Secure by Design

@DanielDeogun @danbjson

Jfokus, Tutorial Day

February 5, 2018

omega point.

About Us...

Daniel Deogun
Coder and Quality Defender

Omegapoint, Sweden

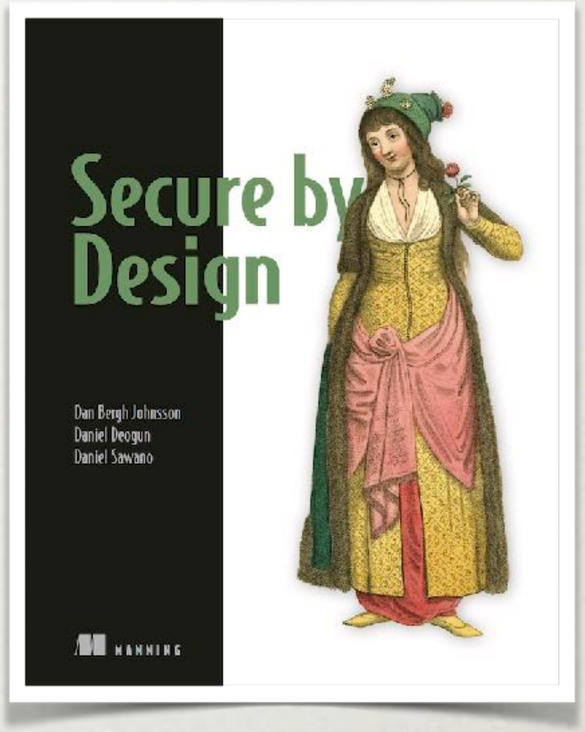
Dan Bergh Johnsson
Secure Domain Philosopher

Key Take Aways

- Good design can yield a lot of security benefits
- Secure by design is a mindset
- Domain primitives enhances security in depth
- Model sensitive data in each context
- Reduce complexity of entities
- The three R's significantly reduces the attack window

Agenda

- A Bank Robbery
- Secure by Design
- Technical solution to XSS
- Domain Primitives
- Sensitive Data Exposure
- XML and a Billion Laughs
- Automatic (security) unit tests
- Domain Primitives Applied on Legacy
- Fighting Entity Complexity
- Cloud Thinking



A Bank Robbery

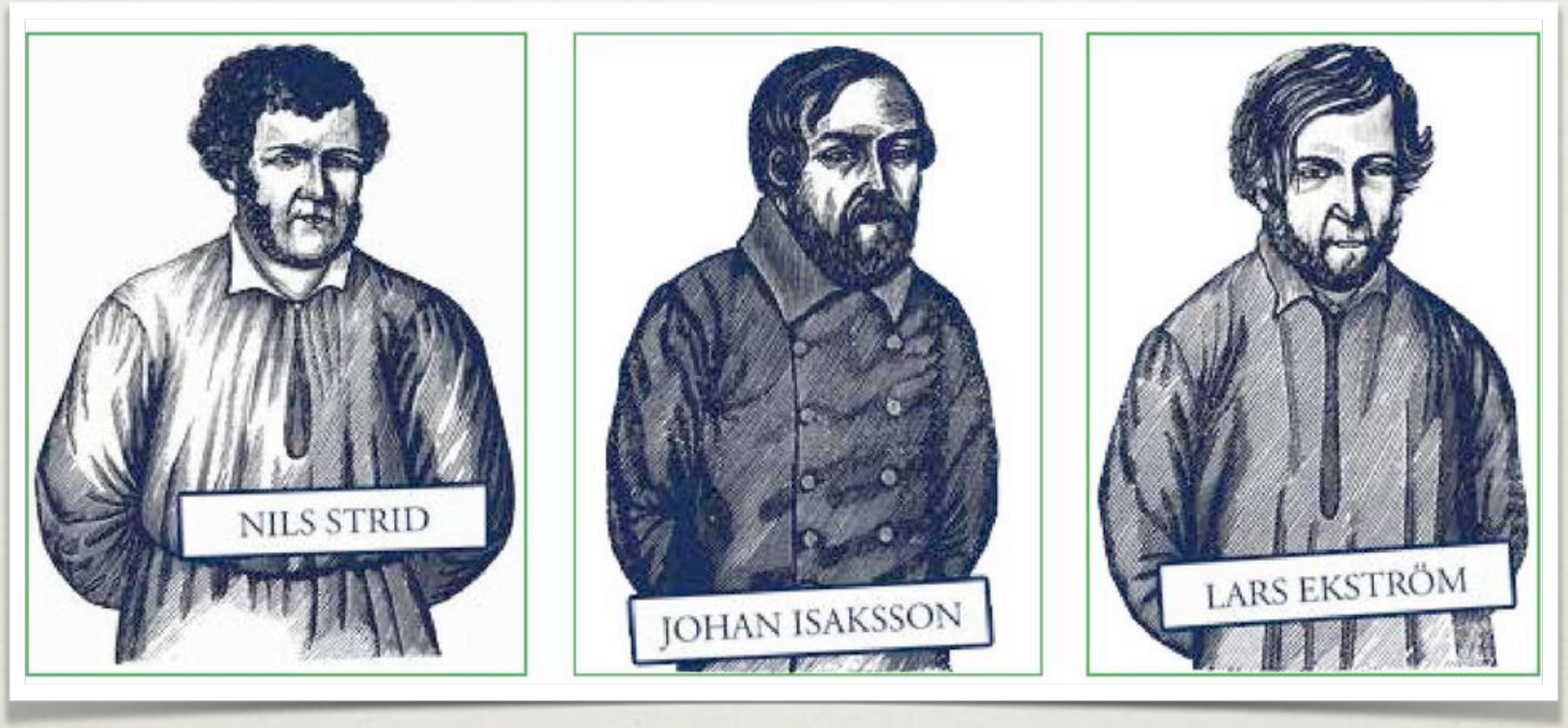


image: Smådisigt Medlemsblad för DIS-Småland

OWASP Top 10

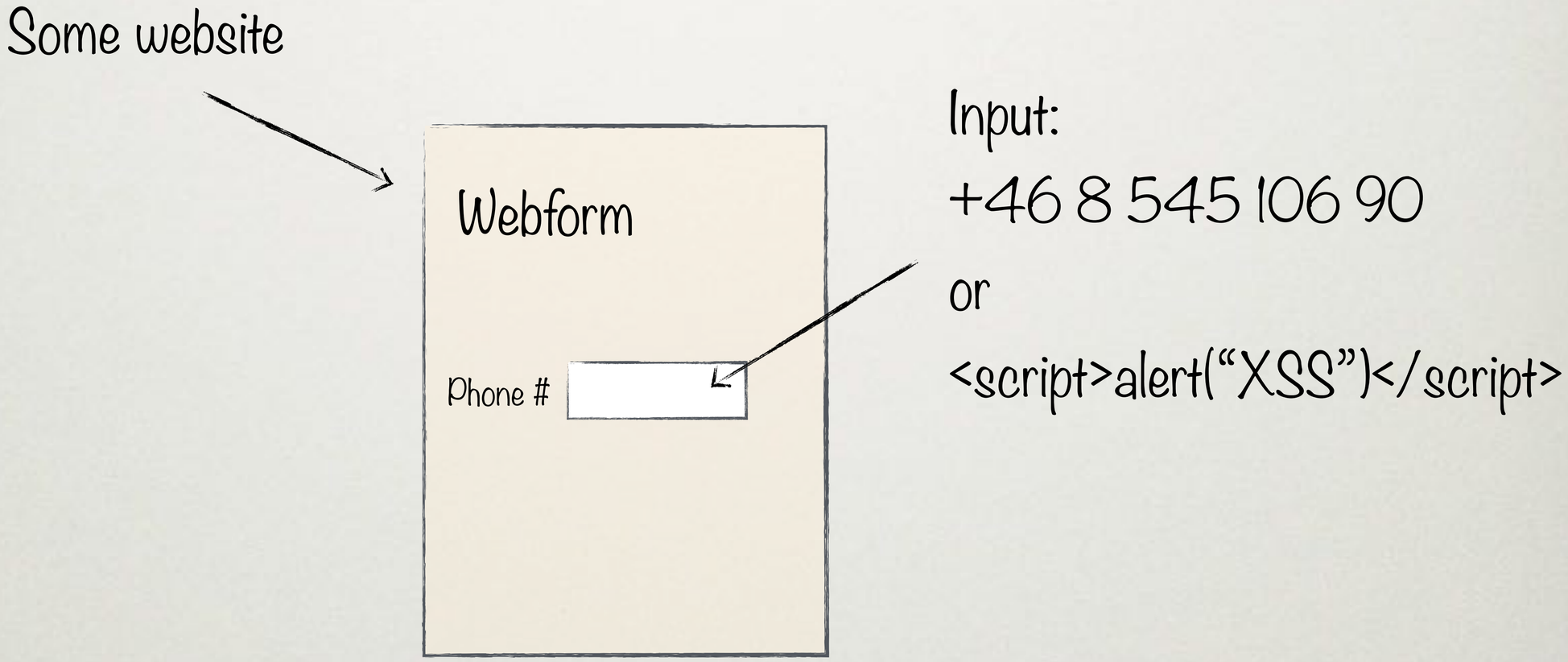
Top 10 2017
A1 - Injection
A2 - Broken Authentication
A3 - Sensitive Data Exposure
A4 - XML External Entities (XXE)
A5 - Broken Access Control
A6 - Security Misconfiguration
A7 - Cross-Site Scripting (XSS)
A8 - Insecure Deserialization
A9 - Using Components with Known Vulnerabilities
A10 - Insufficient Logging&Monitoring

Top 10 2013
A1 - Injection
A2 - Broken Authentication and Session Management
A3 - Cross-Site Scripting (XSS)
A4 - Insecure Direct Object References
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross-Site Request Forgery (CSRF)
A9 - Using Components with Known Vulnerabilities
A10 - Unvalidated Redirects and Forwards

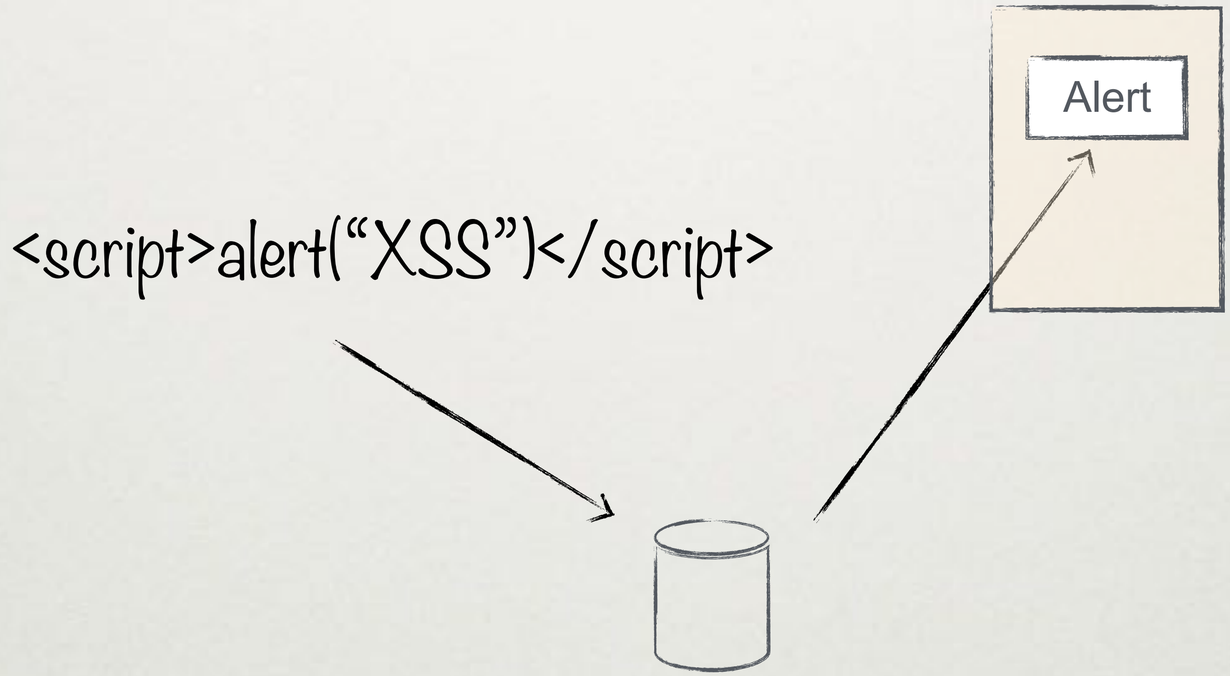
Top 10 2010
A1 - Cross Site Scripting (XSS)
A2 - Injection Flaws
A3 - Malicious File Execution
A4 - Insecure Direct Object Reference
A5 - Cross Site Request Forgery (CSRF)
A6 - Information Leakage and Improper Error Handling
A7 - Broken Authentication and Session Management
A8 - Insecure Cryptographic Storage
A9 - Insecure Communications
A10 - Failure to Restrict URL Access

Top 10 2004
A1 - Unvalidated Input
A2 - Broken Access Control
A3 - Broken Authentication and Session Management
A4 - Cross Site Scripting
A5 - Buffer Overflow
A6 - Injection Flaws
A7 - Improper Error Handling
A8 - Insecure Storage
A9 - Application Denial of Service
A10 - Insecure Configuration Management

Cross Site Scripting (XSS)

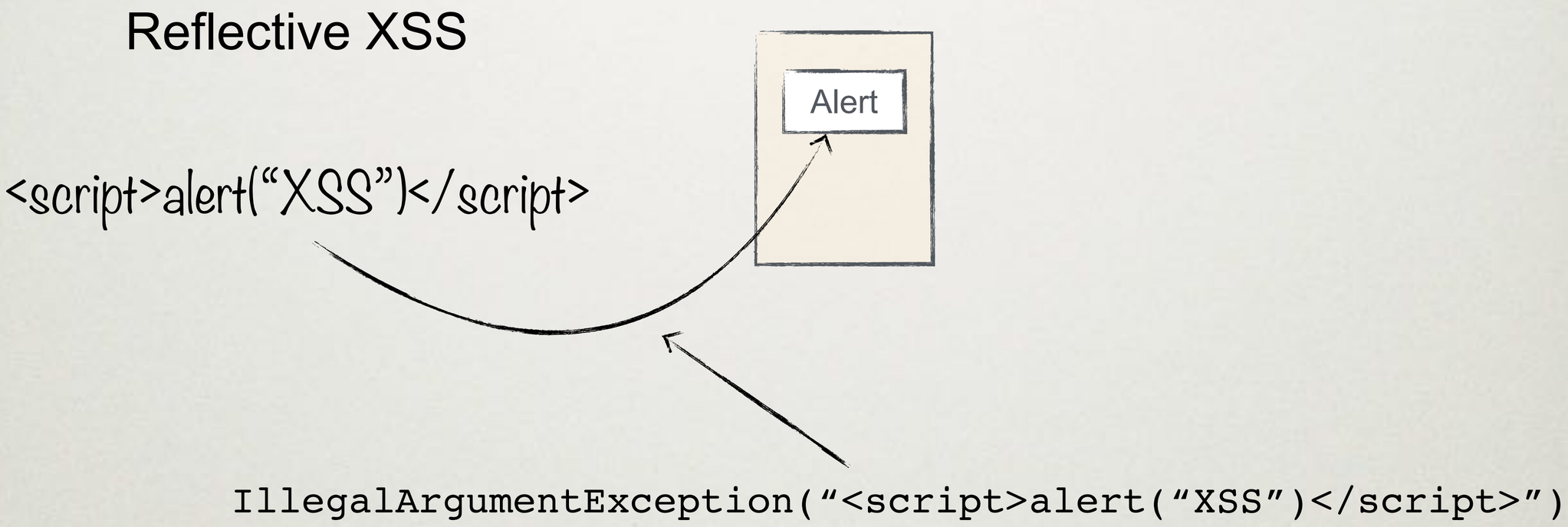


Stored XSS

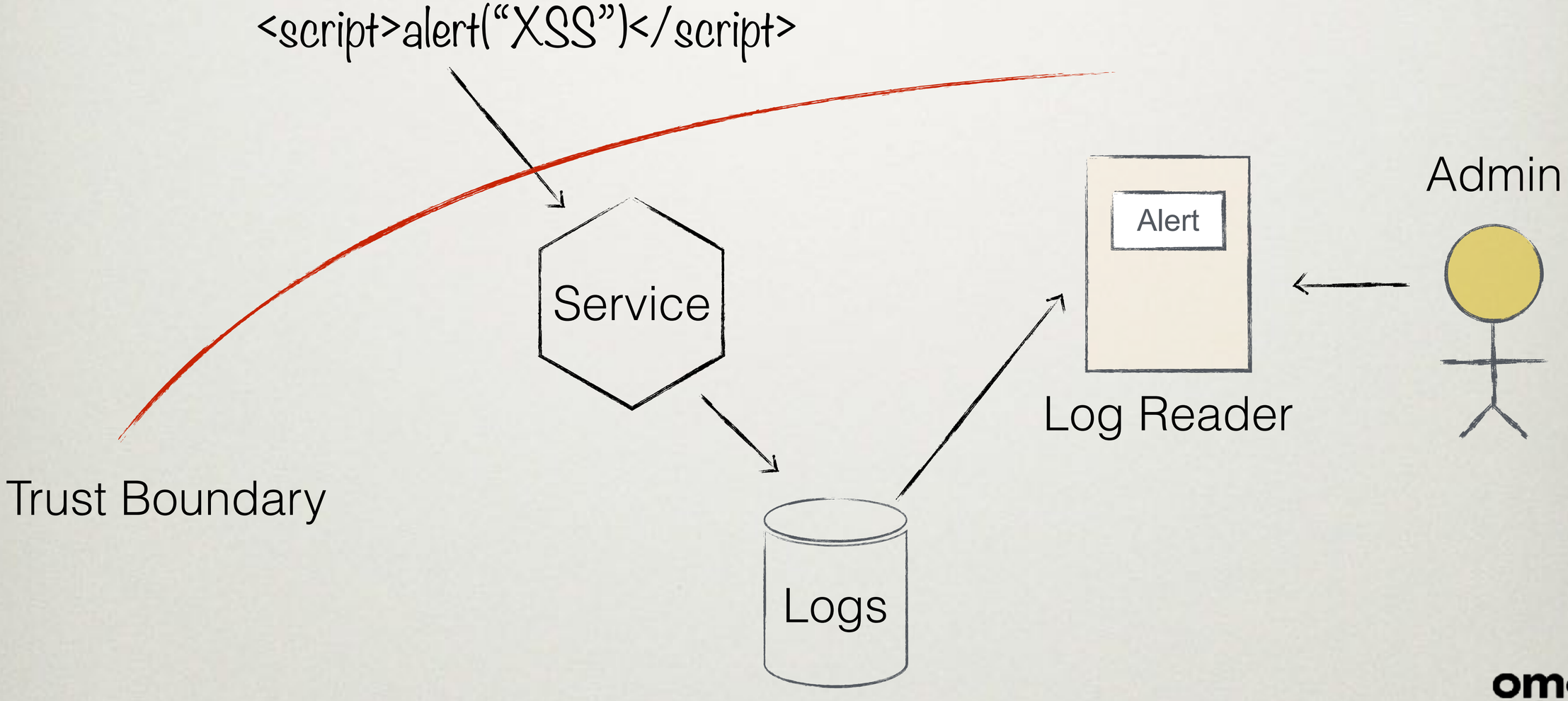


omega point.

