Secure & Robust & Rugged Coding Patterns

@andhallberg

TrueSec

Trust

Domain-Driven Security

The Untrusted Pattern

Immutability

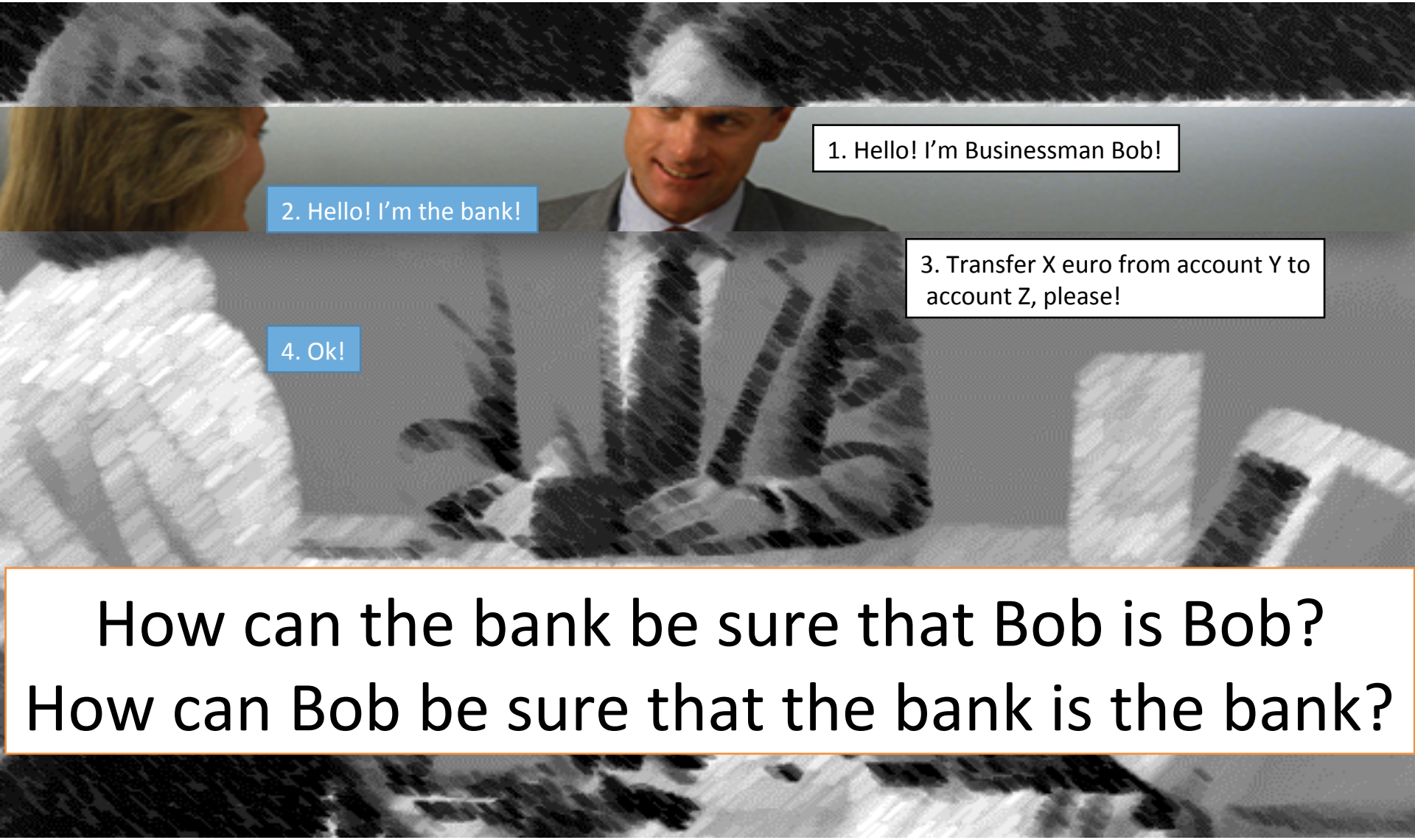
The Inverse Life Coach Pattern

Trust

