OFFZONE 2022

Application Security Design Antipatterns

Aleksei Meshcheriakov

Security Engineer

Moscow, August 25, 2022

What is an antipattern?

- Commonly-used solution that has more bad consequence than good ones
- Another effective solution exists

What are the dangers of implicit use of antipatterns?

- Vulnerability susceptibility
- Difficult to Retrofit

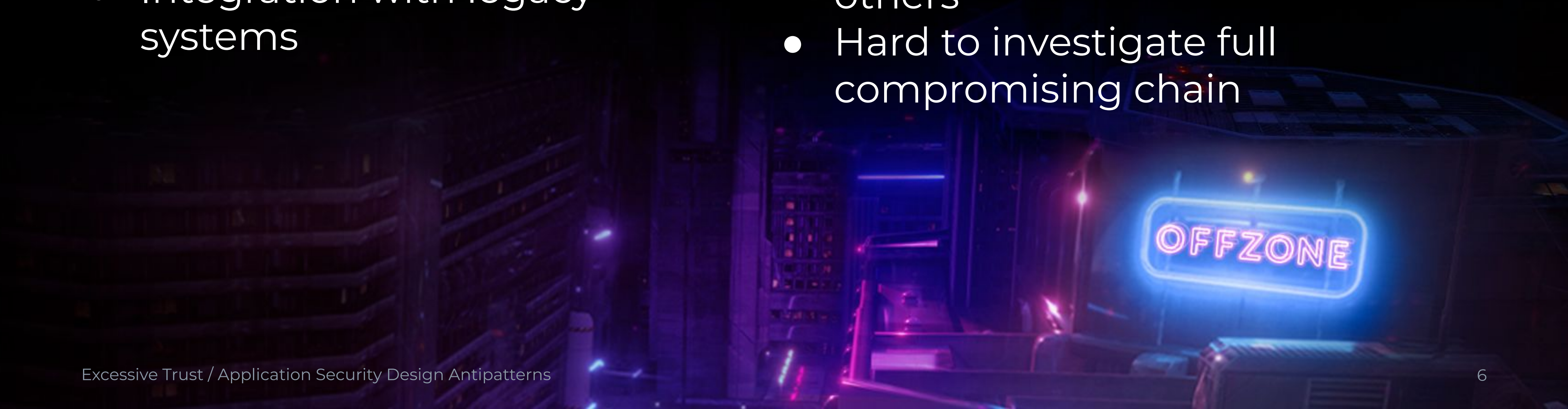
OFF ONE 2022

Antipattern #1: Excessive Trust

Excessive Trust

Trust is based on a weak factor

Excessive Trust



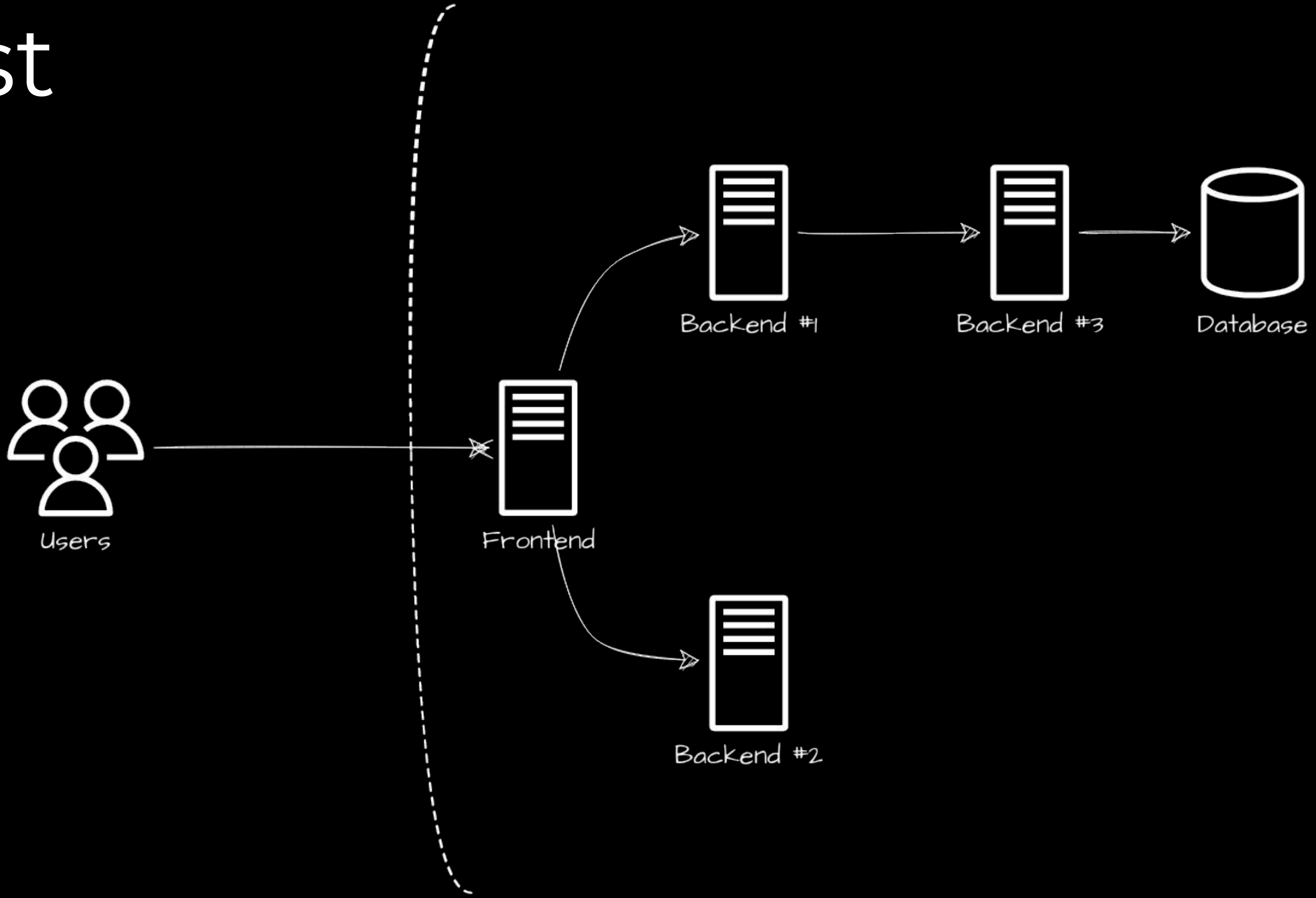
Reasons

- Easy to implement at small scale
- Integration with legacy systems

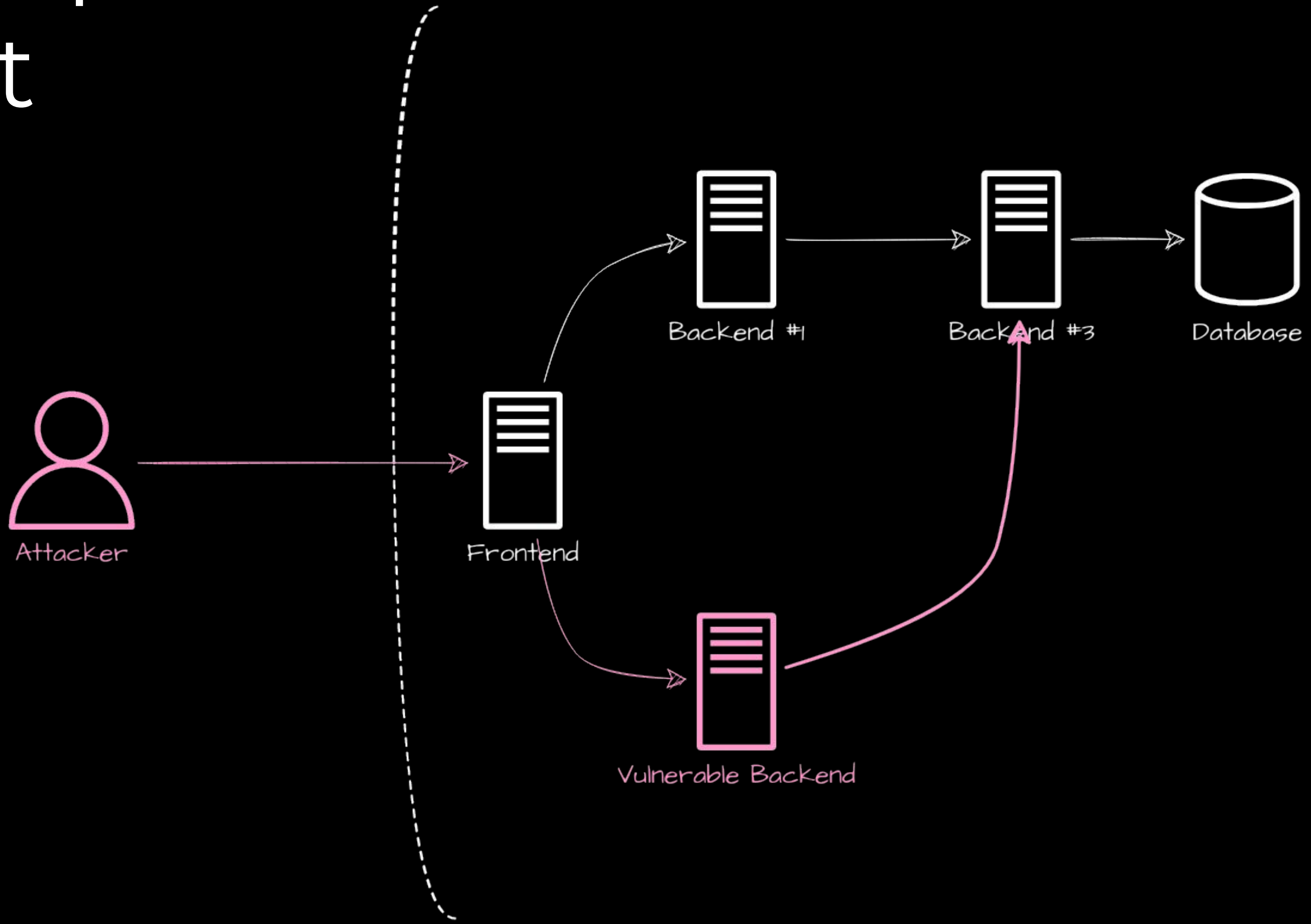
Consequences

- Compromising one component gives access to others
- Hard to investigate full compromising chain

