
Static-Analysis
Now you're playing with power!
Philippe Arteau
Security Researcher at GoSecure
04/11/2017
Hackfest 2017

# Agenda

- Definitions
- Motivation
- Different levels of sophistication
- Internals and applications
  - AST based analyzer
  - Taint analysis
  - Continuous integration
  - Automate code refactoring
- Additional considerations

- Expect multiple demos !

# Who Am I ?

- Philippe Arteau
- Security Researcher at GoSecure
- Open-source developer
  - Find Security Bugs (SpotBugs - Static Analysis for Java)
  - Security Guard (Roslyn – Static Analysis for .NET)
  - Burp and ZAP Plugins (Retire.js, CSP Auditor)
- Volunteer for the nsec conference and former trainer

# Definition

# Definition

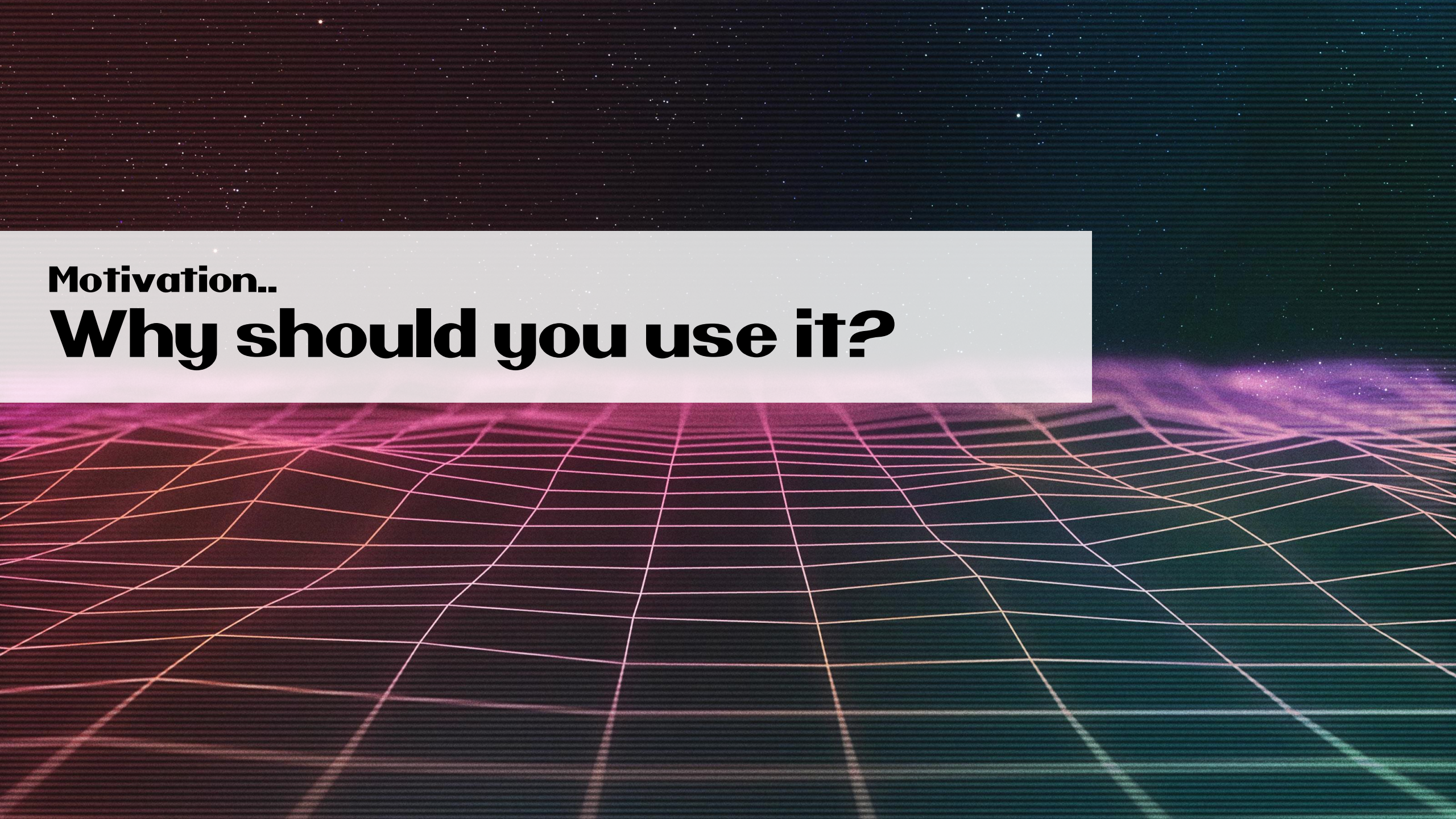
Static Analysis is

- “The analysis of computer software that is performed **without actually executing** programs”

In the context of this presentation

- Finding **vulnerabilities** by looking at the **code**

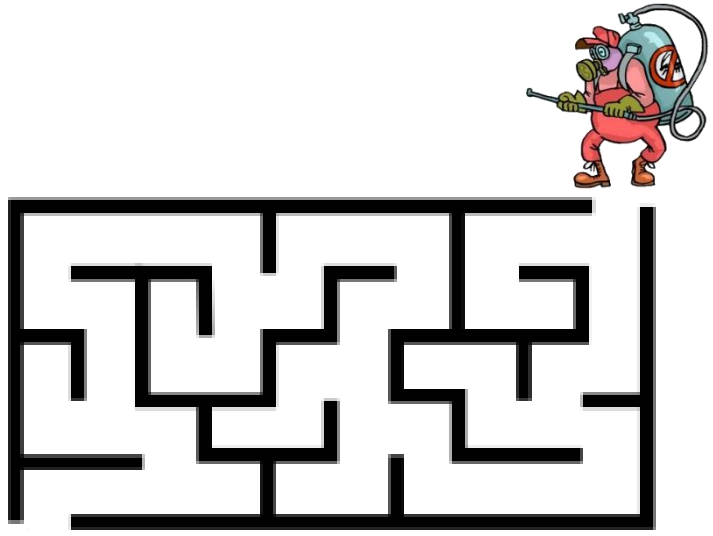
(with the help of tools)


Motivation..
Why should you use it?

# Motivation

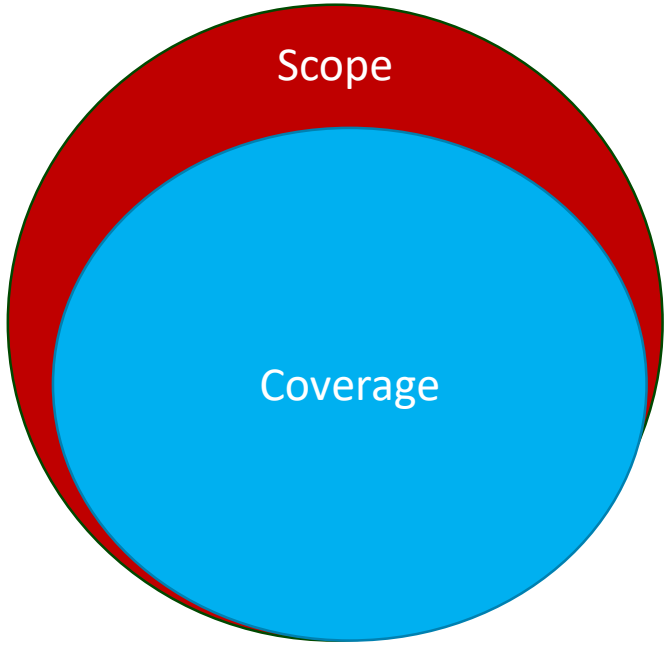
Why would I use Static Analysis?