The foundation of software security


1. Hello! I’m Businessman Bob!
2. Hello! I’m the bank!
3. Transfer X euro from account Y to account Z, please!
4. Ok!

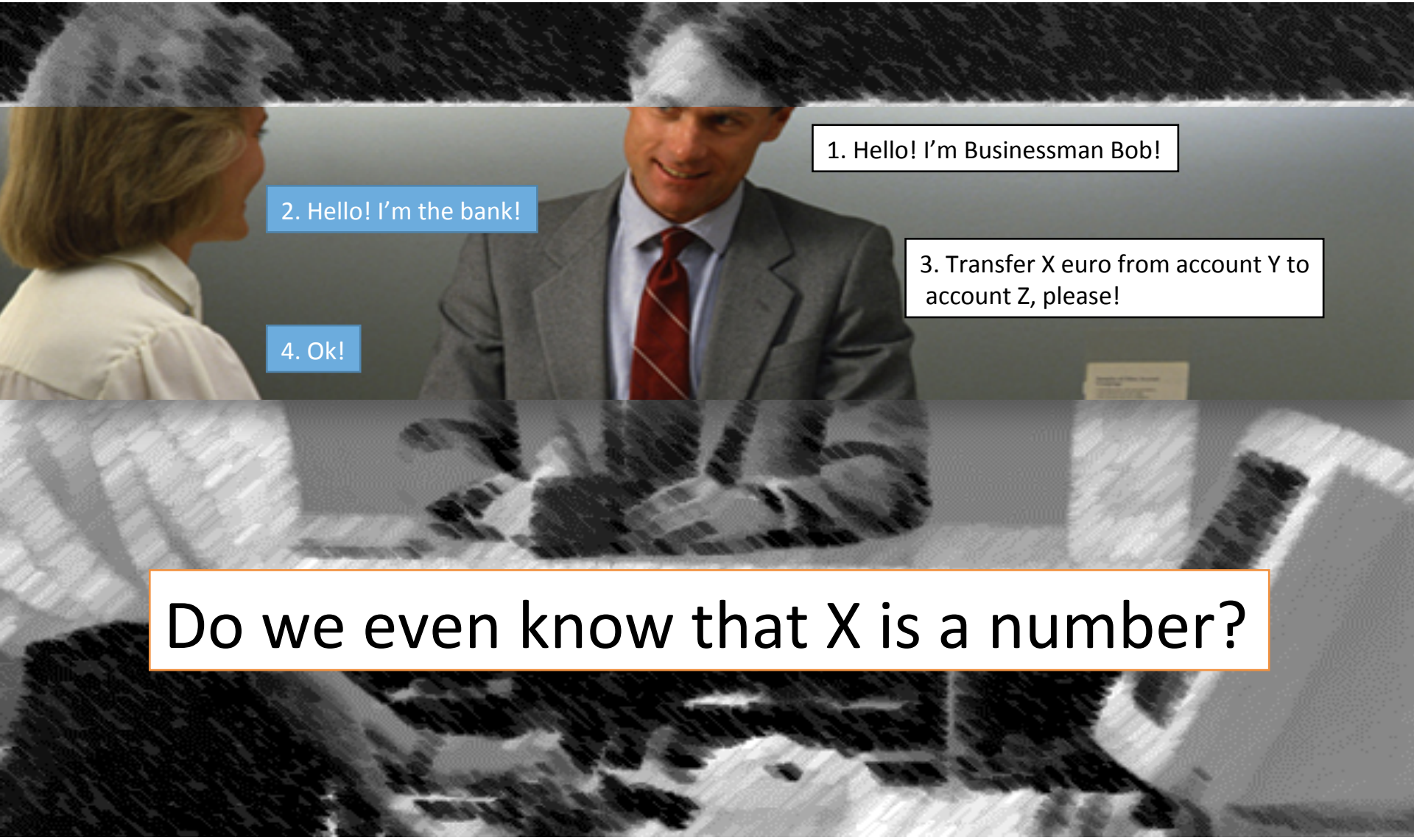
What might go wrong?

