

Software detective

Python in security research

Przemek Samsel, 19.08.25

Whoami

- Przemek
- Tietoevry
- Network security



Agenda

- Setup
- Security research
- Methodology
- Techniques
- Tools
- Case study



Security Research

What is it

- Understand digital threats & attacks
- Discover vulnerabilities
- Improve system defenses



Security Research

How to start

- **Build knowledge**

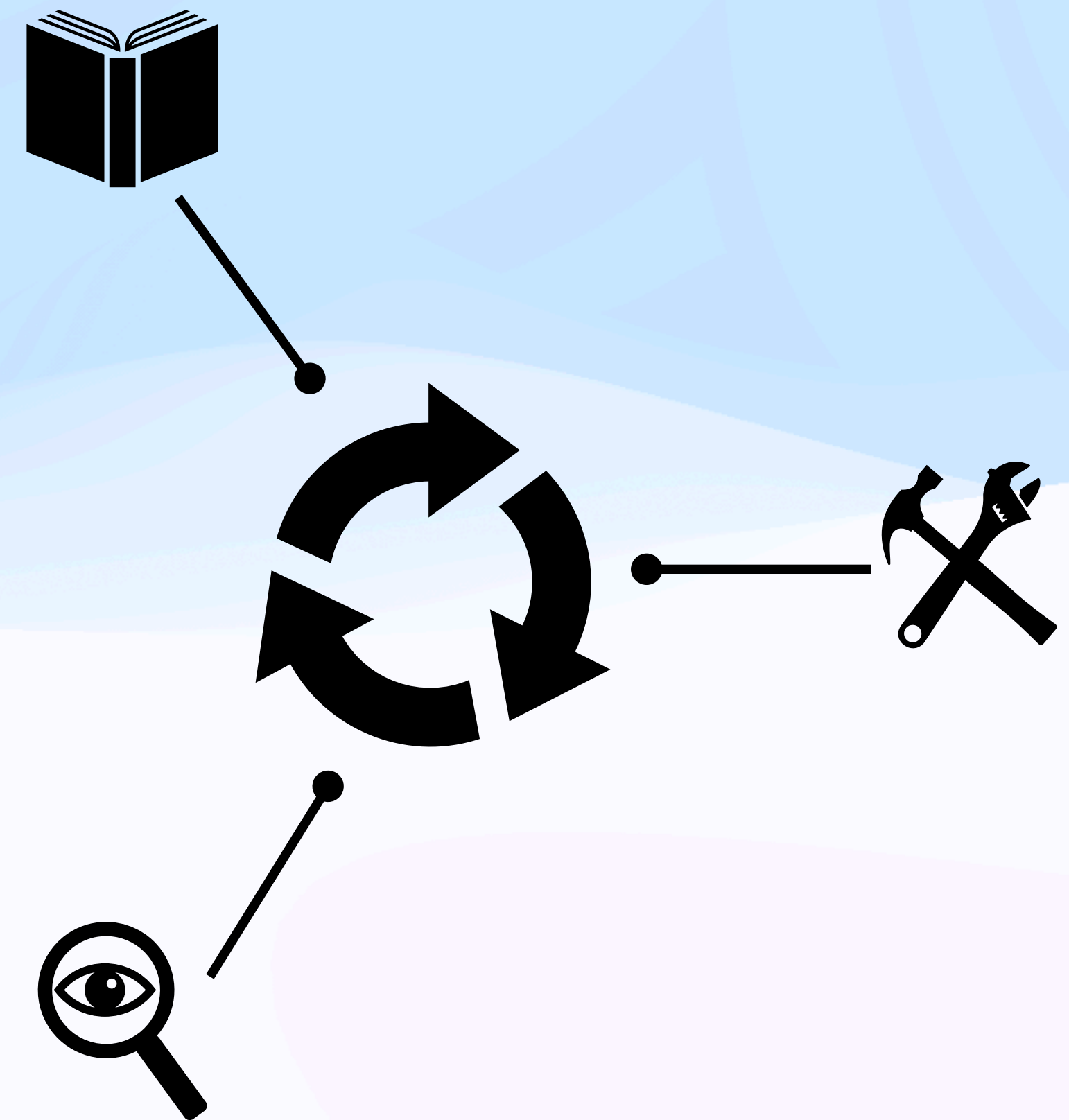
Read published research, experiment

- **Learn codebase**

Use software, understand code

- **Refine skills**

Solve challenges, identify patterns



Methodology

How to approach

- User interaction
- Threat model
- Attack scenarios

```
import ...

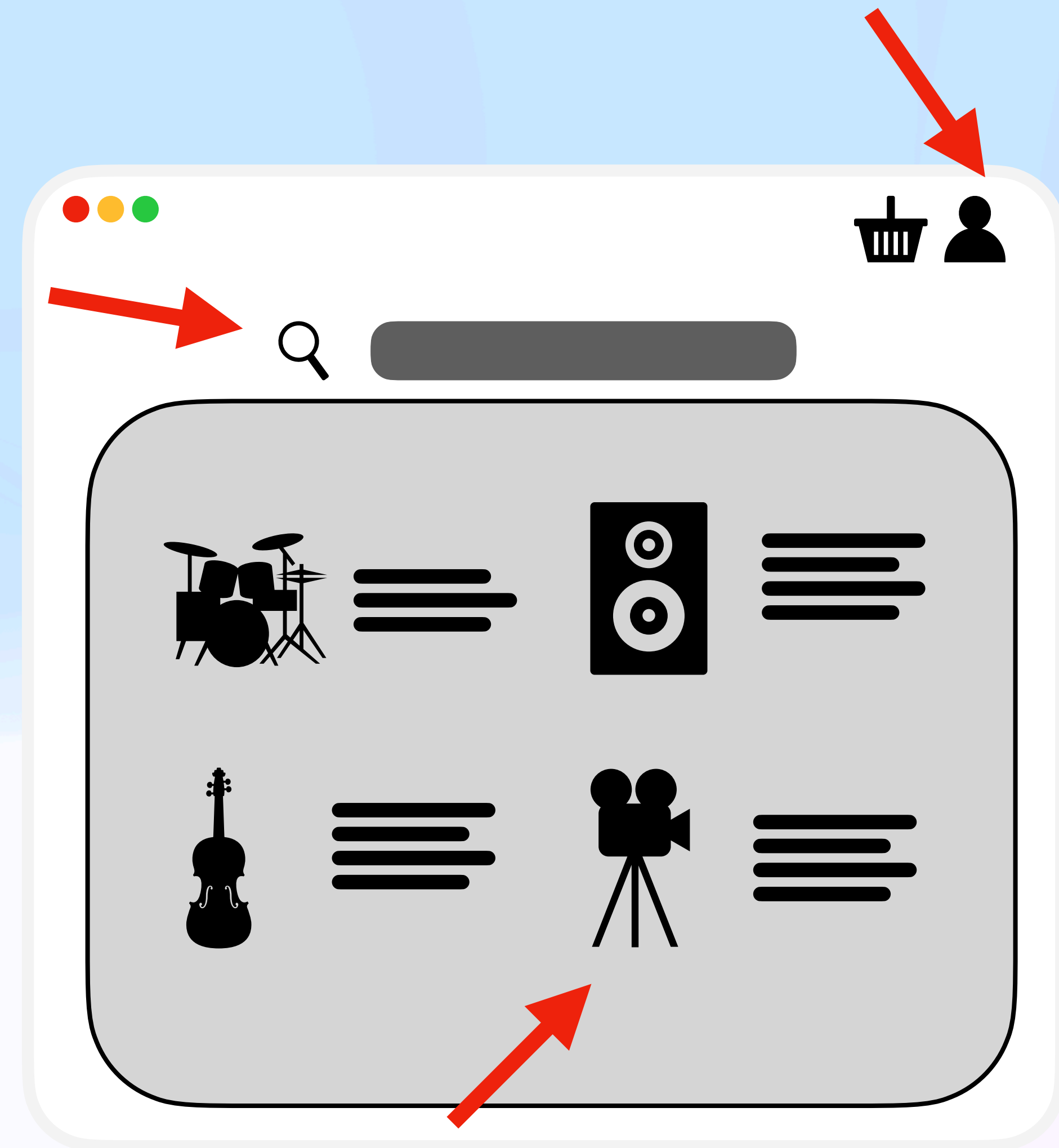
def list_products():
    ...

def add_to_basket():
    ...

def archive_product():
