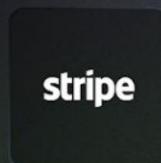
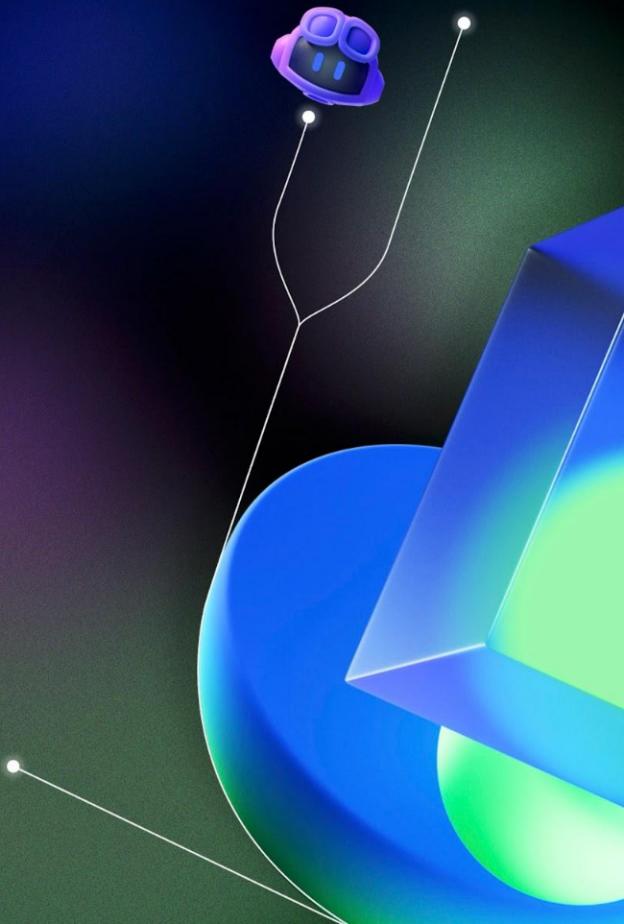


