ESOF 422:
Advanced Software Engineering: Cyber Practices

Secure by Design (Part 2)

Benefits of DDD, Immutability, Input Validation

Reese Pearsall

Spring 2025

Exam

If you *want* a third exam (finals week)

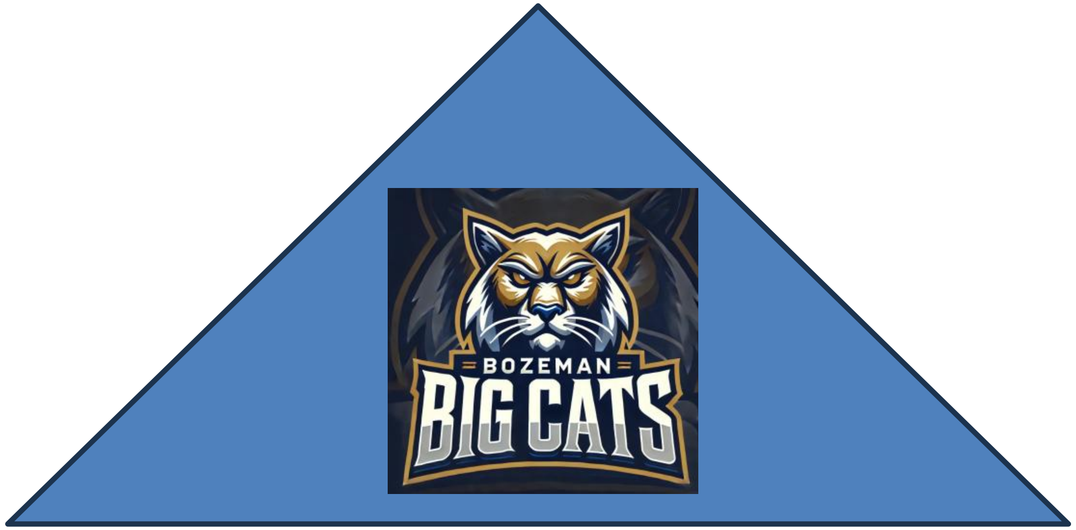
It would be optional

- If you don't take it, the average of your first two exams will be used

https://etc.ch/FTqK

Domain Expert

Doesn't know how to code
Expert in the field
Not concerned about business logic



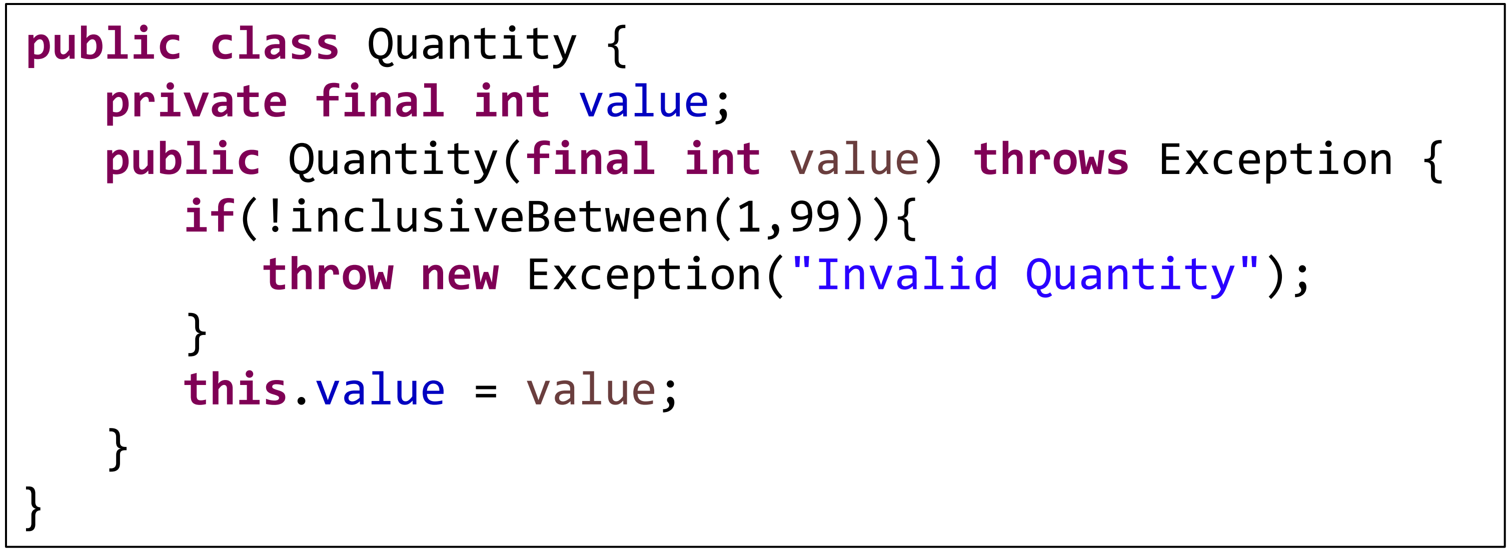
Stakeholder

Doesn't know how to code
Might not know details of the field
Knows business logic

Programmer

Knows how to code
Might not know details of the field
Knows how to implement business logic

Domain-Driven Design: focus of modeling software to match a domain according to input from domain experts. Divide system into bounded contexts (domain primitives), each having their own model with strict contraints



```java
public class Quantity {
   private final int value;
   public Quantity(final int value) throws Exception {
      if(!inclusiveBetween(1,99)){
         throw new Exception("Invalid Quantity");
      }
      this.value = value;
   }
}
```

Domain primitive enforce domain rule validation at creation time

Tightens our design by explicitly stating requirements and assumptions

Deeper modeling

Another important question:

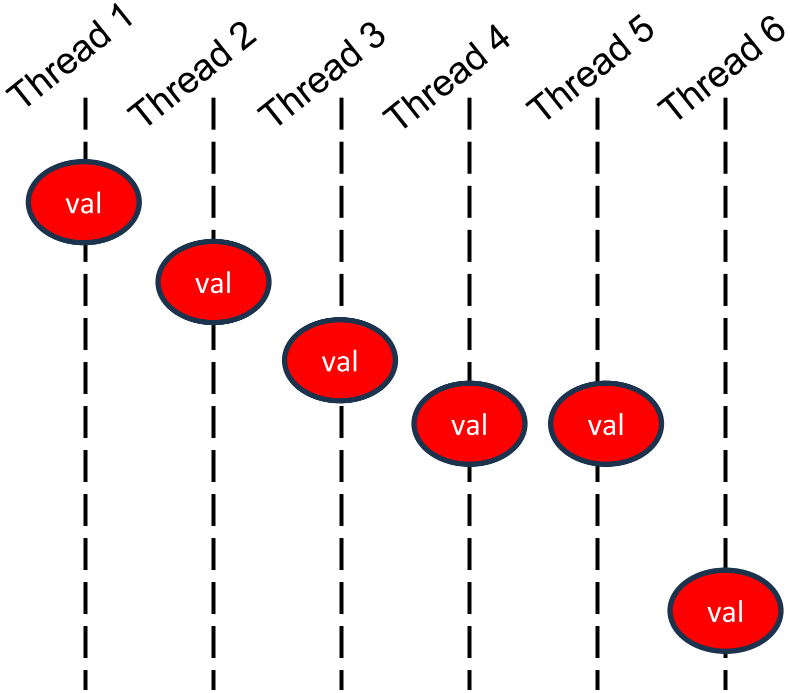
Is the object **mutable** or **immutable** ?

