Lecture 2: Web Security

- Initially, world wide web consisted of only websites
  - Communication only from server to client
  - Authentication of users not required
  - Attackers cannot get users' sensitive information
- Today most sites on the internet are web applications
  - Two-way communication between the browser and server
  - Support registration, login, financial transaction, etc
  - Most information sent to users is private and sensitive
- Some web application examples:
  - online banking, shopping, Social networking, Web search, …
- Web application code found at:
  - *Client (browser)*: Javascript
  - *Server*: Java, PHP, ASP, Ruby
- Web application security is a big issue
  - Users' sensitive data should not leak
  - Adequate input validation and/or encoding required

In this course (OWASP Web Top 10 for 2013): Injection, Broken Authentication and Session Management, XSS, CSRF.
OWASP: The Open Web Application Security Project, an online community, produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security.

Javascript in Web pages

- Most web attacks occur due to malicious Javascript code
- Javascript
  - Creates interactive web pages at the client-side
  - Can be executed before, during and after page is loaded/viewed
  - Enable asynchronous communication using asynchronous JavaScript and XML (AJAX)
  - Accesses the document object model (DOM), cookie, …
- Language features
  - *Dynamic typing* (variables have not static type)
    - `var animal = new Animal();`
  - *Dynamic code generation* (eval, Function, setInterval, …)
    - `var cookie =eval("docum"+"t.cookie");`
  - *Prototype-based inheritance*
    - `var cat =Animal.prototype;`

Embedded to web pages in several ways

- Inserting in the script element
  - `<script> alert("hi") </script>`
- Adding as file in the src attribute of script
  - `<script type = "text/JavaScript" src = "file.js" />`
- Event handler attributes
  - `<body onload = "alert('hi')"> … </body>`
- Using javascript: keyword
  - `<img src = "javascript: alert('hi')"/>`

- JS is basically the language of the whole web. Other client-side languages are dead. Web-Assembly is coming up these days.
- **Why can JS be attacked so easily?** JS was not meant to be a server-side language, also it wasn't created having security in mind. You can introduce new code at runtime e.g. with eval, Takes a string and excecutes as it is source code. There are so many different places where you can add/embed js.

Same Origin Policy (SOP)

- Restricts scripts' access to resources from other origins
  - Script embedded on a page can only access resources from the same origin with the page
  - Script's real origin not relevant to SOP
- Two web pages have the same origin if:
  - They have the same protocol, domain, and port
  - E.g., for the URL http://company.com/page.html the table below illustrates the SOP result

URL	Outcome	Reason
http://company.com/page1.html	Sucess	
http://company.com/dir/other.html	Sucess	
https://company.com/page2.html	Fail	Different protocol
http://news.comany.com/page3.html	Fail	Different domain
http://company.com:90/page4.html	Fail	Different port

- SOP applied only on the client-side (Browser)
  - Cookie sent with the request from the browser
  - Cookie is data from website stored on the browser
  - It contains personal information of the user
  - Server-side code does not send cookie to other websites
- Cookies are used for
  - *Session management:* e.g, logging into websites
  - *Personalization:* important to show relevant content to the user
  - Tracking browsing habits (used by advertisements)
- SOP for cookies is based on *domain* and *path*
  - Cookie writing considers domain
  - Cookie reading considers both domain and path
  - Secure cookie reading considers protocol (https) as well

To make the JS vulnerabilities a little bit more secure SOP was introduced. This policy says that you cannot get information about a different web site in your browser. Why is that important? If I have 2 tabs open, one is my bank account and the other is smth else, then I do not want to get the other side and information about my bank account. They should be clearly separated. This is what SOP tries to enforce.
It's not really consistent and not working for everything. The SOP ist just some best effort, it works but not super great, can be avoided in some cases. **Only applied on the client side**.

Same-origin policy

The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. It is a critical security mechanism for isolating potentially malicious documents.

In computing, the same-origin policy is an important concept in the web application security model. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of URI scheme, host name, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.

This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions. A strict separation between content provided by unrelated sites must be maintained on the client-side to prevent the loss of data confidentiality or integrity.

The same-origin policy mainly applies to data access from scripts; embedding resources across origins, such as images, CSS and scripts via the corresponding HTML tags is not restricted (with fonts being a notable exception).

History: The concept of same-origin policy dates back to Netscape Navigator 2 in 1995. The policy was originally designed to protect access to the **Document Object Model**, but has since been broadened to protect sensitive parts of the **global JavaScript object**.

