
Secure coding
Back to basics
SecAppDev 2024

Erlend Oftedal

Security researcher @ Crosspoint Labs

Software Developer for 20 years

Builds and maintains free open source security tools such as retire.js

OWASP Oslo chapter lead

@webtonull

Disclaimer

This presentation is being given in the presenter's individual capacity and reflect the subjective views and opinions of the presenter. This presentation is not meant to express any views of Crosspoint Labs or any of its affiliates. The statements contained herein cannot be independently verified and are subject to change.

"Complexity is the enemy of security"

Code and systems rarely start out as complex

Vulnerabilities are bugs

"50% of them are in business logic"

Bug types in order of personal preference

1. 🤩 Compile time errors
2. 🤗 Caught by automated tests
3. 😅 Caught by code inspection
4. 🤨 Caught by a manual test
5. 🤕 Runtime bug in production
6. 😬 Vulnerability
7. 😱 Exploited vulnerability
8. (😵 Pesky runtime bugs that only occur once per full moon when the stars are aligned)

[Security] Why input validation is not the solution for avoiding SQL injection and XSS

August 20, 2006 - 16:53 UTC - Tags: sql injection XSS input validation sql injection

Many web sites have SQL-injection and XSS (Cross Site Scripting) vulnerabilities, and security articles often mention lack of input validation as the reason for these problems. This isn't necessarily correct.

The metacharacter problem

Both SQL-injection and XSS are metacharacter problems. A metacharacter is a control character used in a part of the system to control the display or flow of data. These problems occur every time a system communicates with a system of a different flavour, be it a browser, a database or a legacy system.

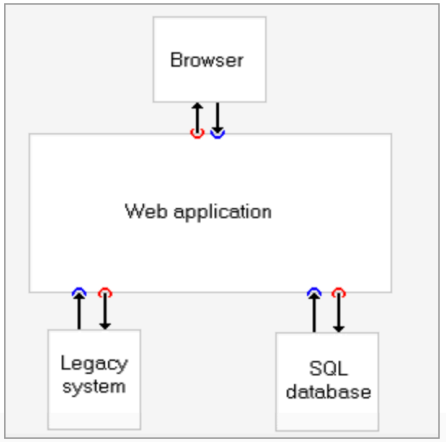
Why input validation can fail to solve these problems

Consider a blogging system allowing users to post comments to the entries. While this is a simple system, it contains enough functionality for me to explain. The blogging system has a comment form containing the fields: name, e-mail, headline and comment body.

Let's start out with the name field. To avoid getting XSS or SQL injection, input validation needs to block out any letters which are not used in a name. Using best practice, we create a white list of allowed values. While creating this white list would be easy if all names had the same format ("Joe Smith"), a problem arises when Conan O'Brian comes along. The ' in his name is a SQL control character and can also be used for strings in javascript or HTML tags. So how do we handle this using input validation. We have to allow this character. We also have to allow foreign names which do not necessarily follow standard formats.

Next, consider the comment field. If a user wants to post some code for showing how a certain javascript is written, maybe the blog should allow that. This means input validation will fail to remove XSS, or SQL injection for that matter.

So what do people do? During input validation, many people escape the HTML in the data, or escape the quote tags (', "). But is this really the best solution? Why should a comment exist in escaped format in the application. Java or C# or whatever language you are using, does not require the data to be escaped while contained in a string. You can quickly run into trouble when you start displaying data from multiple sources. What data is escaped, and what data isn't? Is all data escaped? What is it escaped for? HTML? SQL? Both? Some weird legacy system?



The solution

Don't get me wrong. I still think input validation should be present in every application. But to avoid metacharacter problems, data needs to be escaped when it leaves the system, not when it enters it. This means that the web application needs to escape data just before sending it to the database (preferably by using prepared statements) or a legacy system. Data presented on an HTML page needs to be escape when it's written to the HTML page. And best practice for escaping should be "Escaping by default", which means you need a reason if you are printing unescaped data.

In the figure on the right, I have marked where input validation (blue) or output escaping (red) should be performed in a web application.

Domain Driven Security

Original idea by

- Dan Berg Johnsson (Omegapoint AB)
- John Wilander (Apple)