Mutable: allows an object to change (setters are used)
Immutable: object is not allowed to change

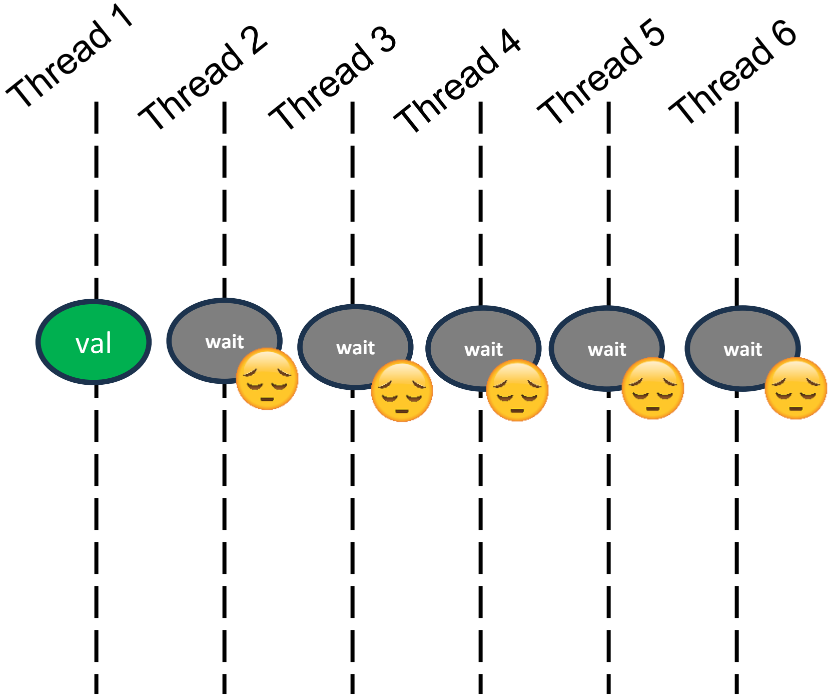
Immutability

Mutable: allows an object to change
Immutable: object is not allowed to change

Immutable objects are safe to share between threads



If an object is mutable, then *thread contention* is a problem (threads will have to wait until the previous thread is done with it)

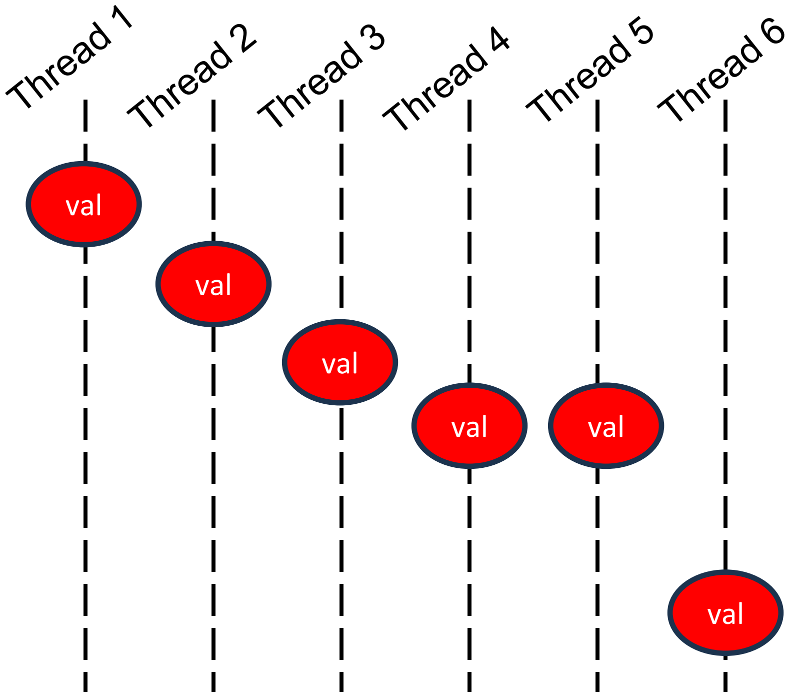


Immutability

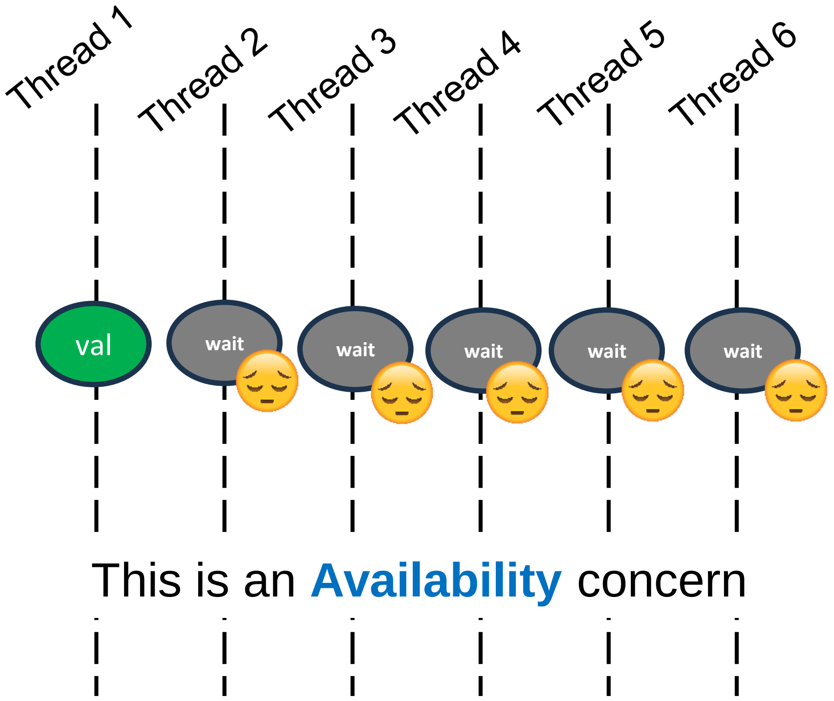
Mutable: allows an object to change
Immutable: object is not allowed to change

CONFIDENTIALITY
INTEGRITY
AVAILABILITY
Data protected by CIA Traid

Immutable objects are safe to share between threads



If an object is **mutable**, then *thread contention* is a problem (threads will have to wait until the previous thread is done with it)



This is an **Availability** concern

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    private Id id;
    private Name name;
    private Order order;
    private CreditScore creditScore;
```

Customers with at least 500 credit score can pay by invoice (good thing). Customers with a credit score less than 500 must pay by credit card (bad thing?)

Credit score is based on payment history, and is a dynamic value

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized void setCreditScore(CreditScore creditScore) {
        this.creditScore = creditScore;
    }

    public synchronized boolean isAcceptedForInvoicePayment() {
        return creditScore.compute() > MIN_INVOICE_SCORE;
    }
}
```

This takes some time to compute…

Synchronized = only one thread is allowed to use method at a time

Many users = threads have to **wait**

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized void setCreditScore(CreditScore creditScore) {
        this.creditScore = creditScore;
    }

    public synchronized boolean isAcceptedForInvoicePayment() {
        return creditScore.compute() > MIN_INVOICE_SCORE;
    }
}
```

Problem	Category	Probable cause
Long waits and poor performance	Availability	System fails to access customer data in a reliable way and times out
Orders timing out at checkout	Availability	The system fails to retrieve necessary data to process the order in a timely fashion

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized CreditScore getCreditScore() {
        return creditScore;
    }

    public synchronized void setCreditScore(CreditScore creditScore) {
        this.creditScore = creditScore;
    }

    public synchronized boolean isAcceptedForInvoicePayment() {
        return creditScore.compute() > MIN_INVOICE_SCORE;
    }
```

Let’s look at another issue with mutable design

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized CreditScore getCreditScore() {
        return creditScore;
    }

    public synchronized void setCreditScore(CreditScore creditScore) {
        this.creditScore = creditScore;
    }

    public synchronized boolean isAcceptedForInvoicePayment() {
        return creditScore.compute() > MIN_INVOICE_SCORE;
    }
```

We expect credit score to only be modified with the setter or the compute() method (synchronized)

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized CreditScore getCreditScore() {
        return creditScore;
    }

    public synchronized void setCreditScore(CreditScore creditScore) {
        this.creditScore = creditScore;
    }

    public synchronized boolean isAcceptedForInvoicePayment() {
        return creditScore.compute() > MIN_INVOICE_SCORE;
    }
```

However, the getCreditScore() method returns a pointer to a mutable object!