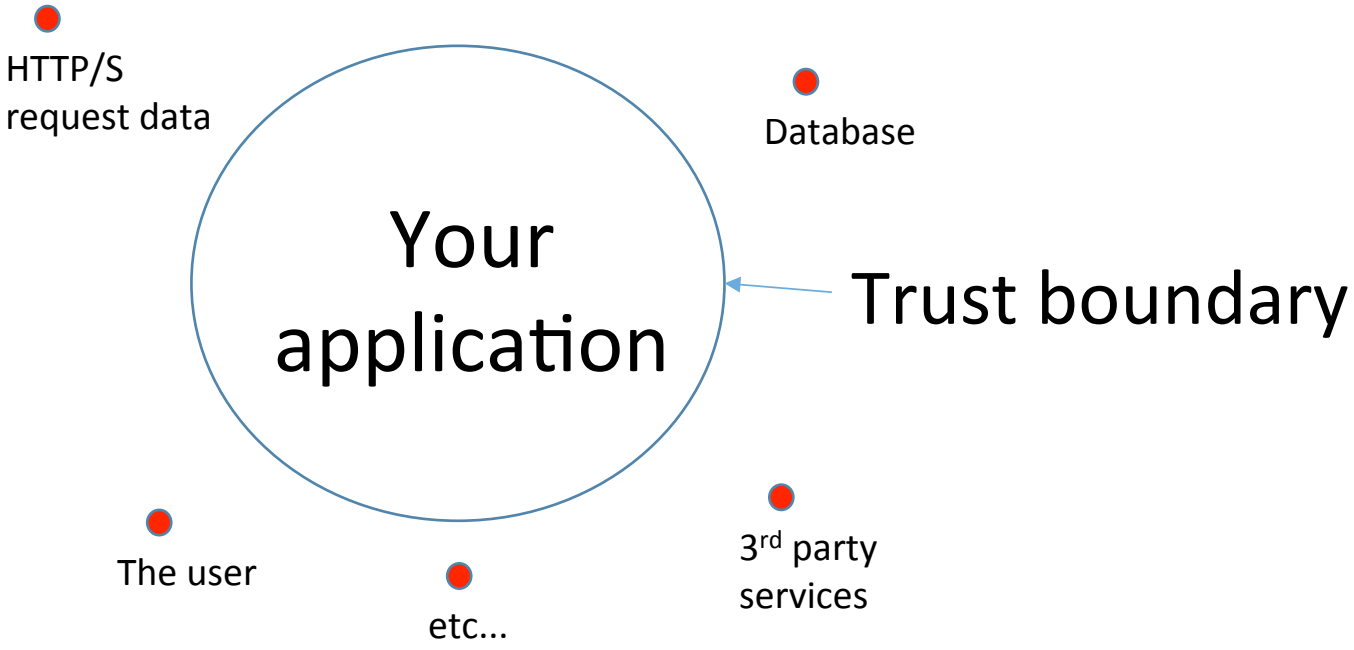


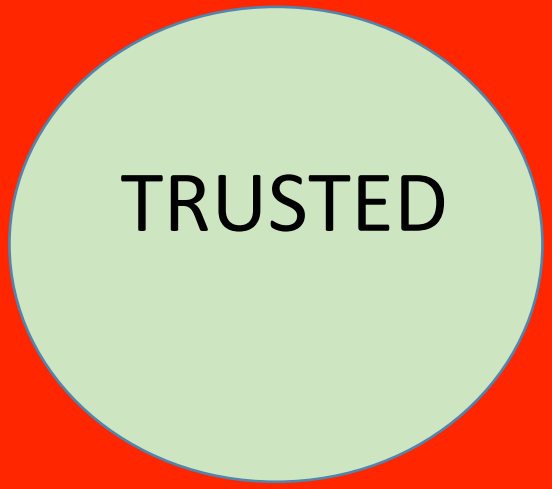
How can the bank be sure that Bob is Bob?
How can Bob be sure that the bank is the bank?


1. Hello! I’m Businessman Bob!
2. Hello! I’m the bank!
3. Transfer X euro from account Y to account Z, please!
4. Ok!
Do we know that Bob owns account Y?

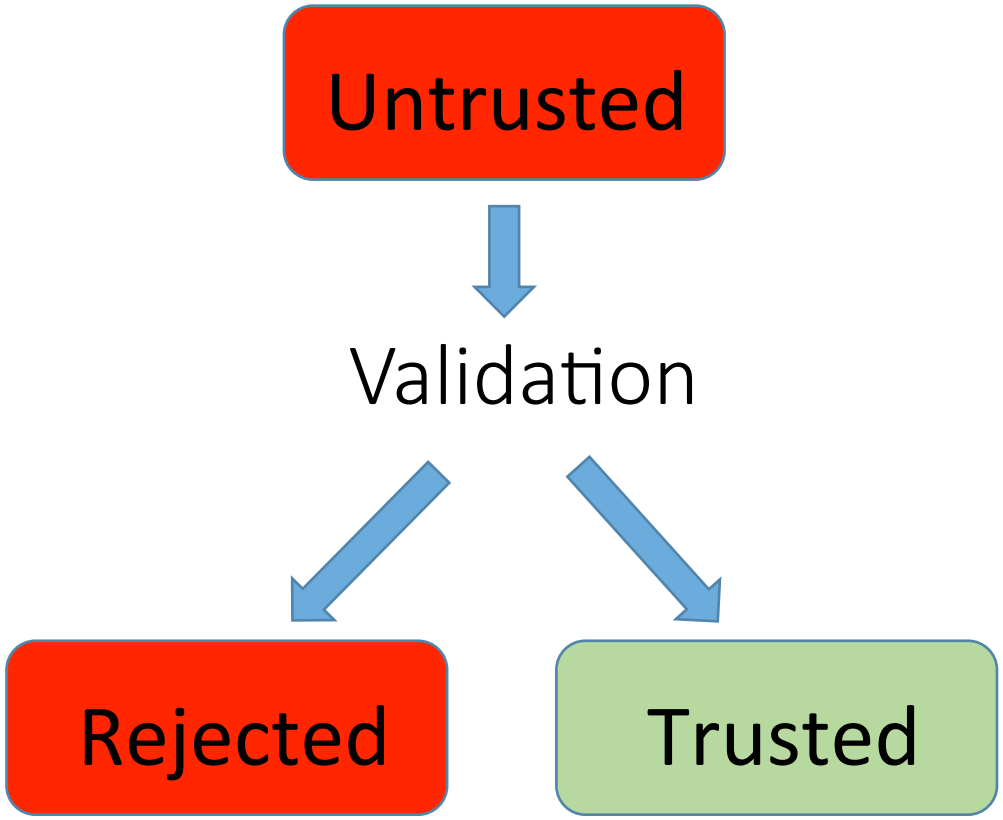

1. Hello! I’m Businessman Bob!
2. Hello! I’m the bank!
3. Transfer X euro from account Y to account Z, please!
4. Ok!
Do we know that account Y holds X euro?


1. Hello! I’m Businessman Bob!
2. Hello! I’m the bank!
3. Transfer X euro from account Y to account Z, please!
4. Ok!
Do we even know that X is a number?


