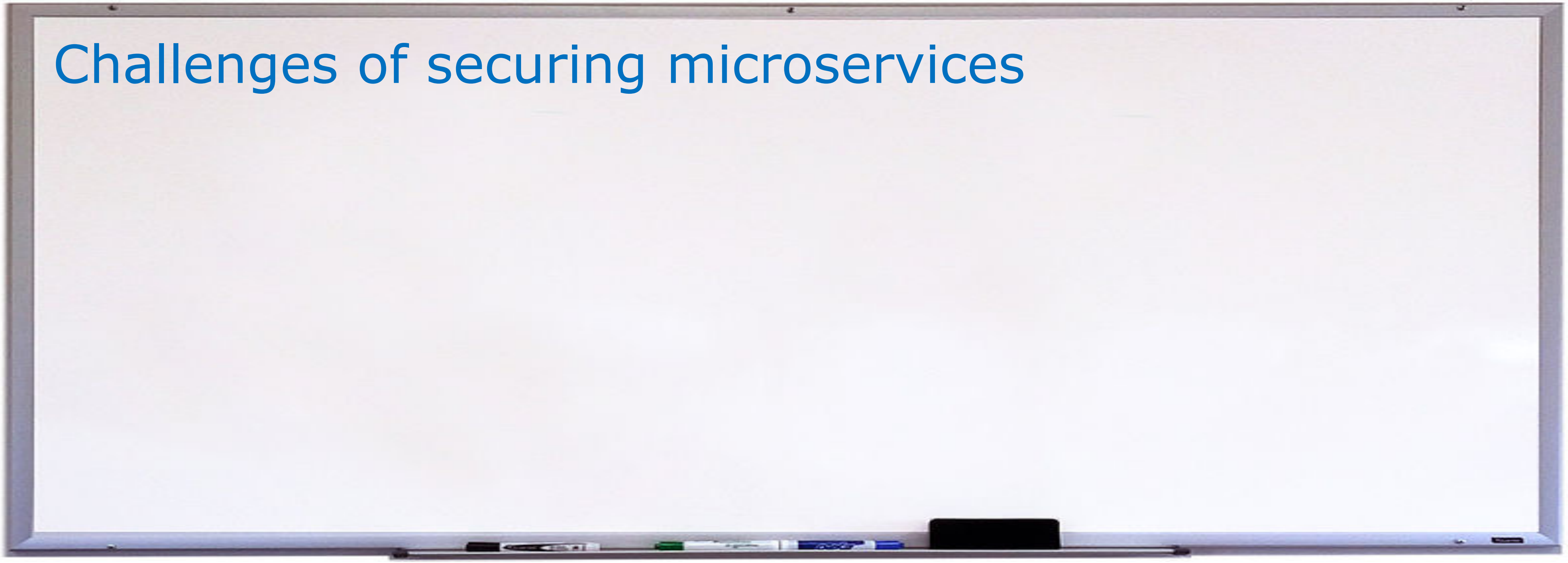
Security and microservices

Antonio Brogi

Department of Computer Science
University of Pisa



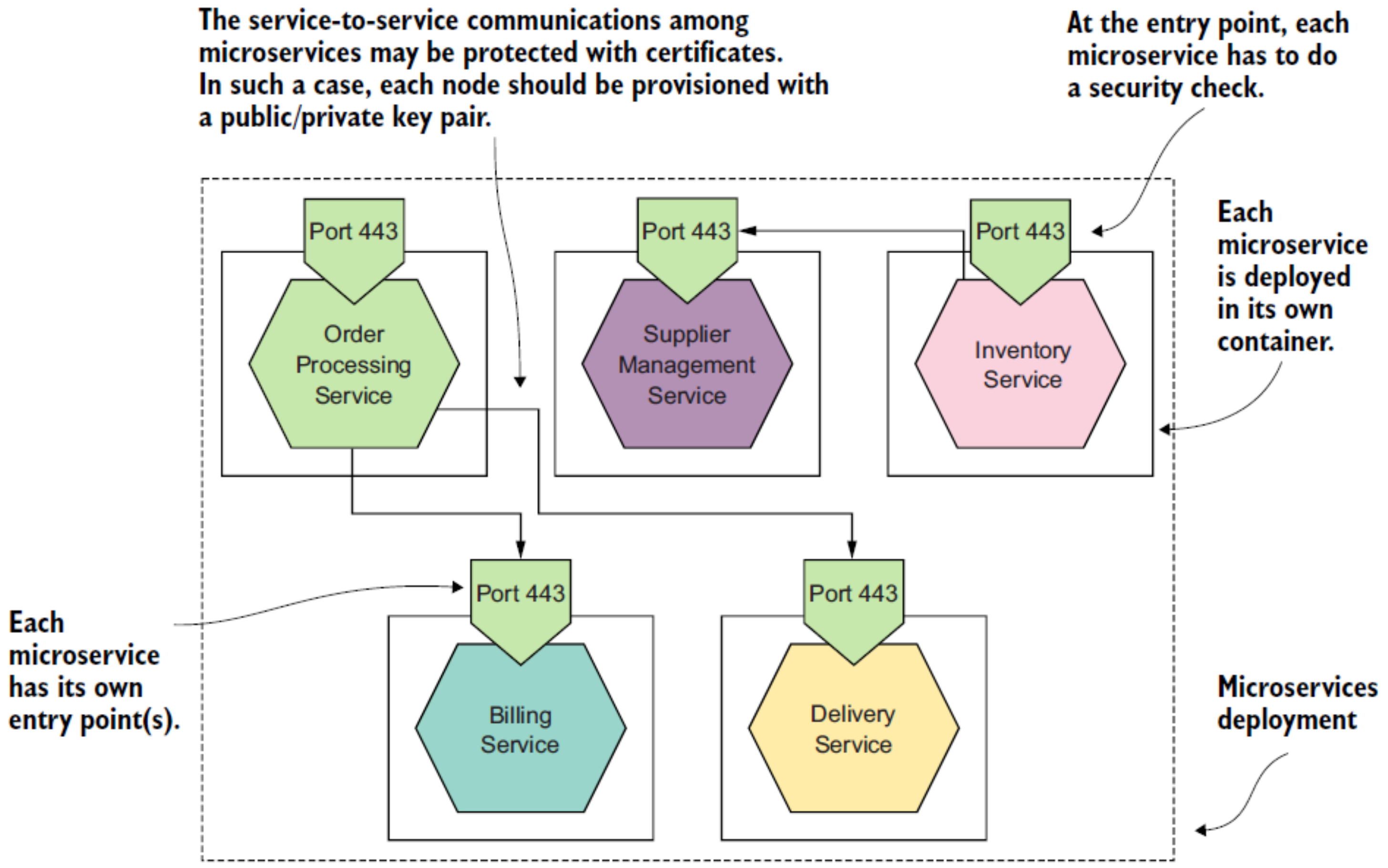
Challenges of securing microservices

The broader the attack surface, the higher the risk

Monolith: inter-component communications within single process

Microservices:

- inter-service communication via remote calls
- large number of entry points, **attack surface broadens**
- security of app = strength of weakest link


The service-to-service communications among microservices may be protected with certificates. In such a case, each node should be provisioned with a public/private key pair.
At the entry point, each microservice has to do a security check.
Each microservice is deployed in its own container.
Port 443
Order Processing Service
Supplier Management Service
Inventory Service
Billing Service
Delivery Service
Each microservice has its own entry point(s).
Microservices deployment

Distributed security screening affects performance

Monolith: security screening done once, request dispatched to corresponding component

Each microservice has to carry out independent security screening
- may need to connect to a remote security token service
- repeated, distributed security checks affect **performance**

Work around: trust-the-network (and skip security checks at each microservice)

Industry trend: ~~trust-the-network~~ zero-trust networking principles

Still, overall performance must be taken into consideration

Bootstrapping trust among microservices needs automation

Service-to-service communication must take place on protected channels

Suppose that you use certificates:

- Each microservice must be provisioned with a certificate (and corresponding private key) to authenticate itself to another microservice during service-to-service interactions
- Recipient microservice must know how to validate the certificate associated with calling microservice
- Need a way to **bootstrap trust** between microservices
- (Need also to revoke and rotate certificates)