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized CreditScore getCreditScore() {
        return creditScore;
    }

    public synchronized void setCreditScore(CreditScore creditScore) {
        this.creditScore = creditScore;
    }

    public synchronized boolean isAcceptedForInvoicePayment() {
        return creditScore.compute() > MIN_INVOICE_SCORE;
    }
```

However, the getCreditScore() method returns a pointer to a mutable object!

This value can be modified outside of the class without requiring a lock! (scary)

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private CreditScore creditScore;

    public synchronized CreditScore getCreditScore() {
        return creditScore;
    }
```

Lessons Learned:

The mutability of an object can impact the availability and integrity of your system

Problem	Category	Probable cause
Long waits and poor performance	Availability	System fails to access customer data in a reliable way and times out
Orders timing out at checkout	Availability	The system fails to retrieve necessary data to process the order in a timely fashion
Inconsistent payment options	Integrity	Credit score can be changed in an illegal way

```
public class Customer {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private final CreditScore creditScore;

    public boolean isAcceptedForInvoicePayment() {
        return creditScore.check();
    }
```

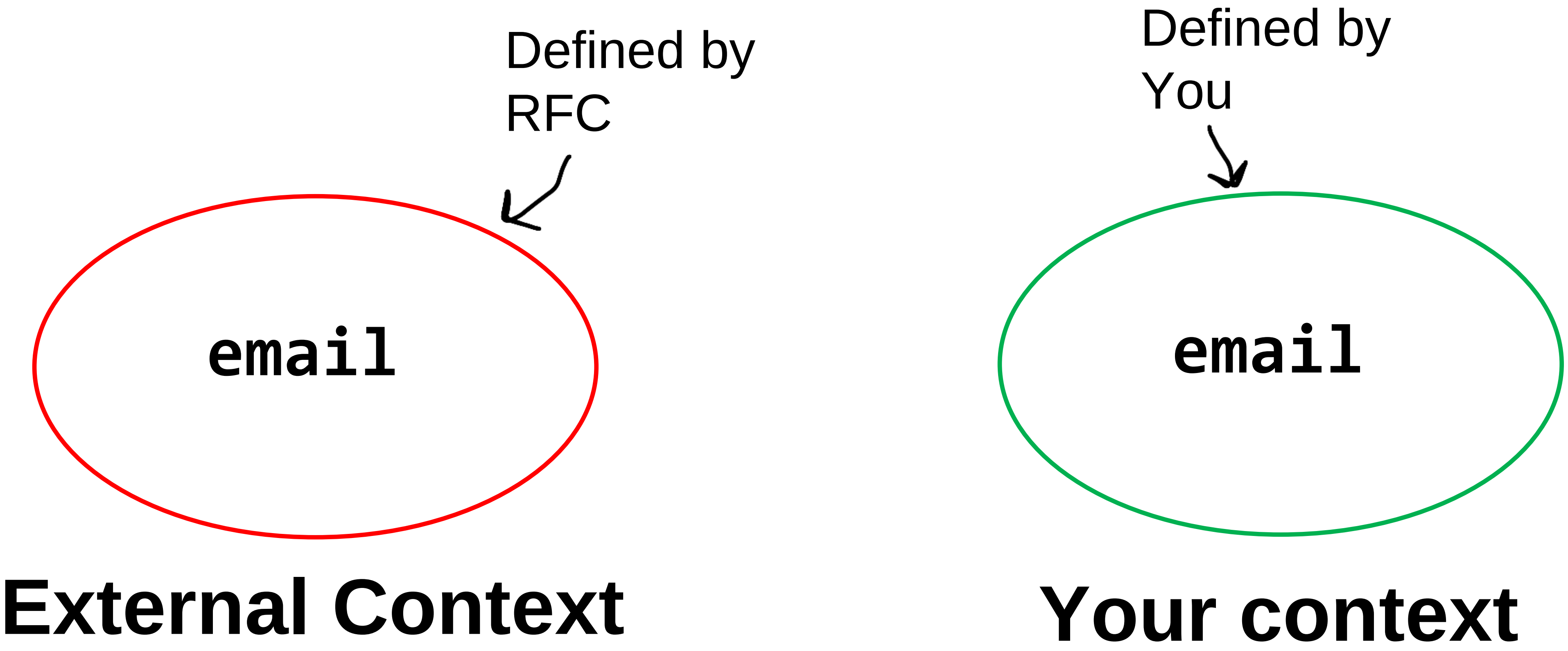
```
public class CreditScore {
    private static final int MIN_INVOICE_SCORE = 500;
    …
    private final int score;

    public CreditScore(final int computedCreditScore){
         //validation checks
         this.score = computedCreditScore;
    }
    public boolean check() {
        return score > MIN_INVOICE_SCORE;
    }
}
```

The **`final`** keyword in Java makes an object immutable

Choosing a design that favors **immutability**, the need for locks and protections against illegal change disappear

How to change Customer Data if its Immutable?
→ See “Entity Snapshot pattern”

External vs Internal Primitive



Defined by RFC

`email`

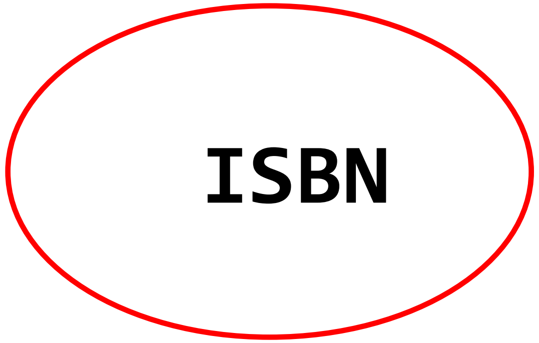
External Context

Defined by You

`email`

Your context

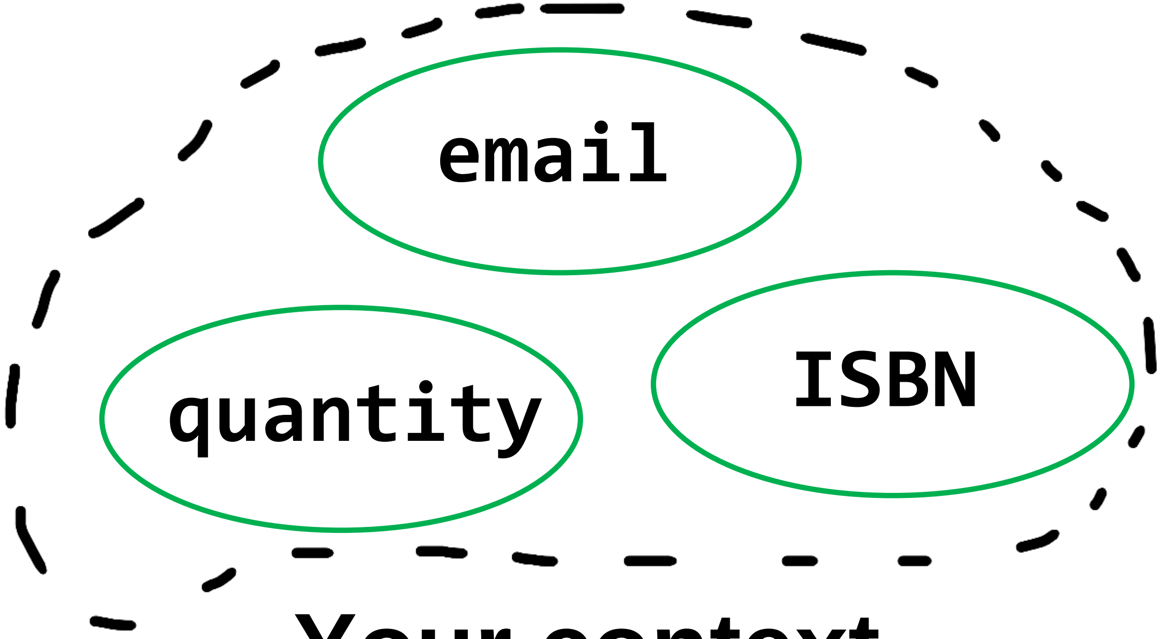
External vs Internal Primitive



ISBN

External Context

Our “library” of domain primitives that don’t rely on external definitions



email

quantity

ISBN

Your context

Passing Primitives

```
public void sendAuditLogsToServerAt(java.net.InetAddress serverAddress) {

}
```

Refactor

By accepting a domain primitive (`InternalAddress`), it ensures the logs will be sent to a valid IP address

- Remember, the existence of a domain primitive means that is has to be valid!

```
public void sendAuditLogsToServerAt(InternalAddress serverAddress) {
      notNull(serverAddress)
}
```

If you are building a public API, you should not reveal domain information

```
class Order {
    private BookRepository bookCatalog;
    private ArrayList<Object> items;
    private boolean paid = false;
    Inventory inventory;

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

            Book book = bookCatalog.findByISBN(isbn);
            if (inventory.availableBooks(isbn) >= qty) {
                items.add(new OrderLine(book, qty));
            }
        }
    }
    ...
}
```

```java
class Order {
    private BookRepository bookCatalog;
    private ArrayList<Object> items;
    private boolean paid = false;
    Inventory inventory;

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

            Book book = bookCatalog.findByISBN(isbn);
            if (inventory.availableBooks(isbn) >= qty) {
                items.add(new OrderLine(book, qty));
            }
        }
    }
    ...
}
```