GitHub Secure Open Source Fund

powered by GitHub Sponsors



Module: Finding vulnerabilities with CodeQL



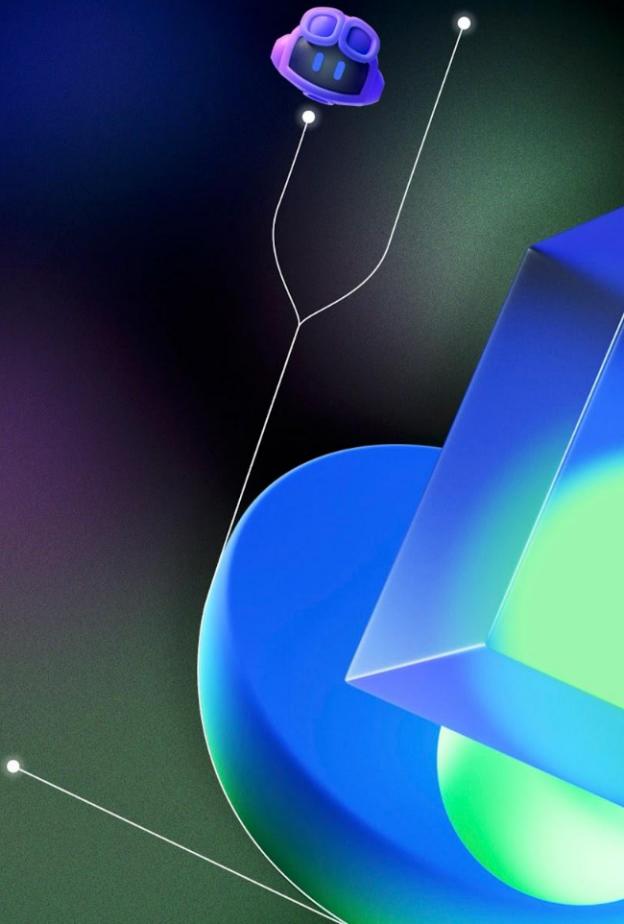
About Me



Sylwia Budzynska
GitHub Security Lab
Security Researcher

GitHub: @sylwia-budzynska

LinkedIn: <https://www.linkedin.com/in/sylwia-budzynska>
securitylab.github.com



Agenda

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

<https://gh.io/soss-codeql>

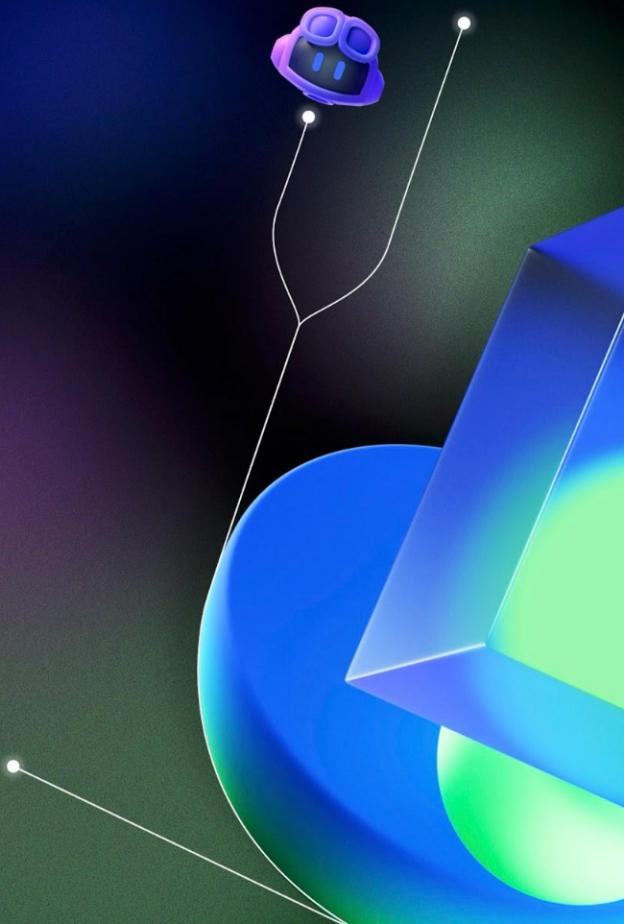


Q: Have you ever found a vulnerability before or worked with fixing one?



How do we find vulnerabilities?

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



Example vulnerability

[CVE-2024-32022, CVE-2024-32026,](#)
[CVE-2024-32025, CVE-2024-32027](#)

```
● ● ●

import gradio as gr
import os

def execute_cmd(folder):
    cmd = f"python caption.py --dir={folder}"
    os.system(cmd)

with gr.Blocks() as demo:
    gr.Markdown("Create caption files for images")
    with gr.Row():
        folder = gr.Textbox(label="Directory to
caption")
        logs = gr.Checkbox(label="Add verbose logs")

    btn = gr.Button("Run")
    btn.click(fn=execute_cmd, inputs=[folder])

if __name__ == "__main__":
    demo.launch(debug=True)
```

Identifying the vuln

This command injection is caused by the user-controlled **folder** (**source**) used directly in a command, in the `os.system` call (**sink**).

Example PoC payload:

```
|| rm -rf /
```

```
import gradio as gr
import os

def execute_cmd(folder):
    cmd = f"python caption.py --dir={folder}"
    os.system(cmd)

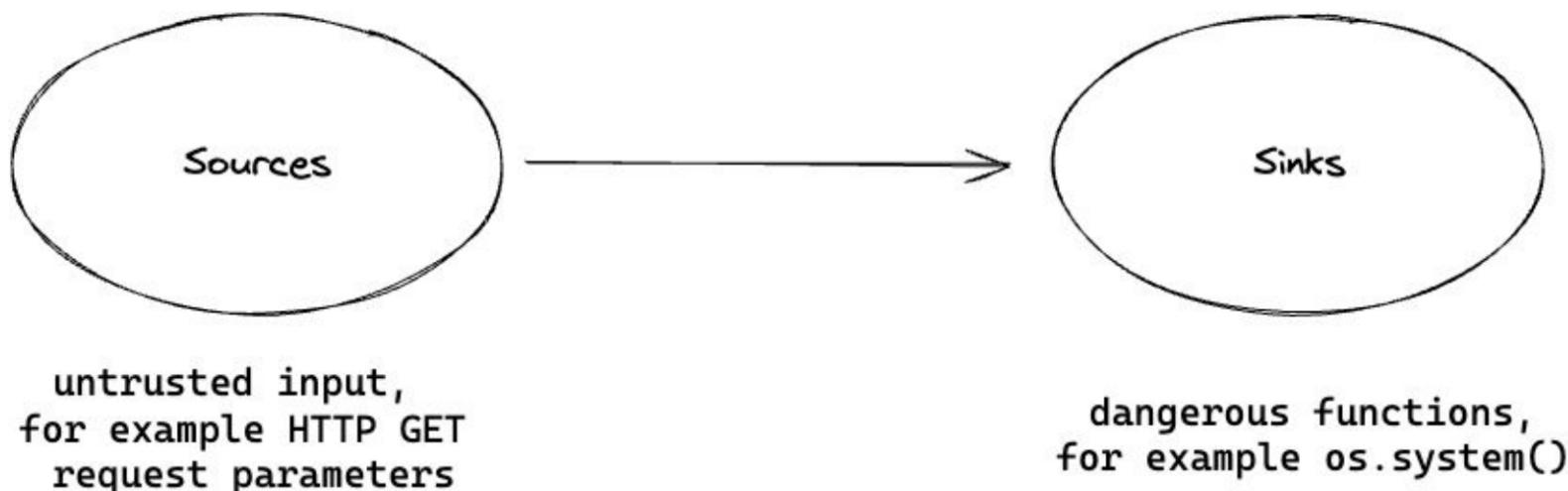
with gr.Blocks() as demo:
    gr.Markdown("Create caption files for images")
    with gr.Row():
        folder = gr.Textbox(label="Directory to
caption")
        logs = gr.Checkbox(label="Add verbose logs")

    btn = gr.Button("Run")
    btn.click(fn=execute_cmd, inputs=[folder])

if __name__ == "__main__":
    demo.launch(debug=True)
```