→To manage large-scale deployments of hundreds of microservices, **automation** is needed

Tracing requests spanning multiple microservices is challenging

A *log* is the recording of an event of a service

Logs can be aggregated to produce *metrics* that reflect the system state (e.g. average invalid access requests per hour) and that may trigger alerts

Traces help you track a request from the point where it enters the system to the point where it leaves the system

Unlike in monolithic applications, a request to a microservices deployment may span multiple microservices
→ **correlating requests among microservices is challenging**

Tools
- e.g. Prometheus and Grafana to monitor incoming requests
- e.g. Jaeger and Zipkin for distributed tracing

Containers complicate credentials/policies handling

Containers are **immutable servers**, they don’t change state after spin up

great to simplify deployment and to achieve horizontal-scalability

But for each service we need to maintain a dynamic list of allowed clients and a dynamic set of access-control policies

e.g. get updated policies from some policy admin endpoint (with a push or pull model)

Each service must also maintain its own credentials, which need to be rotated periodically

e.g. keep credentials in container filesystem and inject them at boot time

Distribution makes sharing user context harder

User context has to be passed explicitly from one microservice to another

Challenge: Build trust between microservices so that receiving microservice accepts user context passed from calling microservice

Popular solution: Using JSON Web Token (JWT) to share user context among microservices

- Idea: messages carrying user attributes in a cryptographically safe way

Security responsibilities distributed among different teams

Polyglot architecture: Different squads can use different technology stacks

Different teams can use different security best practices, and different tools for static code analysis and dynamic testing

Security responsibilities distributed across different teams

Organizations often adopt hybrid approach, with centralized security team and security experts in the teams

Summary

Challenges of securing microservices

- The broader the attack surface, the higher the risk
- Distributed security screening affects performance
- Bootstrapping trust among microservices needs automation
- Tracing requests spanning multiple microservices is challenging
- Containers complicate credentials/policies handling
- Distribution makes sharing user context harder
- Security responsibilities distributed among different teams

Challenges of securing microservices

Smells and refactorings for microservices security

Multivocal review of white and grey literature

Review of white and grye literature (58 references) to answer two research questions

- **#1** Which are the most recognised smells indicating possible security violations in microservice-based applications?
- **#2** How to refactor a microservice-based application to mitigate the effect of a security smell?

Result: Taxonomy with

10 refactorings associated with

10 smells potentially affecting

3 security properties

Properties

Taken by ISO/IEC 25010 software quality standard

Confidentiality

degree to which a product or system ensures that data are accessible only to those authorized to have access

Integrity

degree to which a system, product, or component prevents unauthorized modification of computer programs or data

Authenticity

degree to which the identity of a subject or resource can be proved to be the one claimed

Confidentiality ⇠------- **Insufficient access control** ⟶ **Use Oauth 2.0**

Microservice-based application does not enact access control in some microservices
→ Potential “confused deputy problem” with attacker getting data it shouldn’t be able to get
→ Potential violation of confidentiality of data (and business functions)

Client permissions need to be verified at *request time*

Client identity should be verified without introducing extra latency and contention with frequent calls to a centralized service

Most suggested refactoring: employ OAuth 2.0
- token-based security framework for delegated access control
- resource owner can grant client access to a resource on its behalf
- access granted for limited time and with limited scope

Confidentiality ← Publicly accessible microservices → Add API gateway

Some microservices directly accessible by external clients
→ Each such microservice must check authentication and authorization for each request
  → Increased exposure of credentials → Increased likelihood of confidentiality violations
  → Higher cost of application maintenance

Most cited refactoring: Add API gateway
- microservices not directly accessible by external clients
- API gateway can enforce authentication, authorization, throttling, message content validation
- microservices can stay behind a firewall

$$\begin{tikzcd}
\text{Confidentiality} & & \\
& \text{Unnecessary privileges to microservices} \arrow[ul, dashed] \arrow[dl, dashed] \arrow[r] & \text{Follow the least privilege principle} \\
\text{Integrity} & &
\end{tikzcd}$$

Microservices are granted unnecessary access levels, permissions, or functionalities that are not needed to deliver their business functions