- High coverage of the **application code**
- Quick discovery in the development lifecycle
- Identification of the source of the problem not just the symptoms



# Limitations

- Low coverage of **the infrastructure code**
- False positives
  - Exploitability is always an estimate
- Many vulnerability classes are not covered
  - Misconfigurations
  - CSRF vulnerabilities
  - Logic flaws

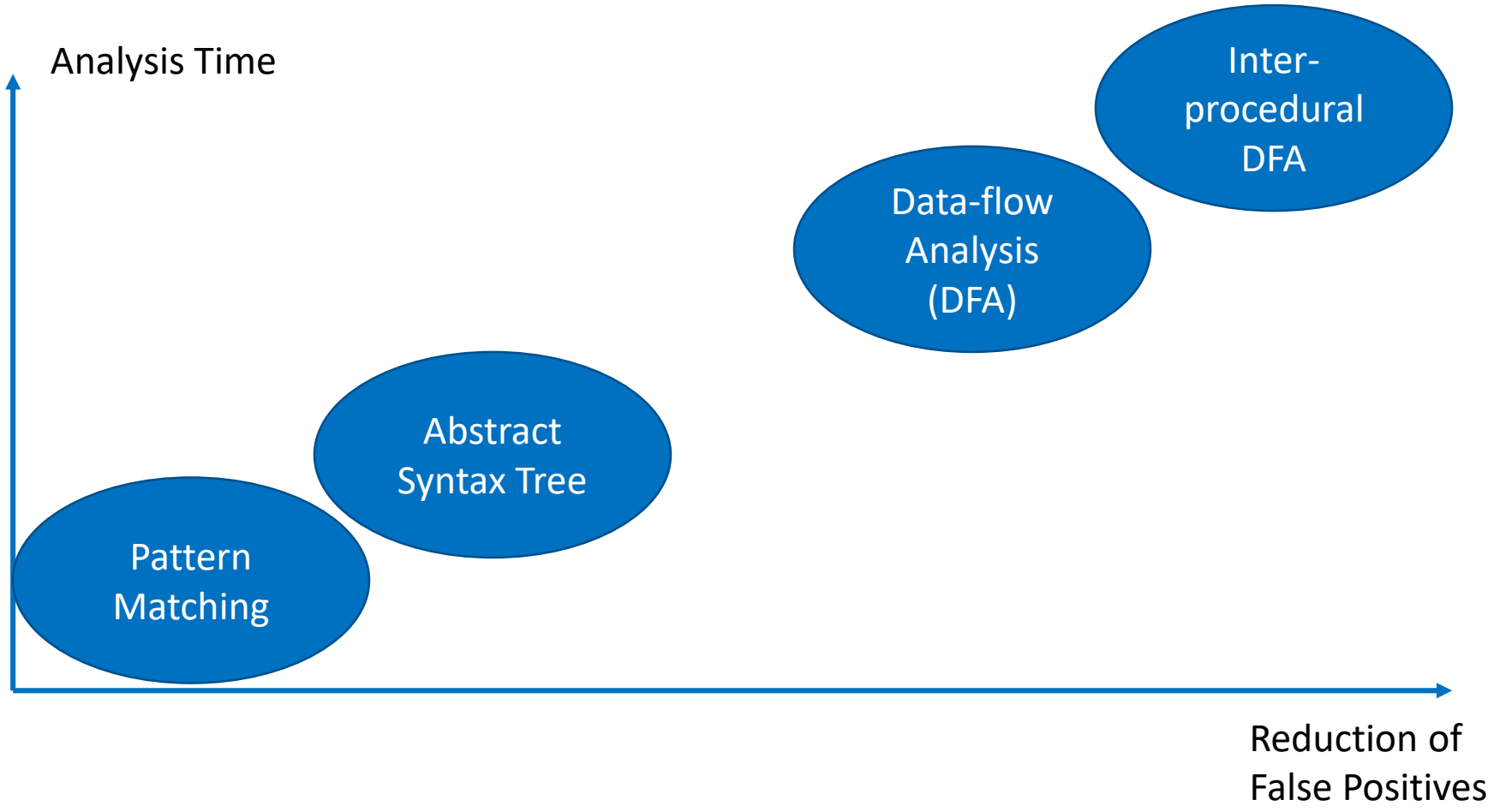


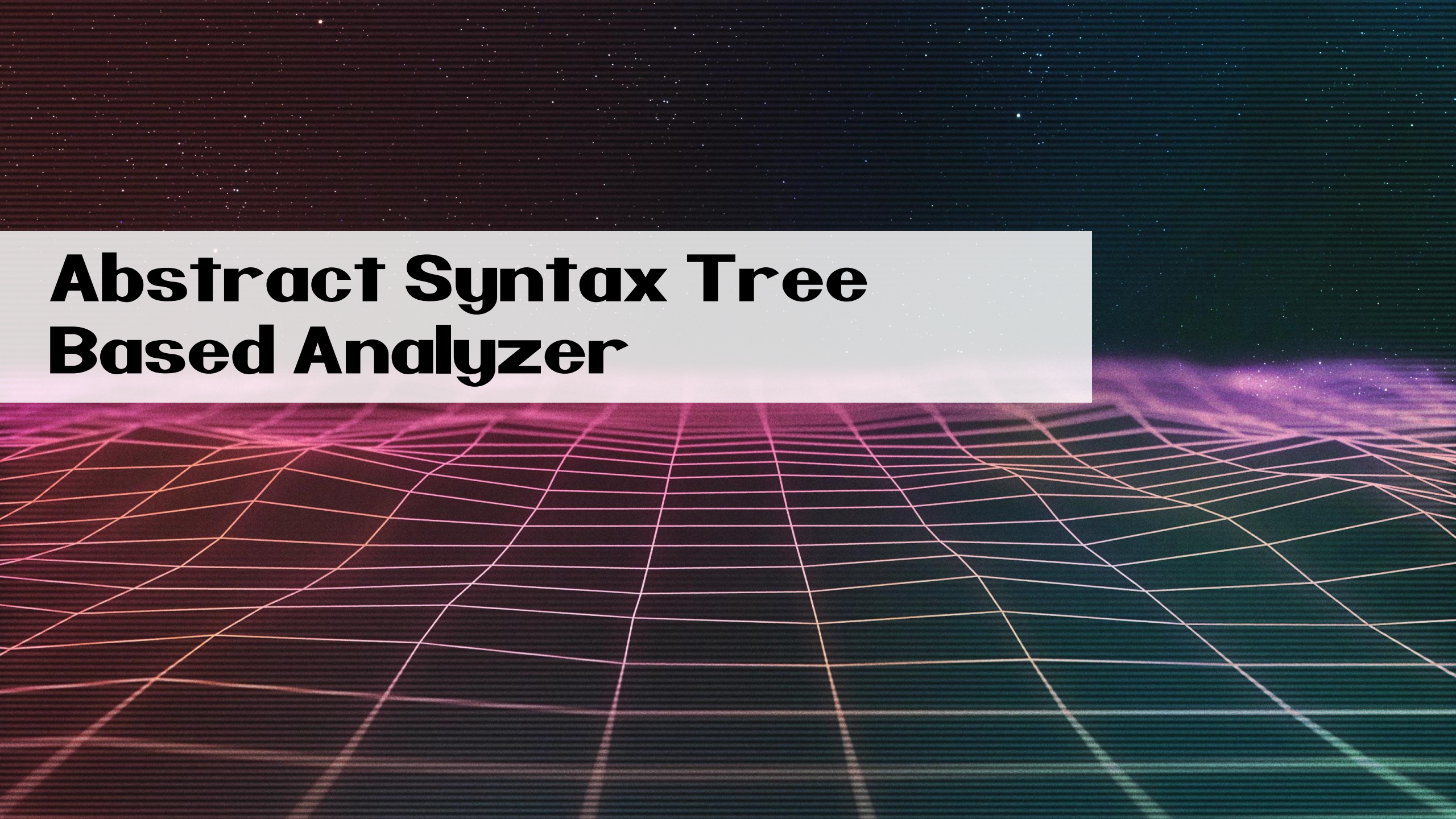
# Different levels of sophistication

# Techniques

| Techniques | Description / Behavior |
| --- | --- |
| Pattern Matching | • Analog to grep |
| Abstract Syntax Tree | • Parsing of the code base<br>• Inline heuristic |
| Data-Flow Analysis | • Simulation of the execution<br>• Tainted analysis |
| Inter-procedural Data-Flow Analysis | • Taint tracking across function (procedure) |

