Risk-Based and Effective Security Testing

TOREON | Ruben De Visscher

Alice & Bob’s Digital Products Inc.

Agenda: Risk Based and Effective Security Testing

Tool-centric security testing is not sufficient

- You, *not the tool vendor*, are accountable for the testing strategy.
- Active threats and legislation mandate *risk-based* and *effective* testing
- Justifying your testing strategy is not straight forward

Take ownership of your security testing strategy

- Validating the known is a **mechanical** process
- Discovering the unknown is a **creative** process

Vulnerability *handling* is not just vulnerability *fixing*

- Reflect on *every* detected vulnerability and meaningfully improve the SDLC.

Who Am I?

Ruben De Visscher (ruben.devisscher@toreon.com)

Principal Product Security Consultant @ Toreon

Software Engineer / Architect background (10+ YoE)

Worked on backend of security products

SDLC Training & Coaching (OWASP SAMM)

(Static) AppSec Testing & Vulnerability Handling

@ SME, Government and large enterprises

IEC 62443-4-2: security reqs. for industrial control systems

You, *not the tool vendor*, are accountable for the testing strategy.

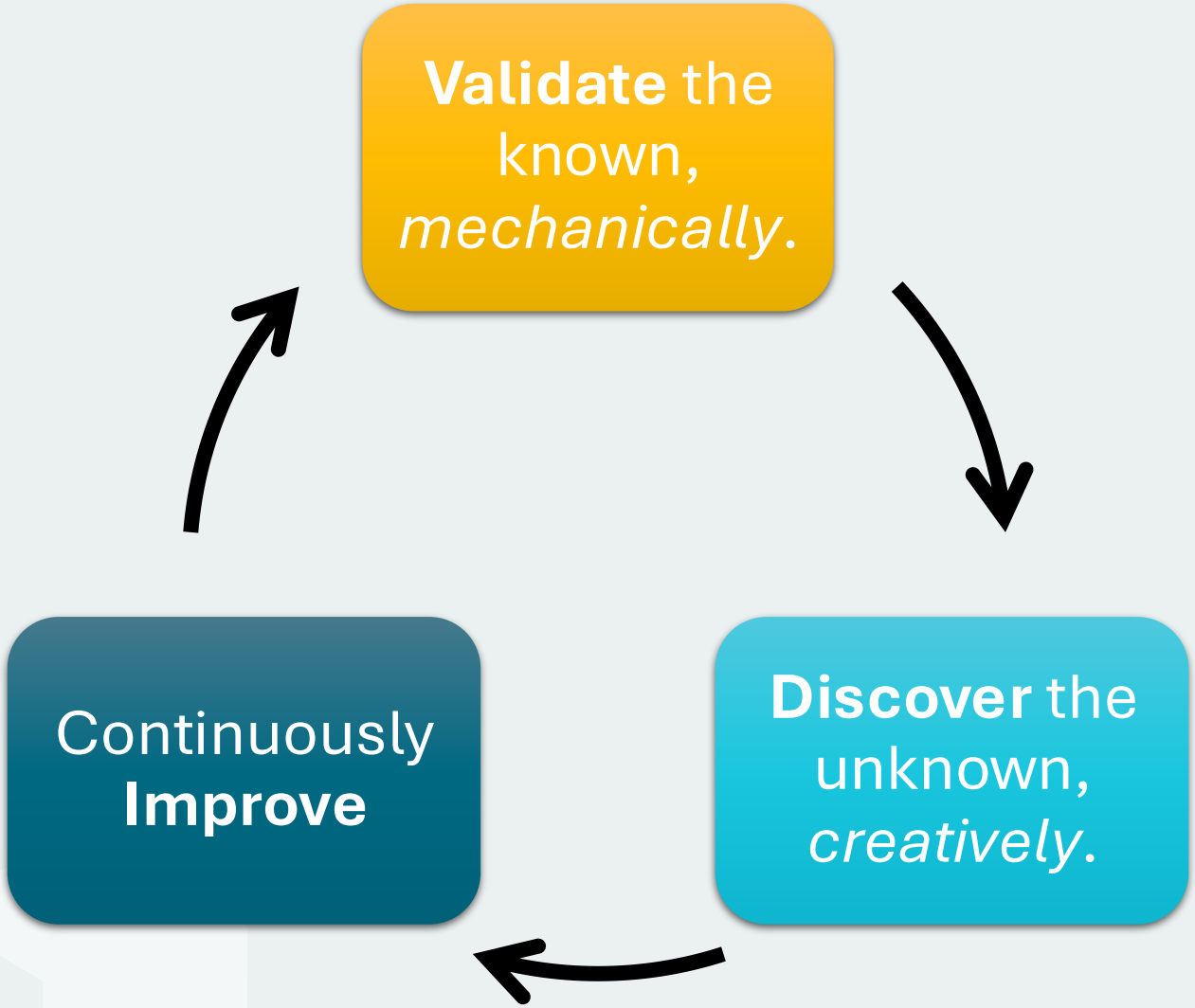
- Tools assume a **generic security context**.
- Tool **vendors want to reduce false positives**.
- Many tool **vendors are not transparent** on testing coverage and accuracy.
- Many **vendors sell “peace of mind”** and **checkbox compliance**.