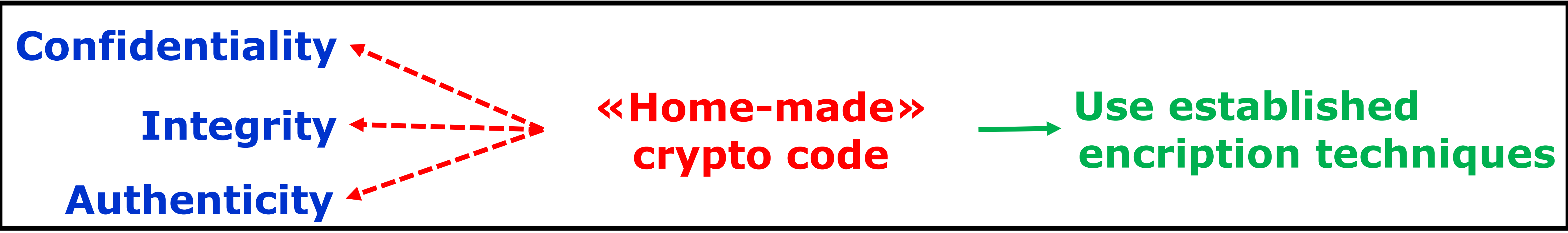
e.g. service can read/write data in db or messages in queue even if db or queue not needed by the service to deliver its business function

Resources are unnecessarily exposed

$\rightarrow$ Increased attack surface against confidentiality and integrity properties

Most cited refactoring: Follow Least privilege principle

> ***Least Privilege Principle:*** *Allow running code only the permissions needed to complete the required tasks and no more.*

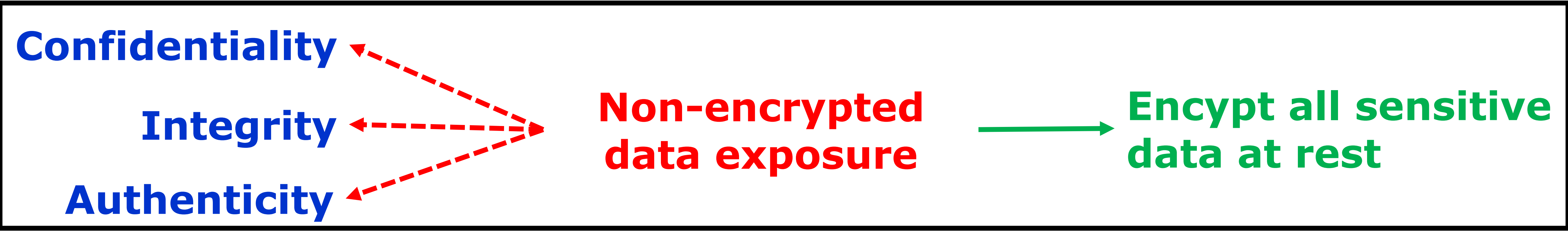


Use of “home-made” crypto code can cause confidentiality, integrity and authenticity issues

Can get even worse than no encryption (false sense of security)

Solution

- Exploit encryption libraries heavily tested by the community and regularly reviewed/patched
- Avoid experimental encryption algorithms (not yet known vulnerabilities)


Confidentiality
Integrity
Authenticity
Non-encrypted data exposure
Encypt all sensitive data at rest

Microservice-based application accidentally exposes sensitive data

e.g. data was stored without any encryption or employed protection has vulnerabilities/flaws

→ intruders can access/modify data, including credentials

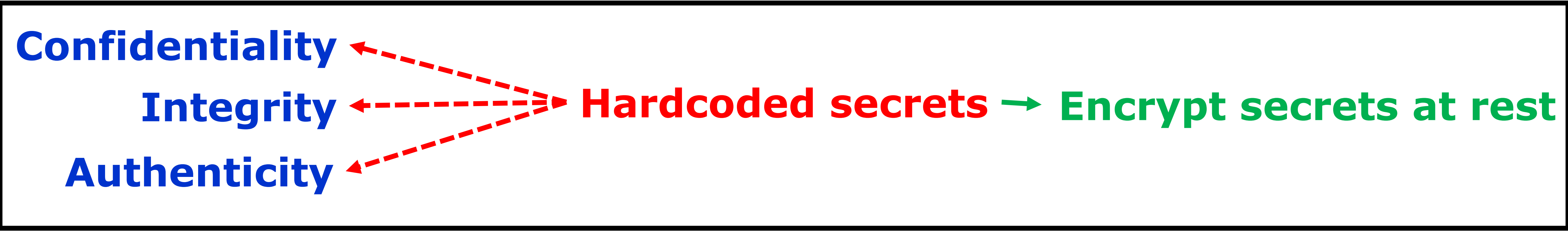
→ confidentiality and integrity can be violated