Inspired by Domain Driven Design

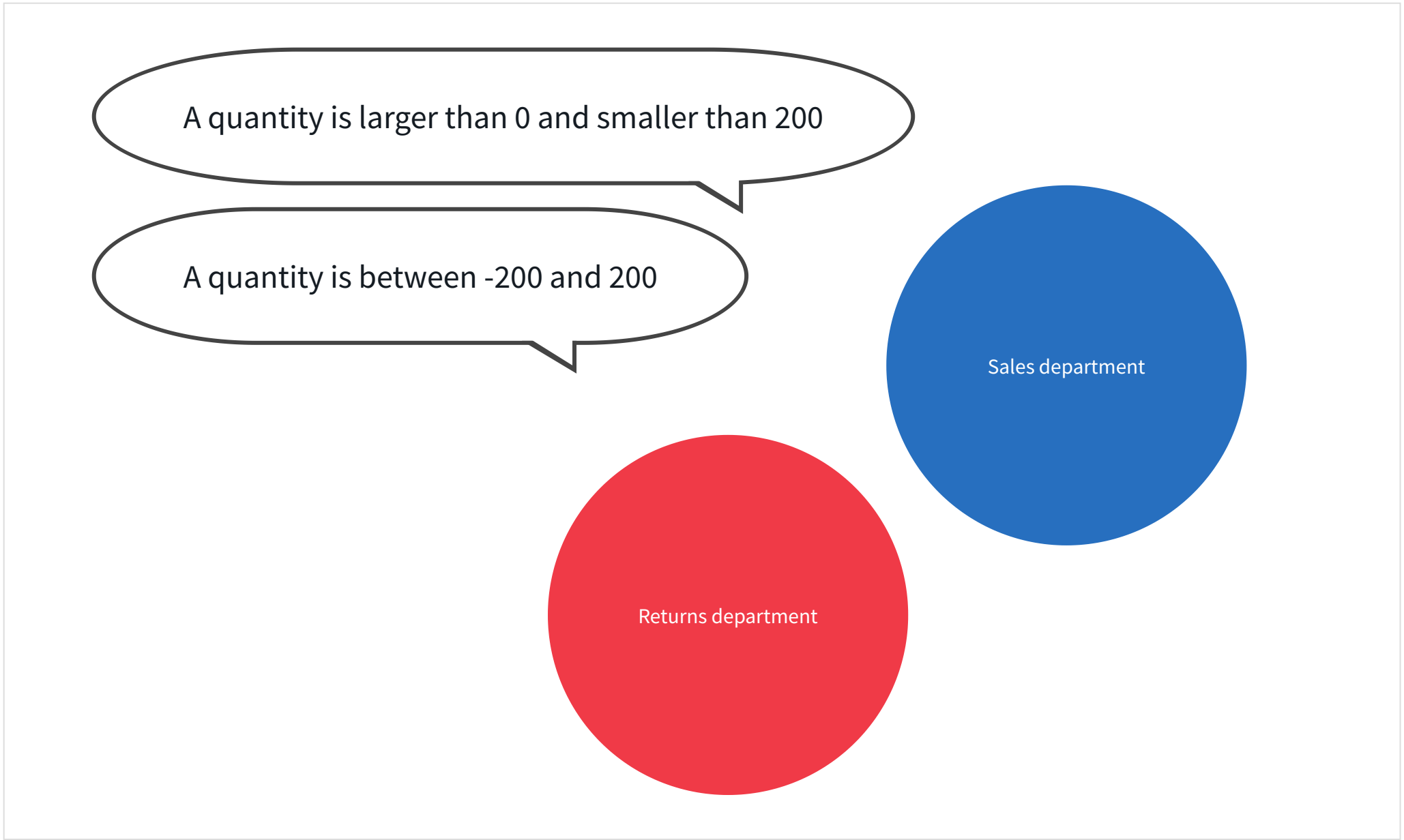
Book: Secure by Design

Terminology - Domain Driven Design/Security

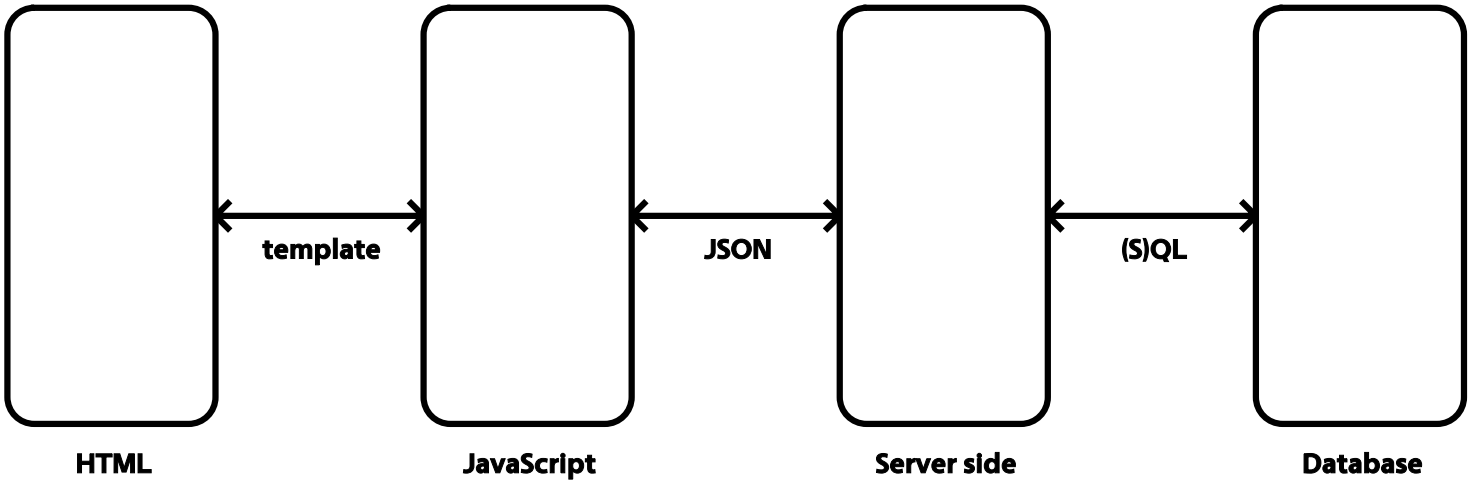
Ubiquitous language

Bounded context

Anti-corruption layer


A quantity is larger than 0 and smaller than 200
A quantity is between -200 and 200
Sales department
Returns department

The multifaceted entity




Input validation

Input validation

Goal: Input should be valid in the domain of the system (DDD: anti-corruption layer)

Side effect: Stops or limits many types of attacks

Syntax versus semantics

Syntax	"correct format?"
Semantics	"do these values make sense in the context?"
Domain	"are these values valid according to our data?"

Validating a birth date

Syntax: `/^[0-9]{4}-[0-9]{2}-[0-9]{2}$/`

Semantics: `date.isAfter('1900-01-01') && date.isBefore(now())`

Validating a productId

Syntax: `isNumeric()`

Domain: `productExists(id)`

Validating comments

"Thank you for this blog post."

```
/^[A-Za-z .,0-9]$/
```

"There is an error in your second <script>-tag."

```
/^[A-Za-z .,0-9<>"']$/
```

```
if (isValid(data)) {
   use(data)
}
```

Encoding / Charset

Unicode ≈ list of symbols (U+0000000 - U+10FFFF)

UTF-8, UTF-16, UTF-32 are Unicode encodings

ISO 8859-1 is an ASCII encoding and uses 1 byte per character

UTF-8 is compatible with ASCII for U+000000-U+00001F (0-9, A-Z, a-z ++)

Æ, Ø, Å is 1 byte in ISO-8859-1, but two bytes i UTF-8

ISO-8859-1: `ø = 11111000`

UTF8:

```
0XXXXXXX
1100XXXX:10XXXXXX
1110XXXX:10XXXXXX:10XXXXXX
1111XXXX:10XXXXXX:10XXXXXX:10XXXXXX
```

UTF8: `ø = 11000011:10111000 = C3 B8 = Ã¸` (ISO-8859-1)

UTF32: `㰼 = 00 00 3C 00 = □□<□` (ASCII)

Encoding / Charset - JavaScript

```
'abc'.length
3
'a≡c'.length
4
"á" == "á"
false
"á".length
2
"á".length
1
"á".length
9
"👥".length
11
```

Encoding / Charset - JavaScript

```
▸ "👨‍🚀".length
  5
▸ "👨‍🚀".split('')
  (5) ['\uD83D', '\uDC68', '', '\uD83D', '\uDE80']
▸ "👨‍🚀".split('')
  (2) ['👨', '🚀']
```

Encoding / Charset - MySQL

Tables with charset `"utf8"` only support 1-3 byte characters, but UTF-8 is 1-4 bytes...

`'abc' = 'abc'`

`'æøå' = 'æøå'`

`'a≡c' = 'a'`

Use `"utf8mb4"`

Source: Mathias Bynens - Hacking with Unicode

Encoding / Charset - JavaScript

Specify character sets on requests and responses

```
<meta charset="utf-8">
```

```
Content-Type: application/json;charset=utf-8
```

Default to UTF-8

Test with 4-byte characters (astral symbols) like 𝌆

The Pile of Poo Test™:
`Iñtërnâtiônàlizætiøn☃💩`

IP-adresses - Deny lists are hard...

169.254.169.254

425.510.425.510

2852039166

7147006462

0xA9.0xFE.0xA9.0xFE

0xA9FEA9FE

0x414141410A9FEA9FE

0251.0376.0251.0376

0251.00376.000251.0000376

Source: http://www.agarri.fr/docs/AppSecEU15-Server_side_browsing_considered_harmful.pdf

Principle: Normalize input

Common format before validation

Examples:

- **XML**: `<middlename></middlename>` vs. `<middleName />`
- **Character sets**: Unicode - NFD, NFC, NFKD, NFKC
- **Email / URLs / host names**: Punycode

Principle: Reject invalid input

Reject - don't clean

```
<scri<script>pt>alert(1)</scri</script>pt>
```

400 Bad Request

Input validation steps

1. Normalize
2. Check length
3. Check format/syntax
4. Check semantics
5. Use the normalized validated value
6. Verify values (domain)

Heartbleed

Heartbeat:

"Here are x bytes of data. Please echo it if connection is still alive."

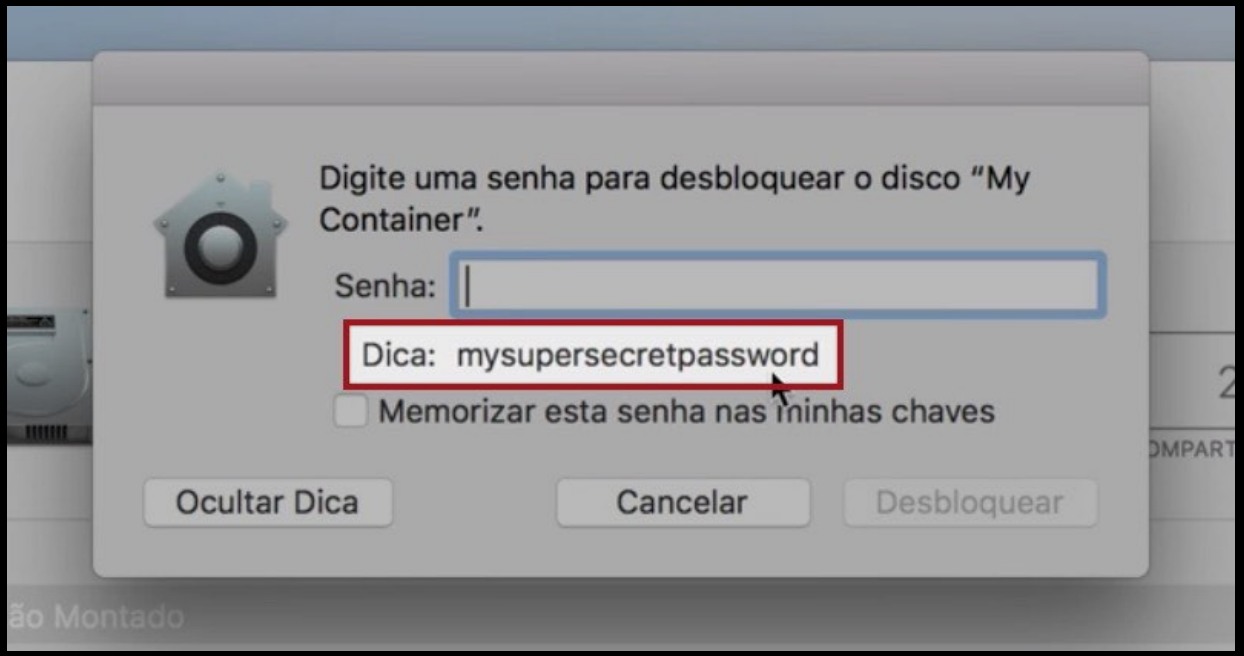
OpenSSL vulnerability

"Here are x bytes data" (actually sends y < x bytes)