    ...
```

Methodology

- **User interaction**
- Threat model
- Attack scenarios

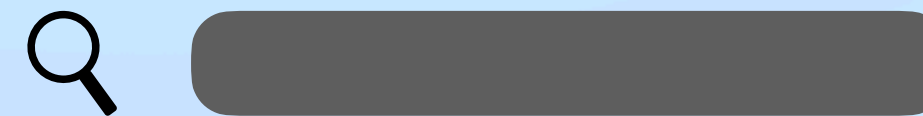


Methodology

- User interaction
- **Threat model**
- Attack scenarios

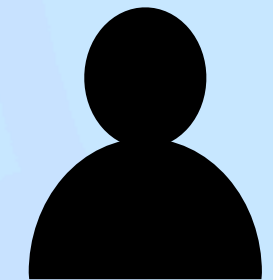
Can I...

list hidden products?



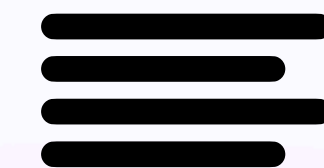
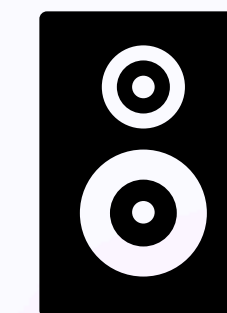
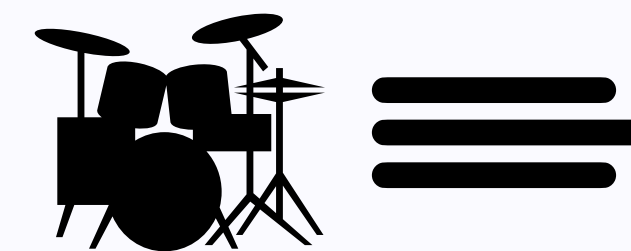
list other information?

Change my data
without password?

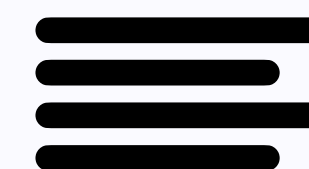


Access other users?

Modify products?



Change price?



Access basket of another user?

Methodology

- User interaction
- Threat model
- **Attack scenarios**

User account

Change email address ➡ Requires password?

Access user information ➡ Requires authorization?

Provide invalid login ➡ Descriptive error?

