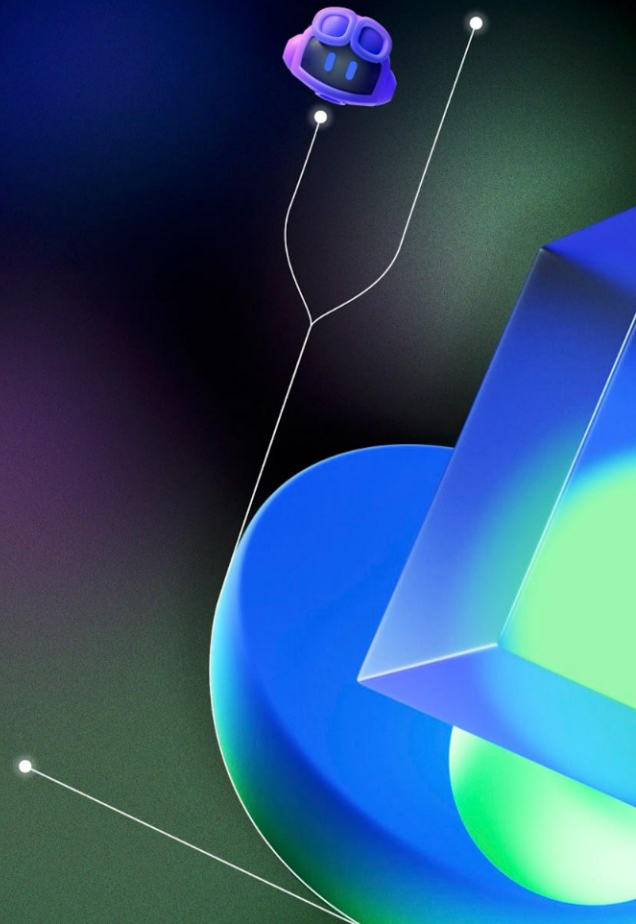
GitHub Secure
Open Source Fund
powered by GitHub Sponsors

Agenda

1. Identifying vulnerabilities in code
2. Sources and sinks in CodeQL
3. Hands-on session

How do we find vulnerabilities?

Let's start by examining an example vulnerability and how it could be detected.



What is this vulnerability?


```python
import os
from flask import Flask, request

app = Flask(__name__)

@app.route("/show-files")
def show_files():
    file = request.args.get('file', '')
    os.system("cat " + file)
```

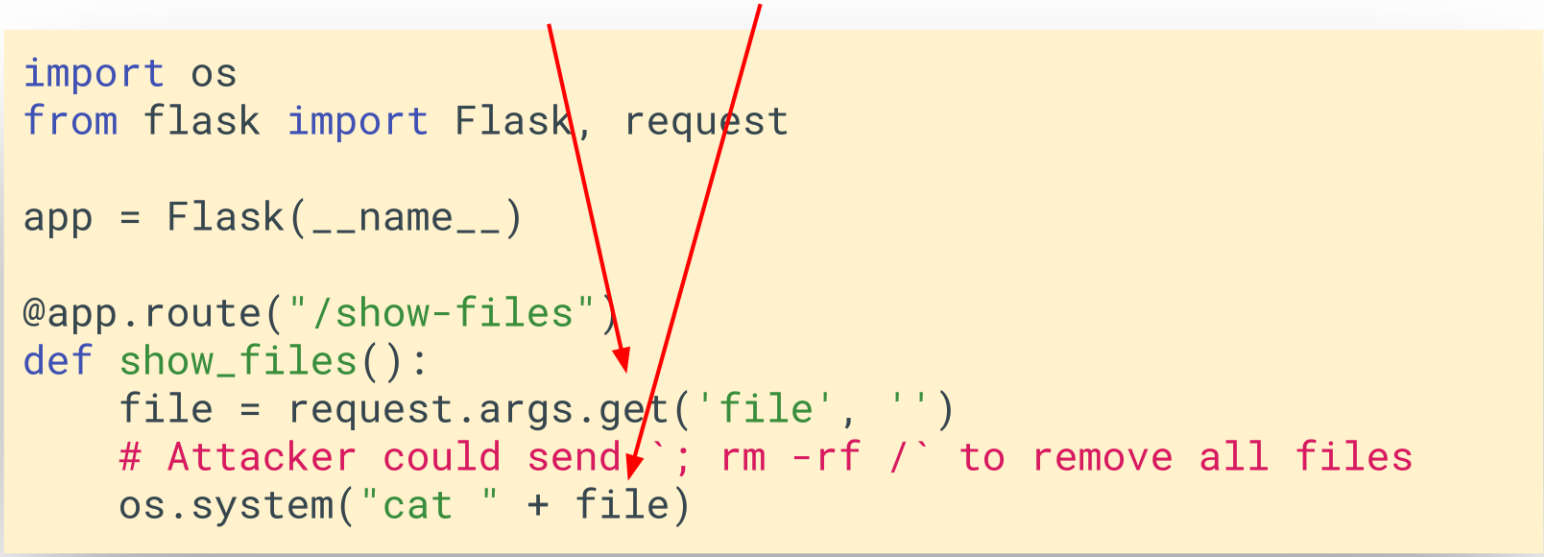
How did we identify the command injection?