HTTP/S
request data
Database
Your
application
Trust boundary
The user
etc...
3rd party
services



UNTRUSTED


Untrusted
Validation
Rejected
Trusted

Validation and friends

- Validation
  - Making sure data is valid in the domain

    I can't transfer amount "a" or -1
- Canonicalization and/or normalization
  - Must happen *before* validation!

    c:\public\fileupload\..\..\secrets\keys => c:\secrets\key
- Sanitization
  - Clean up dangerous/unknown data

    log injection

Validation, cont.

- **Always** prefer whitelisting over blacklisting
  - It's easier to figure out what's valid over what's not valid
- Strict validation finds bugs early!

Ask yourself…

What is the minimal acceptable range for this parameter?

Don’t accept any more than that!

Trust

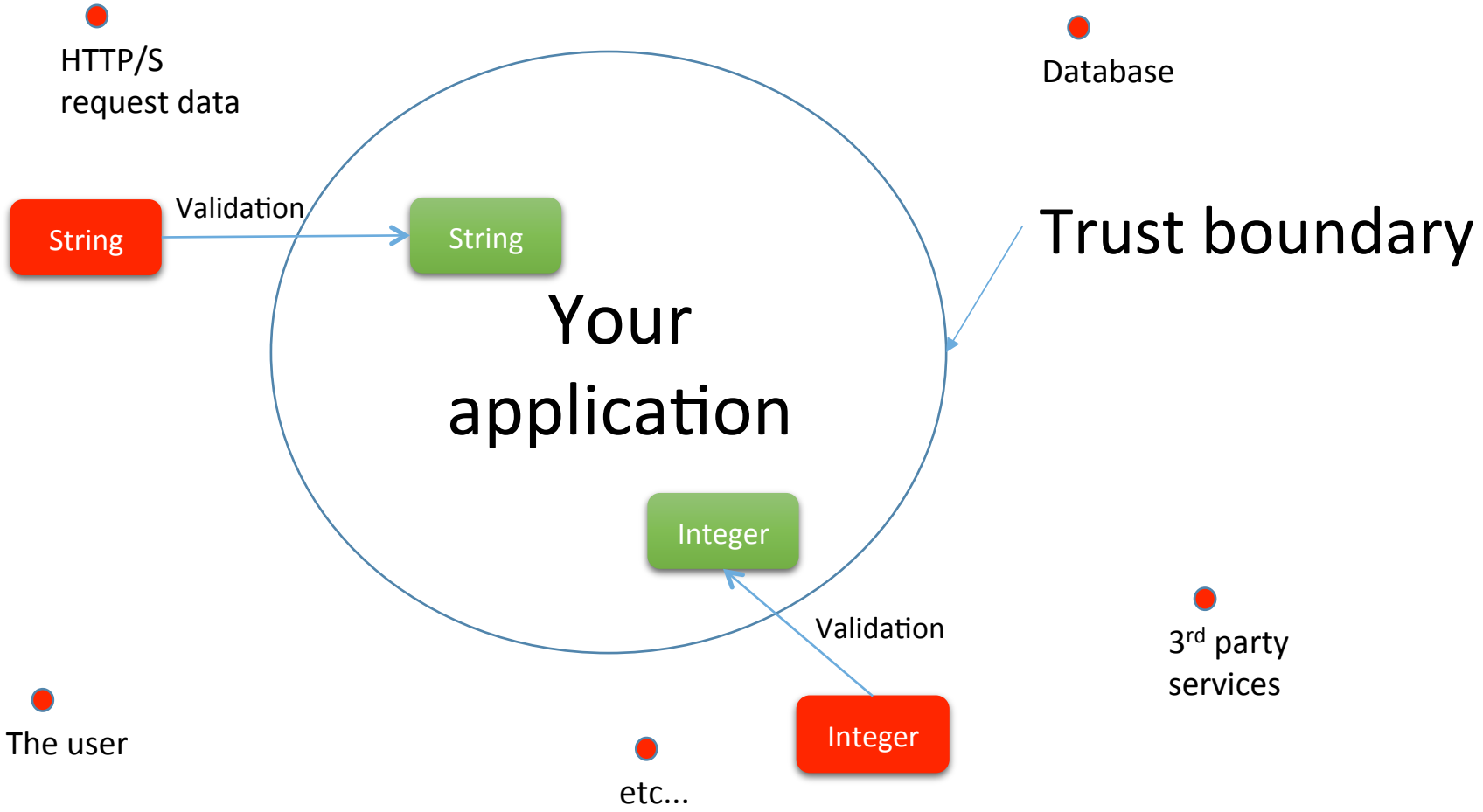
Domain-Driven Security

Domain Driven Design + conventions for validation


1. Hello! This is Bob again!
2. Hello Bob! I'm still the bank!
3. Transfer -1000 euro from account Y to account Z, please!
4. Ok!