# Techniques overview




Abstract Syntax Tree
Based Analyzer

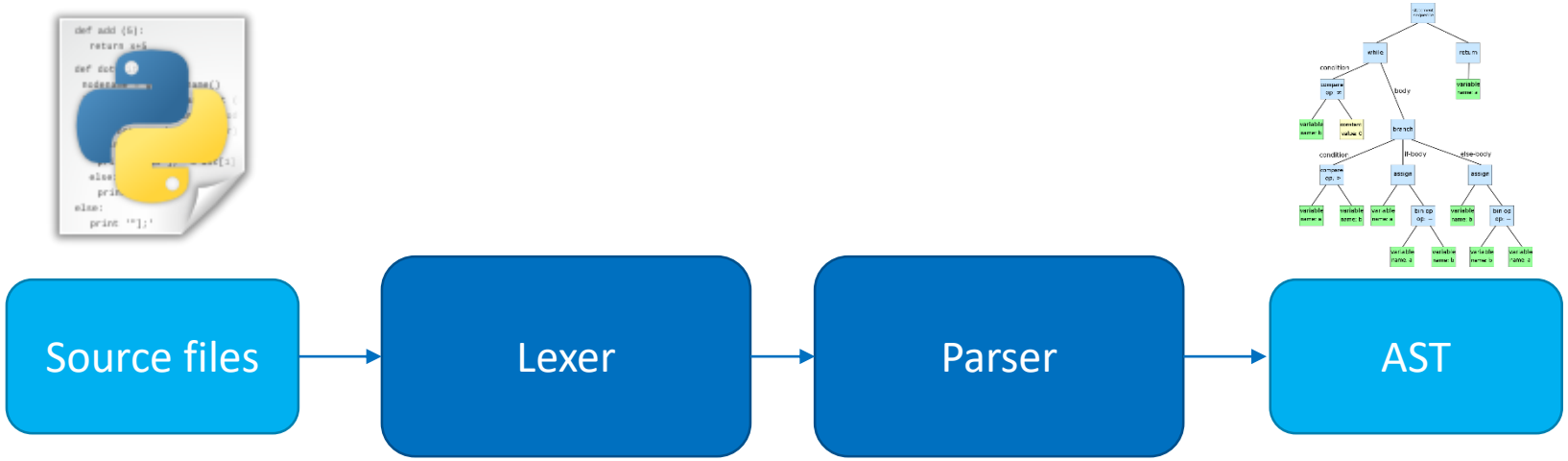
# Demonstration Bandit


- https://github.com/openstack/bandit

# Abstract Syntax Tree

## Definition

Tree representation of the abstract syntactic structure of the source code

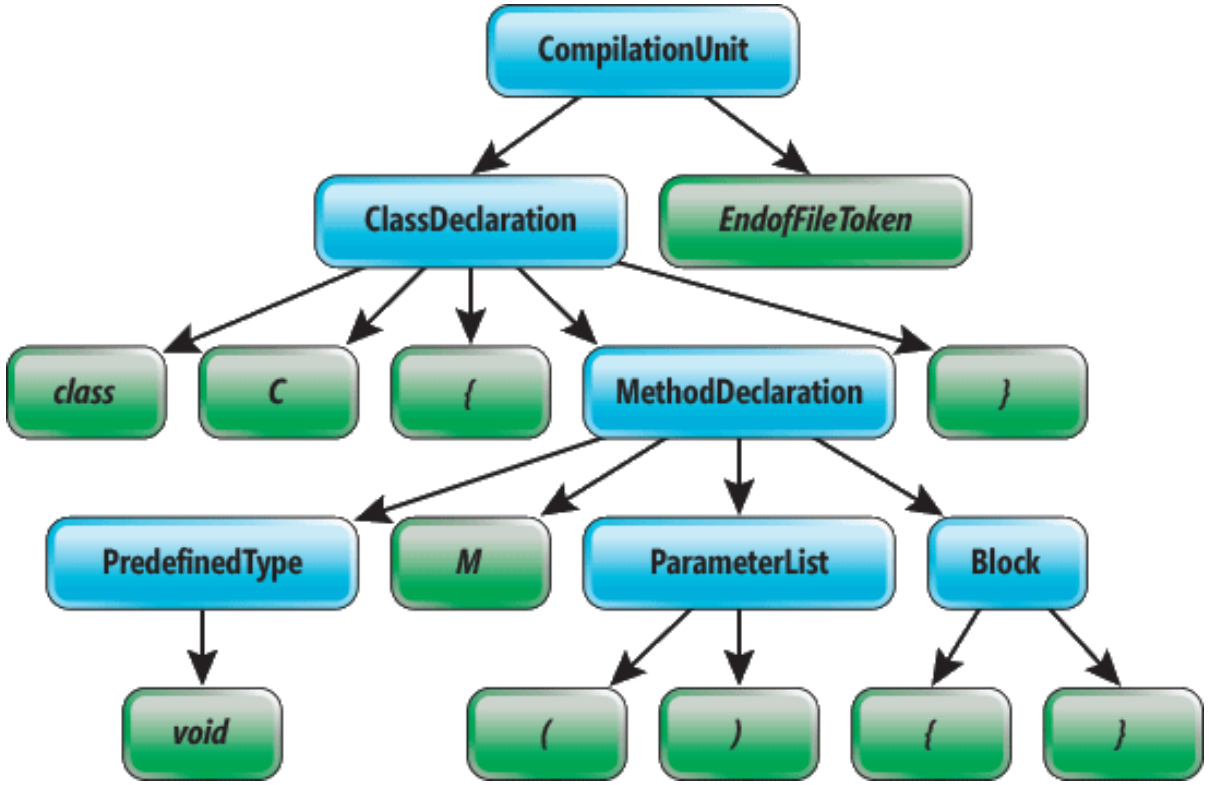


# Abstract Syntax Tree

Abstract Syntax Tree main features:

- Handling of spacing and nested method calls
  - Take away the complexity regex to handle spaces, indentation, new lines, etc.
- Resolution of types (optional – depends of the language)
  - Allow matching of the class name **not just method**
- Possibility to do some heuristic on the inline value
  - This means less false positives