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?



untrusted input,
for example HTTP GET
request parameters

dangerous functions,
for example `os.system()`

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

CodeQL



Q: Have you ever used a static analysis security tool?

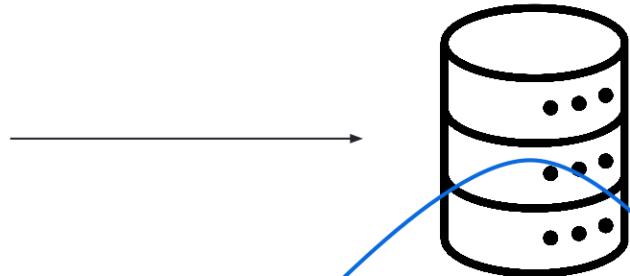


How CodeQL works

Source code

```
004     message":  
005         if not hasattr(self, '_headers_buffer'):  
006             self._headers_buffer = []  
007             self._headers_buffer.appendf("%s %d %s\r\n" %  
008                 (self.protocol_version, code, message)).encode(  
009                     'latin-1', 'strict'))  
010  
011     def send_header(self, keyword, value):  
012         """Send a MIME header to the headers buffer."""  
013         if self.request_version == "HTTP/0.9":  
014             if not hasattr(self, '_headers_buffer'):  
015                 self._headers_buffer = []  
016             self._headers_buffer.appendf(  
017                 "%s: %s\r\n" % (keyword, value)).encode('latin-1', 'strict')  
018  
019         if keyword.lower() == 'connection':  
020             if value.lower() == 'close':  
021                 self.close_connection = True  
022             elif value.lower() == 'keep-alive':  
023                 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

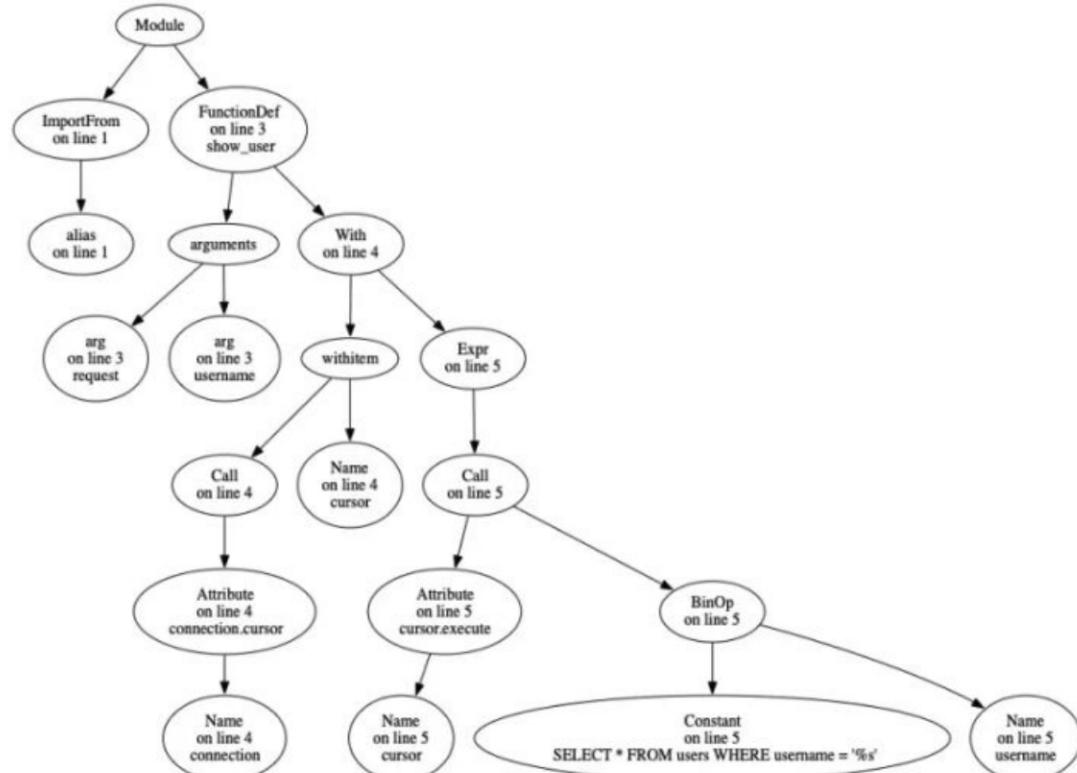
②

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

How CodeQL works pt.2