This method does not treat ISBN and Quantity as domain primitives

It does validation checking on a few different things

It is missing checks for negative values!

```
class Order {
    private BookRepository bookCatalog;
    private ArrayList<Object> items;
    private boolean paid = false;
    Inventory inventory;

    public void addItem(String isbn, int qty) {
        if (this.paid == false) {
            notNull(isbn);
            isTrue(isbn.length() == 10);
            isTrue(isbn.matches("[0-9X]+"));
            isTrue(isbn.matches("[0-9]{9}[0-9X]"));
            isNotNegative(qty)
            isLessThan(99)
            Book book = bookCatalog.findByISBN(isbn);
            if (inventory.availableBooks(isbn) >= qty) {
                items.add(new OrderLine(book, qty));
            }
        }
    }
    ...
}
```

If a method does validation checks on several different arguments, this method can become **cluttered**, which increases the chances of missing something

Tip: Leave domain validation to domain primitives

```
class Order {
    private BookRepository bookCatalog;
    private ArrayList<Object> items;
    private boolean paid = false;
    Inventory inventory;

    public void addItem(ISBN isbn, Quantity qty) {
        notNull(isbn);
        notNull(qty);
        if (this.paid == false) {
            Book book = bookCatalog.findByISBN(isbn);
            if (inventory.availableBooks(isbn) >= qty) {
                items.add(new OrderLine(book, qty));
            }
        }
    }
    ...
}
```

```
Class ISBN {
    private final String value;
    public ISBN(String isbn){
        //validation checks here
        value = isbn
    }
}
```

```
Class Quantity {
    private final int value;
    public ISBN(int q){
        //validation checks here
        value = q
    }
}
```

isbn and **qty** are already valid before entering method

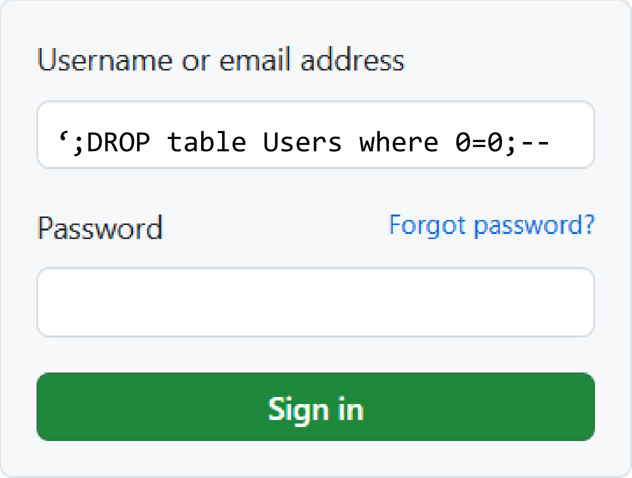
Validation only needs to happen in the domain primitive class, resulting in less cluttering, and less chance of missing something

Input Checking

The **attack surface** of your system will usually always include areas of **untrusted user input**

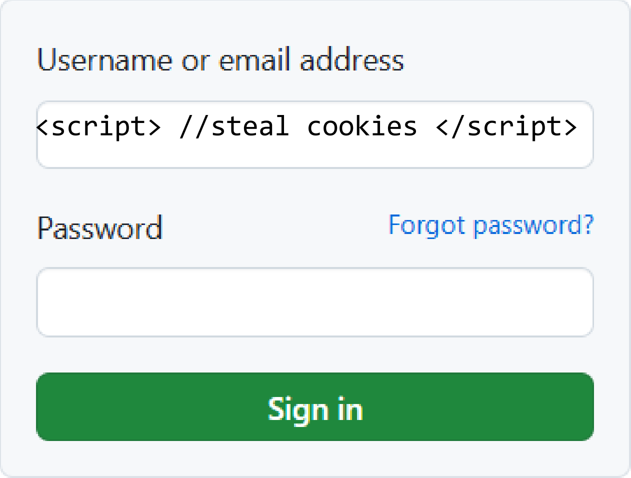
The most severe vulnerabilities are usually due to lack of input checking or input validation

SQL Injections



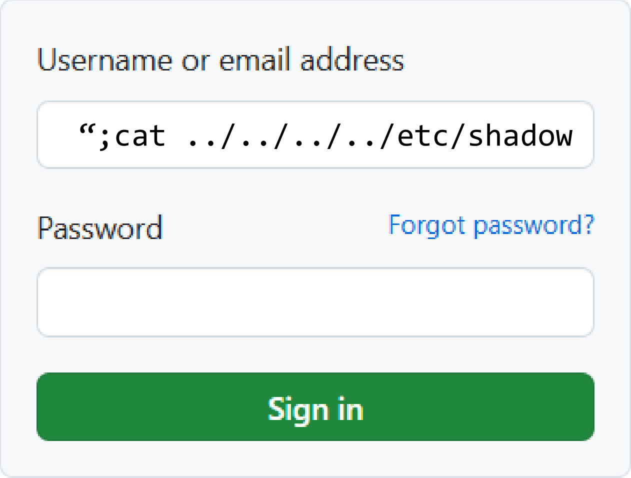
```
‘;DROP table Users where 0=0;--
```

XSS Attacks



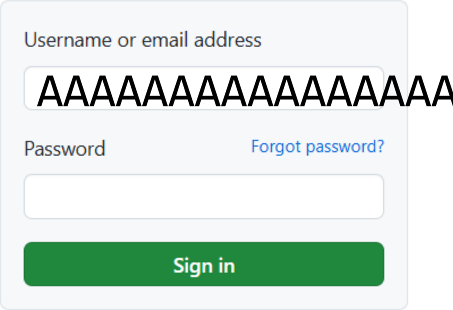
```
<script> //steal cookies </script>
```

Path Traversal



```
“;cat ../../../../etc/shadow
```

Availability Attacks



AAAAAAAAAAAAAAA

Input Checking

Is this a “valid” input?

```
‘;DROP table Users where 0=0;--
```

99.9% of the time, this isn’t valid

Bad

Unexpected inputs → Unexpected Behaviors → Potential Vulnerabilities

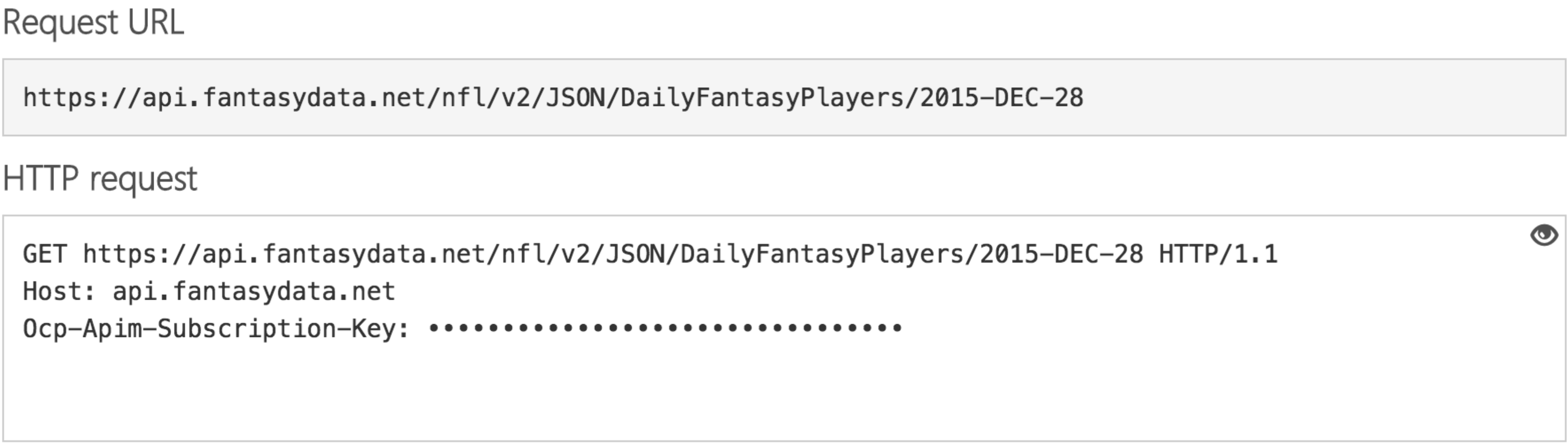
Good

Unexpected inputs → Input Validation → Only expected behaviors → More secure

Steps of input validation

1. **Origin** – Is the data from a legitimate sender?