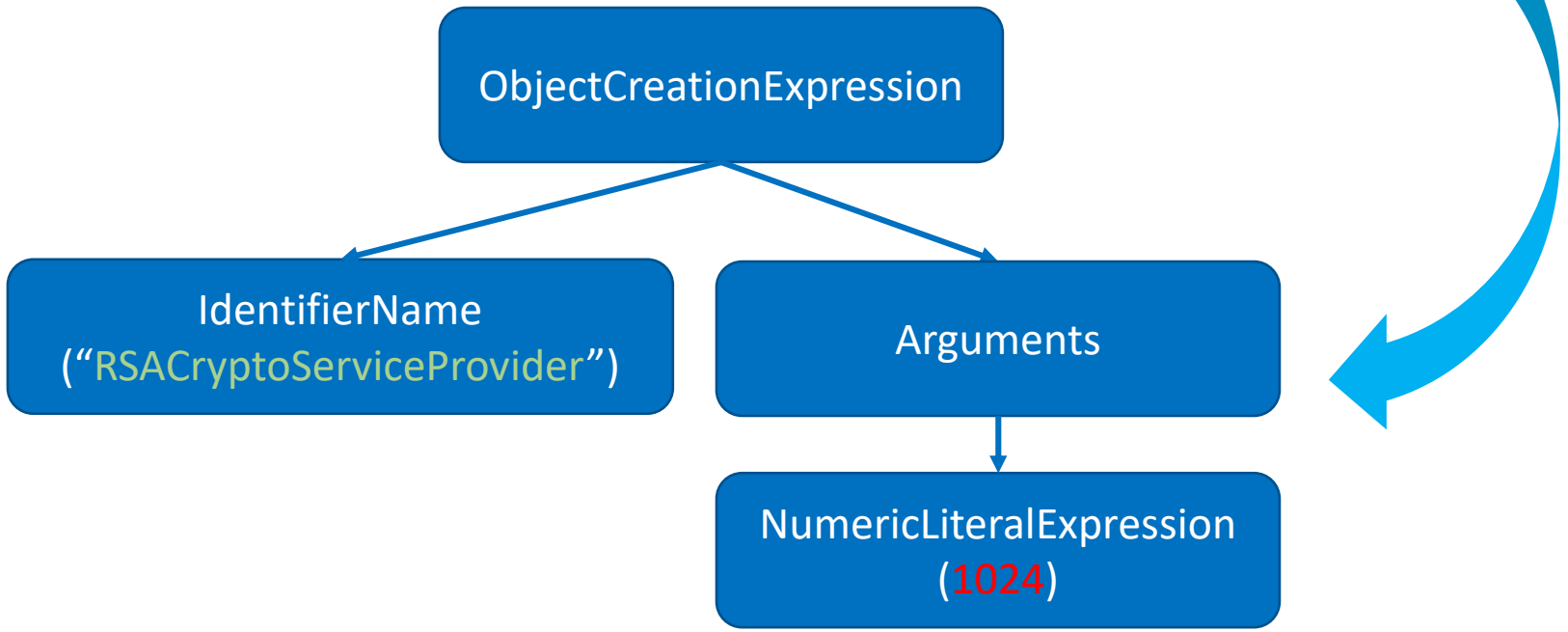
# Abstract Syntax Tree

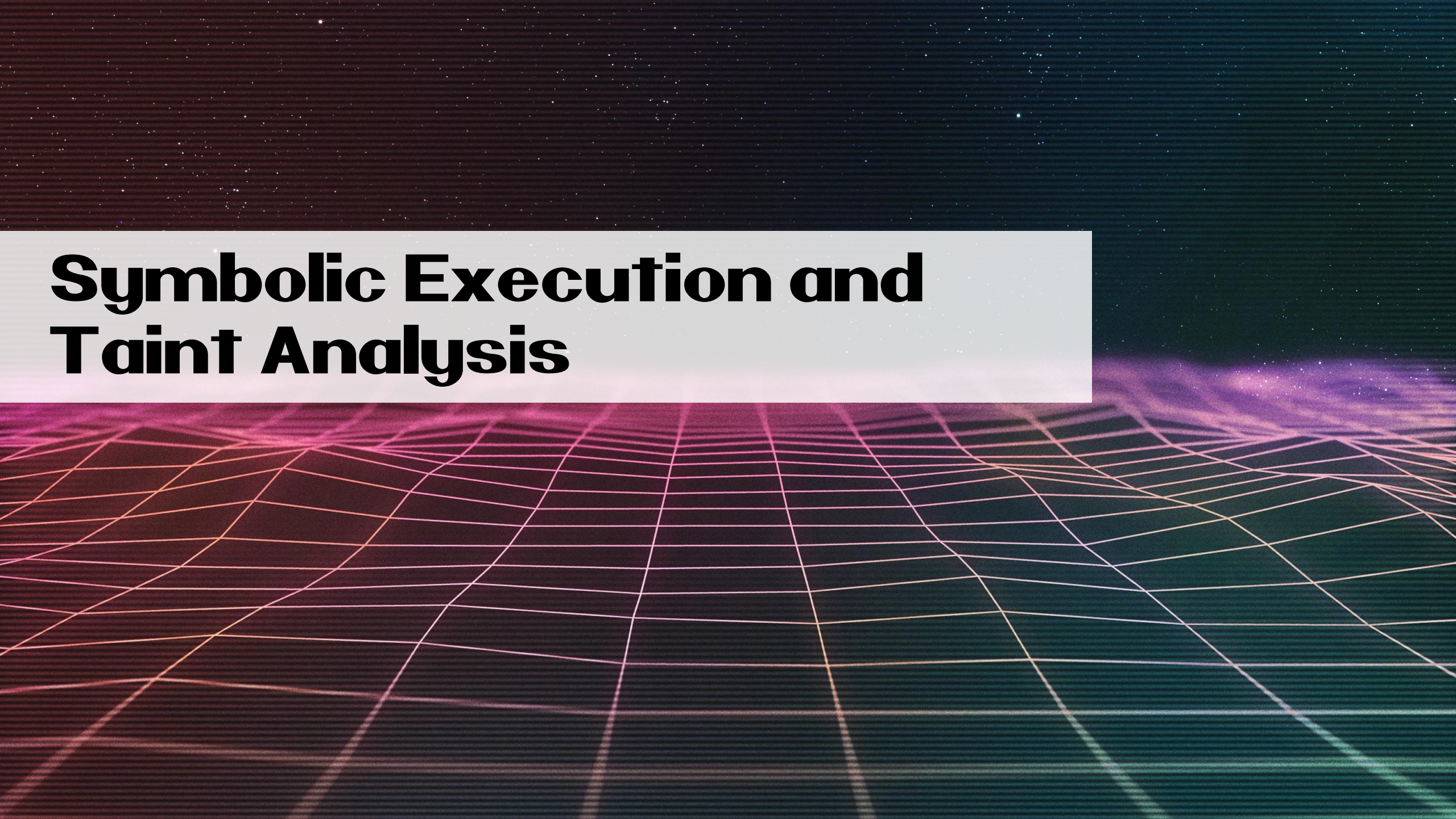



Roslyn AST : https://msdn.microsoft.com/en-us/magazine/dn904670.aspx

# Basic AST Analysis

```
var rsa = new RSACryptoServiceProvider(1024);
```




Symbolic Execution and
Taint Analysis

```
def quiz(int a, int b) {
    c = a*6
    if(c + b < 50) {
        if(a-40 == b) {
            if(a + b > 0) {
                //How to get here?
            }
        }
    }
}
```

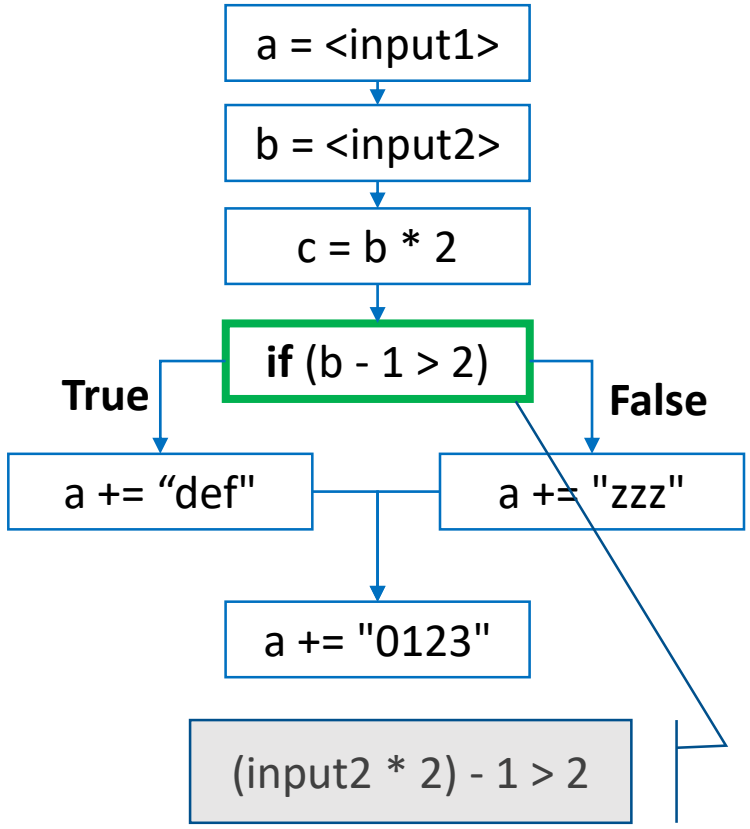
How can we find values need to a reach specific path? (programmatically)

# Symbolic execution

## Symbolic execution

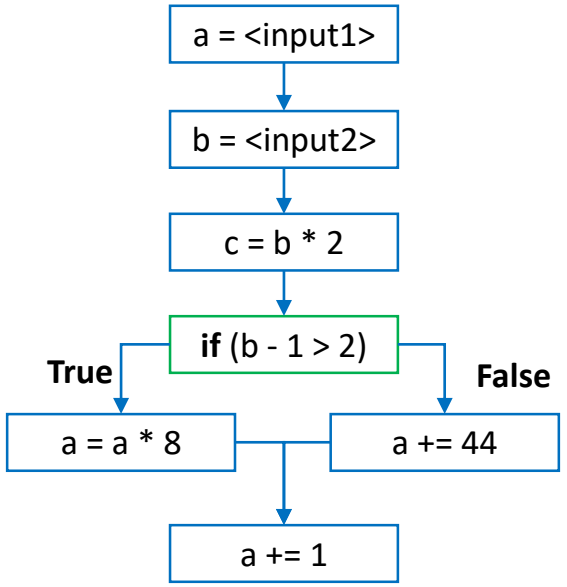
Simulating the code execution using expression rather than concrete data

To determine how to reach specific code location, conditions must be transform in mathematical equation.




Reference : [Symbolic Execution for Software Testing: Three Decades Later]

# Symbolic in action




| a | b | c |
|---|---|---|
| input1 | | |
| input1 | input2 | |
| input1 | input2 | input2*2 |

(b - 1 > 2) == true

| a | b | c |
|---|---|---|
| input1*8 | input2 | input2*2 |
| input1*8+1 | input2 | input2*2 |

(b - 1 > 2) == false

| a | b | c |
|---|---|---|
| input1+44 | input2 | input2*2 |
| input1+45 | input2 | input2*2 |

Symbolic execution mainly focuses on resolving **input values** to **reach a specific path**

Many vulnerabilities analyzers need to monitor **validation state** of variables. One additional concept is needed…

