SECURE CODE

WARRIOR

Nathan Desmet

Lead Engineer

- Degree in Applied Informatics - Computer and Cyber Crime Professional
- Co-founder of Sensei Security (which is merged with SCW)
- Leading the development of Sensei.

Pieter De Cremer, ir.

Engineer and Ph.D Candidate

- M.Eng – in Computer Science Engineering
- Thesis on binary patch diffing
- Personal grant from Flemish Government to work as a Ph.D. student
- One of the R&D leads at Secure Code Warrior

Vendor Pitch-Free Zone

Promise

SECURE CODE WARRIOR

> Today’s challenges

SECURE CODE WARRIOR

22M

Software developers around the world

~ Evans Data

Source: https://evansdata.com/reports/viewRelease.php?reportID=9

SECURE CODE WARRIOR

111BN

Lines of code written by developers every year ~ CSO Online

Source: https://www.csoonline.com/article/3151003/application-development/world-will-need-to-secure-111-billion-lines-of-new-software-code-in-2017.html

SECURE CODE WARRIOR

1 to 4

Exploitable Security Bugs in every 50 000 Lines of Code

Source: StackOverflow

SECURE CODE WARRIOR

90%

Security incidents result from defects in the design or code ~ DHS

Source: https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf

SECURE CODE WARRIOR

21%

Of data breaches caused by software vulnerability ~ Verizon

Source: Verizon, Data Breach Report, 2018 (but in there the last 10 years)

SECURE CODE WARRIOR

1 in 3

of newly scanned applications had SQL injections over the past 5 yrs ~ Cisco

Source: Cybersecurity as a Growth Advantage, Cisco, 2016


CRY
CRY

> How did we end up here?

AppSec in 2000

Corporates had a branding website, the Internet was mostly for geeks

> *AppSec was virtually non-existent in corporate world*

> *Hacking was focussed on exploiting infrastructure vulnerabilities (bof, race conditions, fmt str*)*

> *Research on first web app weaknesses*

> *OWASP started and Top 10 released!*

> Penetration testing was black magic

F**k it. We've got bigger problems (Y2K) than worrying about Application Security

AppSec in 2010

Companies started offering web-based services; Web 2.0 and Mobile are new

> Penetration testing was THE thing

> Web Application Firewalls will stop everything

> Paper-based secure coding guidelines

> Static Code Analysis Tools (SAST) emerge


Monthly data breaches,
Hackers everywhere,
Privacy, GDPR, PCI-DSS, HIPAA
Putin

AppSec in 2018

Everything runs on software.
Cybersecurity & AppSec are hot topics.

> SAST is still here…

> Runtime Application Security Protection (RASP)

> Dynamic Application Security Testing (DAST)

> Interactive Application Security Testing (IAST)

> Crowd-Sourced Security Testing *(CSST?)*

> **DevSecOps** is getting traction
- Containerisation
- Integrating security and ops into dev
- Security pipelining

> **SHIFT Left**

AppSec in 2018

Challenge - Pen-testing mostly sucks

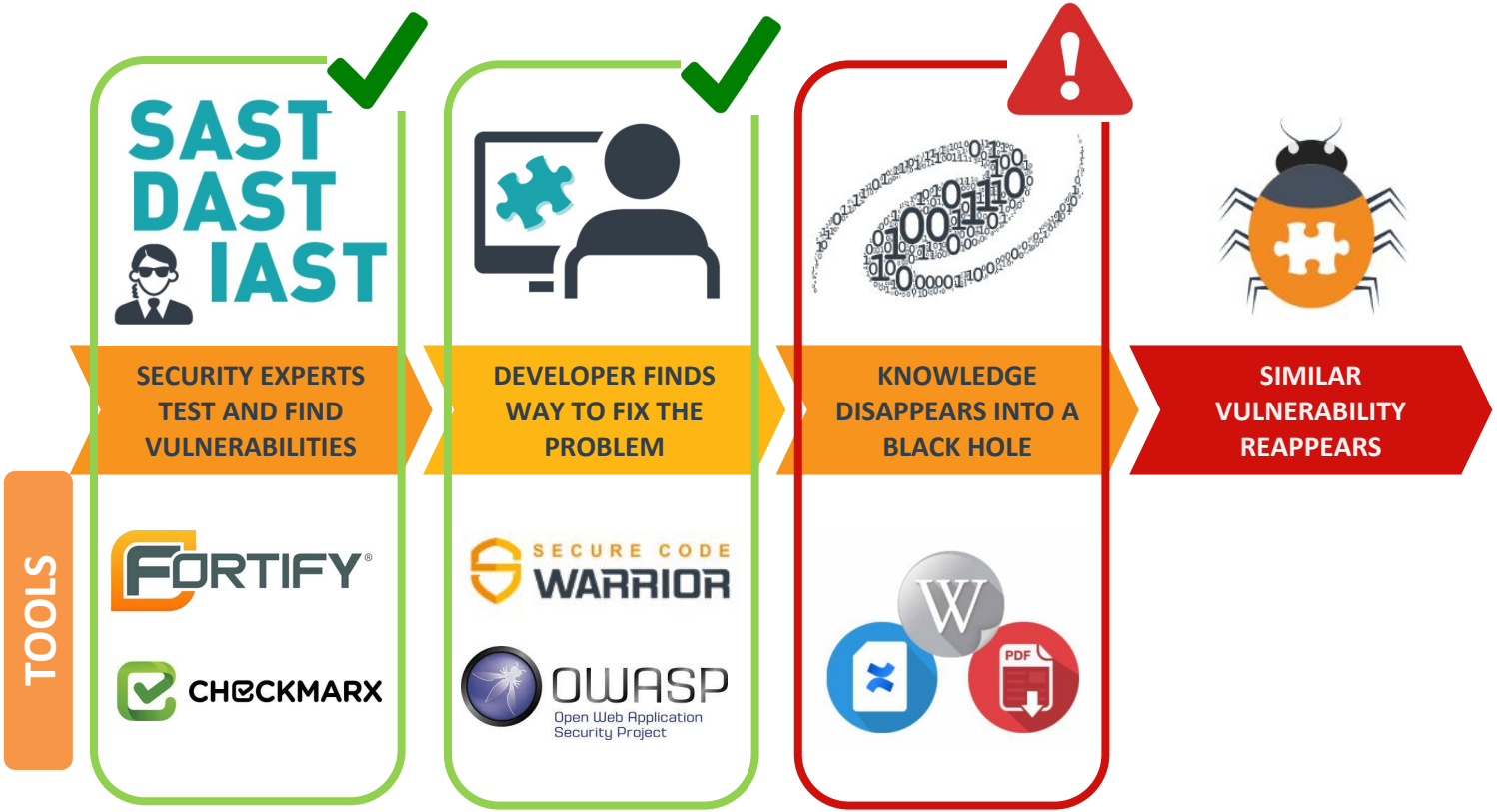
Security Experts

Developers

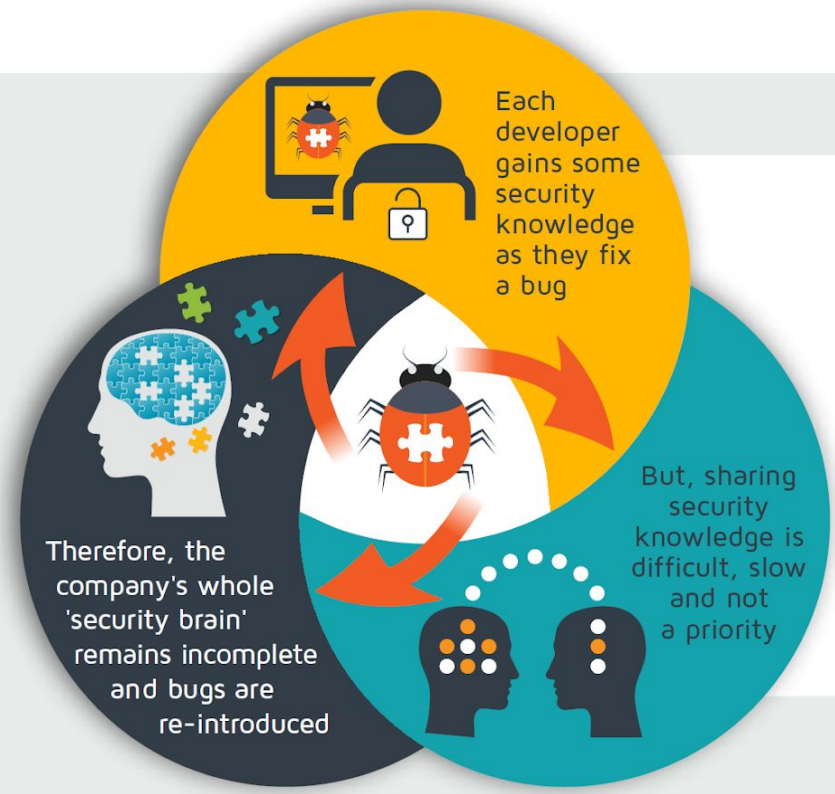
AppSec in 2018

Challenge - “Black Hole” of security knowledge

The current “black hole” of security knowledge



We're failing in Learning from Our Mistakes



- 125+ frequently occurring vulnerability types
- Same vulnerabilities continually re-introduced
- New vulnerabilities also introduced into code
- Today's apps ripe with vulnerabilities


BBC
BBC NEWS

~~SHIFT~~ **START** left

Scale and Make an Impact as an AppSec Pro

> Where does

enforcing coding guidelines

come in?

Developer 1

Coding guideline

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

XXE

B.3.2 XML External Entity (XXE) Processing

High

Description

An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

It was possible to upload and receive data using the XML upload functionality on page:

- http://127.0.0.1:8080/beneficiaries
- http://127.0.0.1:8080/transfers

Evidence

- interbanking

Uploading XML files to the web application allows the attacker to read the server's system files. The example below can be applied to the reported list above. The XML processor parses the uploaded XML and processes the external entity that has been included. This allows the attacker to load and read files of the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY>
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<beneficiaries>
    <beneficiary>
        <name>&xxe;</name>
        <account_number>BE45000100020003</account_number>
        <address>Grotestofstraat 13</address>
```

XXE

Google

XXE

All Images Videos News Maps More Settings Tools

About 16.200.000 results (0,26 seconds)

Top 10-2017 A4-XML External Entities (XXE) - OWASP
https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_(XXE)
Jan 1, 2018 - If the application uses SOAP prior to version 1.2, it is likely susceptible to **XXE** attacks if XML entities are being passed to the SOAP framework.

XML external entity attack - Wikipedia
https://en.wikipedia.org/wiki/XML_external_entity_attack
An XML External Entity attack is a type of attack against an application that parses XML input. .... Detailed guidance on how to disable **XXE** processing, or otherwise defend against **XXE** attacks is presented in the XML External Entity (**XXE**) ...
Description · Examples

PayloadsAllTheThings/XXE injection at master · swisskyrepo ... - GitHub
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XXE%20injection
DOCTYPE replace [<!ENTITY **xxe** SYSTEM "php://filter/convert.base64-encode/resource=index.php">]> <contacts> <contact> <name>Jean &xxe; ...

What is an XXE Attack? - InfoSec Resources - InfoSec Institute
https://resources.infosecinstitute.com/xxe-attacks/
May 15, 2018 - IT Security Training & Resources by InfoSec Institute.

XXE

Top 10-2017 A4-XML External Entities (XXE)

← A3-Sensitive Data Exposure

2017 Table of Contents

PDF version

A5-Broken Access Control →

Threat Agents / Attack Vectors		Security Weakness		Impacts	
App Specific	Exploitability: 2	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or integrations.		By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. SAST tools can discover this issue by inspecting dependencies and configuration. DAST tools require additional manual steps to detect and exploit this issue. Manual testers need to be trained in how to test for XXE, as it not commonly tested as of 2017.		These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks. The business impact depends on the protection needs of all affected application and data.	

Is the Application Vulnerable?

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attack if:

- The application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor.
- Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled. As the exact mechanism for disabling DTD processing varies by processor, it is good practice to consult a reference such as the OWASP Cheat Sheet 'XXE Prevention'.
- If the application uses SAML for identity processing within federated security or single sign on (SSO) purposes. SAML uses XML for identity assertions, and may be vulnerable.
- If the application uses SOAP prior to version 1.2, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.
- Being vulnerable to XXE attacks likely means that the application is vulnerable to denial of service attacks including the Billion Laughs attack

How to Prevent

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
- Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.
- Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
- SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.

If these controls are not possible, consider using virtual patching, API security gateways, or Web Application Firewalls (WAFs) to detect, monitor, and block XXE attacks.

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
```

References

OWASP

- OWASP Application Security Verification Standard
- OWASP Testing Guide: Testing for XML Injection
- OWASP XXE Vulnerability
- OWASP Cheat Sheet: XXE Prevention
- OWASP Cheat Sheet: XML Security

External

XXE

Top 10-2017 A4-XML External Entities (XXE)

← A3-Sensitive Data Exposure

2017 Table of Contents

PDF version

A5-Broken Access Control →

Threat Agents / Attack Vectors		Security Weakness		Impacts	
App Specific	Exploitability: 2	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies or integrations.		By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. SAST tools can discover this issue by inspecting dependencies and configuration. DAST tools require additional manual steps to detect and exploit this issue. Manual testers need to be trained in how to test for XXE, as it not commonly tested as of 2017.		These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks. The business impact depends on the protection needs of all affected application and data.	