```
Module(  
    body=[  
        ImportFrom(  
            lineno= 2,  
            col_offset= 0,  
            end_lineno= 2,  
            end_col_offset= 32,  
            module= 'django.db',  
            names=[ alias(lineno= 2, col_offset= 22, end_lineno= 2,  
end_col_offset= 32, name='connection', asname=None) ],  
            level= 0,  
        ),  
        FunctionDef(  
            lineno= 5,  
            col_offset= 0,  
            end_lineno= 7,  
            end_col_offset= 78,  
            name= 'show_user',  
            args= arguments(  
                posonlyargs=[],  
                args=[  
                    arg(lineno= 5, col_offset= 14, end_lineno= 5,  
end_col_offset= 21, arg='request', annotation=None, type_comment=None),  
                    arg(lineno= 5, col_offset= 23, end_lineno= 5,  
end_col_offset= 31, arg='username', annotation=None, type_comment=None),  
                ],  
                vararg=None,  
                kwonlyargs=[],  
                kw_defaults=[],  
                kwarg=None,  
                defaults=[],  
            ),  
        ),  
    ],  
)
```

```
. from django.db import connection  
. .  
. def show_user(request, username):  
.     with connection.cursor() as cursor:  
.         cursor.execute("SELECT * FROM users WHERE username = '%s'" % username)
```

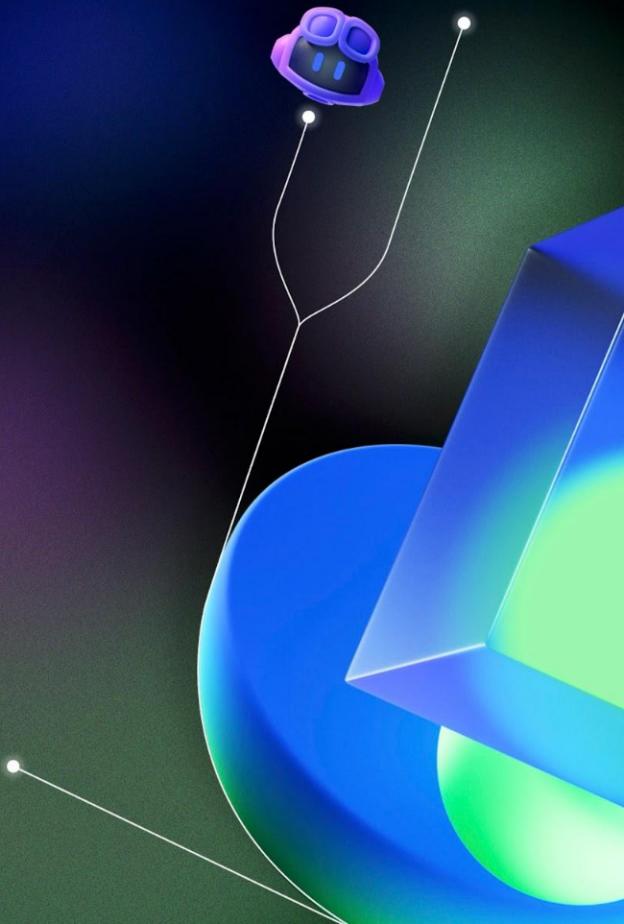


**Q: Have you ever written
a CodeQL query?**



Workshop setup

<https://gh.io/soss-codeql>



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

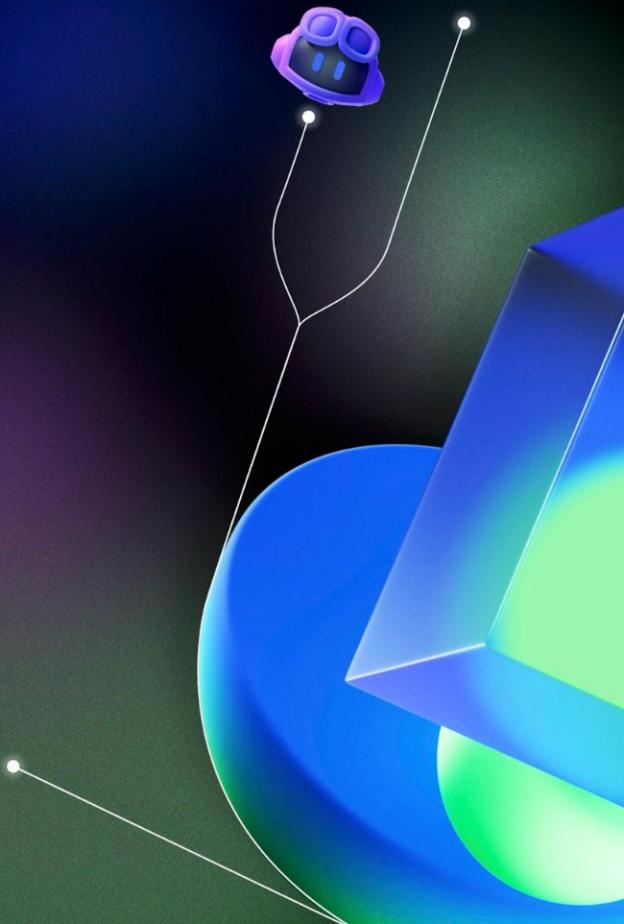
```
import python
import semmle.python.ApiGraphs
```