Reflective XSS

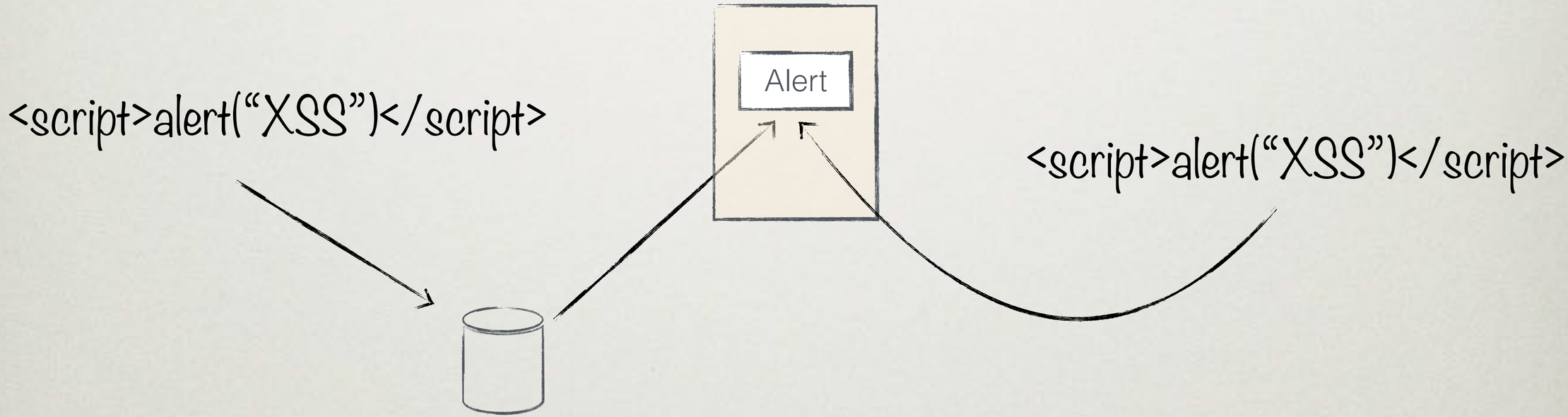


2nd order XSS



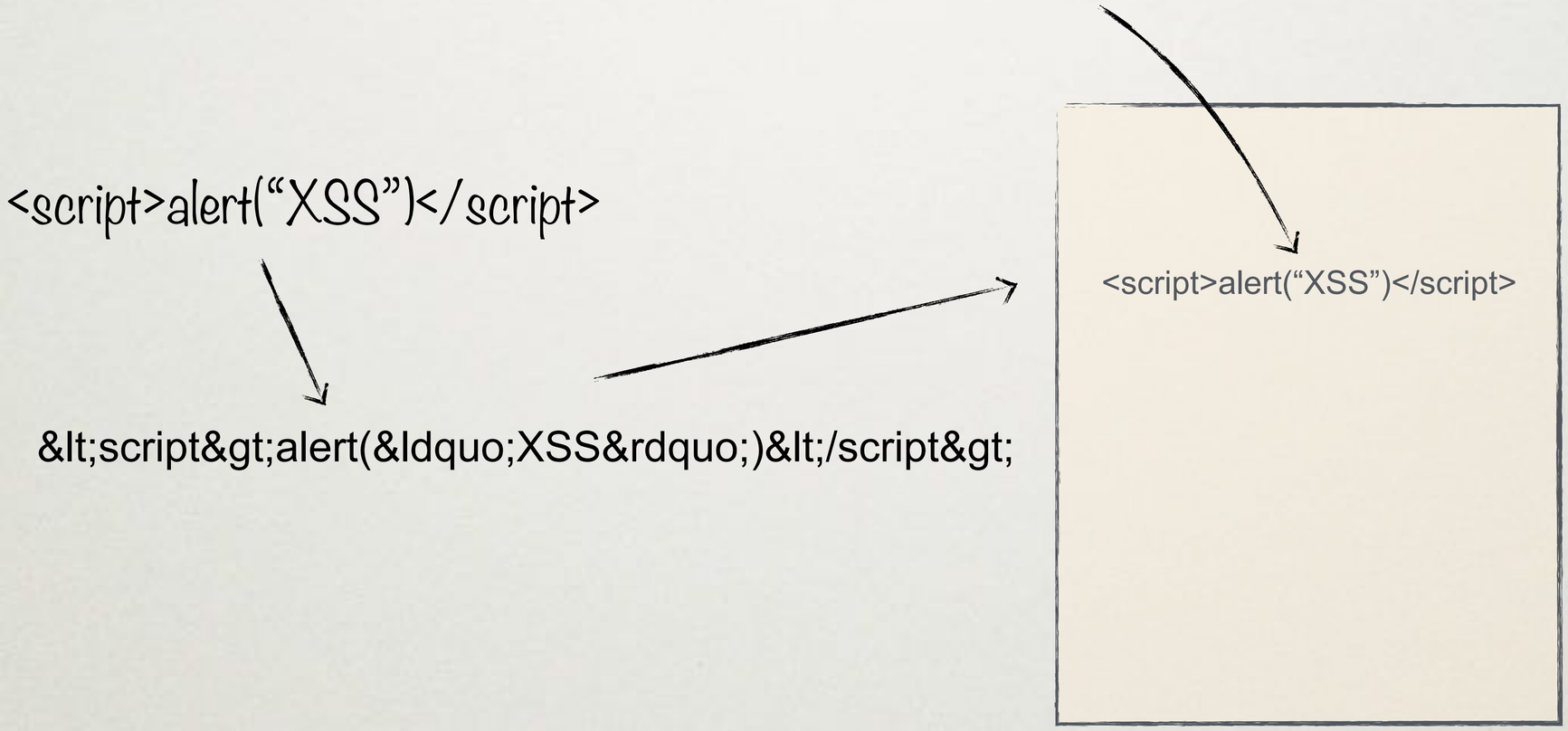
Technical Analysis

"Phone number" isn't escaped properly when rendered on the website – hence, it gets interpreted as code!



Technical Solution

Escape phone number so it can be rendered as text



OWASP Top 10

Top 10 2017
A1 - Injection
A2 - Broken Authentication
A3 - Sensitive Data Exposure
A4 - XML External Entities (XXE)
A5 - Broken Access Control
A6 - Security Misconfiguration
A7 - Cross-Site Scripting (XSS)
A8 - Insecure Deserialization
A9 - Using Components with Known Vulnerabilities
A10 - Insufficient Logging&Monitoring

Top 10 2013
A1 - Injection
A2 - Broken Authentication and Session Management
A3 - Cross-Site Scripting (XSS)
A4 - Insecure Direct Object References
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross-Site Request Forgery (CSRF)
A9 - Using Components with Known Vulnerabilities
A10 - Unvalidated Redirects and Forwards

Top 10 2010
A1 - Cross Site Scripting (XSS)
A2 - Injection Flaws
A3 - Malicious File Execution
A4 - Insecure Direct Object Reference
A5 - Cross Site Request Forgery (CSRF)
A6 - Information Leakage and Improper Error Handling
A7 - Broken Authentication and Session Management
A8 - Insecure Cryptographic Storage
A9 - Insecure Communications
A10 - Failure to Restrict URL Access

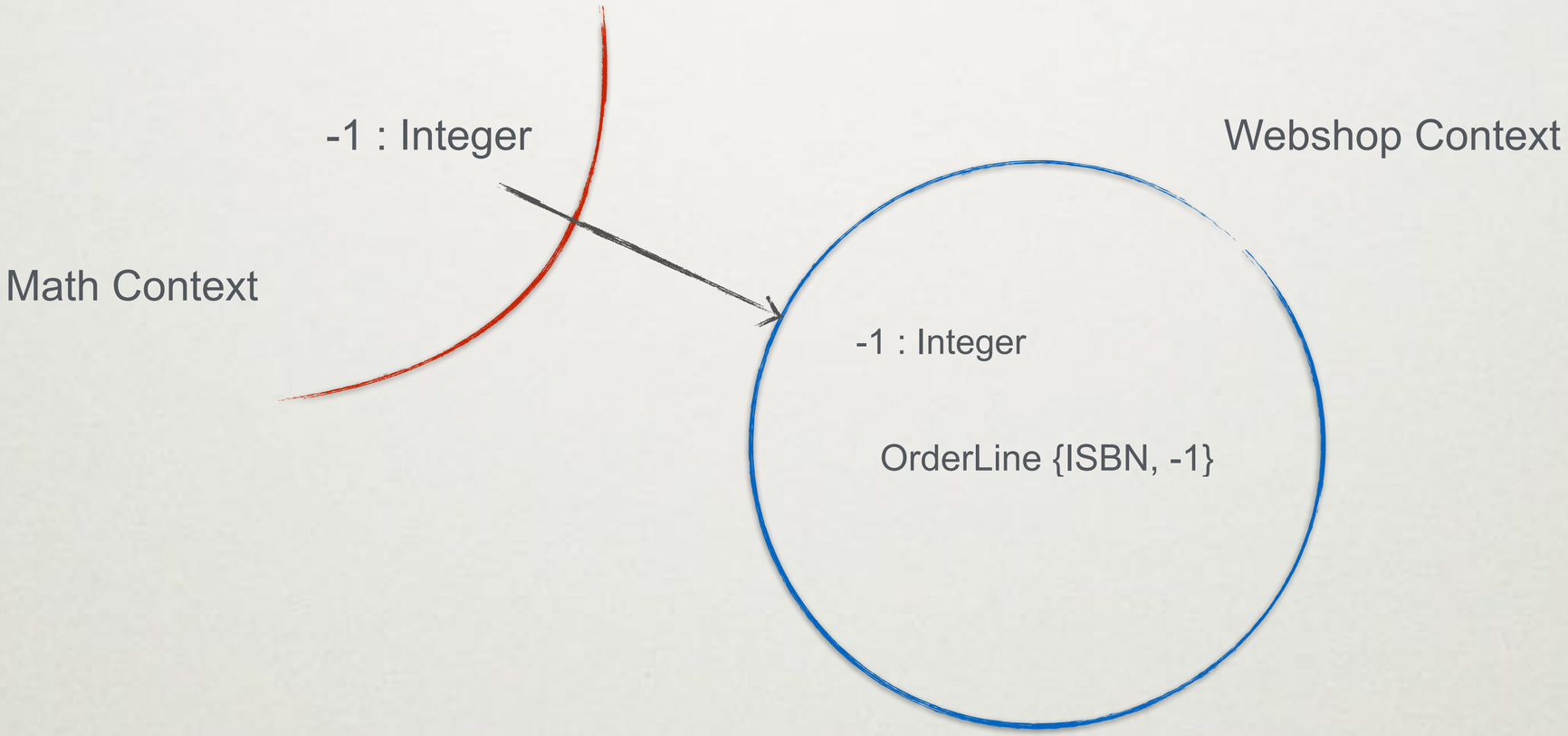
Top 10 2004
A1 - Unvalidated Input
A2 - Broken Access Control
A3 - Broken Authentication and Session Management
A4 - Cross Site Scripting (XSS)
A5 - Buffer Overflow
A6 - Injection Flaws
A7 - Improper Error Handling
A8 - Insecure Storage
A9 - Application Denial of Service
A10 - Insecure Configuration Management

Buying -1 books

Shopping Cart

	1 Secure by Design	$49.99
	-1 Hamlet	$40.50
	1 Hitchhiker's Guide to the Galaxy	$30.00
	Total	$39.49

Analysis


-1 : Integer
Webshop Context
Math Context
-1 : Integer
OrderLine {ISBN, -1}

But Quantity is "just" an integer...

Integers form an Abelian Group

- Closure: a + b = integer
- Associativity: a + (b + c) = (a + b) + c
- Commutativity: a + b = b + a
- Identity: a + 0 = a
- Inverse: a + (–a) = 0

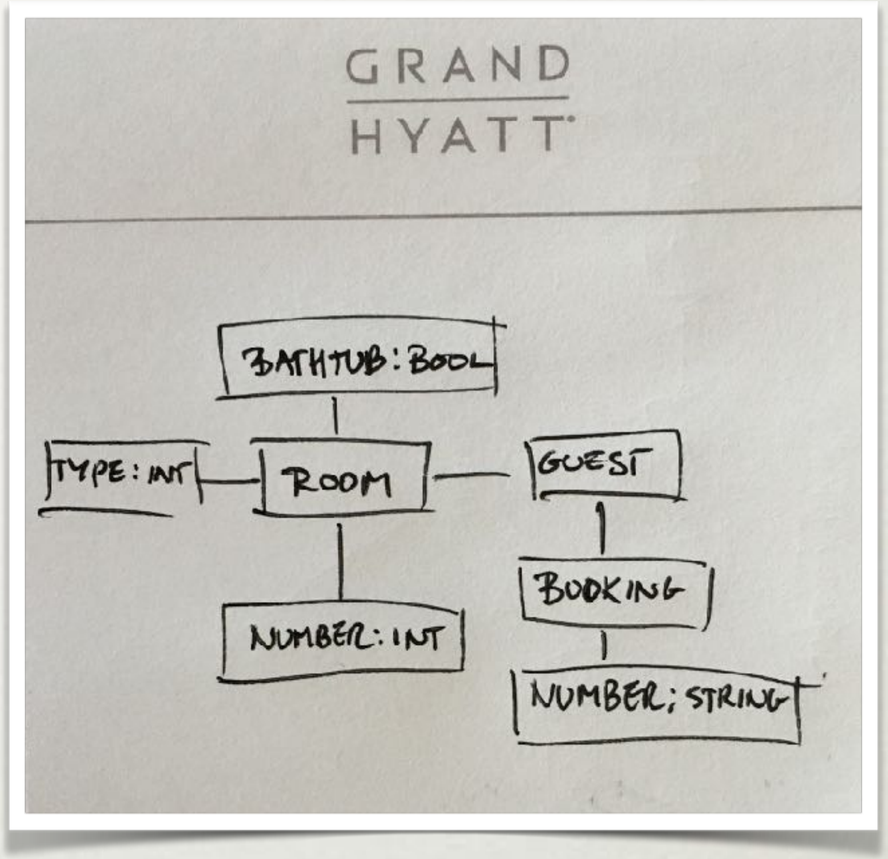
Quantity

- a concept that's well defined with strict boundaries
- not closed under addition
- cannot be negative

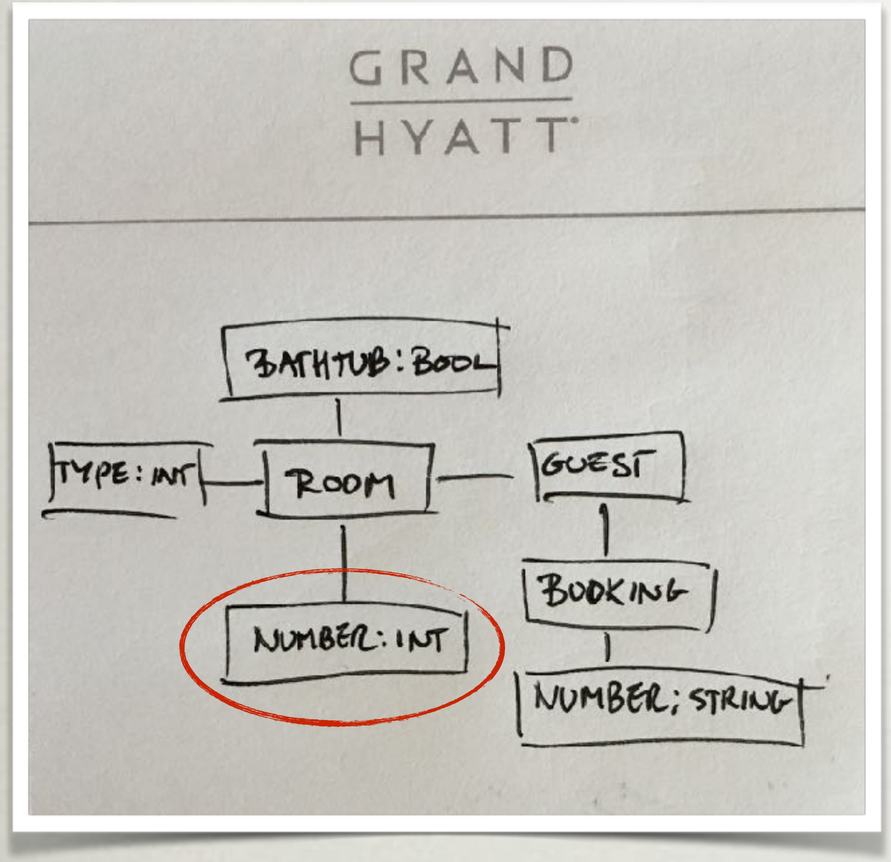
...or is Quantity a String?

- What characters are legal?
- Which operations are allowed?
- Does this make sense?

Let's Model a Hotel Room



But Room Number is "just" an Integer...