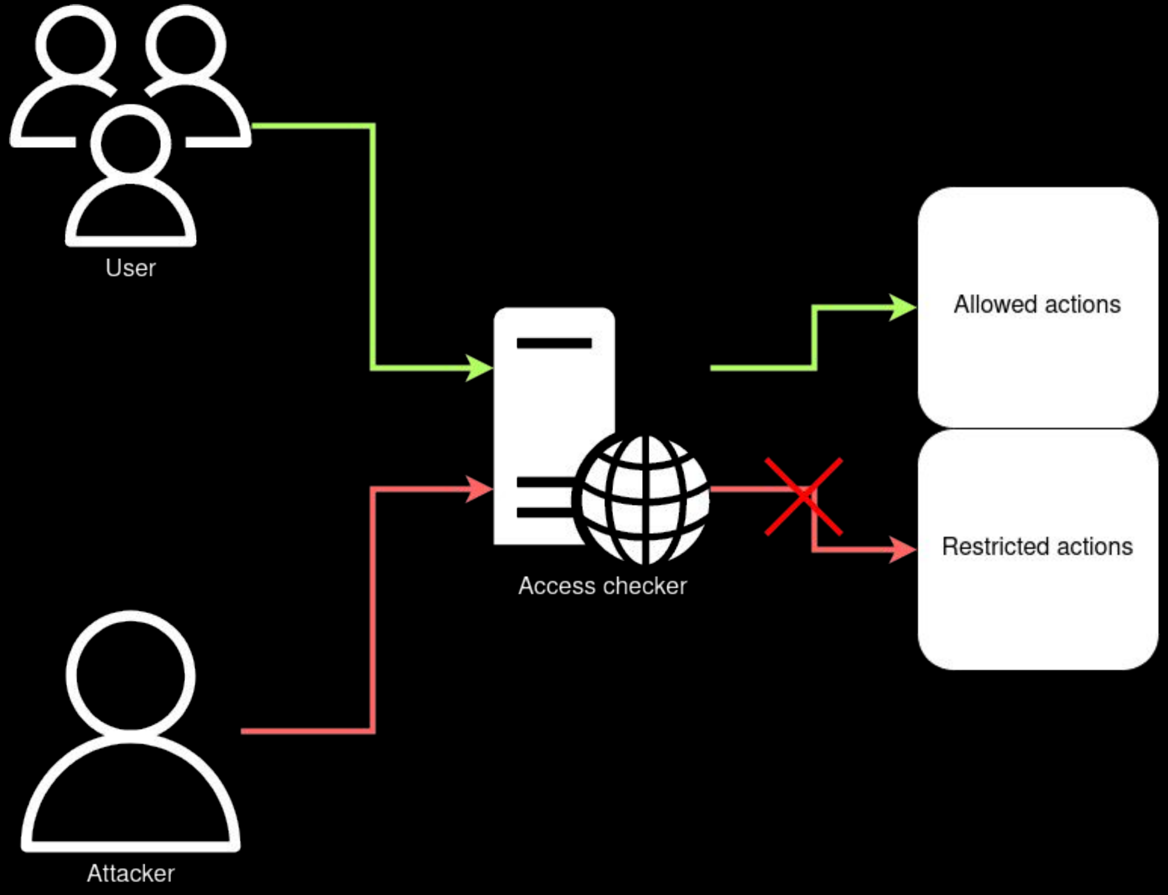
Example #1: Internal Network Trust



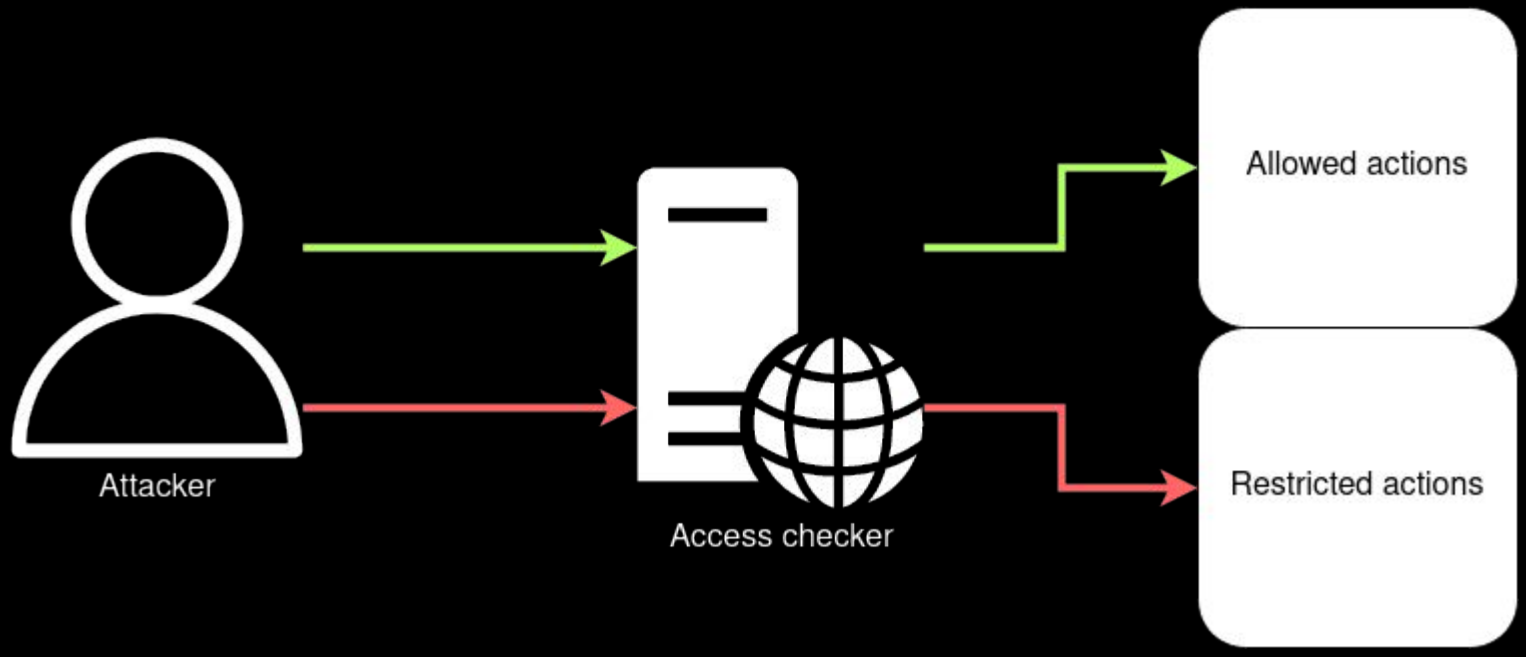
Example #1: Internal Network Trust



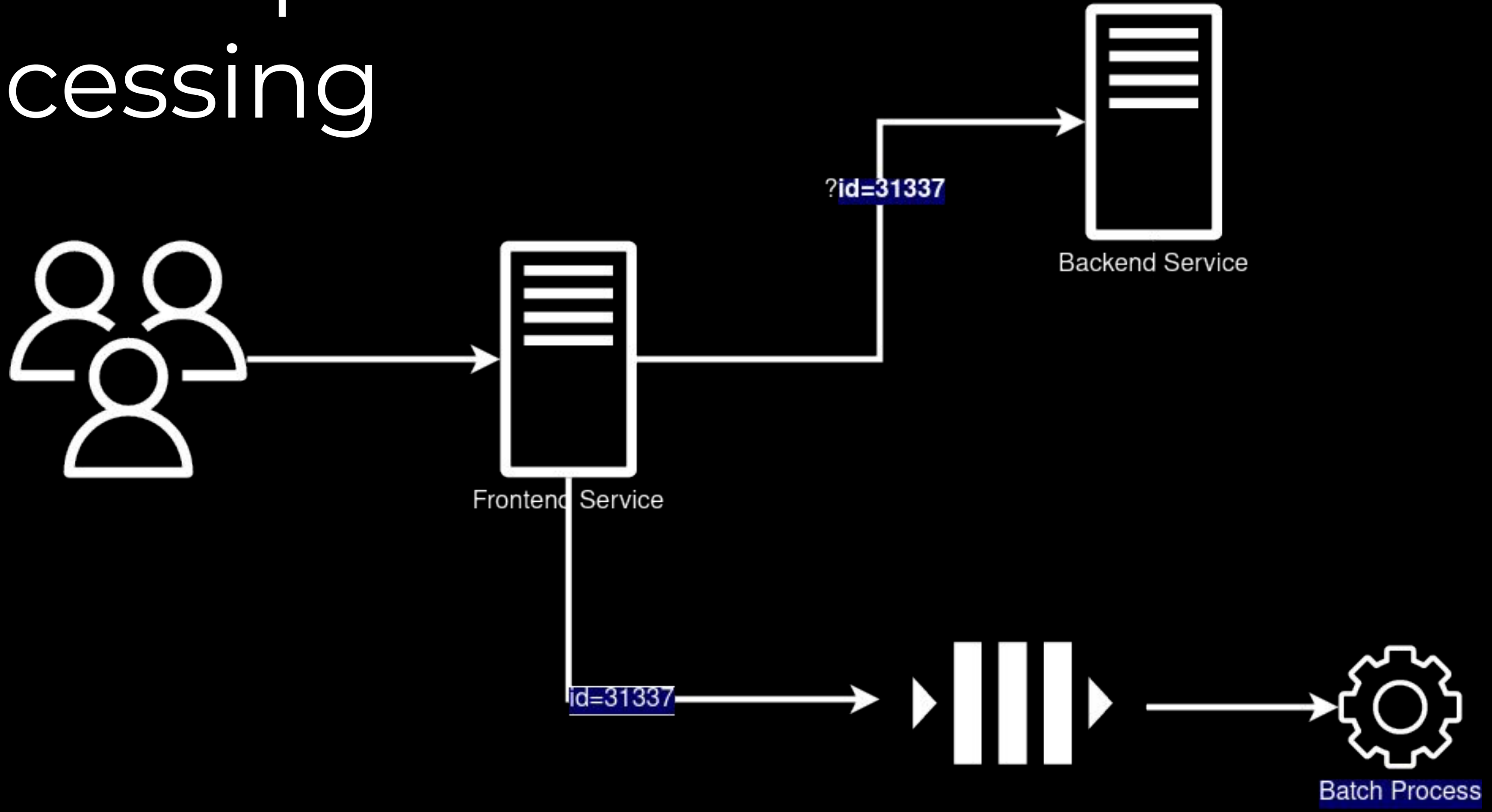
Example #2: Insufficient Access Control



Example #2: Insufficient Access Control



Example #3: UserID in interservice requests and batch processing



How to detect

- For each connection in the data flow diagram:
  How does one component authenticate another?