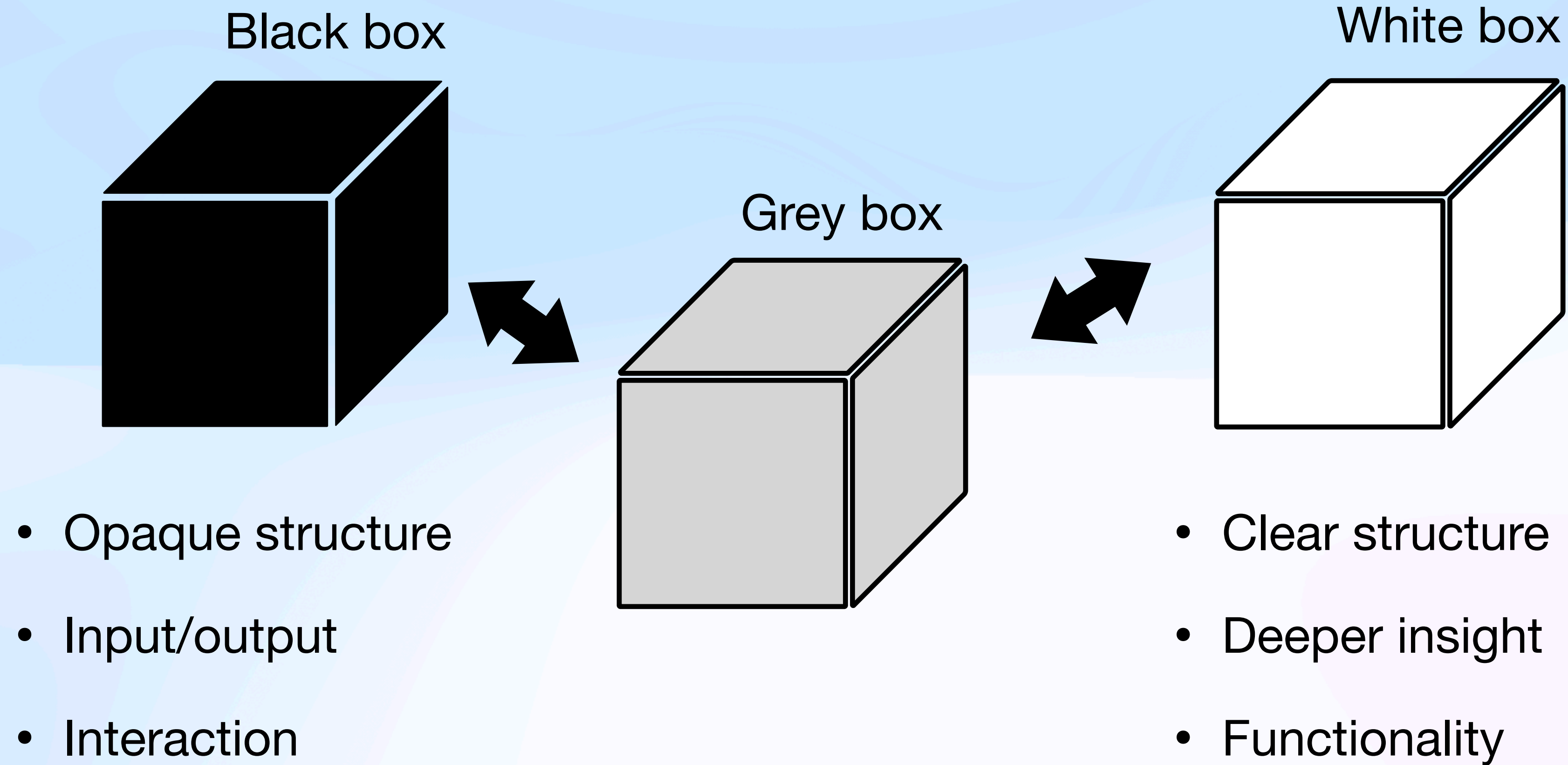
Search products

Use special characters ➡ Affects query results?

Purchase products

Change price in basket ➡ Are changes reflected?

Methodology

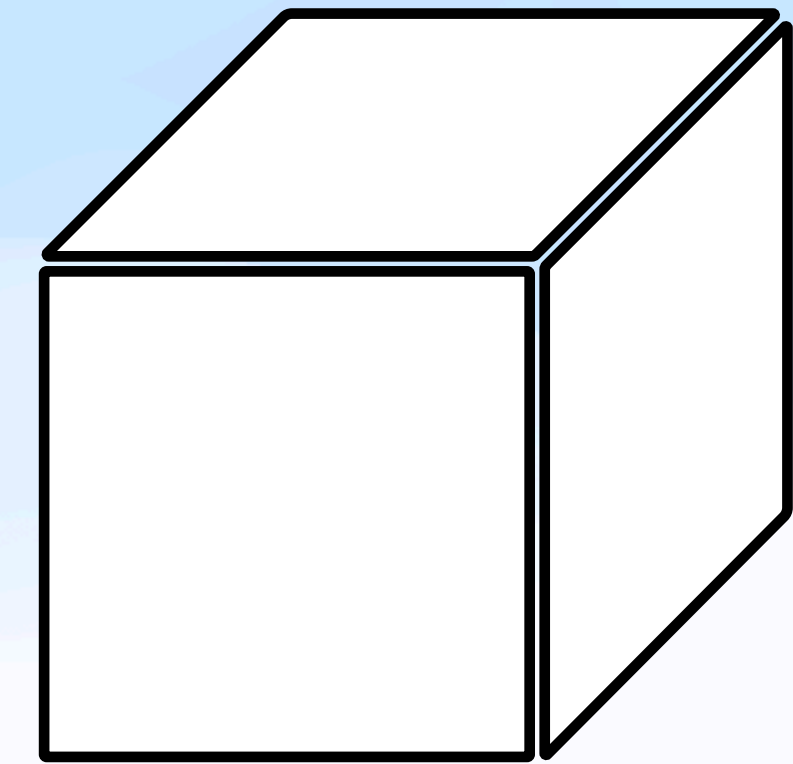


Methodology

White box testing

```
import ...  
  
def list_products():  
    ...  
  
def add_to_basket():  
    ...  
  
def archive_product():  
    ...
```

