Automated Vulnerability Testing

on the application level

Radboud University, 27 September 2024

Frans van Buul
fvbuul@opentext.com

Introducing myself

- Application security specialist. Currently product manager for Fortify SAST, SCA and AI (we'll talk about what all of that is)
- Before that, worked as
  - Appsec Presales engineer
  - Java developer/architect at various companies for around 5 years. Still coding in my current work.
  - security consultant/auditor at PwC for some 8 years – my connection with Eric Verheul.
- Nijmegen alumnus – completed theoretical physics master in 1999.

Fortify by opentext™

(before "OpenText", Fortify was part of Micro Focus, before that HP, before that an independent company)

Java

pwc

Radboud Universiteit

Feel free to reach out: fvbuul@opentext.com

Agenda

- **The problem space**: What are we looking for, and why?
- **Technical solutions**:
  - Requirements. What would an ideal solution look like?
  - Today's (main) approaches: SAST, SCA, DAST
- **(Quick) market overview**, vendors and open-source projects implementing these approaches.
- **How is Generative AI changing this landscape?**

Questions and comments at any time are much appreciated!

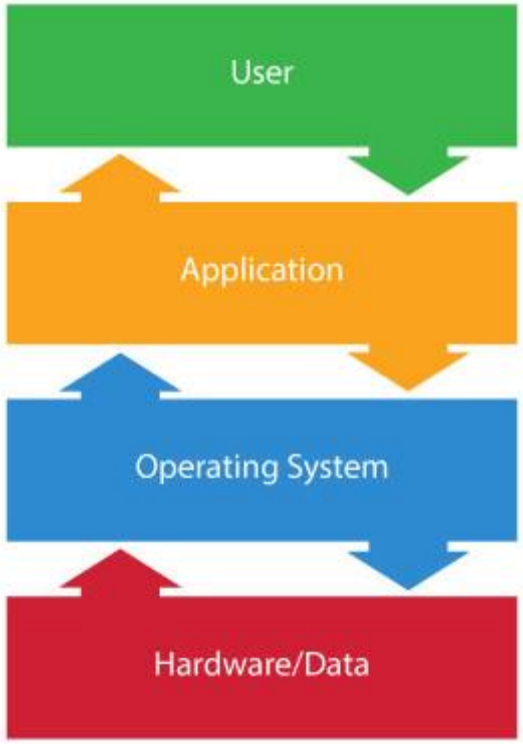
The problem space

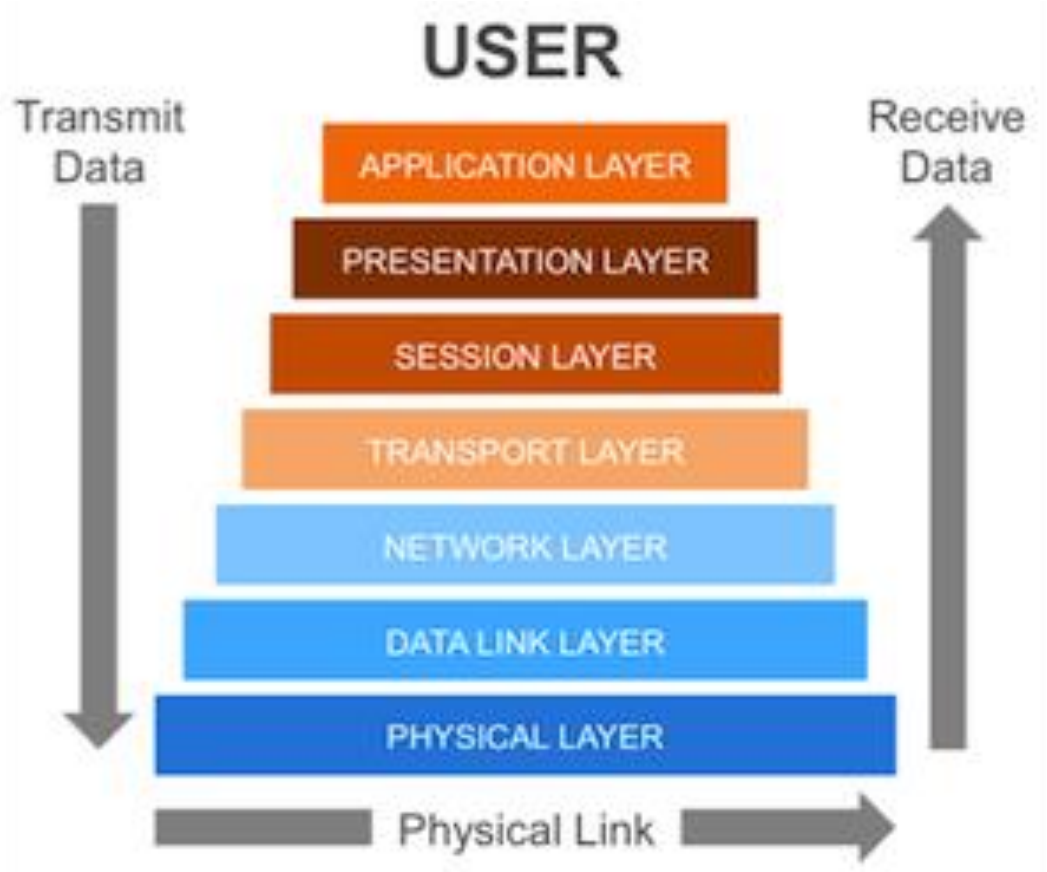
What are we looking for, and why?


The security of
controlled access

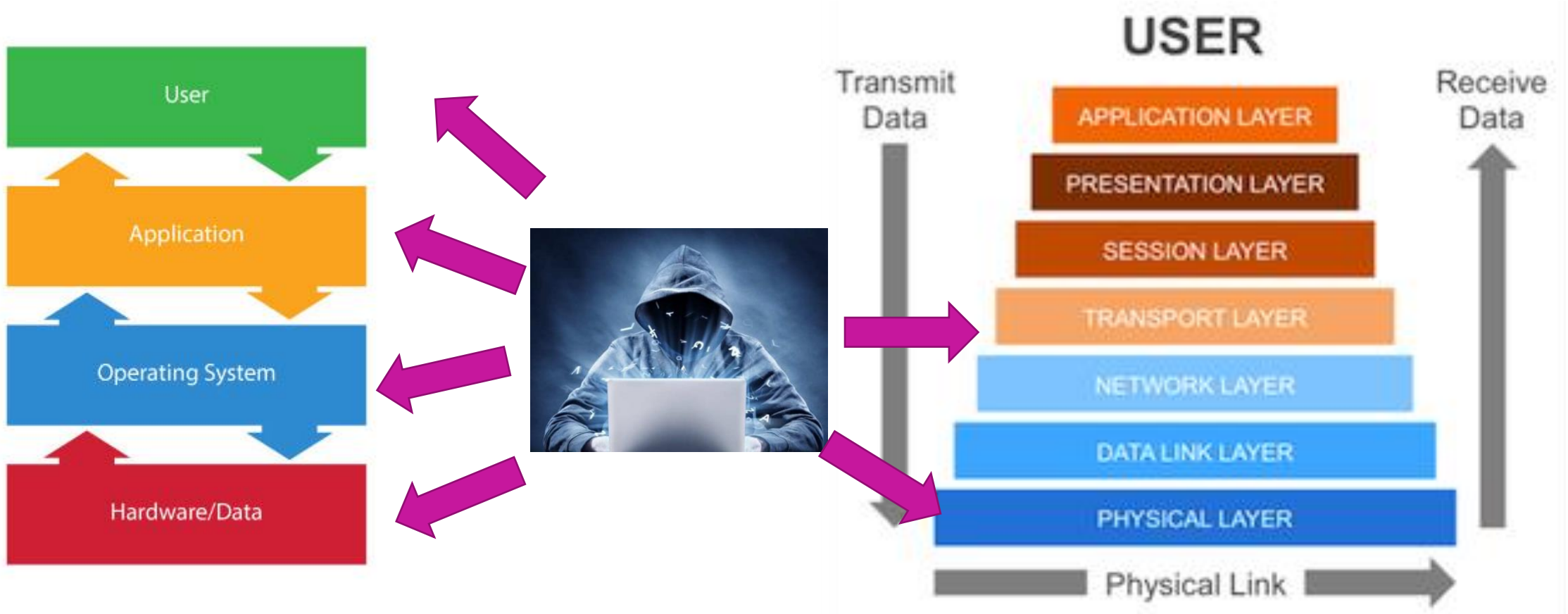

The security of controlled access
Required for
The security of not bypassing security functionality; *baseline* security

IT systems and networks are *layered.* The layers represent increasing levels of abstraction.

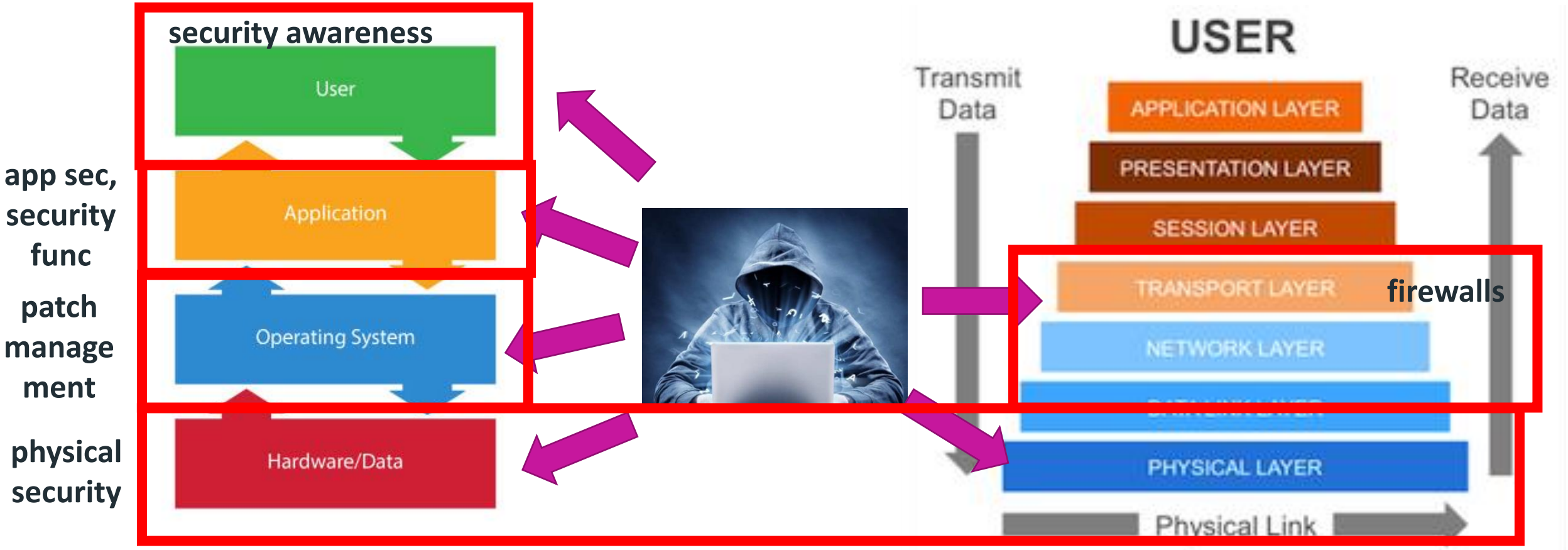




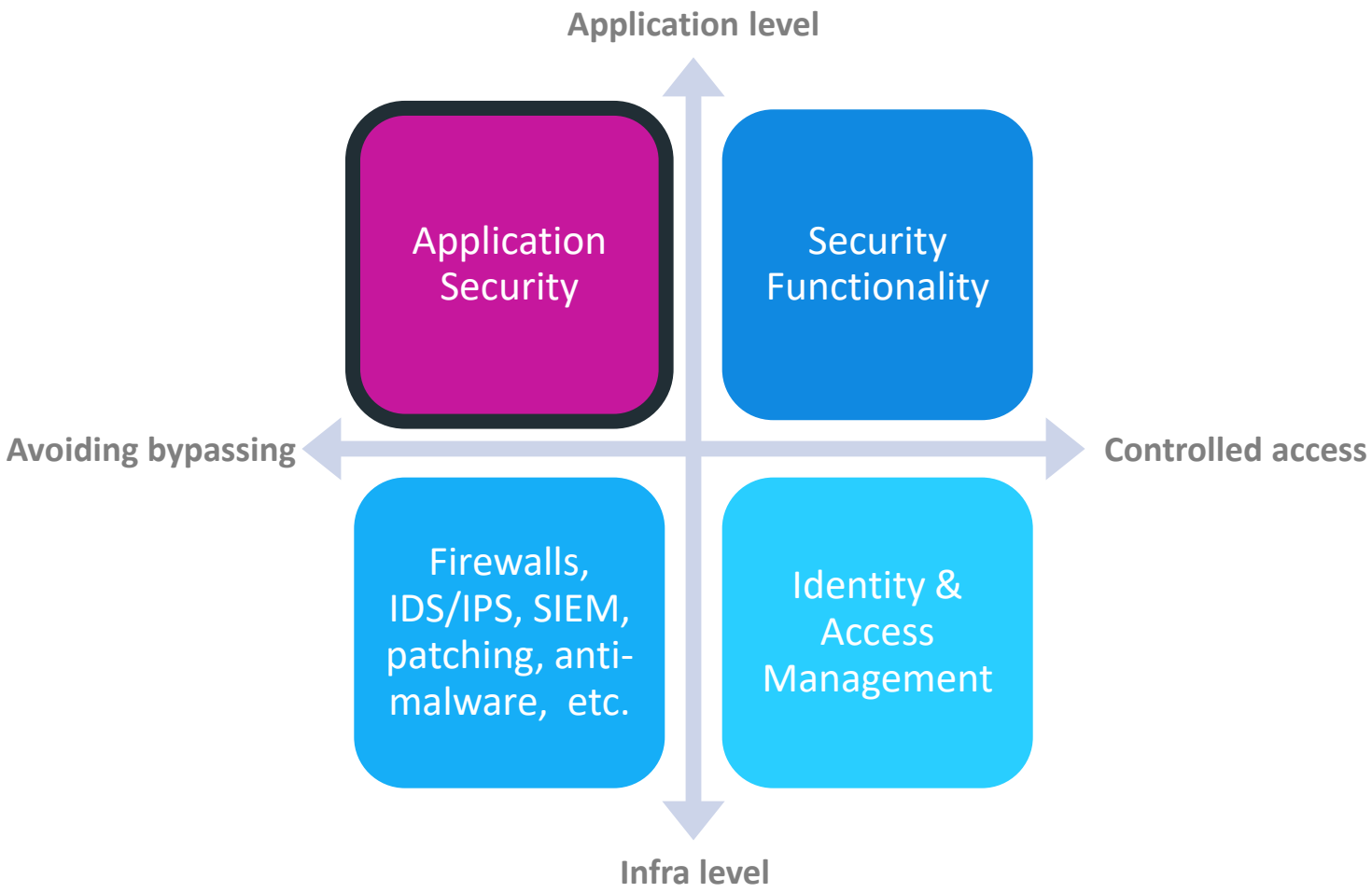
An attacker can potentially target any layer