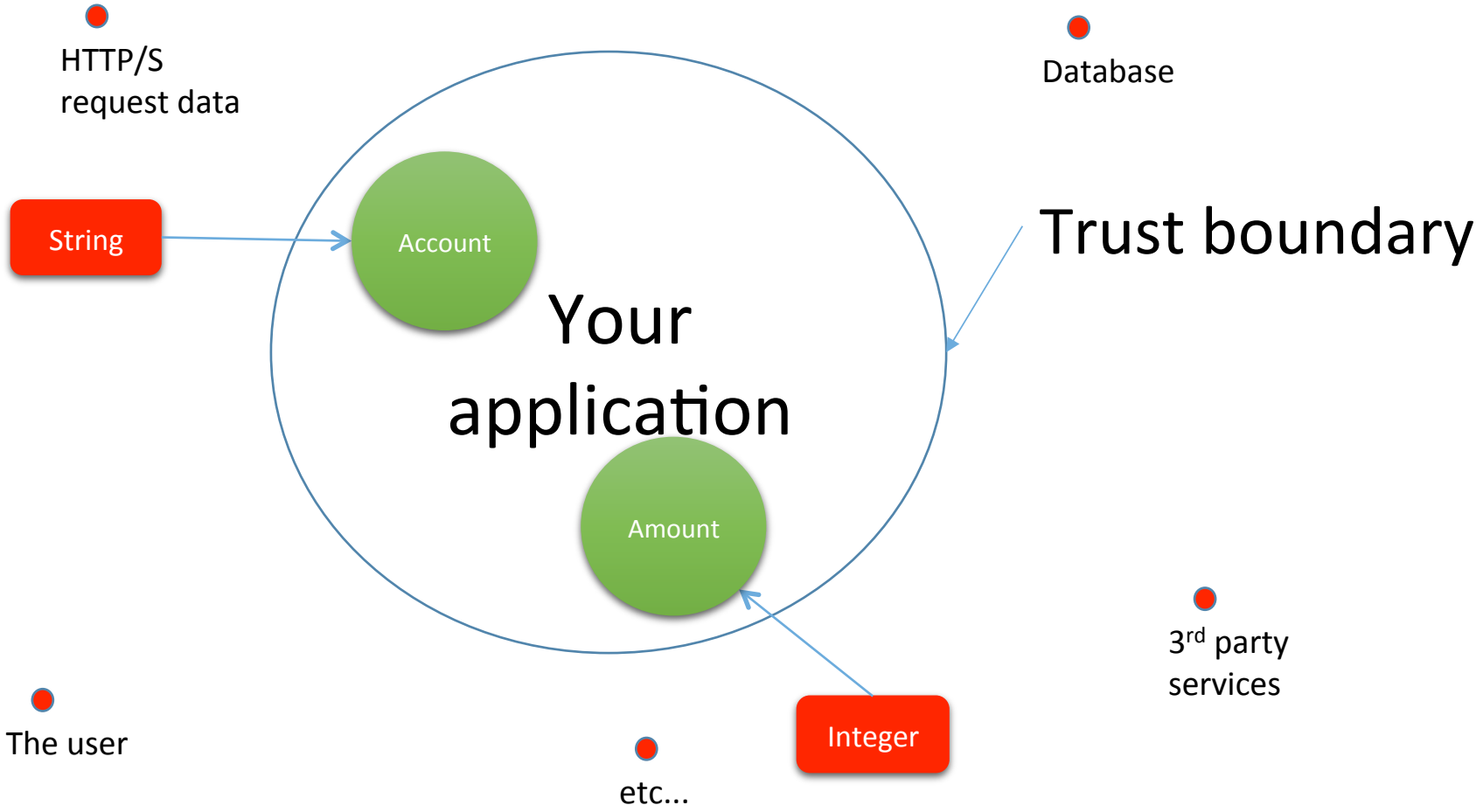
The same validation has to be performed over and over

- Easy to forget to validate somewhere
- Validation ends up everywhere in the code, but (because of this?) is easily forgotten
- Should validate even from “internal” sources such as databases
  Example: stored XSS


HTTP/S
request data
Database
String
Validation
String
Trust boundary
Your
application
Integer
Validation
3rd party
services
The user
Integer
etc...

Domain-Driven Security

- Primitive types and data structures are untrusted by default
  - Strings, integers, byte arrays, collections etc.
- Domain objects
  - Built-in validation
  - (Immutability – more on this later!)