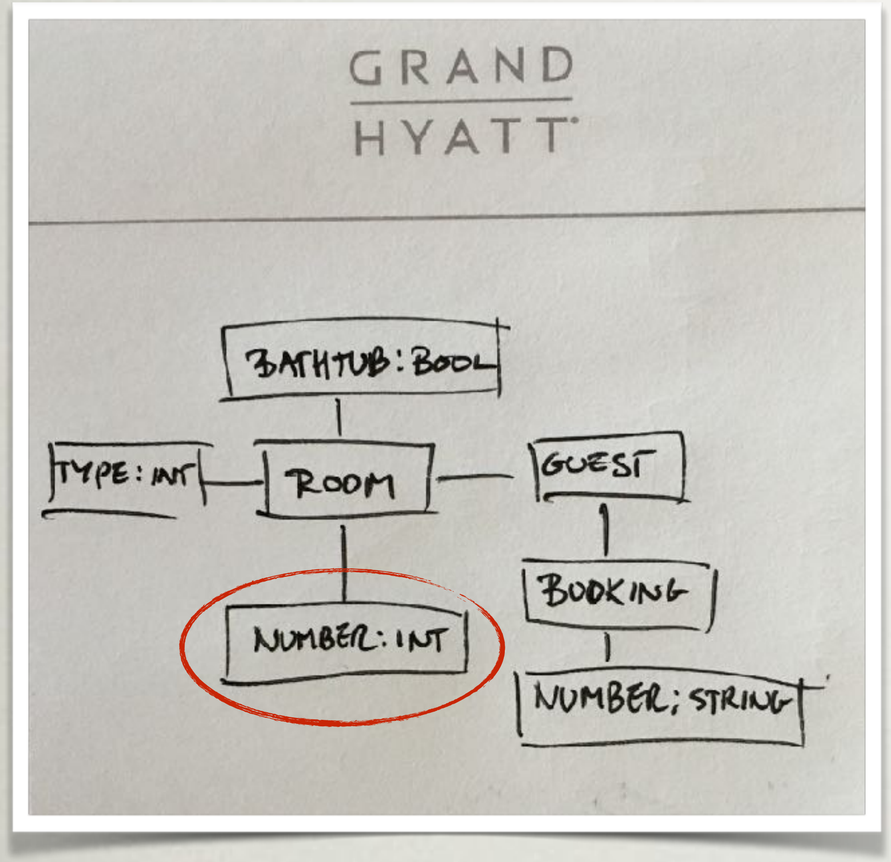
Peano's Axioms

1. Zero is a number
2. If *n* is a number, the successor of *n* is a number
3. Zero isn't the successor of a number
4. Two numbers of which the successors are equal are themselves equal
5. If a set S of numbers contains zero and also the successor of every number in S, then every number is in S

Abelian group

- Closure: a + b = integer
- Associativity: a + (b + c) = (a + b) + c
- Commutativity: a + b = b + a
- Identity: a + 0 = a
- Inverse: a + (−a) = 0

...or is Room Number a String?



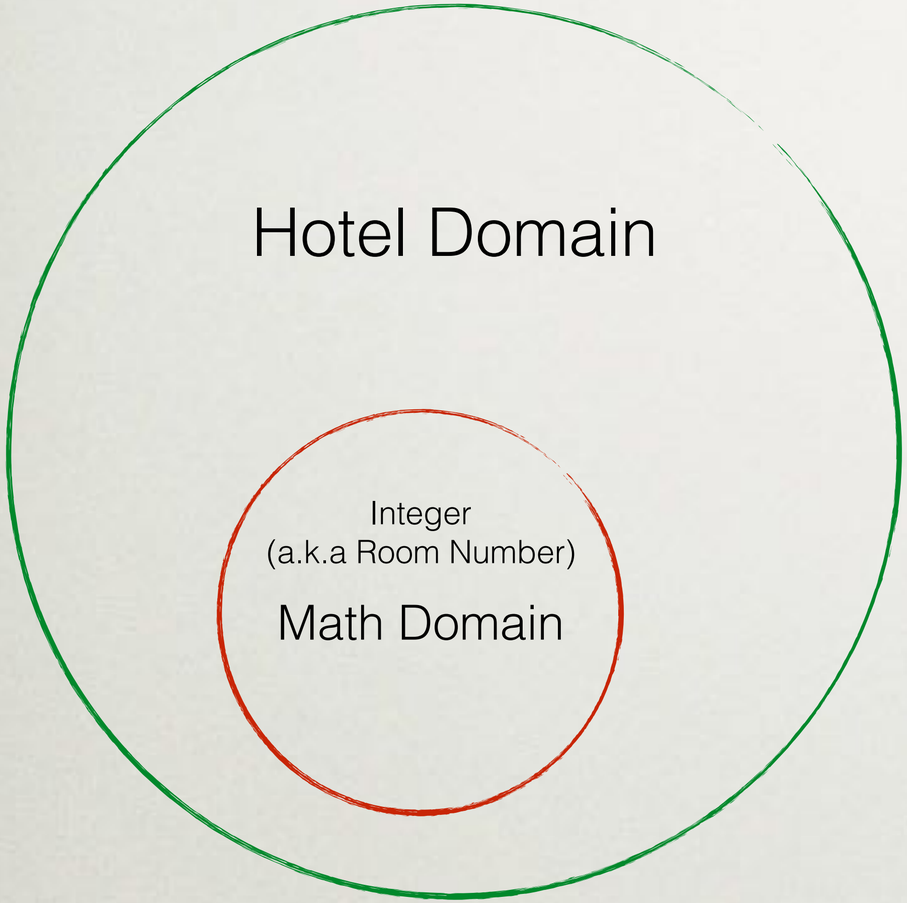
- What characters are legal?
- Which operations are allowed?
- Does this make sense?

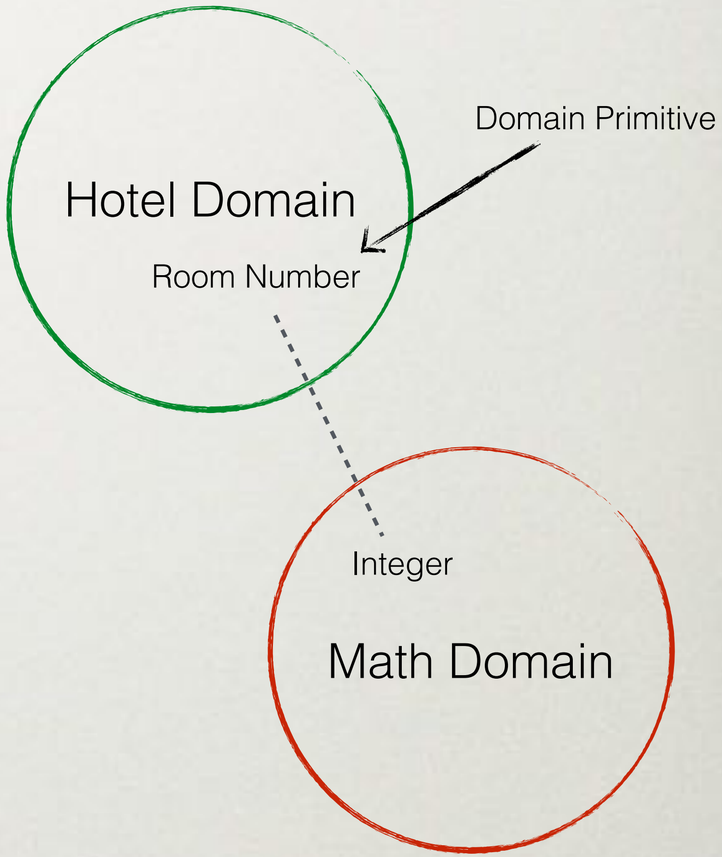
Domain Primitives

> *"A value object so precise in its definition that it, by its mere existence, manifests its validity is called a Domain Primitive."*
>
> - Secure by Design

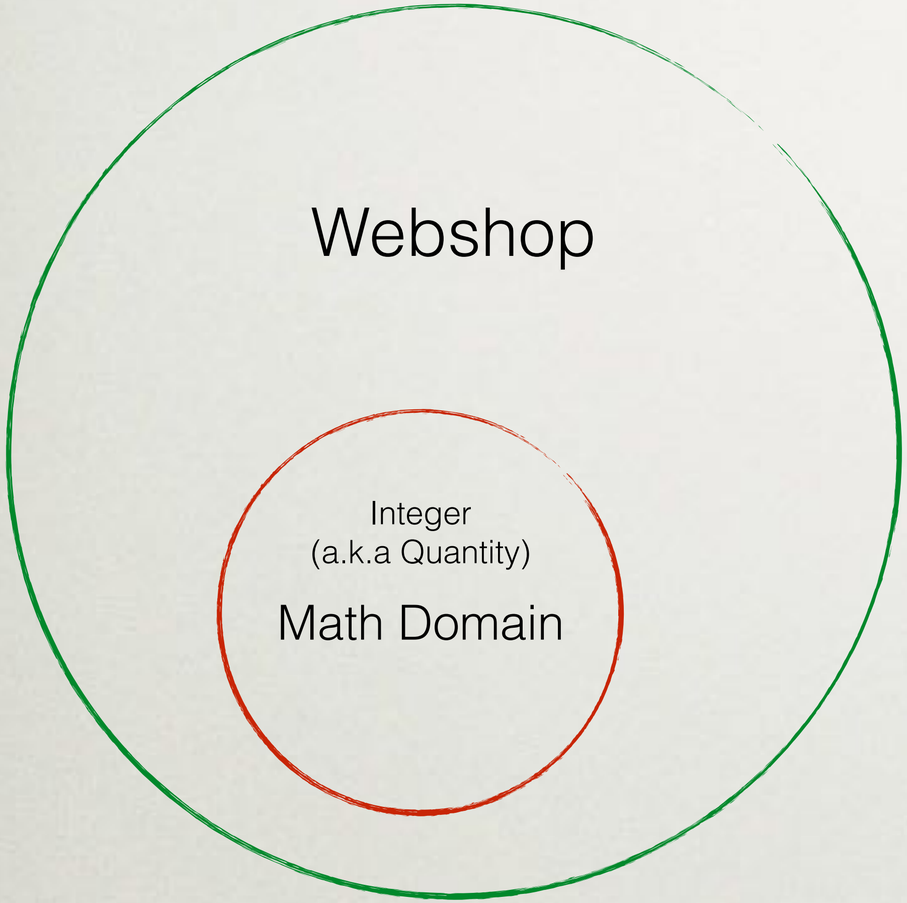
- Can only exist if its value is ***valid***
- Building block that's native to ***your domain***
- Valid in the ***current context***
- ***Immutable*** and resemble a value object in DDD

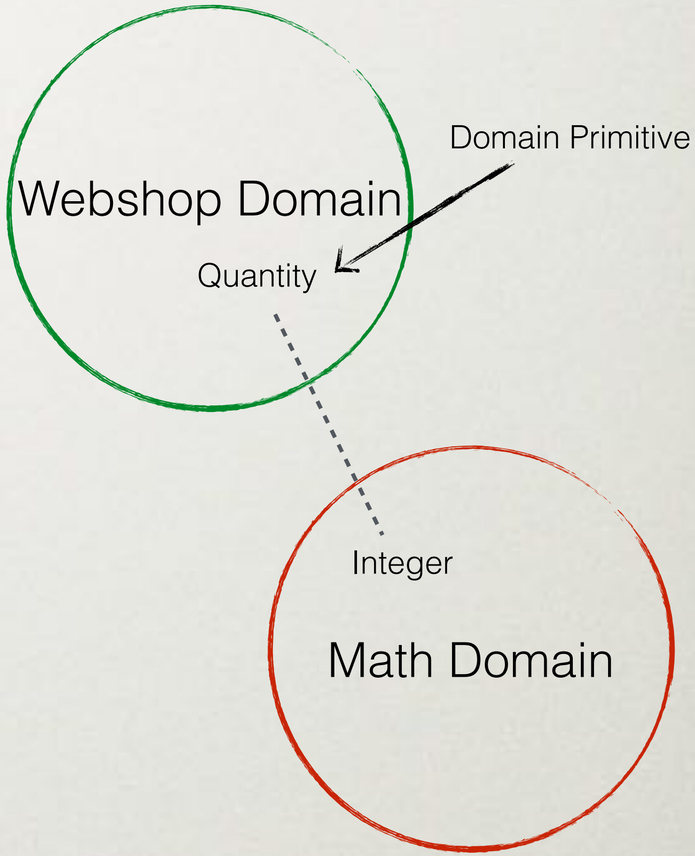
Math and Hotel are different Contexts





Math and Webshop are different Contexts





Quantity as a Domain Primitive

```
public final class Quantity {
    private final int value;

    public Quantity(final int value) {
        inclusiveBetween(1, 99, value);

        this.value = value;
    }

    //Domain specific quantity operations...
}
```

Room Number as a Domain Primitive

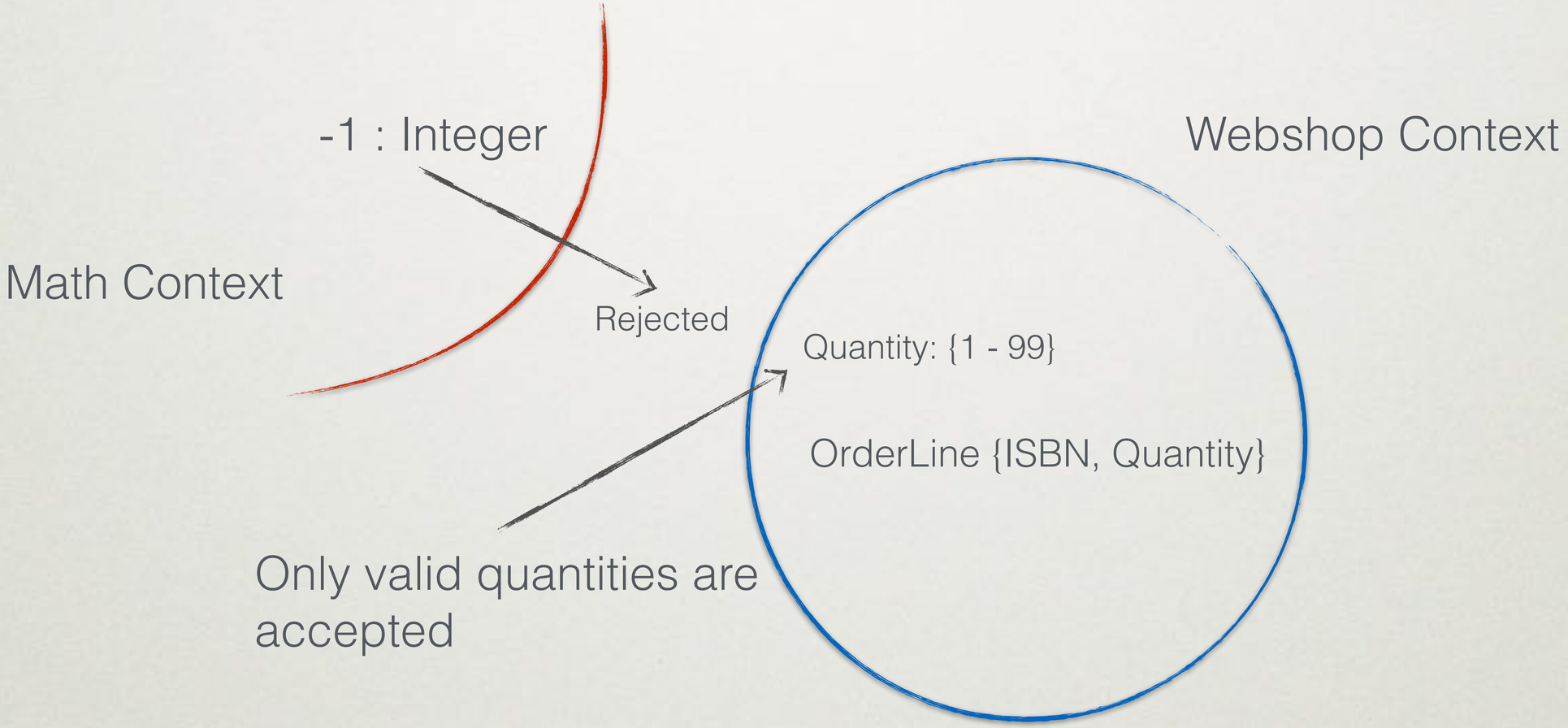
```
public final class RoomNumber {
    private final int value;

    public RoomNumber(final int value) {
        isTrue(Floor.isValid(value));
        inclusiveBetween(1, 50, value % 100);
        this.value = value;
    }

    public Floor floor() {
        return new Floor(value);
    }

    // other logic …
}
```

Invalid quantities are rejected by Definition!



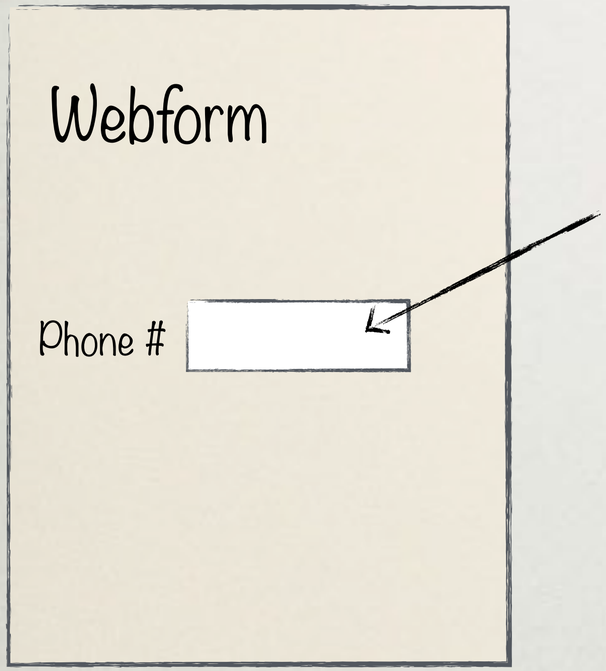
Domain Primitives tighten your design

- Domain Primitives tighten your design by explicitly stating requirements and assumptions.

- They also make it harder to inject data that doesn't meet the expectations.

- Let's see if this pattern allows us to address XSS attacks implicitly.