How to avoid

- Zero Trust Principle

OFF ONE 2022

Antipattern #2: Unlimited Blast Radius

Unlimited blast radius

Lack of strict boundaries
between components

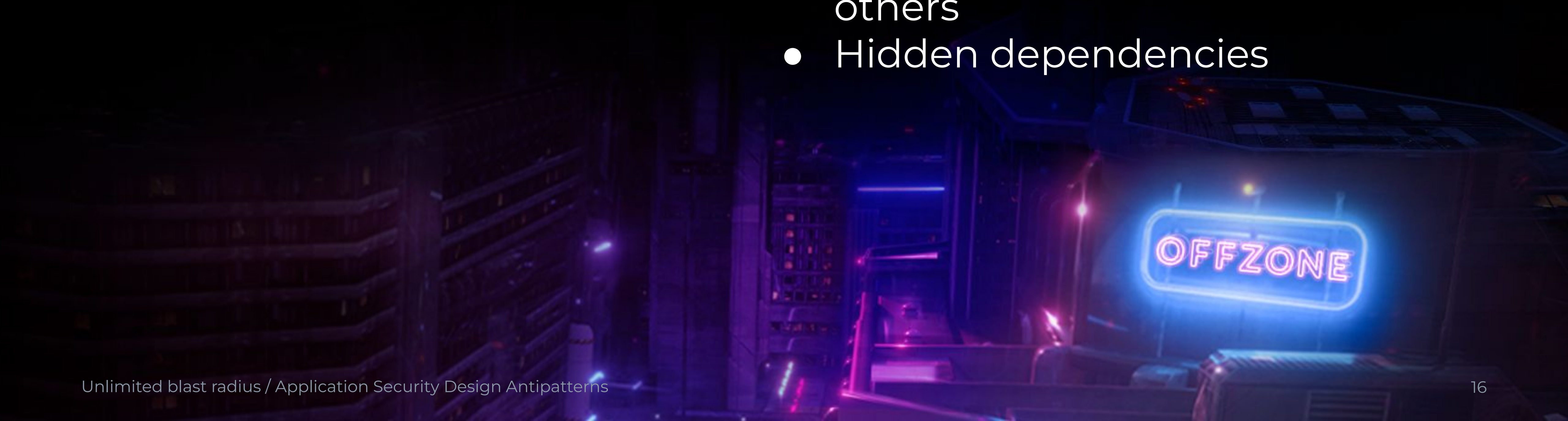
Unlimited blast radius

Reasons

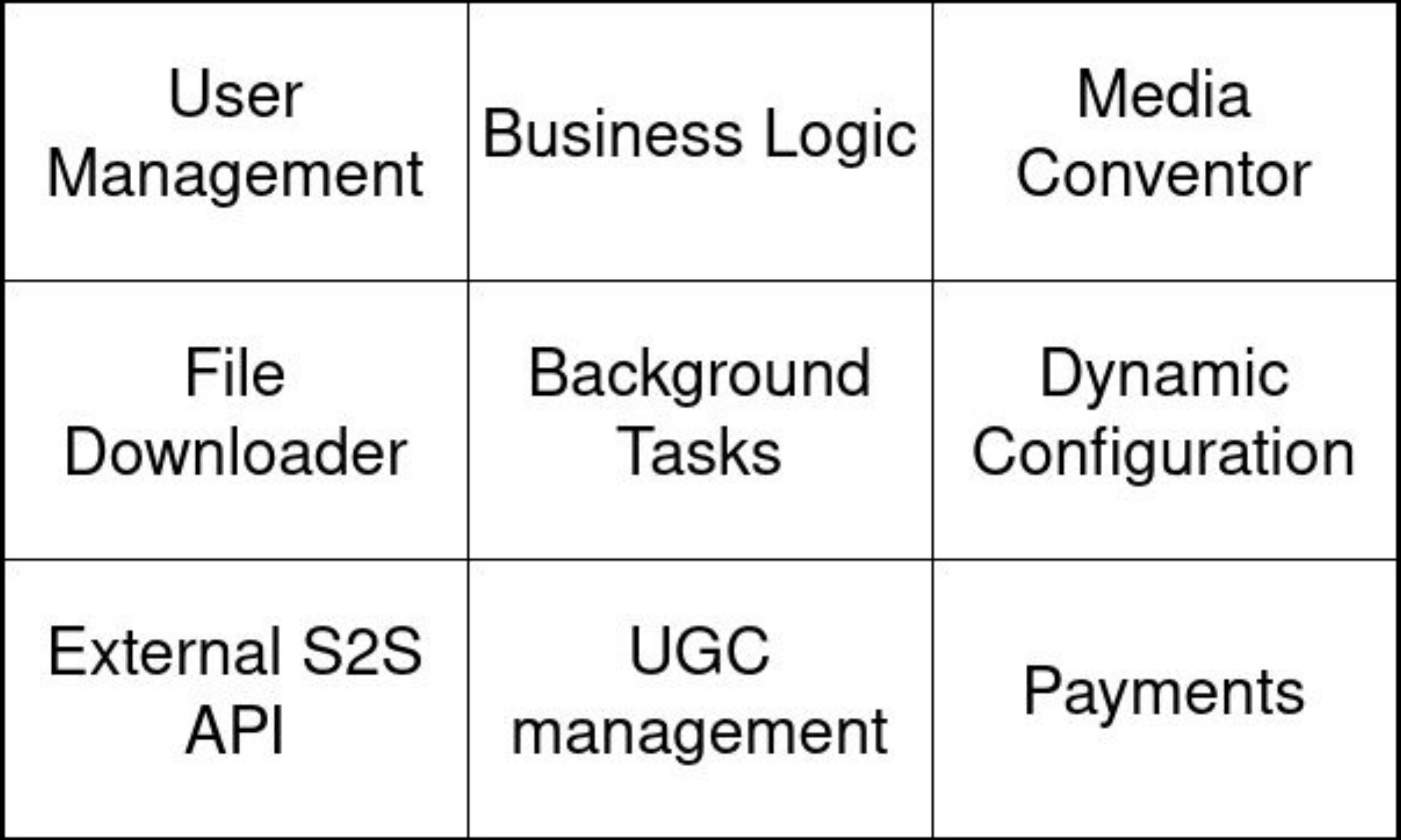
- Fast growing service
- Monolith's legacy