“apply effective and regular tests and reviews of the security of the product with digital elements”

Justifying your testing strategy is not straight forward

“Risk-based”	“Effective”
High likelihood and/or high impact.	“Good enough” precision, High recall.
Focus on **high-risk** components and threats.	Testing results in measurable **SSDLC improvements**.
Based on the **threat model** in an **intended operational environment**.	Low cost to business.

Take ownership of your security testing strategy



Validate the known by mechanically following checklists

Validating the known is a *mechanical* process

Security Requirements

Threat Models

Well-Known Attack Methods

Secure Coding Policies

Automated Security Testing

(Abuse) Test Cases

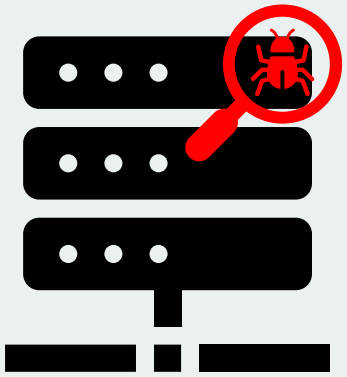
Architecture & Design Reviews

Enforce the rules and cover the basics with automated security testing

Well-Known Attack Methods

Secure Coding Policies

Automated Security Testing



Runtime testing to detect misconfiguration and hardening issues

Static testing to detect vulnerable code early and deterministically.

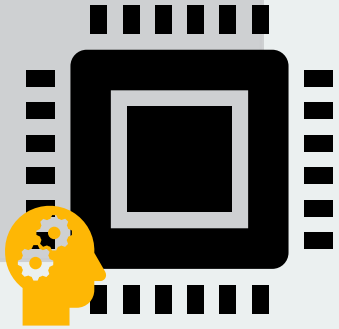
Only syntactic weaknesses can be effectively tested with fully automated tools

Syntactic Weaknesses

Wrong in (almost) every product

Examples:

- Injection issues
- Weak crypto
- Insecure configuration (headers, cookie attributes, IaC)
- Secure coding violation



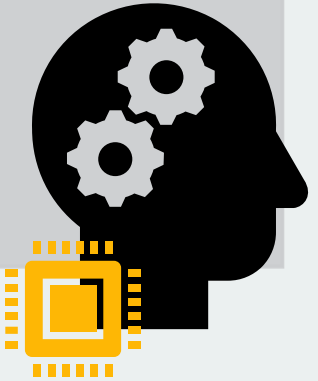
Off-the-shelf automated security testing
Configure tools to increase effectiveness

Semantic Weaknesses

Product dependent

Examples:

- Broken access control
- Business logic violations
- Insecure design
- Exposure of sensitive data
- Insufficient logging
- Broken error handling



Manual test creation / design review
Automate where possible

Syntactic weaknesses are easy to find with automated tools

CWE-78: OS Command Injection
https://cwe.mitre.org/data/definitions/78.html

Detection Methods

Method	Details
Automated Static Analysis	This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives. Automated static analysis might not be able to recognize when proper input validation is being performed, leading to false positives - i.e., warnings that do not have any security consequences or require any code changes. Automated static analysis might not be able to detect the usage of custom API functions or third-party libraries that indirectly invoke OS commands, leading to false negatives - especially if the API/library code is not available for analysis. **Note:**This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Automated Dynamic Analysis	This weakness can be detected using dynamic tools and techniques that interact with the product using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The product's operation may slow down, but it should not become unstable, crash, or generate incorrect results. **Effectiveness: Moderate**
Manual Static Analysis	Since this weakness does not typically appear frequently within a single software package, manual white box techniques may be able to provide sufficient code coverage and reduction of false positives if all potentially-vulnerable operations can be assessed within limited time constraints. **Effectiveness: High**
Automated Static Analysis - Binary or Bytecode	According to SOAR [REF-1479], the following detection techniques may be useful: Highly cost effective: • Bytecode Weakness Analysis - including disassembler + source code weakness analysis • Binary Weakness Analysis - including disassembler + source code weakness analysis **Effectiveness: High**
Dynamic Analysis with Automated Results Interpretation	According to SOAR [REF-1479], the following detection techniques may be useful: Cost effective for partial coverage: • Web Application Scanner • Web Services Scanner • Database Scanners

Semantic weaknesses require manual / custom testing

CWE-306: Missing Authentication for Critical Function
https://cwe.mitre.org/data/definitions/306.html

Detection Methods

Method	Details
Manual Analysis	This weakness can be detected using tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. Specifically, manual static analysis is useful for evaluating the correctness of custom authentication mechanisms. **Note:** These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Automated Static Analysis	Automated static analysis is useful for detecting commonly-used idioms for authentication. A tool may be able to analyze related configuration files, such as .htaccess in Apache web servers, or detect the usage of commonly-used authentication libraries. Generally, automated static analysis tools have difficulty detecting custom authentication schemes. In addition, the software's design may include some functionality that is accessible to any user and does not require an established identity; an automated technique that detects the absence of authentication may report false positives. **Effectiveness: Limited**
Manual Static Analysis - Binary or Bytecode	According to SOAR [REF-1479], the following detection techniques may be useful: Cost effective for partial coverage: • Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies **Effectiveness: SOAR Partial**
Dynamic Analysis with Automated Results Interpretation	According to SOAR [REF-1479], the following detection techniques may be useful: Cost effective for partial coverage: • Web Application Scanner • Web Services Scanner
Architecture or Design Review	According to SOAR [REF-1479], the following detection techniques may be useful: Highly cost effective: • Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.) • Formal Methods / Correct-By-Construction Cost effective for partial coverage: • Attack Modeling **Effectiveness: High**