Taint analysis

# False Positive vs Real Positive

- Safe

```
a = "userId = "
b = "1"
c = a + b

User.applyFilter(c)
```

- Unsafe

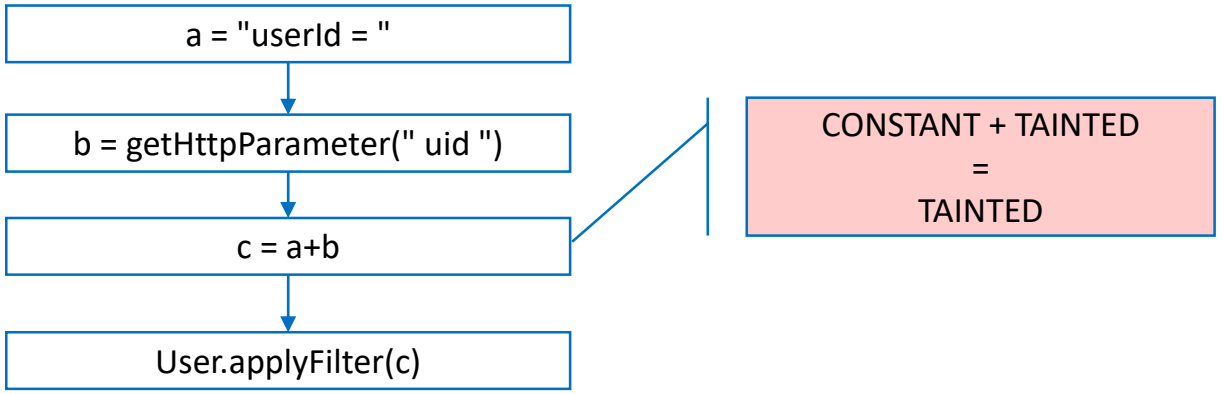
```
a = "userId = "
b = getHttpParameter("uid")
c = a + b

User.applyFilter(c)
```

How to avoid reporting an issue for the left code sample?

# Taint analysis in Find Security Bugs

## Pseudo-code evaluate




## State of symbolic variables

| a | b | c |
|---|---|---|
| CONSTANT | | |
| CONSTANT | TAINTED | |
| CONSTANT | TAINTED | TAINTED |

# Taint analysis in Find Security Bugs

Base state

- **Tainted** : Unsafe user input
- **Unknown** : Value from unknown source. It could be coming from user input
- **Safe** : Dynamic value from a safe source
- **Constant** : Hardcoded value

Context specific state (tags):

- XSS Safe, SQL Safe, XML Safe, URL Safe, etc.

# Demo Android APK analysis

- Tools available


dex2jar


# Obstacles of Symbolic Execution

```
class Sample {
    def sql = new Sql(datasource)

    def getUserById(int userId) {
        return getUserId(userId)
    }

    def getUserById(String userId) {
        return sql.execute("SELECT * FROM Users WHERE uid="+userId)
    }
}
```

Is this code vulnerable?

# More Obstacles

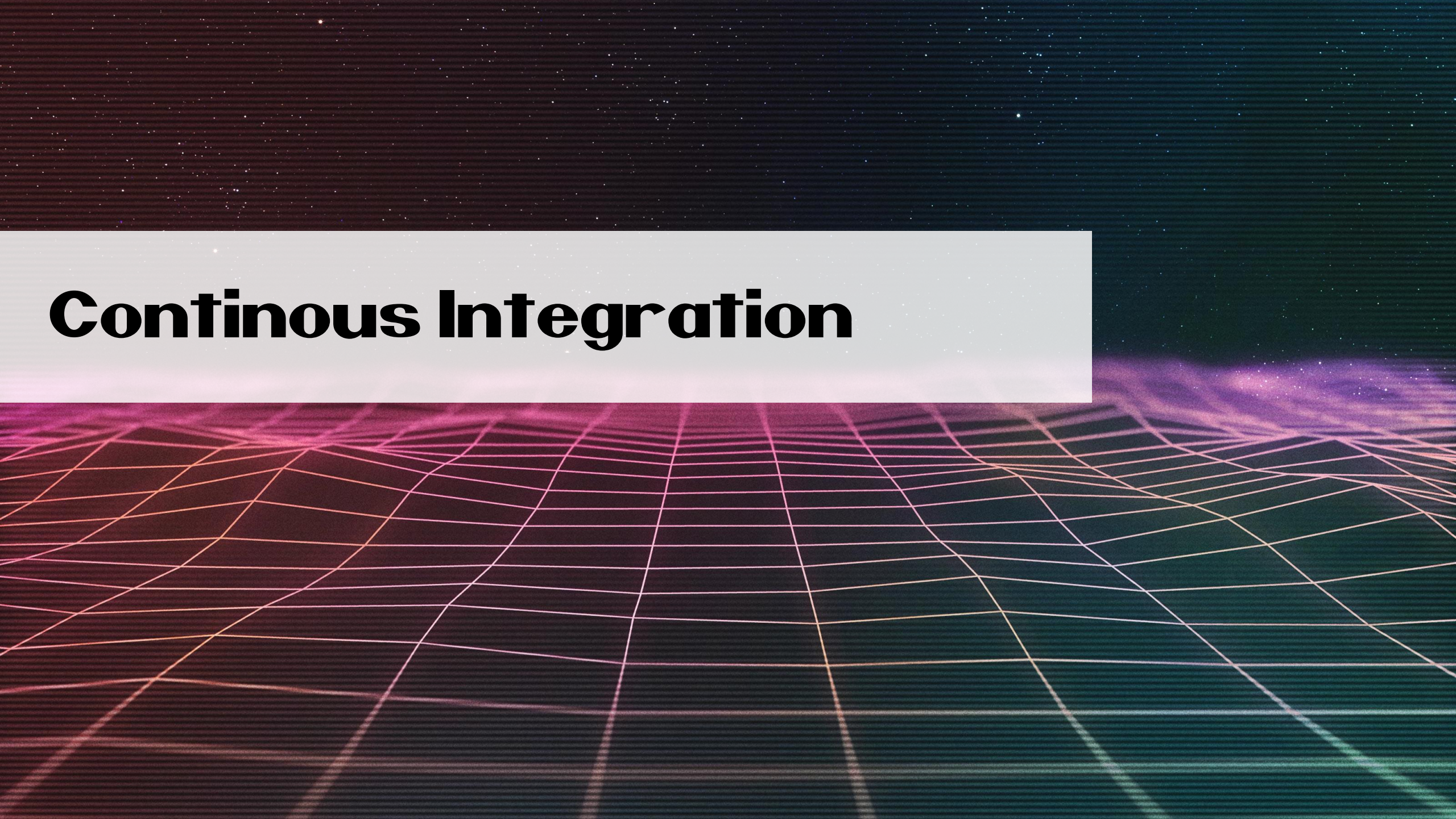
Other obstacles that static analyzers must consider:

 Reflection

 Dependency injection

 Second order vulnerability

 Encapsulation


Continous Integration

# Brakeman CLI

Before continuing .. Here's a new tool that analyze Ruby applications.

Brakeman

- Target mainly Rails API
- 67 rules and growing


- https://brakemanscanner.org/

# Continuous integration

Continuous Integration (CI):

- The practice of merging all developer working copies to a shared mainline several times a day.

- The most basic form will include compiling the application
  - Additional tasks such as running tests and code analysis can be added

- Most static-analysis tool integrate with Continuous integration

# Deployment stategy

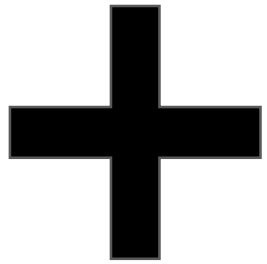

- Usually implemented in this order.
- One deployment does not replace another

# Continuous integration in action

- Demonstration with Brakeman ran from a Jenkins instance

- Job configuration
  - Brakeman command
  - Post Build Jenkins Plugin





Jenkins

# Continuous integration : Jenkins + Docker

- How easy can it be to deployed ?