Consequences

- Compromising one component compromise others
- Hidden dependencies

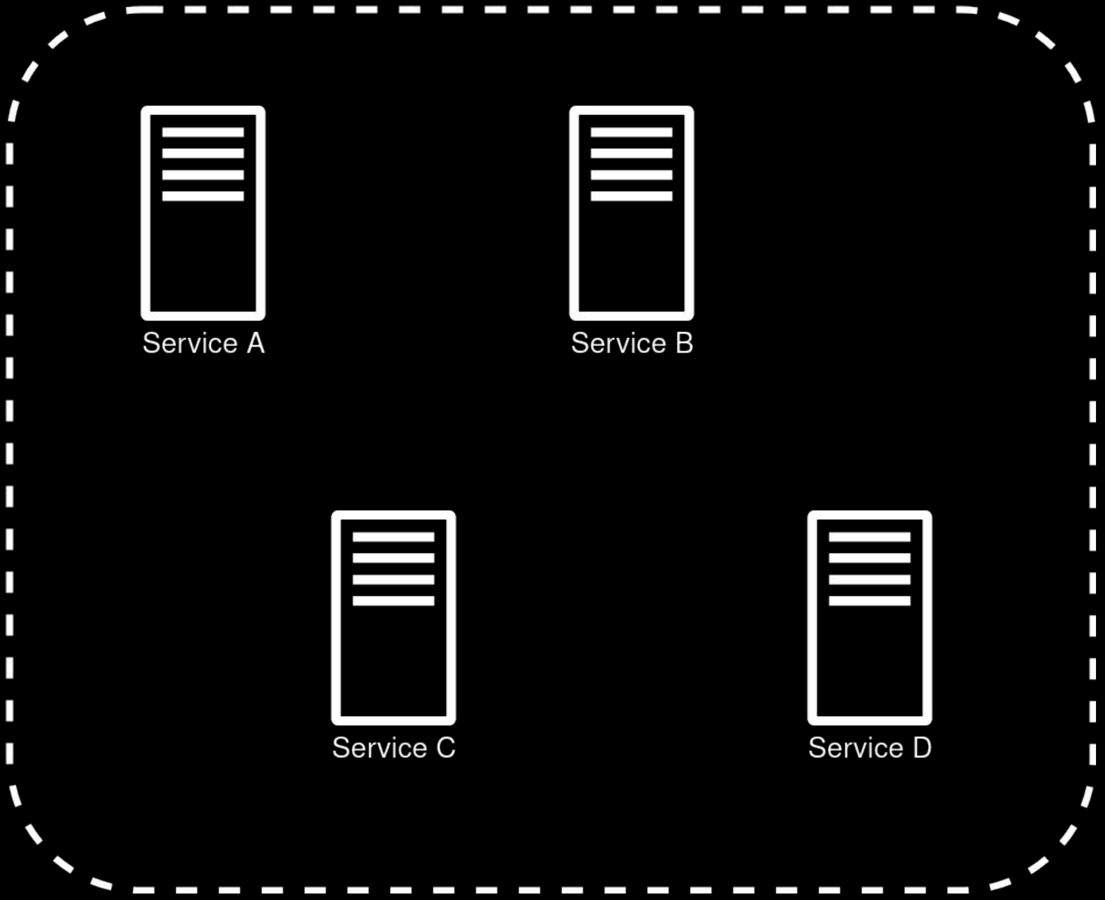


Example #1: Monolith

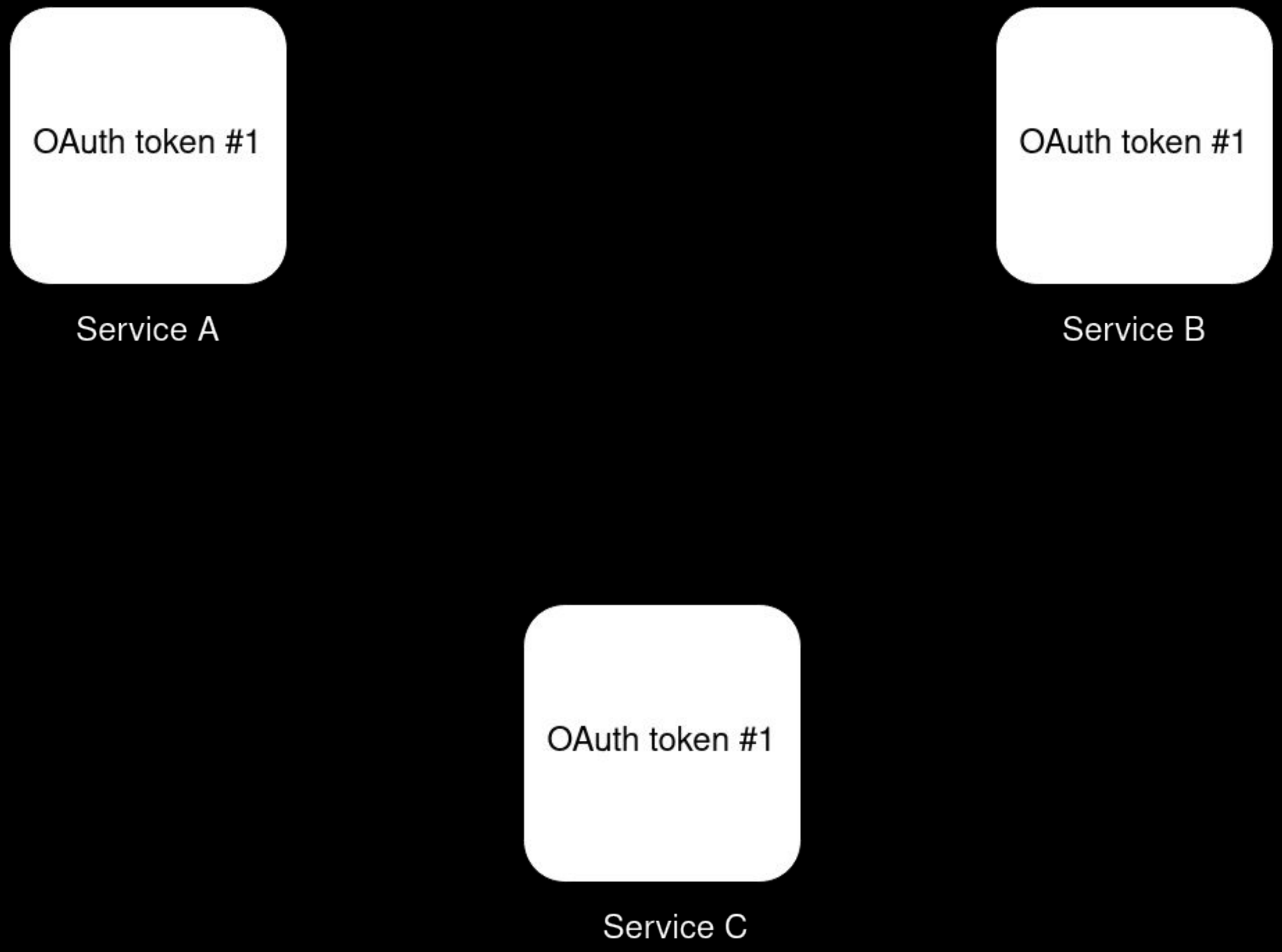


User Management	Business Logic	Media Conventor
File Downloader	Background Tasks	Dynamic Configuration
External S2S API	UGC management	Payments

Example #2: Cloud account overcrowding



Example #3: Shared secrets



How to detect

- What happens if some components are compromised?

How to avoid

- Separation & Isolation

Antipattern #3: Insecure by default

Insecure by default

The contract offers non-secure defaults or makes unclear assumptions about the calling code. The consumer has to make efforts for secure usage.

Insecure by default

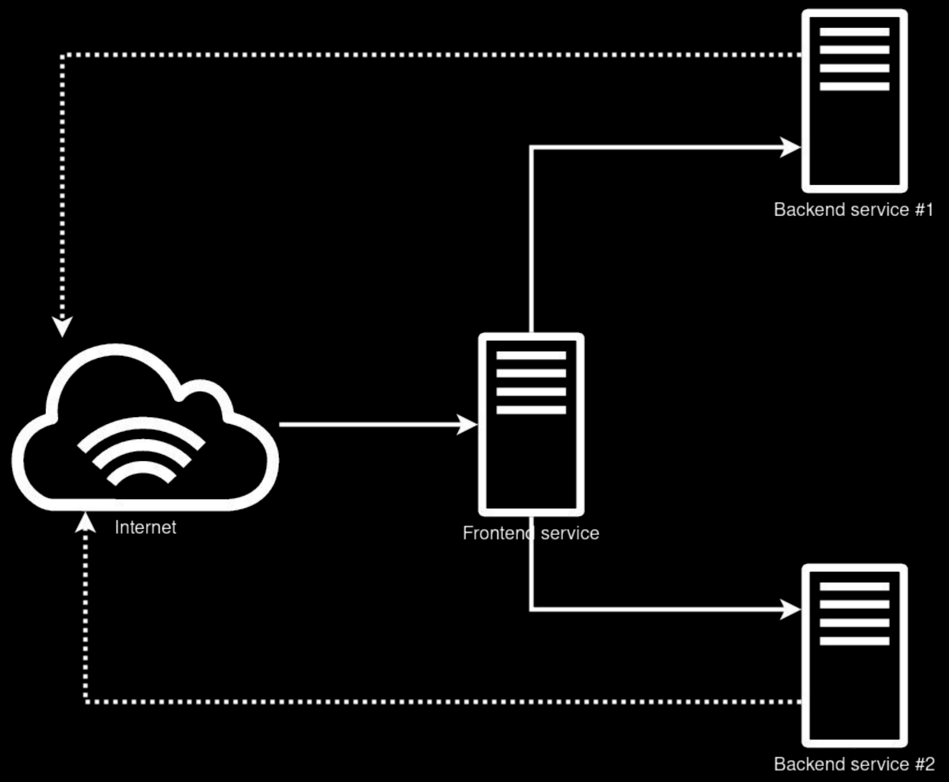
Reasons

- Provide "easy" way to request/call for all cases via hidden complexity

Consequences

- Prone to vulnerabilities

Example #1: Direct Internet Access



Example #2: "Allow by default" policy

```
@role(MODERATOR)
 int moderatorHandler() {}
 @role(ADMIN)
 void adminHandler() {}
 // ???
 void anotherHandler() {}
```

Example #3: Confusion naming

```
dangerouslySetInnerHTML =
{{__html: data}}
```

VS

```
el.innerHTML = data;
```

Example #4: Implicit features

```
SAXParserFactory factory = SAXParserFactory.newInstance();
// to be compliant, completely disable DOCTYPE declaration:
factory.setFeature("http://apache.org/xml/features/disallow-doctyp
e-decl", true);
```

How to detect

- What assumptions do we have about data, caller code, etc.?

How to avoid

- Defaults should be safe for use
- Explicit is better than implicit
- Deny by default