Defensive measures act on specific layers

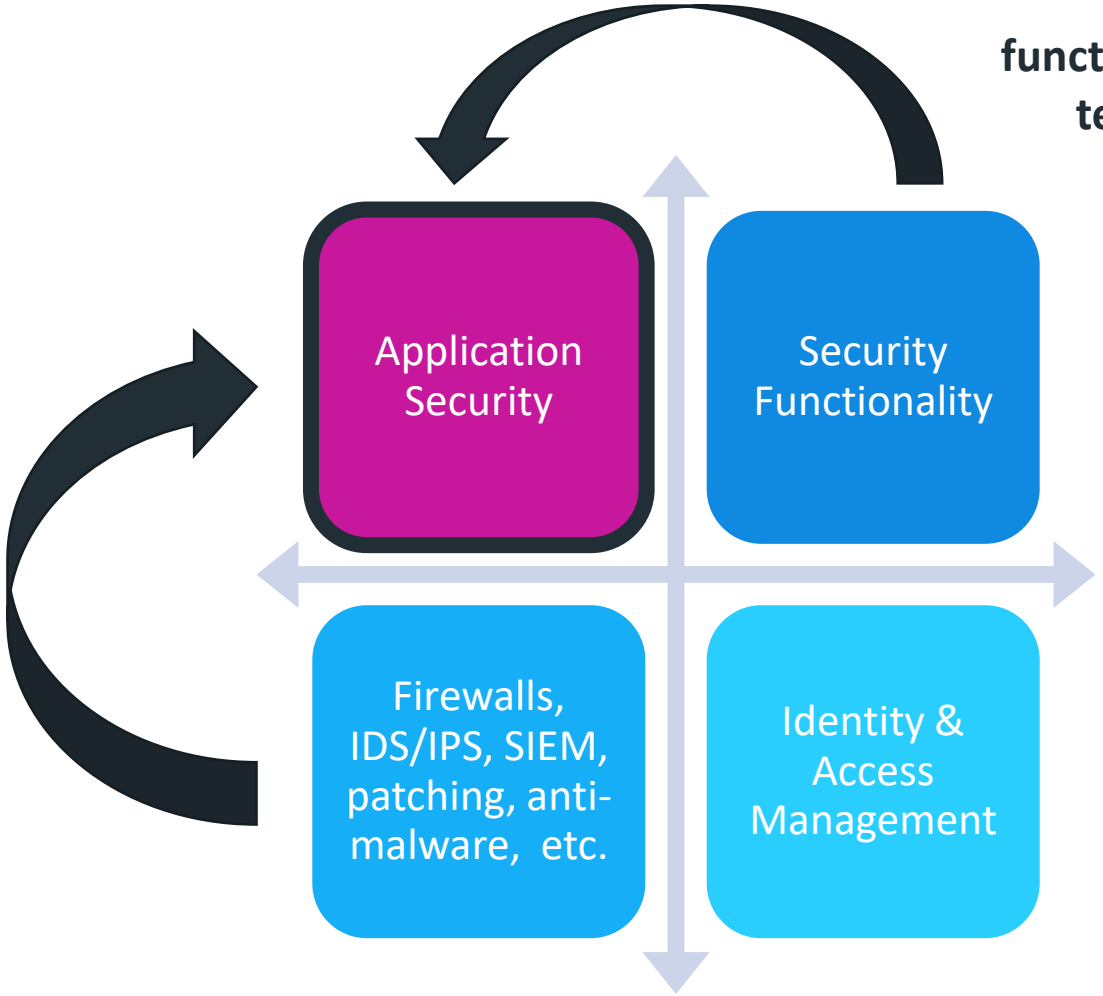


Combining both dimensions: a security quadrant



AppSec needs specific attention

Testing for security functionality is different from testing for application security!



Infra-level security measures do not protect against this type of problem!

Types of applications

Standard

- “COTS” / open source
- Used by many organizations
- Actively researched for security issues.
- Known vulnerabilities (CVE = Common Vulnerability Enumeration)
- Key security measure: **apply patches**, and configure securely

Custom

- Developed in-house or by 3rd party.
- Used only in the own organization.
- Therefore, not actively researched by the security community.
- No patches unless you write them yourself.
- May have weaknesses (CWE = Common Weakness Enumeration)

Custom applications are the most important target for application vulnerability testing.

The focus of this lecture

- …finding vulnerabilities
- …on the application level
- …in the baseline (as opposed to “security functionality”)
- …in custom applications.

Before we go into techniques to do that, we’ll cover:

- examples based on OWASP Top-10 to make this more concrete (with demo);
- the issue of open-source dependencies;
- drivers to do this automatically.

Looking into AppSec issues
OWASP Top-10 2021

A01 Broken Access Control	A02 Cryptographic Failures	A03 Injection	A04 Insecure Design
A05 Security Misconfiguration	A06 Vulnerable and Outdated Components	A07 Identification and Authentication Failures	A08 Software and Data Integrity Failures
	A09 Security Logging and Monitoring Failures	A10 Server Side Request Forgery (SSRF)	

Still the latest as of September 2023, although newer *specialized* OWASP Top-10s exist, like the API Security Top 10 2023.

Live demo SQL injection and cross-site scripting

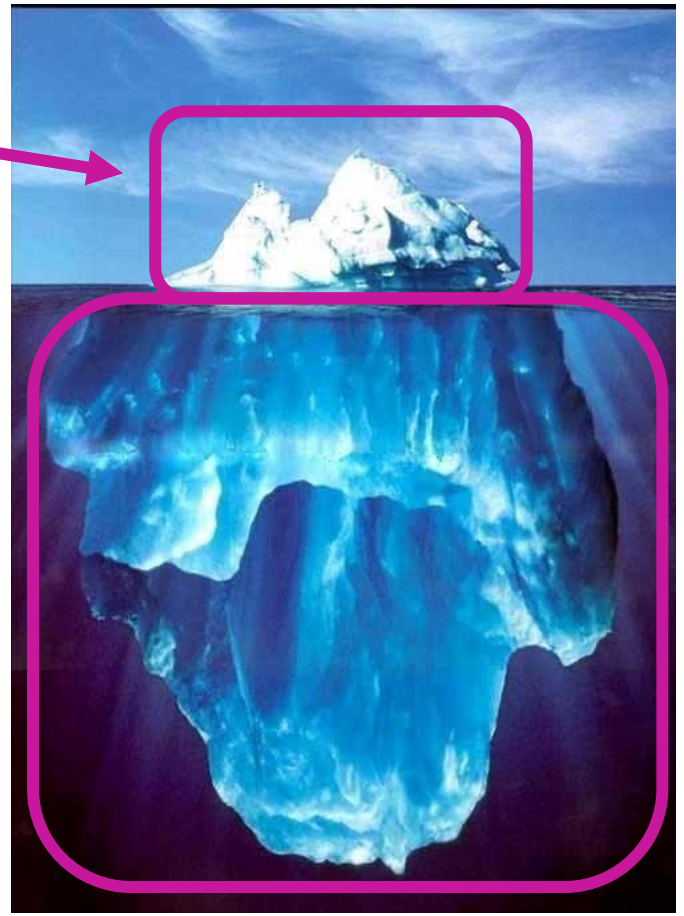
Both are OWASP Top-10 2021 A03 Injection

In a modern business application, how much of the code is usually open source?

1. 0 – 5%
2. 20-50%
3. 60-90%
4. >95%

A modern business application

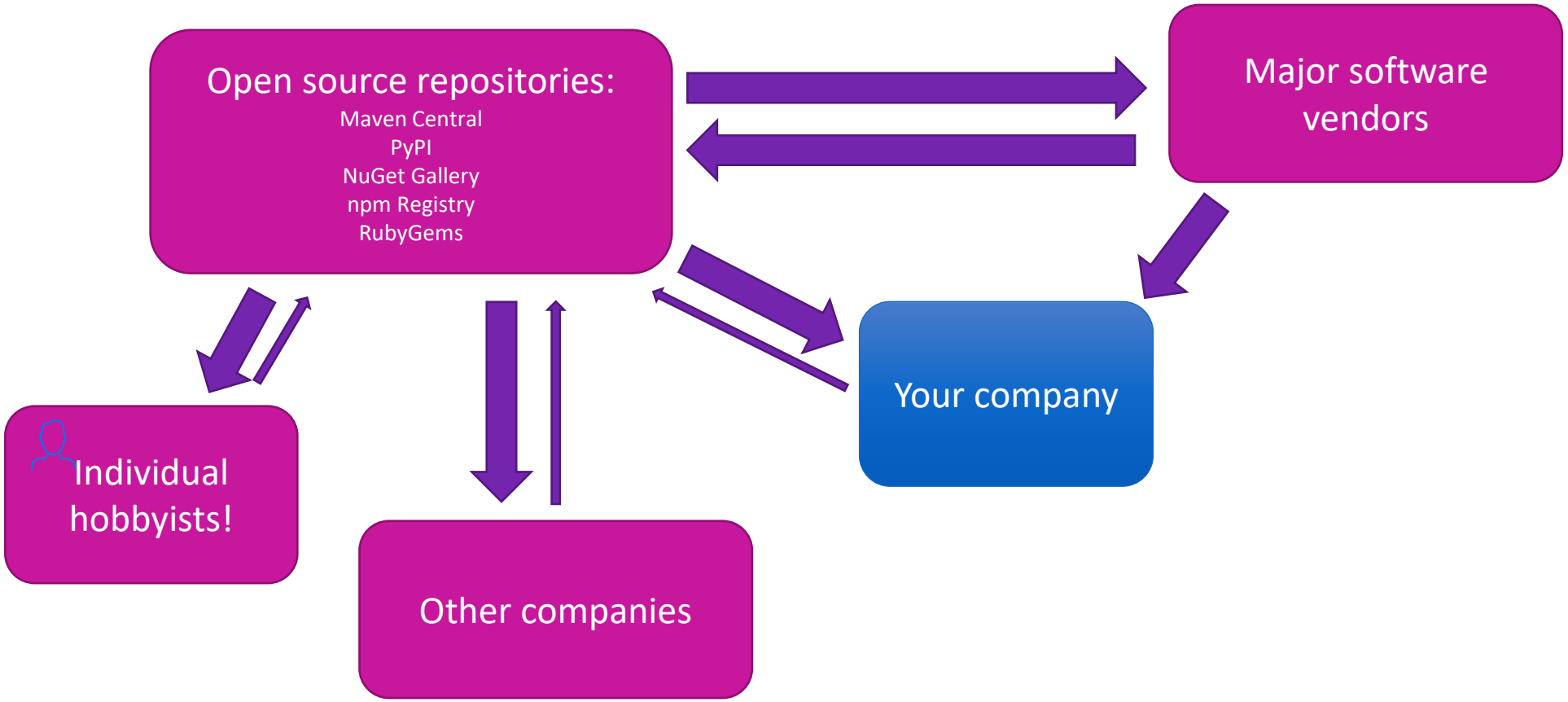
The code that developers write themselves in Java, C#, Python, etc.