White box

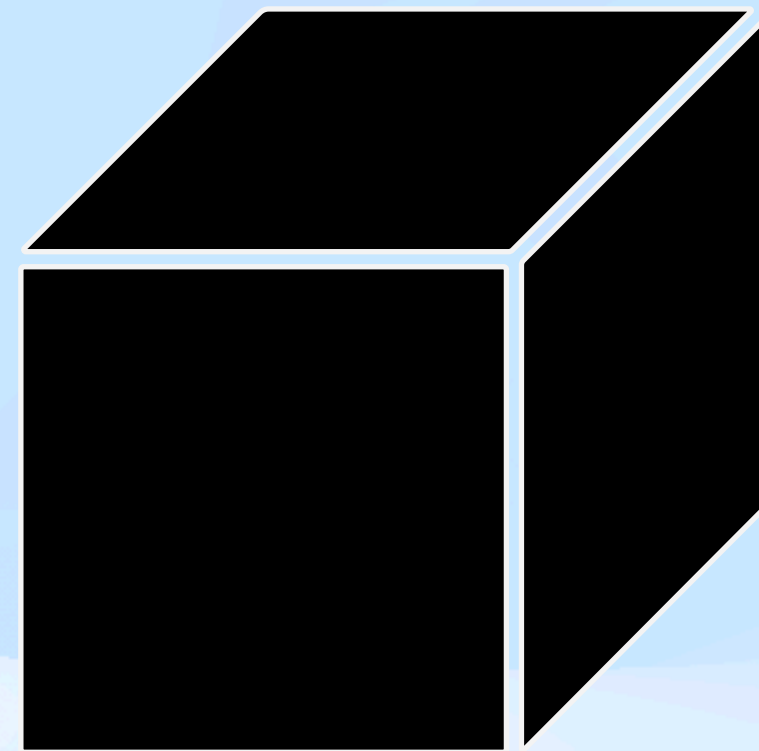


- Clear structure
- Deeper insight
- Functionality

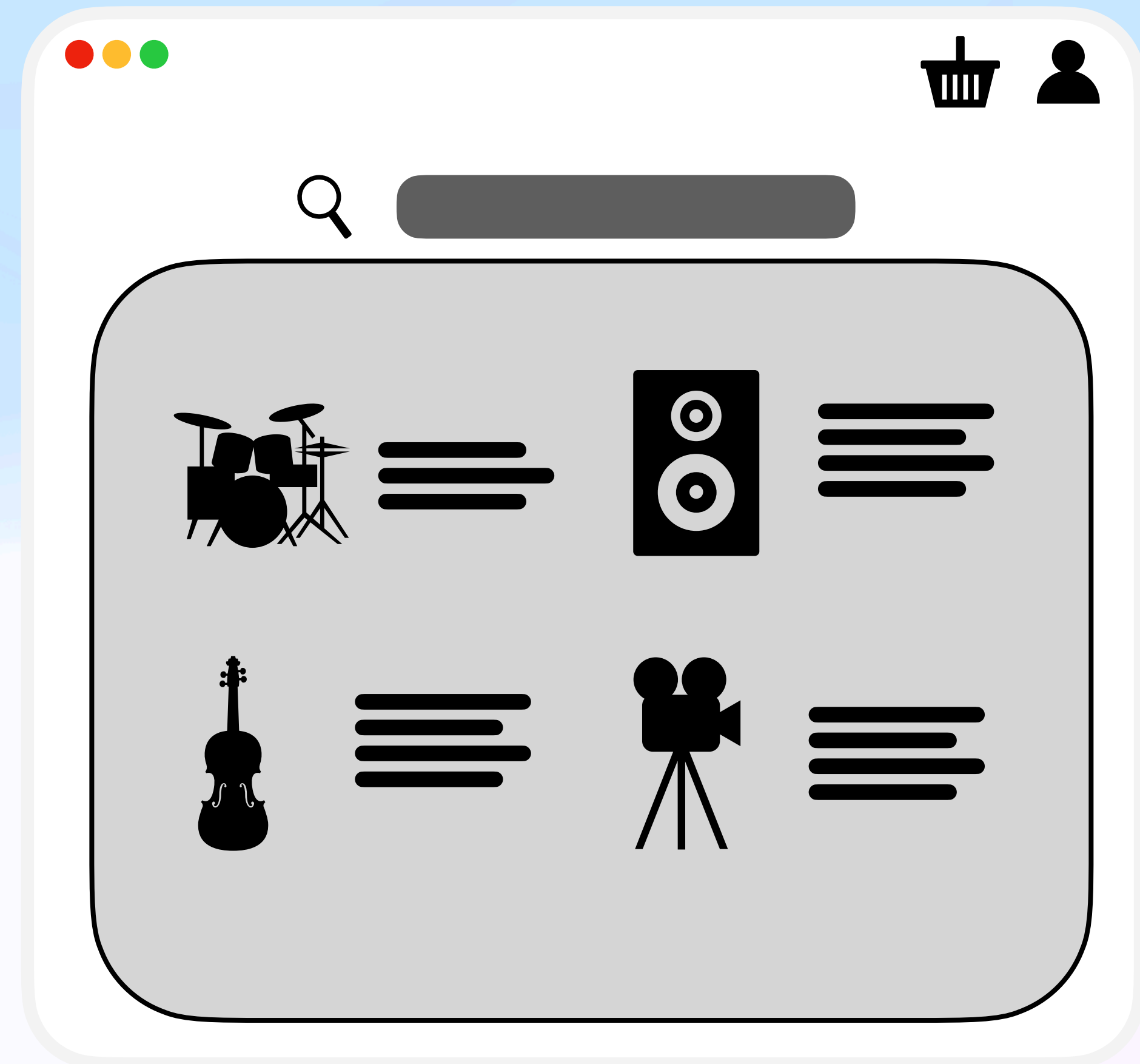
Methodology

Black box testing

Black box



- Opaque structure
- Input/output
- Interaction



Techniques

What to use

DAST

Dynamic Application Security Testing



SAST

Static Application Security Testing



Techniques

DAST

Dynamic Application Security Testing



Activities

- Behavioral analysis
- Fuzzing
- Instrumentation

Tools

- BurpSuite
- Fuzzotron
- Frida

Techniques

Activities

- Code review
- Signature analysis
- Analyze Abstract Syntax Tree (AST)