ON
FF
ONE
2022

Antipattern #4: Security by obscurity

Security by obscurity

Security is based on fact that attacker doesn't know implementation details

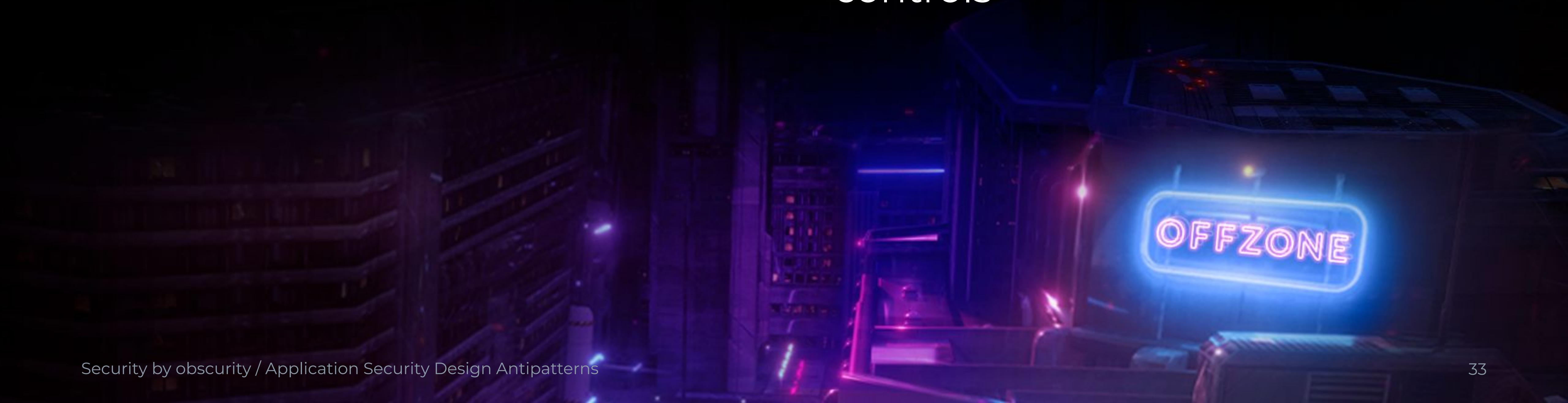
Security by obscurity

Reasons

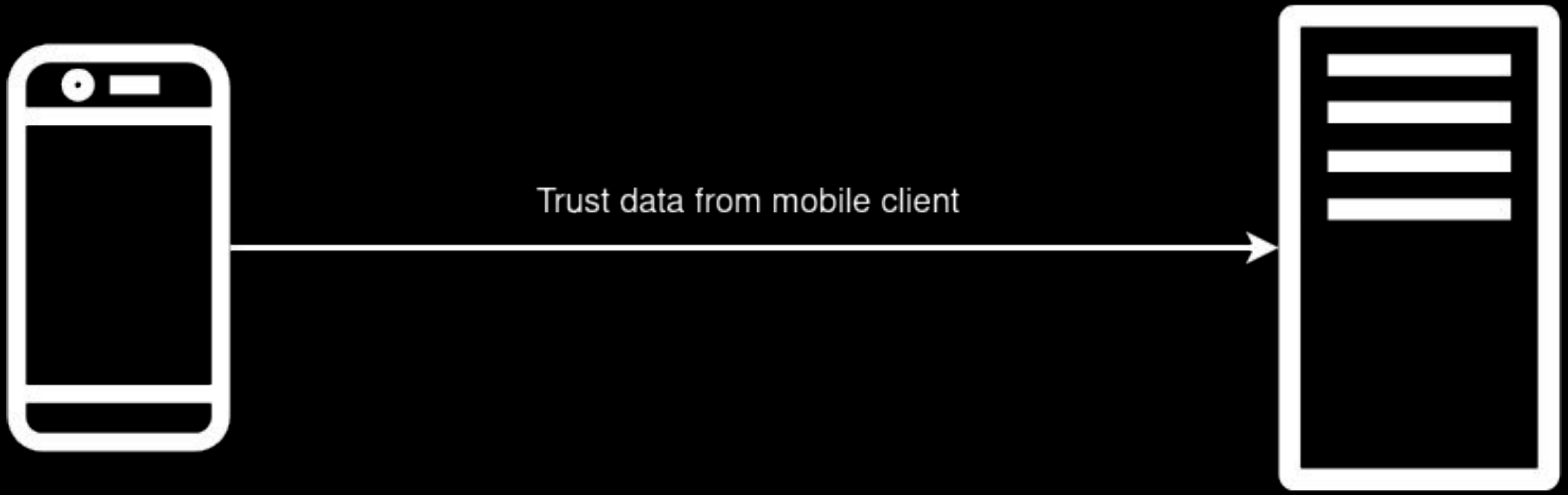
- Lack of full knowledge about platform

Consequences

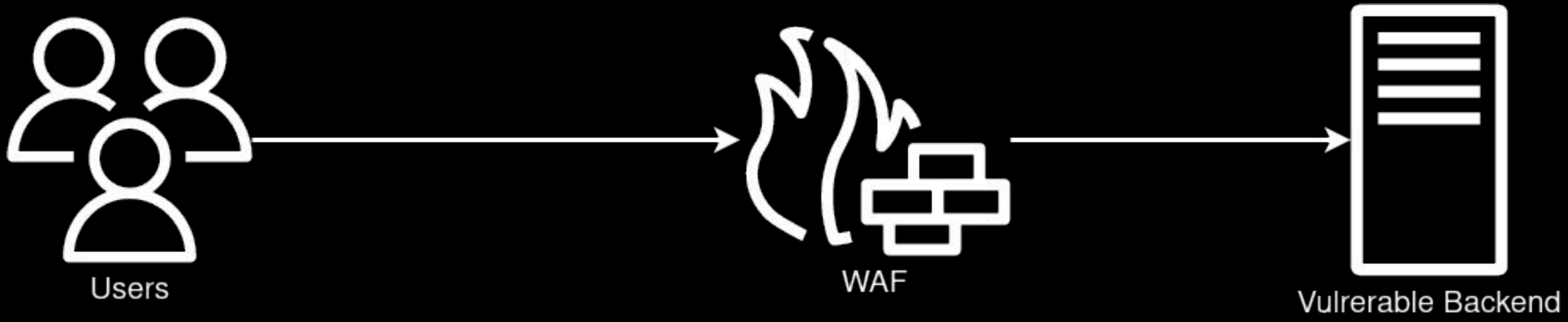
- Reverse engineering can find way to bypass security controls