```
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.

Command execution sinks are modelled as

```
● ● ●

import gradio as gr
import os

def execute_cmd(folder):
    cmd = f"python caption.py --dir={folder}"
    os.system(cmd)

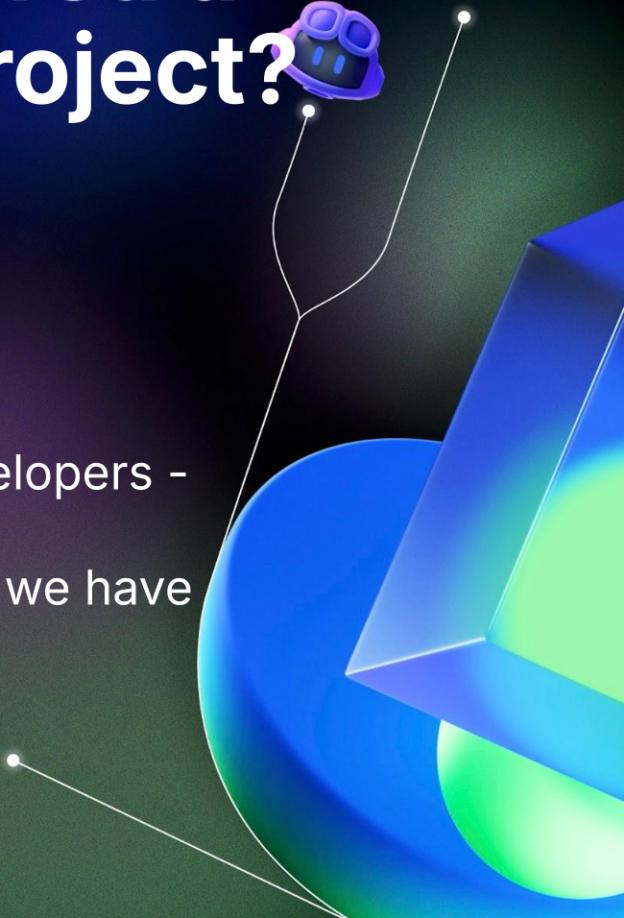
with gr.Blocks() as demo:
    gr.Markdown("Create caption files for images")
    with gr.Row():
        folder = gr.Textbox(label="Directory to
caption")
        logs = gr.Checkbox(label="Add verbose logs")

    btn = gr.Button("Run")
    btn.click(fn=execute_cmd, inputs=[folder])

if __name__ == "__main__":
    demo.launch(debug=True)
```

Q: What could be considered a source or a sink in your project?

- For web applications:
 - CodeQL will find most sources and sinks
- For libraries:
 - What endpoints are we exposing to the developers - could they be considered a source or a sink
 - What potentially dangerous functionality do we have internally in our code





Thank you