Tools

- CodeQL
- Bandit
- Semgrep
- Nuclei

SAST

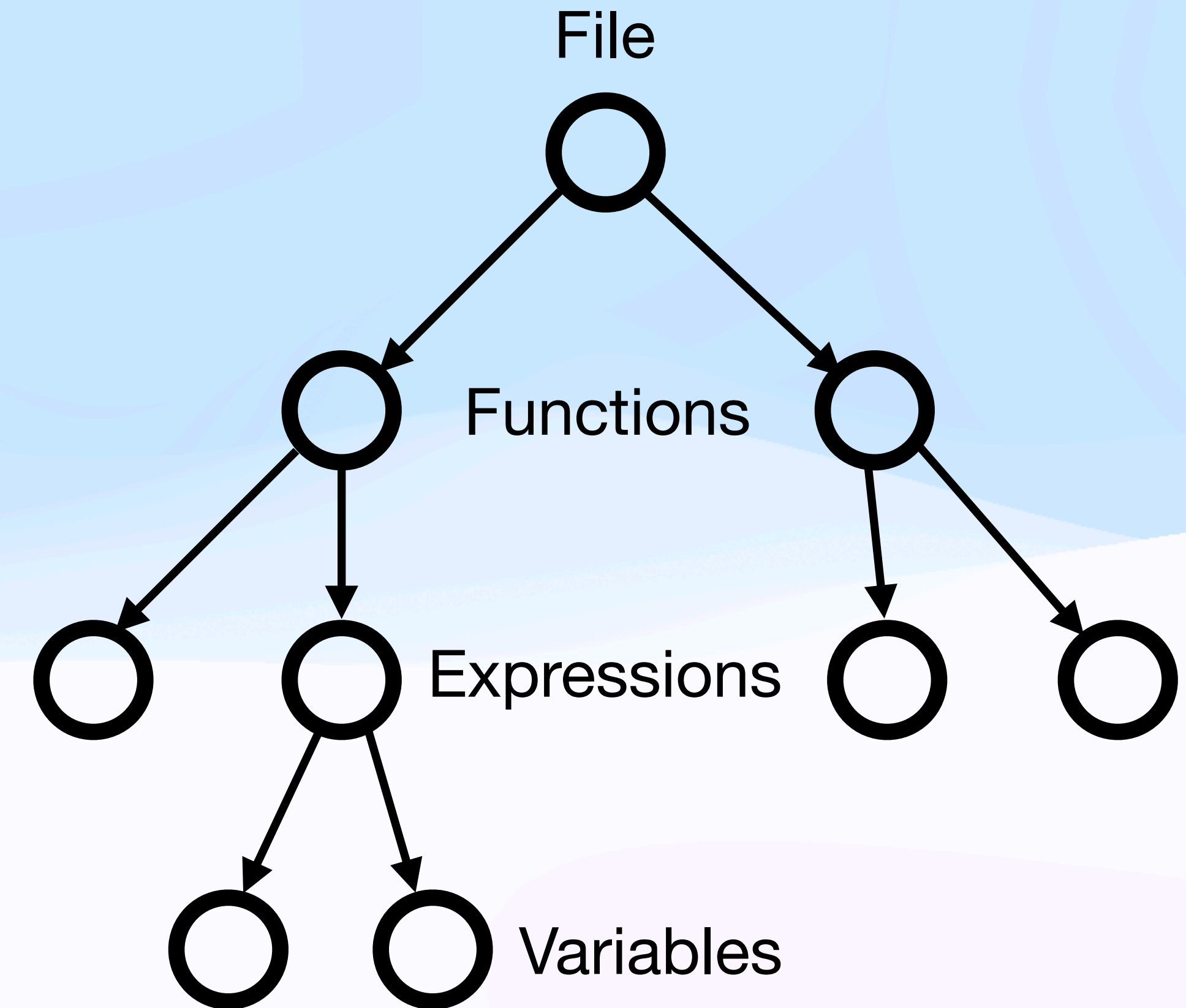
Static Application Security Testing



Abstract Syntax Trees

Basic information

- Data structure
- Graphical representation
- Abstract syntactic structure
- Used in compilers