Example #1: Client side controls



Example #2: Using WAF instead of real patching



How to detect

- Check trust boundaries

How to avoid

- Always implement controls on server side

OFFZONE 2022

Antipattern #5: Uncontrolled access

Uncontrolled access

Lack of sufficient control over access to important data

Uncontrolled access

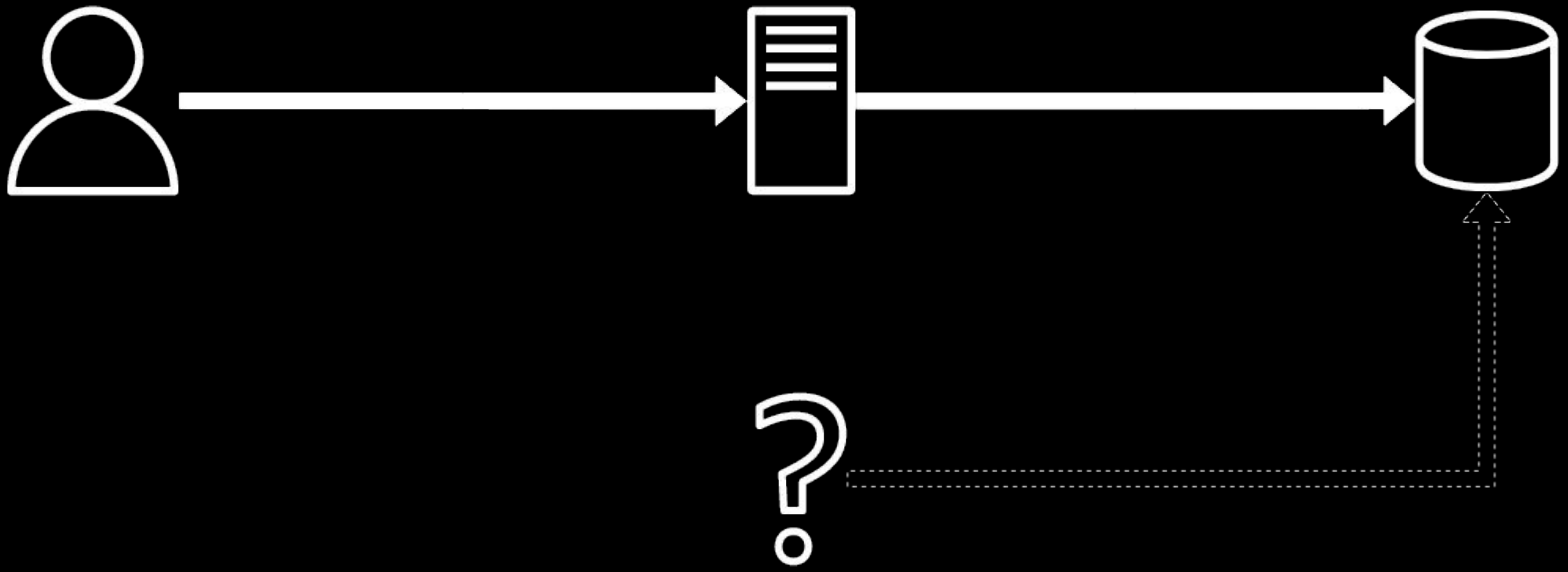
Reasons

- Lack of control and inventory

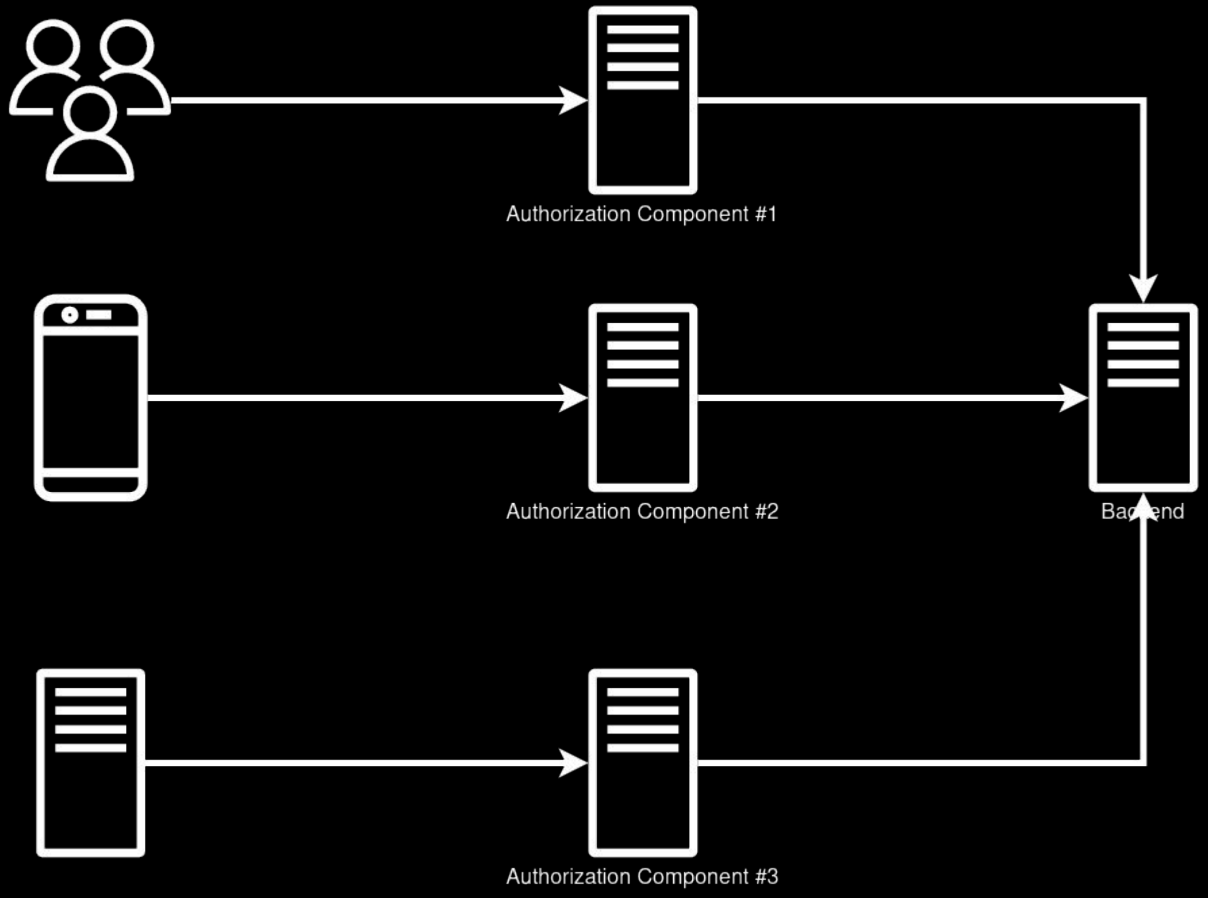
Consequences

- Inconsistent access control

Example #1: Uncontrolled 3d-party access



Example #2: Multiple Authorization Points


Authorization Component #1
Authorization Component #2
Backend
Authorization Component #3

How to detect

- Is there any other way to access data?
- What should we do to change the access policy?

How to avoid

- Enforce access control policy in one place
- Inventory of all access points

Antipattern #6: Incidental complexity