Is the Application Vulnerable?

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attack if:

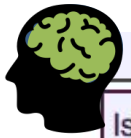
- The application accepts XML directly or XML uploads, especially from untrusted

How to Prevent

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and avoiding

XXE



as it not commonly tested as of 2017.

Is the Application Vulnerable?

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attack if:

- The application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor.
- Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled. As the exact mechanism for disabling DTD processing varies by processor, it is good practice to consult a reference such as the OWASP Cheat Sheet 'XXE Prevention'.
- If the application uses SAML for identity processing within federated security or single sign on (SSO) purposes. SAML uses XML for identity assertions, and may be vulnerable.
- If the application uses SOAP prior to version 1.2, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.
- Being vulnerable to XXE attacks likely means that the application is vulnerable to denial of service attacks including the Billion Laughs attack

How to Prevent

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
- Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.
- Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
- SAST tools can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.

If these controls are not possible, consider using virtual patching, API security gateways, or Web Application Firewalls (WAFs) to detect, monitor, and block XXE attacks.

Example Attack Scenarios

References

XXE

attacks.

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
  <foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
  <!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
  <!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

References

OWASP

- OWASP Application Security Verification Standard
- OWASP Testing Guide: Testing for XML Injection
- OWASP XXE Vulnerability
- OWASP Cheat Sheet: XXE Prevention
- OWASP Cheat Sheet: XML Security

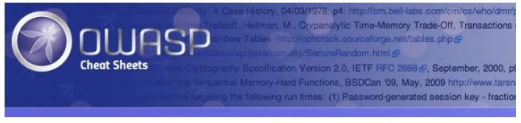
External

- CWE-611: Improper Restriction of XXE
- Billion Laughs Attack
- SAML Security XML External Entity Attack
- Detecting and exploiting XXE in SAML Interfaces

Preventing XXE

SECURE CODE WARRIOR

XML External Entity (XXE) Prevention Cheat Sheet



Last revision (mm/dd/yy): **08/28/2018**

[hide]

Introduction

XML eXternal Entity injection (XXE), which is now part of the OWASP Top 10, is a type of attack against an application that parses XML input. This attack occurs when untrusted **XML input containing a reference to an external entity is processed by a weakly configured XML parser**. This attack may lead to the disclosure of confidential data, denial of service, Server Side Request Forgery (SSRF), port scanning from the perspective of the machine where the parser is located, and other system impacts. The following guide provides concise information to prevent this vulnerability. For more information on XXE, please visit XML External Entity (XXE) Processing.

General Guidance

The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:

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

Disabling DTDs also makes the parser secure against denial of services (DOS) attacks such as Billion Laughs. If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that's specific to each parser.

Detailed XXE Prevention guidance for a number of languages and commonly used XML parsers in those languages is provided below.

C/C++

libxml2

The Enum xmlParserOption should not have the following options defined:

- XML_PARSE_NOENT: Expands entities and substitutes them with replacement text
- XML_PARSE_DTDLOAD: Load the external DTD

Note: Per: https://mail.gnome.org/archives/xml/2012-October/msg00045.html, starting with libxml2 version 2.9, XXE has been disabled by default as committed by the following patch: http://git.gnome.org/browse/libxml2/commit/?id=4629ee02ac649c27f9c0cf98ba017c6b5526070f.

Search for the usage of the following APIs to ensure there is no "XML_PARSE_NOENT" and "XML_PARSE_DTDLOAD" defined in the parameters.

- xmlCtxtReadDoc 、xmlCtxtReadFd 、xmlCtxtReadFile 、xmlCtxtReadIO 、xmlCtxtReadMemory 、xmlCtxtUseOptions 、xmlParseInNodeContext 、xmlReadDoc 、xmlReadFd 、xmlReadFile 、xmlReadIO 、xmlReadMemory

libxerces-c

Use of XercesDOMParser do this to prevent XXE:

```
XercesDOMParser *parser = new XercesDOMParser;
parser->setCreateEntityReferenceNodes(false);
```

Use of SAXParser, do this to prevent XXE:

```
SAXParser* parser = new SAXParser;
parser->setDisableDefaultEntityResolution(true);
```

Use of SAX2XMLReader, do this to prevent XXE:

```
SAX2XMLReader* reader = XMLReaderFactory::createXMLReader();
parser->setFeature(XMLUni::fgXercesDisableDefaultEntityResolution, true);
```

Java

Java applications using XML libraries are particularly vulnerable to XXE because the default settings for most Java XML parsers is to have XXE enabled. To use these parsers safely, you have to explicitly disable XXE in the parser you use. The following describes how to disable XXE in the most commonly used XML parsers for Java.

JAXP DocumentBuilderFactory, SAXParserFactory and DOM4J

DocumentBuilderFactory, SAXParserFactory and DOM4J XML Parsers can be configured using the same techniques to protect them against XXE. Only the DocumentBuilderFactory example is presented here. The JAXP DocumentBuilderFactory setFeature method allows a developer to control which implementation-specific XML processor features are enabled or disabled. The features can either be set on the factory or the underlying XMLReader setFeature method. Each XML processor implementation has its own features that govern how DTDs and external entities are processed.

For a syntax highlighted example code snippet using SAXParserFactory, look here.

```
import javax.xml.parsers.DocumentBuilderFactory;
 import javax.xml.parsers.ParserConfigurationException; // catching unsupported features
 ...

     DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
     String FEATURE = null;
```

```
// Xerces 1 - http://xerces.apache.org/xerces-j/features.html#external-general-entities
// Xerces 2 - http://xerces.apache.org/xerces2-j/features.html#external-general-entities
// JDK7+ - http://xml.org/sax/features/external-general-entities
FEATURE = "http://xml.org/sax/features/external-general-entities";
dbf.setFeature(FEATURE, false);

// Xerces 1 - http://xerces.apache.org/xerces-j/features.html#external-parameter-entities
// Xerces 2 - http://xerces.apache.org/xerces2-j/features.html#external-parameter-entities
// JDK7+ - http://xml.org/sax/features/external-parameter-entities
FEATURE = "http://xml.org/sax/features/external-parameter-entities";
dbf.setFeature(FEATURE, false);

// Disable external DTDs as well
FEATURE = "http://apache.org/xml/features/nonvalidating/load-external-dtd";
dbf.setFeature(FEATURE, false);

// and these as well, per Timothy Morgan's 2014 paper: "XML Schema, DTD, and Entity Attacks"
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);

// And, per Timothy Morgan: "If for some reason support for inline DOCTYPEs are a requirement, then
// ensure the entity settings are disabled (as shown above) and beware that SSRF attacks
// (http://cwe.mitre.org/data/definitions/918.html) and denial
// of service attacks (such as billion laughs or decompression bombs via "jar:") are a risk."

// remaining parser logic
...
} catch (ParserConfigurationException e) {
        // This should catch a failed setFeature feature
        logger.info("ParserConfigurationException was thrown. The feature '" +
                FEATURE + "' is probably not supported by your XML processor.");
        ...
}
catch (SAXException e) {
        // On Apache, this should be thrown when disallowing DOCTYPE
        logger.warning("A DOCTYPE was passed into the XML document");
        ...
}
catch (IOException e) {
        // XXE that points to a file that doesn't exist
        logger.error("IOException occurred, XXE may still possible: " + e.getMessage());
        ...
}
DocumentBuilder safebuilder = dbf.newDocumentBuilder();
```

Xerces 1 Features:

- Do not include external entities by setting this feature to `false`.
- Do not include parameter entities by setting this feature to `false`.
- Do not include external DTDs by setting this feature to `false`.

Xerces 2 Features:

- Disallow an inline DTD by setting this feature to `true`.
- Do not include external entities by setting this feature to `false`.
- Do not include parameter entities by setting this feature to `false`.
- Do not include external DTDs by setting this feature to `false`.

Note: The above defenses require Java 7 update 67, Java 8 update 20, or above, because the above countermeasures for DocumentBuilderFactory and SAXParserFactory are broken in earlier Java versions, per: CVE-2014-6517.

XMLInputFactory (a StAX parser)

StAX parsers such as XMLInputFactory allow various properties and features to be set.

To protect a Java XMLInputFactory from XXE, do this:

```
xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false); // This disables DTDs entirely for that factory
xmlInputFactory.setProperty("javax.xml.stream.isSupportingExternalEntities", false); // disable external entities
```

TransformerFactory

To protect a javax.xml.transform.TransformerFactory from XXE, do this:

```
TransformerFactory tf = TransformerFactory.newInstance();
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
```

Validator

To protect a javax.xml.validation.Validator from XXE, do this:

```
SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
Schema schema = factory.newSchema();
Validator validator = schema.newValidator();
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
```

SchemaFactory

To protect a javax.xml.validation.SchemaFactory from XXE, do this:

```
SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
Schema schema = factory.newSchema(Source);
```

SAXTransformerFactory

To protect a javax.xml.transform.sax.SAXTransformerFactory from XXE, do this:

```
SAXTransformerFactory sf = SAXTransformerFactory.newInstance();
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
sf.newXMLFilter(Source);
```

Preventing XXE

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
String FEATURE = null;
FEATURE = "http://apache.org/xml/features/disallow-doctype-decl";
dbf.setFeature(FEATURE, true);
DocumentBuilder safebuilder = dbf.newDocumentBuilder();
```

Preventing XXE

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
String FEATURE = null;
FEATURE = "http://apache.org/xml/features/disallow-doctype-decl";
dbf.setFeature(FEATURE, true);
DocumentBuilder safebuilder = dbf.newDocumentBuilder();
```