Test your tools by introducing the vulnerabilities you want to find on purpose

```python
def read_file(filename: str) -> str:
    file_path = os.path.join(BASE_DIR, filename)
    with open(file_path, "r") as f:
        return f.read()
```

```python
@app.route("/static-read", methods=["GET"])
def static_read():
    filename = "../../etc/passwd"
    content = read_file(filename)
    return {"content": content}, 200
```

```python
@app.route("/dynamic-read", methods=["GET"])
def dynamic_read():
    filename = request.args.get("filename")
    content = read_file(filename)
    return {"content": content}, 200
```

Check if your static analysis tool supports these features

Coverage for all important CWEs on *your* tech stack

Tracking tainted data **across file and function** boundaries

Tracking tainted data through **execution flow**: branches, loops

Built-in **support** for the **framework(s)** you are using

Customizable taint sources / sinks / validators / sanitizers

In a mature SDLC: creating fully custom rules

Validating the known is a *mechanical* process

Security Requirements

Threat Models

Well-Known Attack Methods

Secure Coding Policies

Automated Security Testing

(Abuse) Test Cases

Architecture & Design Reviews

Test cases follow automatically from your requirements

Security Requirements

(Abuse) Test Cases

V5.2 File Upload and Content

File upload functionality is a primary source of untrusted files. This section outlines the requirements for ensuring that the presence, volume, or content of these files cannot harm the application.

#	Description	Level
5.2.1	Verify that the application will only accept files of a size which it can process without causing a loss of performance or a denial of service attack.	1
5.2.2	Verify that when the application accepts a file, either on its own or within an archive such as a zip file, it checks if the file extension matches an expected file extension and validates that the contents correspond to the type represented by the extension. This includes, but is not limited to, checking the initial 'magic bytes', performing image re-writing, and using specialized libraries for file content validation. For L1, this can focus just on files which are used to make specific business or security decisions. For L2 and up, this must apply to all files being accepted.	1
5.2.3	Verify that the application checks compressed files (e.g., zip, gz, docx, odt) against maximum allowed uncompressed size and against maximum number of files before uncompressing the file.	2
5.2.4	Verify that a file size quota and maximum number of files per user are enforced to ensure that a single user cannot fill up the storage with too many files, or excessively large files.	3

Anti-requirements tell you what the application should *not* do.

"A user must never access objects belonging to a different tenant"

"A user cannot use more than 1GB of file storage on the backend"

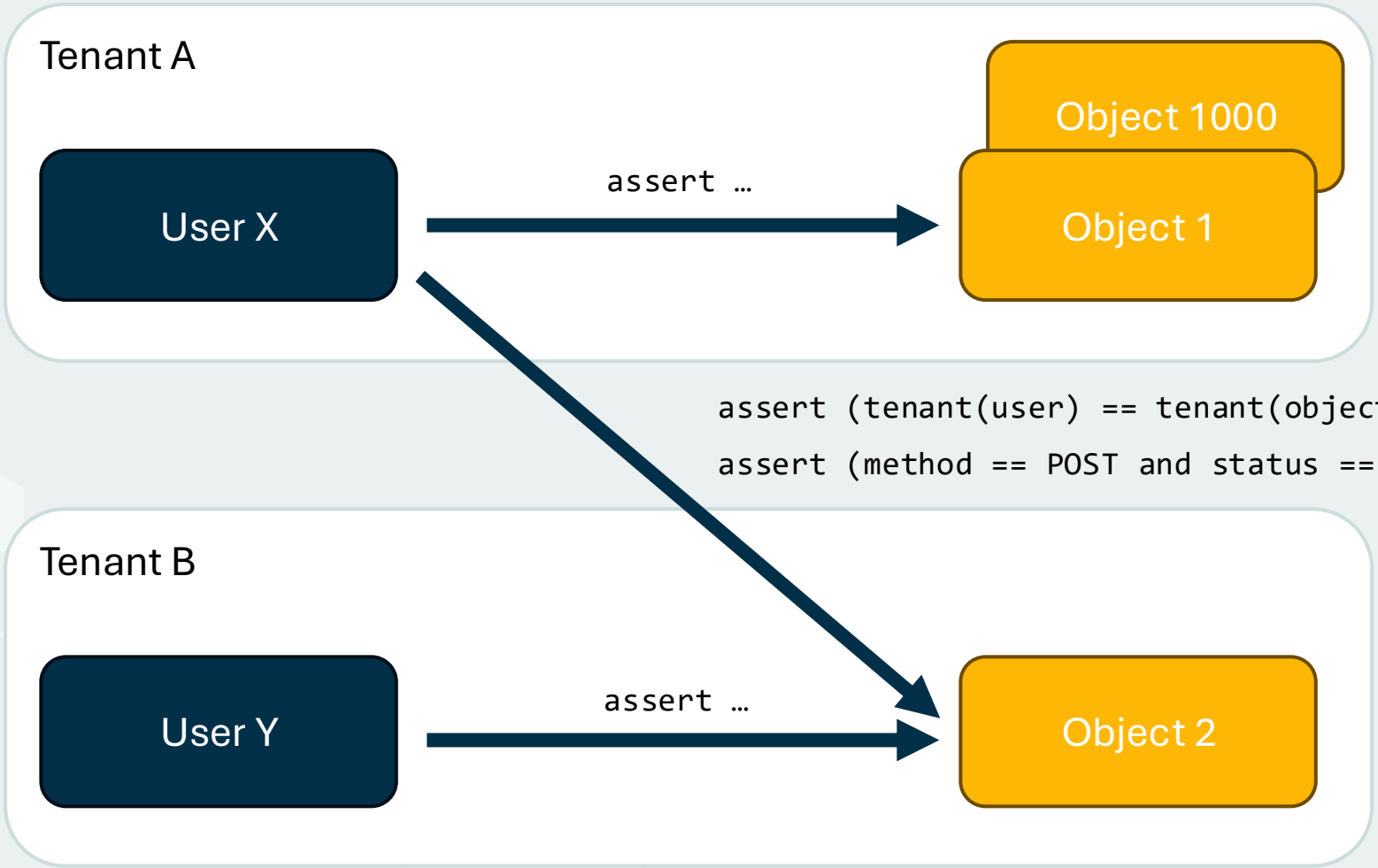
List the invariants: properties that should always hold.