Incidental complexity

Solution that is hard to verify from security perspective. Solution can be simplified

Incidental complexity

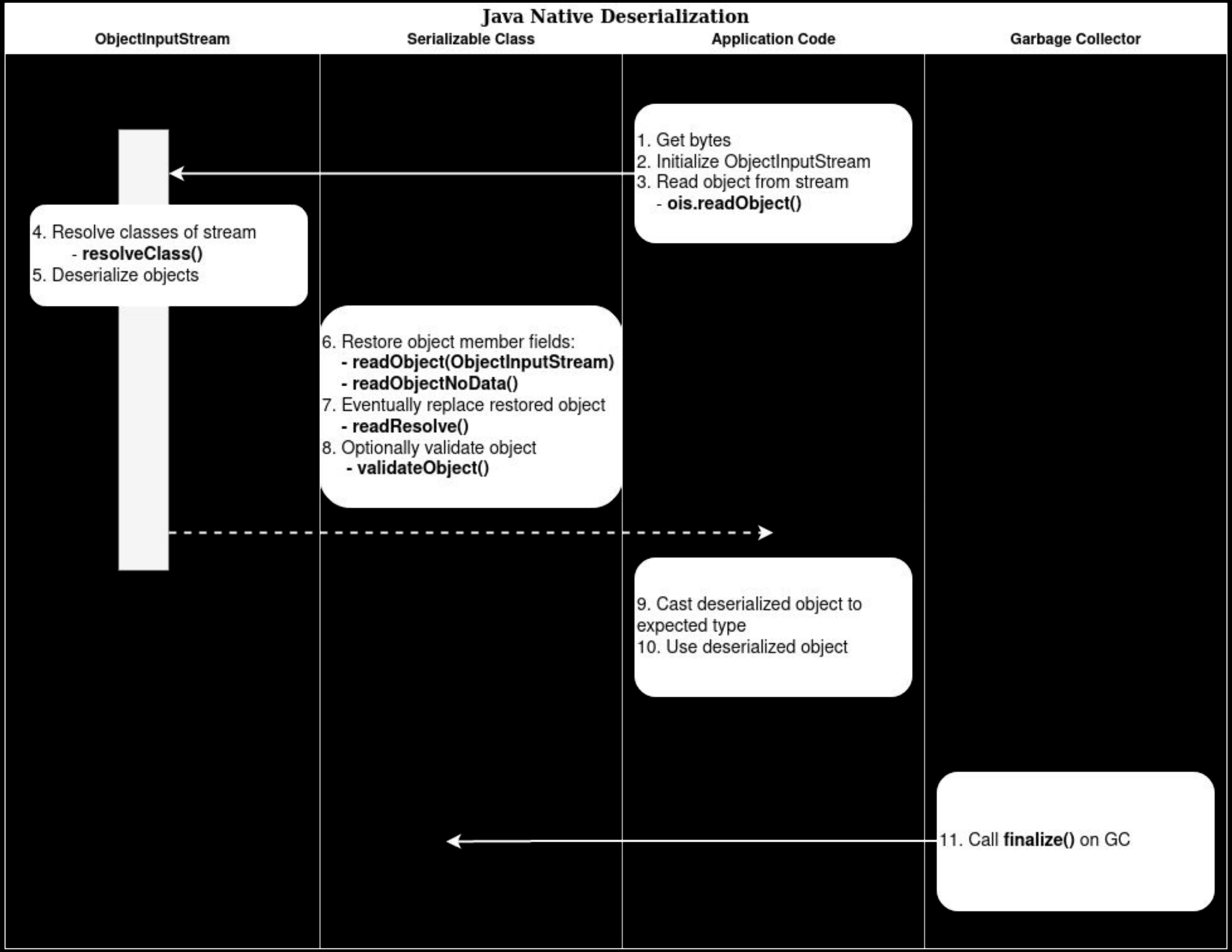
Reasons

- Too customizable

Consequences

- Vulnerabilities in "hidden" functionality

Example #1: Java deserialization


Java Native Deserialization
ObjectInputStream
Serializable Class
Application Code
Garbage Collector
1. Get bytes
2. Initialize ObjectInputStream
3. Read object from stream
- **ois.readObject()**
4. Resolve classes of stream
- **resolveClass()**
5. Deserialize objects
6. Restore object member fields:
- readObject(ObjectInputStream)
- readObjectNoData()
7. Eventually replace restored object
- readResolve()
8. Optionally validate object
- validateObject()
9. Cast deserialized object to expected type
10. Use deserialized object
11. Call **finalize()** on GC

How to detect

- Can we simplify the functionality?