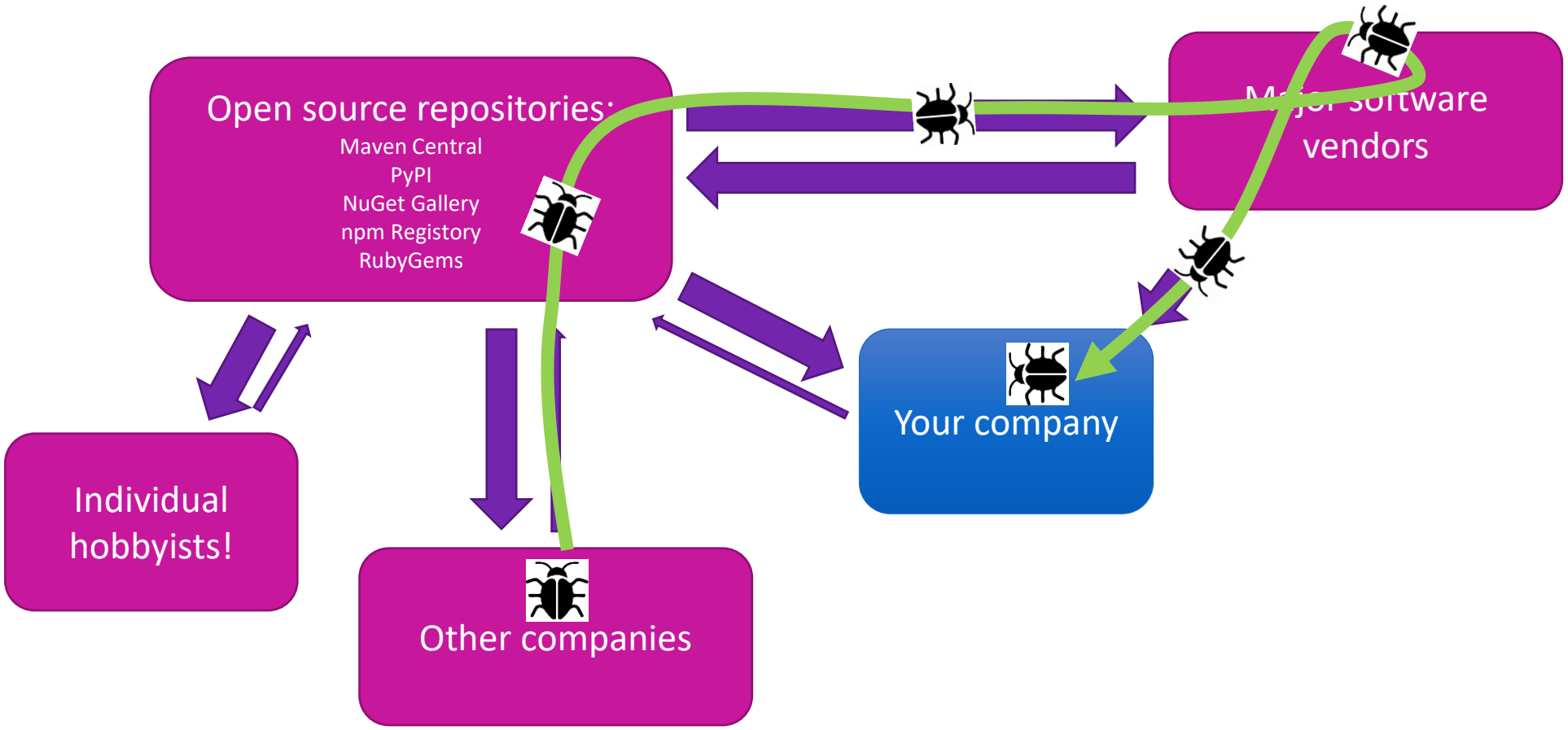
The open source code that developers include in apps through Maven, Nuget, pip, npm, etc.

This is 80-90% of the app!

Development is now a *connected* activity



…where bugs can fly around



What does open source usage by developers look like?

- Downloads happen automatically based on project configuration.
- Dependencies may automatically pull in other, “transitive” dependencies.
- In most organizations, there is no control on this downloading process.
- Getting all dependencies in compatible versions is sometimes difficult. Developers are happy when it works. And: “if it works, don’t fix it!”
- No incentive at all for developers to periodically review dependencies and update them when necessary. This will usually only happen when there is a functional reason.

Altogether, this creates substantial risk that insecure libraries are present in business applications.

Security for the modern business application

The custom code

The OSS dependencies

Vulnerabilities may be present in either part. You must be in control of both.

But, the method of doing so is totally different!

A security tool must scan your custom code for **unknown** vulnerabilities.

Recommendations should include how to **change the source code**.

A security tool must scan your bill-of-materials for **known** vulnerable components.

Recommendations should include how to **switch to a better version.**

How can you find app issues (either in your code or in dependencies) before hackers do?

- Manual approaches:
  - Code review
  - (Application-level) pentesting
- Probably, these are still the most reliable ones!
- However, their scalability is extremely limited.

The economics of appsec automation

Business / IT development

- Ever faster release cycles. Agility!
  - ACCELERATE “State of DevOps 2022”: 17% of respondents release more than once per day, another 62% at least once per month.
- Introducing DevOps culture and practices, automation to support this.
- Fast growing application portfolios.
- Fast growing DevOps teams.

Security

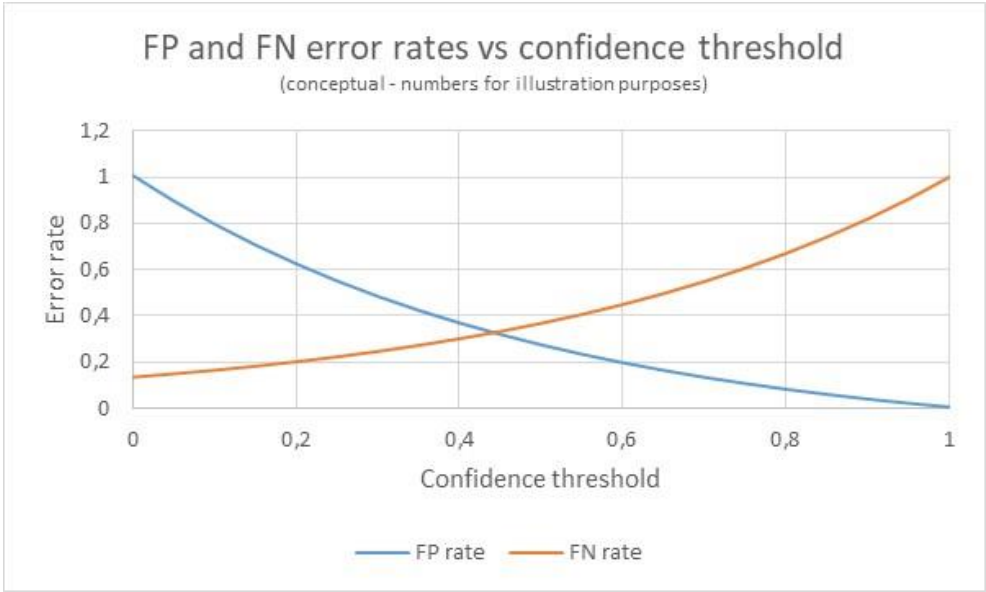
- Security remains important; needs security approval of every release put into production.
- But not growing in terms of staffing.
- Often kept out of the DevOps loop. Different culture.
- Manual reviews take weeks and cost several €10k per release.

1000 apps * 50 release/year = 50k checks/year
Manual review doesn’t work. You need to automate.
It’s not about the tools being “better” than an expert.

Technical solutions

Some notes on testing tool reliability

- When evaluating the reliability of appsec testing tools (or biometry devices, or COVID-19 tests), there are different types of errors to consider:
  - False Positive (FP): test says something is the case, that in reality is not the case
  - False Negative (FN): test fails to report something that is there in reality
  - FP and FN rates are a trade-off. Raising the “confidence threshold” reduces FP rate at the expense of increasing the FN rate.



FP and FN in software security testing

- For software security testing, it's not possible to express a tool's FP and FN rate as a number because we don't have something like a clear universe of software or a representative sample thereof. These rates are qualitative and/or context-dependent.
- In reality, "FP" is a somewhat blurry notion.
  - "Hard" FP, tool shouldn't have reported anything, tool bug.
  - "Unavoidable" FP, it's an FP but with current state of art the tool can't do better.
  - Issue is correct, but risk categorization too high.
  - Issue is correct, but there are mitigation measures outside of the software (contextual factors) making it irrelevant.
  - Issue is correct, but risk can be accepted for this application (log forging is a prime example).

FP or FN: what's the bigger concern?

1. False Negatives
2. False Positives
3. Equally Important
4. It depends

The ideal appsec testing tool

Property	Should be	Explanation
Applicability	Universal	No restrictions on types of applications/languages that can be scanned.
Issue scope	Universal	No restrictions on types of issues to be found, including both custom and open source issues.
Speed	Infinite	No delays
Reliability: false negative rate	Zero	Not missing any issues that should have been reported.
Reliability: false positives rate	Zero	Not reporting any issues that are irrelevant or untrue.
Integration in CI/CD tooling	Universal and simple	The tool should be easy to integrate with build servers (Jenkins etc.), IDEs (Eclipse, Visual Studio etc.), source code repos (Git etc.), issue trackers (JIRA etc.)
Ease of use	Maximal	Easy to set-up and use, without training.
Feedback	Detailed and actionable ***Trend: auto-remediate***	Fix the reported issue in the correct way, without thinking. It would be even better if the tool would just fix it automatically.

Of course, this ideal tool doesn't exist!

Techniques we'll discuss

- **SAST** – Static Application Security Testing. "Automatic code review"
- **SCA** – Software Composition Analysis. "Open source/bill of material review"
- **DAST** – Dynamic Application Security Testing. "Automatic Pentest"
- Typical deployment scenarios

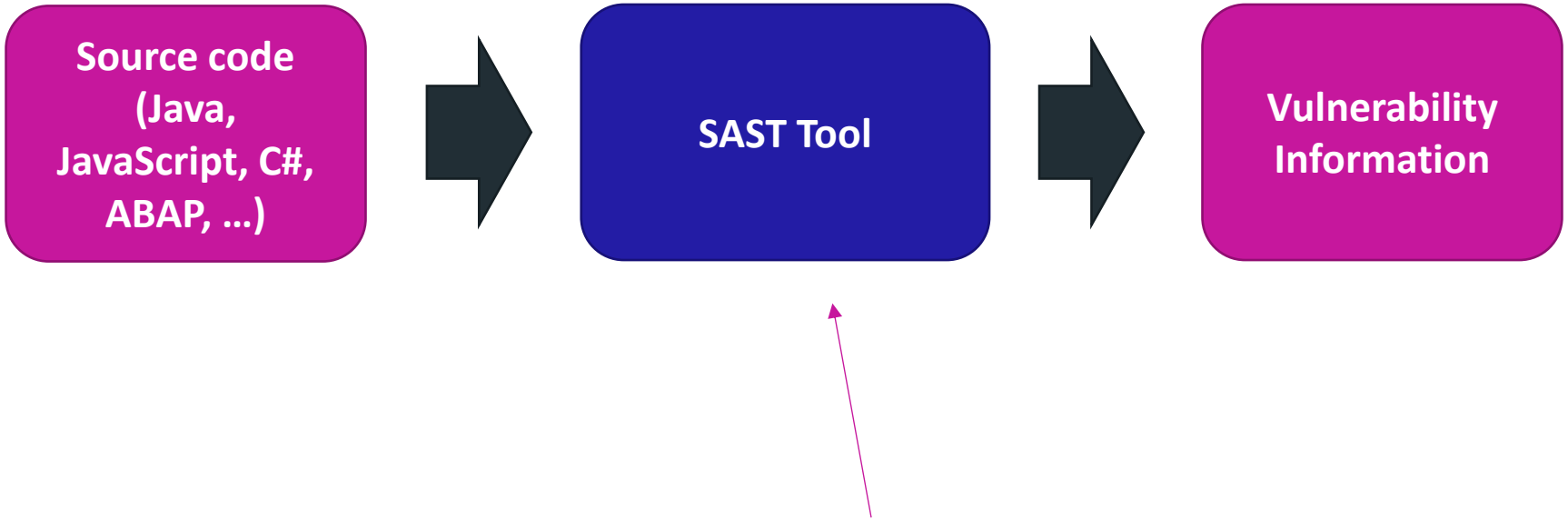
We'll skip the following related techniques that are about protection rather than detection:

- RASP – Runtime Application Self Protection
- WAF – Web Application Firewall

We'll also skip (mobile) binary analysis, sometimes called MAST, and IAST – Interactive Application Security Testing, as they are somewhat edge cases.

Technical solutions - SAST

SAST process – the big picture



What happens under the hood?

Multiple algorithms

- Ways to look at a codebase.
- E.g. data-flow-analysis, structural analysis.
- Any decent SAST tool uses a variety of algorithms.
- By themselves, do not have “content”.
- Nearly all algorithms presume that the tool can parse the code into a syntax tree.

The SAST tool must understand the language

Many rules

- “If a Java method parameters is annoted as @RequestParam, it will receive user input.”
- “User input being sent to a JDBC query triggers a SQL injection risk”
- “Something like ‘String Password = “bla”’ may represent a hardcode password.”

The SAST tool must understand the frameworks.

SAST algorithms: structural analysis

- Evaluates expressions on the syntax tree to find bad patterns.
- Simple, fast algorithm.
- Works well for things like
  - Hardcoded password and keys
  - Weak encryption settings, insecure randomness
  - API abuse
- Also performed by many code quality tools, not just security.