Abstract Syntax Trees

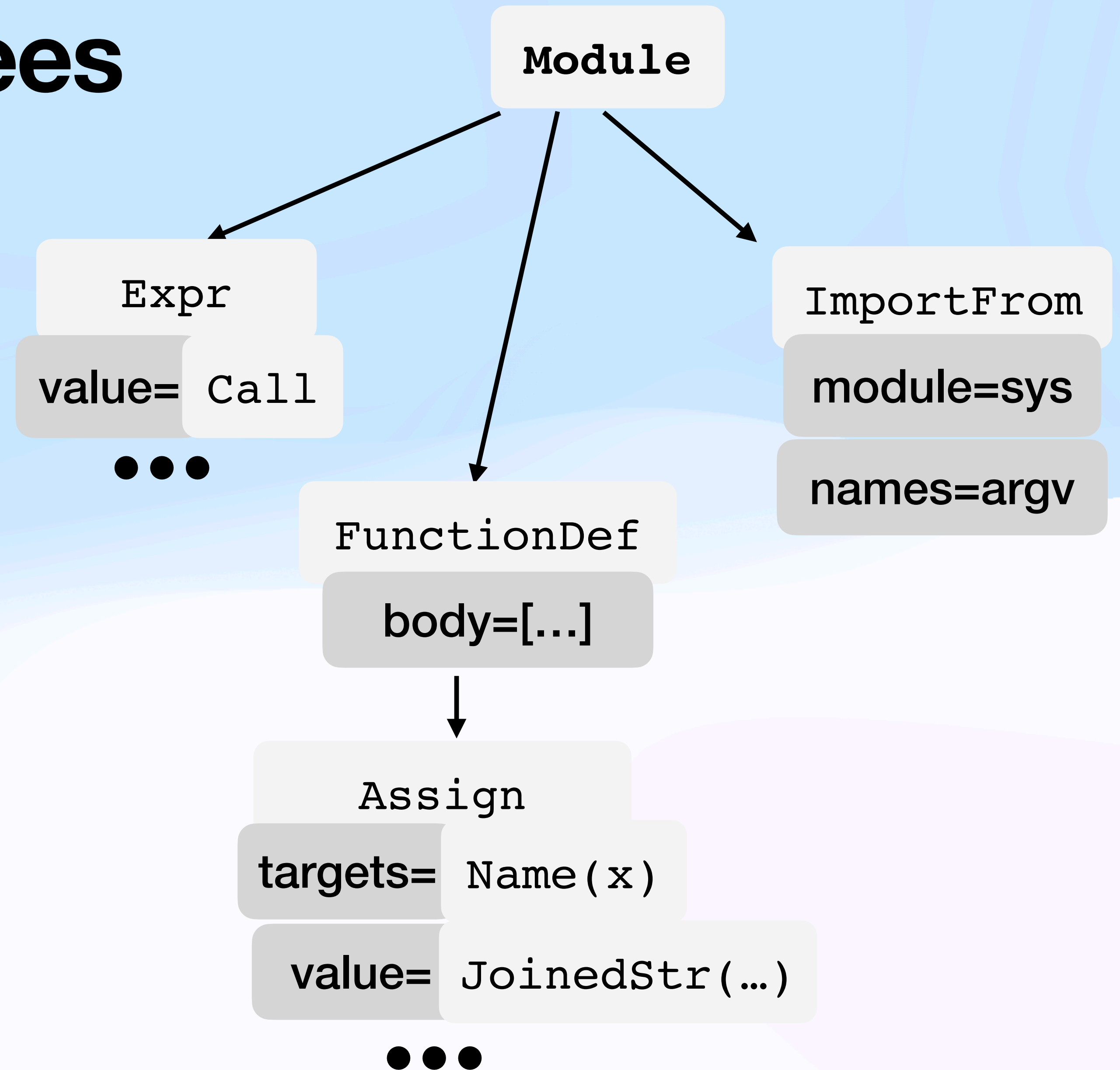
- Data structure
- Graphical representation
- Abstract syntactic structure
- Used in compilers

```
from ast import parse, dump  
from sys import argv
```

```
file = open(argv[1])  
tree = parse(file.read())  
print(dump(tree, indent=2))
```


Abstract Syntax Trees

```
1 from sys import argv
2
3 def hello(name):
4     x = f"hello {name}!"
5     return x
6
7 print(hello(argv[1]))
```



Abstract Syntax Trees

Exercise

Exercises/ex1.py

```
1 from this import s
2
3 words = [x for x in s.split()]
4 print(len(words))
```

```
$ get_ast.py Exercises/ex1.py
```

Bandit

Overview

- Open-source security linter for Python
- Builds AST -> checks against set of defined plugins
- Active, occasionally updated, several open issues
- <http://bandit.readthedocs.io/>

Install

```
$ python -m venv bvenv  
$ source bvenv/bin/activate  
(bvenv)$ pip install bandit
```


Bandit

Basic use

- List default plugins
- Run plugins for specific issue type
- Run specific plugin

```
$ bandit -r file.py
```

```
$ bandit -r file.py -s MEDIUM
```

```
$ bandit -r file.py -t B703
```

Bandit

Implementation

- Process target file -> build AST -> run set of plugins against nodes
- Uses a variant of the NodeVisitor ([bandit/core/node_visitor.py](#)) paradigm
- Reporting vulnerabilities, categorizing their impact and filtering ([bandit/core/issue.py](#))

Bandit

Plugins

- Rules are written with Python using the Bandit API
- Default - reports severity levels of **medium** and higher
- Use `# nosec` on line above specific issue to ignore it

Test results:

Issue: [`<ID>`] Use of weak MD5 for hashing.

Severity: Low

Confidence: High

Location: `example.py:5`

More Info: <https://bandit.readthedocs.io/plugins/...>

MD5 is a known broken hash algorithm. Avoid using it.

Bandit

Plugins

Bandit informs user about vulnerabilities as follows

- **File path** - Python file where the issue was detected
- **Line number** - where the issue is located
- **Test ID** - plugin unique identifier related to vulnerability type
- **Severity** - severity level of the issue (low, medium, high)
- **Confidence** - if issue is actually a security problem (low, medium, high)
- **Message** - brief description of the security issue

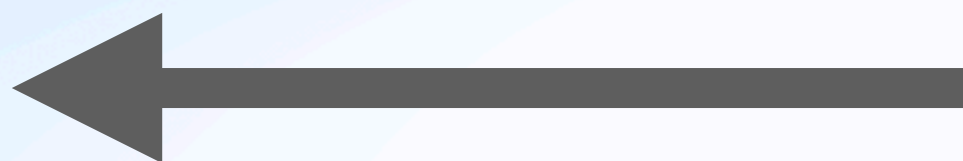
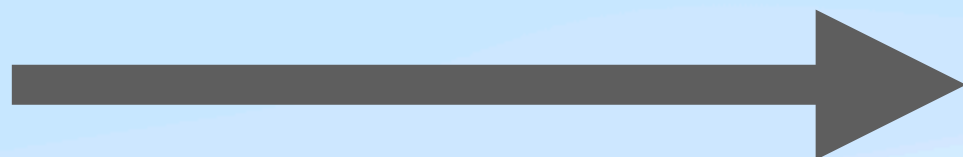
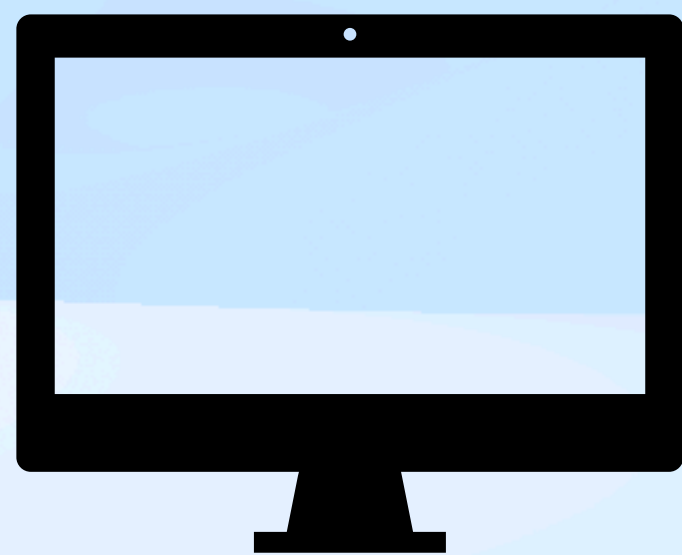
SQL Injection