Information about an object in tenant A is sent to user X only if user X is in tenant A

Accessing an object without having permission for that object results in HTTP error 404.

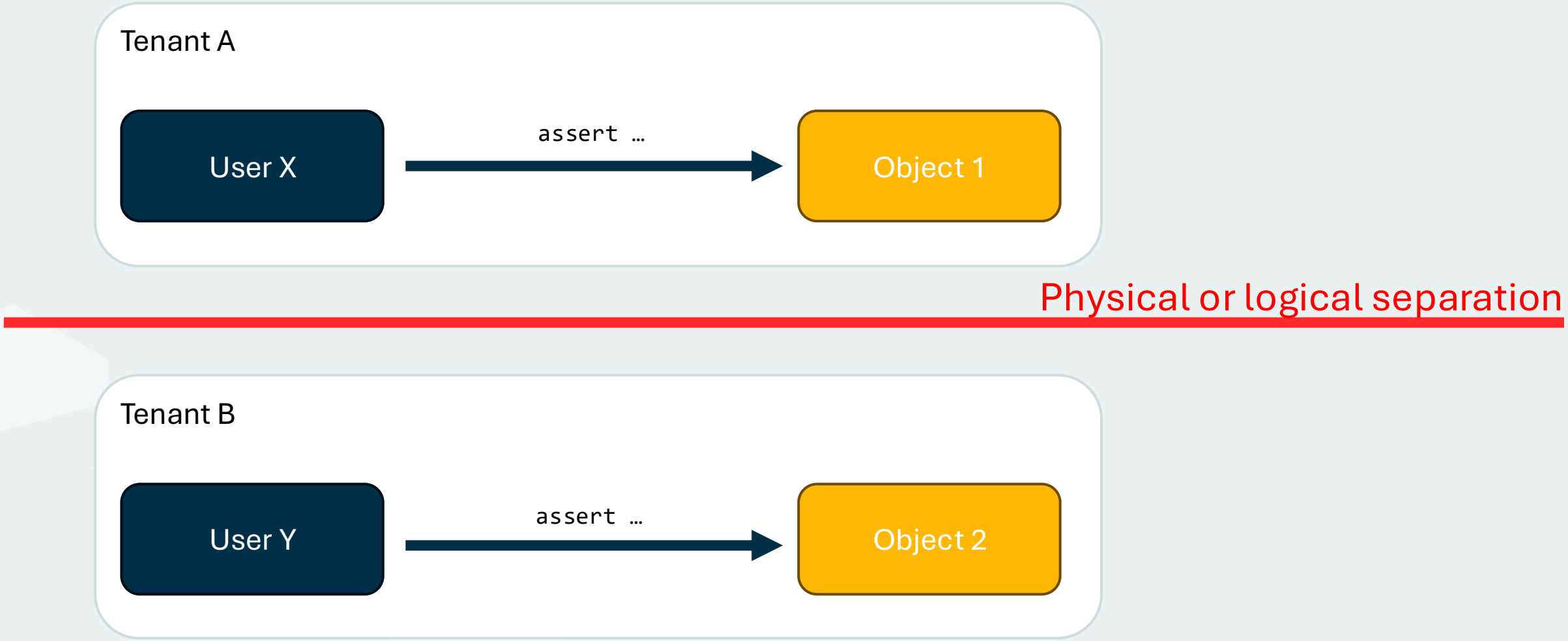
The total storage size used by any user is < 1GB

Create test cases that probe the boundaries of your invariants




```
assert (tenant(user) == tenant(object) and status == 200) or status == 404
assert (method == POST and status == 403) or storage(user) < 1GB
```