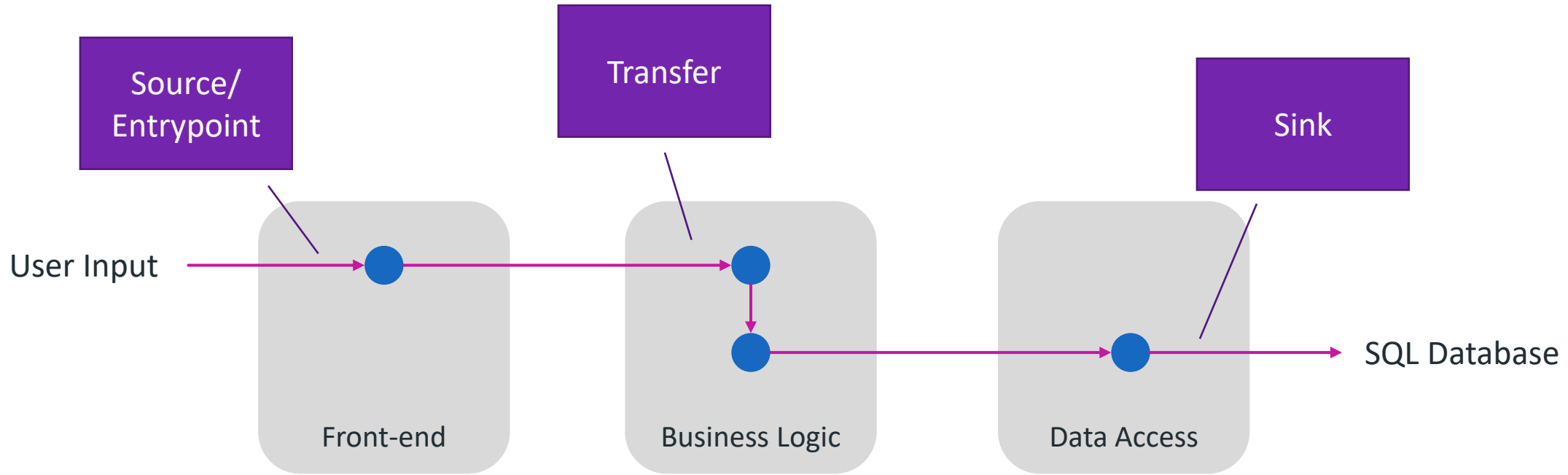
SAST algorithms: buffer analysis

- Tries to evaluate the allocated size of the buffer and the actually used size of the buffer.
- Flags overwrites/overreads.
- Specific to compiled, unmanaged languages like C/C++.
- Inherently difficult problem to evaluate, won't catch everything.

SAST algorithms: Taint Analysis

- Essence: tracking **taint**, which is the property that a piece of data may contain an attack payload.
- Simplest example: any piece of user input should be considered *tainted.* Such places are **sources** of taint.
- Taint can be **transfered**:
  - suppose the value of variable x is tainted;
  - and then y gets assigned the value of x;
  - then y is tainted as well.
- Commercially, this is also referred to as “dataflow analysis”. Scientifically, the word “dataflow analysis” has a broader meaning that, for example, also covers buffer analysis.

SAST algorithms: Taint Analysis



Why is taint analysis so important?

- Many issues can be found like this: SQLi, command injection, XSS, privacy, etc.
- Important source of *critical* findings:
  - Issues are reliable.
  - Issues have high impact.
- Not covered by standard quality-oriented tools.

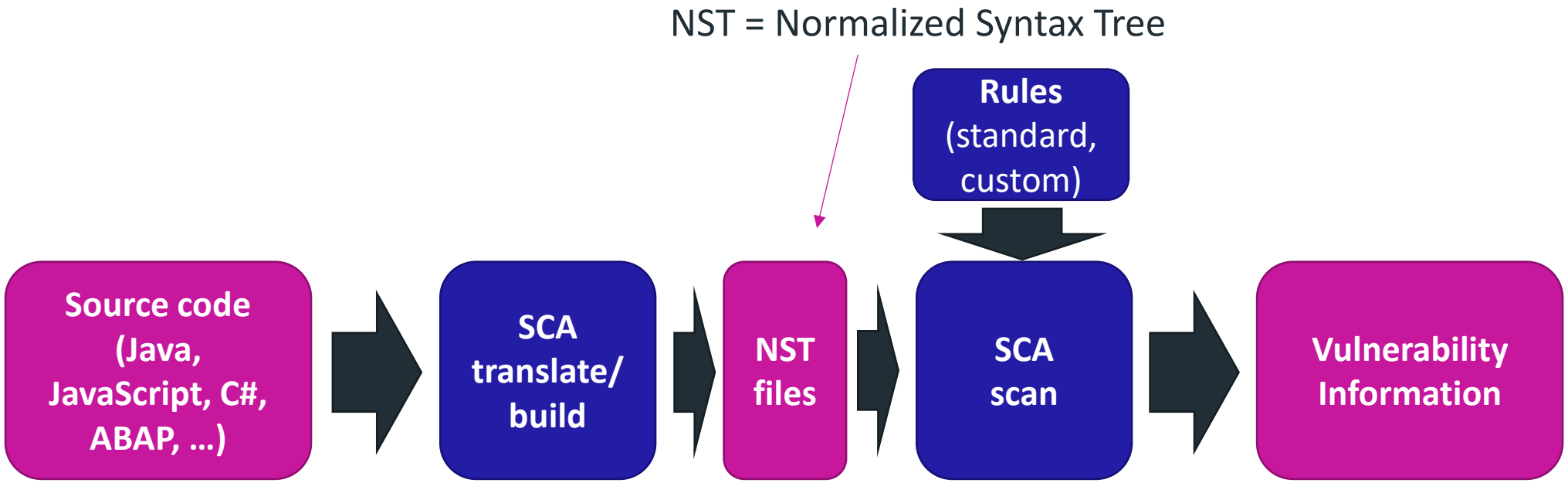
SAST algorithms - others

- **Control-flow analysis. Is the program always in a secure state?**
  - Modeled as a state-machine.
  - Finds things like unreleased resources.
- **Regular expression matching**
  - Super basic algorithm; but actually useful to find things like hardcoded access tokens.
- **Higher-order analysis**
  - Not a detection algorithm in its own right, but required for data-flow and control-flow analysis on languagues which allow assigning functions to variables.
- **Constant propagation**
  - Needed to make many other things work.

Code Security or Application Security?

- **SAST is going beyond “application security” because of the everything-as-code trend**
  - **Infrastructure-as-code** (often related to cloud; think Terraform, Ansible, etc.)
  - **CI/CD-as-code** (e.g. GitHub workflows)
  - **Contracts-as-code** (Smart Contracts, in particular Solidity living on Ethereum Blockchain)

SAST implementation in Fortify – 2 step approach



Translate: implemented separately for various supported languages, using different architectures.

Scan: one technology for all supported languages

Evaluating SAST

Property	Should be	For SAST
Applicability	Universal	You need to have the source code, and the source code + libraries must be supported by the SAST tool. Doesn't need a web interface; works on web, mobile, desktop, embedded, …
Issue scope	Universal	Works for custom code. Use on open source code is theoretical.
Speed	Infinite	Minutes to hours, depending on code size. Small subset can be done in realtime using "security spell-checkers".
Reliability: false negative rate	Zero	Strong point of SAST. Can be pretty good. Will not cover business logic issues.
Reliability: false positives rate	Zero	Weak point of SAST. Can be very high if not properly tweaked.
Integration in CI/CD tooling	Universal and simple	Can be done; is highly vendor dependent.
Ease of use	Maximal	Basic operations are usually simple, but advanced tweaking to optimize FN and FP rates require deeper understanding.
Feedback	Detailed and actionable	In principle, can provide detailed feedback to developers on what to fix.

SAST Demo

Technical solutions - SCA

Software Composition Analysis (SCA)

- This is conceptually and technically *a lot* simpler than SAST.

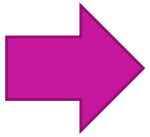
- Essentially, just 2 steps:

  1. Determine the list of libraries/dependencies present in a piece of software. (The so-called "Bill-of-Materials".)

  2. Check these against a database of known vulnerabilities.

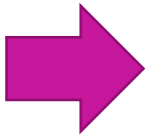
A naive SCA implementation

1. Determine the list of libraries/dependencies present in a piece of software. (The so-called “Bill-of-Materials”.)



Most build environments (e.g. Maven in Java) can produce the list of dependency coordinates, name and version. This gives you the BOM.

2. Check these against a database of known vulnerabilities.



The National Vulnerability Database offers machines-readable CVE data, where you can query for the elements of the BOM.

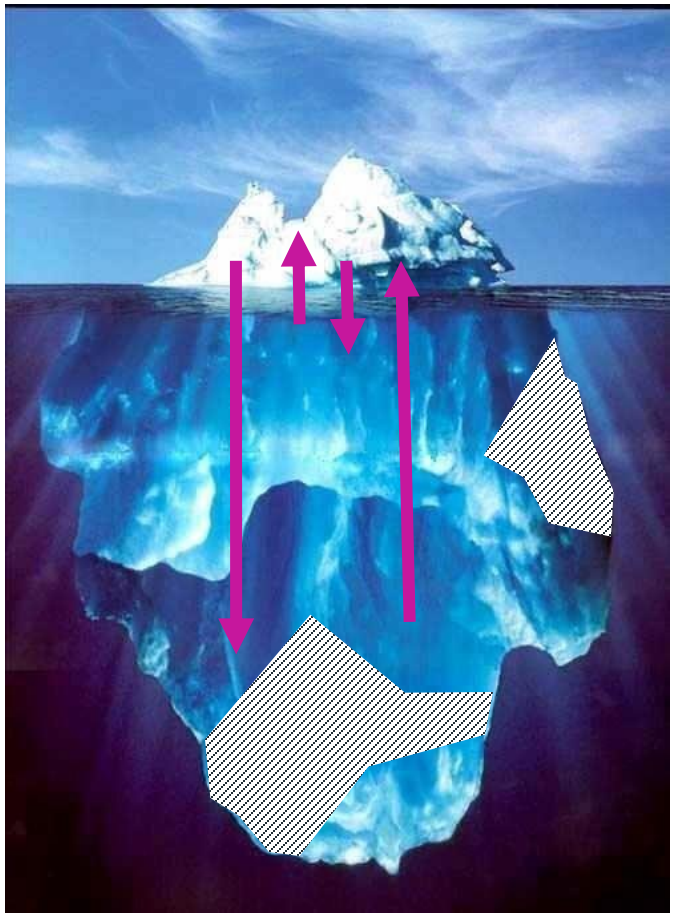
Limitations of this naive implementation

- It will break on renames, forks etc. of dependencies.
- Because of limitations in the dataset:
  - Both false positives and false negatives
  - Limited advice to the developer if something is found
- Only covers pure security issues. In reality, use of open source also introduces
  - Licensing risk (e.g. you're not allowed to use a GPL-licenses library in a piece of software you'll license commercially)
  - Architectural risk (making your product dependent on an open source library that hasn't been maintained for 3 years is often a bad idea regardless of currently known security issues)

SCA – more advanced implementations

- Obtain the BOM by calculating hashes of all the files in a piece of software rather than relying on dependency names.
- Test against a dedicated open source dependency database rather than the general NVD.
- Provides advise to change to a better version, taking into account transitive dependencies.
- Check for security, architecture, license. Allows configurable policies.
- Can operate in a detective setup (scanning) but also in a preventive setup (avoiding download of bad libraries).

Revisiting our application



- The custom code and the included open source code in an application are not independent; they call eachother and data flows between them.
- Open source libraries are usually included for a reason, but never used *entirely.* It's completely normal for an application to ship with significant amounts of unused code.

What if…

… your SCA tool detects your application is using a library with a known critical vulnerability?

What if…

… your SCA tool detects your application is using a library with a known critical vulnerability?

It depends!

- The application is actually using the vulnerable part of the library and this usage is controllable by attackers ➔ **Probably a critical problem in your app**
- The application is using the vulnerable part of the library, but not in a way that is controllable by attackers ➔ **May be an issue, further research makes sense**
- The application is not using that part of the library ➔ **Housekeeping issue**

Susceptibility/reachability analysis

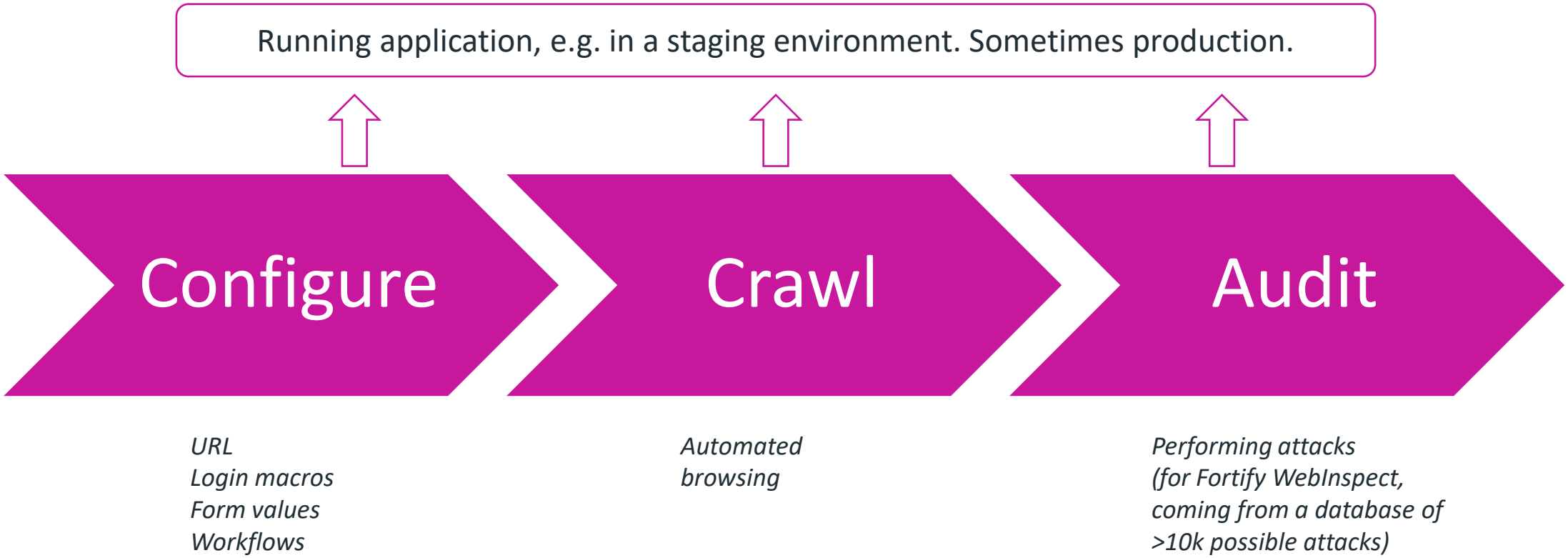
- SCA alone is not enough to find these vulnerabilities and make the distinction between the cases.
- Functionality to make this distinction is called “susceptibility analysis” or “effective usage analysis”.
- Requires a combination of SAST features (data-flow or structural analysis) and SCA features.