Secure coding guideline

On **`DocumentBuilderFactory`**

call **`setFeature`** with these parameters

before calling **`newDocumentBuilder`**

Developer 2

Coding guideline

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

Coding guideline

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

Secure coding guideline

On **`DocumentBuilderFactory`**

call **`setFeature`** with these parameters

before calling **`newDocumentBuilder`**

Coding guideline

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setFeature( name: "http://apache.org/xml/features/dissalow-doctype-decl",  value: true);
DocumentBuilder builder = factory.newDocumentBuilder();
```

Secure coding guideline

On **`DocumentBuilderFactory`**

call **`setFeature`** with these parameters

before calling **`newDocumentBuilder`**

XXE

B.3.2 XML External Entity (XXE) Processing

High

Description

An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

It was possible to upload and receive data using the XML upload functionality on page:

- http://127.0.0.1:8080/beneficiaries
- http://127.0.0.1:8080/transfers

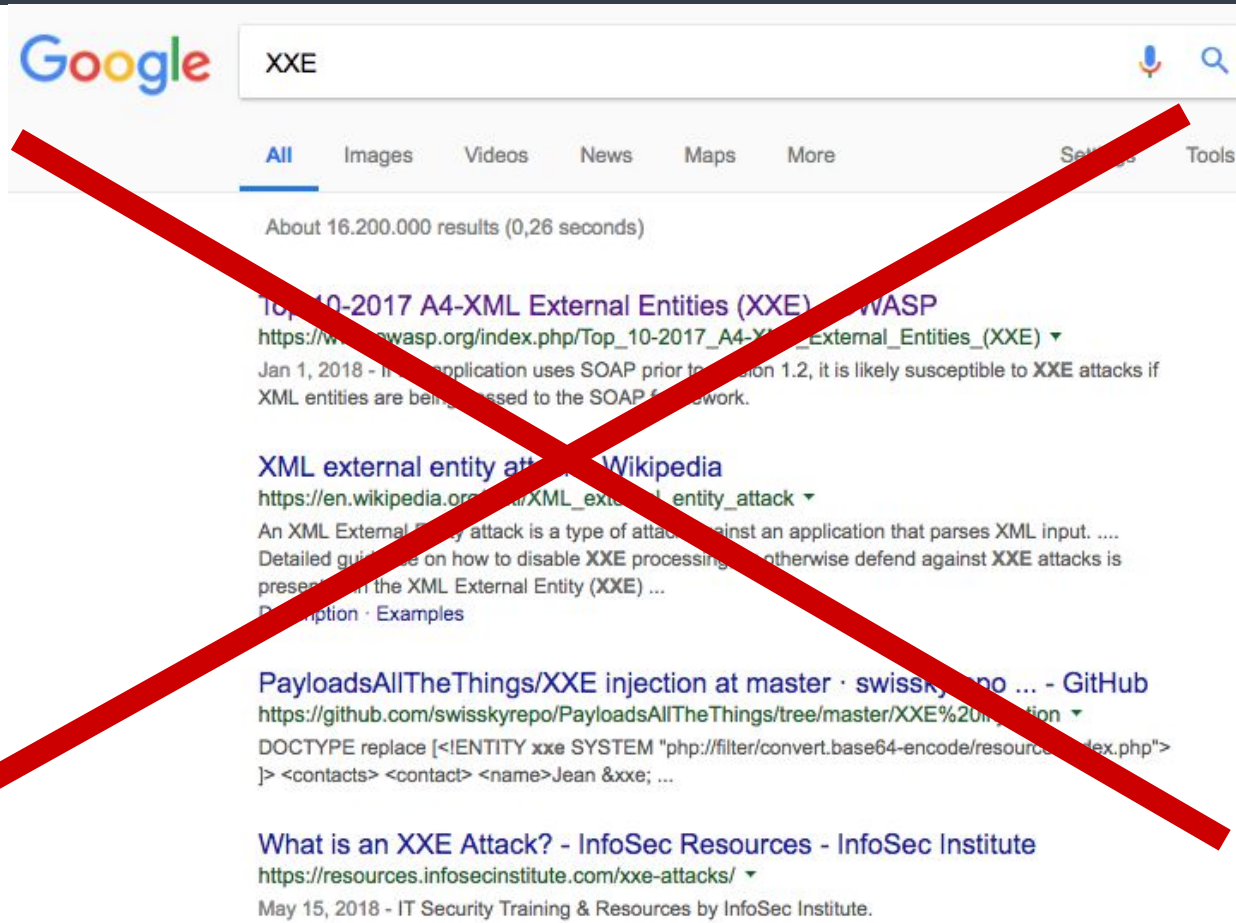
Evidence

- interbanking

Uploading XML files to the web application allows the attacker to read the server's system files. The example below can be applied to the reported list above. The XML processor parses the uploaded XML and processes the external entity that has been included. This allows the attacker to load and read files of the server.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY>
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<beneficiaries>
    <beneficiary>
        <name>&xxe;</name>
        <account_number>BE45000100020003</account_number>
        <address>Grotestofstraat 13</address>
```

XXE


Google
XXE
All
Images
Videos
News
Maps
More
Tools
About 16.200.000 results (0,26 seconds)
XML External Entities (XXE)
https://en.wikipedia.org/wiki/XML_entity_attack
PayloadsAllTheThings/XXE injection at master
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XXE%20
What is an XXE Attack? - InfoSec Resources - InfoSec Institute
https://resources.infosecinstitute.com/xxe-attacks/
May 15, 2018 - IT Security Training & Resources by InfoSec Institute.

XXE

Top 10-2017 A4-XML External Entities (XXE)

← A3-Sensitive Data Exposure

2017 Table of Contents

PDF version

A5-Broken Access Control →

Threat Agents / Attack Vectors		Security Weakness		Impacts	
App Specific	Exploitability: 2	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
kers can exploit vulnerable XML processors if they ca XML or include hostile content in an XML documen ing vulnerable code, dependencies or integrations.		By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. SAST tools can discover this issue by inspecting dependencies and configuration. DAST tools require additional manual steps to detect and exploit this issue. Manual testers need to be trained in how to test for XXE, as it not commonly tested as of 2017.		These flaws can be used to extract data, execute a r request from the server, scan internal systems a denial-of-service attack, as well as exec attacks. The business impact depends on th ction needs of all affected application and d	

Is the Application Vulnerable?

Applications and in particular XML-based web services or d m integrations might be vulnerable to attack if:

- The application accepts XML directly or XML uploads, especially from d sources, or inserts untrusted data into XML documents, which is then parsed XML processor.
- Any of the XML processors in the application or SOAP based web se as document type definitions (DTDs) enabled. As the exact me for disabling DTD processing varies by processor, it is good practice t a reference such as the OWASP Cheat Sheet 'XXE Prevention'.
- If the application uses SAML for identity pr within federated security or single sign on (SSO) purposes. SAM ML for identity assertions, and may be vulnerable.
- If the application uses SO to version 1.2, it is likely susceptible to XXE attacks if XML entit being passed to the SOAP framework.
- Being vulne XXE attacks likely means that the application is vulnerable to denial ce attacks including the Billion Laughs attack

How to Prevent

Developer train sential to identify and mitigate XXE. Besides that, preventing XXE requ

- ever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on derlying operating system. Use dependency checkers. Update SOAP to SO or higher.
- Disable X nal entity and DTD processing in all XML parsers in the application, as p OWASP Cheat Sheet 'XXE Prevention'.
- Implement positive ("w ng") server-side input validation, filtering, or sanitization to prevent hostil thin XML documents, headers, or nodes.
- Verify that XML or XSL file upload ality validates incoming XML using XSD validation or similar.
- SAST tools can help detect XXE in source cod gh manual code review is the best alternative in large, complex applications wi integrations.

If these controls are not possible, consider using virtual patching curity gateways, or Web Application Firewalls (WAFs) to detect, monitor, an XXE attacks.

Example Attack Scenarios

Numerous public XXE issues have been discovered, including attacking embedded devices. XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted:

Scenario #1: The attacker attempts to extract data from the server:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
```

References

OWASP

- OWASP Application Security Verification Standard
- OWASP Testing Guide: Testing for XML Injection
- OWASP XXE Vulnerability
- OWASP Cheat Sheet: XXE Prevention
- OWASP Cheat Sheet: XML Security

External

Preventing XXE

SECURE CODE WARRIOR

XML External Entity (XXE) Prevention Cheat Sheet

OWASP Cheat Sheets

Last revision (mm/dd/yy): **08/28/2018**

[hide]

Introduction

XML eXternal Entity injection (XXE), which is now part of the OWASP Top 10, is a type of attack against an application that parses XML input. This attack occurs when untrusted **XML input containing a reference to an external entity is processed by a weakly configured XML parser**. This attack may lead to the disclosure of confidential data, denial of service, Server Side Request Forgery (SSRF), port scanning from the perspective of the machine where the parser is located, and other system impacts. The following guide provides concise information to prevent this vulnerability. For more information on XXE, please visit XML External Entity (XXE) Processing.

General Guidance

The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:

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

Disabling DTDs also makes the parser secure against denial of services (DOS) attacks such as Billion Laughs. If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that's specific to each parser.

Detailed XXE Prevention guidance for a number of languages and commonly used XML parsers in those languages is provided below.

C/C++

The Erl[illegible]rOptions should not have the following options defined:

- XML_PARSE[illegible] expands entities and substitutes them with replacement text
- XML_PARSE_DTL[illegible] the external DTD

Note: Per: https://mail.gnome.[illegible]/xml/2012-October/msg00045.html, starting with libxml2 [illegible] XXE has been disabled by default as committed by the following patch: http://git.gnome.org[illegible]mit/?id=4629ee02ac649c27f9c0cf98[illegible]20070f.

Search for the usage of the following APIs to [illegible] is no "XML_PARSE_NOENT[illegible]XML_PARSE_DTDLOAD" defined in the parameters.

- xmlCtxtReadDoc 、xmlCtxtReadFd 、xmlCtxtRe[illegible]CtxtReadIO[illegible]adMemory 、xmlCtxtUseOptions 、xmlParseInNodeContext 、xmlReadDoc 、xmlReadFd 、xmlReadFile 、xmlReadIO 、xmlRead[illegible]

libxerces-c

Use of XercesDOMParser do this to prevent XXE:

```
XercesDOMParser *parser = [illegible]sDOMParser;
parser->setCreateEntit[illegible]eNodes(false);
```

Use of SAXParser[illegible]revent XXE:

```
SA[illegible] parser = new SAXParser;
[illegible]setDisableDefaultEntityResolution(true);
```

Use of SAX2XMLReader, do this to prevent XXE:

```
SAX2XMLReader* reader = XMLReaderFactory::createXMLReader();
parser->setFeature(XMLUni::fgXercesDisableDefaultEntityResolution, true);
```

Java

Java applications using XML libraries are particularly vulnerable to XXE because the default settings for most Java XML parsers is to have XXE enabled. To use these parsers safely, you have to explicitly disable XXE in the parser you use. The following describes how to disable XXE in the most commonly used XML parsers for Java.

JAXP DocumentBuilderFactory, SAXParserFactory and DOM4J

DocumentBuilderFactory, SAXParserFactory and DOM4J XML Parsers can be configured using the same techniques to protect them against XXE. Only the DocumentBuilderFactory example is presented here. The JAXP DocumentBuilderFactory setFeature method allows a developer to control which implementation-specific XML processor features are enabled or disabled. The features can either be set on the factory or the underlying XMLReader setFeature method. Each XML processor implementation has its own features that govern how DTDs and external entities are processed.

For a syntax highlighted example code snippet using SAXParserFactory, look here.

```
import javax.xml.parsers.DocumentBuilderFactory;
 import javax.xml.parsers.ParserConfigurationException; // catching unsupported features
 ...

     DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
     String FEATURE = null;
```

```
// Xerces 1 - http://xerces.apache.org/xerces-j/features.html#external-general-entities
// Xerces 2 - http://xerces.apache.org/xerces2-j/features.html#external-general-entities
// JDK7+ - http://xml.org/sax/features/external-general-entities
FEATURE = "http://xml.org/sax/features/external-general-entities";
dbf.setFeature(FEATURE, false);

// Xerces 1 - http://xerces.apache.org/xerces-j/features.html#external-parameter-entities
// Xerces 2 - http://xerces.apache.org/xerces2-j/features.html#external-parameter-entities
// JDK7+ - http://xml.org/sax/features/external-parameter-entities
FEATURE = "http://xml.org/sax/features/external-parameter-entities";
dbf.setFeature(FEATURE, false);

// Disable external [illegible]well
FEATURE = "http://[illegible]/xml/features/nonvalidating/load-external-dtd";
dbf.setFeature[illegible]e);

// and [illegible], per Timothy Morgan's 2014 paper: "XML Schema, DTD, and Entity Attacks"
dbf.[illegible]ource(false);
[illegible]ntityReferences(false);

[illegible], per Timothy Morgan: "If for some reason support for inline DOCTYPEs are a requirement, then
// ensure the entity settings are disabled (as shown above) and beware that SSRF attacks
// (http://cwe.mitre.org/data/definitions/918.html) and denial
// of service attacks (such as billion laughs or decompression bombs via "jar:") are a risk."