   - Check source IP address (Whitelists, Cloud configuration)
   - Require use of API key or Access Token



Request URL

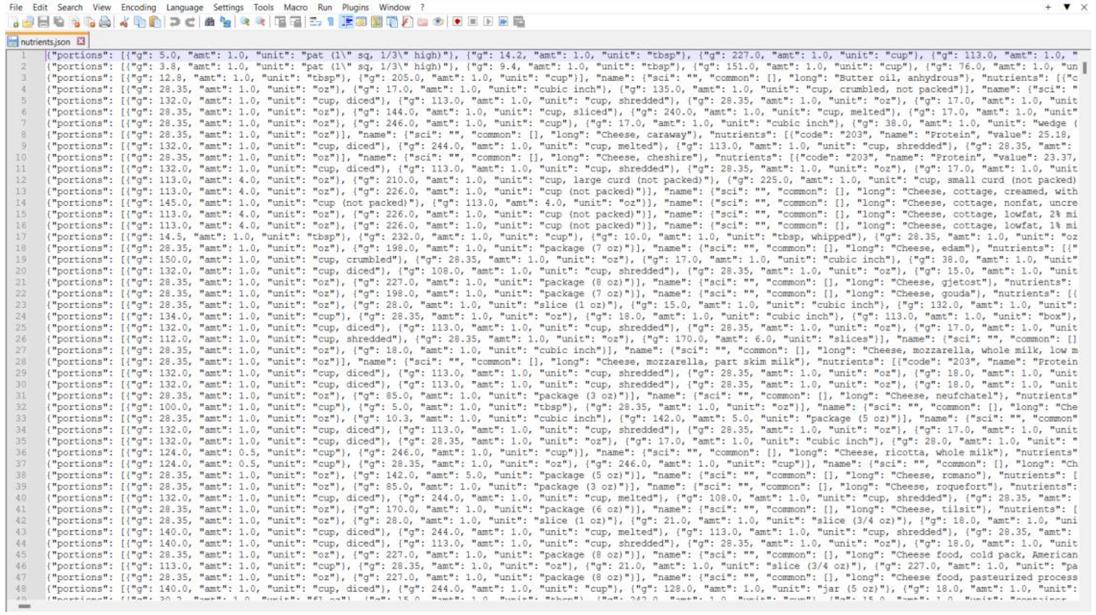
```
https://api.fantasydata.net/nfl/v2/JSON/DailyFantasyPlayers/2015-DEC-28
```

HTTP request

```
GET https://api.fantasydata.net/nfl/v2/JSON/DailyFantasyPlayers/2015-DEC-28 HTTP/1.1
Host: api.fantasydata.net
Ocp-Apim-Subscription-Key: ••••••••••••••••••••••••••••••••
```

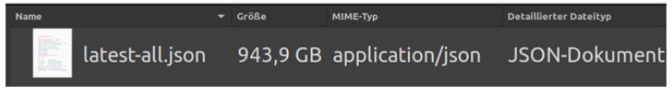
Steps of input validation

1. **Origin** – Is the data from a legitimate sender?
2. **Size** – Is the input reasonably large?
   - HTTP **`Content-Length`** header
   - Is 10 MB of JSON reasonable? What about 10 GB of JSON?



```
{"portions": [{"g": 5.0, "amt": 1.0, "unit": "pat (1\" sq, 1/3\" high)"}, {"g": 14.2, "amt": 1.0, "unit": "tbsp"}, {"g": 227.0, "amt": 1.0, "unit": "cup"}, {"g": 113.0, "amt": 1.0, "
{"portions": [{"g": 3.8, "amt": 1.0, "unit": "pat (1\" sq, 1/3\" high)"}, {"g": 9.4, "amt": 1.0, "unit": "tbsp"}, {"g": 151.0, "amt": 1.0, "unit": "cup"}, {"g": 76.0, "amt": 1.0, "un
{"portions": [{"g": 12.8, "amt": 1.0, "unit": "tbsp"}, {"g": 205.0, "amt": 1.0, "unit": "cup"}], "name": {"sci": "", "common": [], "long": "Butter oil, anhydrous"}, "nutrients": [{"c
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit": "cubic inch"}, {"g": 135.0, "amt": 1.0, "unit": "cup, crumbled, not packed"}], "name": {"sci": "
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 144.0, "amt": 1.0, "unit": "cup, sliced"}, {"g": 240.0, "amt": 1.0, "unit": "cup, melted"}, {"g": 17.0, "amt": 1.0, "unit"
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 246.0, "amt": 1.0, "unit": "cup"}, {"g": 17.0, "amt": 1.0, "unit": "cubic inch"}, {"g": 38.0, "amt": 1.0, "unit": "wedge (
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}], "name": {"sci": "", "common": [], "long": "Cheese, caraway"}, "nutrients": [{"code": "203", "name": "Protein", "value": 25.18,
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 244.0, "amt": 1.0, "unit": "cup, melted"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt":
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}], "name": {"sci": "", "common": [], "long": "Cheese, cheshire"}, "nutrients": [{"code": "203", "name": "Protein", "value": 23.37,
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit
{"portions": [{"g": 113.0, "amt": 4.0, "unit": "oz"}, {"g": 210.0, "amt": 1.0, "unit": "cup, large curd (not packed)"}, {"g": 225.0, "amt": 1.0, "unit": "cup, small curd (not packed)
{"portions": [{"g": 113.0, "amt": 4.0, "unit": "oz"}, {"g": 226.0, "amt": 1.0, "unit": "cup (not packed)"}], "name": {"sci": "", "common": [], "long": "Cheese, cottage, creamed, with
{"portions": [{"g": 145.0, "amt": 1.0, "unit": "cup (not packed)"}, {"g": 113.0, "amt": 4.0, "unit": "oz"}], "name": {"sci": "", "common": [], "long": "Cheese, cottage, nonfat, uncre
{"portions": [{"g": 113.0, "amt": 4.0, "unit": "oz"}, {"g": 226.0, "amt": 1.0, "unit": "cup (not packed)"}], "name": {"sci": "", "common": [], "long": "Cheese, cottage, lowfat, 2% mi
{"portions": [{"g": 113.0, "amt": 4.0, "unit": "oz"}, {"g": 226.0, "amt": 1.0, "unit": "cup (not packed)"}], "name": {"sci": "", "common": [], "long": "Cheese, cottage, lowfat, 1% mi
{"portions": [{"g": 14.5, "amt": 1.0, "unit": "tbsp"}, {"g": 232.0, "amt": 1.0, "unit": "cup"}, {"g": 10.0, "amt": 1.0, "unit": "tbsp, whipped"}, {"g": 28.35, "amt": 1.0, "unit": "oz
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 198.0, "amt": 1.0, "unit": "package (7 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, edam"}, "nutrients": [{"
{"portions": [{"g": 150.0, "amt": 1.0, "unit": "cup, crumbled"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit": "cubic inch"}, {"g": 38.0, "amt": 1.0, "unit"
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 108.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 15.0, "amt": 1.0, "unit
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 227.0, "amt": 1.0, "unit": "package (8 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, gjetost"}, "nutrients":
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 198.0, "amt": 1.0, "unit": "package (7 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, gouda"}, "nutrients": [{
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 28.0, "amt": 1.0, "unit": "slice (1 oz)"}, {"g": 15.0, "amt": 1.0, "unit": "cubic inch"}, {"g": 132.0, "amt": 1.0, "unit":
{"portions": [{"g": 134.0, "amt": 1.0, "unit": "cup"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 18.0, "amt": 1.0, "unit": "cubic inch"}, {"g": 113.0, "amt": 1.0, "unit": "box"},
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit
{"portions": [{"g": 112.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 170.0, "amt": 6.0, "unit": "slices"}], "name": {"sci": "", "common": []
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 18.0, "amt": 1.0, "unit": "cubic inch"}], "name": {"sci": "", "common": [], "long": "Cheese, mozzarella, whole milk, low m
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}], "name": {"sci": "", "common": [], "long": "Cheese, mozzarella, part skim milk"}, "nutrients": [{"code": "203", "name": "Protein
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 18.0, "amt": 1.0, "unit
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 18.0, "amt": 1.0, "unit
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 85.0, "amt": 1.0, "unit": "package (3 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, neufchatel"}, "nutrients"
{"portions": [{"g": 100.0, "amt": 1.0, "unit": "cup"}, {"g": 5.0, "amt": 1.0, "unit": "tbsp"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}], "name": {"sci": "", "common": [], "long": "Che
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 10.3, "amt": 1.0, "unit": "cubic inch"}, {"g": 142.0, "amt": 5.0, "unit": "package (5 oz)"}], "name": {"sci": "", "common"
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 17.0, "amt": 1.0, "unit": "cubic inch"}, {"g": 28.0, "amt": 1.0, "unit": "
{"portions": [{"g": 124.0, "amt": 0.5, "unit": "cup"}, {"g": 246.0, "amt": 1.0, "unit": "cup"}], "name": {"sci": "", "common": [], "long": "Cheese, ricotta, whole milk"}, "nutrients"
{"portions": [{"g": 124.0, "amt": 0.5, "unit": "cup"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 246.0, "amt": 1.0, "unit": "cup"}], "name": {"sci": "", "common": [], "long": "Ch
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 142.0, "amt": 5.0, "unit": "package (5 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, romano"}, "nutrients": [
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 85.0, "amt": 1.0, "unit": "package (3 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, roquefort"}, "nutrients":
{"portions": [{"g": 132.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 244.0, "amt": 1.0, "unit": "cup, melted"}, {"g": 108.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt":
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 170.0, "amt": 1.0, "unit": "package (6 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese, tilsit"}, "nutrients": [
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 28.0, "amt": 1.0, "unit": "slice (1 oz)"}, {"g": 21.0, "amt": 1.0, "unit": "slice (3/4 oz)"}, {"g": 18.0, "amt": 1.0, "uni
{"portions": [{"g": 140.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 244.0, "amt": 1.0, "unit": "cup, melted"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt":
{"portions": [{"g": 140.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 113.0, "amt": 1.0, "unit": "cup, shredded"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 18.0, "amt": 1.0, "unit
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 227.0, "amt": 1.0, "unit": "package (8 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese food, cold pack, American
{"portions": [{"g": 113.0, "amt": 1.0, "unit": "cup"}, {"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 21.0, "amt": 1.0, "unit": "slice (3/4 oz)"}, {"g": 227.0, "amt": 1.0, "unit": "pa
{"portions": [{"g": 28.35, "amt": 1.0, "unit": "oz"}, {"g": 227.0, "amt": 1.0, "unit": "package (8 oz)"}], "name": {"sci": "", "common": [], "long": "Cheese food, pasteurized process
{"portions": [{"g": 140.0, "amt": 1.0, "unit": "cup, diced"}, {"g": 244.0, "amt": 1.0, "unit": "cup"}, {"g": 128.0, "amt": 1.0, "unit": "jar (5 oz)"}, {"g": 18.0, "amt": 1.0, "unit":
{"portions": [{"g": 30.2, "amt": 1.0, "unit": "fl oz"}, {"g": 15.0, "amt": 1.0, "unit": "tbsp"}, {"g": 242.0, "amt": 1.0, "unit": "cup"}, {"g": 15.0, "amt": 1.0, "unit": "container
```