We want to prevent invalid phone numbers...



Input:

+46 8 545 106 90

or

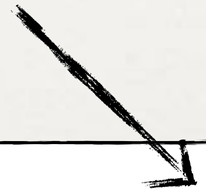
<script>alert("XSS")</script>


```
public void register(final String phoneNumber) {
    // some phone number logic
}
```

But a String Accepts Anything...

Use a Domain Primitive Instead

Only valid phone numbers in your domain



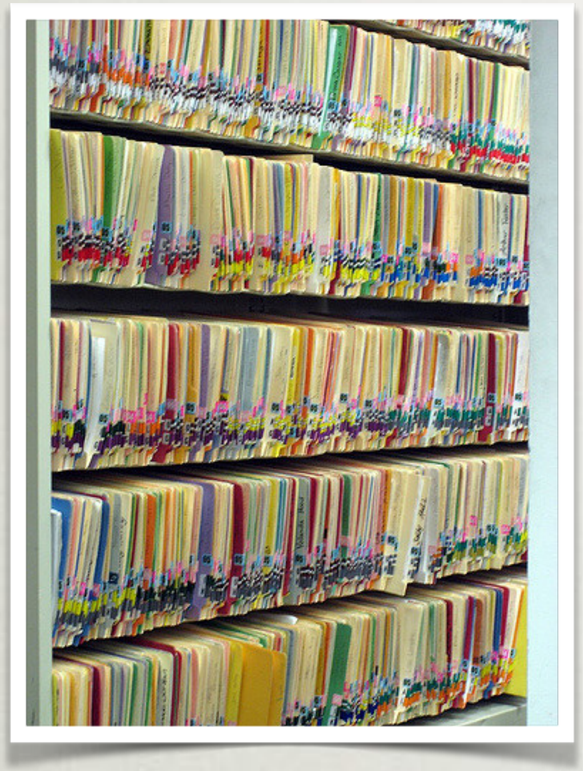
```
public void register(final PhoneNumber phoneNumber) {
    // some phone number logic
}
```

Domain Primitives "prevent" XSS

- The `PhoneNumber` domain primitive enforce domain rule validation at creation time.
- This reduces the attack vector to data that meets the rules in the context where it's used.
- `<script>alert("XSS")</script>` doesn't meet the rules and is rejected *by design*.
- But what about escaping – do we still need it?

But...

… it becomes a lot of classes!



https://flic.kr/p/2pvb2T
https://creativecommons.org/licenses/by/2.0/

… isn't it overly complex?

https://flic.kr/p/7Ro4HU
https://creativecommons.org/licenses/by/2.0/

… what about performance?

https://flic.kr/p/eGYhMw
https://creativecommons.org/licenses/by/2.0/

Secure by Design

"A mindset and strategy for creating secure software by focusing on good design"

- Secure by Design

OWASP Top 10

Top 10 2017
A1 - Injection
A2 - Broken Authentication
A3 - Sensitive Data Exposure
A4 - XML External Entities (XXE)
A5 - Broken Access Control
A6 - Security Misconfiguration
A7 - Cross-Site Scripting (XSS)
A8 - Insecure Deserialization
A9 - Using Components with Known Vulnerabilities
A10 - Insufficient Logging&Monitoring

Top 10 2013
A1 - Injection
A2 - Broken Authentication and Session Management
A3 - Cross-Site Scripting (XSS)
A4 - Insecure Direct Object References
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross-Site Request Forgery (CSRF)
A9 - Using Components with Known Vulnerabilities
A10 - Unvalidated Redirects and Forwards

Top 10 2010
A1 - Cross Site Scripting (XSS)
A2 - Injection Flaws
A3 - Malicious File Execution
A4 - Insecure Direct Object Reference
A5 - Cross Site Request Forgery (CSRF)
A6 - Information Leakage and Improper Error Handling
A7 - Broken Authentication and Session Management
A8 - Insecure Cryptographic Storage
A9 - Insecure Communications
A10 - Failure to Restrict URL Access

Top 10 2004
A1 - Unvalidated Input
A2 - Broken Access Control
A3 - Broken Authentication and Session Management
A4 - Cross Site Scripting
A5 - Buffer Overflow
A6 - Injection Flaws
A7 - Improper Error Handling
A8 - Insecure Storage
A9 - Application Denial of Service
A10 - Insecure Configuration Management

Sensitive Data

- What is sensitive data?
- Where and how sensitive data is consumed should be defined explicitly
- But how can we detect that sensitive data has been illegally exposed?

Read Once Pattern

```
public final class SensitiveValue implements Externalizable {
   private transient final AtomicReference<String> value;

   public SensitiveValue(final String value) {
      this.value = new AtomicReference<>(validate(value));
   }

   public String value() {
      return notNull(value.getAndSet(null), "Sensitive value has already been consumed");
   }

   private static String validate(final String value) {
      // Check domain-specific invariants
      return notBlank(value).trim();
   }

   @Override
   public String toString() {
      return "SensitiveValue{value=*****}";
   }

   @Override
   public void writeExternal(final ObjectOutput out) { deny(); }
   @Override
   public void readExternal(final ObjectInput in) { deny(); }

   private static void deny() { throw new UnsupportedOperationException("Not allowed"); }
}
```

OWASP Top 10

Top 10 2017
A1 - Injection
A2 - Broken Authentication
A3 - Sensitive Data Exposure
A4 - XML External Entities (XXE)
A5 - Broken Access Control
A6 - Security Misconfiguration
A7 - Cross-Site Scripting (XSS)
A8 - Insecure Deserialization
A9 - Using Components with Known Vulnerabilities
A10 - Insufficient Logging&Monitoring

Top 10 2013
A1 - Injection
A2 - Broken Authentication and Session Management
A3 - Cross-Site Scripting (XSS)
A4 - Insecure Direct Object References
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross-Site Request Forgery (CSRF)
A9 - Using Components with Known Vulnerabilities
A10 - Unvalidated Redirects and Forwards

Top 10 2010
A1 - Cross Site Scripting (XSS)
A2 - Injection Flaws
A3 - Malicious File Execution
A4 - Insecure Direct Object Reference
A5 - Cross Site Request Forgery (CSRF)
A6 - Information Leakage and Improper Error Handling
A7 - Broken Authentication and Session Management
A8 - Insecure Cryptographic Storage
A9 - Insecure Communications
A10 - Failure to Restrict URL Access

Top 10 2004
A1 - Unvalidated Input
A2 - Broken Access Control
A3 - Broken Authentication and Session Management
A4 - Cross Site Scripting
A5 - Buffer Overflow
A6 - Injection Flaws
A7 - Improper Error Handling
A8 - Insecure Storage
A9 - Application Denial of Service
A10 - Insecure Configuration Management

XML and a Billion Laughs

omega point.

Internal XML Entity

- Powerful constructs that allow simple abbreviations in XML

- Defined in the Document Type Definition (DTD) and written in the form

```
<!ENTITY name "value">
```

Internal XML Entity

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<!DOCTYPE example [
<!ELEMENT example (#PCDATA)>
<!ENTITY title "Secure by Design">
]>
<example>&title;</example>
```

Internal XML Entity

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<!DOCTYPE example [
<!ELEMENT example (#PCDATA)>
<!ENTITY title "Secure by Design">
]>
<example>Secure by Design</example>
```

A Billion Laughs (XXE Attack)

- Billion Laughs attack is as simple as it is effective

- The main idea is to **exploit** the **expandability** property of XML entities by defining **recursive** definitions that expand into a huge memory footprint

A Billion Laughs

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE lolz [
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol "lol">
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

A Billion Laughs

```
lol9
```

A Billion Laughs

```
lol8lol8lol8lol8lol8lol8lol8lol8lol8lol8
```

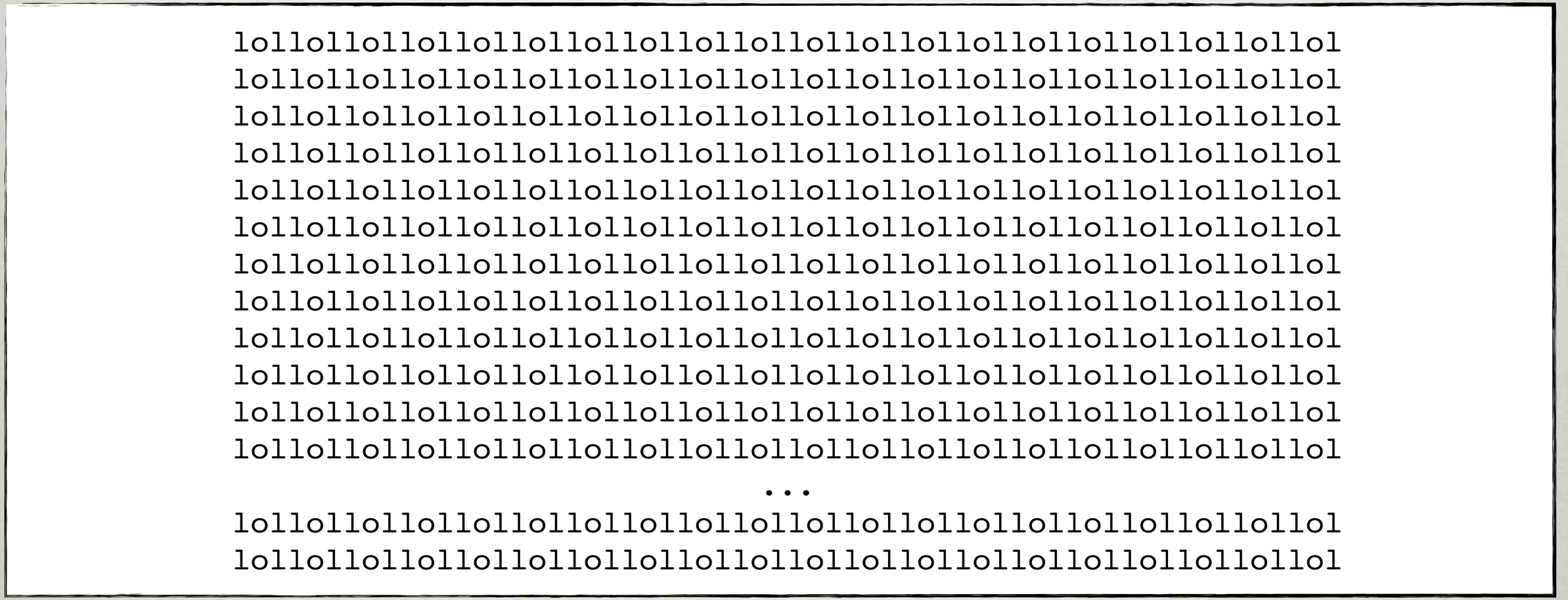
A Billion Laughs

lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7
lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7
lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7
lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7
lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7lol7

A Billion Laughs

After a few more expansions…

A Billion Laughs



Analysis - A Billion Laughs

- How's this possible?
- Where does the vulnerability lie?
- How can we protect against XXE attacks?

Configuring the XML Parser

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

factory.setExpandEntityReferences(false);

factory.setFeature(
"http://javax.xml.XMLConstants/feature/secure-processing", true);

factory.setFeature(
"http://apache.org/xml/features/disallow-doctype-decl", true);

factory.setFeature(
"http://xml.org/sax/features/external-general-entities", false);

factory.setFeature(
"http://xml.org/sax/features/external-parameter-entities", false);

factory.setFeature(
"http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
```

Analysis - Relying on Parser Config

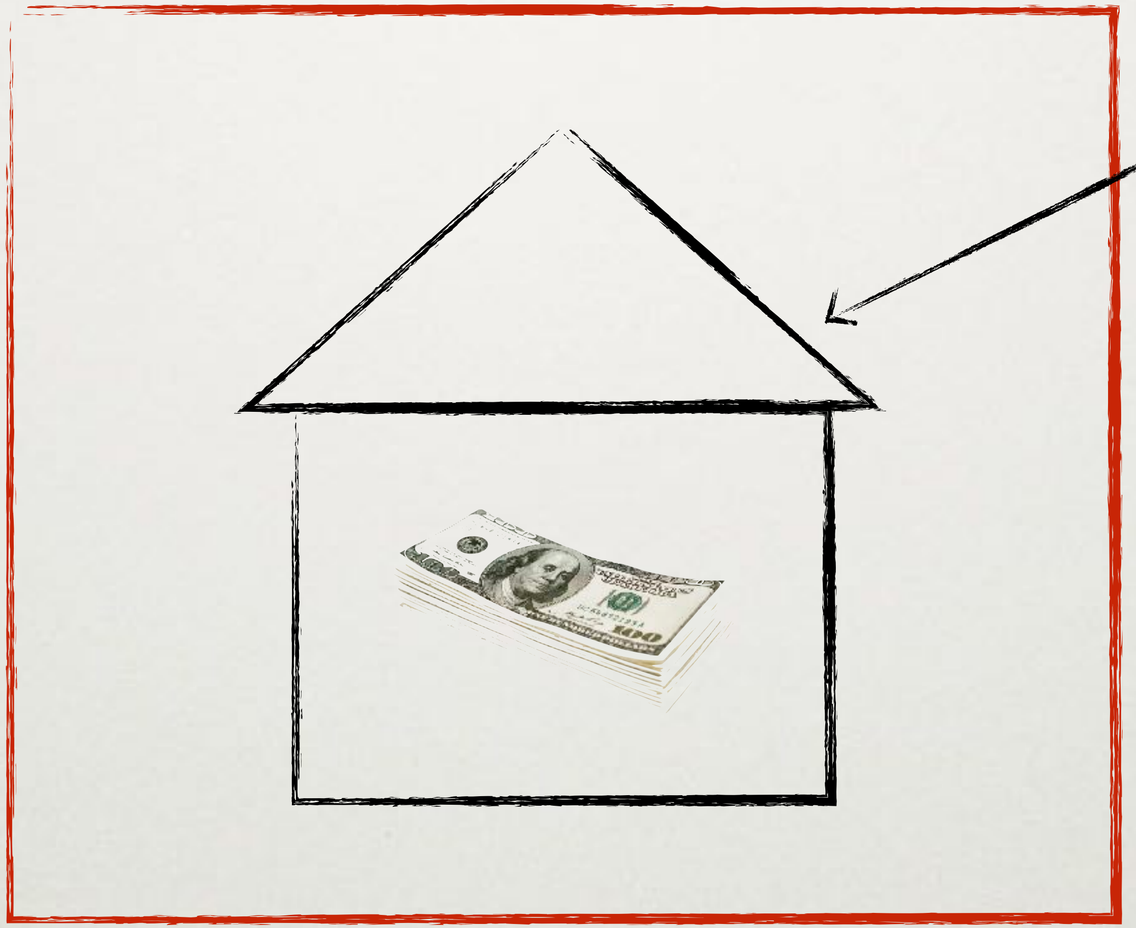
- What happens if we forget a setting?

- Does all parsers behave the same way?
- How do you know which settings to apply?
- Does this protect against XXE attacks?

Security in Depth

- Multiple layers of security
- Not enough to breach one layer
- Makes it really hard to hack a system

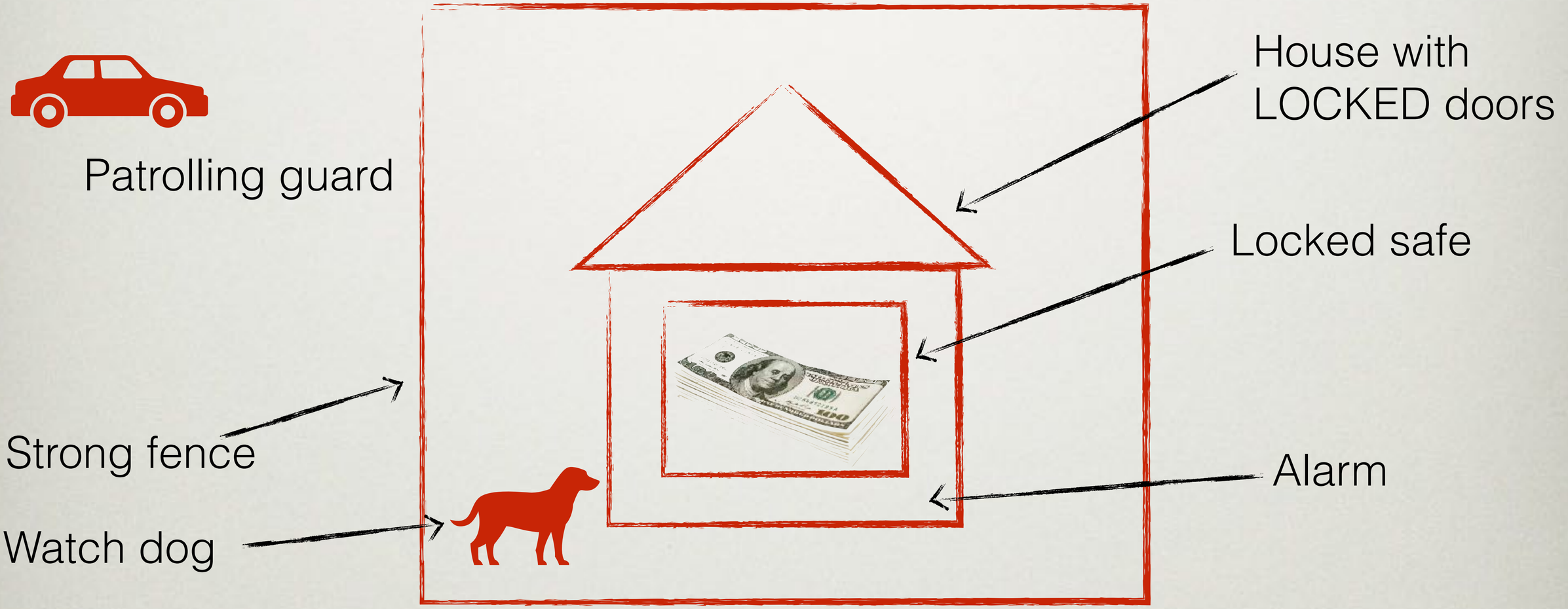
Security in Depth



House without locks

Strong fence

Security in Depth



Customer XML

- Assume we want to exchange Customer data between services