// remaining parser logic
...
} catch (ParserConfigurationException e) {
      // This should catch a failed setFeature feature
      logger.info("ParserConfigurationException was thrown. The feature '" +
                  FEATURE + "' is probably not supported by your XML processor.");
      ...
}
catch (SAXException e) {
      // On Apache, this should be thrown when disallowing DOCTYPE
      logger.warning("A DOCTYPE was passed into the XML document");
      ...
}
catch (IOException e) {
      // XXE that points to a file that doesn't exist
      logger.error("IOException occurred, XXE may still possible: " + e.getMessage());
      ...
}
DocumentBuilder safebuilder = dbf.newDocumentBuilder();
```

Xerces 1 Features:

- Do not include external entities by setting this feature to `false`.
- Do not include parameter entities by setting this feature to `false`.
- Do not include external DTDs by setting this feature to `false`.

Xerces 2 Features:

- Disallow an inline DTD by setting this feature to `true`.
- Do not include external entities by setting this feature to `false`.
- Do not include parameter entities by setting this feature to `false`.
- Do not include external DTDs by setting this feature to `false`.

Note: The above defenses require Java 7 update 67, Java 8 update 20, or above, because the above countermeasures for DocumentBuilderFactory and SAXParserFactory are broken in earlier Java versions, per: CVE-2014-6517.

XMLInputFactory (a StAX parser)

StAX parsers such as XMLInputFactory allow various properties and features to be set.

To protect a Java XMLInputFactory from XXE, do this:

```
xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false); // This disables DTDs entirely for that factory
xmlInputFactory.setProperty("javax.xml.stream.isSupportingExternalEntities", false); // disable external entities
```

TransformerFactory

To protect a javax.xml.transform.TransformerFactory from XXE, do this:

```
TransformerFactory tf = TransformerFactory.newInstance();
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
```

Validator

To protect a javax.xml.validation.Validator from XXE, do this:

```
SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
Schema schema = factory.newSchema();
Validator validator = schema.newValidator();
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
[illegible]lidator.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
```

Sch[illegible]

To protect a [illegible]maFactory from XXE, do this:

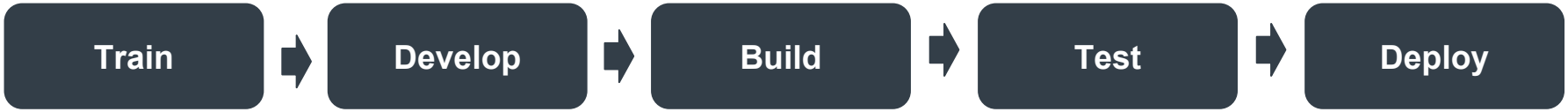
```
SchemaFactory [illegible]ctory.newInstance("http://www.w3.org/2001/XMLSchema");
factory.setProperty([illegible]CCESS_EXTERNAL_DTD, "");
factory.setProperty(XMLCo[illegible]D_EXTERNAL_SCHEMA, "");
Schema schema = factory.[illegible]
```

SAXTransformerFactory

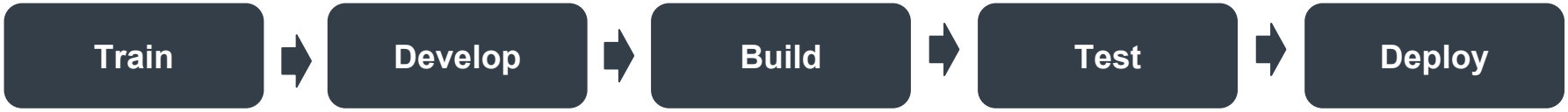
To protect a javax.xml.transform.sax.SAXTransformerFactory from XXE, do this:

```
SAXTransformerFactory sf = SAXTransformerFactory.newInstance();
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
sf.newXMLFilter(Source);
```

SDLC



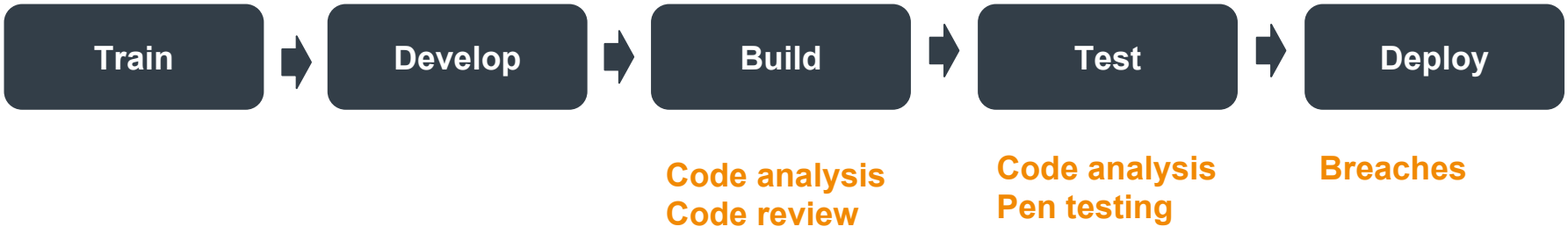
Security started as part of software testing



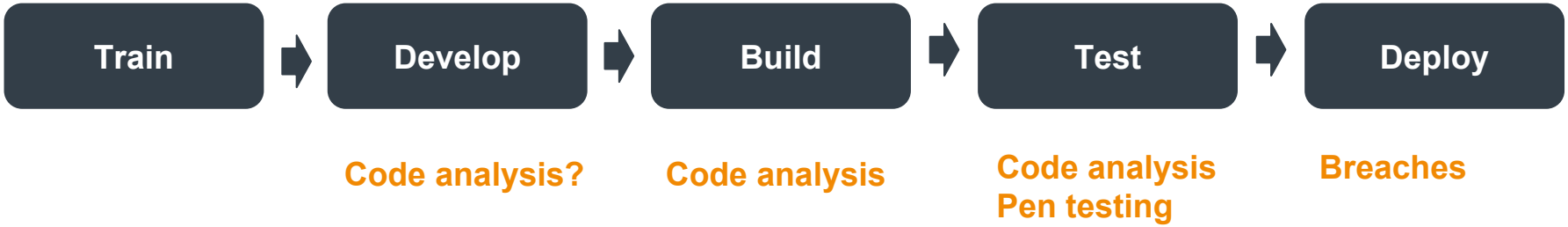
Pen testing
Code analysis

Breaches

Shift left

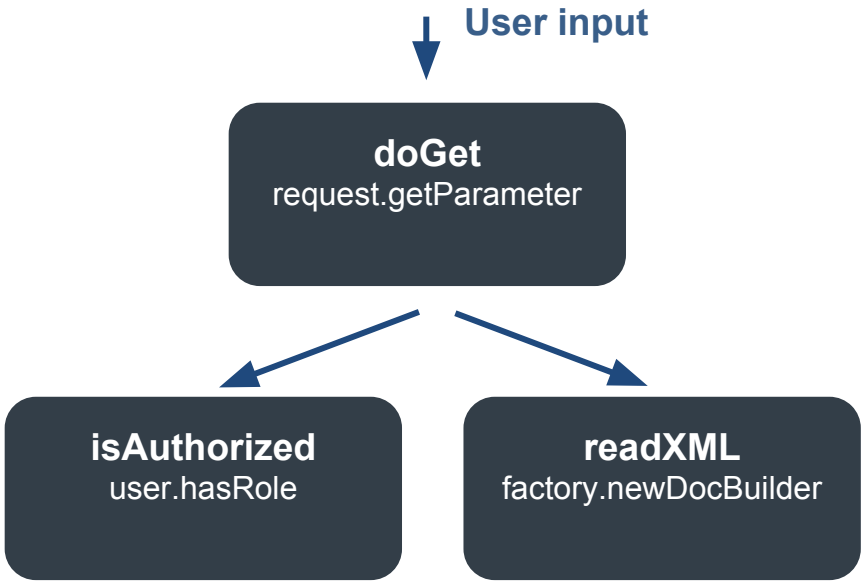

Train
Develop
Build
Test
Deploy
Code analysis
Code review
Code analysis
Pen testing
Breaches

Keep shifting left?



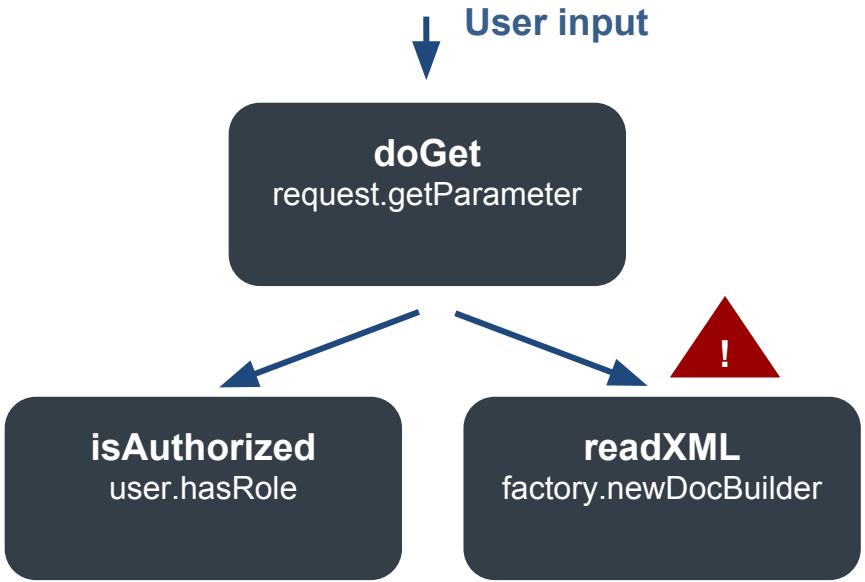
Code analysis

Train → Develop → **Build** → Test → Deploy

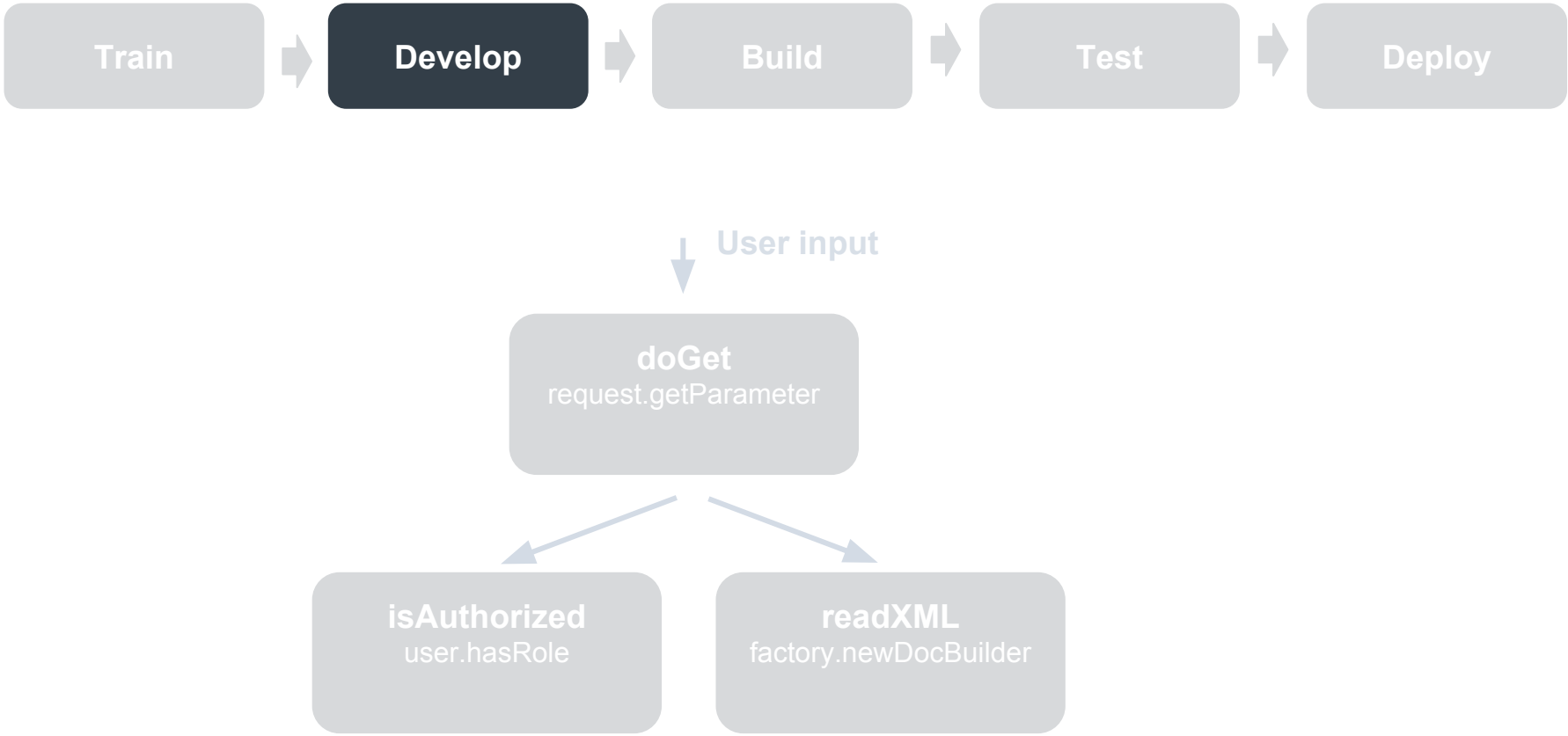


Code analysis

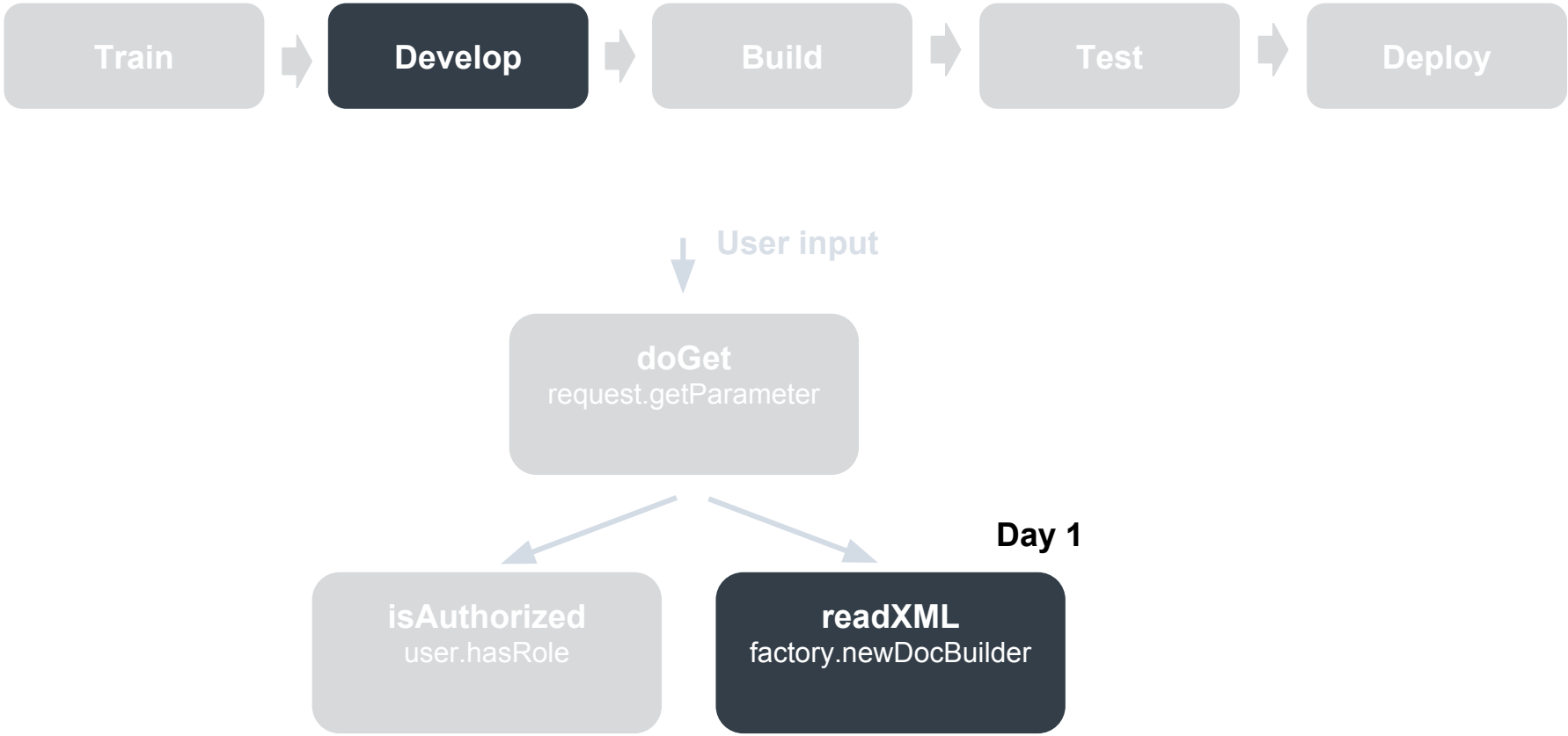