This command injection is caused by the user-controlled HTTP GET parameter `file` used directly in a command, in the `os.system` call.

```
import os
from flask import Flask, request

app = Flask(__name__)

@app.route("/show-files")
def show_files():
    file = request.args.get('file', '')
    # Attacker could send `|| rm -rf /` to remove all files
    os.system("cat " + file)
```

Sources and sinks

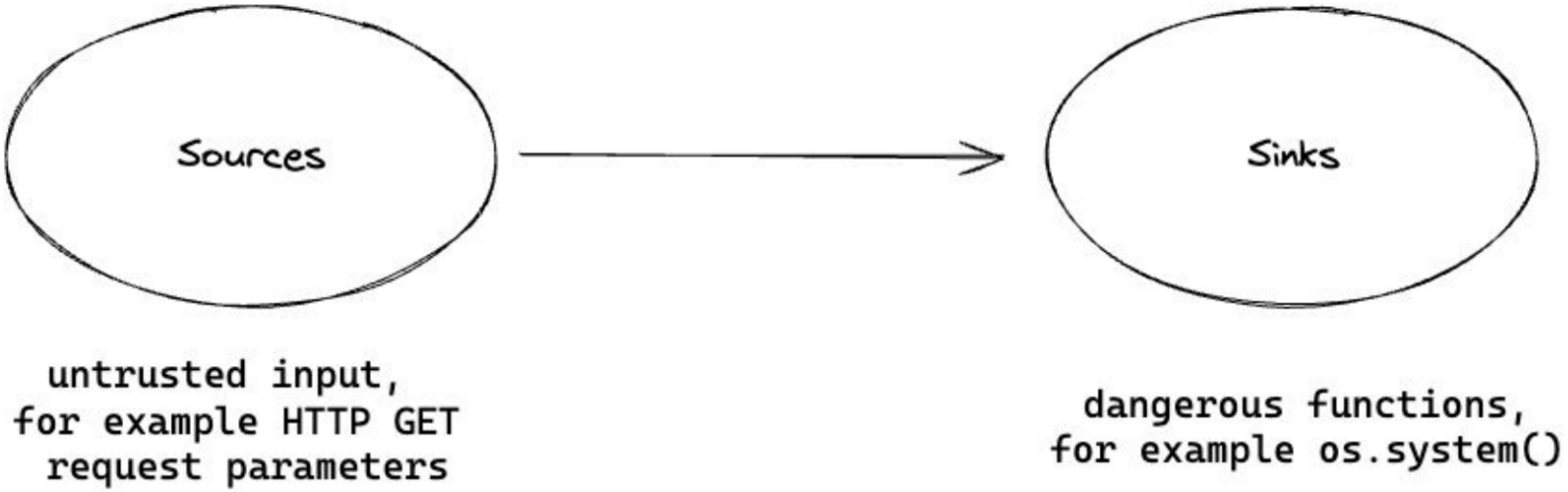
The cause of this injection vulnerability is untrusted, user-controlled **input** being used in sensitive or **dangerous functions** of the program. To represent these in static analysis, we use terms such as **data flow**, **sources**, and **sinks**.



```python
import os
from flask import Flask, request

app = Flask(__name__)

@app.route("/show-files")
def show_files():
    file = request.args.get('file', '')
    # Attacker could send `; rm -rf /` to remove all files
    os.system("cat " + file)
```

Sources and sinks

For a vulnerability to be present, there has to be a code path between the source and the sink— a “**data flow**”.