```
<customer>
      <phone>212-111-2222</phone>
      <email>jane.doe@example.com</email>
      <address>
         <street>Fifth Ave</street>
         <city>New York</city>
         <country>USA</country>
      </address>
</customer>
```

Customer XML

- **Before parsing** the XML, we should verify that it only contains valid elements

- This can be done using a **lexical scanner**

Lexical Scanner

```
public class LexicalScanner {
   public static boolean isValid(final InputStream data) {
      notNull(data);

      try {
         final SAXParser saxParser = SAXParserFactory.newInstance().newSAXParser();
         final ElementHandler handler = new ElementHandler();
         saxParser.getXMLReader()
                  .setProperty("http://xml.org/sax/properties/lexical-handler", handler);
         saxParser.parse(data, handler);
         return true;
      } catch (Exception e) {
         return false;
      }
   }
}
```

Lexical Scanner

```
public class ElementHandler extends org.xml.sax.ext.DefaultHandler2 {

   private static final Set<String> VALID_ELEMENTS;

   static {
      VALID_ELEMENTS = new HashSet<>();
      VALID_ELEMENTS.add("customer");
      VALID_ELEMENTS.add("email");
      VALID_ELEMENTS.add("phone");
      VALID_ELEMENTS.add("address");
      VALID_ELEMENTS.add("street");
      VALID_ELEMENTS.add("city");
      VALID_ELEMENTS.add("country");
   }

   @Override
   public void startElement(final String uri, final String localName,
                            final String qName, final Attributes attributes) {
      notNull(qName);
      inclusiveBetween(1, 100, qName.length(),
                       format("Invalid element length. Got: %s characters", qName.length()));
      isTrue(VALID_ELEMENTS.contains(qName.toLowerCase()), "Invalid element found.");
   }

   @Override
   public void startEntity(final String name) {
      throw new IllegalArgumentException("Entities are illegal");
   }
}
```

Customer

```
public final class Customer {
    private final Address address;
    private final Email email;
    private final PhoneNumber phoneNumber;
    private final Street street;
    private final City city;
    private final Country country;

    public Customer(final InputStream data) {
        notNull(data);

        isTrue(LexicalScanner.isValid(data));

        // parse XML

        // assign fields
    }

    // other domain logic
}
```

Interesting Observation

- By explicitly whitelisting valid elements in the Customer XML, we reject any XML that doesn't meet the domain requirements

- By explicitly rejecting entities, XXE attacks are avoided

- But you should still configure the parser accordingly (remember: Security in Depth)

Order of Validation

1. **Origin** – Is the data from a legitimate sender?
2. **Size** – Is it reasonably big?
3. **Lexical content** – Does it contain the right characters and encoding?
4. **Syntax** – Is the format right?
5. **Semantics** – Does the data make sense?

Attr: Dr. John Wilander

OWASP Top 10

Top 10 2017
A1 - Injection
A2 - Broken Authentication
A3 - Sensitive Data Exposure
A4 - XML External Entities (XXE)
A5 - Broken Access Control
A6 - Security Misconfiguration
A7 - Cross-Site Scripting (XSS)
A8 - Insecure Deserialization
A9 - Using Components with Known Vulnerabilities
A10 - Insufficient Logging&Monitoring

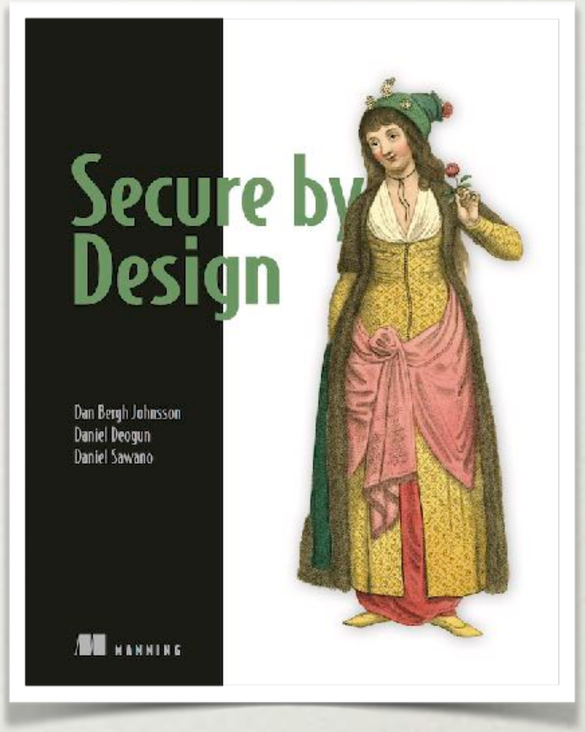
Top 10 2013
A1 - Injection
A2 - Broken Authentication and Session Management
A3 - Cross-Site Scripting (XSS)
A4 - Insecure Direct Object References
A5 - Security Misconfiguration
A6 - Sensitive Data Exposure
A7 - Missing Function Level Access Control
A8 - Cross-Site Request Forgery (CSRF)
A9 - Using Components with Known Vulnerabilities
A10 - Unvalidated Redirects and Forwards

Top 10 2010
A1 - Cross Site Scripting (XSS)
A2 - Injection Flaws
A3 - Malicious File Execution
A4 - Insecure Direct Object Reference
A5 - Cross Site Request Forgery (CSRF)
A6 - Information Leakage and Improper Error Handling
A7 - Broken Authentication and Session Management
A8 - Insecure Cryptographic Storage
A9 - Insecure Communications
A10 - Failure to Restrict URL Access

Top 10 2004
A1 - Unvalidated Input
A2 - Broken Access Control
A3 - Broken Authentication and Session Management
A4 - Cross Site Scripting
A5 - Buffer Overflow
A6 - Injection Flaws
A7 - Improper Error Handling
A8 - Insecure Storage
A9 - Application Denial of Service
A10 - Insecure Configuration Management

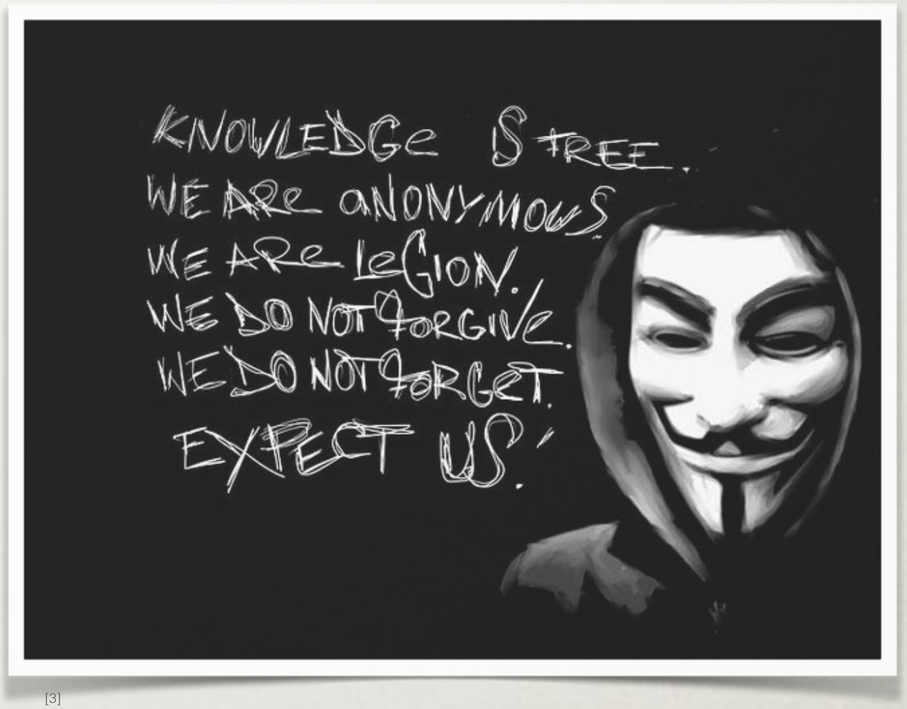
Agenda

- **A Bank Robbery**
- **Secure by Design**
- **Technical solution to XSS**
- **Domain Primitives**
- **Sensitive Data Exposure**
- **XML and a Billion Laughs**
- Automatic (security) unit tests
- Domain Primitives Applied on Legacy
- Fighting Entity Complexity
- Cloud Thinking



Automated (Security) Unit Tests

- Normal Behavior
- Boundary Behavior
- Invalid Input Behavior
- Extreme Input Behavior



[3]

The Hospital Case

[4]

Email Address Domain Rules

[5]

- The format of an email address must be *local-part@domain*
- The local part cannot be longer than 64 characters
- The domain must be *hospital.com*
- Subdomains are not accepted
- The minimum length of an email address is 15 characters
- The maximum length of an email address is 77 characters
- The local part may only contain alphabetic characters (a-z), digits (0-9), and one period
- The local part may not start or end by a period

Testing normal behavior

- Focus on input that clearly meets the domain rules

```
class EmailAddressTest {
   @TestFactory
   Stream<DynamicTest> should_be_a_valid_address() {
      return Stream.of(
            "jane@hospital.com",
            "jane01@hospital.com",
            "jane.doe@hospital.com"
      ).map(input -> dynamicTest("Accepted: " + input,
                                 () -> new EmailAddress(input)));
   }
}
```

1st version of EmailAddress

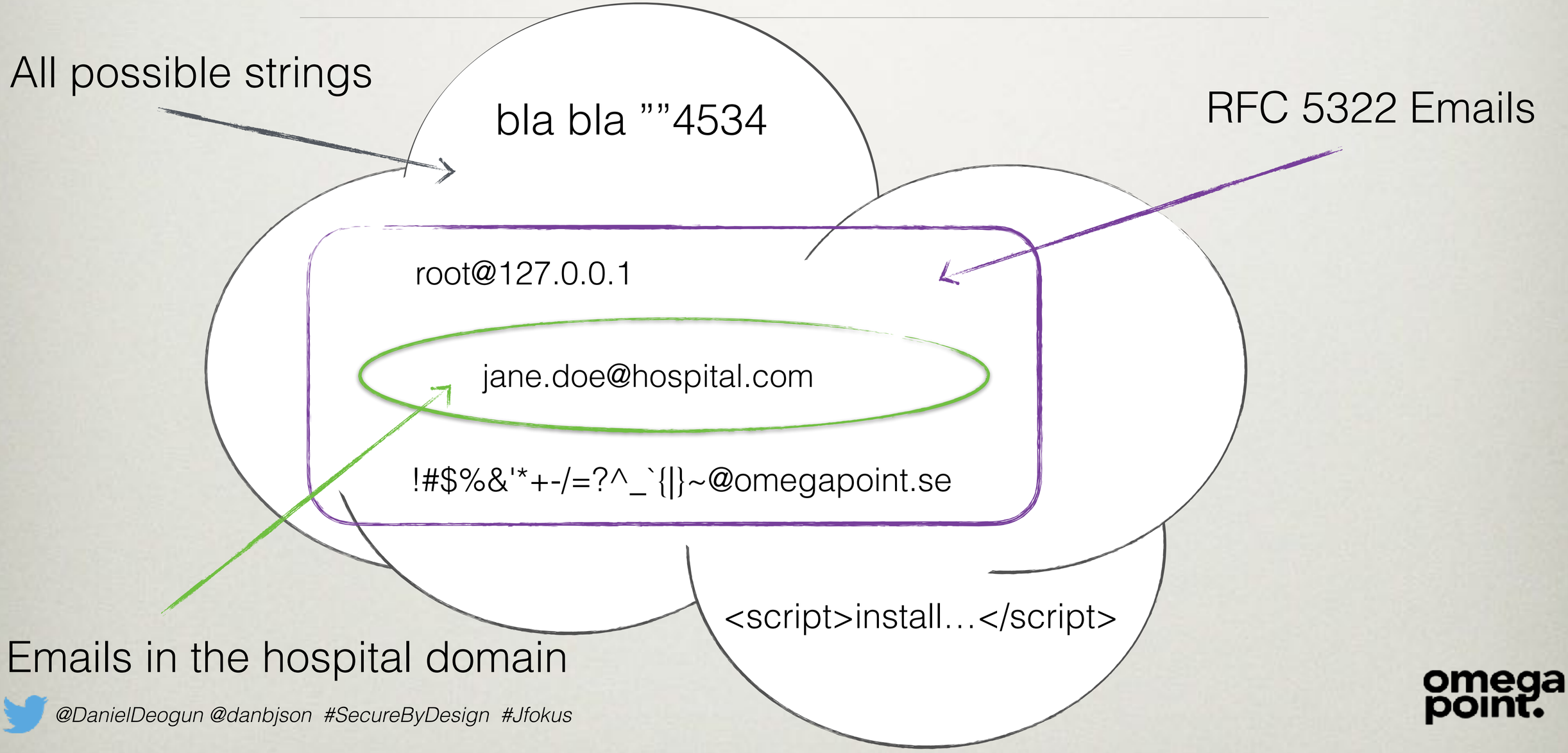
```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      matchesPattern(value.toLowerCase(),
                     "^[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }
}
```

Email is defined by the hospital domain



omega point.

Testing boundary behavior - Acceptance

- Accept an address that's exactly 15 characters long.
- Accept an address with a local part that's 64 characters long.
- Accept an address that's exactly 77 characters long.

```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      matchesPattern(value.toLowerCase(),
         "^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }
}
```

Testing boundary behavior - Rejection

- Reject an address that's 14 characters long
- Reject an address with a local part that's 65 characters long
- Reject an address with a local part containing an invalid character
- Reject an address with multiple '@' symbols
- Reject an address with a domain other than *hospital.com*
- Reject an address with a subdomain
- Reject an address with a local part that starts with a period
- Reject an address with a local part that ends with a period
- Reject an address with sequential periods in the local part

Testing boundary behavior - Rejection

```
@TestFactory
Stream<DynamicTest> should_be_rejected() {
   return Stream.of(
         "a@hospital.com",
         repeat("X", 65) + "@hospital.com",
         "address_with_invalid_char_in_local_part@hospital.com",
         "jane@doe@hospital.com",
         "jane.doe@hospital.se",
         "jane.doe@hospital.io",
         "jane.doe@hospital.gov",
         "jane.doe@example.com",
         “jane.doe@cardio.hospital.com",
         ".jane.doe@hospital.com",
         "jane.doe.@hospital.com",
         "jane..doe@hospital.com")
               .map(input -> dynamicTest("Rejected: " + input,
                                         assertInvalidEmail(input)));
}
```

2nd version of EmailAddress

```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      matchesPattern(value.toLowerCase(),
         "^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }

}
```

Testing with invalid input

- Any input that doesn't satisfy the domain rules is considered invalid

- For some reason, `null`, empty strings, or "strange" characters tend to result in unexpected behavior

Testing with invalid input

```
@TestFactory
Stream<DynamicTest> should_reject_invalid_input() {
    return Stream.of(
            null,
            "null",
            "nil",
            "0",
            "",
            " ",
            "  ",
            "\t",
            "\n",
            "john.doe\n@hospital.com",
            "   @hospital.com",
            "%20@hospital.com",
            "john.d%20e@hospital.com",
            "john.doe.jane@hospital.com",
            "--",
            "e x a m p l e @ hospital . c o m",
            "=0@$*^%;<!->.:\\()&#\"",
            "©@£$∞§|[]≈±´•Ωé®†µüıœπ¨~ßðfˌ¸√ªfiøæ™≤÷≈ç‹›‘’,…")
                    .map(input -> dynamicTest("Rejected: " + input,
                                              assertInvalidEmail(input)));
}
```

3rd version of EmailAddress

```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      notNull(value, "Input cannot be null");
      matchesPattern(value.toLowerCase(),
         "^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }

}
```

Testing with invalid input - identifying eventual exploits

- Identifying input that eventually exploits a weakness in the system is especially interesting because it's the underlying basis of 2nd order injection attacks

Testing with invalid input - simple SQL injection

- As it turns out, the email address is used in SQL queries

```
@TestFactory
Stream<DynamicTest> should_reject_SQL() {
   return Stream.of(
         "'or%20select *",
         "admin'--",
         "<>\"'%;)(&+",
         "'%20or%20''='",
         "'%20or%20'x'='x",
         "\"%20or%20\"x\"=\"x",
         "')%20or%20('x'='x",
         "0 or 1=1",
         "' or 0=0 --",
         "\" or 0=0 --")
         .map(input -> dynamicTest("Rejected: " + input,
                        assertInvalidEmail(input)));
}
```

Testing the extreme
- with the purpose of breaking things

- Testing the extreme is all about identifying weaknesses in the design that makes the application break or behave strangely when handling extreme values.

Testing the extreme
- with the purpose of breaking things

- This clearly doesn't meet the domain rules

```
@TestFactory
Stream<DynamicTest> should_reject_extreme_input() {
    return Stream.<Supplier<String>>of(
            () -> repeat("X", 10000),
            () -> repeat("X", 100000),
            () -> repeat("X", 1000000),
            () -> repeat("X", 10000000),
            () -> repeat("X", 20000000),
            () -> repeat("X", 40000000))
            .map(input -> dynamicTest("Rejecting extreme input",
                                      assertInvalidEmail(input.get())));
}
```

Final version of EmailAddress

```
public final class EmailAddress {

   public final String value;

   public EmailAddress(final String value) {
      notNull(value, "Input cannot be null");
      matchesPattern(value.toLowerCase(),
         "^(?=[a-z0-9.@]{15,77}$)[a-z0-9]+\\.?[a-z0-9]+@\\bhospital.com$");

      this.value = value.toLowerCase();
   }

   …

}
```

Domain Primitives in Action on Legacy

Green Field

https://flic.kr/p/Q7zV https://creativecommons.org/licenses/by-sa/2.0/

vs

Brown Field

https://flic.kr/p/djYc9H https://creativecommons.org/licenses/by/2.0/

Three steps

Draw the line

https://flic.kr/p/nEZKMd

https://creativecommons.org/licenses/by/2.0/

Harden your API

https://flic.kr/p/YgfS6s
https://creativecommons.org/licenses/by/2.0/

Untangle inside

https://flic.kr/p/wdBcT
https://creativecommons.org/licenses/by/2.0/

“Draw the Line”
- Find a Semantic Border