Encrypt all sensitive data at rest

All sensitive data should stay always encrypted, and decrypted only when needed

- most DBMSs support automatic encryption
- OSs support disk-level encryption
- application-level encryption is another option
- cached data should be encrypted too

Encryption is resource consuming → indetify critical data and encrypt only "as needed"



Hardcoded secrets (e.g. API keys, client secrets, credentials) contained

- in microservice source code
- in application deployment scripts (e.g. environment variables passing secrets in a Docker Compose spec)

Never store secrets in environment variables

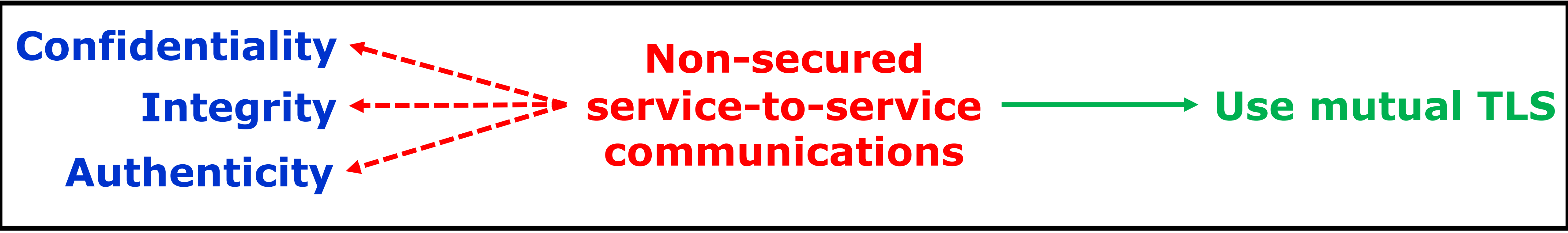
→ they could be accidentally exposed (e.g. exception handlers may send info to logging platform)

→ confidentiality and integrity of secrets may get violated

Solution: Encrypt secrets at rest

Best practices:

- Do not store credentials alongside applications or in source code repositories
- Do not employ environment variables to pass secrets



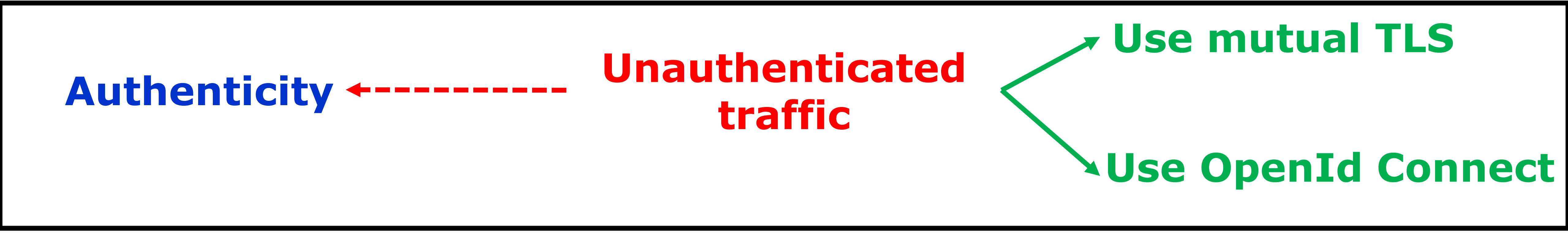
Microservices interacting without enacting a secure communication channel

$\rightarrow$ data is exposed to man-in-the-middle, eavesdropping, and tampering attacks