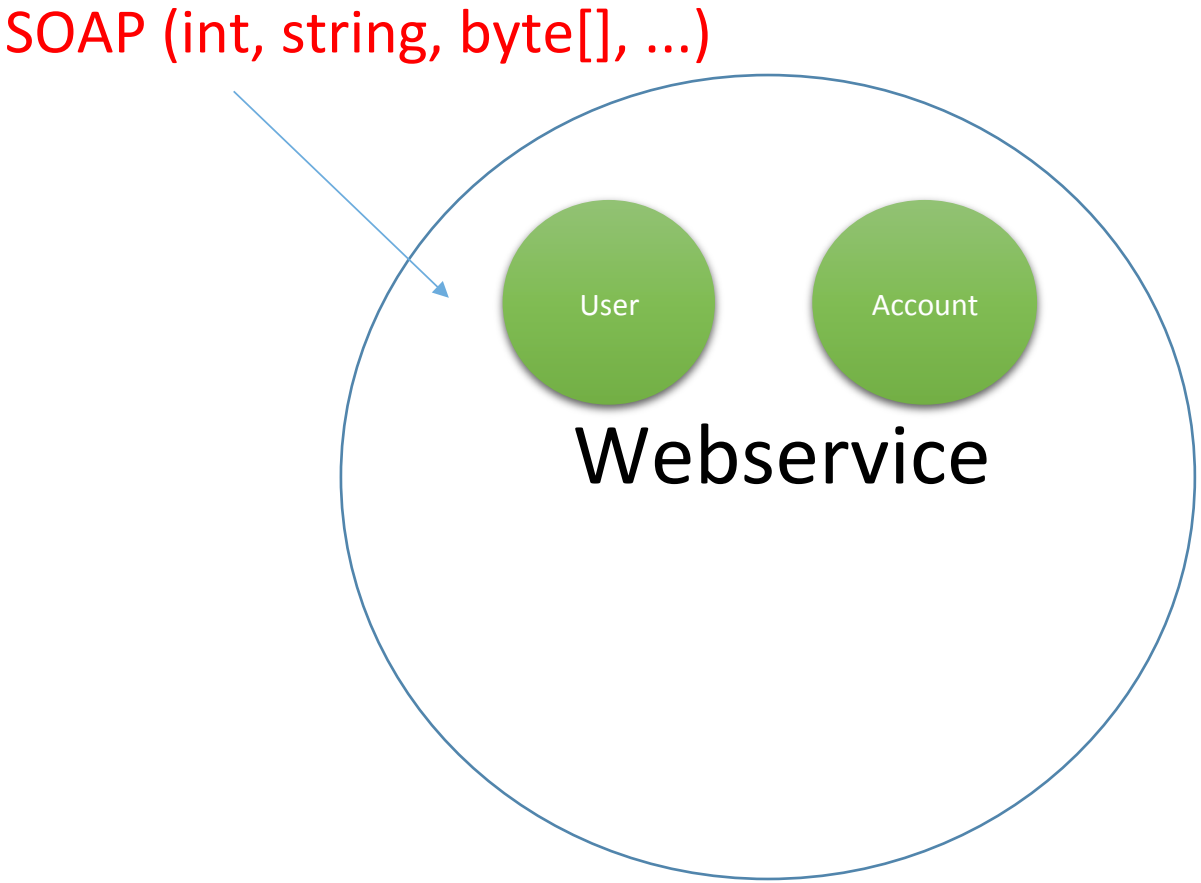

HTTP/S
request data
Database
String
Account
Trust boundary
Your
application
Amount
3rd party
services
The user
Integer
etc...

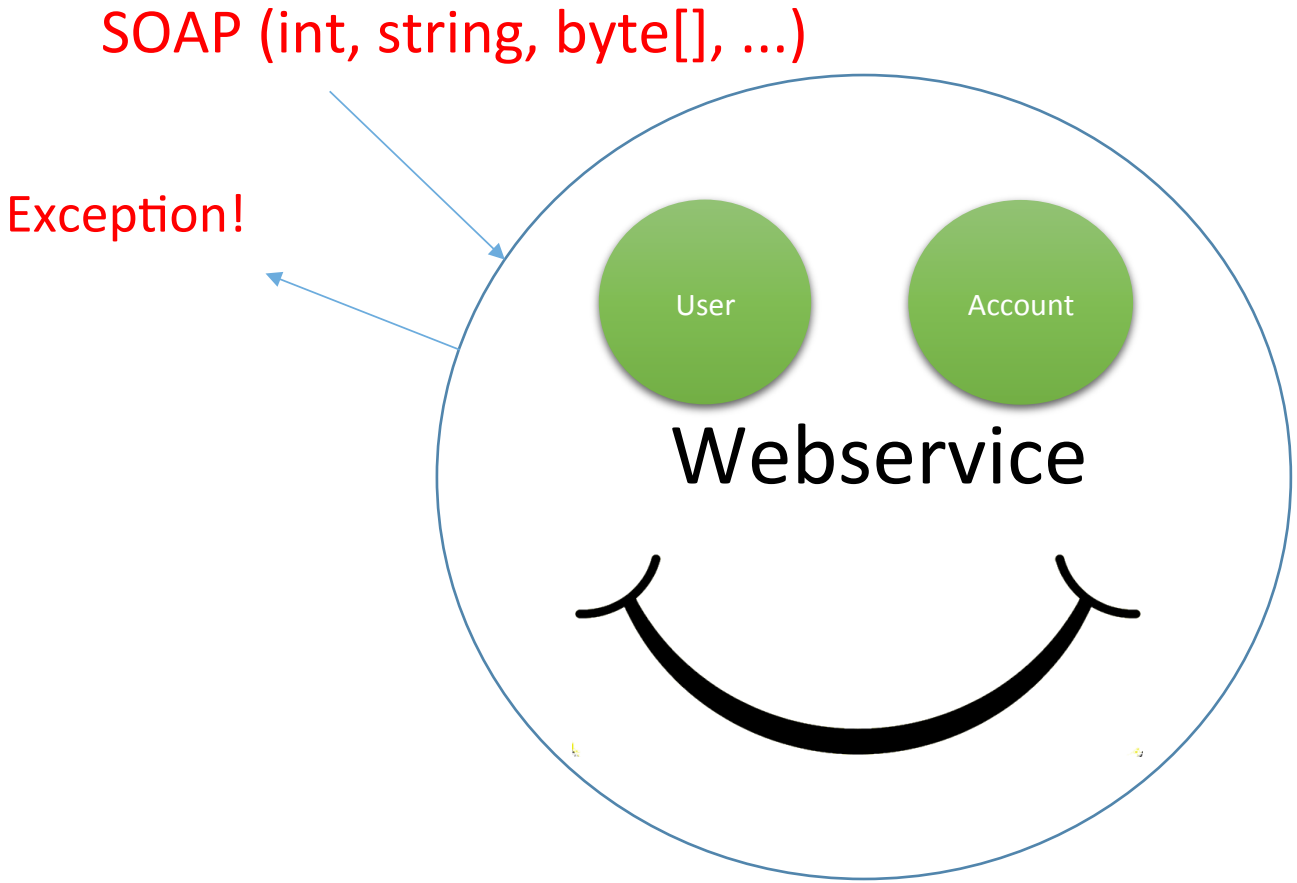
```
public final class AccountNumber {

    private final String value;

    public AccountNumber(String value) {
        if(!isValid(value)){
            throw new IllegalArgumentException("Invalid account
number");
        }
        this.value = value;
    }

   public static boolean isValid(String accountNumber){
        return accountNumber != null
          && hasLength(accountNumber, 10, 12)
          &&  isNumeric(accountNumber);
   }
}
```