```
public void checkout(final int roomNumber) {
    new RoomNumber(roomNumber); //throws exception if invalid

    houseKeepingService.registerForCleaning(roomNumber);
    minibarService.replenish(roomNumber);
    // other operations ...
}
```

```
public void checkout(final int roomNumber) {
    if (!RoomNumber.isValid(roomNumber)) {
        reporter.logInvalidRoomNumber(roomNumber);
    }

    houseKeepingService.registerForCleaning(roomNumber);
    minibarService.replenish(roomNumber);
    // other operations ...
}
```

“Draw the Line” Demo

Customer Management

`addAddress(String street, String zip, String city)`

`print(String streetline, String cityline)`

Address Importer

```
Nordin Backe 92
66024, Ramstalund
===
Arthurliden 6063
64036, Gemla
===
Småstigen 5309
95160, Rönneshytta
===
```

Envelope Printer

```
Nordin Backe 92
66024, Ramstalund
===
Arthurliden 6063
64036, Gemla
===
Småstigen 5309
95160, Rönneshytta
===
```

“Draw the Line” Memo

- Soft check - observe and log data quality
- Canonical form
- Bad data
- Hard check

https://flic.kr/p/nEZKMd https://creativecommons.org/licenses/by/2.0/

Harden your API
- Enforce Data Quality

Generic `public void checkout(final int roomNumber)`

Enforce data quality

Specific `public void checkout(final RoomNumber roomNumber)`

Harden your API

https://flic.kr/p/YgfS6s
https://creativecommons.org/licenses/by/2.0/

Untangle Inside - Cluttered Entity

```
class Order {
    private ArrayList<Object> items;
    private boolean paid;


    public void addItem(String isbn, int qty) {
        if(this.paid == false) {
            notNull(isbn);
            inclusiveBetween(10, 10, isbn.length());
            isTrue(isbn.matches("[0-9X]*"));
            isTrue(isbn.matches("[0-9]{9}[0-9X]"));

            Book book = bookCatalogue.findByISBN(isbn);

            if (inventory.availableBooks(isbn) >= qty) {
                items.add(new OrderLine(book, qty));
            }
        }
    }
    //Other logic...
}
```

Untangle inside

https://flic.kr/p/wdBcT
https://creativecommons.org/licenses/by/2.0/

De-Cluttered Entity

```
class Order {
    private ArrayList<Object> items;
    private boolean paid;

    public void addItem(ISBN isbn, Quantity qty) {
        notNull(isbn);
        notNull(qty);

        if(this.paid == false) {
            Book book = bookCatalogue.findByISBN(isbn);

            if (inventory.availableBooks(isbn).greaterOrEqualTo(qty)) {
                items.add(new OrderLine(book, qty));
            }
        }
    }
    //Other logic...
}
```

Untangle inside

https://flic.kr/p/wdBcT
https://creativecommons.org/licenses/by/2.0/

Three steps

Draw the line

https://flic.kr/p/nEZKMd

https://creativecommons.org/licenses/by/2.0/

Harden your API

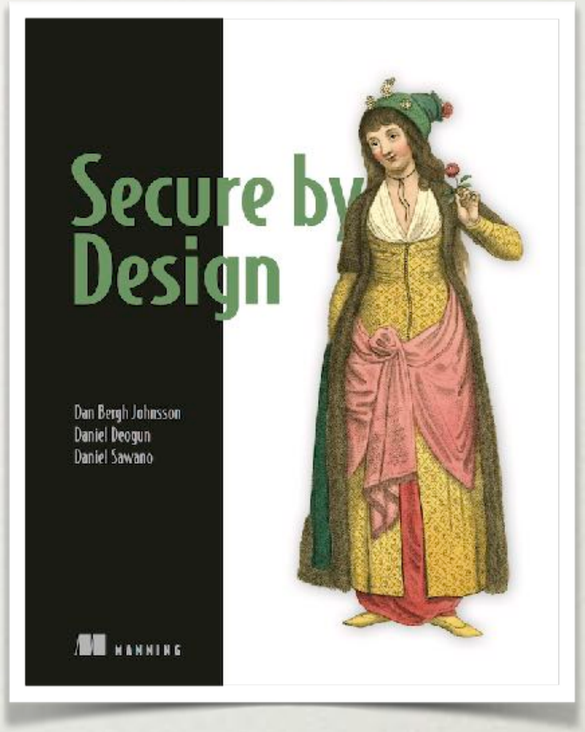
https://flic.kr/p/YgfS6s
https://creativecommons.org/licenses/by/2.0/

Untangle inside

https://flic.kr/p/wdBcT
https://creativecommons.org/licenses/by/2.0/

Agenda

- **A Bank Robbery**
- **Secure by Design**
- **Technical solution to XSS**
- **Domain Primitives**
- **Sensitive Data Exposure**
- **XML and a Billion Laughs**
- **Automatic (security) unit tests**
- **Domain Primitives Applied on Legacy**
- Fighting Entity Complexity
- Cloud Thinking



Fighting Entity Complexity

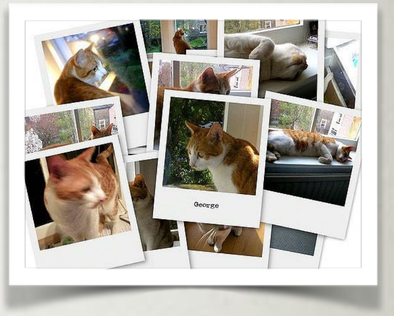
Entity Snapshot

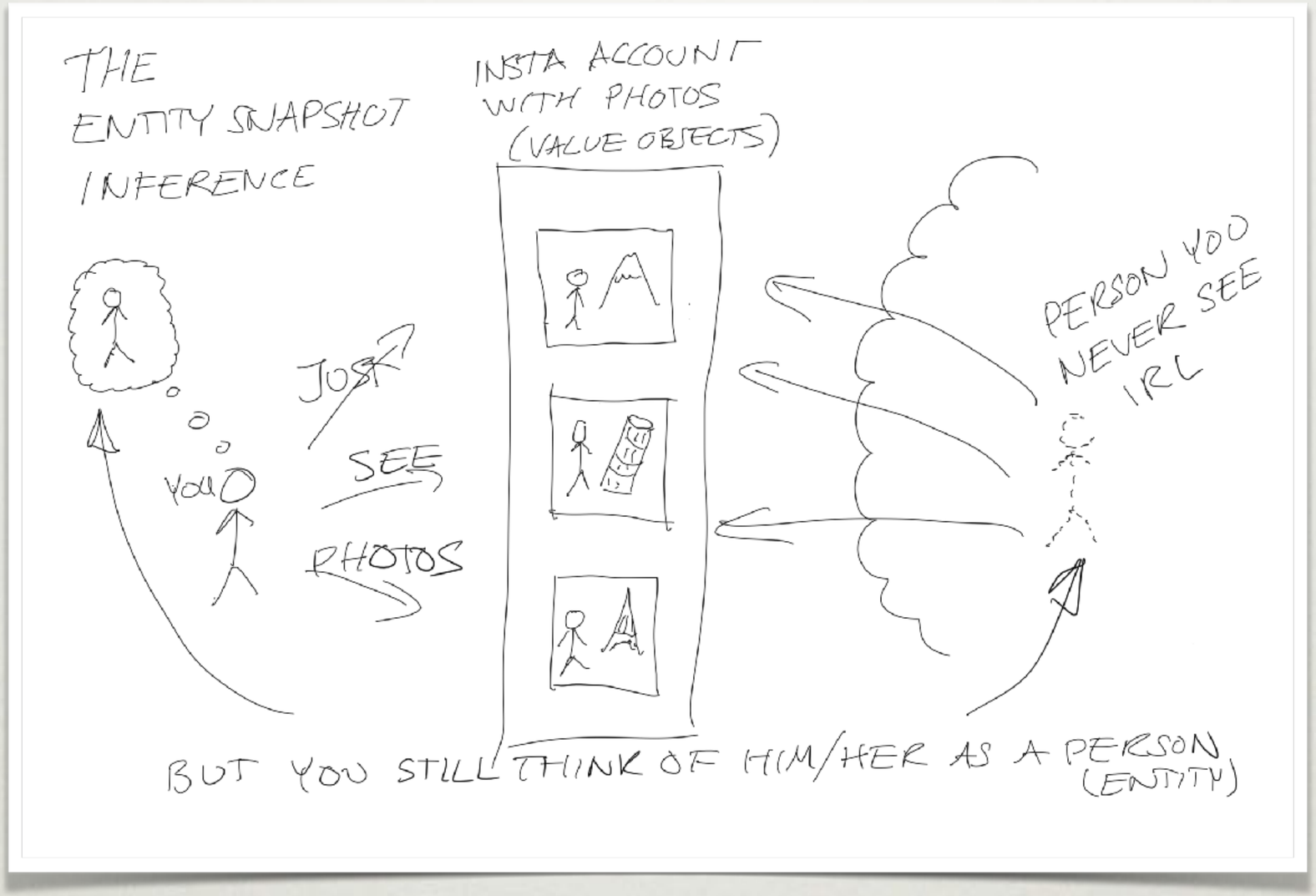
https://flic.kr/p/2TqFF https://creativecommons.org/licenses/by-sa/2.0/

Entity Relay

https://flic.kr/p/einqkQ https://creativecommons.org/licenses/by/2.0/

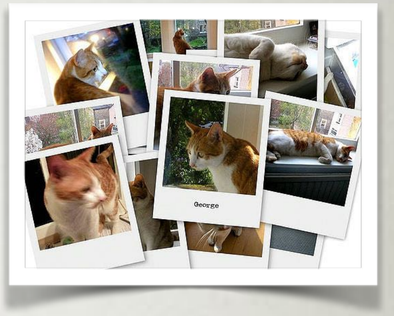
Entity Snapshot

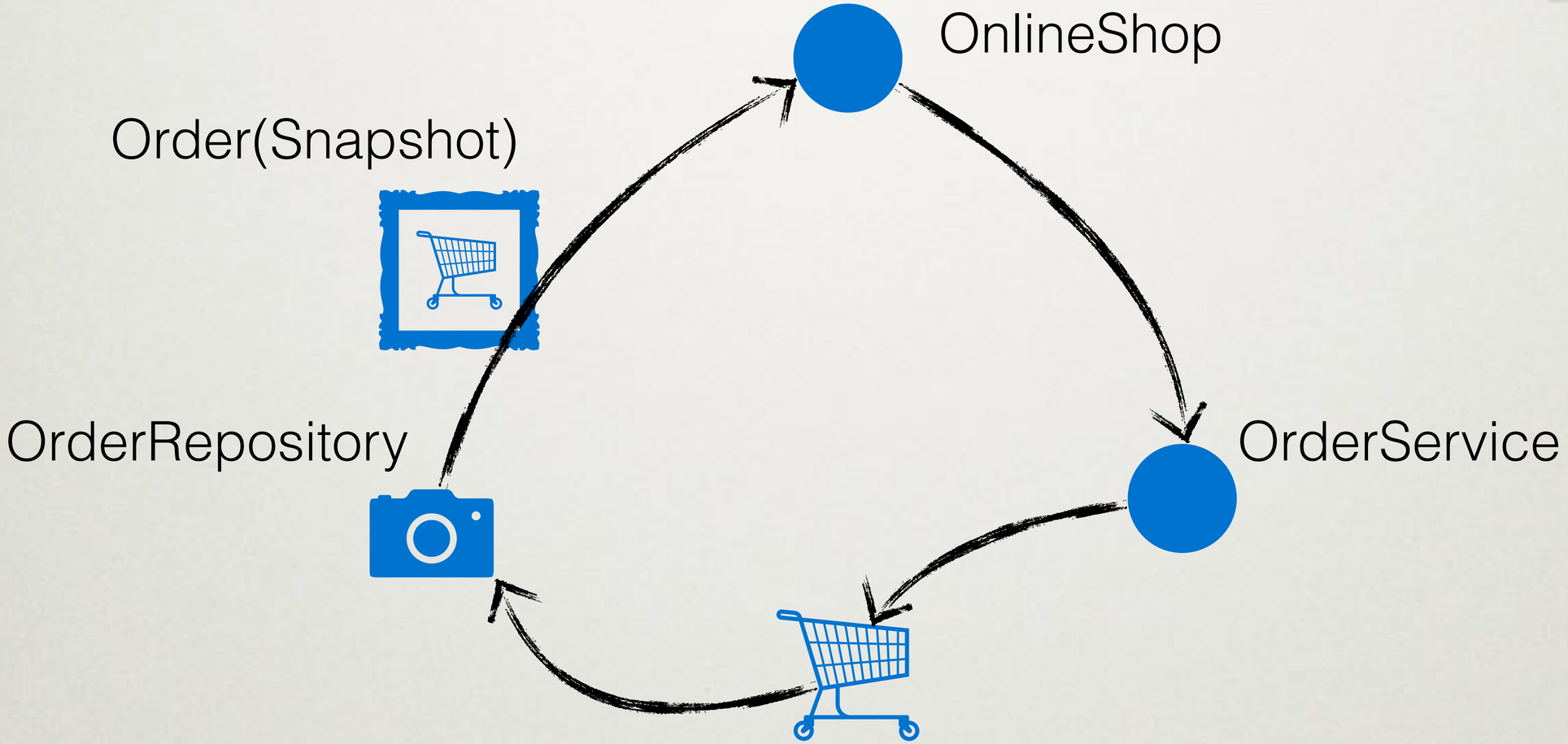




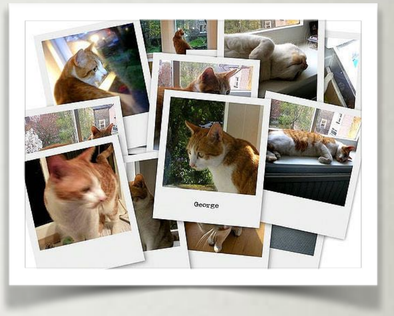
omega point.

Entity Snapshot





Order(Snapshot)




```
public class OrderSnapshot {
    final public OrderID orderid;
    final public CustomerID custid;
    final private List<OrderItem> orderItemList;

    public OrderSnapshot(OrderID orderid,
                         CustomerID custid,
                         List<OrderItem> orderItemList) {
        this.orderid = notNull(orderid);
        this.custid = notNull(custid);
        this.orderItemList =
                Collections.unmodifiableList(notNull(orderItemList);
        checkBusinessRuleInvariants();
    }

    public List<OrderItem> orderItems() {
        return orderItemList;
    }

    public int nrItems() {
        return …
    }

    private void checkBusinessRuleInvariants() {
        validState(nrItems() <= 38, "Too large for ordinary shipping");
    }