Implementation: All modern browsers implement some form of the Same-Origin Policy as it is an important security cornerstone. The policies are not required to match an exact specification but are often extended to define roughly compatible security boundaries for other web technologies, such as Microsoft Silverlight, Adobe Flash, or Adobe Acrobat, or for mechanisms other than direct DOM manipulation, such as XMLHttpRequest.

Security Applications

The same-origin policy helps protect sites that use authenticated sessions. The following example illustrates a potential security risk that could arise without the same-origin policy. Assume that a user is visiting a banking website and doesn't log out. Then, the user goes to another site that has some malicious JavaScript code running in the background that requests data from the banking site. Because the user is still logged in on the banking site, the malicious code could do anything the user could do on the banking site. For example, it could get a list of the user's last transactions, create a new transaction, etc. This is because the browser can send and receive session cookies to the banking site based on the domain of the banking site.

The user visiting the malicious site would expect that the site he or she is visiting has no access to the banking session cookie. While it is true that the JavaScript has no direct access to the banking session cookie, it could still send and receive requests to the banking site with the banking site's session cookie. Because the script can essentially do the same as the user would do, even CSRF protections by the banking site would not be effective.

Every time you send a request the answer contains a cookie, saying this is a certain person for example. Each web site can store a little bit of data on your browser saying this is the person I was interacting with.

It has to be clear on the **server-side** that the same cookie is only sent to one person, not more than one.

Cookies are used for – Session management: When you login to a website and when next time you access the same web server, that web server needs to know that is still you. But **HTTP** is **stateless**. So it doesn't know you ever contacted/visited that web site. The only way to make that clear is by sending additional information from the browser back to the server that "proofs" that you are the same person. **This is a cookie**, it's a secret information that nobody else knows about and only those people that have the cookie can use the service/website the way you are allowed to use it.

This means if someone is able to steal your cookie, he can do the same things as you do, since the server **won't** check if the cookie comes from the same browser.

Even if you are not logged in, sites still use cookies. Most website that use cookies has to adhere to some EU-Union regulation that they have to tell you that they use cookies. Only option is to say “ok”. Basically it is used for some kind of a personalization, so when you come back it will know that last time you were looking for a certain thing and maybe they will use that information in a meaningful way, suggestions etc.

This is particularly interesting for advertisers, they will track what you are doing on the internet. They can even do that in multiple websites that you are browsing and will know everything about you. What google, facebook etc is doing. What you were searching yesterday, with whom did you interact and so on, then they will show you ads e.g. All of this is possible with those cookies.

- The same that applies for the web page directly is also true for cookies. Cookies **cannot** be accessed by some other websites, otherwise they could steal your credentials (bank account).

➢ You have the option to say that cookies can only be sent to the server if you are using **https**. Because if the cookie is sent over the internet without encryption then everybody could steal that cookie and pretend that they are you. There is an option nowadays on browsers, telling it not to send a cookie without using **https**.

Secure cookies are a type of cookie that are transmitted over encrypted HTTP connections. When setting the cookie, the Secure attribute instructs the browser that the cookie should only be returned to the application over encrypted connections. The secure attribute does not protect the cookie in transit from the application to the browser; both Firefox and Internet Explorer allow cookies with the Secure attribute to be set over HTTP.

To fully protect a cookie, the HttpOnly and SameSite attributes should also be applied to the cookie. The HttpOnly protects the cookie from being accessed by, for instance, JavaScript, while the SameSite attribute only allows the cookie to be sent to the application if the request originated from the same domain.

An **HTTP cookie** is a small packet of data that is sent from a web server to a user's web browser. Since HTTP is a stateless protocol, it cannot relay information from one page to the other and so there was a need of a cookie. There are two types of cookies:

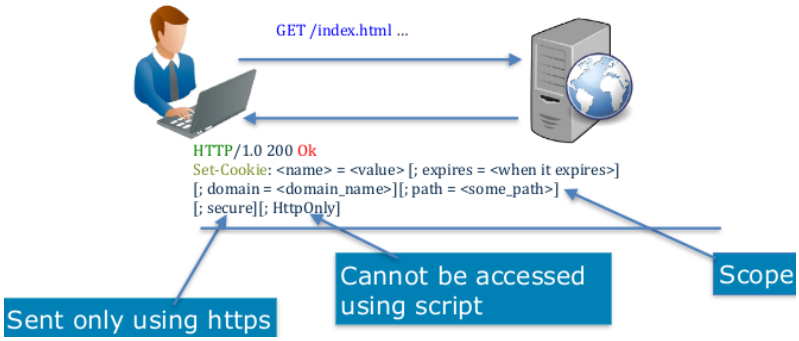
- **Persistent cookies** - Cookies that store information in user's browser for a long time.
- **Non-persistent cookies** - Cookies that generally expire once the browser is closed.