$\rightarrow$ potential confidentiality, integrity and authenticity issues (intruders can access alter data in transit)

Solutions

- Use mutual Transport Layer Security: widely accepted solution, encrypting data in transit and ensuring its integrity and confidentiality
- Mutual TLS also allows a microservice to legitimately identify the microservice it is talking to (mutual authentication)



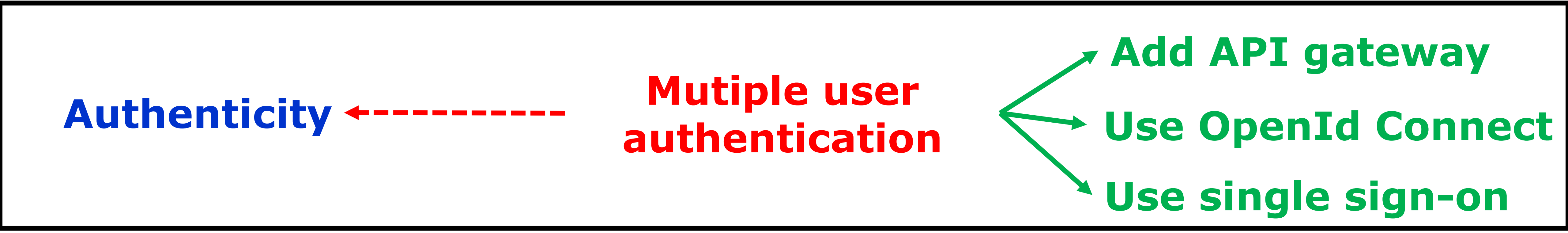
Unauthenticated requests (external or microservice-to-microservice)

Crucial that microservices can authenticate one another (especially when user context is passed)

If traffic not authenticated then microservices are exposed to security attacks like tampering with data, denial of service, or elevation of privileges

Refactorings:

- Use Mutual TLS
- Use OpenID Connect
  - -- uses JWT containing authenticated user info
  - -- microservices verify user identity with authorization server



Application provides multiple access points to handle user authentication
→ Each access point constitutes a potential attack vector for intruder to authenticate as end user
→ Higher cost of application maintenance

Use a Single Sign-On approach
- single entry point to handle user authentication and to enforce security for all user requests
- facilitates log storage and auditing

Single sign-on can be achieved by
- adding API gateway (if not there), and
- employing OpenID Connect (to share user contexts)

$$\begin{tikzcd}\text{Authenticity} & \text{Centralised authorization} \arrow[l, dashed] \arrow[r] & \text{Use decentralised authorization}\end{tikzcd}$$

Authorization can be enforced at the edge of the application (API gateway) and/or by each microservice of the application.