SOAP (int, string, byte[], ...)
User
Account
Webservice


SOAP (int, string, byte[], ...)
Exception!
User
Account
Webservice

```
public void Reticulate(Spline spline, int angle);
```

WTF ??

```
public void Reticulate(Spline spline, Angle angle);
```

```
public void Heat(int duration, int temperature);

reactor.Heat(100, 5);  // Boil for 5 minutes
```

```
public void Heat(Duration duration,
                 Temperature temperature);
```

```
int accountNumber = database.getAccountNumberForUser(user);

if (accountNumber <= 0 || accountNumber > 100000) {
    throw new IllegalStateException(”Invalid accountNumber!");
}

pension.transferTo(accountNumber);
```

VS

```
AccountNumber accountNumber =
        new AccountNumber(database.getAccountNumberForUser(user));

pension.transferTo(accountNumber);
```

Domain Driven Security essentials

- You know that all domain objects are valid
- You know you forgot to validate something when you see primitive types being passed around
- The type system ensures that the correct domain object must be used
- Remember: you still need to validate your business rules! But at least you don't have to worry about the building blocks being invalid.

One more thing...

Never use null!

1. “Value might not exist” – make it explicit!

```
public class Optional<T>
{
    public bool IsPresent;

    public T Get;
}
```

~~`int? foo = null;`~~

2. “This shouldn’t happen!” - throw!

```
public Account GetDefaultAccountForUser(User user)
{
    Optional<Account> account = _defaultAccountRepository.GetForUser(user);
    if (!account.IsPresent)
    {
        throw new InvalidOperationException("No default account for user “
            + user.Id + ", should not happen!");
    }
    return account.Get;
}
```

Trust
Domain-Driven Security

The Untrusted Pattern

Make trust a first-class concept at trust boundaries

```
public void Foo(string bar)
{
     if (!IsValid(bar))
     {
          throw new ValidationException();
     }

     DoSomethingWith(bar);
}
```

```
public void Foo(string untrusted_bar)
{
     if (!IsValid(untrusted_bar))
     {
          throw new ValidationException();
     }
     var bar = untrusted_bar;

     DoSomethingWith(bar);
}
```

```
public void Foo2(string untrusted_bar,
                 string untrusted_frob,
                 byte[] data);
```

WTF ??

```
public void Foo(string untrusted_bar)
{
      var bar = Validate(untrusted_bar);

      DoSomethingWith(bar);
}
```

```
public void Foo(Untrusted<string> bar);
```

```
public class Untrusted<T>
{
      readonly T _value;

      public Untrusted(T value)
      {
          _value = value;
      }

      private T Value
      {
          get { return _value };
      }
}

[assembly: InternalsVisibleTo("Validation")]
```

```
// In the "Validation" assembly

public abstract class Validator<T>
{
    public T Validate(Untrusted<T> untrusted)
    {
        if (!InnerValidate(untrusted.Value))
        {
            throw new ValidationException();
        }
        return untrusted.Value;
    }

    protected abstract bool InnerValidate(T value);
}
```

```
public void HandleAcctNbr(Untrusted<string> accountNbr)
{
    var trusted = new
      AccountNumberValidator().Validate(accountNbr);

    DoSomethingWith(trusted);
}
```

```
public void CreateAccount(string nbr)
{
    var untrustedNbr = new Untrusted<string>(nbr);
    HandleAccountNbr(untrustedNbr);
        ...
}
```