Train → Develop → **Build** → Test → Deploy



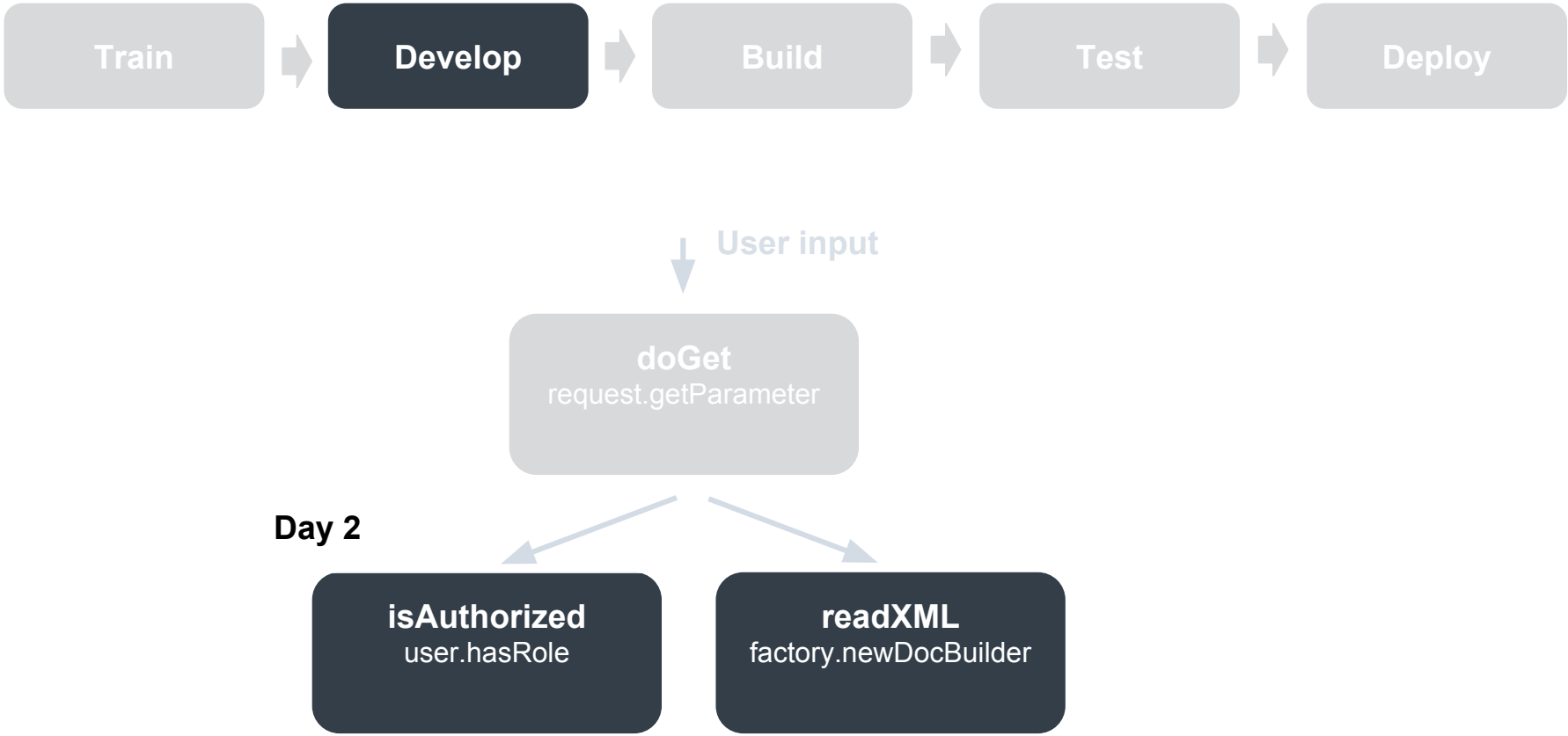
Code analysis



Code analysis

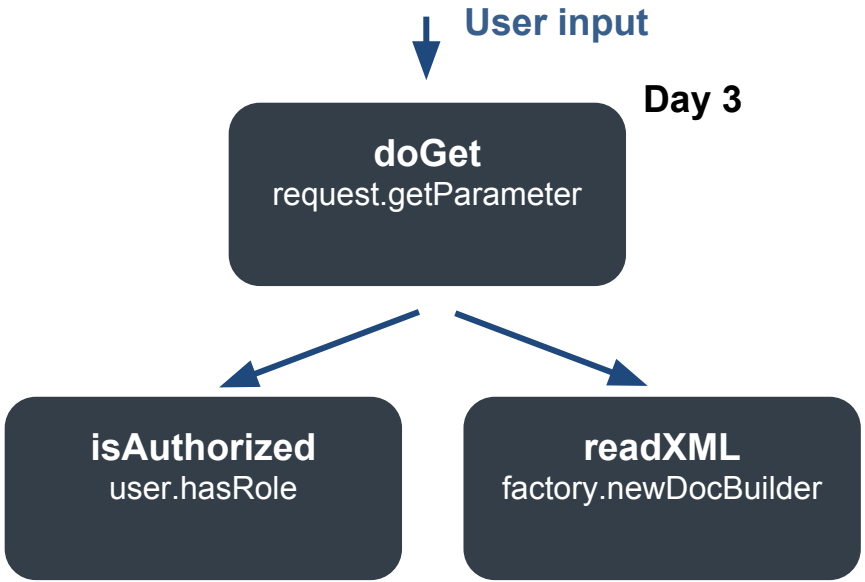


Code analysis



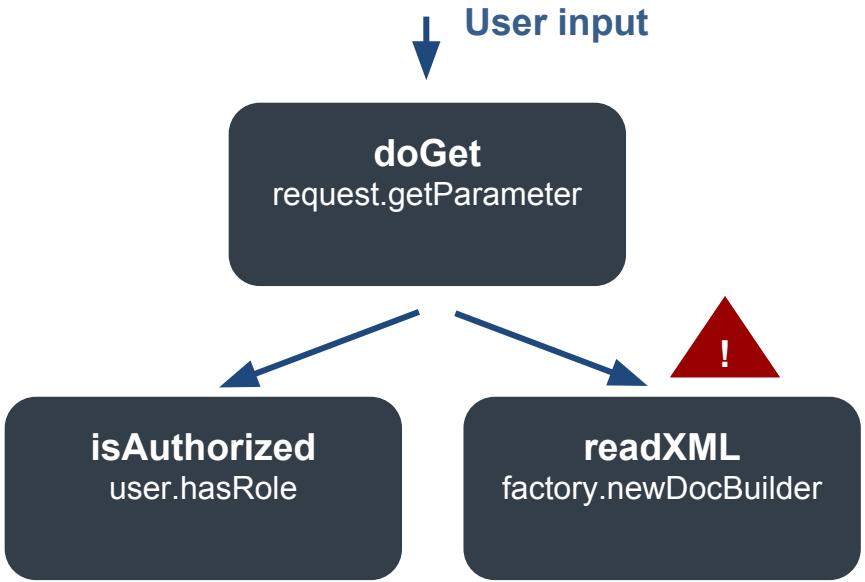
Code analysis

Train → **Develop** → Build → Test → Deploy



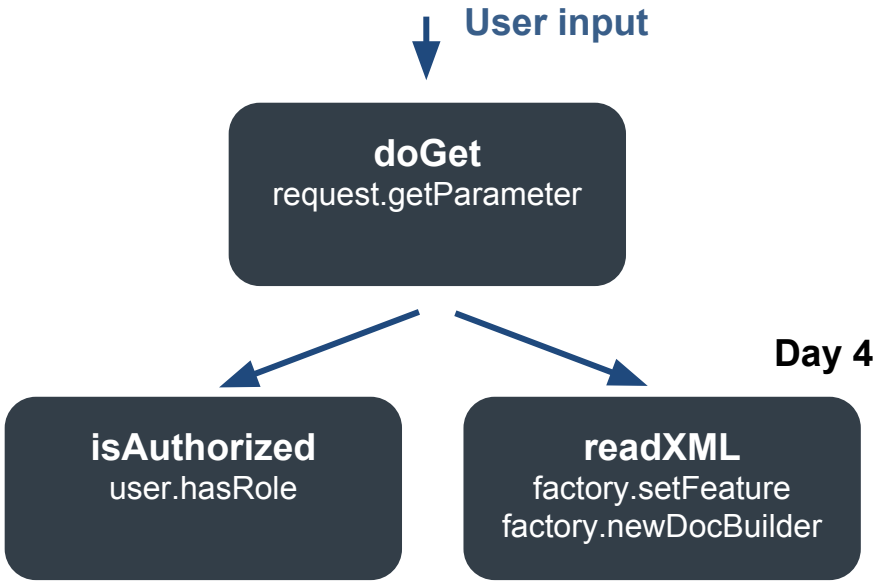
Code analysis

Train → **Develop** → Build → Test → Deploy



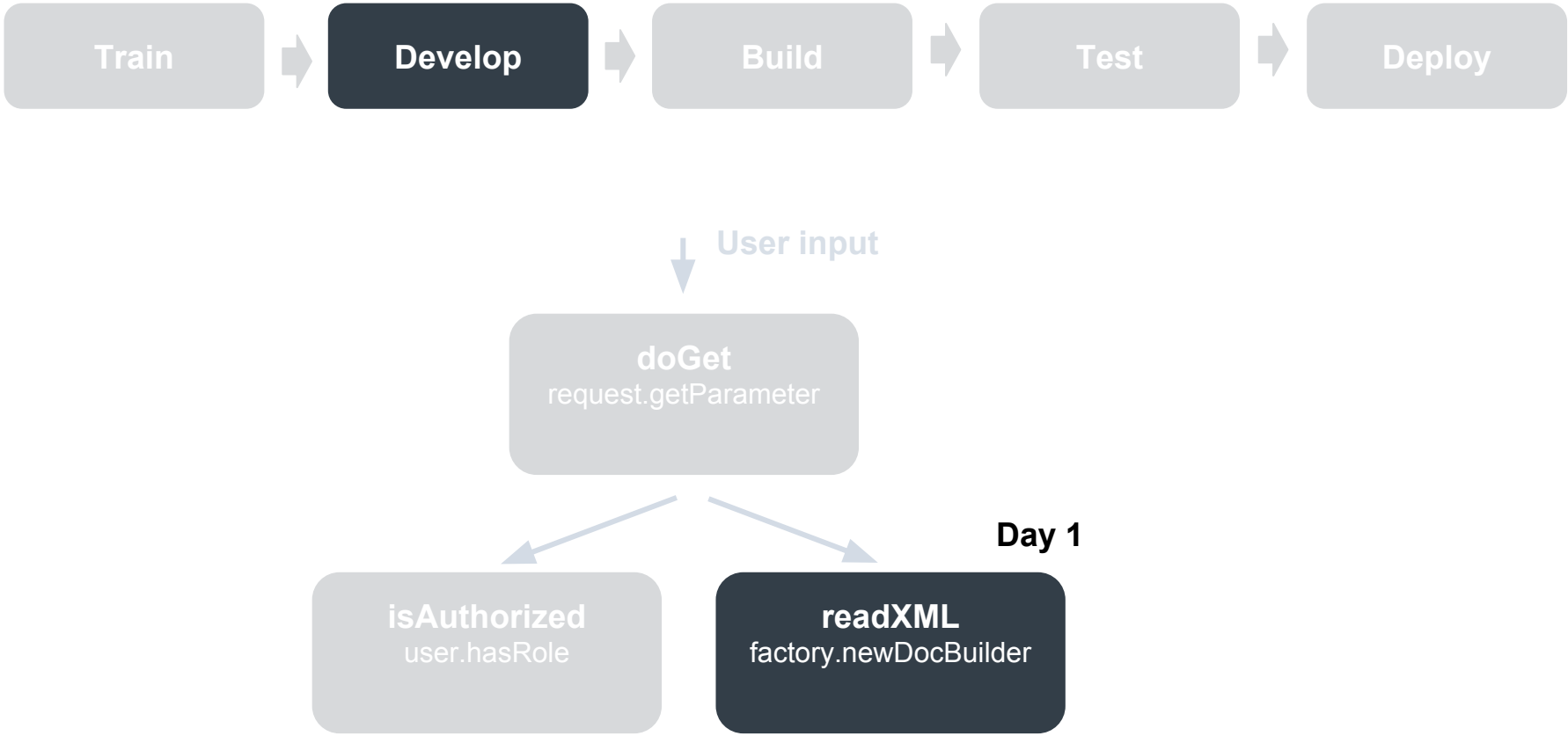
Code analysis

Train → **Develop** → Build → Test → Deploy

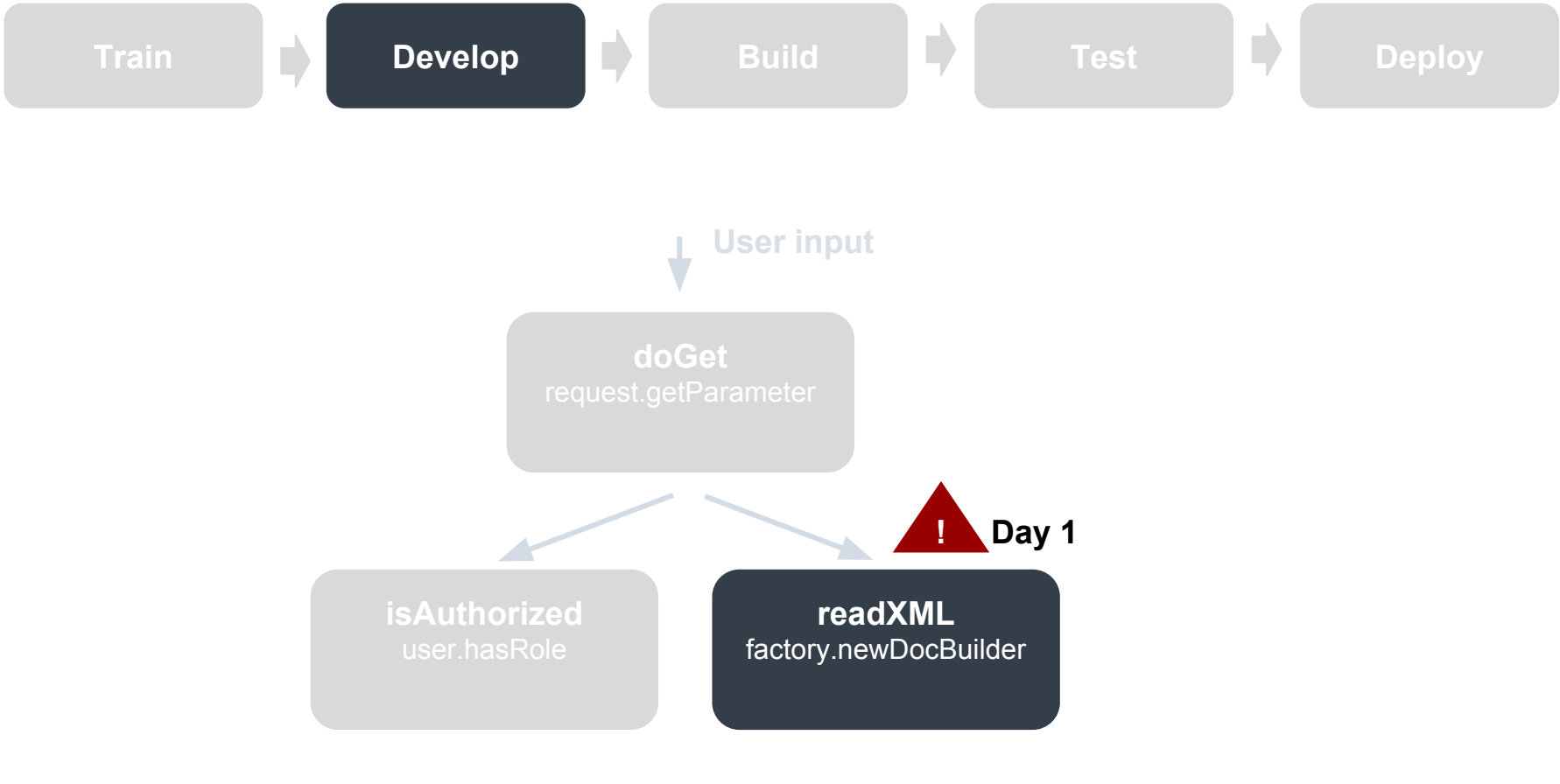


Keep shifting left?

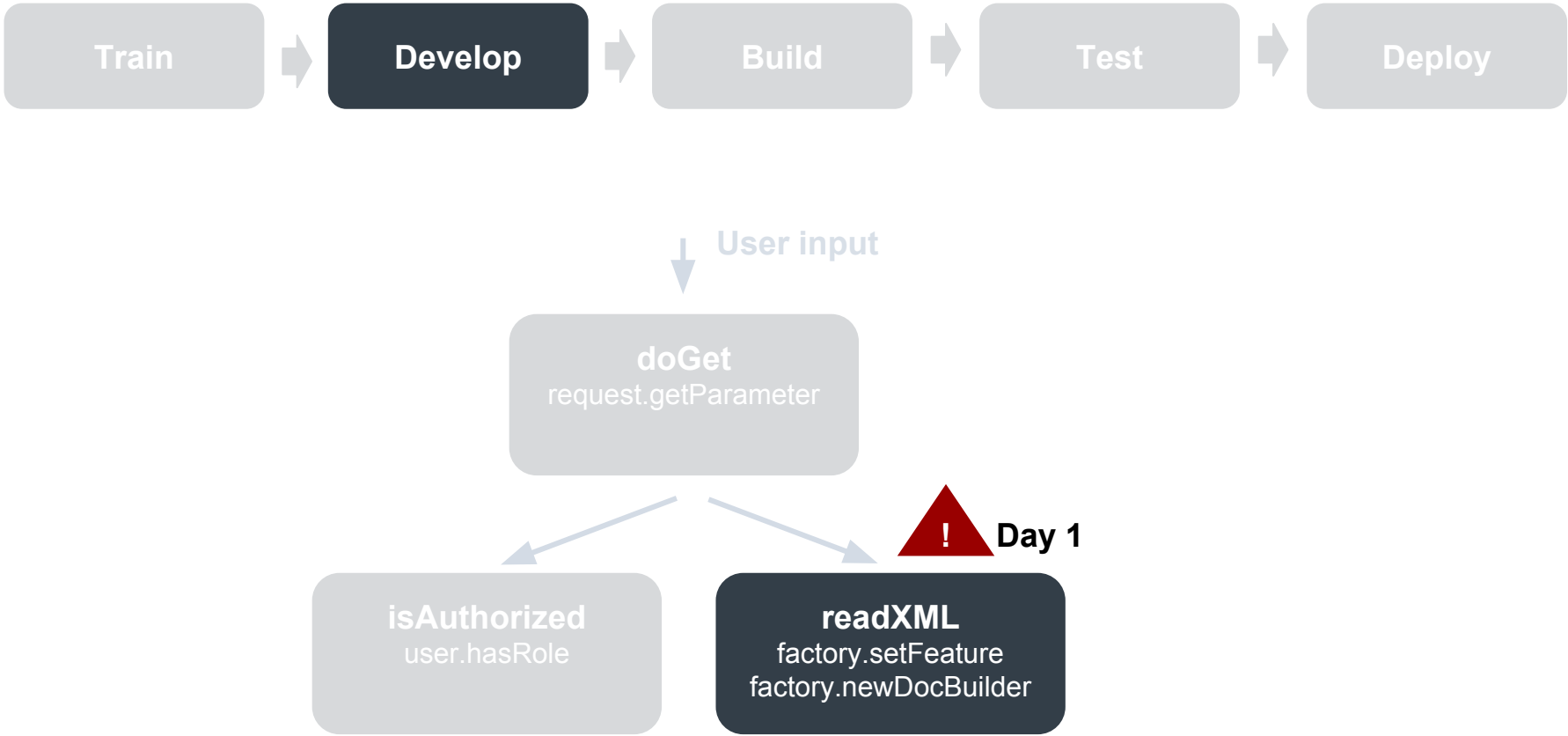
Code analysis



Code analysis

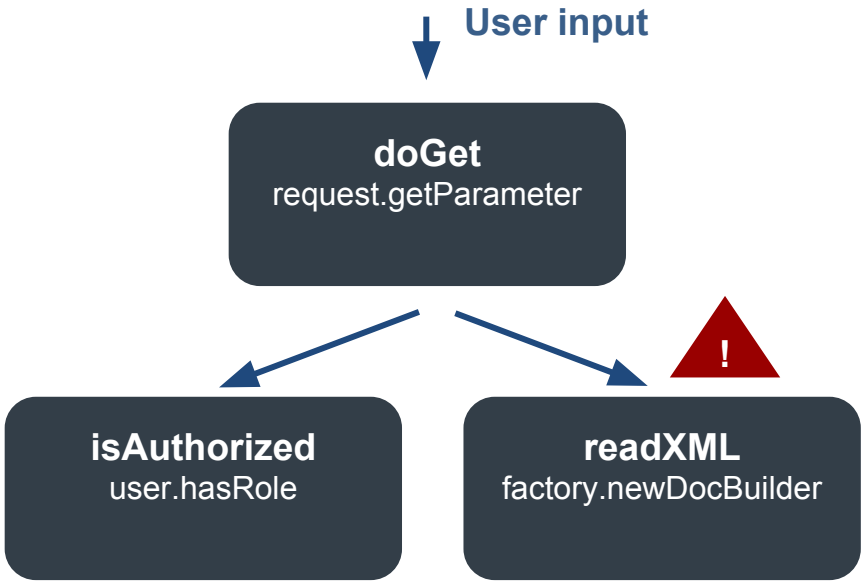


Code analysis

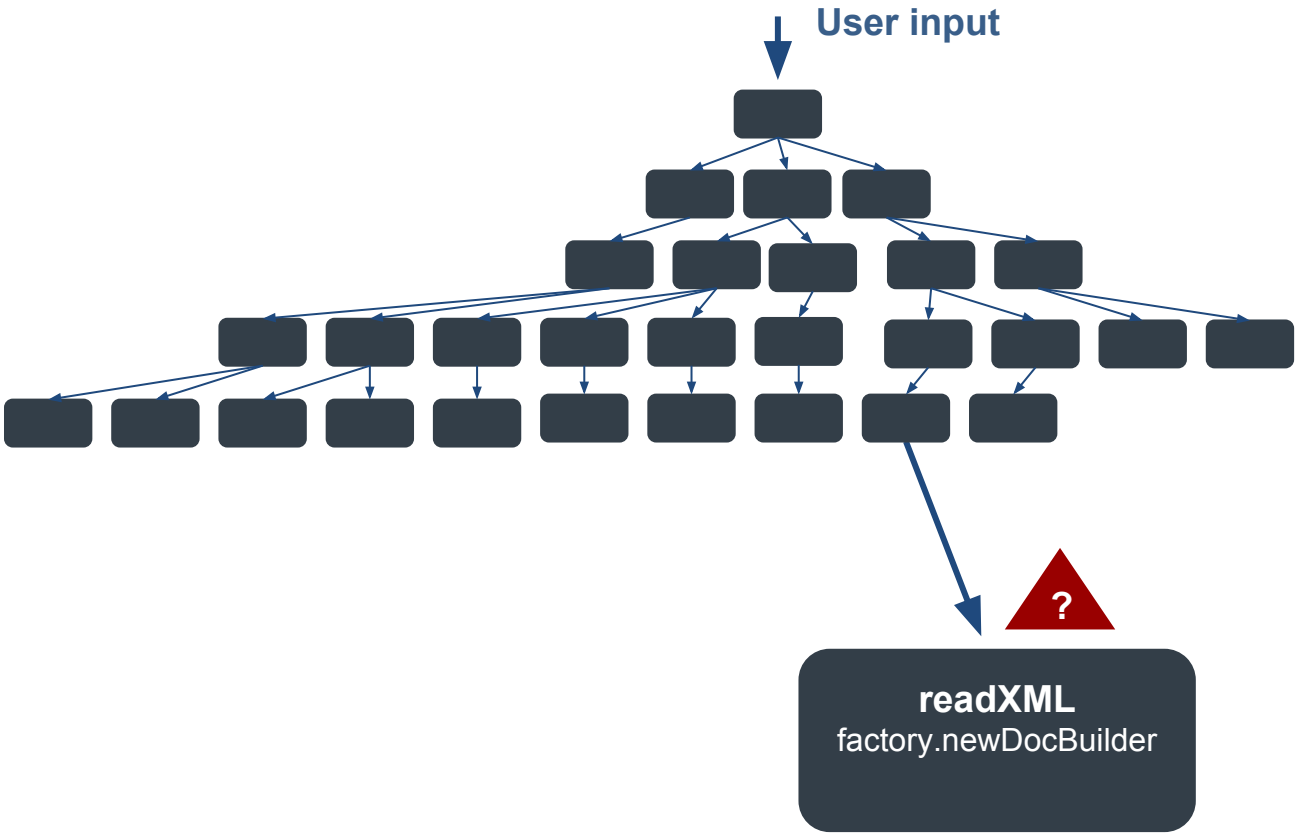