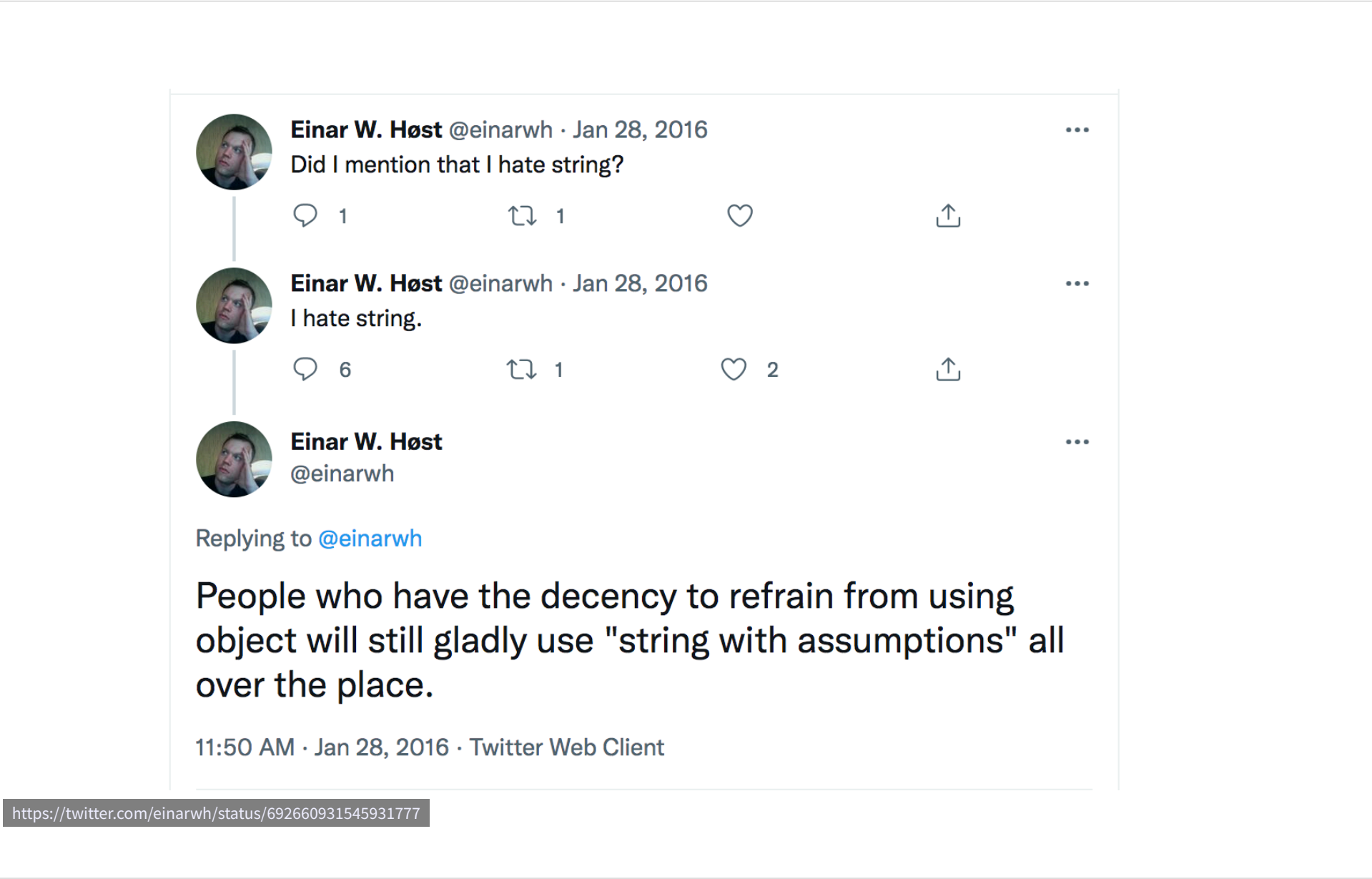
Server responds with x bytes

Syntactic vs. semantic input validation


Bringing in Domain Driven Security

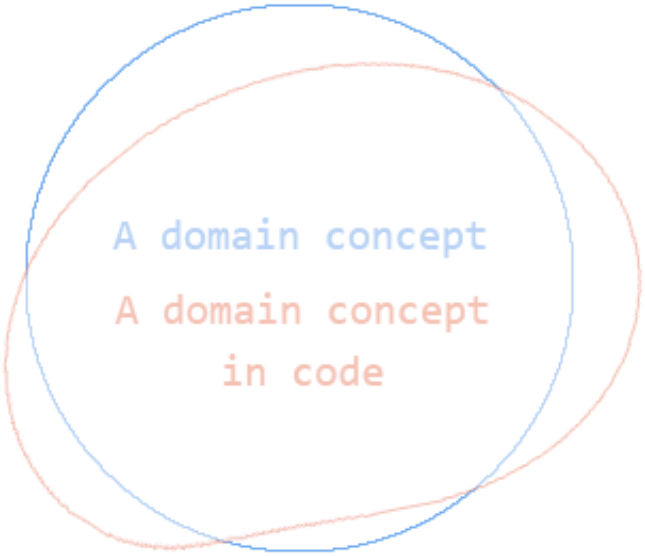


https://hotforsecurity.bitdefender.com/blog/apple-fixes-flaw-that-displayed-actual-password-rather-than-password-hint-19042.html



https://twitter.com/einarwh/status/692660931545931777

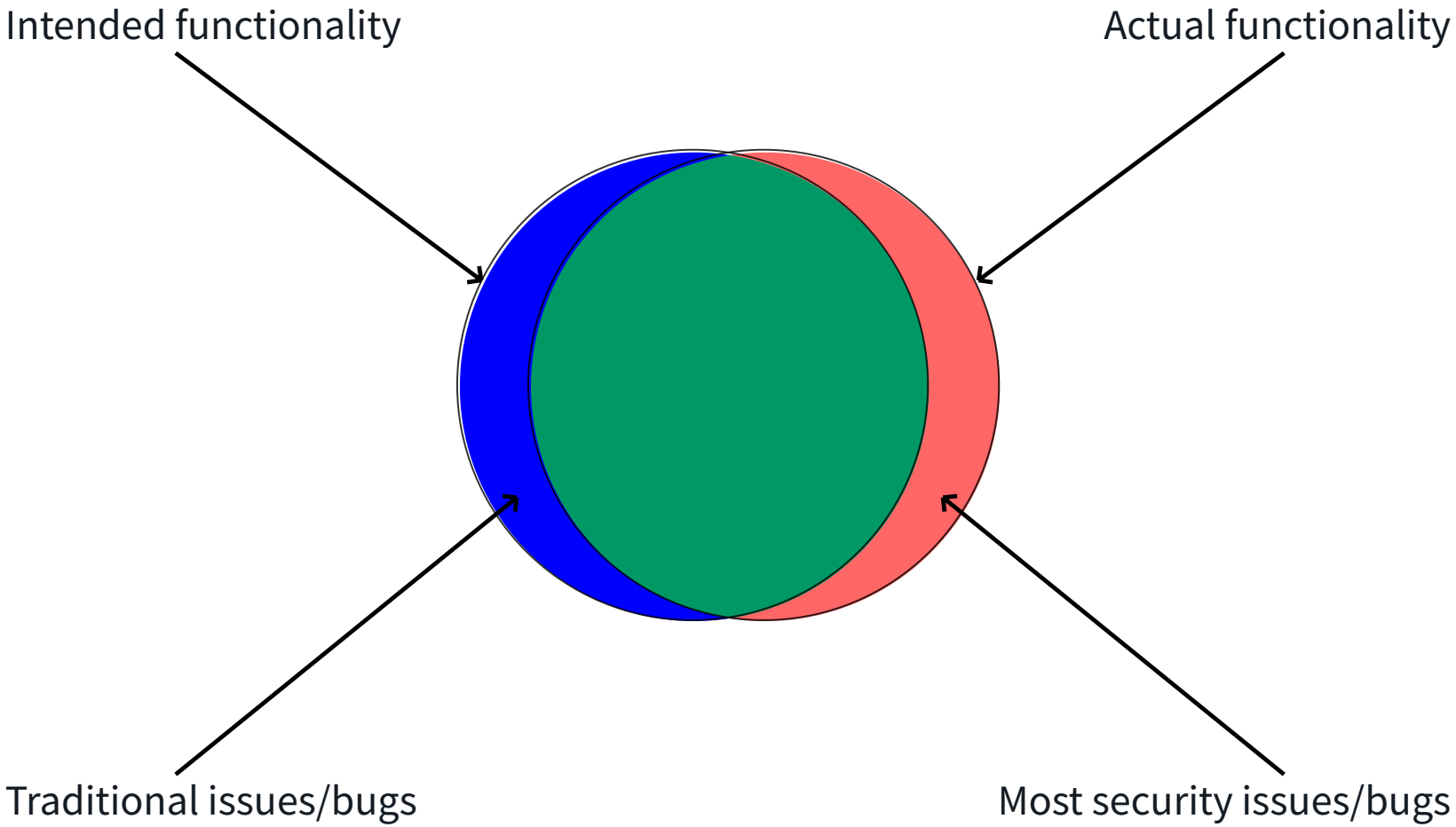
In the context of programming, I assert a congruence problem is when a domain provides explicit concepts, but a programmer chooses to code the concept using abstractions that don't match, and are often at a lower level of abstraction. Essentially, what is in the programmer's head is *incongruent* with the code; they don't superimpose well.