If application only handles authorization at the edge

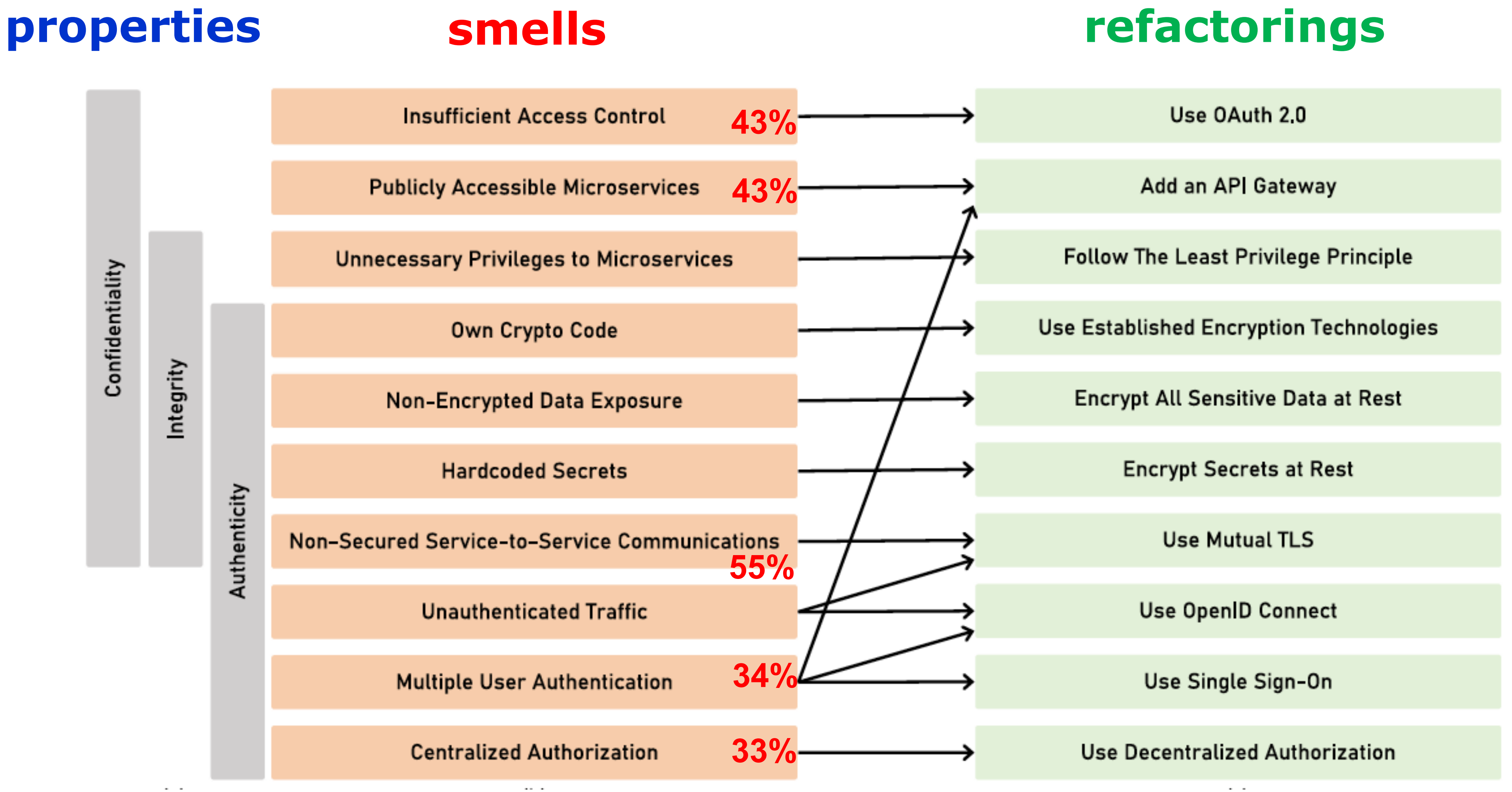
→ "central" authorization point becomes bottleneck → reduced performance and efficiency

Possibility of (another form of) "confused deputy problem": microservices trust the gateway based on its mere identity -> potential violation of authenticity

Solution: enacting a decentralized authorization approach

- transmit an access token (e.g. JWT) together with each request to a microservice
- access to microservice is granted to caller only if a known and correct token is passed

Summary


properties
smells
refactorings
Confidentiality
Integrity
Authenticity
Insufficient Access Control
43%
Publicly Accessible Microservices
43%
Unnecessary Privileges to Microservices
Own Crypto Code
Non-Encrypted Data Exposure
Hardcoded Secrets
Non-Secured Service-to-Service Communications
55%
Unauthenticated Traffic
Multiple User Authentication
34%
Centralized Authorization
33%
Use OAuth 2.0
Add an API Gateway
Follow The Least Privilege Principle
Use Established Encryption Technologies
Encrypt All Sensitive Data at Rest
Encrypt Secrets at Rest
Use Mutual TLS
Use OpenID Connect
Use Single Sign-On
Use Decentralized Authorization

Challenges of securing microservices

Smells and refactorings for microservices security

To refactor or not to refactor?

Solving a smell may spoil other properties!