Everybody Gangsta until the json File is 900GB



Name	Größe	MIME-Typ	Detaillierter Dateityp
latest-all.json	943,9 GB	application/json	JSON-Dokument

Steps of input validation

1. **Origin** – Is the data from a legitimate sender?
2. **Size** – Is the input reasonably large?
3. **Lexical Content** – Does it contain the right characters and encoding ?

`; , < > --` Have special meanings in programming languages, which often makes them part of an attack. Do those characters make sense in an email address input box?

```
isTrue(isbn.matches(“[0-9x]*”));
```

Filter out or encode special characters

`<script>` → `&lt;script&gt;`

Steps of input validation

1. **Origin** – Is the data from a legitimate sender?
2. **Size** – Is the input reasonably large?
3. **Lexical Content** – Does it contain the right characters and encoding ?
4. **Syntax**– Is the format right?

XML – Do all opening tags have an ending tag?
JSON – Are key value pairs correctly defined and follow correct JSON syntax ?
HTTP – Does it contain the HTTP method? Does it have the necessary headers?

Steps of input validation

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

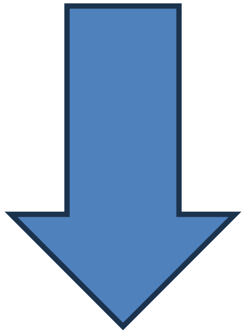
Are the actual input values *valid* for the domain ?

If input is selecting an item to add to cart, does the item actually exist?

Taint Analysis

Tainted code analysis looks at the flow of potentially tainted input and flags potentially malicious before it is used

```
Username: input1     Password: input2       Email: input3
```

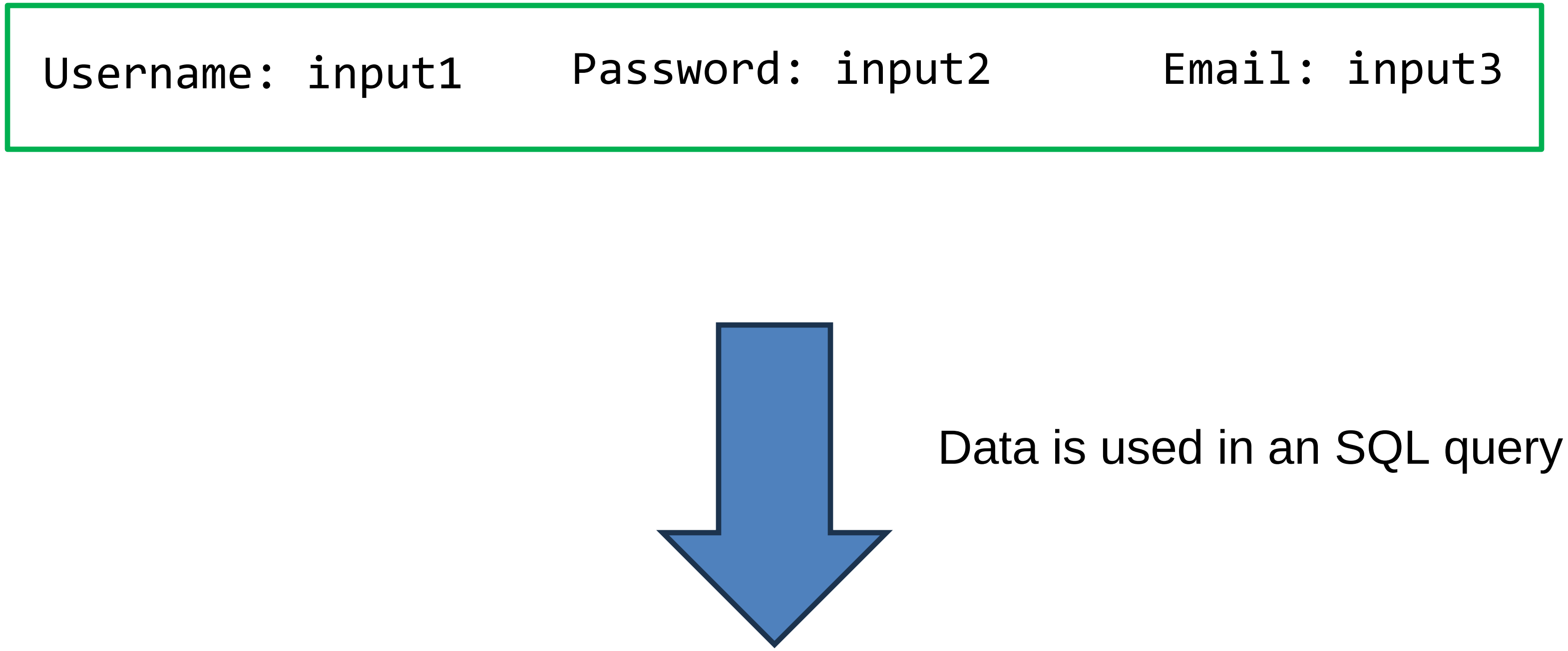


Data is used in an SQL query

```
SELECT * WHERE Username==input1 and Password==input2 and Email == input3
```