Evaluating SCA

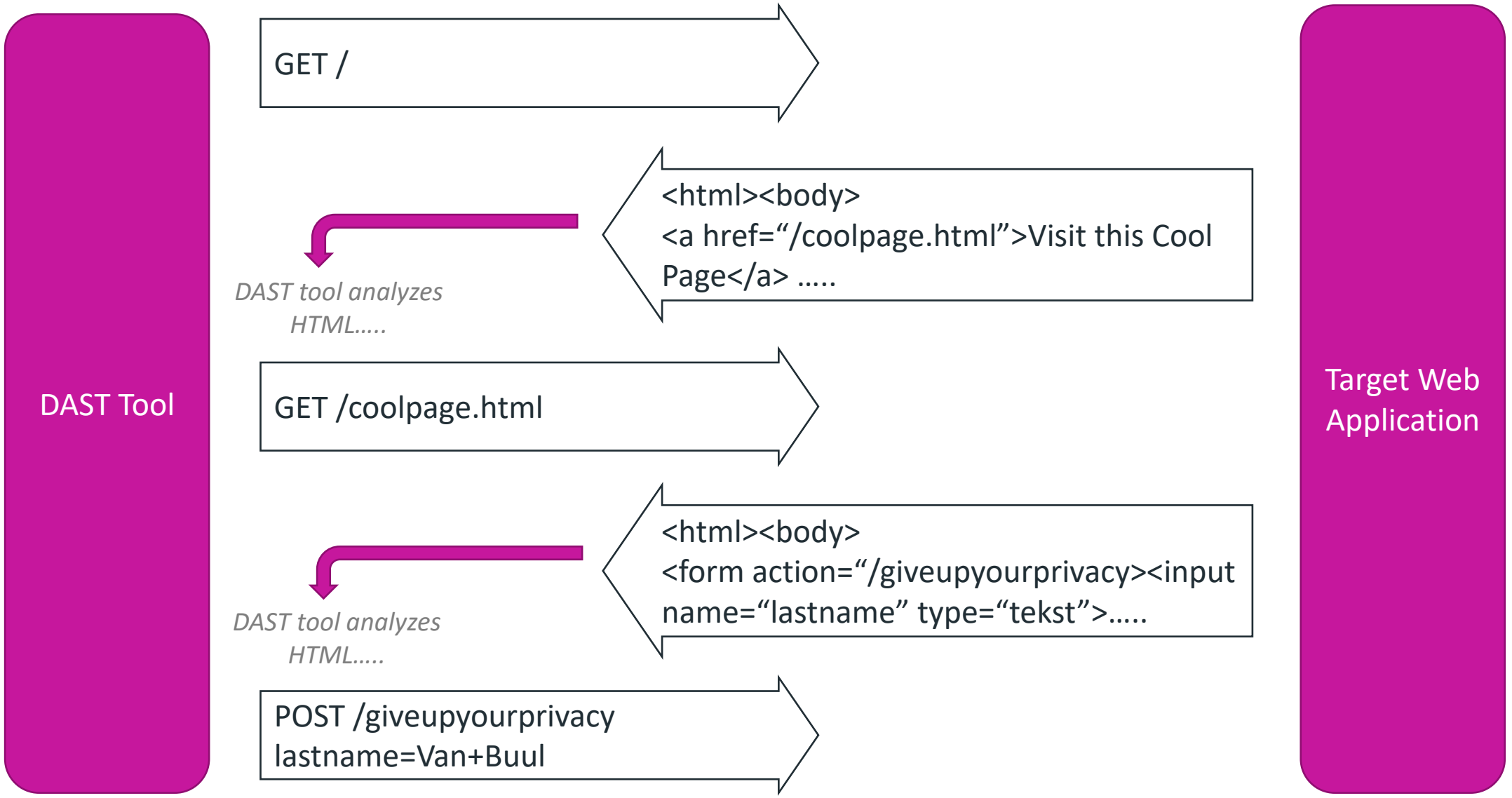
Property	Should be	For SCA
Applicability	Universal	Source code language/ecosystem needs to be supported. Actual source code may or may not be necessary.
Issue scope	Universal	Limited, only for open-source dependency issues, not for your own code.
Speed	Infinite	Very fast. Minutes, max.
Reliability: false negative rate	Zero	Dependent on the quality of te database, but can be really good.
Reliability: false positives rate	Zero	May produce many results that don't lead to exploitable vulnerabilities. Susceptibility analysis helps here.
Integration in CI/CD tooling	Universal and simple	Can be done; is highly vendor dependent.
Ease of use	Maximal	Generally, easy to use.
Feedback	Detailed and actionable	Vendor-dependent

Technical solutions – DAST

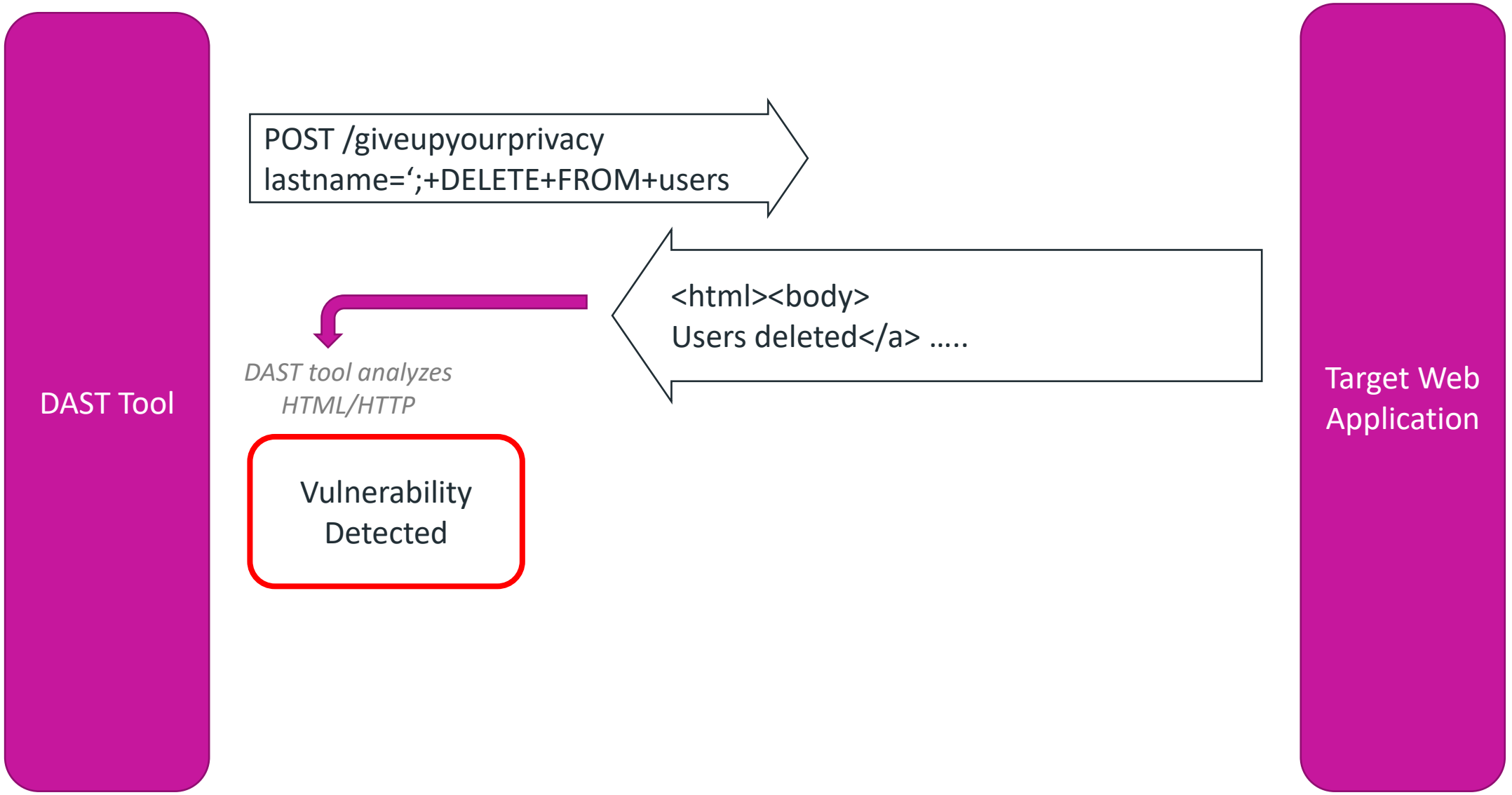
DAST operations



Crawling (naively)



Auditing (naively)



Some crawling challenges

- Some knowledge on acceptable answers may be required to pass basic input validation.
  - E.g. postal code in NL is something like “3456 EF”
- In some cases, the DAST tool will need even deeper knowledge to consistently walk through a website.
  - E.g. applying for an insurance policy, doing a payment – might need workflow.
- In most cases, authentication is required to get to the interesting parts.
  - Basic case: recording a login macro, configuring client certificate
  - More advanced case: setting something up to bypass non-automatable login
- There may be scan “black holes” to be avoided, e.g. all product pages on a big e-commerce site.

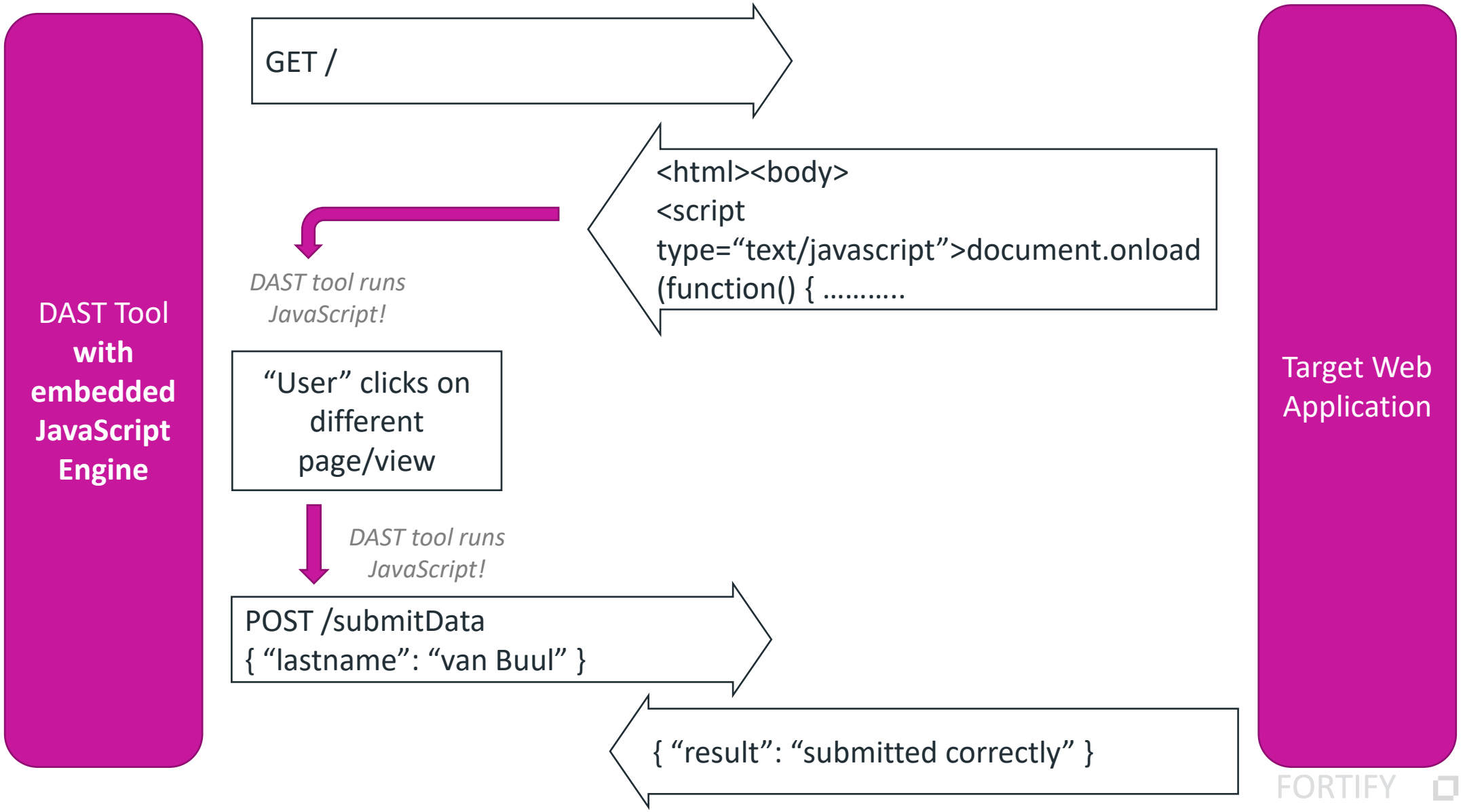
Some auditing challenges

- Trying a database of 10,000 attacks on 1,000 pages won't work. Auditing needs to be smart.
- Detection of success/failure is not always easy.
  - 404-response checking is a very elementary optimization in this area
  - In addition to analyzing the response data, analysis of response *timings* is also used.
- An attack may fail even though in principle it could be successful
  - Stale Anti-CSRF token is a common case.
  - There may even be anti-scanning security measures, blacklisting etc. in place in the infrastructure
  - The process of scanning may easily overload the (staging) server. Manage parallelism and monitor behavior of the system-under-test.
- Successful attacks may corrupt the system-under-test and/or bring it down.