How to avoid

- Keep It Simple Stupid (KISS)

OFF ONE 2022

Antipattern #7: Reinventing the wheel

Reinventing the wheel

Re-implementing the same solution over and over again for different services

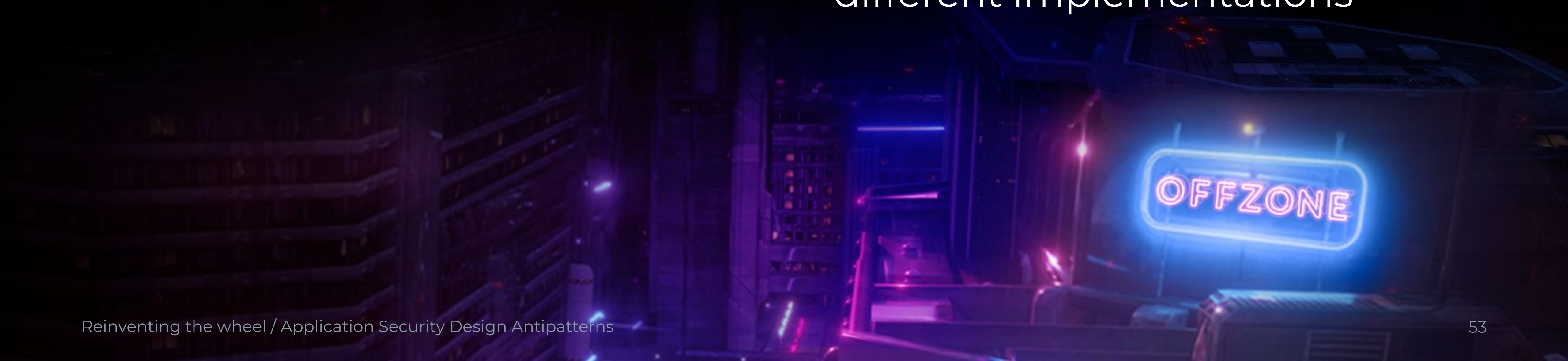
Reinventing the wheel

Reasons

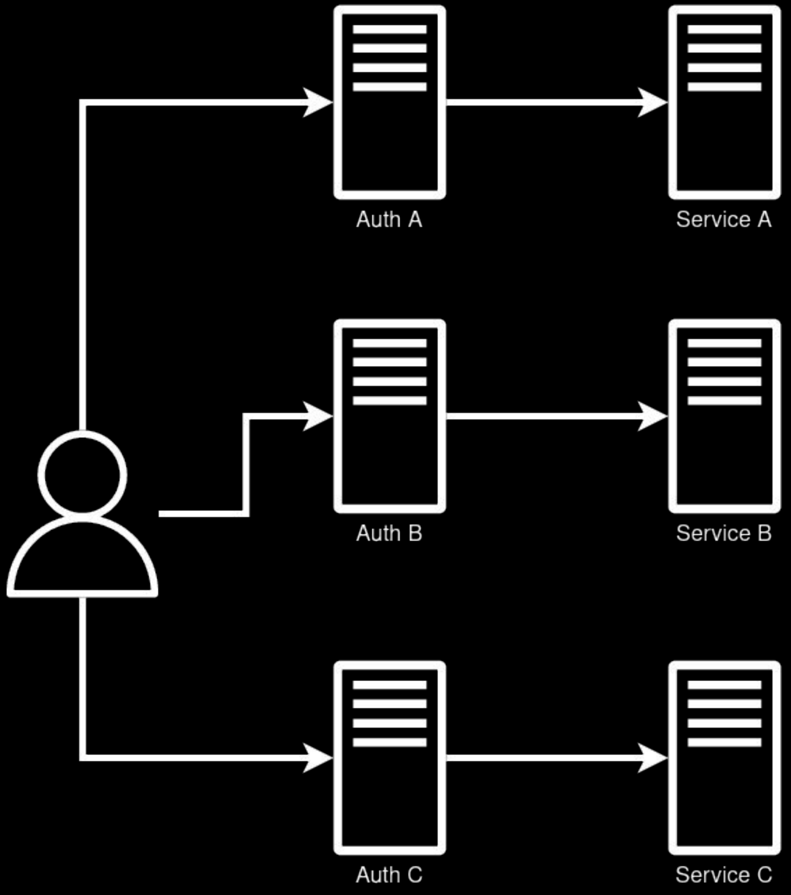
- Lack of customisation for a centralized solution

Consequences

- Difficulty of scaling centralized solutions
- The same problems occur in different implementations



Example #1: Custom Auth for each service



How to detect

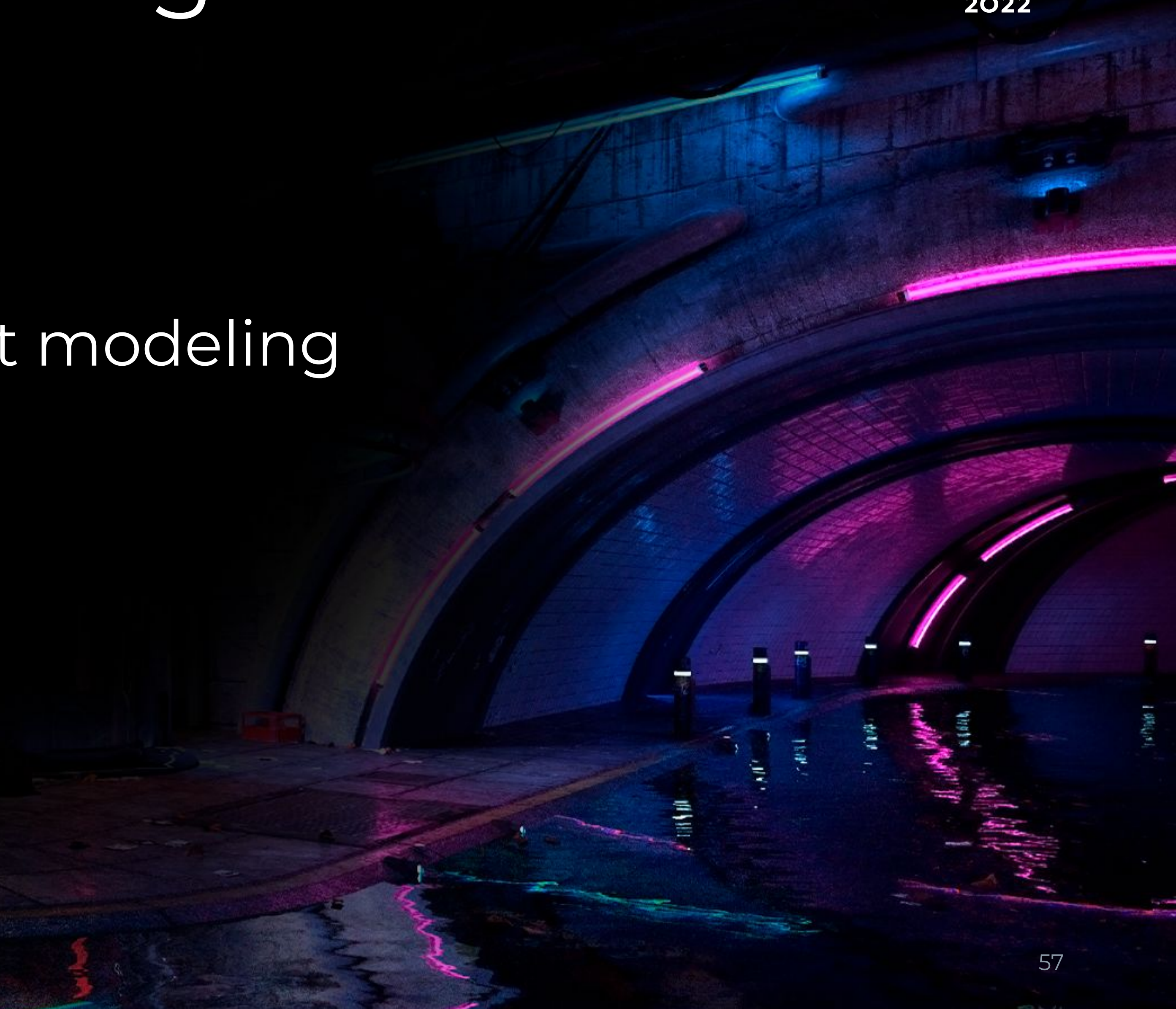
- Do we already have a solution to this problem?
- Do we solve similar problems over and over again?
- Can custom functionality be more efficient if it's a centralized solution?

How to avoid

- Use a centrally approved solution

Strategies for working with antipatterns

- Developer awareness
- Questions during a threat modeling session



Conclusion

- Antipatterns have a long-term impact on security
- The implicit use of an antipattern can create additional security problems

Contacts:
@aameshcheriakov
aameshcheriakov@gmail.com

https://github.com/tank1st99/appsec-antipatterns