}
```

Fighting Entity Complexity

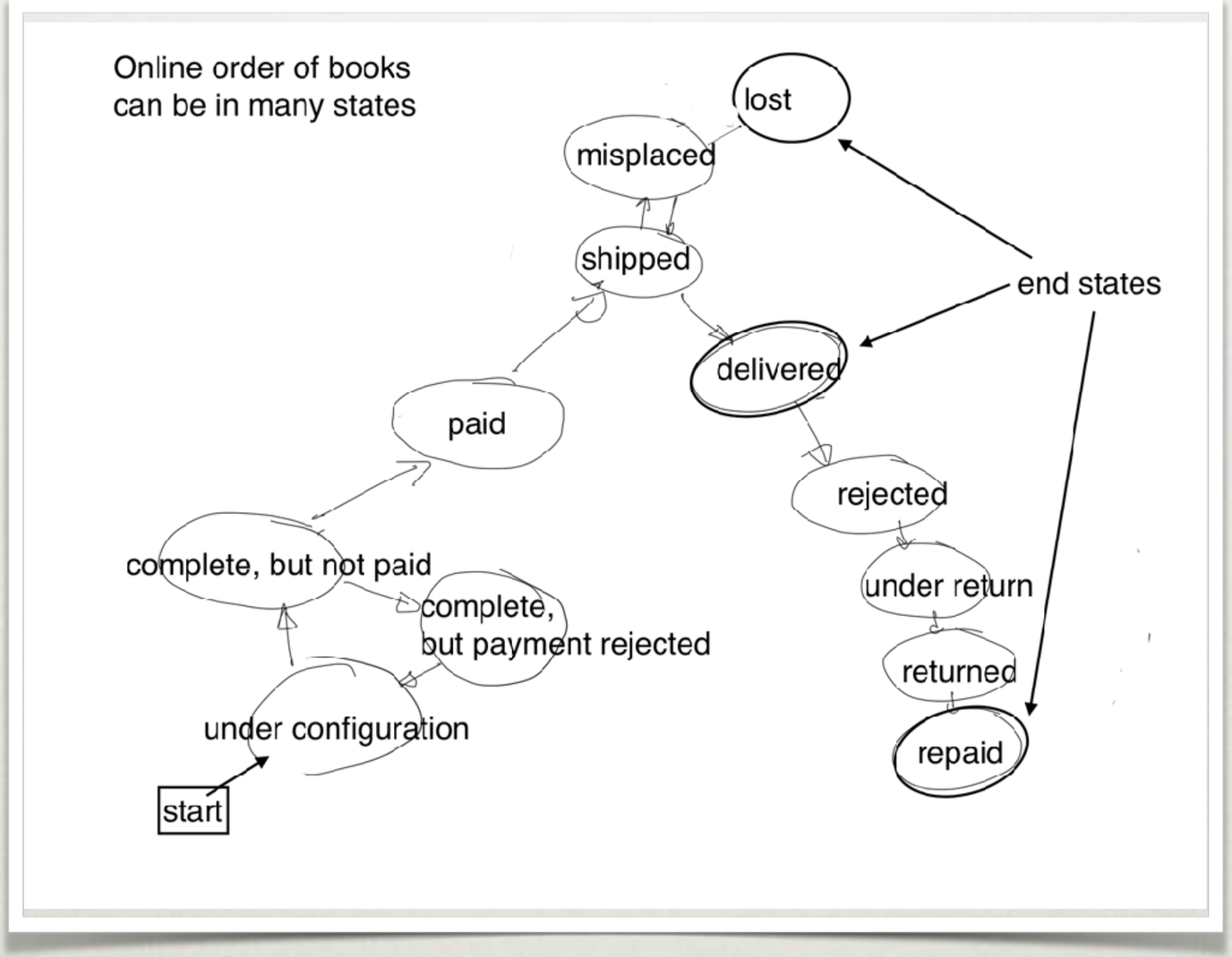
Entity Snapshot

https://flic.kr/p/2TqFF https://creativecommons.org/licenses/by-sa/2.0/

Entity Relay

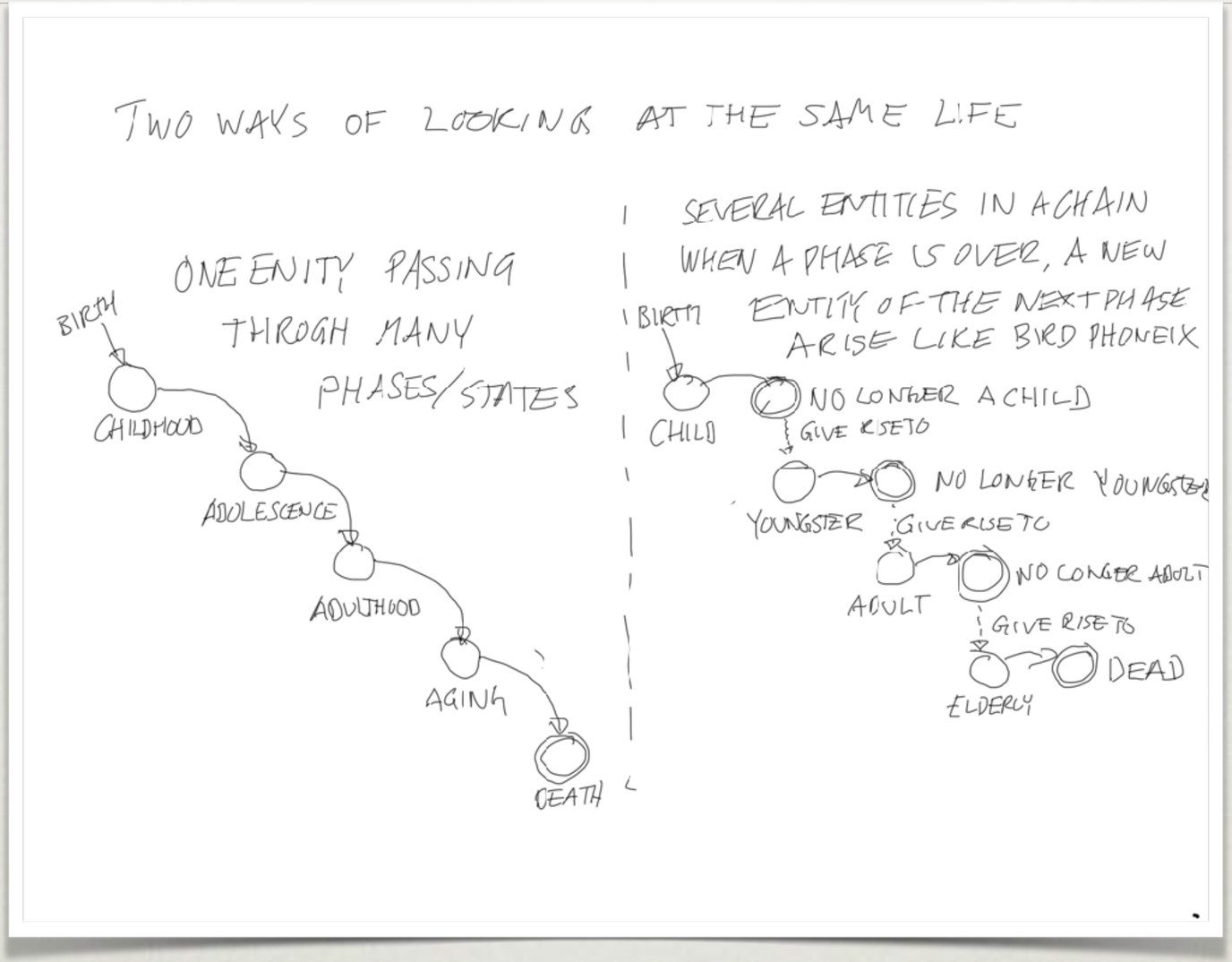
https://flic.kr/p/einqkQ https://creativecommons.org/licenses/by/2.0/

Order States all over



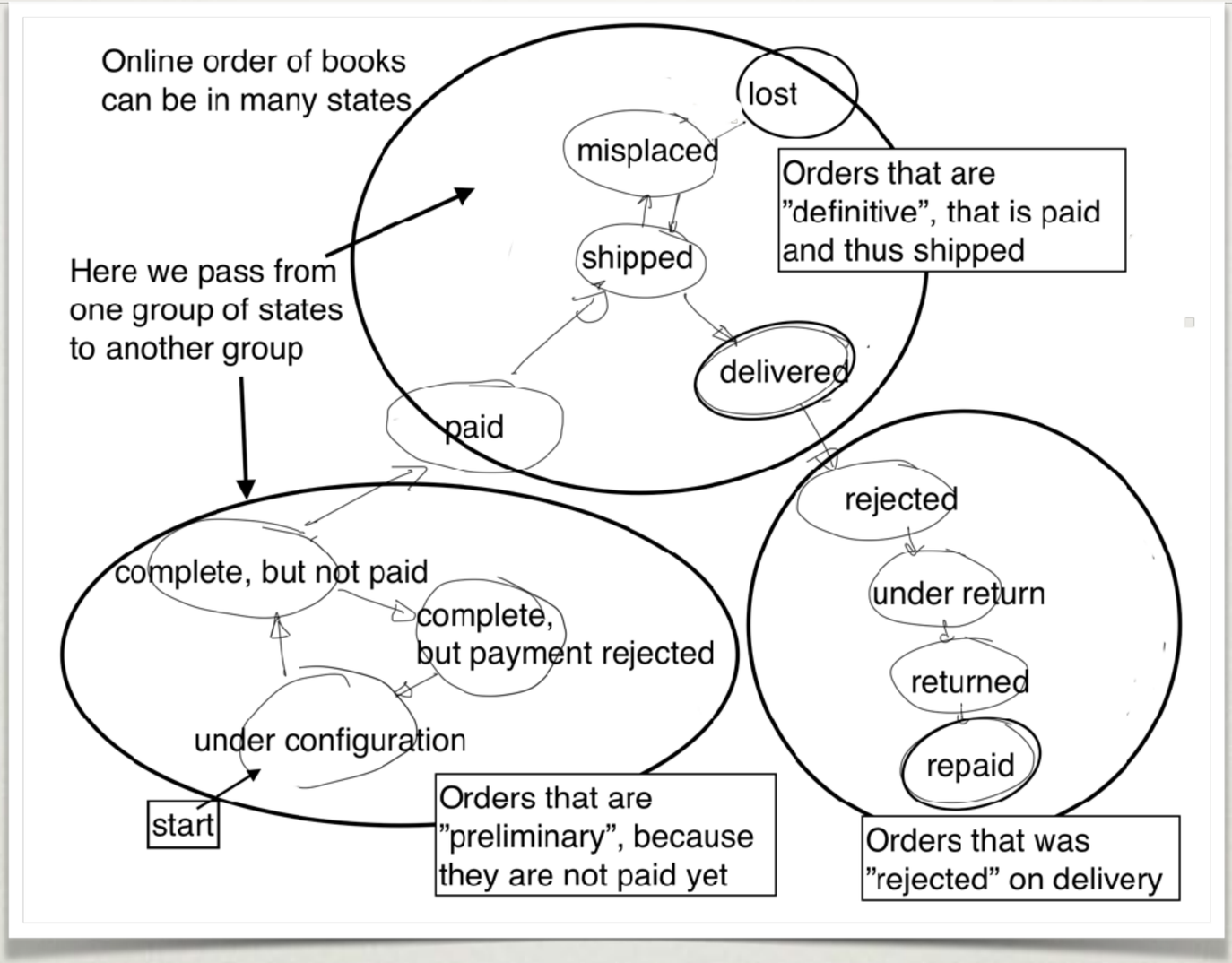
omega point.

Two views of entity States



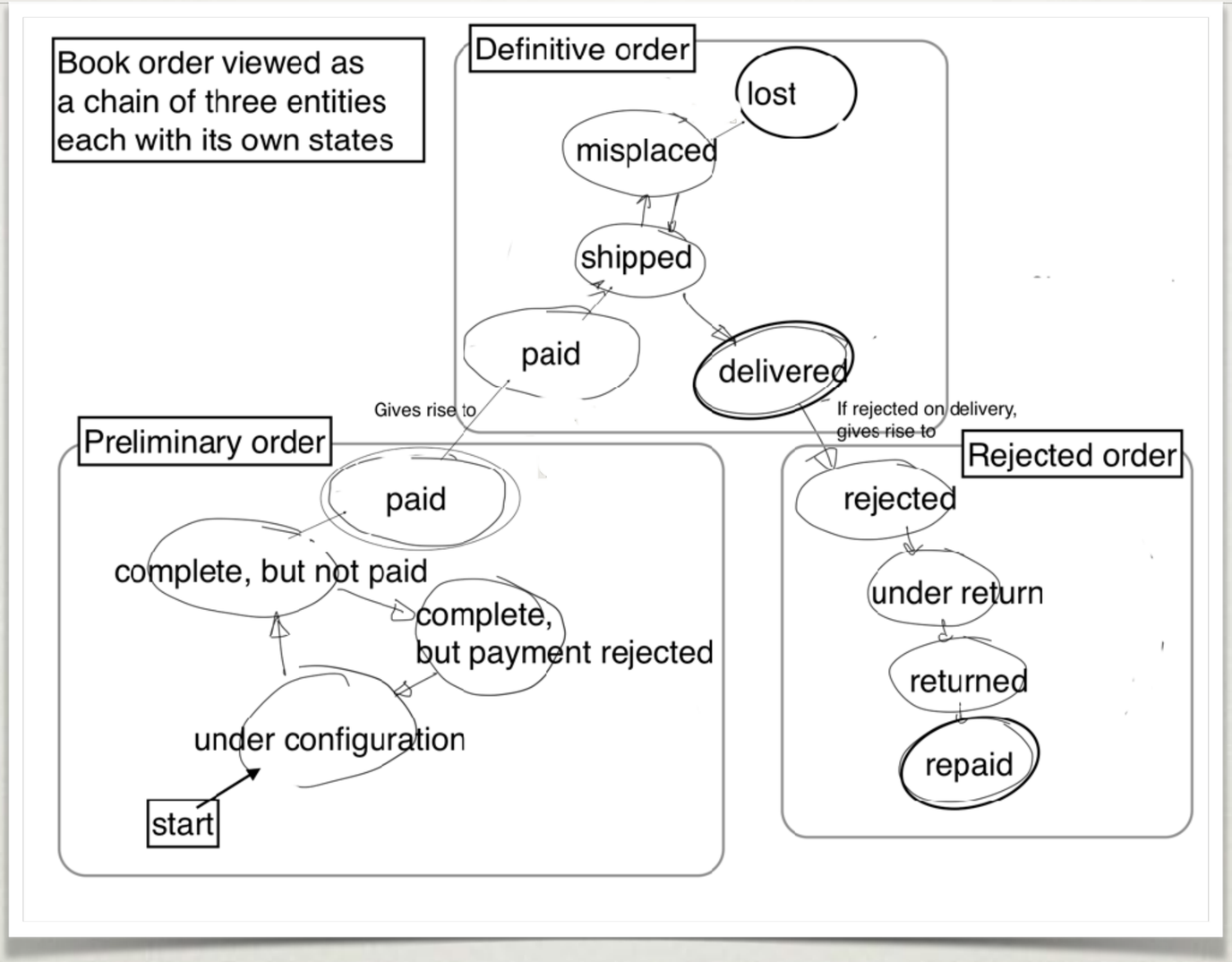
omega point.

Order Phases



omega point.

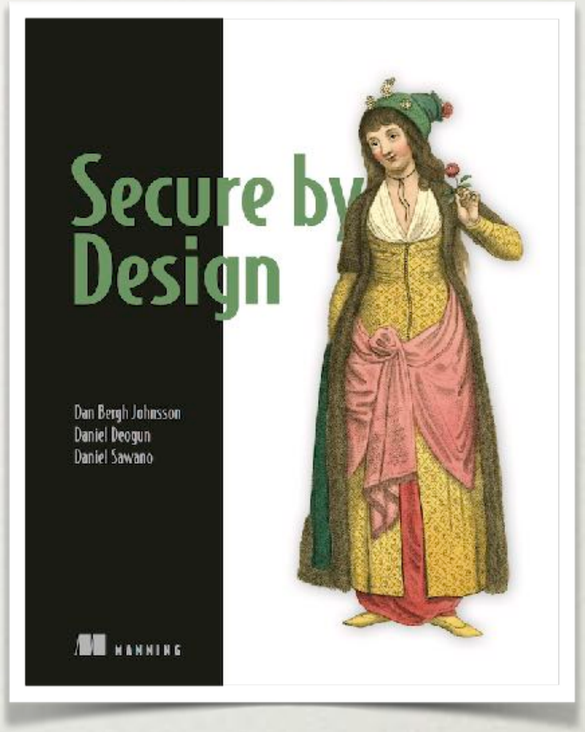
Order Phases



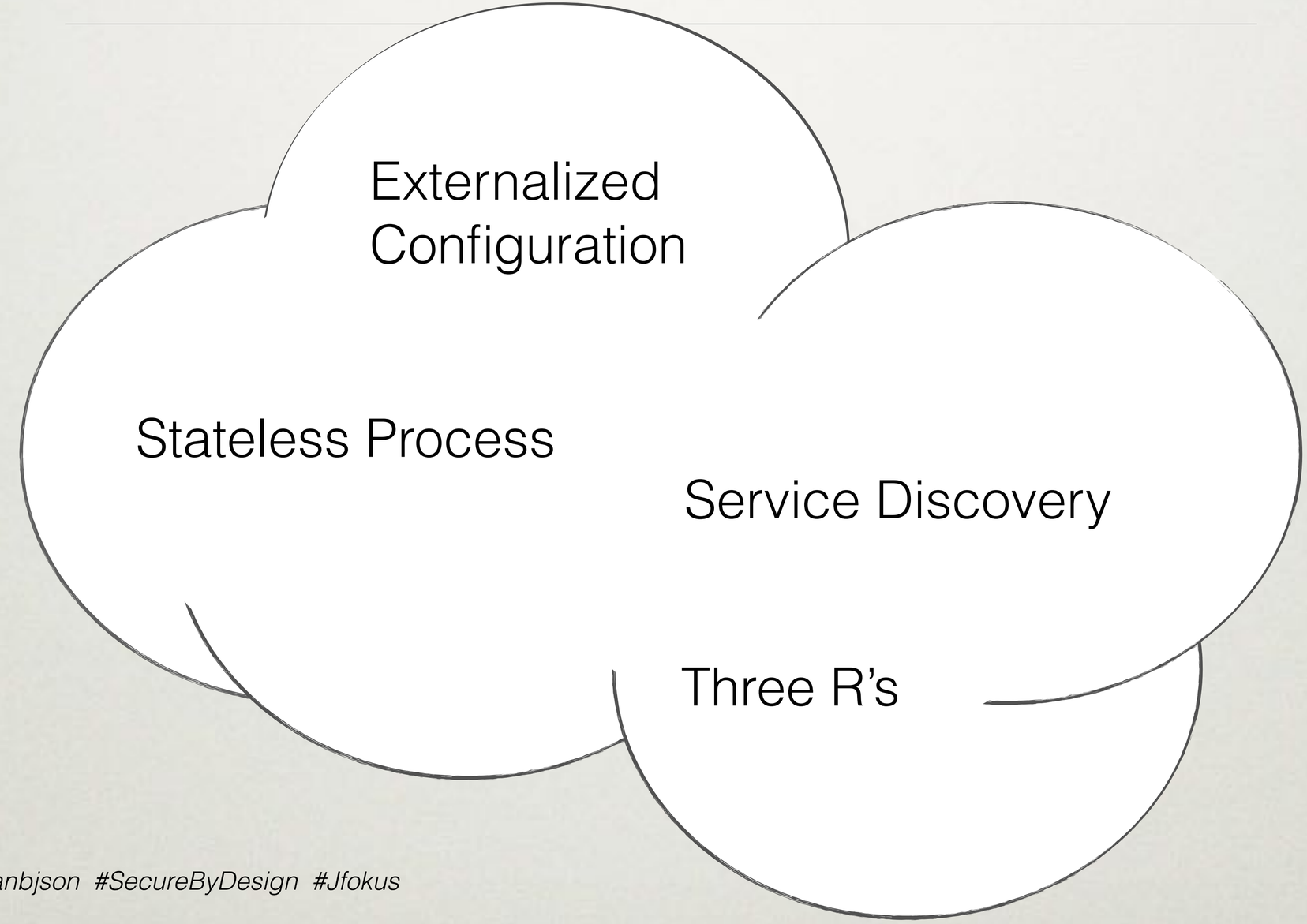
omega point.

Agenda

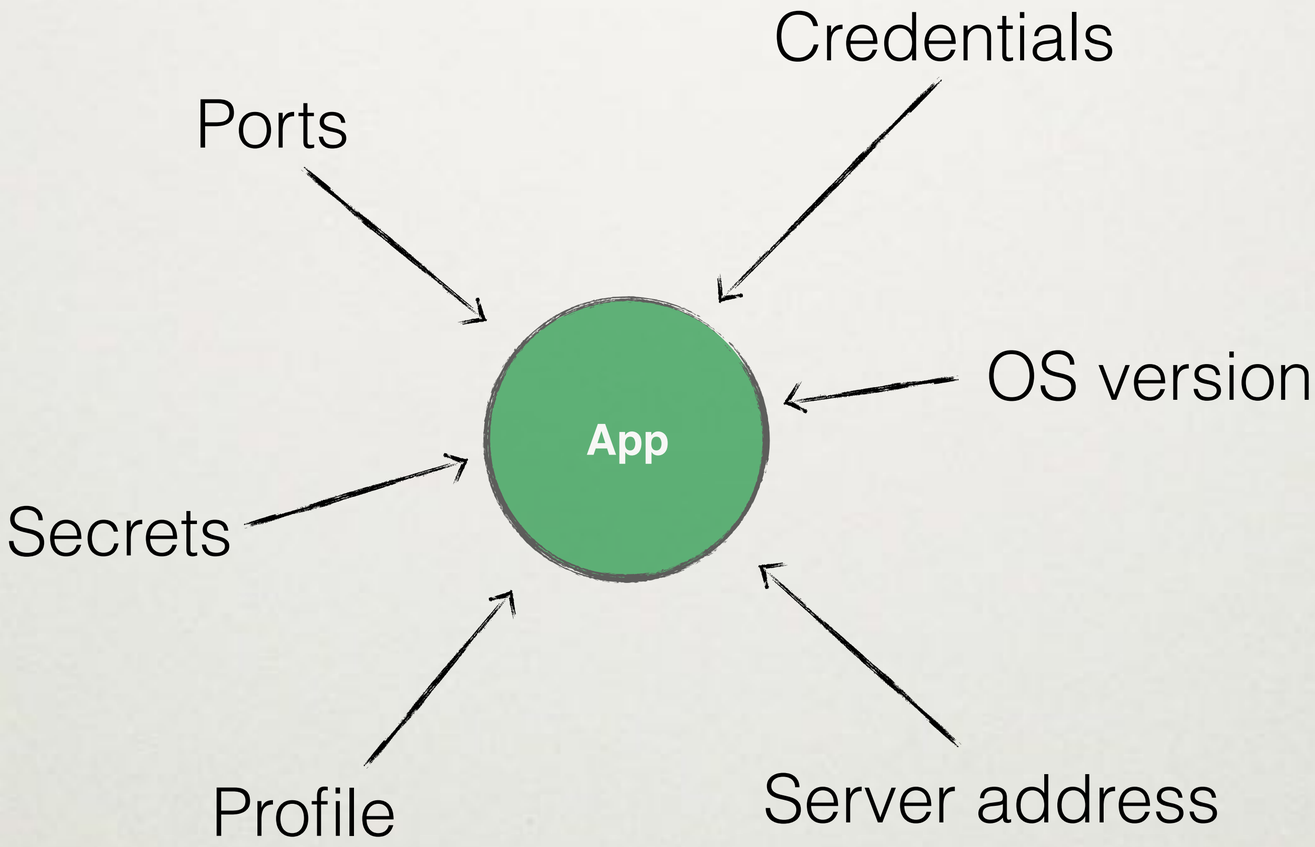
- **A Bank Robbery**
- **Secure by Design**
- **Technical solution to XSS**
- **Domain Primitives**
- **Sensitive Data Exposure**
- **XML and a Billion Laughs**
- **Automatic (security) unit tests**
- **Domain Primitives Applied on Legacy**
- **Fighting Entity Complexity**
- Cloud Thinking



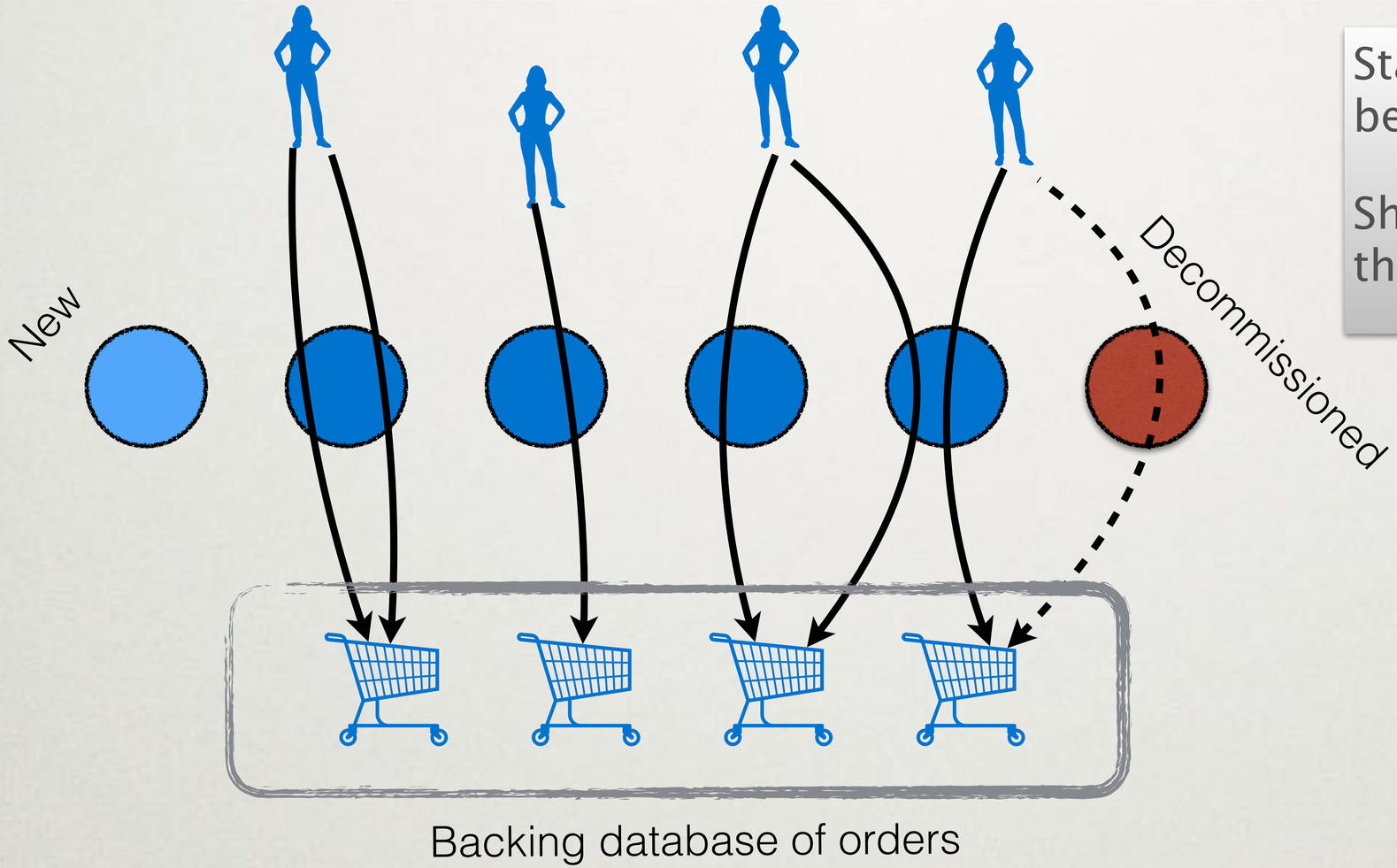
Cloud Thinking



Externalized Configuration



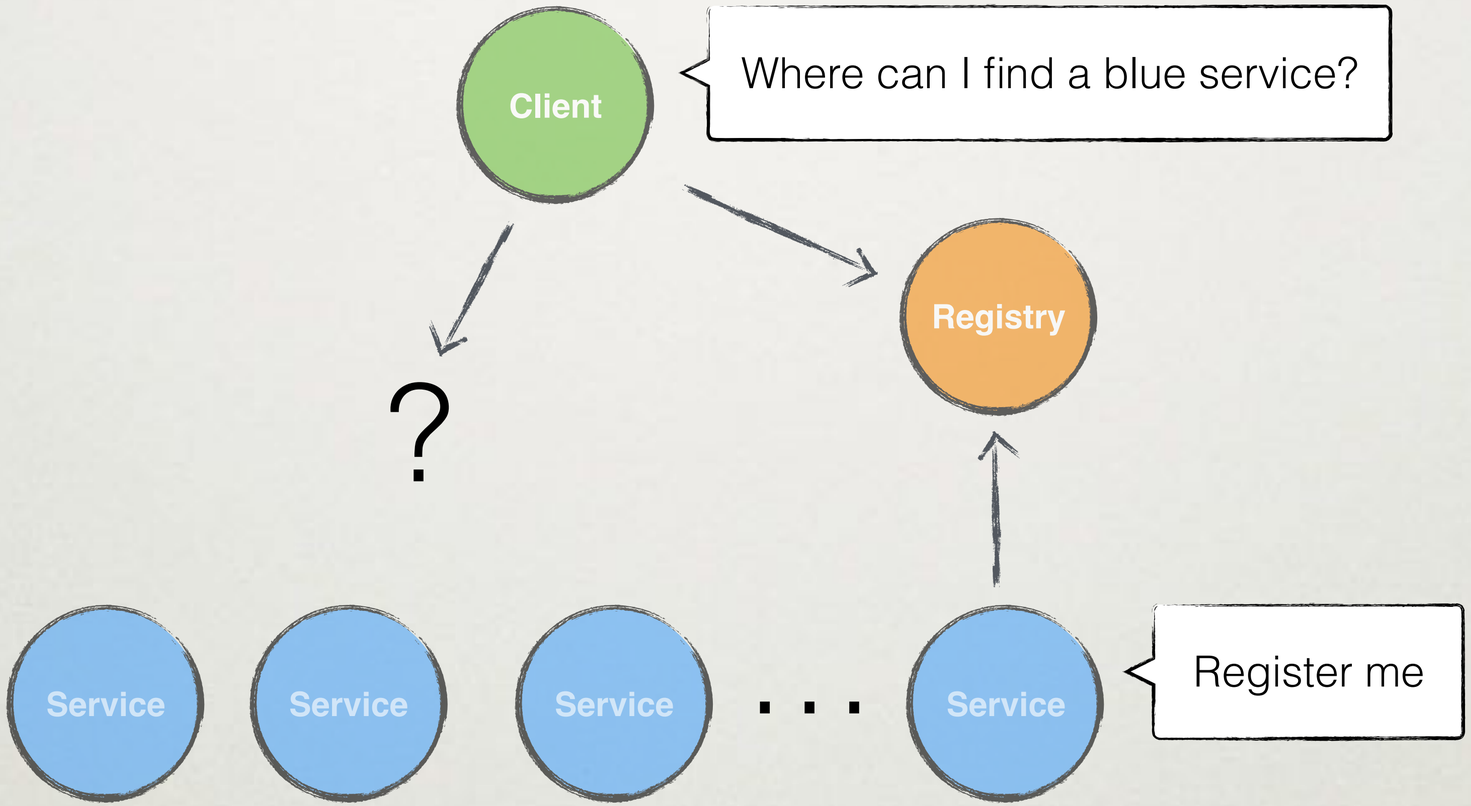
Stateless Processes



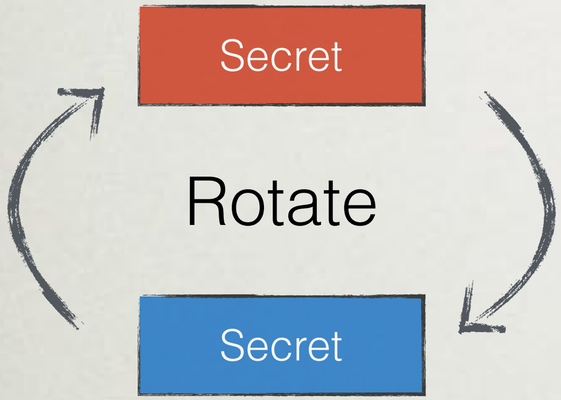
Stateless - no client state between calls