Taint Analysis

Tainted code analysis looks at the flow of potentially tainted input and flags potentially malicious before it is used



```
Username: input1    Password: input2     Email: input3
```

These are the **sources**

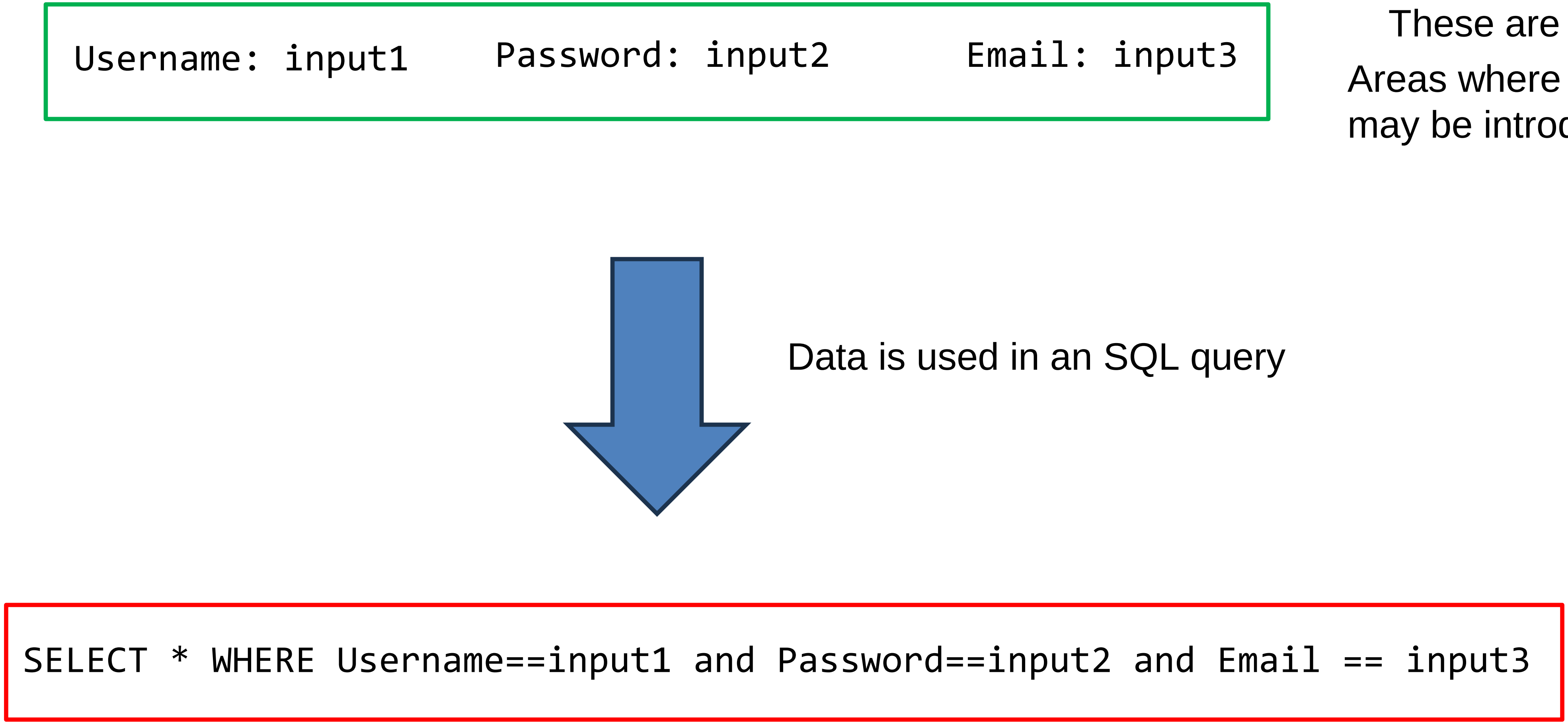
Areas where malicious code may be introduced

Data is used in an SQL query

```
SELECT * WHERE Username==input1 and Password==input2 and Email == input3
```

Taint Analysis

Tainted code analysis looks at the flow of potentially tainted input and flags potentially malicious before it is used



```
Username: input1     Password: input2      Email: input3
```

These are the **sources**

Areas where malicious code may be introduced

Data is used in an SQL query

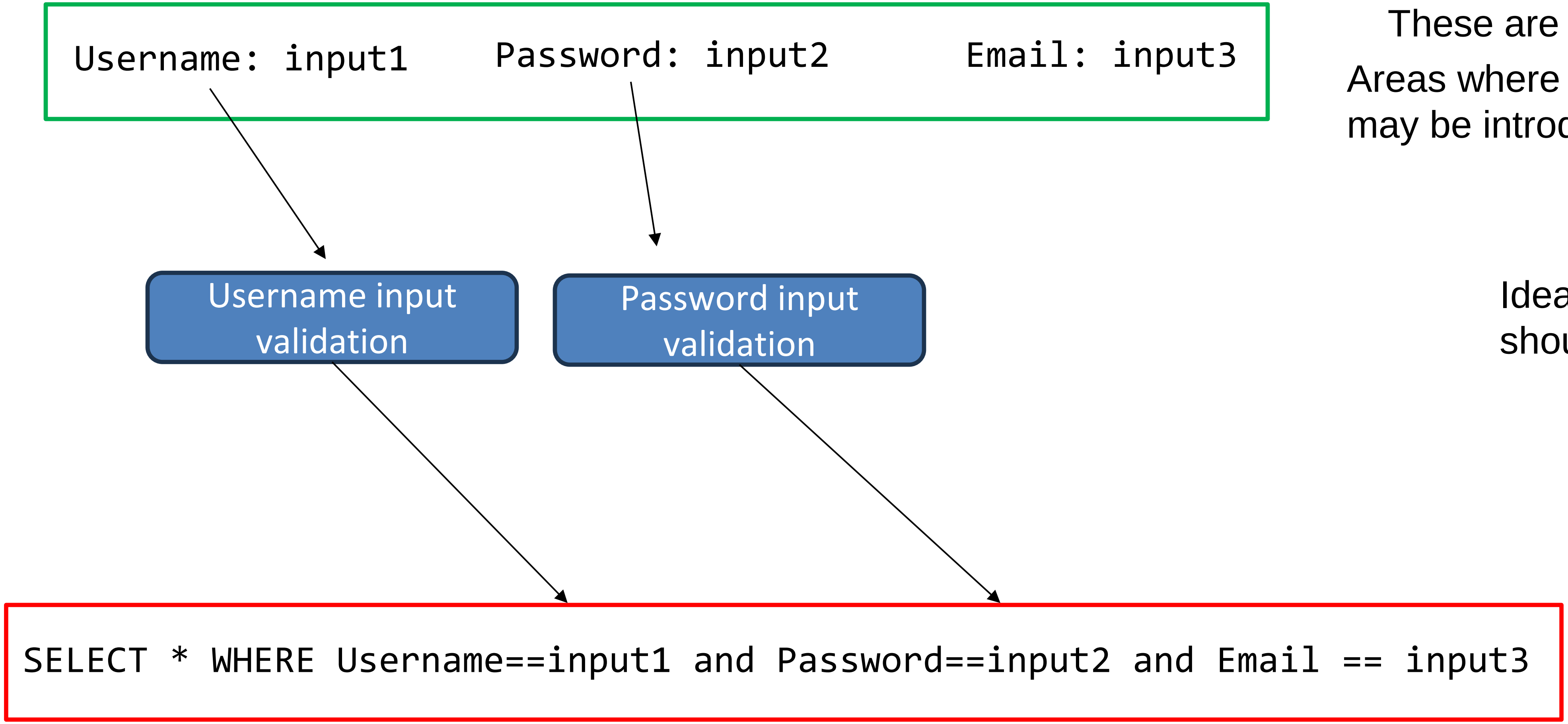
```
SELECT * WHERE Username==input1 and Password==input2 and Email == input3
```

These is a **sink**

Areas where input is used in application

Taint Analysis

Tainted code analysis looks at the flow of potentially tainted input and flags potentially malicious before it is used



```
Username: input1     Password: input2       Email: input3
```