DAST – advanced/modern cases

- Single Page Applications (SPAs)
- (REST) APIs

Crawling – modern Single Page Apps



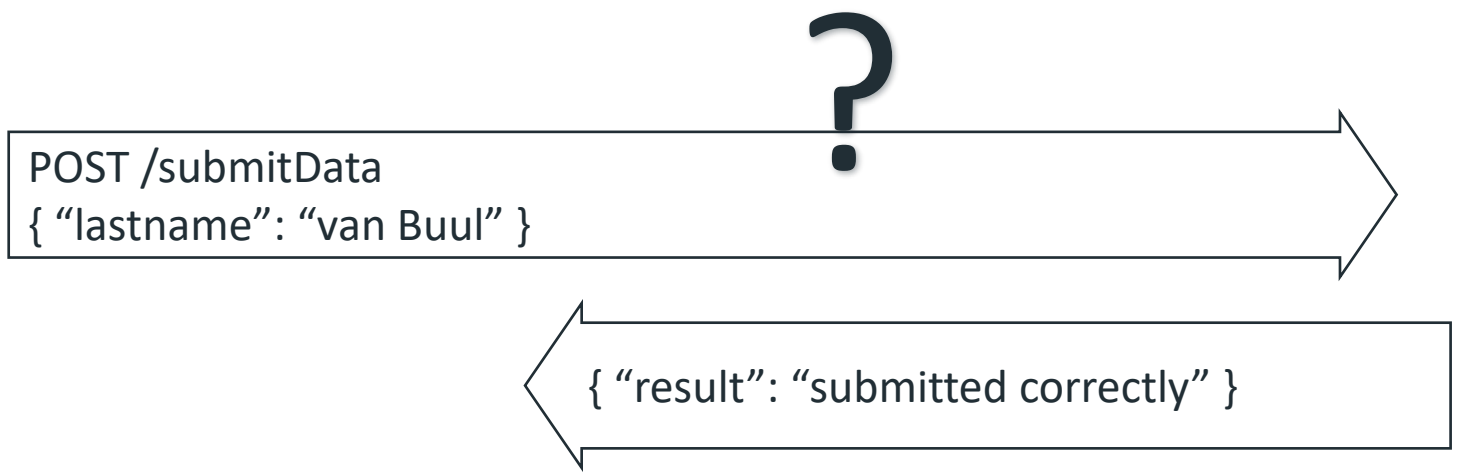
Crawling – the API challenge

DAST Tool with embedded JavaScript Engine

An API is like a modern single-page application, but only the data-exchange parts.
No HTML/JavaScript.

“Crawling” doesn’t make any sense.

How can the DAST tool know what to attack?



Target Web Application

API only!

How can any DAST tool test a REST API?

- In theory: manual configuration.
  - A lot of work!
- In practice:
  - Either: parse some form of machine-readable REST API description.
    - ~~Web Application Description Language (WADL)~~
    - Swagger
    - Open Data Protocol (Odata) Metadata
  - Or: leverage some form of automatic functional testing of the REST API
    - Postman
    - Any other tool, intercept the functional testing tool's traffic using a proxy.
- (And for completeness: SOAP APIs can be tested by parsing the WSDL, but nobody writes these anymore.)

Evaluating DAST

Property	Should be	For DAST
Applicability	Universal	Language irrelevant, but most practical tools require a web interface (HTTP). Support for advanced cases (SPA, API) varies per tool
Issue scope	Universal	Broad, although some issues can't be found dynamically (hardcoded password, bad practices not leading to a vulnerability yet)
Speed	Infinite	Slowest of all technologies, may takes hours, days in bad cases.
Reliability: false negative rate	Zero	May be quite high and is very dependent on proper configuration. Someone who doesn't understand the field may get a high-FN scan.
Reliability: false positives rate	Zero	Low, because this is based on actual behaviour.
Integration in CI/CD tooling	Universal and simple	Can be done, but more difficult than for SAST, because of the requirement of a deploy, config needs, and longer testing times.
Ease of use	Maximal	Easy to press the start button, difficult to to a really good scan in all cases.
Feedback	Detailed and actionable	By its nature, less detailed than SAST. Recommendations can't refer to the actual source code, so less actionable to developers.

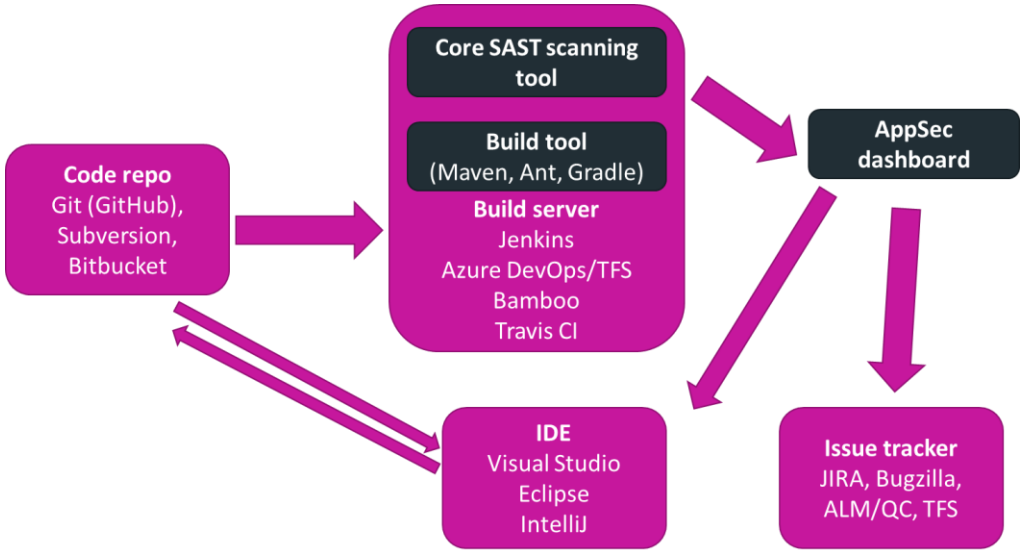
Technical solutions – Deployment scenarios

Automating appsec testing

- In theory, you can use these appsec tools as a thing invoked manually by someone responsible for testing appsec.
- In reality, given the drivers for appsec, deployments focus on
  - automation
  - integration
- Given the limitations that each of the technologies have, companies serious about security combine them rather than pick one.

The typical appsec customer (now)

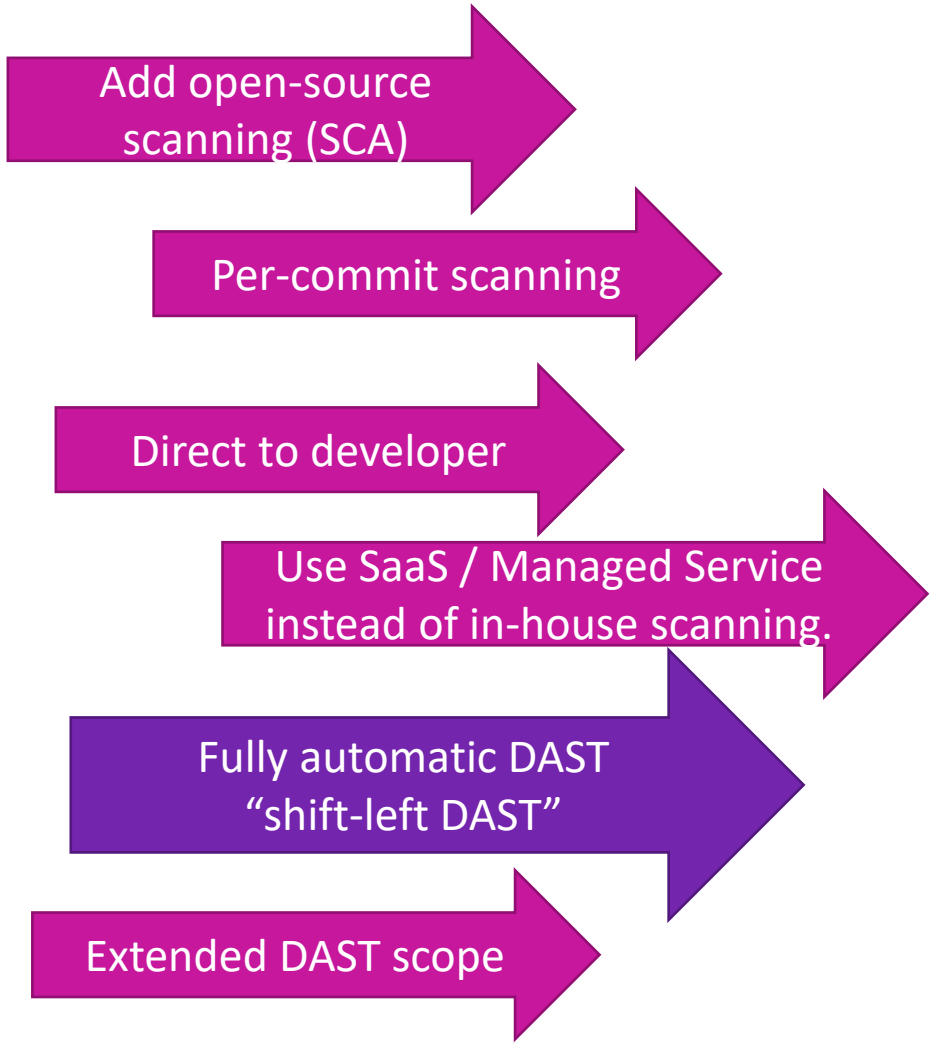
- SAST is integrated in the CI/CD process, for example:
  - SAST is part of the pull request checks and regular build of main. (SAST as part of the “nightly” is a bit outdated in 2023.)
  - Results sent to a central system for triage.
  - SAST issues that warrant fixing are sent from central system to an issue tracker / planning system.



- DAST takes place relatively late:
  - in a staging or even production environment;
  - manually triggered;
  - with limited scope.

Trends: earlier, faster, more frequently, more secure

- SAST is integrated in the CI/CD process, for example:
  - SAST is part of the pull request checks and regular build of main. (SAST as part of the “nightly” is a bit outdated in 2023.)
  - Results sent to a central system for triage.
  - SAST issues that warrant fixing are sent from central system to an issue tracker / planning system.
- DAST takes place relatively late:
  - in a staging or even production environment;
  - manually triggered;
  - with limited scope.



Market overview

Gartner (2023, still current) on appsec market

Gartner (2023) on appsec market

- Most vendors in the “Leader Quadrant” are all traditional, dedicated AppSec players.
- There are differences in approach/focus
  - Synopsys (Coverity) is strong in embedded
  - Veracode is a SaaS-only player focusing on simple, binary scanning.
  - Micro Focus Fortify is a complete offering, covering all functionalities on-prem and in-the-cloud, security focused.
  - Checkmarx tends to be a developers’ favourite, easy to use.
- “Snyk” is different; they come from the SCA space and have broadened to also do SAST.

Gartner (2023) on appsec market

- Contrast Security an innovative IAST-only player, pitching stand-alone IAST as a silver bullet.
- “Mend.io” is the new name of a company previously called “Whitesource”.

Gartner (2023) on appsec market

- GitLab and GitHub are DevOps platforms, also entering the appsec market, with simple, cheap, well-integrated SAST.
- HCL has bought IBM's "AppScan" portfolio a few years back; used to be leader, but has dropped out.

Open Source Solutions

- In general, in this market there are no open source solutions that can truly compete with the commercial solutions.
- For SAST, there are all kinds of static analysis solutions that do static analysis for quality and also do a little bit of security. (E.g. SonarQube community edition)
  - There is a trend for these solution to evolve towards true appsec including dataflow/controlflow analysis.
- Facebook has released several open-source language-specific SAST taint-analysis solution (Facebook Infer, Mariana Trench).
- For DAST, Zed Attack Proxy is popular, but mainly as a workbench. Its automated attack repertoire is limited. This used to be an OWASP project, but the team left last month and the future of it as an open-source solution is uncertain.
- For SCA, OWASP Dependency Checker is popular, but implements the naive approach to SCA we discussed.

The impact of Generative AI on AppSec

Introduction: LLMs

- Generative AI are AI systems that can generate content, as opposed to just recognizing/classifying things.
- Multiple types of GenAI exist; the type most relevant for AppSec are Large Language Models (LLMs). They are so relevant because they can *reason* and are trained on source code.
- The fundamental thing an LLM does is provide the most likely continuation of a given input prompt, on “token” at a time. On top of this basic capability, many are trained on *dialogue.*
- Important fact on LLMs: *training* and *inference* are completely separate things. ChatGPT seeming to learn from what you write is an illusion built *on top* of the LLM.
- Another important fact: the number of tokens that the LLM can consider is limited and is called the “context size”.

LLM impact on AppSec: an overview

Use SAST technology to look for AI/LLM-specific vulnerabilities in applications.

Attackers using AI to perform smarter, more dangerous attacks.

Opportunities to reduce risk

Growing risk level, making AppSec testing more important, but in a certain sense business as usual for AppSec tools.

Using AI/LLM technology to create better AppSec tooling: fewer FP, auto-remediate, more coverage, etc.

Developers using AI/LLM tech to write code, which may contain AppSec problems.