The cookies could contain sensitive information such as passwords and credit card numbers. These are sent over an HTTP connection and are stored in web browsers as plain text, and so can be targeted and be used by attackers to steal the information stored in it. To prevent such information exposure cookies are secured with attributes.

Various cookie hijacking techniques exist. All the methods are not difficult to implement and can do a significant damage to a user or an organization.

- **Network threats** Cookies that are sent over unencrypted channels can be subject to eavesdropping, i.e. the contents of the cookie can be read by the attacker.
- **End system threats** Cookies can be stolen or copied from the user, which could either reveal the information in the cookies or allow the attacker to edit the contents of the cookies and impersonate the users.
- **Cookie harvesting** The attacker can try to impersonate a website by accepting cookies from the users. Once the attacker gets the cookies, he can use these harvested cookies for websites that accept third-party cookies.

The Set-Cookie HTTP response header is used to send cookies from the server to the user agent.



SOP for writing cookies

- **Domain**:
  - Current resource's top domain and its sub domains
  - Not for another domain, and top-level-domain(TLD).
- **Path**: any path
- E.g., which cookie domains can be set by docs.foo.com/accounts

Allowed Domains	Disallowed domains
.foo.com/	user.foo.com/
docs.foo.com/	other.com/
	.com/ (TLD)
	uni-potsdam.de/

Setting the **HttpOnly** flag on a cookie makes it a little bit safer since JS cannot access that cookie.

But this is only enabled if the web server tells you so! If not any JS can access.

SOP for reading cookies

- Let request be: GET //Domain/Path
  - *Cookie domain:* must be domain-suffix of Domain
  - *Cook-path:* is path-prefix of Path
  - *Protocol:* must be https if secure
- Assume $cookie_1$ and $cookie_2$ set by docs.foo.com/accounts
  - Cookie1: name = n_1; value = v_1; domain = docs.foo.com; path = /accounts; secure
  - $Cookie_2$: name = n_1; value = v_1; domain = .foo.com; path = /;
- Which cookies can be accessed by these sites?

http://docs.foo.com/	$cookie_2$
http://user.foo.com/	$cookie_2$
https://docs.foo.com/	$cookie_1$ and $cookie_2$
https://docs.foo.com/otherpath	$cookie_2$

Limitation of SOP

- SOP does not restrict sending
  - Active content (script) can send anywhere

Large web sites

- A company may have many sub domains
  - customers.company.com, products.company.com
  - SOP does not allow http://customers.company.com to read resources from products.company.com.
  - Can be solved by setting cookie domain to company.com in both

Cross origin communication

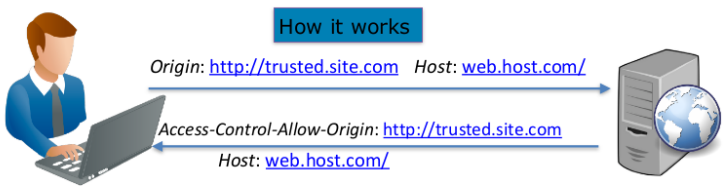
- An application may want to access from other trusted origins
- SOP does not support this
- Solution: Apply cross origin resource sharing (CORS) policy

SOP does not restrict sending. E.g. you can always send out a request for an image, and nobody will check if the image is on another domain. That means if you can send out a request to some other server, you can encode something in the URL that has additional information, leaking info to the internet.

If your server allows it. **Cross-Origin Resource Sharing (CORS)** is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin. A web application makes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.

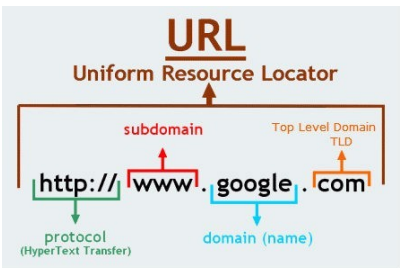
CORS

- Allows controlled cross-origin communication
- Extends HTTP with a new
  - *Origin*: request header and
  - *Acess-Control-Allow-Origin*: response header
- The response header explicitly lists the origins that may request a file or uses wildcard (*).
  - If wildcard is used the file can be requested by any site



→ Your browser is now allowed to also access something from the “trusted.site.com”.