… and/or review that (some) invariants hold by design



Validating the known is a *mechanical* process

Security Requirements

Threat Models

Well-Known Attack Methods

Secure Coding Policies

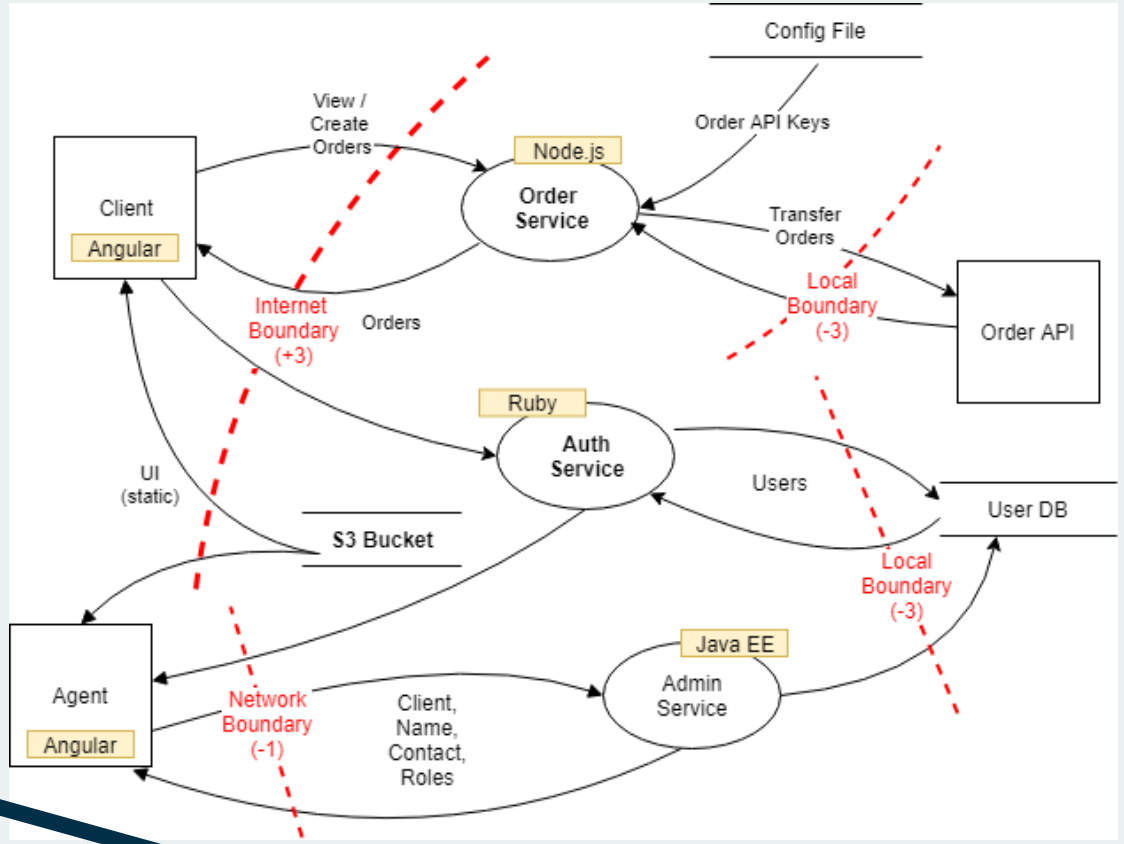
Automated Security Testing

(Abuse) Test Cases

Architecture & Design Reviews

Threat models inform *where* and *how deep* to test, and how to interpret findings!