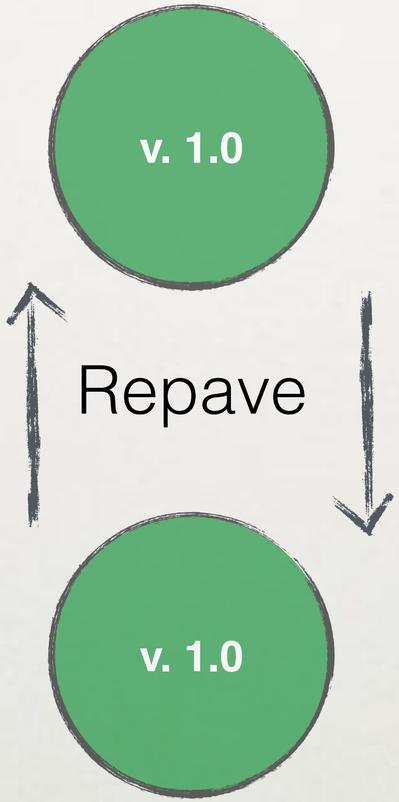
Share nothing - except through backing services

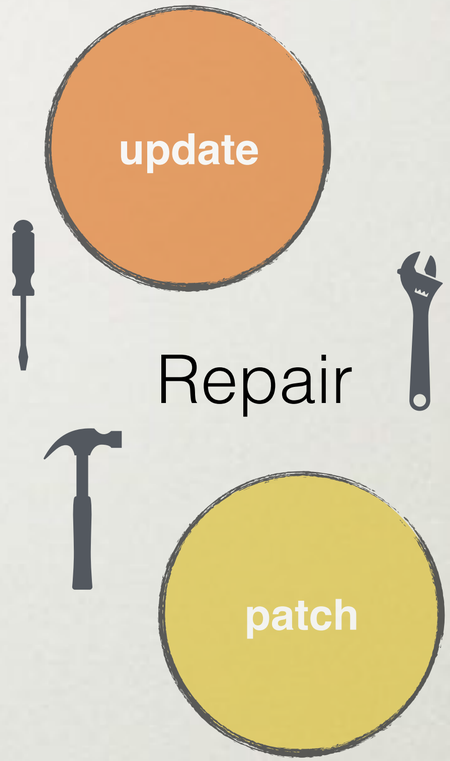
Service Discovery



Three R's





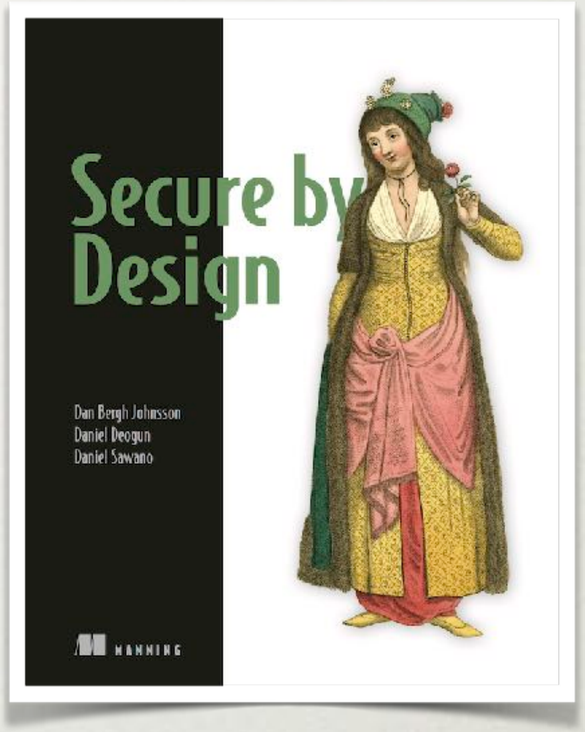


Ref: Justin Smith, Pivotal @justinjsmith

omega point.

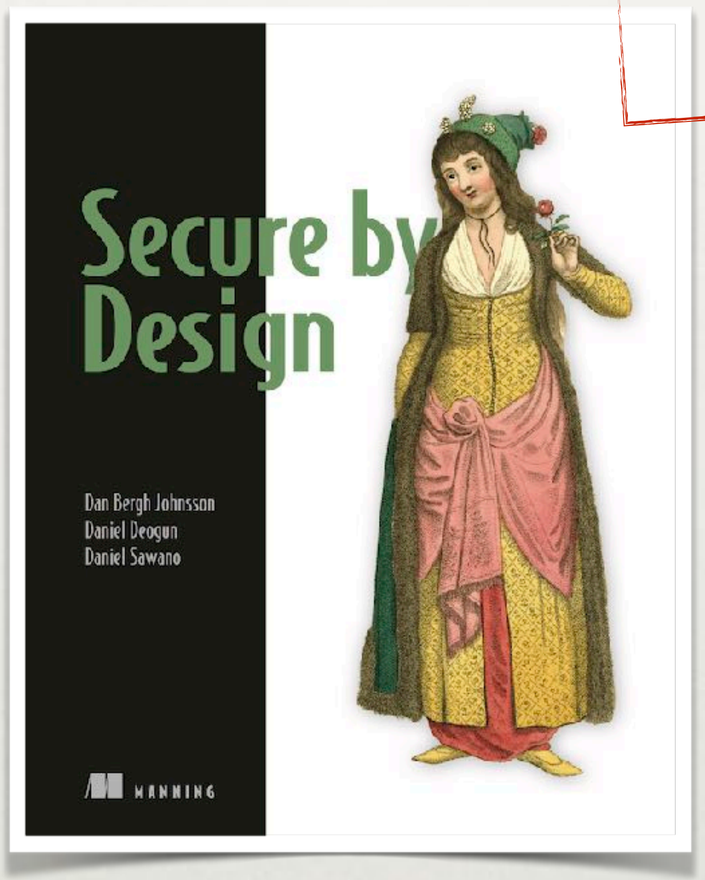
Agenda

- A Bank Robbery
- Secure by Design
- Technical solution to XSS
- Domain Primitives
- Sensitive Data Exposure
- XML and a Billion Laughs
- Automatic (security) unit tests
- Domain Primitives Applied on Legacy
- Fighting Entity Complexity
- Cloud Thinking



There's a Book ...
in Progress...

Design secure software without "thinking" about security



Early Access

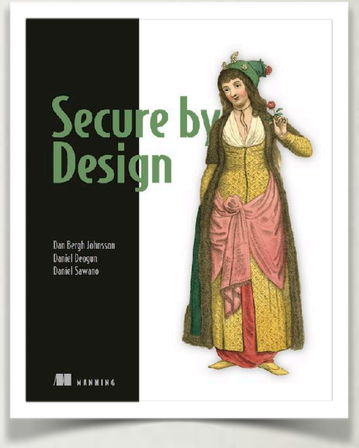
Discount code
ctwjfokus18
(40% off)

Key Take Aways

- Good design can yield a lot of security benefits
- Secure by design is a mindset
- Domain primitives enhances security in depth
- Model sensitive data in each context
- Reduce complexity of entities
- The three R's significantly reduces the attack window

Q&A

https://flic.kr/p/9ksxQa https://creativecommons.org/licenses/by-nc-nd/2.0/



ctwjfokus18

Thanks

@DanielDeogun @danbjson #SecureByDesign #Jfokus

omega point.