~~LLM-driven SAST~~

- Give a small test program with seeded vulnerabilities to an LLM, and ask it to find them… you will get **amazing** results.
- So, can we get rid of traditional SAST algorithms and just use an LLM instead?
- **No!**

- The context size prevents it from considering an entire program.
- It doesn't do true analysis; it just predicts the next tokens. This looks like the same thing in trivial examples, but really isn't.
- No understanding of what was (implicitly) checked/what wasn't checked.
- Insofar as it does work, slow and expensive.

AppSec testing: the remaining human work

For each finding, we need to determine:

- Is it a TP or FP? Is there something to fix? (auditing)
- If we need to fix it, how exactly should we do that? (remediation)

These things take up a *lot* of time and cause a "security backlog".

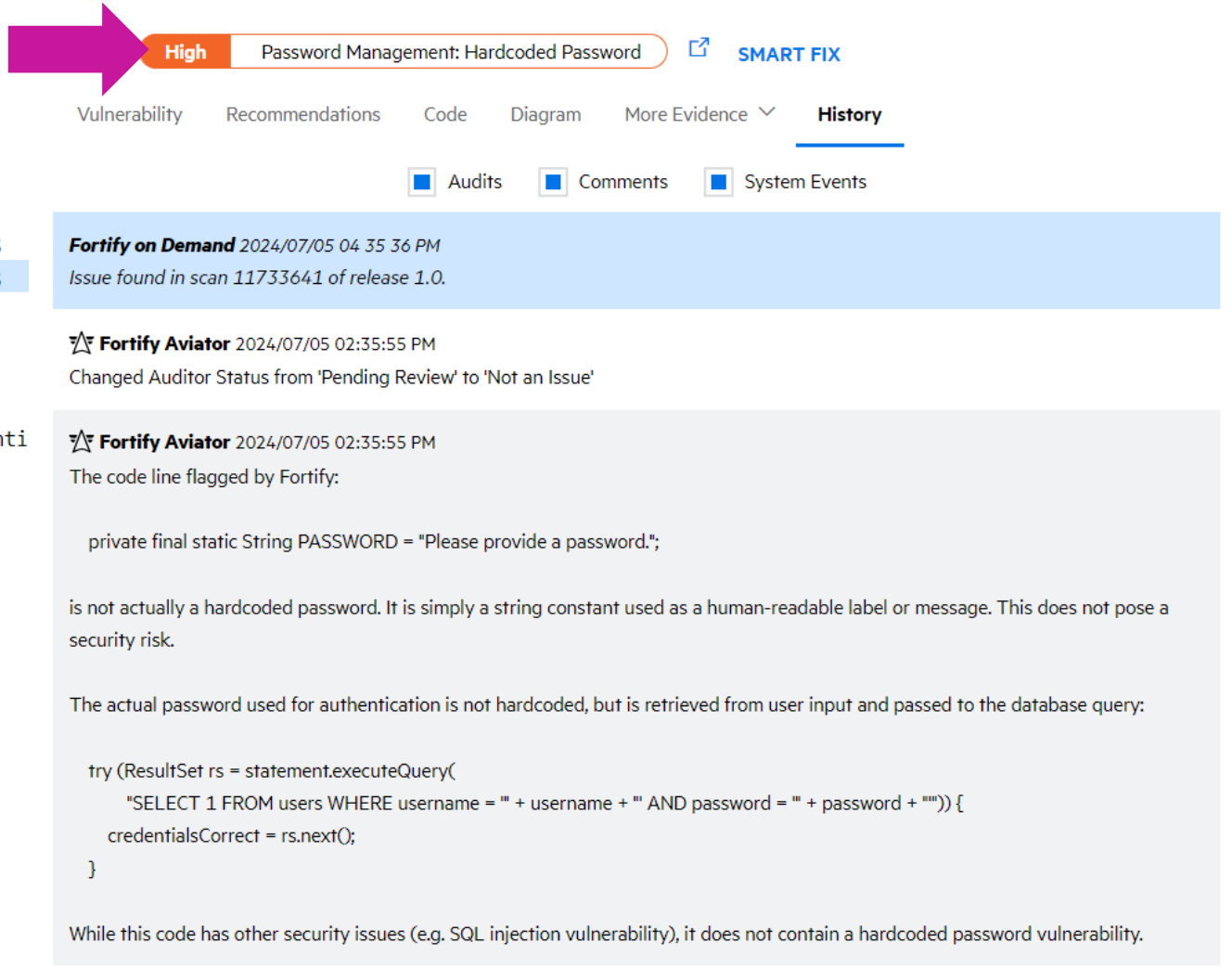
Many AppSec vendors have or are working systems that address these things. For Fortify, it's called "Fortify Aviator".

Example 1: A false positive “Hardcoded Password” issue.

Source code

```
final static String ATTRIB_MESSAGE = "message";
final static String ATTRIB_USERNAME = "username";
final static String USERNAME = "Please provide a username.";
final static String PASSWORD = "Please provide a password.";
final static String WRONG_CREDS = "Wrong credentials.";
final static String SUCCESS = "Logged in successfully.";

tMapping(value = "/auth/login", method = RequestMethod.POST)
ModelAndView loginSubmit(@RequestBody LoginCredentials credenti
                         @RequestParam String redirectUrl,
```

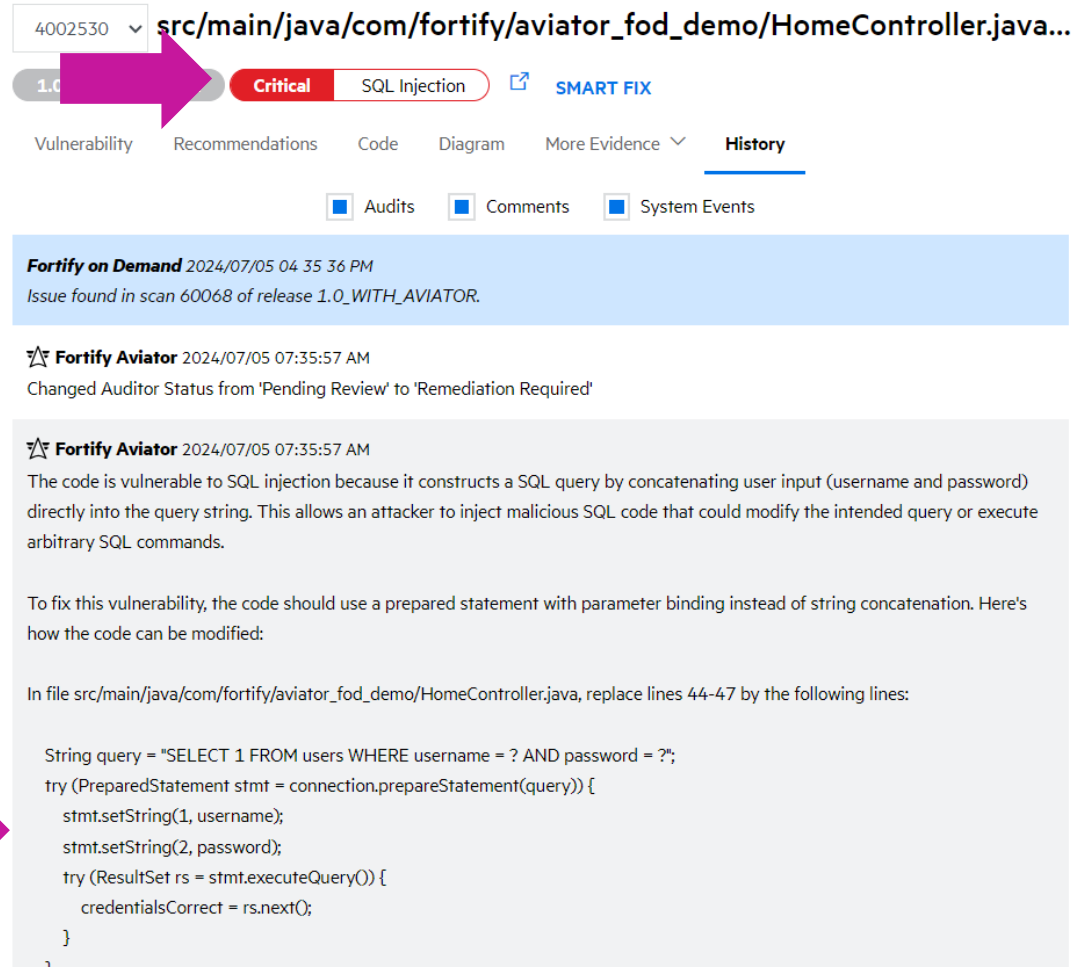


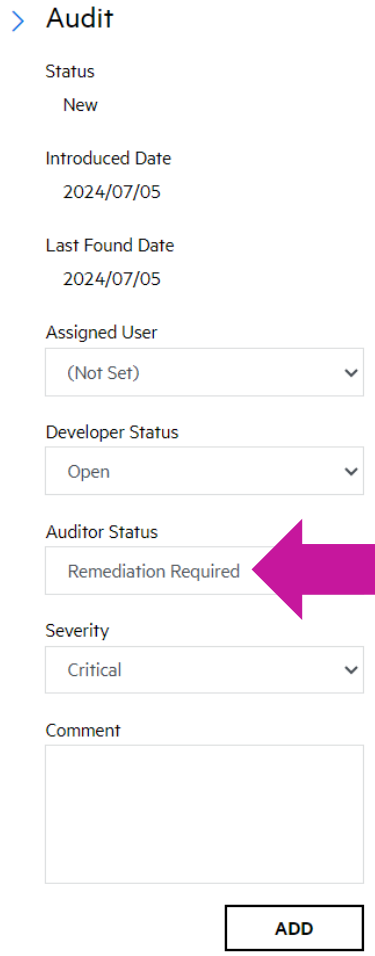
Example 2: A true positive “SQL Injection” issue.

Source code

```
try (Statement statement = connection.createStatement()) {
    try (ResultSet rs = statement.executeQuery(
            "SELECT 1 FROM users WHERE username = '" +
            username + "' AND password = '" + password +
            "'")) {
        credentialsCorrect = rs.next();
    }
}
```

Ready to copy-paste





Example 3: A false positive “mass assignment” issue.

Source code

```
25
26  @RequestMapping(value = "/auth/login", method = RequestMethod.POST)
27  public ModelAndView loginSubmit(@RequestBody LoginCredentials credentials,
28                                  @RequestParam String redirectUrl,
29                                  HttpServletRequest request,
30                                  HttpSession session) throws SQLException {
31      String username = credentials.getUsername();
32      String password = credentials.getPassword();
33      if(username.isEmpty())
34          request.setAttribute(ATTRIB_MESSAGE, USERNAME);
```

```
1   package com.fortify.aviator_fod_demo;
2
3   public class LoginCredentials {
4
5       private String username;
6       private String password;
7
8
9       public String getUsername() {
10          return username;
11      }
12
13      public void setUsername(String username) {
14          this.username = username;
15      }
16
17      public String getPassword() {
18          return password;
19      }
20
21      public void setPassword(String password) {
22          this.password = password;
23      }
24
25  }
26
```

4002524 src/main/java/com/fortify/aviator_fod_demo/HomeController.java...

High Mass Assignment: Insecure Binder Configuration SMART FIX

Vulnerability | Recommendations | Code | Diagram | More Evidence | History

Audits | Comments | System Events

Fortify on Demand *2024/07/05 04 35 36 PM*
Issue found in scan 60068 of release 1.0_WITH_AVIATOR.

Fortify Aviator 2024/07/05 07:35:57 AM
Changed Auditor Status from 'Pending Review' to 'Not an Issue'

Fortify Aviator 2024/07/05 07:35:57 AM
The vulnerability reported by Fortify is a false positive. The LoginCredentials class is a simple data transfer object (DTO) used to bind the username and password fields from the HTTP request body. There is no automatic binding or mass assignment happening here that could lead to insecure data being assigned to the object's fields.

The binding is done explicitly in the HomeController class, where the username and password values are retrieved from the LoginCredentials object using the getUsername() and getPassword() methods. This is a safe and controlled way of handling the data, without any risk of mass assignment vulnerabilities.

Therefore, this issue can be safely ignored as it does not represent an actual security vulnerability in the given code.

Audit

Status
New

Introduced Date
2024/07/05

Last Found Date
2024/07/05

Assigned User
(Not Set)

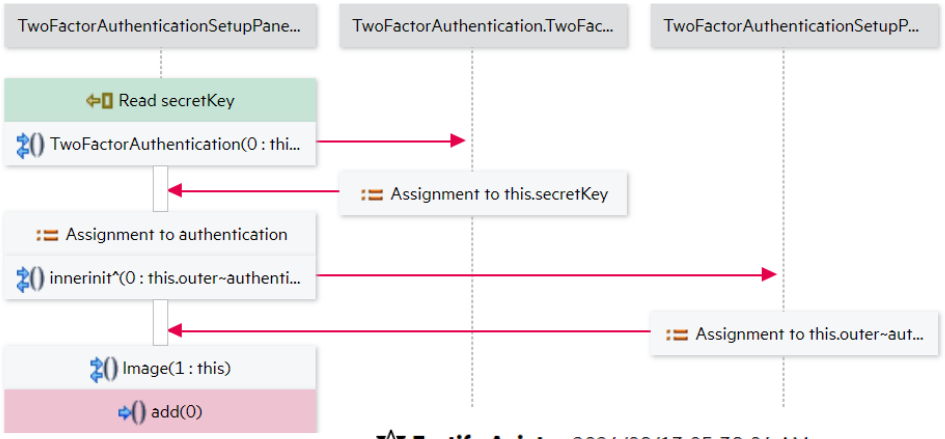
Developer Status
Open

Auditor Status
Not an Issue

Severity
High

Comment

Example 4: A false positive “Privacy Violation” issue



This is code to set-up 2FA by scanning a QR code.