Trust
Domain-Driven Security
The Untrusted Pattern

Immutability

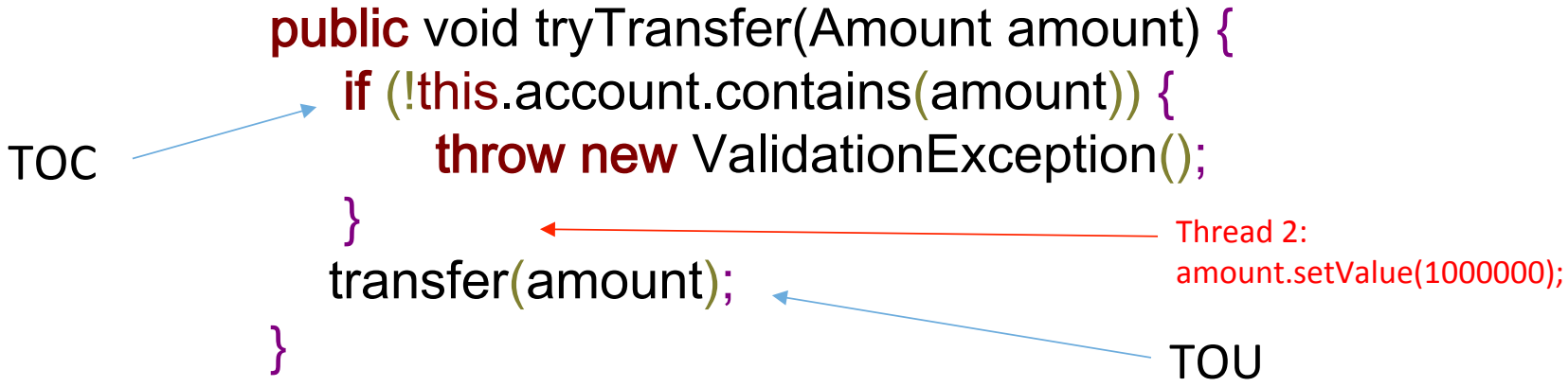
Stuff passed over a trust boundary, regardless of direction, should not be able to change later.

Does your application handle concurrency?

- Hundreds of threads?
- How does that affect validation?
- The thing you just validated, is it still valid?

TOCTTOU

Time Of Check To Time Of Use


public void tryTransfer(Amount amount) {
if (!this.account.contains(amount)) {
throw new ValidationException();
}
transfer(amount);
}
TOC
Thread 2:
amount.setValue(1000000);
TOU

```
public class Amount {
        private final Integer value;

        public Amount(Integer value) {
              if (!isValid(value) {
                   throw new IllegalArgumentException();
              }
              this.value = value;
        }

        public Integer getValue() {
                return this.value;
        }
}
```

Immutability

- Immutability significantly reduces TOCTTOU-problems
- Plays very well with Domain Driven Security
  - … and readability
  - … and parallelization
  - … and event sourcing
  - ... etc

Race condition, web example

```
static Dictionary<Guid, Data> wizardData = new Dictionary<Guid, Data>();

public Guid Wizard_Step1()
{
    var key = Guid.NewGuid();
    wizardData.Add(key, new Data());
    return key;
}

public void Wizard_Step2(Guid key, string productId)
{
    wizardData[key].ProductId = productId;
}

public void Wizard_Step3(Guid key)
{
    var data = wizardData[key];
    if (UserHasAccess(HttpContext.Current.User, data.ProductId)) // TOC
    {
        DoSomethingWith(data);  // TOU
    }
}
```

{ Wizard_Step2(key, secret_productId) }

```
static Dictionary<Guid, ImmutableData> wizardData = new Dictionary<Guid, ImmutableData>();

public Guid Wizard_Step1()
{
    var key = Guid.NewGuid();
    wizardData.Add(key, new ImmutableData());
    return key;
}
public void Wizard_Step2(Guid key, string productId)
{
    var data = wizardData[key];
    var newData = data.CloneWithProductId(productId); // Copies data, new productId
    wizardData[key] = newData;
}

public void Wizard_Step3(Guid key)
{
    var data = wizardData[key];
    if (UserHasAccess(HttpContext.Current.User, data.ProductId)) // TOC
    {
        DoSomethingWith(data);  // TOU
    }
}
```

Immutability

- Security spray
- Should be the norm!

Trust

Domain-Driven Security

The Untrusted Pattern

Immutability

The Inverse Life Coach Pattern

Be a pessimist!

```
boolean success = true;
```


```
return success;
```

Assume failure!

```
boolean success = false;
```


```
return success;
```

Fail fast and force a narrow path of success

```
public ResultData doStuff(Account account) {
  if (!hasAccess(account)) {
    throw new Exception();
  }

  Bunch o' code

    return new ResultData(stuffFromCode);
}
```

Fail fast by throwing

No way of exiting without a valid object

Consider your Trust Boundaries

Enjoy Domain-Driven Security

Immutability should be the norm

Null is a burning bag of dog poop

Fire your Life Coach