To refactor or not to refactor – that is the question

security properties

design principles

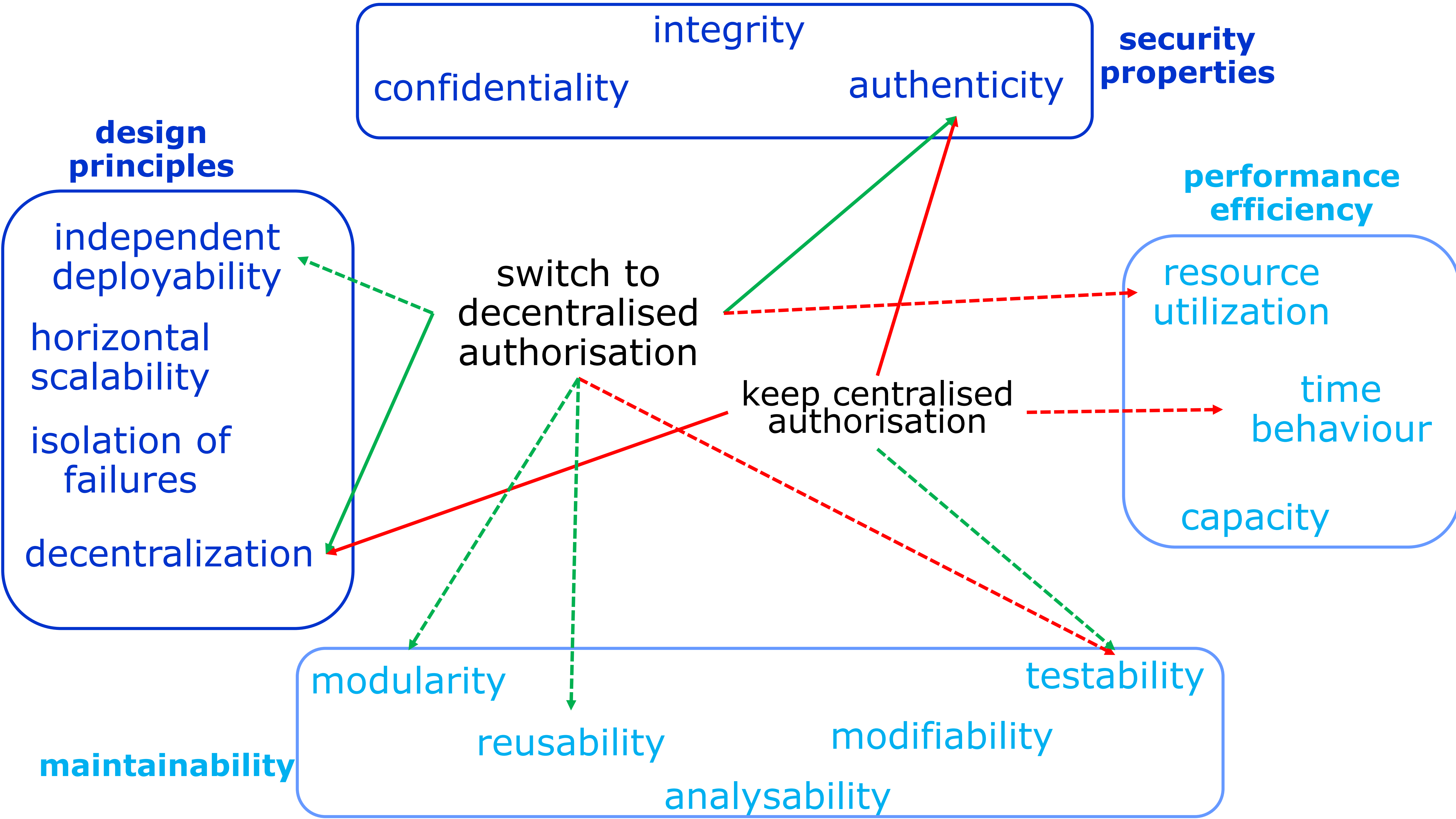
switch to decentralised authorisation

keep centralised authorisation

performance efficiency

maintainability

Modelling softgoal interdependencies as **graphs** can help visualization and (automated) trade-off analysis


integrity
confidentiality
authenticity
security properties
design principles
independent deployability
horizontal scalability
isolation of failures
decentralization
switch to decentralised authorisation
keep centralised authorisation
performance efficiency
resource utilization
time behaviour
capacity
modularity
reusability
testability
modifiability
analysability
maintainability

Challenges of securing microservices

Smells and refactorings for microservices security

To refactor or not to refactor?