Threat Models

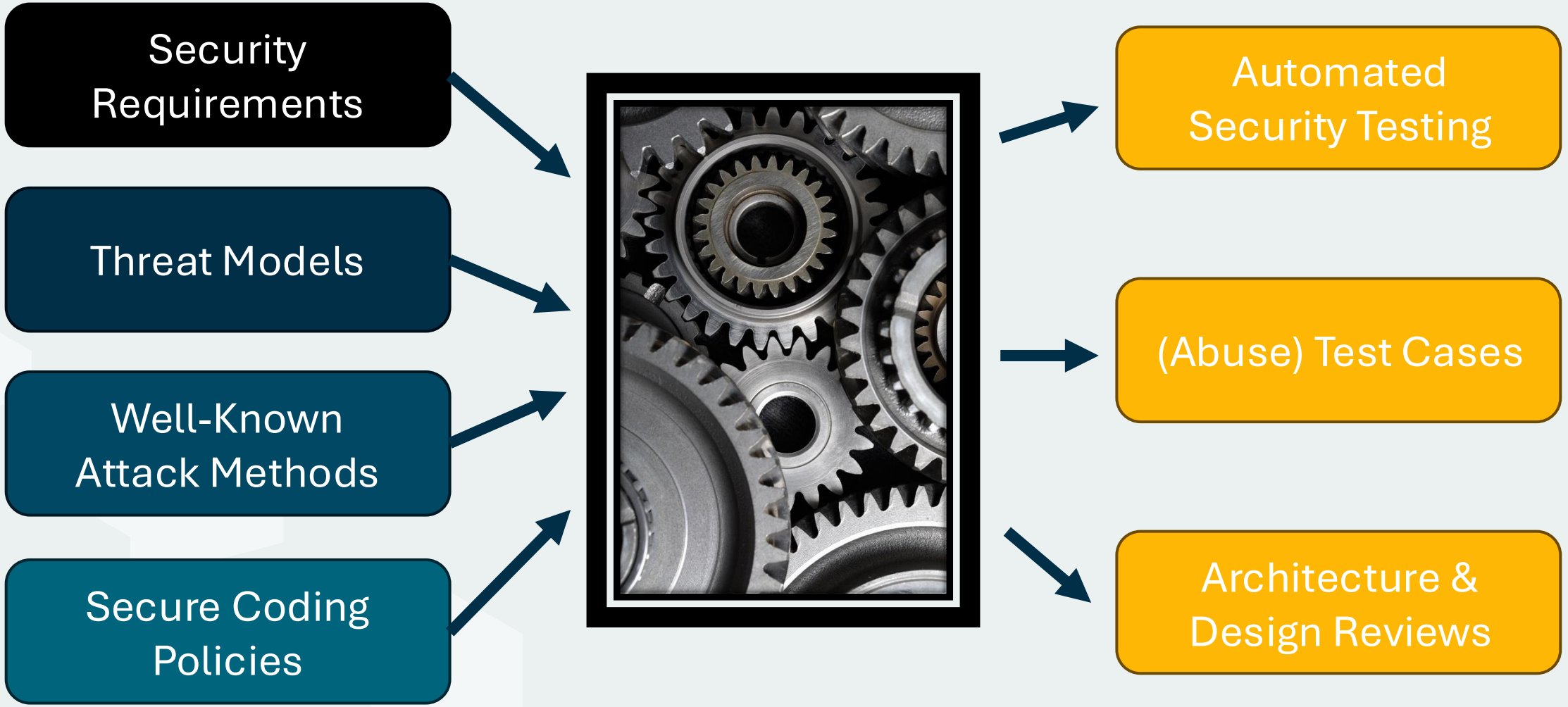


Automated Security Testing

(Abuse) Test Cases

Architecture & Design Reviews

“Effective testing”? Show that these are covered…
… to the best of your knowledge.



What if these inputs are *not* used or available?

… then you don't know ***what*** or ***why*** you are testing.

- Security Requirements
- Threat Models
- Well-Known Attack Methods
- Secure Coding Policies

- Automated Security Testing
- (Abuse) Test Cases
- Architecture & Design Reviews

Discovering the unknown is a *creative* process

Most SDLC participants are not security experts, resulting in gaps.

- Occasional expert review is needed.
- Review tool capabilities and configuration.

Attackers will be creative and constantly evolve their methods.

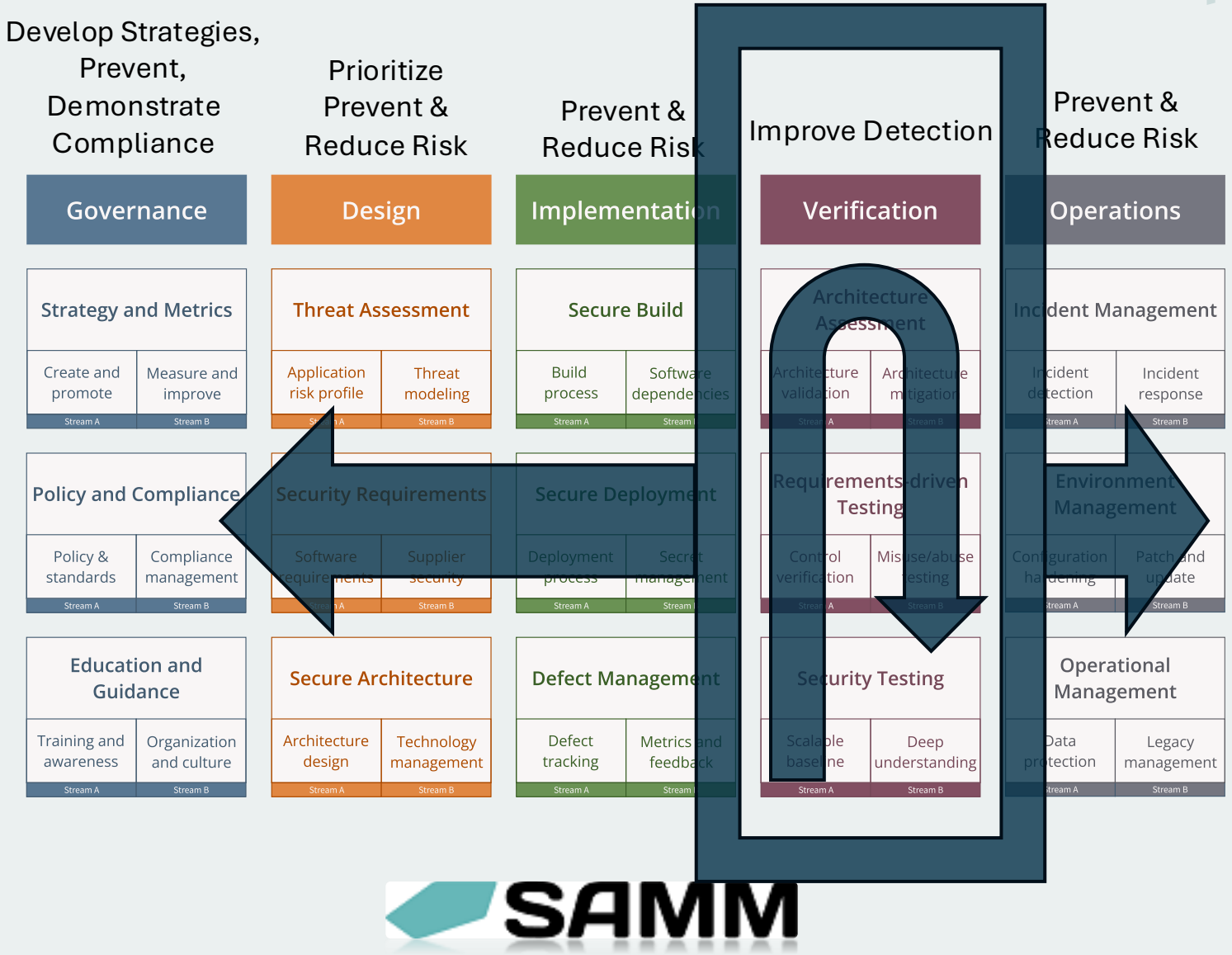
- Prioritize based on real world threat intelligence
- Try different tools and techniques
- Perform independent, **white box** penetration tests