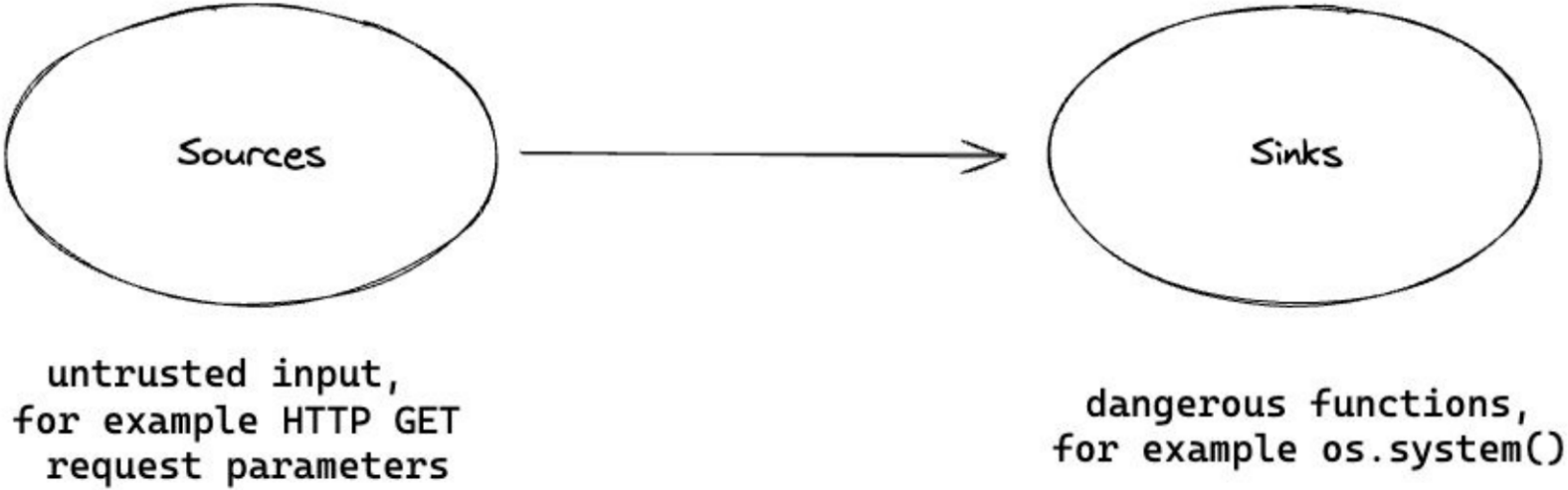


Problem

An application often has **hundreds** of sources and **hundreds** of sinks.

Checking manually, if any of the sources connect to the sinks (or vice versa, any of the sinks to the sources), takes **time**.

Is it possible to **automate** finding sources, sinks and data flow paths between them?

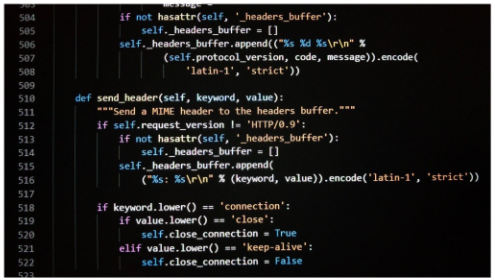


Yes,
we can check for data flows
between sources and sinks
by using

CodeQL

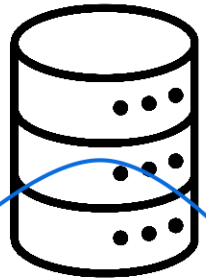
How CodeQL works

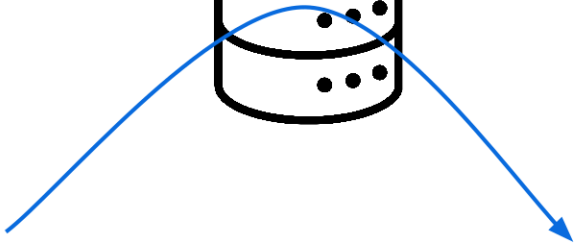
Source code



```
            if not hasattr(self, '_headers_buffer'):
                self._headers_buffer = []
            self._headers_buffer.append(("%s %d %s\r\n" %
                    (self.protocol_version, code, message)).encode(
                        'latin-1', 'strict'))

    def send_header(self, keyword, value):
        """Send a MIME header to the headers buffer."""
        if self.request_version != 'HTTP/0.9':
            if not hasattr(self, '_headers_buffer'):
                self._headers_buffer = []
            self._headers_buffer.append(
                ("%s: %s\r\n" % (keyword, value)).encode('latin-1', 'strict'))

        if keyword.lower() == 'connection':
            if value.lower() == 'close':
                self.close_connection = True
            elif value.lower() == 'keep-alive':
                self.close_connection = False
```

CodeQL database





CodeQL queries

Vulnzzz

CodeQL detects vulnerabilities in two steps:

https://github.blog/2023-03-31-codeql-zero-to-hero-part-1-the-fundamentals-of-static-analysis-for-vulnerability-research/