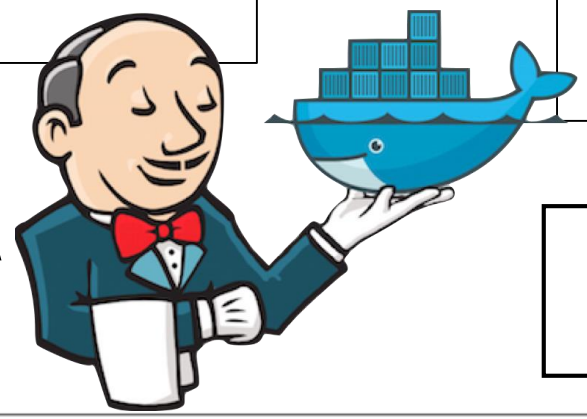
**Dockerfile**

```
FROM jenkins/jenkins:lts

USER root
RUN apt-get update &&
      apt-get install -y ruby rubygems &&
      gem install brakeman
USER jenkins
```

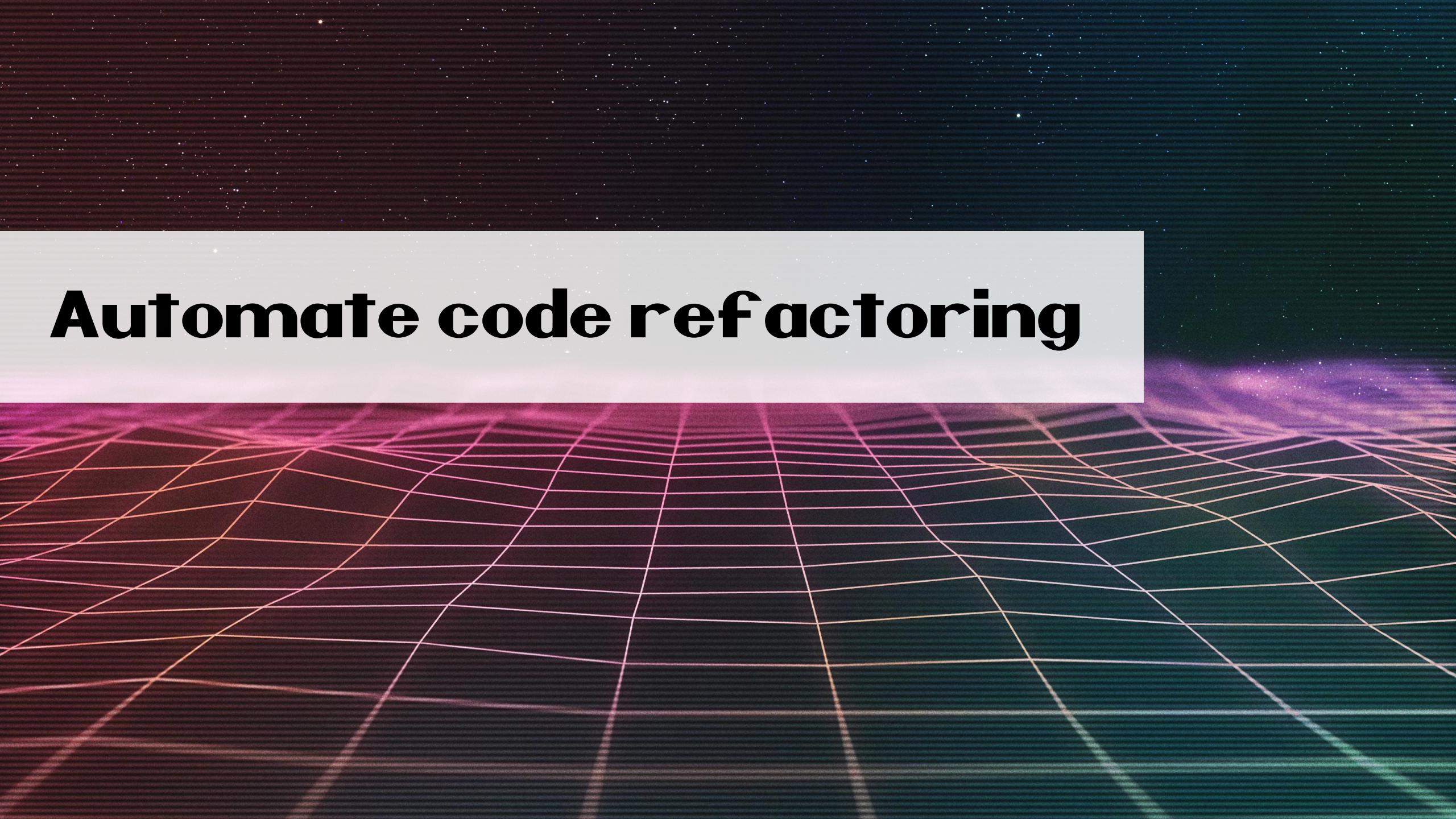
**docker-compose.yml**

```
version: '2'
services:
  jenkins:
    build: .
    ports:
      - 8080:8080
    volumes:
      - ./jenkins_home:/var/jenkins_home
```



**Customize the package available from Jenkins Jobs**

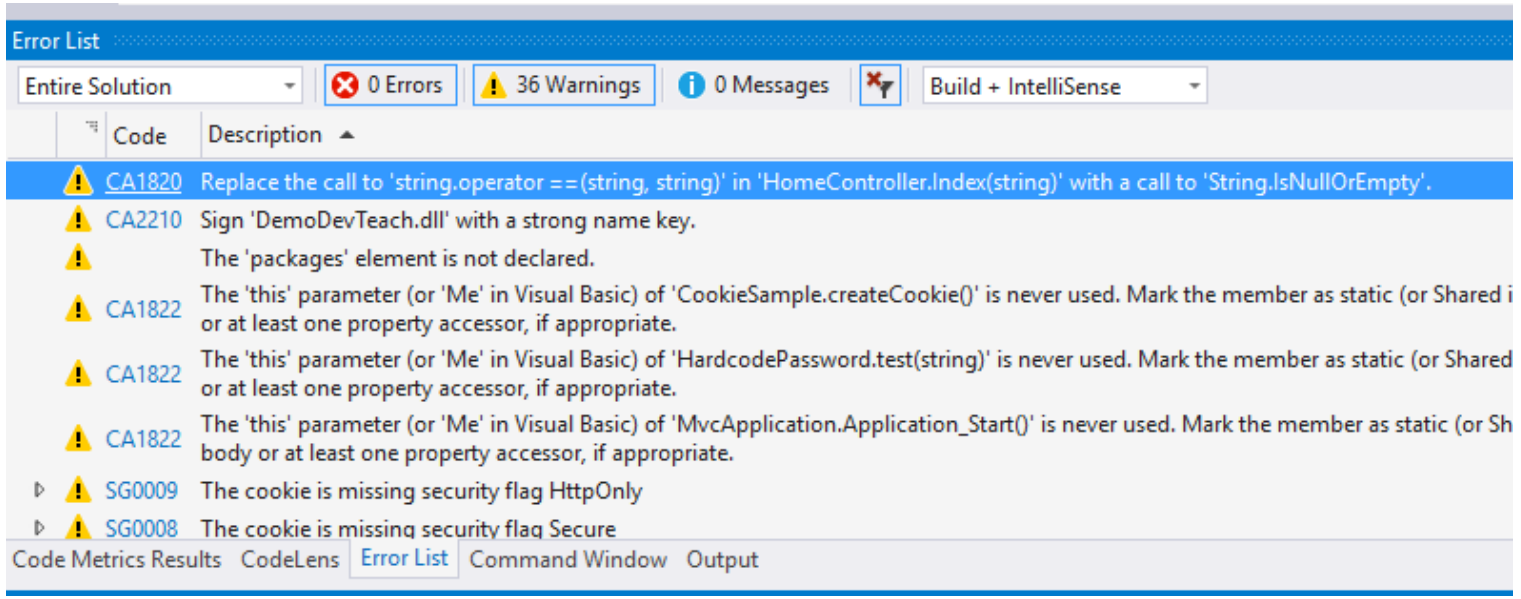
**Mount main Jenkins folder for easy backup and migration**