- Fortify SAST is smart enough to detect that a secret key is shown through the QR code image, and flags this.
- Fortify Aviator is smart enough to understand that in this context, that’s fine.

Fortify Aviator 2024/08/13 05:38:04 AM

Changed Auditor Status from 'Pending Review' to 'Not an Issue'

Fortify Aviator 2024/08/13 05:38:04 AM

The code in question is creating an image resource that displays a QR code for two-factor authentication setup. The QR code contains the user's secret key, which is necessary for setting up two-factor authentication. This is not a privacy violation because:

1. The secret key is not being logged or written to an external location. It is only being used to generate the QR code image, which is then displayed to the user.

2. The QR code image is only accessible to the authenticated user who is setting up two-factor authentication. It is not being exposed to any unauthorized parties.

3. The secret key is a necessary part of the two-factor authentication setup process and is not considered sensitive information in this context.

Therefore, this code does not represent a privacy violation or an exploitable vulnerability. The use of the secret key to generate the QR code is a legitimate and expected part of the two-factor authentication setup process.

Last Found Date

2024/03/26

Assigned User

(Not Set)

Developer Status

Open

Auditor Status

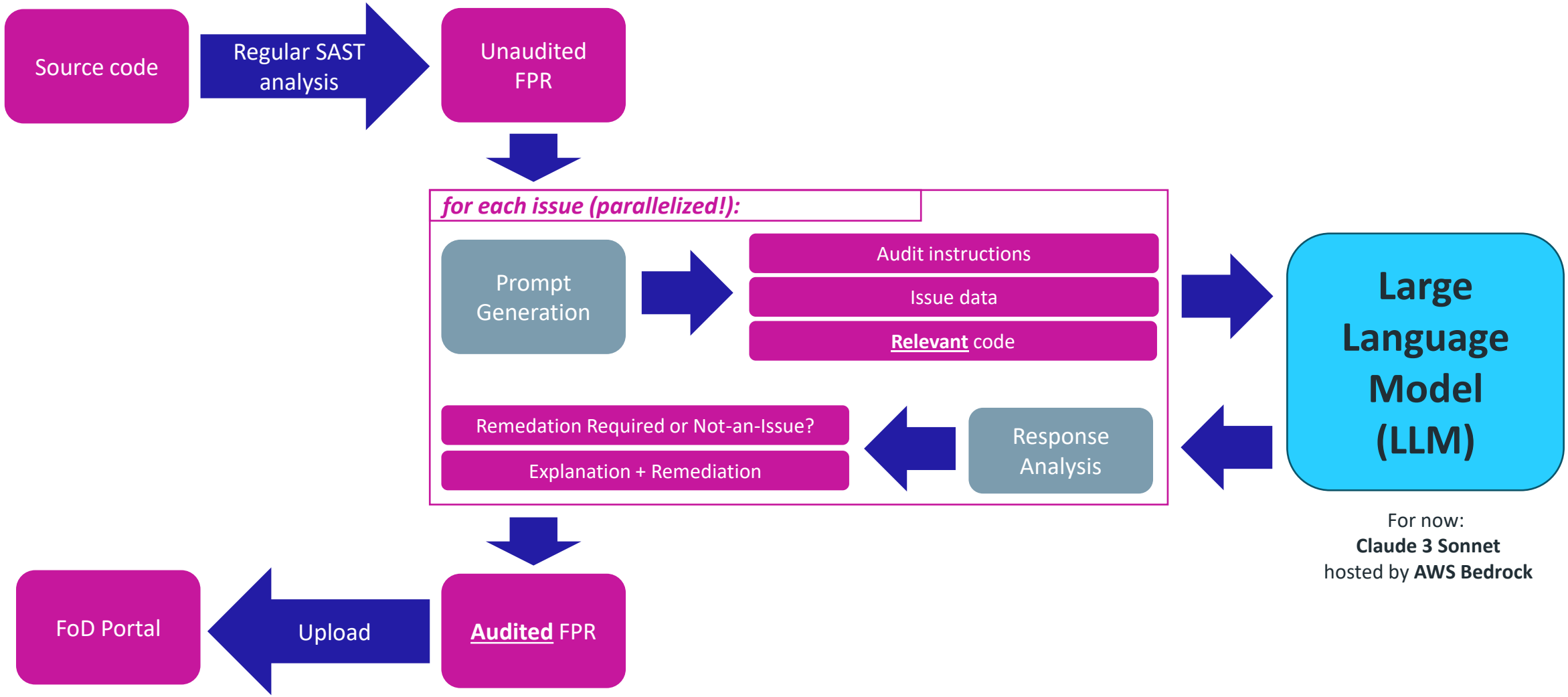
Not an Issue

Severity

Critical

Comment

How Fortify Aviator Works



Comparing Fortify Aviator and Audit Assistant

	Audit Assistant	Fortify Aviator
Launch date	2015	2024 (26 July, to be precise)
Functionality	TP/FP classification	TP/FP classification Justification of classification (English) Remediation advice (English/code)
Underlying ML technology	Random forest	LLM
Training	Supervised learning using audited FPRs.	LLM's base training + in-context learning
Deployments models	Fortify on Demand (Public Cloud) Fortify Hosted (Private Cloud) Off-cloud with cloud Audit Assistant (AA) Off-cloud with AA on-premise	Fortify on Demand (all other models are roadmap)
Scope	14 languages, most categories	Java, 22 most prevalent categories (other languages/categories are roadmap)
Access to Source Code	No, metadata and metrics only.	Yes, but only those files that are relevant to a vulnerability.
Reliability (qualitative notes)	Reliability is inherently limited because it doesn't have access to source code. Also depends on prediction policy, customer-specific training, etc.	Very high. See previous discussion.
Overall product future	Maintenance of both the software and the model.	Aggressive development in the next 12-18 months.

FAQ – consistency, reliability, customizability

Question	Answer
LLMs are not-deterministic. How do you prevent inconsistent audit results?	1. LLM parameters (temp, top-P, top-K) are tuned to near-determinism. 2. An already audited issue is never reaudited.
How reliable is this? Can you give a percentage?	It's impossible to answer the reliability question with a percentage for many methodological reasons. Qualitatively: 1. It occasionally makes mistakes. It's not as good (yet) as the best human expert with sufficient time. 2. Practically, we believe Fortify Aviator will often outperform most human teams. It outperforms Fortify Audit Assistant and most non-expert human auditors (developers and/or more junior security auditors)
Can we train this on customer-specific data to optimize results?	Fortify Aviator doesn't get any task-specific training. It relies on the LLM base training, with in-context learning. That means it cannot be trained on customer data either. As a roadmap item, we intend to make LLM prompt construction allow for customer-specific additional instructions.

FAQ – auto-remediation, data confidentiality

Question	Answer
Fortify Aviator provides remediation advice in comments. Why not have full auto-remediation and automatically apply the advice?	Technically, this is a relatively easy extension. We need the LLM to produce exact replacement instructions, which we store in the audited results for an IDE plugin to pick up. To make this a great developer UX, reliability and accuracy will need to be even higher than they are now. We expect this to happen in 1-2 years as models improve. Even then, auto-remediation will not be practicable in all cases.
Does Fortify Aviator send source code to the LLM?	Yes, this is at the core of how it works. There are a few important limitations: 1. We only use LLMs in the same FoD data center: data residency is respected. 2. We only send relevant parts of the source code, not the entire code base.
Is the confidentiality of the source code guaranteed?	Yes. First, we only use the LLM in inference, during which it does not automatically learn (a fact that's not always correctly perceived), and does not store the data. To mitigate the risk that the LLM provider would nevertheless store or use this data: 1. There's an EULA in place between us and the LLM provider that guarantees confidentiality. 2. The separation between the hosting party (AWS) and LLM developer (Anthropic) creates another layer of security.

Using Fortify SAST to detect AI/LLM risk in applications

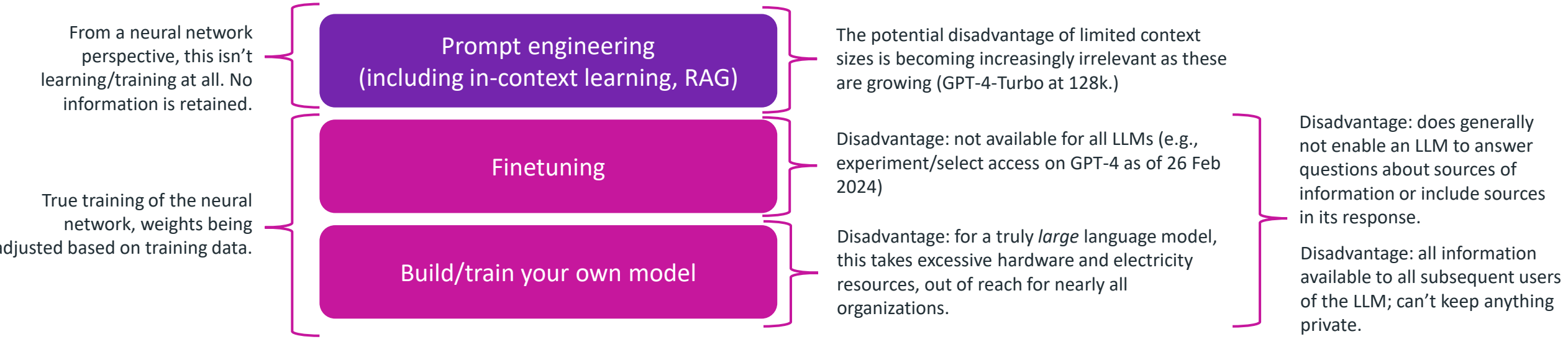
- **General notes:**

- We're trying to cover the specific risks related to the current wave of LLM-powered AI.
  (as opposed to machine learning in the general sense)
- As an AppSec tool, Fortify's focus is to assess the security of applications using LLMs.
  (as opposed to security issues in the LLM itself)

- **Question to discuss:**

- What LLM development strategies to focus on?
- Which vulnerability categories exist and can be covered by SAST?
- Which languages and libraries to support?

Development strategies for LLM applications



There are multiple ways to build an LLM-powered application.
Supporting the "prompt engineering" model is the priority for Fortify, given its popularity.

OWASP Top-10 for LLM Applications

LLM01: Prompt Injection

This manipulates a large language model (LLM) through crafty inputs, causing unintended actions by the LLM. Direct injections overwrite system prompts, while indirect ones manipulate inputs from external sources.

LLM02: Insecure Output Handling

This vulnerability occurs when an LLM output is accepted without scrutiny, exposing backend systems. Misuse may lead to severe consequences like XSS, CSRF, SSRF, privilege escalation, or remote code execution.

LLM03: Training Data Poisoning

This occurs when LLM training data is tampered, introducing vulnerabilities or biases that compromise security, effectiveness, or ethical behavior. Sources include Common Crawl, WebText, OpenWebText, & books.

LLM04: Model Denial of Service

Attackers cause resource-heavy operations on LLMs, leading to service degradation or high costs. The vulnerability is magnified due to the resource-intensive nature of LLMs and unpredictability of user inputs.

LLM05: Supply Chain Vulnerabilities

LLM application lifecycle can be compromised by vulnerable components or services, leading to security attacks. Using third-party datasets, pre- trained models, and plugins can add vulnerabilities.

LLM06: Sensitive Information Disclosure

LLMs may inadvertently reveal confidential data in its responses, leading to unauthorized data access, privacy violations, and security breaches. It's crucial to implement data sanitization and strict user policies to mitigate this.

LLM07: Insecure Plugin Design

LLM plugins can have insecure inputs and insufficient access control. This lack of application control makes them easier to exploit and can result in consequences like remote code execution.

LLM08: Excessive Agency

LLM-based systems may undertake actions leading to unintended consequences. The issue arises from excessive functionality, permissions, or autonomy granted to the LLM-based systems.

LLM09: Overreliance

Systems or people overly depending on LLMs without oversight may face misinformation, miscommunication, legal issues, and security vulnerabilities due to incorrect or inappropriate content generated by LLMs.

LLM10: Model Theft

This involves unauthorized access, copying, or exfiltration of proprietary LLM models. The impact includes economic losses, compromised competitive advantage, and potential access to sensitive information.

In scope of Fortify SAST support:

- LLM02, LLM07 and LLM08 are all about recognizing LLM output as a taint source. This has been modeled in rules for select libraries.
- For LLM02, a specific sink category has been added to distinguish LLM-reflected XSS from other types of XSS.
- LLM01 is not yet supported. Roadmap: support for injection into the *system* prompt. Injection into the user prompt is out of scope since no standard cleansing method exists.

Out of scope for SAST *for now*:

- LLM03, LLM10 (not irrelevant for our focus use case of prompt engineering)
- LLM06 (currently unclear how we could support this in a way that doesn't generate massive FP)
- LLM04 (generally an operational issue; potentially, we could flag unprotected access to a backend LLM),

Out of scope for SAST:

- LLM05 (covered by composition analysis)
- LLM09 (human issue)

LLM language and library focus

- Support for the LLM categories needs to implemented through Fortify rules, which is means it's separate for each library that we want to support.

- Fortify is currently focusing on Python as the most important language in the AI/LLM space.
- Currently supported libraries: OpenAI, AWS SageMaker, LangChain, Tensorflow, Claude
- Roadmap: Google Vertex, HuggingFace Transformers

- We're very open to customer feedback in this area to help shape our roadmap!


MICRO FOCUS
FORTIFY
Questions?