① Parses your code into a relational **database** of facts, which contains information about **elements of our source code**, f.ex. classes and functions and relationships between these elements

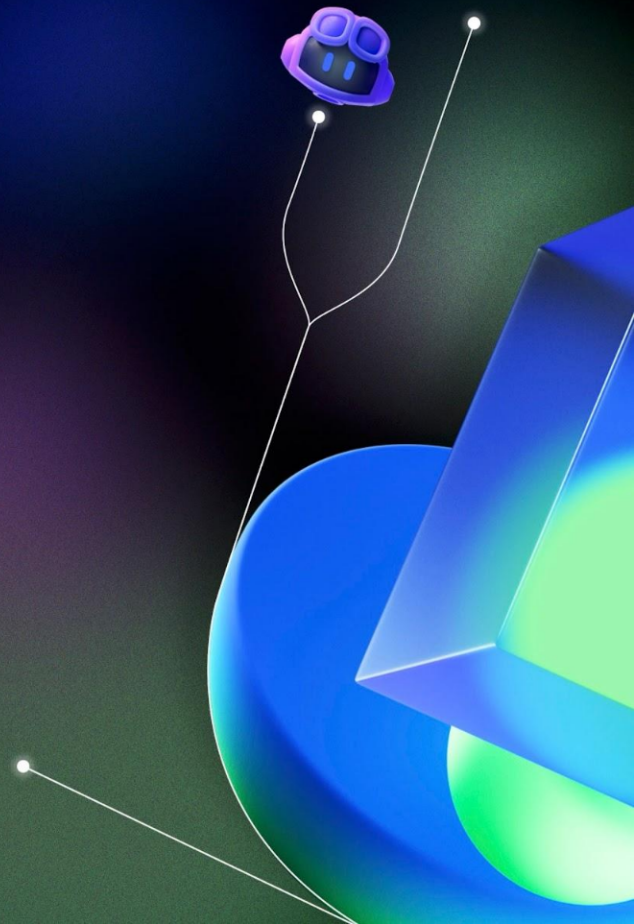
② Runs CodeQL queries on that database, which detect **vulnerable patterns** in your code, e.g. SQL injection, command injection

What is CodeQL query language?

CodeQL is...

- Logical
- Declarative
- Object-oriented
- Read-only
- Equipped with rich standard libraries for analyzing source code
- Toolchain: CLI and IDE

What does a query look like?



Import: lets us reuse logic defined in other libraries

```
import python
```

```
from <type> <name>          //variables used in the query

where <conditions for variables>

select <output>          //results, referring to the variables
```

Query clause: describes what we are trying to find

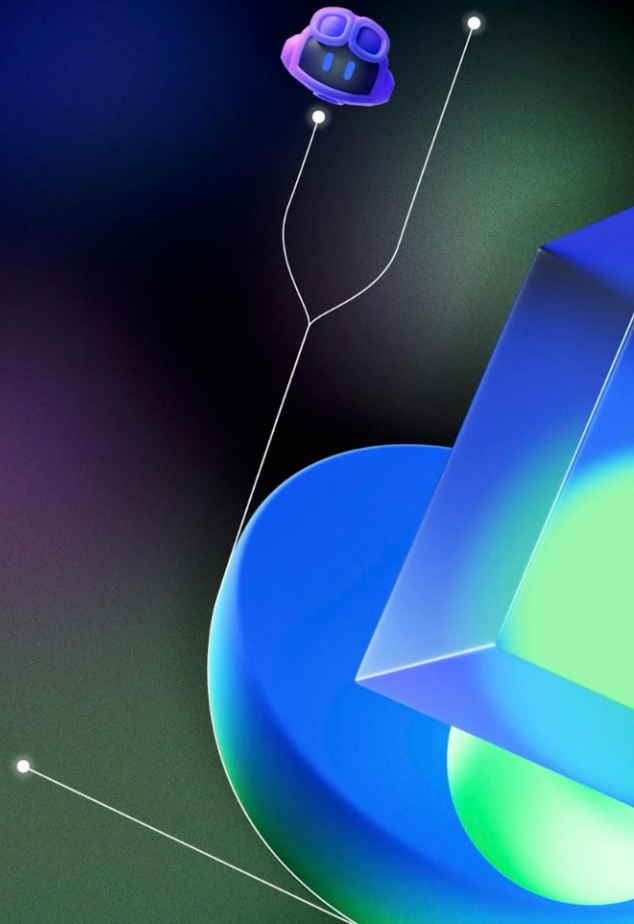
Import: lets us reuse logic
defined in other libraries

```
import python
import semmle.python.ApiGraphs
```

```
from  API::CallNode call
where call = API::moduleImport("os")
        .getMember("system")
        .getACall()
select call, "Call to os.system"
```

Query clause: describes what we are trying to find

Building blocks of a query



Predicates

Like functions, but better!