SQL Injection

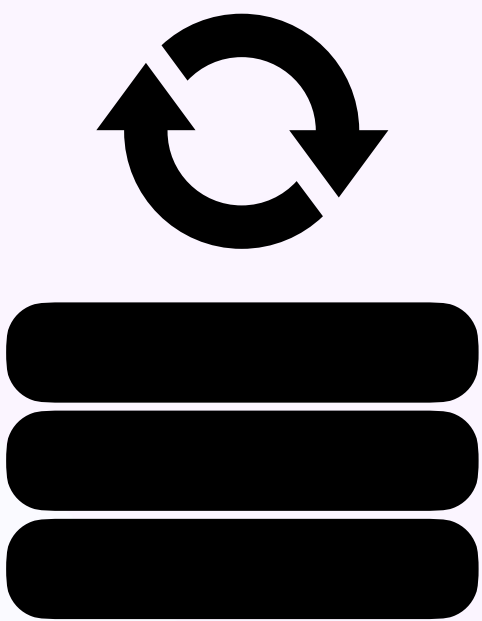
Database query

SELECT * FROM Products WHERE name = "Guitar"



ID	Name	Category	Price
1	Guitar	Music	200
2	Bow	Sport	500
3	Desk	Furniture	800
...

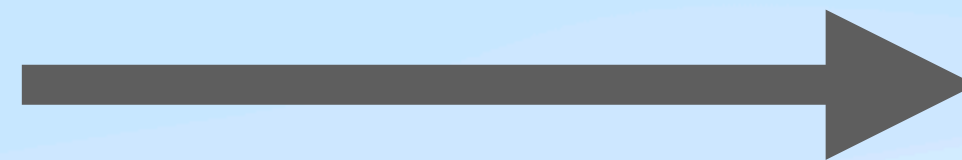
1	Guitar	Music	200
---	--------	-------	-----



SQL Injection

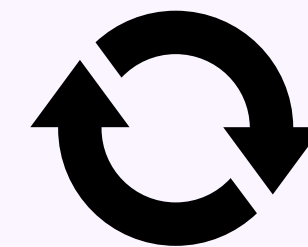
Attack idea

SELECT * FROM Products WHERE name = "" OR 1=1;-- "



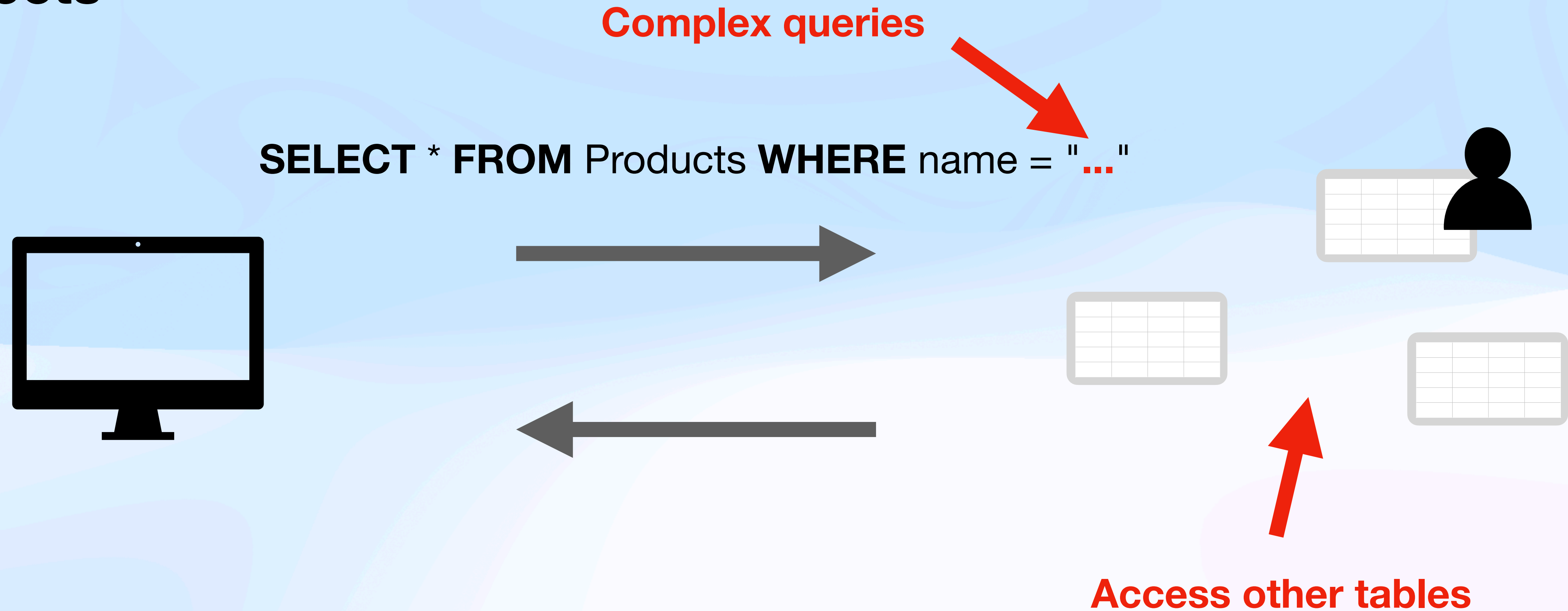
List of all products...