Automate code refactoring

# Automate code refactoring

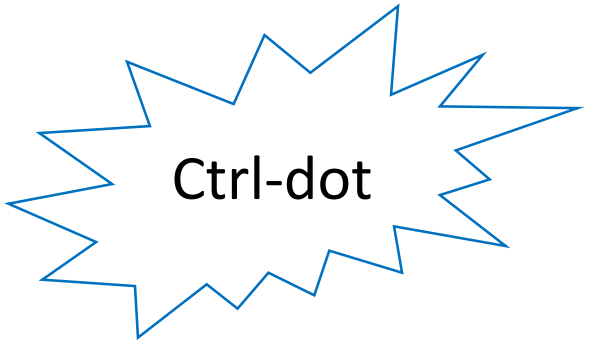
- Identifying bugs and vulnerabilities is nice but…

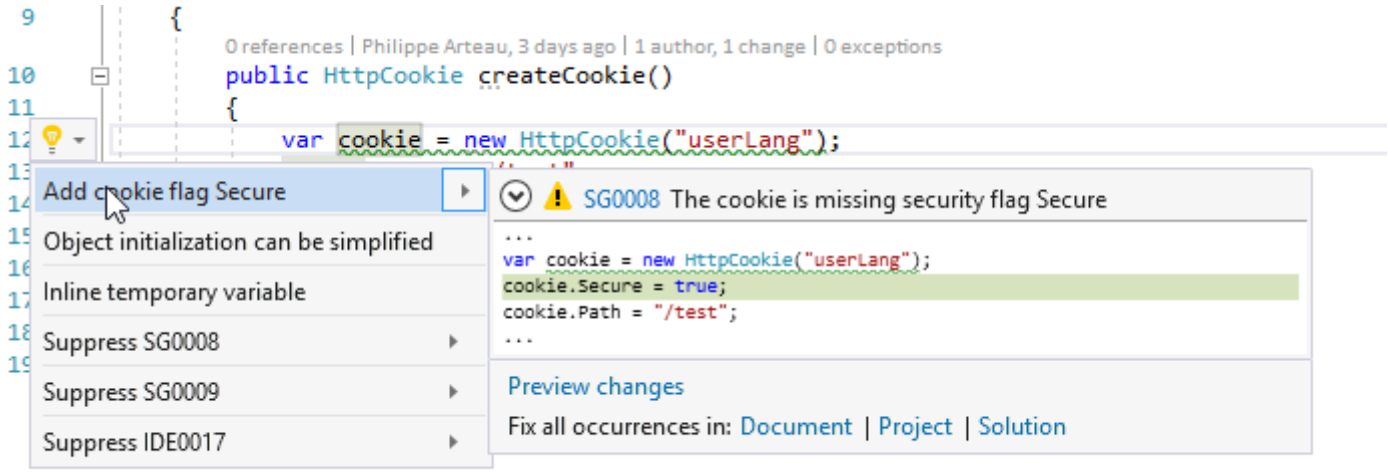
```
0 references | Philippe Arteau, 3 days ago | 1 author, 1 change
public class HomeController : Controller
{
    0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
    public ActionResult Index(string input)
    {
        if (input == "") {
            return View();
        }
```



# Automate code refactoring

- Providing fix is even better!




- Some vulnerabilities require high-level understanding of the application.

# Additional considerations

# How to evaluate tools?

- WASC Static-Analysis Technologies Evaluation Criteria

Samples

- Juliet Test Suite (Java and C++)
- OWASP Benchmark (Java)
- Used vulnerable applications
  - OWASP_Vulnerable_Web_Applications_Directory_Project
  - See Juliet Test Suite Page
- Make your own vulnerable samples
  - Required good security expertise

# Building your own tools

- Do not reinvent the wheel
  - Reuse existing static analysis tools (if available)
  - Search for more than one tool for comparison
  - Reuse existing lexer/parser libraries
- Thinking about the maintenance of your custom rules
  - Do you have the time to maintain those?
  - Will your colleague be able to troubleshoot them?

# Questions ?

**Contact**

parteau@gosecure.ca

gosecure.net/blog/

@h3xStream @GoSecure_Inc

# References

# Tools Presented

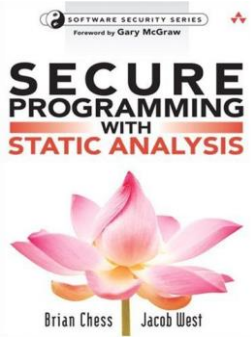
- Openstack Bandit (Python)
- Brakeman (Ruby)
- Find Security Bugs (Java, Scala, Groovy)
- .NET Security Guard (C# and VB.net)

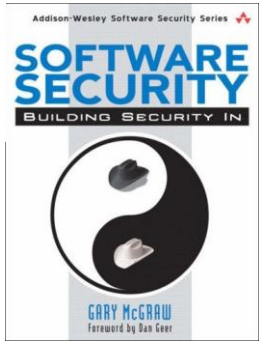
# Useful resources

- NIST SAMATE Project :
  - Source Code Security Analyzers
  - Byte Code Scanners

Books

- Brian Chess et Jacob West, **Secure Programming with Static Analysis**, 2007, Addison-Wesley
- Gary McGraw, **Software Security: Building Security In**, 2006, Addison-Wesley



# Samples for Tools evaluation

Samples

- Juliet Test Suite (Java and C++)
  - https://samate.nist.gov/SRD/testsuite.php
- OWASP Benchmark
  - https://github.com/OWASP/benchmark
- Used vulnerable applications
  - https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project#tab=Off-Line_apps

Criteria

- WASC Static-Analysis Technologies Evaluation Criteria