Username input validation

Password input validation

```
SELECT * WHERE Username==input1 and Password==input2 and Email == input3
```

These are the **sources**

Areas where malicious code may be introduced

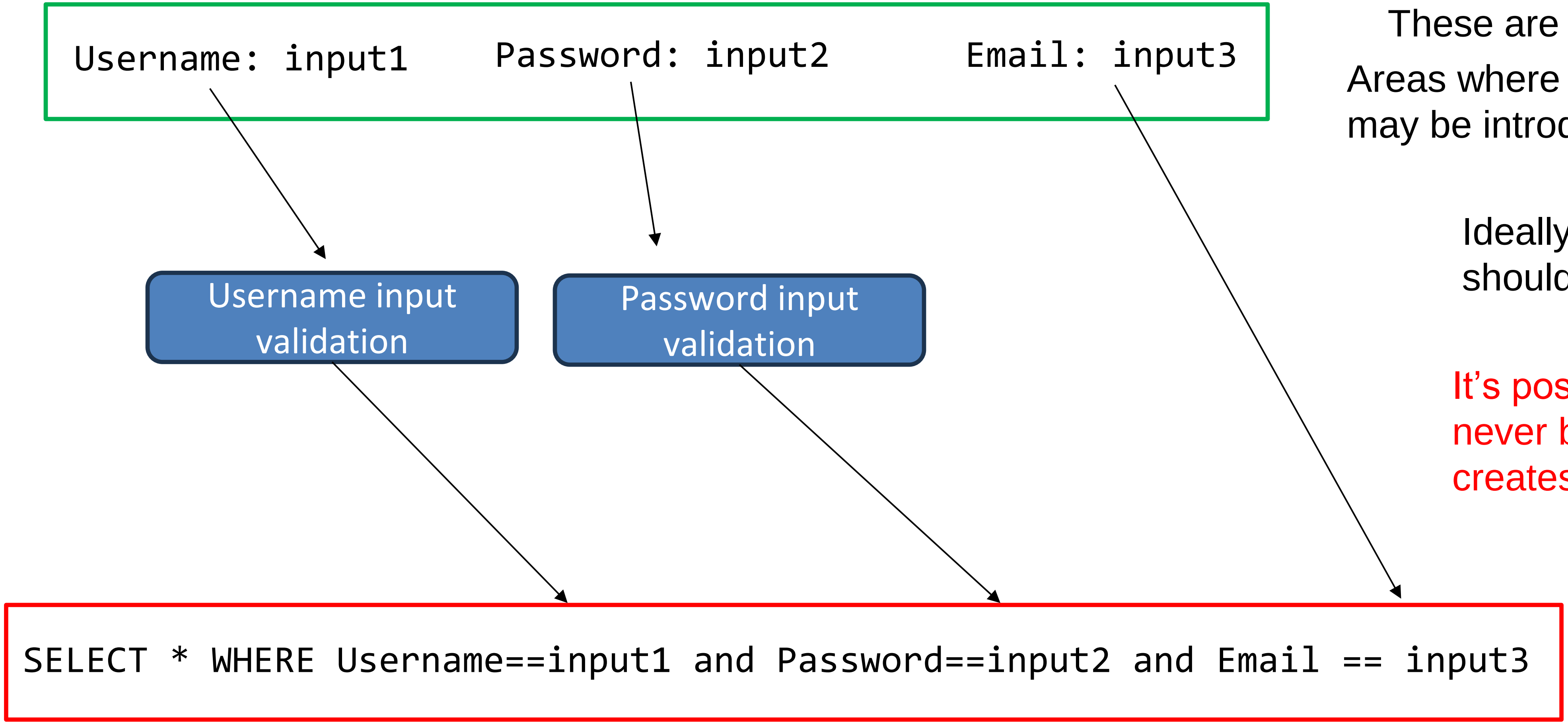
Ideally, all user inputs should be **sanitized**

These is a **sink**

Areas where input is used in application

Taint Analysis

Tainted code analysis looks at the flow of potentially tainted input and flags potentially malicious before it is used



Username: input1   Password: input2   Email: input3

Username input validation

Password input validation

```
SELECT * WHERE Username==input1 and Password==input2 and Email == input3
```

These are the **sources**

Areas where malicious code may be introduced

Ideally, all user inputs should be **sanitized**

It’s possible an input could never be sanitized, which creates security concern

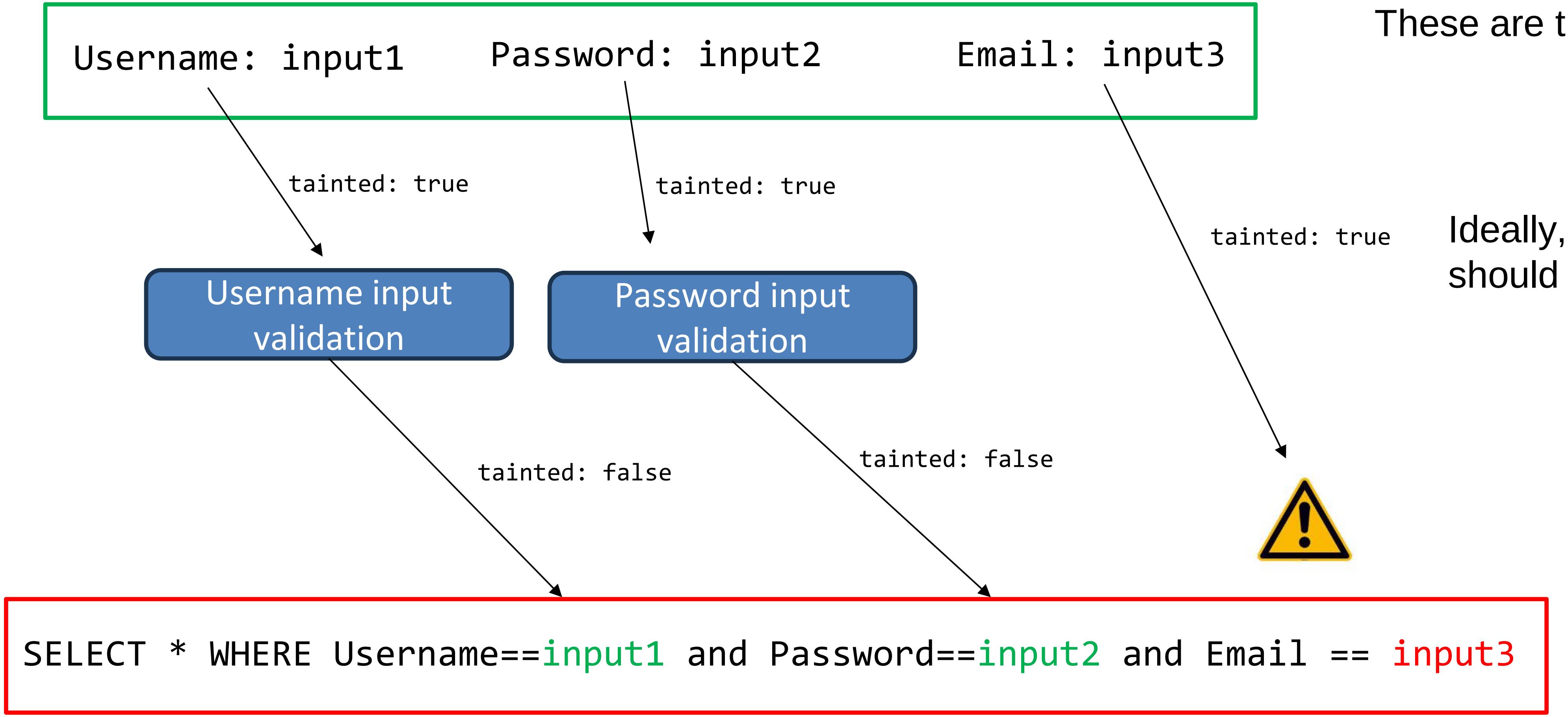
These is a **sink**

Areas where input is used in application

Taint Analysis

Tainted code analysis looks at the flow of potentially tainted input and flags potentially malicious before it is used

Taint analysis keeps track of the flow of user input data, and makes sure all sources are sanitized



```
SELECT * WHERE Username==input1 and Password==input2 and Email == input3
```

These are the **sources**

Ideally, all user inputs should be **sanitized**

These is a **sink**
Areas where input is used in application