> Other advantages for coding guidelines

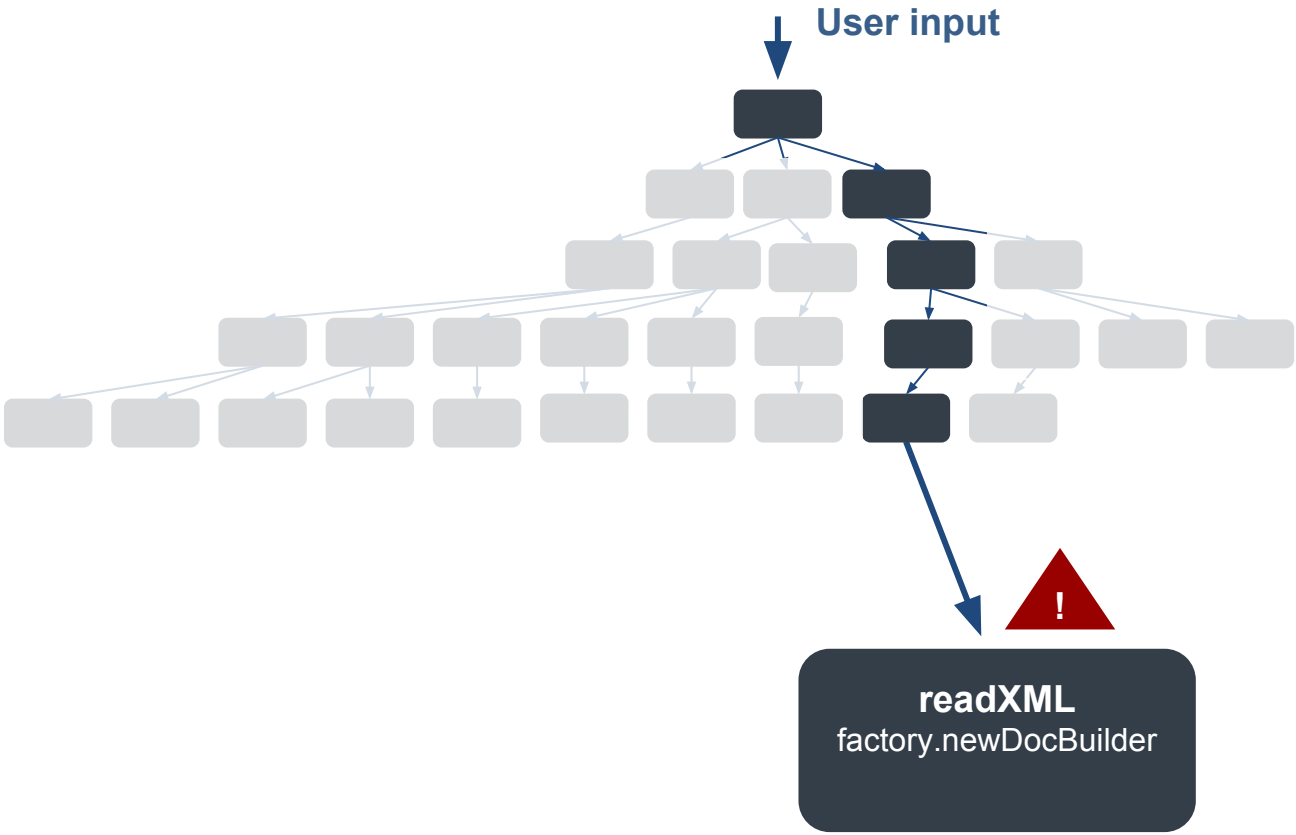
Scalable


User input
doGet
request.getParameter
!
isAuthorized
user.hasRole
readXML
factory.newDocBuilder

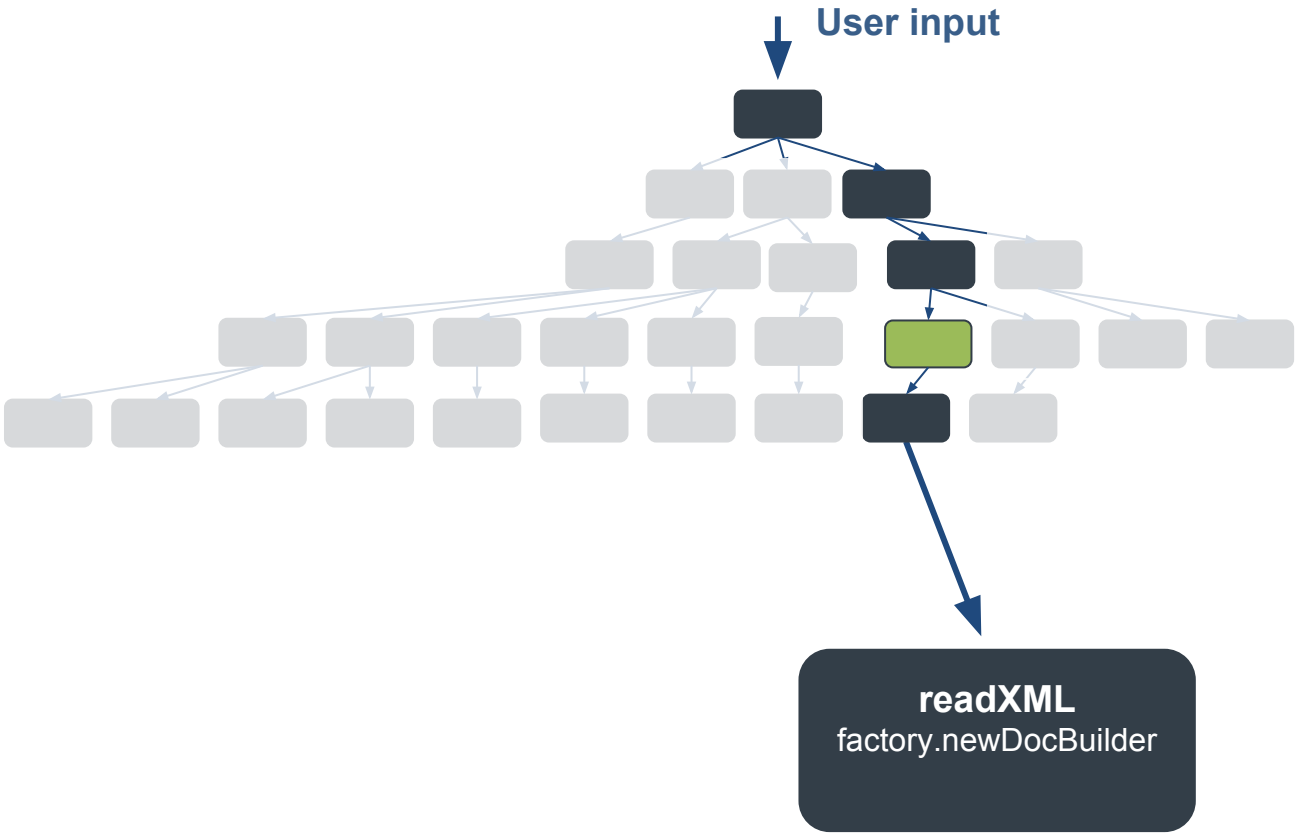
Scalable



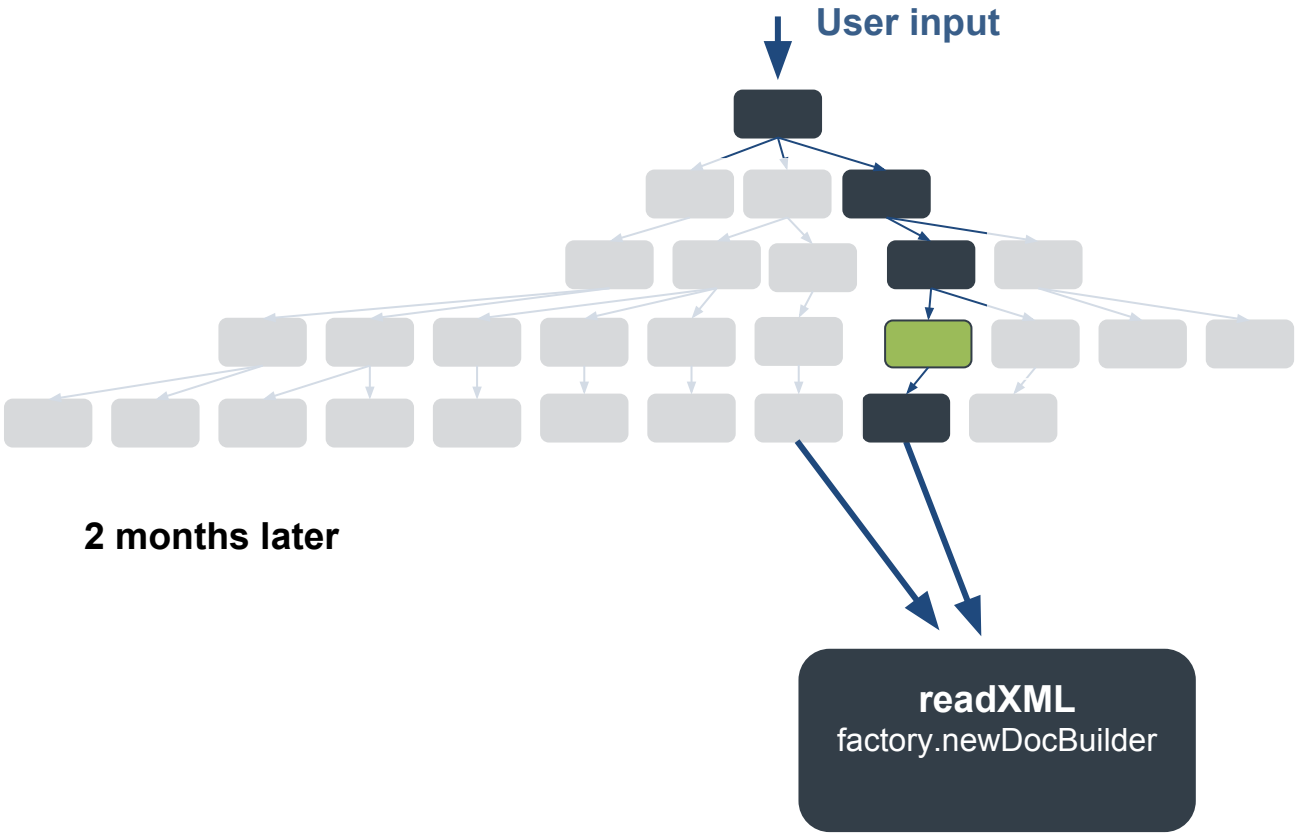
Scalable



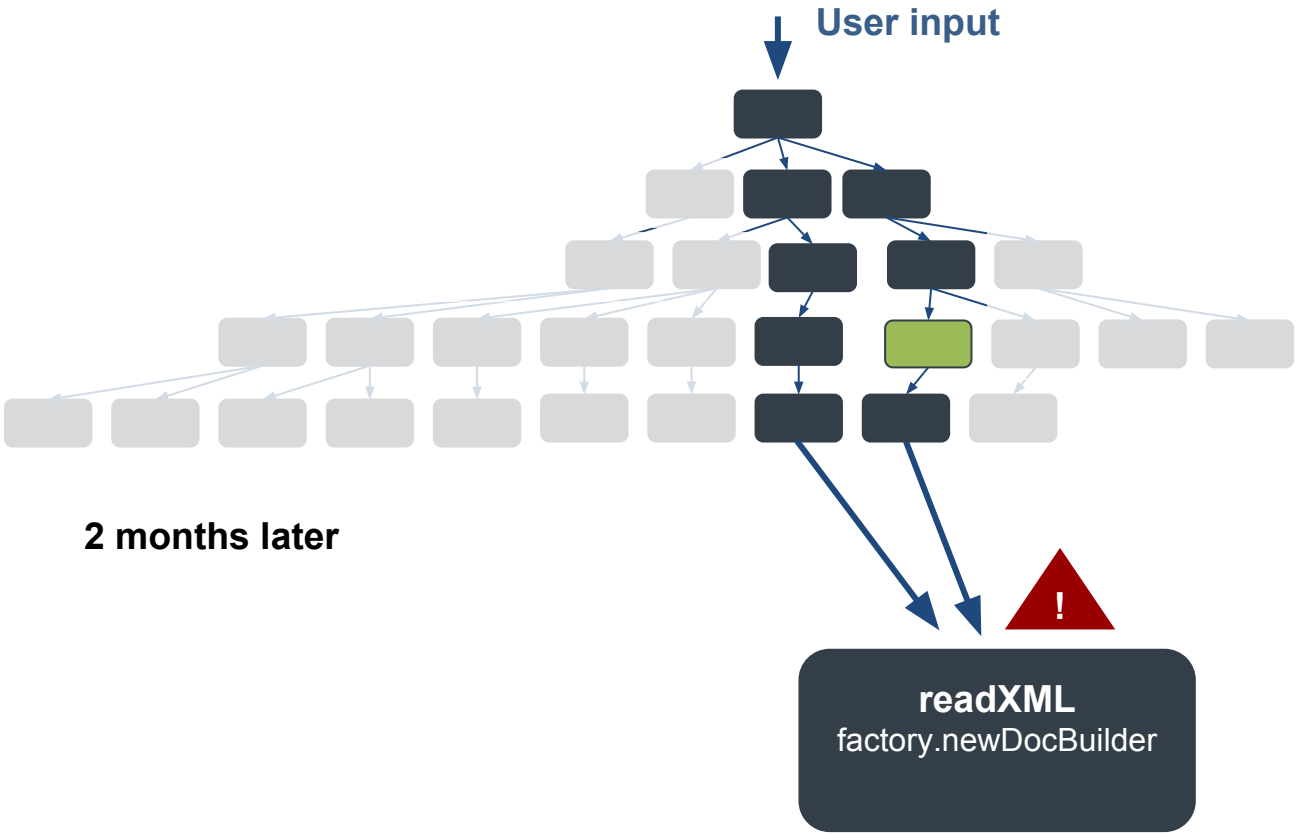
Uniformity



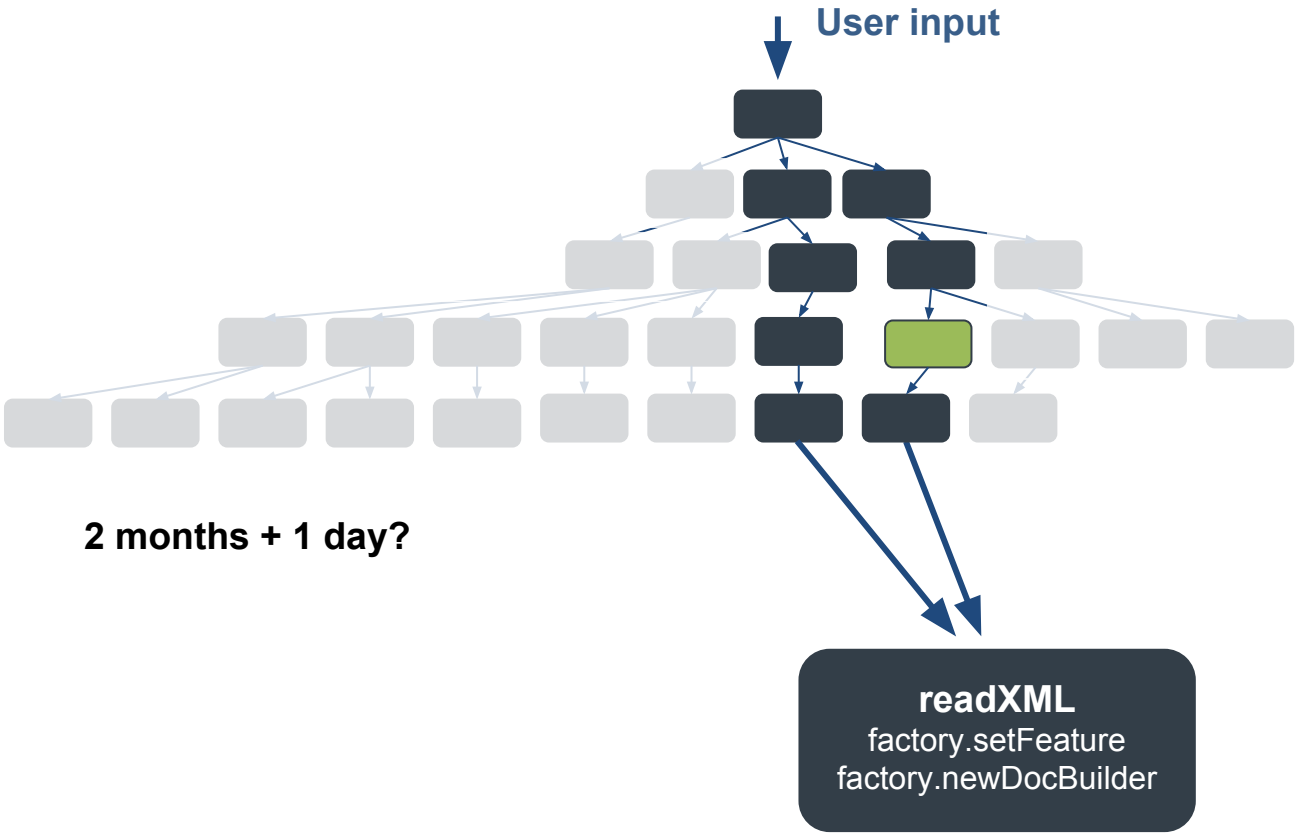
Protect from future use



Protect from future use



Protect from future use



2 months + 1 day?

> Final case for tool-based support

You

Coding guideline

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

factory.setFeature( name:                                                    , value: true);

DocumentBuilder builder = factory.newDocumentBuilder();
```

?

Secure coding guideline

On **`DocumentBuilderFactory`**

call **`setFeature`** with these parameters

before calling **`newDocumentBuilder`**

Coding guideline

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

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

DocumentBuilder builder = factory.newDocumentBuilder();
```

Secure coding guideline

On **`DocumentBuilderFactory`**

call **`setFeature`** with these parameters

before calling **`newDocumentBuilder`**

SECURE CODE
WARRIOR

APPLICATIONSECURITYINSIGHTS.SECURECODEWARRIOR.COM

SECURECODEWARRIOR.COM

@SECCODEWARRIOR

LINKEDIN.COM/COMPANY/SECURE-CODE-WARRIOR

FACEBOOK.COM/SECURECODEWARRIOR/