WTF, right?

It sounds a lot like "primitive obsession", which is using primitive types to represent domain concepts. The difference is the congruence problem speaks to our tendency to ignore certain parts of the domain and instead map our thoughts to concepts outside of the domain. Primitive obsession is often the result. It can also result in dependence on other domain objects to compose functionality in situations where creating new abstractions is a better idea.

https://thesecretsquad.wordpress.com/2016/02/27/something-isnt-something-else/ think that using the tools provided to us is the simplest thing to do?



Source: "How to break software security" - J. A. Whittaker and Hugh Thompson, 2003

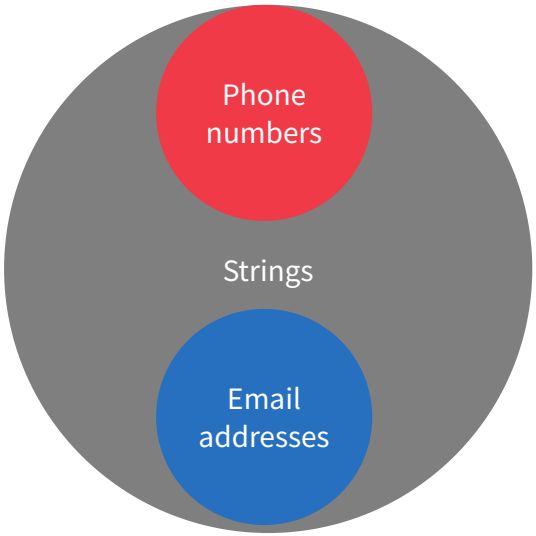
Domain Driven Security by example

Stringly typed != Strongly typed

```
public void register(String emailAddress, String password, String phoneNumber)

public void register(EmailAddress email, Password password, PhoneNumber phone)

public void register(UserRegistration registration)
```



The Domain Primitive

Wraps a value and enforces invariants

Makes an implicit concept explicit

```
public class PhoneNumber {
   public final String value;

   public PhoneNumber(String phoneNumber) {
      if (!isValidPhoneNumber(phoneNumber)) {
         throw new ValidationException("Invalid phone number:" + phoneNumber);
      }
      this.value = phoneNumber;
   }
   public static boolean isValidPhoneNumber(String phoneNumber) {
      //Validation rules
      ...
   }
   @Override
   public boolean equals(Object anotherObject) {...}

   @Override
   public int hashCode() {...}
   ...
   // other logic (example: getCountryCode)
}
```

The Domain Primitive - Java Record

```
public record PhoneNumber(
   String value
) {
   public PhoneNumber {
      if (!isValidPhoneNumber(value)) {
         throw new ValidationException("Invalid phone number:" + value);
      }
   }
   // other logic (example: getCountryCode)
}
```

Java Records are immutable data classes

Why immutable classes?

No side effects due to unexpected updates of the object

Forces all changes to rerun the validation logic

Domain Primitive - Kotlin with Either

```
class PhoneNumber private constructor (
      val value: String
) {
   init {
      value.mustSatisfy(checks)
   }

   companion object {
      val checks = arrayOf(
         lengthBetween(8,20),
         match("^[+]?[0-9]+$")
      )

      fun create(candidate: String) : Either<ValidationError, PhoneNumber> {
         checks.forEach(check -> {
            val error = check(candidate)
            if (error != null) return Either.Left(error)
         })
         return Either.Right(PhoneNumber(candidate))
      }

      fun isValid(candidate: String) : Boolean {
         return create(candidate) is Either.Right
      }
   }
   ...
}
```

Domain Primitive - Password

```
class Password(
      private val value
) {

   ...
   override fun toString() : String {
      return "*******"
   }

   fun asString() : String {
      return password
   }
   ...
}
```

Some validation needs to be enforced outside the primitives

Email:

- Must be unique

Phone number:

- Must be a mobile number (for 2 factor SMS)

Password:

- Must not contain the user's email or phone
- Must not contain the user's names
- Must not be a commonly used passwords from breaches

The UserRegistration Entity

```
   fun register(userdata: UserRegistration): Optional<ValidationError> {
      ...
   }
```

```
class UserRegistration private constructor(
...
) {
   companion object {
      fun create(
        email: EmailAddress, phone: PhoneNumber, fullName: FullName, password: Password
      ): Either<ValidationError, UserRegistration> {

         val checks = arrayOf(
            password.mustNotContain(email.value)
            password.mustNotContain(phoneNumber.value)
            fullName.names().forEach { n -> password.mustNotContain(n) }
            //Other validations
         )
         ...
      }
   }
   ...
}
```

Domain validation of Password

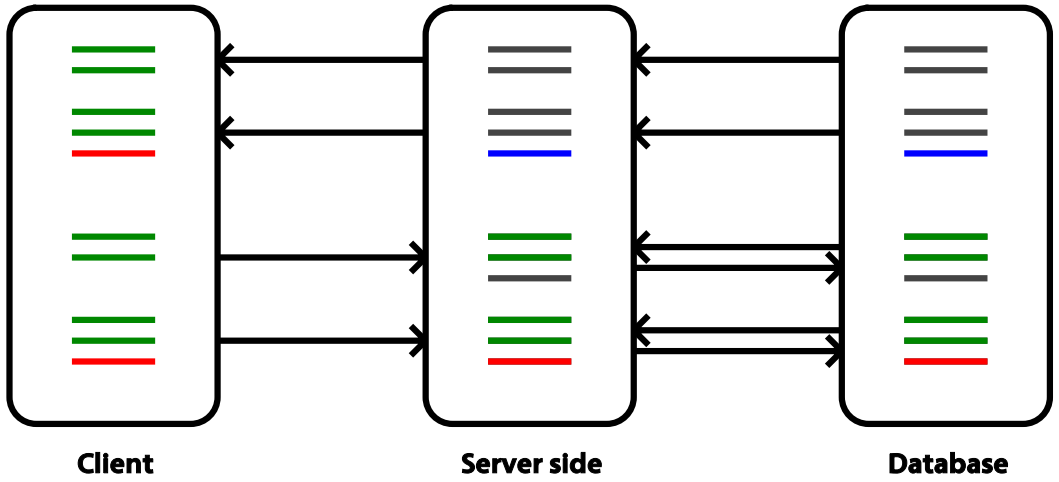
```
fun storeCredentials(username: EmailAddress, password: ValidatedPassword): Optional<ValidationError> {
   ...
}
```

```
class PasswordValidator {
   ...
   fun validate(candidate: Password) : Either<Error, ValidatedPassword> {
      //check not in list of common passwords from breaches
      //other checks
      ...
   }
   ...
}
```

Domain primitives enforce simple syntax and semantics

Services enforce domain invariants and transform objects

API



homakov opened this issue in 1001 years

I'm Bender from Future.