ID	Name	Category	Price
1	Guitar	Music	200
2	Bow	Sport	500
3	Desk	Furniture	800
...



SQL Injection

Effects



SQL Injection

Vulnerable implementation

```
@app.route( '/product' , methods=[ 'GET' ] )
def products():
    name = request.form[ 'name' ]
    conn = get_db_connection()
    products = conn.execute(
        "SELECT * FROM products WHERE product_name = '" + name + "'"
    ).fetchall()
    conn.close()
    ...
```


Bandit Detection

1. Evaluate AST - find instances of string concatenation
2. Extract string statement, compare to regular expression
3. Check methods that use specific statement

Bandit

1. Evaluate AST - find instances of string concatenation

```
def _evaluate_ast(node):  
    if isinstance(node._bandit_parent, ast.BinOp):  
        ...  
    elif isinstance(  
        node._bandit_parent, ast.Attribute  
    ) and node._bandit_parent.attr in ("format", "replace"):  
        ...  
    elif hasattr(ast, "JoinedStr") and isinstance(  
        node._bandit_parent, ast.JoinedStr  
    ):  
        ...
```

Bandit

```
conn.execute("SELECT * FROM products WHERE product_name = '"+name+"'")
```

```
if isinstance(node._bandit_parent, ast.BinOp):  
    out = utils.concat_string(node, node._bandit_parent)  
    wrapper = out[0]._bandit_parent  
    statement = out[1]
```

node	➡	SELECT * FROM products WHERE product_name = '
node->parent	➡	ast.BinOp (ast.Add)
wrapper	➡	ast.Call (execute)
statement	➡	"SELECT * FROM products WHERE product_name = ' '"

Bandit

2. Extract string statement, compare to regular expression

```
SIMPLE_SQL_RE = re.compile(  
    r"(select\s.*from\s|"   
    r"delete\s+from\s|"   
    r"insert\s+into\s.*values\s|"   
    r"update\s.*set\s)",  
    re.IGNORECASE | re.DOTALL,  
)  
  
def _check_string(data):  
    return SIMPLE_SQL_RE.search(data) is not None
```

Bandit

3. Check methods that use specific statement

```
if isinstance(wrapper, Call):  
    names = ["execute", "executemany"]  
    name = utils.get_called_name(wrapper)  
    return (name in names, statement, str_replace)  
else:  
    return (False, statement, str_replace)
```

Bandit

```
@test.checks("Str")
@test.test_id("B667")
def hardcoded_sql_expressions(context):
    execute_call, statement, str_replace = _evaluate_ast(context.node)
    if _check_string(statement):
        return bandit.Issue(
            severity=bandit.MEDIUM,
            confidence=(
                bandit.MEDIUM
                if execute_call and not str_replace
                else bandit.LOW
            ),
            cwe=Cwe.SQL_INJECTION,
            text="Possible SQL injection ...")
```


Add own plugins



Bandit

Plugin structure

- **context** (bandit/core/context.py) - object of analysis
- **helper tools** (bandit/core/utils.py) - make evaluation of AST nodes
- **@checks** (bandit/core/test_properties.py) - determine type of AST node

```
import ...

@test.checks("<AST TYPE>")
@test.test_id("B###")
def main_check(context):
    if ...:
        return Issue(
            severity=bandit.LOW,
            confidence=bandit.HIGH,
            cwe=Cwe.###,
            text=f"Vulnerable ...")
```

Bandit

Context

Main object of analysis

- Function properties (*name, arguments, keywords, named argument*)
- Import properties (*is it imported name? is it aliased?*)
- AST node
- Bytes representation
- File-related properties

Bandit

Python package

bandit_plugin

|—— bandit_plugin

| |—— init.py

| |—— myplugin.py

|—— setup.py

```
from setuptools import setup
```

```
setup(
    name='bandit_plugin_test',
    version='0.0.1',
    description='...',
    packages=['bandit_plugin'],
    author='',
    install_requires=['bandit'],
    entry_points={
        'bandit.plugins': [
            'os_getcwd =
bandit_plugin.myplugin:function',
        ],
    })
```


Bandit

Adding plugin

- Create plugin module
- Add test ID -> `@test_properties.test_id`
- Install local package
- Provide plugin ID to use it
- Modify installed package

```
$ python -m pip install ./bandit_plugin
```

```
$ bandit -t B### -f custom <file.py>
```

```
bvenv/lib/python3.13/site-packages/bandit/plugins/
```

Exercises



Bandit

Future plans

- Extend plugin coverage
- Introduce taint tracking
- Implement VS Code plugin

Bandit

Plugins

ID	Description	Count	Examples	CWE
B1xx	misc tests	13	hardcoded credentials, unsafe functions	CWE-703 , CWE-78 , CWE-732
B2xx	misconfiguration	2	configuration, archive extraction	CWE-94 , CWE-22
B3xx/B4xx	blacklists (calls/imports)	1	Insecure cryptography	CWE-327
B5xx	cryptography	9	Insecure configuration, missing validations	CWE-295 , CWE-327 , CWE-326
B6xx	injection	15	code injection	CWE-78 , CWE-89 , CWE-94
B7xx	XSS	4	Cross-Site Scripting	CWE-94

Static analysis tools

Comparison

Bandit

Open source

Python API

<50 plugins

Semgrep

Open source

YAML

~350 rules

CodeQL

Closed source

Free for research

~50 queries

[GitHub - Python CodeQL queries](#)

[Semgrep - Python security rules](#)

[Bandit - security plugins](#)

Summary

- Security research aims to discover software vulnerabilities
- Whitebox - focus on code structure
- AST - code structure visualisation
- Static analysis provides helps to analyze relationships between code entities





Thank you