Focus efforts on high risk, high uncertainty components.

Leverage LLMs as a discovery tool, not an assurance tool.

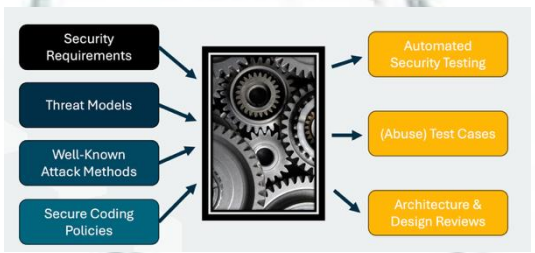
Vulnerability *handling* is not just vulnerability *fixing*.

- Try to **find similar issues** that may have been missed by testing process.
- Address the people / process / technology **root cause** of every finding, *including false positives*.
- Analyze **metrics** that are likely to point to underlying causes on a larger scale.
- Security testing can be a way of **bootstrapping** other SDLC activities in **immature teams**.

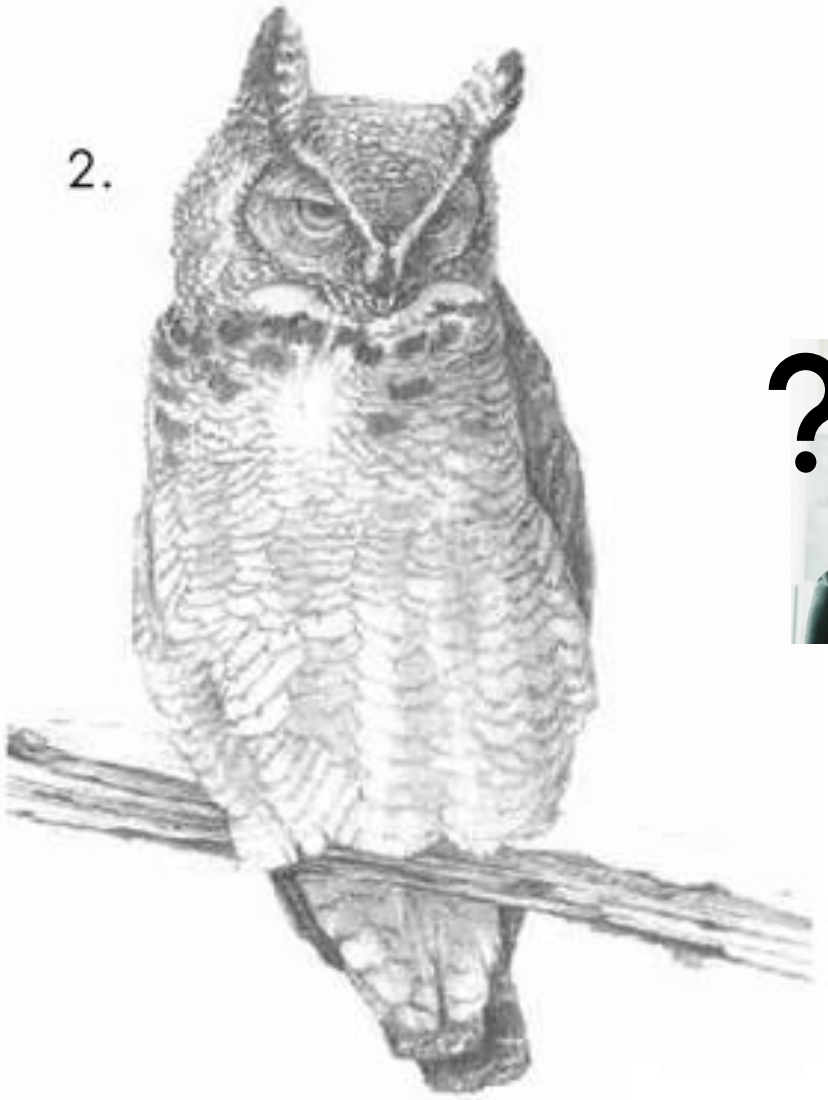


How to draw an owl

1.