No one is assigned

```
Hey. Where
```

homakov commented on Mar 2, 2012

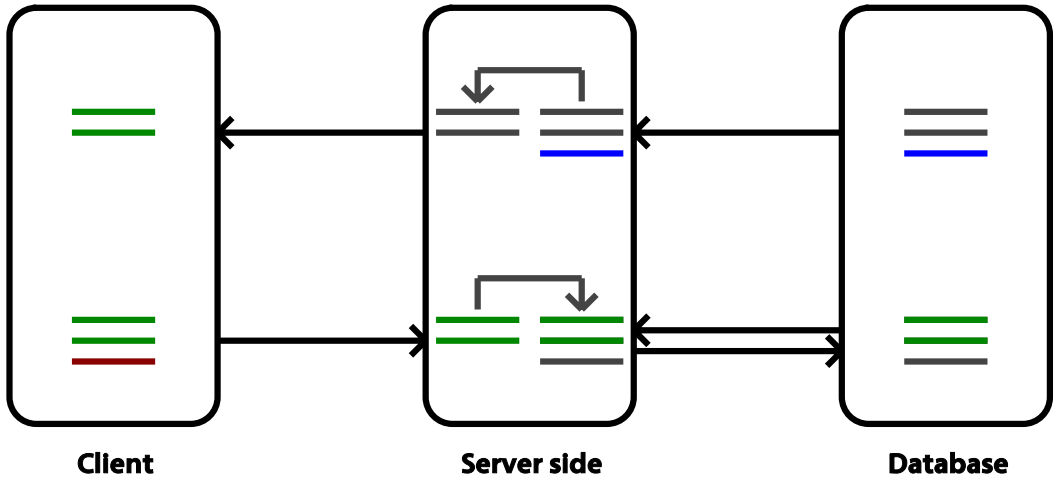
```
Hey. Where is a suicide booth?
```

from 3012 with love

You should check it ... #5228
[CONTENT IS FOR SALE EITHER]

https://github.com/rails/rails/issues/5239

API - DTO/Contract/Anti-corruption layer



The wire format and DTOs

```
{
      "order" : {
         "id": "bfa40dba-86c1-4a89-9e55-06014e3848cf",
         "items" : [
            { "productId" : "6f7ca817-bcc7-4398-b3db-faa6b5a4ce84", "quantity" : 4 },
            { "productId" : "68189787-cf10-44c7-b6b4-fe420ddd1939", "quantity" : 2 }
         ]
      }
}
```

```
class OrderDto(
      val id: String,
      val items: List<OrderLineDto>
) {
   fun toDomain(): Order {
      return Order(
         OrderId(this.id),
         this.items.map { l -> l.toDomain() }.toList()
      )
   }
}

class OrderLineDto(
      val productId: String,
      val quantity: Int
) {
   fun toDomain(): OrderLine {
      return OrderLine(
         ProductId(this.productId),
         Quantity(this.quantity)
      )
   }
}
```

Using simple inheritance

```
class WrappedUuid (
      val uuid: String
) {
   init {
      UUID.parse(uuid)
   }
}

class OrderId(uuid: String) : WrappedUuid(uuid)

class ProductId(uuid: String) : WrappedUuid(uuid)
```

The Order DTO - there and back again

```
class OrderDto(
      val id: String,
      val items: List<OrderLineDto>
) {
   fun toDomain(): Order {
      return Order(
         OrderId(this.id),
         this.items.map { l -> l.toDomain() }.toList()
      )
   }

   companion object {
      fun from(order: Order): OrderDto {
         return OrderDto(
            order.id.value,
            order.orderLines.map { l -> OrderLineDto.from(l) }.toList()
         )
      }
   }
}
```

The DTO is responsible for the mapping

The Order class

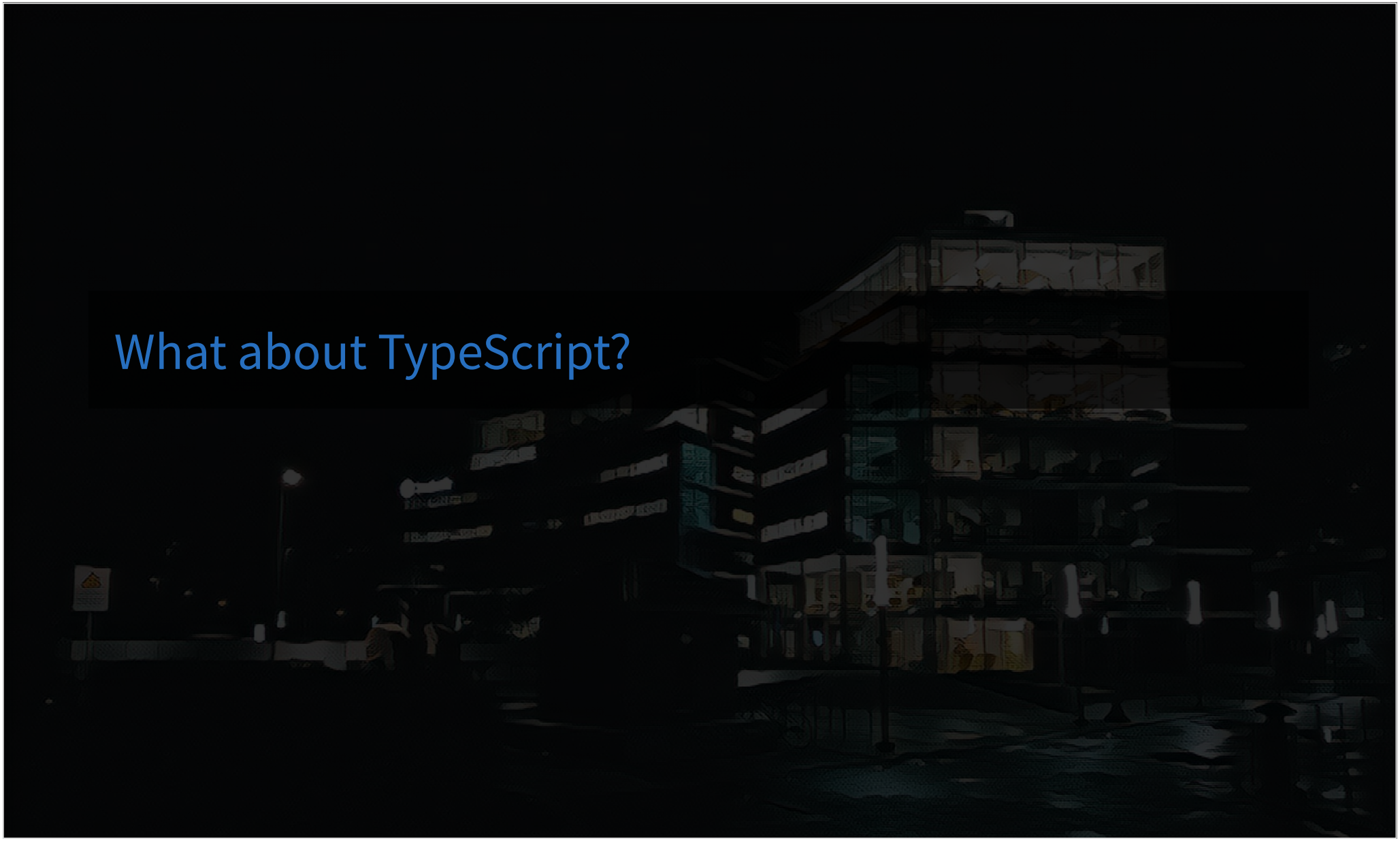
```
class Order private constructor(
      val id: OrderId,
      val orderLines: List<OrderLine>
) {
   fun add(orderLine: OrderLine): Either<ValidationError, Order> {
      return create(id, orderLines + listOf(orderLine))
   }

   companion object {
      fun create(id: OrderId, orderLines: List<OrderLine>): Either<ValidationError, Order> {
         if (orderLines.sumBy { l -> l.quantity.value } > 200) {
            return Either.Left(ValidationError("Order cannot have more than 200 items"))
         }
         return Either.Right(Order(id, orderLines))
      }
   }
}
```

Order itself is immutable - the only way to change it is to create a new one.