Create reusable logic and give it a name.

Mini from-where-select

Just a query

```
from API::CallNode call
where
  call = API::moduleImport("os")
  .getMember("system").getACall()
select call, "Call to os.system"
```

Using a predicate

```
predicate isOsSystemSink(API::CallNode call) {
     call = API::moduleImport("os")
     .getMember("system").getACall()
  }


from API::CallNode call
where isOsSystemSink(call)
select call, "Call to os.system"
```

Classes

Describe a set of values.

Using a predicate

```
predicate isOsSystemSink(API::CallNode call) {
     call = API::moduleImport("os")
     .getMember("system").getACall()
  }


from API::CallNode call
where isOsSystemSink(call)
select call, "Call to os.system"
```

Using a class

```
class OsSystemSink extends API::CallNode {
  OsSystemSink() {
    this = API::moduleImport("os")
    .getMember("system").getACall()
     }
}


from API::CallNode call
where call instanceof OsSystemSink
select call, "Call to os.system"
```

Using a predicate

```
predicate isOsSystemSink(API::CallNode call) {
     call = API::moduleImport("os")
     .getMember("system").getACall()
  }


from API::CallNode call
where isOsSystemSink(call)
select call, "Call to os.system"
```

Using a class

```
class OsSystemSink extends API::CallNode {
  OsSystemSink() {
    this = API::moduleImport("os")
    .getMember("system").getACall()
     }
}


from OsSystemSink s
select s, "Call to os.system"
```

Source and sink models in CodeQL

CodeQL has hundreds of types representing sources, sinks, etc. CodeQL checks for a data flow path between these predefined sources and sinks.

When we find a new source or a sink and write a query for them, we say that we **model** sources and sinks in CodeQL.

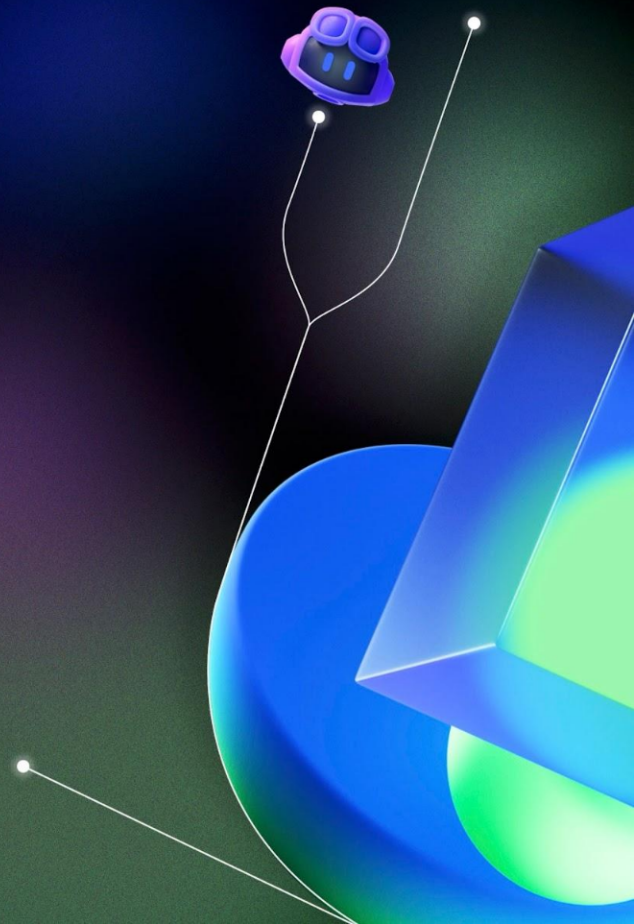
For example, all sources are modelled as `RemoteFlowSource`, which we can query.

```
import os
from flask import Flask, request

app = Flask(__name__)

@app.route("/show-files")
def show_files():
    files = request.args.get('files', '')
    # Attacker could send `; rm -rf /` to remove all files
    os.system("ls " + files)
```

GitHub Secure
Open Source Fund
powered by GitHub Sponsors

Hands-on session

https://gh.io/soss-codeql



GitHub Secure
Open Source Fund
powered by GitHub Sponsors