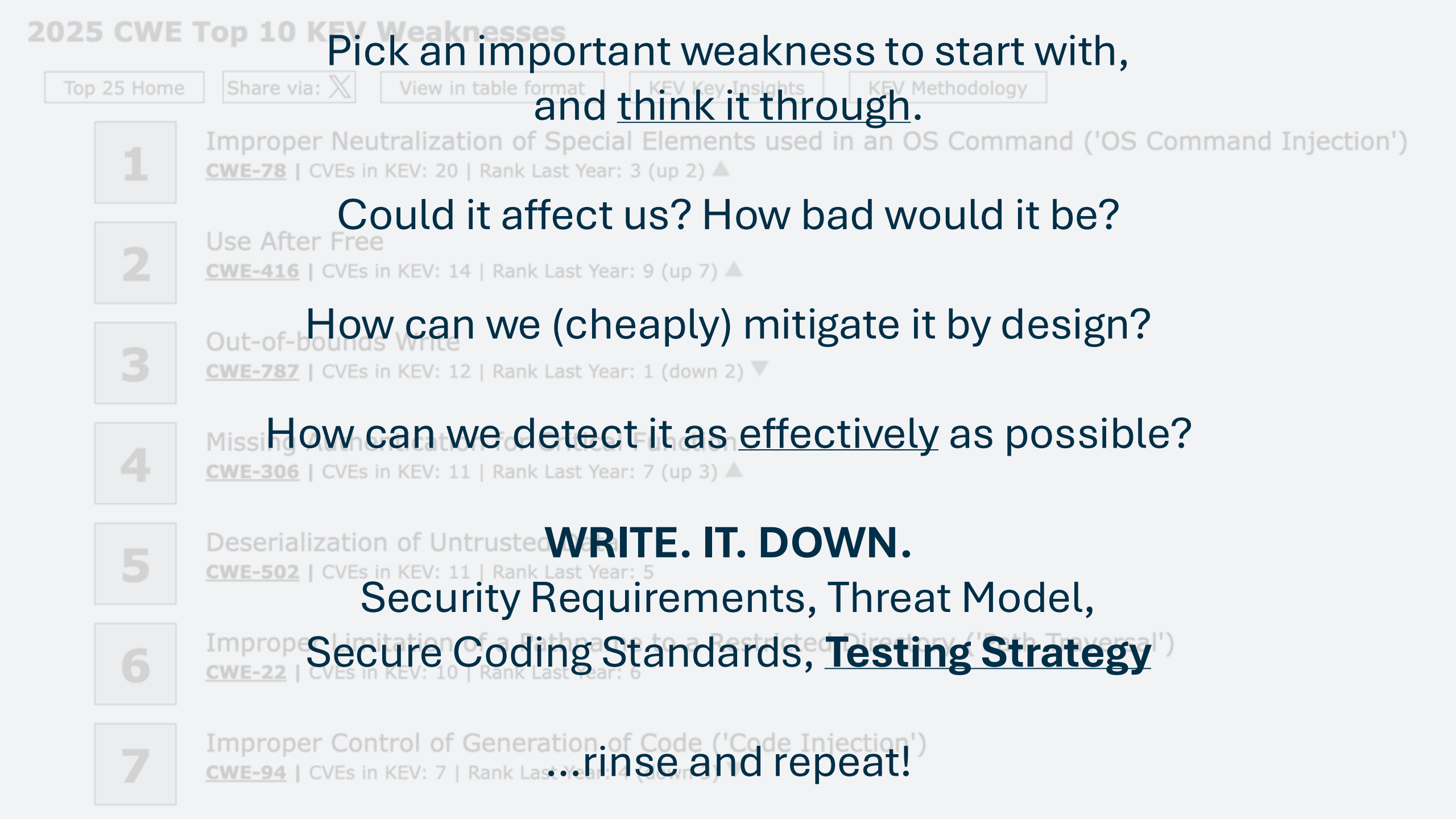


2.



1. Draw some circles

2. Draw the rest of the fucking owl



Pick an important weakness to start with,
and think it through.

Could it affect us? How bad would it be?

How can we (cheaply) mitigate it by design?

How can we detect it as effectively as possible?

WRITE. IT. DOWN.

Security Requirements, Threat Model,
Secure Coding Standards, **Testing Strategy**

...rinse and repeat!

Recap: Risk Based and Effective Security Testing

Tool-centric security testing is not sufficient

- You, *not the tool vendor*, are accountable for the testing strategy.
- Active threats and legislation mandate *risk-based* and *effective* testing
- Justifying your testing strategy is not straight forward

Take ownership of your security testing strategy

- Validating the known is a **mechanical** process
- Discovering the unknown is a **creative** process

Vulnerability *handling* is not just vulnerability *fixing*

- Reflect on *every* detected vulnerability and meaningfully improve the SDLC.

Contact

TOREON

Grotehondstraat 44 1/1
2018 Antwerpen
België

+32 3 369 33 96

Courbevoie 13
1348 Ottignies-Louvain-la-Neuve
Belgique

+32 1 079 00 10

www.toreon.com

toreon

info@toreon.com

toreon_BE