API Controller responsibilities

```
fun someControllerAction(someDto: SomeDto): Response {
   // 1. Map the HTTP request data to domain objects (and thereby validate)
   // 2. Call the correct processing method
   // 3. Map result of the processing to an HTTP response
}
```


What about TypeScript?

Can we use classes?

```
class Username {
   constructor(readonly value: string) {
      ensure(value).satisfies(
         notNullOrUndefined(),
         hasLengthBetween(1, 30),
         matches(nameRegex)
      )
   }
}
class Password {
   constructor(readonly value: string) {
      ensure(value).satisfies(
         notNullOrUndefined(),
         hasLengthBetween(8, 256)
      )
   }
}

const username = new Username("smith");
const password = new Password("not a real password");

function register(username: Username, password: Password) {
   //...
}
```

```
register(username, password); //Compiles
register(password, username); //Also compiles...🙃
```

Yey... duck-typing ... 🦆

```
function register(username: Username, password: Password) {
   //...
}
```

is basically

```
function register(username: { value: string }, password: { value: string }) {
   //...
}
```

In fact, we can pass:

```
register({ value: "" }, { value: "" })
```

Private fields kind of makes it work...

```
class Username {
   private _i = 1; //What?
   constructor(readonly value: string) {
      ensure(value).satisfies(
         notNullOrUndefined(),
         hasLengthBetween(1, 30),
         matches(nameRegex)
      )
   }
}
class Password {
   private _i = 1; //Huh?
   constructor(readonly value: string) {
      ensure(value).satisfies(
         notNullOrUndefined(),
         hasLengthBetween(8, 256)
      )
   }
}


register(password, username);
// Argument of type 'Password' is not assignable to parameter of type 'Username'.
// Types have separate declarations of a private property '_i'.ts(2345)
```

But this is a bit ugly...

Wrapping in entities helps as well

```
class UserRegistration{
   constructor(
      readonly username: Username,
      readonly password: Password
   ) {}
}

function register(registration: UserRegistration) {
   //...
}
```

What about input validation of JSON data?

https://github.com/colinhacks/zod

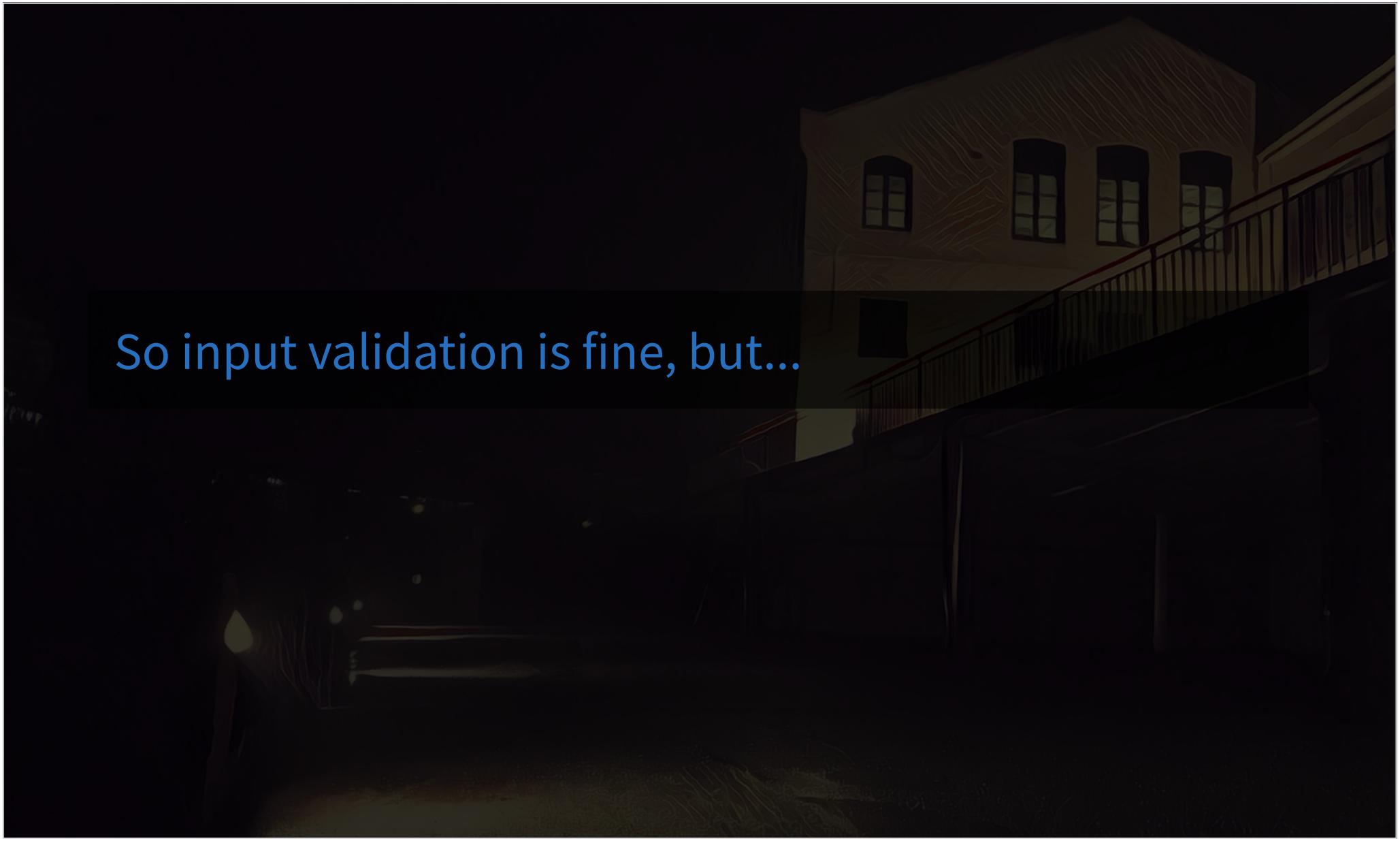
```
import * as z from "zod";

export const registrationParser = z.object({
   email: z.string().min(10).max(60).regex(emailRegex),
   password: z.string().min(8).max(256)
});
```

```
export type Registration = z.infer<typeof registrationParser>;
// { email: string, password: string }
```

```
...
const registration = registrationParser.parse(jsonData); // throws on error
...
```

```
...
const parseResult = registrationParser.safeParse(jsonData);
if (!parseResult.success) {
   log.error("Failed to parse registration", parseResult.error);
   return response.status(400).end();
}
const registration = parseResult.data;
...
```


So input validation is fine, but...

[Security] Why input validation is not the solution for avoiding SQL injection and XSS

August 20, 2006 - 16:53 UTC - Tags: sql injection XSS input validation sql injection

Many web sites have SQL-injection and XSS (Cross Site Scripting) vulnerabilities, and security articles often mention lack of input validation as the reason for these problems. This isn't necessarily correct.

The metacharacter problem

Both SQL-injection and XSS are metacharacter problems. A metacharacter is a control character used in a part of the system to control the display or flow of data. These problems occur every time a system communicates with a system of a different flavour, be it a browser, a database or a legacy system.

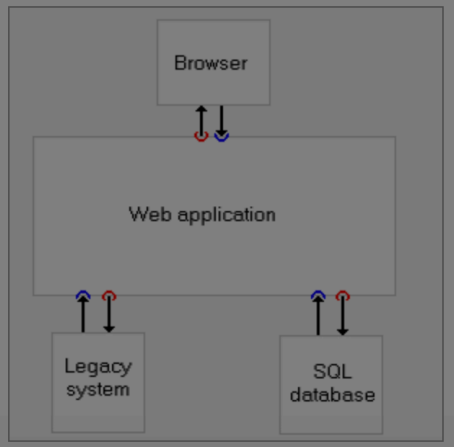
Why input validation can fail to solve these problems

Consider a blogging system allowing users to post comments to the entries. While this is a simple system, it contains enough functionality for me to explain. The blogging system has a comment form containing the fields: name, e-mail, headline and comment body.

Let's start out with the name field. To avoid getting XSS or SQL injection, input validation needs to block out any letters which are not used in a name. Using best practice, we create a white list of allowed values. While creating this white list would be easy if all names had the same format ("Joe Smith"), a problem arises

> ... While creating this allow list would be easy if all names had the same format ("Joe Smith"), a problem arises when Conan O'Brian comes along. ...

So what do people do? During input validation, many people escape the HTML in the data, or escape the quote tags (', "). But is this really the best solution? Why should a comment exist in escaped format in the application. Java or C# or whatever language you are using, does not require the data to be escaped while contained in a string. You can quickly run into trouble when you start displaying data from multiple sources. What data is escaped, and what data isn't? Is all data escaped? What is it escaped for? HTML? SQL? Both? Some weird legacy system?



The solution

Don't get me wrong. I still think input validation should be present in every application. But to avoid metacharacter problems, data needs to be escaped when it leaves the system, not when it enters it. This means that the web application needs to escape data just before sending it to the database (preferably by using prepared statements) or a legacy system. Data presented on an HTML page needs to be escape when it's written to the HTML page. And best practice for escaping should be "Escaping by default", which means you need a reason if you are printing unescaped data.

In the figure on the right, I have marked where input validation (blue) or output escaping (red) should be performed in a web application.

Gijs in 't Veld @gintveld 1d

Great to see that my name still causes SQL errors and that errors thrown are so hacker friendly. ;-)
#integrate2016
pic.twitter.com/xGwsckAzvR

```
newusers ( username, attribute, op, value,
callingstationid, displayname, created_at )
VALUES ( 'cc89fdd01ea0', 'Cleartext-Password',
':=', '70358542', 'cc-89-fd-d0-1e-a0', 'Gijs in 't
Veld', NOW() ),( 'cc89fdd01ea0', 'User-Profile',
':=', 'free750', '', '', NOW() )
```

https://twitter.com/gintveld/status/730317679886794752

Input validation is not enough to stop XSS and SQLi

Cause of injections: Data does not stay data

⚠ Don't attempt to encode during input validation

How do you know what encoding to use?

How is the data used now?

How will it be used in the future?

What if your encoding was wrong?

Encoding and language constructs to the rescue

SQL-injection → parameterized queries

XSS → frameworks like React that (mostly) handle it for us

Other injection attacks → Escape or encode

Confession - I failed to communicate

What we had:

```
result = con.executeQuery("SELECT * FROM Product WHERE id='" + id + "'");
```

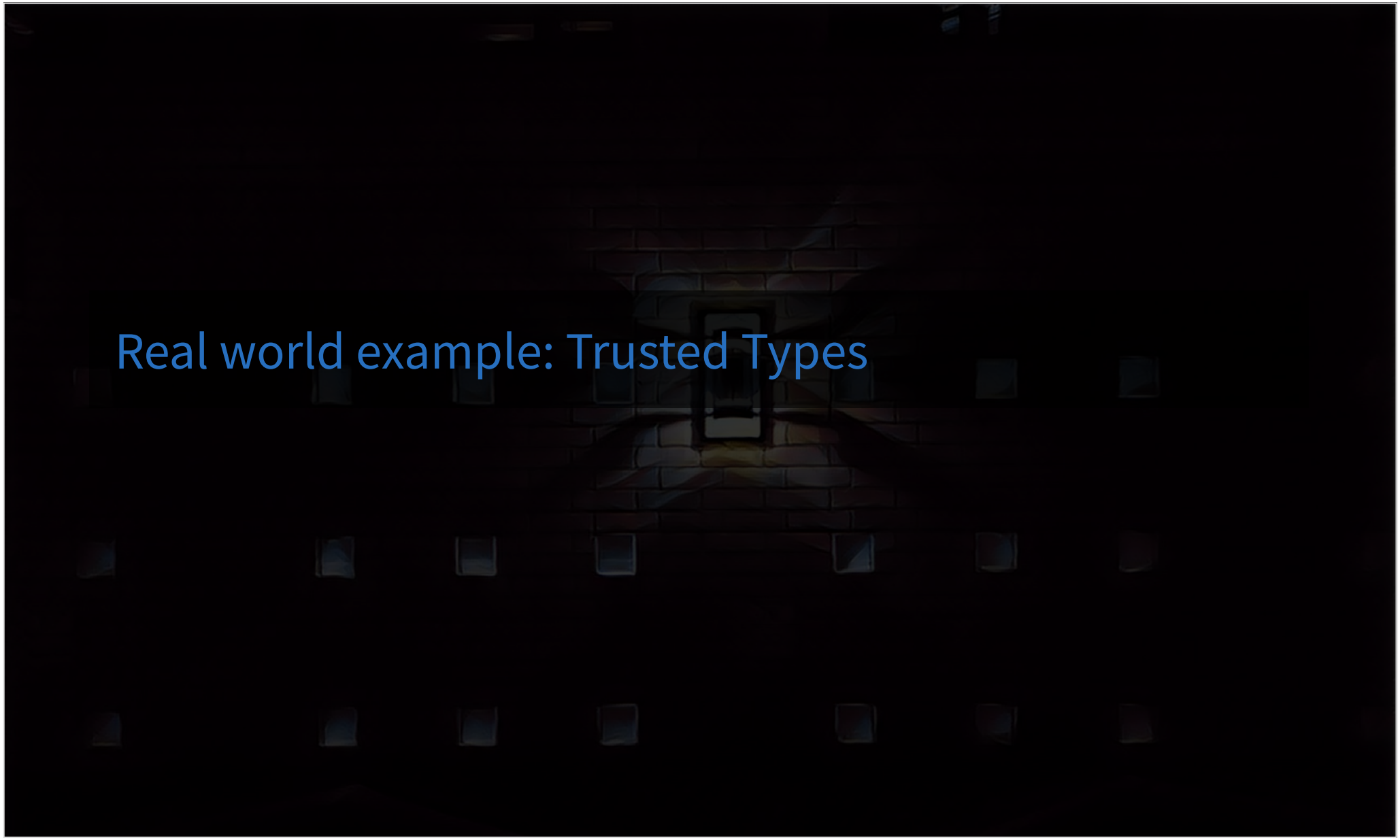
What we expected:

```
PreparedStatement stmt = con.prepareStatement("SELECT * FROM Product WHERE id=?");
stmt.setString(1, id);
result = stmt.executeQuery();
```

What we got:

```
PreparedStatement stmt = con.prepareStatement("SELECT * FROM Product WHERE id='" + id + "'");
result = stmt.executeQuery();
```

Can we do something with the APIs?


Real world example: Trusted Types

Common sources of client-side XSS

```
...
elem.innerHTML = taintedData;
...
```

```
export class MyComponent extends React.Component<...> {
...
  render () {
    return (
      <div dangerouslySetInnerHTML={ {__html: this.props.body} } />
    )
  }
}
```

Trusted Types

HTTP Reponse header:

```
Content-Security-Policy: enforce-trusted-types-for 'script'
```

```
> document.body.innerHTML = "<img src=x onerror=alert(1)>"
✕ ▸This document requires 'TrustedHTML' assignment.
✕ ▸Uncaught TypeError: Failed to set the 'innerHTML' property on 'Element': This document requires 'TrustedHTML' assignment.
    at <anonymous>:1:25
```

```
const examplePolicy = trustedTypes.createPolicy('example-policy', {
      createHTML: (data) => {
         ... //Convert a string to TrustedHTML
      }
});
elem.innerHTML = examplePolicy.createHTML(taintedData)
```

```
elem.innerHTML = DOMPurify.sanitize(taintedData, {RETURN_TRUSTED_TYPE: true})
```

Trusted Types - the default policy

```
Content-Security-Policy: enforce-trusted-types-for 'script'
```

```
trustedTypes.createPolicy('default', {
      createHTML: (data) => DOMPurify.sanitize(data)
});
elem.innerHTML = taintedData; //automatically applies the default policy
```

Hat tip to @PhilippeDeRyck, @kkotowicz and the DOMPurify team.


Real world example: Tink

Tink

Crypto library from Google for Java, Go, C++, Obj-C and Python

Carefully engineered to avoid common pitfalls

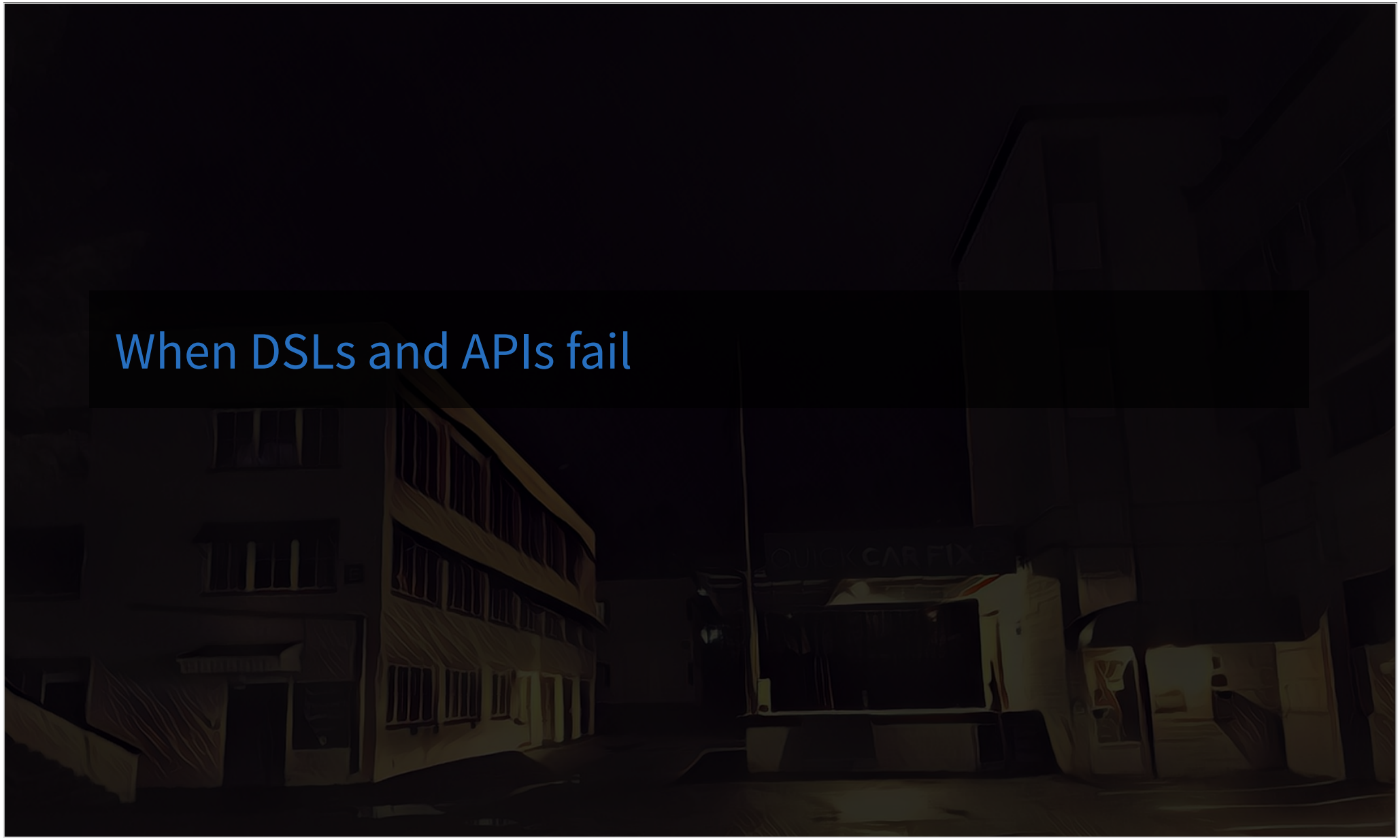
Uses the type system to guide/force developers into making better choices

Deliberate naming to simplify searching for insecure use of the library

Examples for Java:

`CleartextKeysetHandle`

`InsecureSecretKeyAccess`


When DSLs and APIs fail

What's wrong?

```
public class EscapeUtils {
  public static final String[] BAD_CHARS = new String[] { "\"", "'", "`" };
  /**
      Escape strings so we cannot break out of Javascript strings
  */
  public static String escapeJavascript(String data) {
      String result = data;
      for (String c : BAD_CHARS) {
         result = result.replace(c, "\\" + c);
      }
      return result;
  }
   ...
}
```

Fixed escaping of backslashes in JavaScriptHelper#escape_javascript (c...

Browse files

…loses #6302) [sven@c3d2.de]

git-svn-id: http://svn-commit.rubyonrails.org/rails/trunk@5242 5ecf4fe2-1ee6-0310-87b1-e25e094e27de

master v5.2.1 ··· list

dhh committed on 9 Oct 2006 1 parent 2e766b1 commit 4b3e964a1a9d9b5c4a7925ccb8a0090f869fdca4

Showing **3 changed files** with **4 additions** and **1 deletion**. Unified Split

2 actionpack/CHANGELOG View

```
...  ...  @@ -1,5 +1,7 @@
 1    1    *SVN*
 2    2
      3  + * Fixed escaping of backslashes in JavaScriptHelper#escape_javascript #6302 [sven@c3d2.de]
      4  +
 3    5    * Fixed that some 500 rescues would cause 500's themselves because the response had not yet been generated #6329 [cmse
 4    6
 5    7    * respond_to :html doesn't assume .rhtml.  #6281 [Hampton Catlin]
```

2 actionpack/lib/action_view/helpers/javascript_helper.rb View

```
     @@ -149,7 +149,7 @@ def define_javascript_functions
149  149
150  150          # Escape carrier returns and single and double quotes for JavaScript segments.
151  151          def escape_javascript(javascript)
152       -          (javascript || '').gsub(/\r\n|\n|\r/, "\\n").gsub(/["']/) { |m| "\\#{m}" }
     152  +          (javascript || '').gsub('\\','\0\0').gsub(/\r\n|\n|\r/, "\\n").gsub(/["']/) { |m| "\\#{m}" }
153  153          end
154  154
155  155          # Returns a JavaScript tag with the +content+ inside. Example:
```

https://github.com/rails/rails/commit/4b3e964a1a9d9b5c4a7925ccb8a0090f869fdca4#diff-9b0e2026976c362f18664d99ea058657

Unexpected data types

Express.js and MongoDb

```
POST /login HTTP/1.1
host: victim.com
...

username=admin&password=admin123
```

```
db.users.findOne({ username: "admin", password: "admin123"})
```

```
POST /login HTTP/1.1
host: victim.com
...

username=admin&password[$gte]=a
```

```
db.users.findOne({ username: "admin", password: {$gte: "a"}})
```

Input validation could have saved us in this case.

Parsing XML

```
...
DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
InputSource is = new InputSource(new StringReader(xml));
Document doc = builder.parse(is);
doc.getDocumentElement();
...
```

How do we avoid OWASP Top 10 2017 A4 - XML External Entities (XXE).

Parsing XML - verifying

Isolate the code

```
...
public Document parseXML(String xml) {
   //Disable External entities
   DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
   dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
   dbf.setXIncludeAware(false);

   DocumentBuilder builder = dbf.newDocumentBuilder();
   InputSource is = new InputSource(new StringReader(xml));
   return builder.parse(is);
}
...
```

Add unit tests

```
   ...
   @Test
   public void should_protect_against_XXE() {
      String xml = "<!DOCTYPE foo [ <!ENTITY xxe SYSTEM \"file:///etc/passwd\" > ]><foo>&xxe;</foo>";
      try {
         Document doc = parseXML(xml);
      } catch(SAXParseException e) {
         assertThat(e.getMessage(), containsString("DOCTYPE is disallowed"));
      }
   }
   ...
```

Generic advice: Isolate and verify

1. Isolate the logic into a reusable unit
2. Configure the code in that unit to work in a secure manner
3. Add some unit tests for verification


"Domain primitives and all seem fine, but ..."

"...have you ever used this on a real project?"

"...it must take a lot of time to create all those classes"

"...what if we need to change the validation rules?"

In conclusion

Input validation, encoding escaping etc. are all needed → Defense in depth

Domain Driven Security can help us:

- Put input validation in the right place
- Enforce better data quality
- Stop or hamper many types of attacks
- Weed out bugs earlier

Thank you!

erlend@oftedal.no

@webtonull