This way SOP allows it!
This way your browser will do everyting as if it were the same domain



Origin header: where its coming from and **Access-Control-Allow-Origin** is also a header.

For security reasons, browsers restrict **cross-origin HTTP requests** initiated from within scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request HTTP resources from the same origin the application was loaded from, unless the response from the other origin includes the right CORS headers. **(?)**

Cross-site scripting (XSS)

- web application security vulnerability
  - attackers penetrate into web applications.
- Bypasses the same-origin policy
  - Origin of script is origin of its embedding page
  - Actual script origin not analyzed
  - Script can send anywhere
- Enables attackers to inject malicious scripts into trusted sites
  - Access sensitive information (username, password, credit card number, cookies, etc)
- Involves three parties:
  - The attacker, client and website.

XSS attack vectors

Examples

- Script Tag
  - <script> alert("XSS"); </script>
- Img tag/ src attribute
  - <img src = "javascript: alert('XSS')">
- Iframe tag
  - <iframe src = "http://evil.com/xss.html">
- Background attribute
  - <table background = "javascript: alert('XSS')">
- Href attribute
  - <link rel = "stylesheet" href = "javascript: alert('XSS')">

Injection attacks basically work by adding additional scripts somewhere into a web page. The art for the attacker is to find out how and where he can inject it

Types of XSS

- **Reflected (non-persistence) XSS**
  - Injected script reflected off the web server
  - E.g., in error message, search result, or other response
- **Stored (persistence) XSS**
  - Injected script permanently stored on the target servers
  - E.g., in database, message forum, comment field, etc
- **DOM-based XSS**
  - Malicious data reflected by JavaScript code on the client side without touching the web server.

Reflected XSS

- Attacker forces the user to access a malicious link
  - Send by email, blog comments, link in a banner ad
- Bug (lack of sanitization) in the trusted website reflect the malicious script to the user
  - Same origin policy cannot prevent the attack
  - The website is now the origin for the script
- The script can exploit the website content
  - Access cookies, session token, and other sensitive information.

Cross Site Scripting is a vulnerability that allows an attacker to inject JavaScript code into a website, so that it originates from the attacked website from the browser point of view.

This can happen if user input is not sufficiently sanitised. For example a search function may display the string "Your search results for [userinput]". If [userinput] is not escaped an attacker may search for:

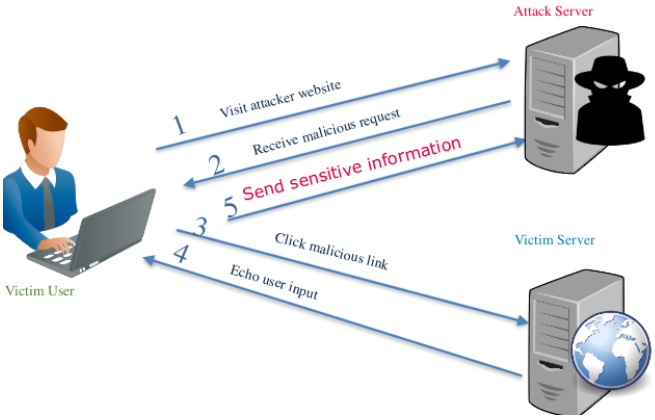
```
<script>alert(document.cookie)</script>
```

The browser has no way to detect that this code was not provided by the website owner, so it will execute it. Nowadays cross site scripting is a major issue, so there is work done to prevent this vulnerability. Most notable is the Content Security Policy approach.

- **Sadece XSS yapılan internet sitesinin bilgileri çalınabilir**. Böyle değil: 2 tab açık, **banka** ve **shop.com**. **Shop.com** xss vulnerabilitysi var. Ben linke tıkladım, shop.com'a request gönderildi. Shop.com'un serverında yeni bir HTML page generate ediliyor (mesela php ile, yada java). O xss scriptli olan HTML sayfası bana response olarak geri gönderiliyor. Ve shop.com'dan gelen bir script olduğu için bizim broswer (same origin'den geldiği için policy) kabul ediyor ve o scrip çalışınca shop.com ile alakalı her türlü bilgiye ulaşma imkanı oluyor. **Ancak yan sekmedeki banka verilerine ulamaşamaz çünkü SOP, script bize shop.com'dan geldi**.

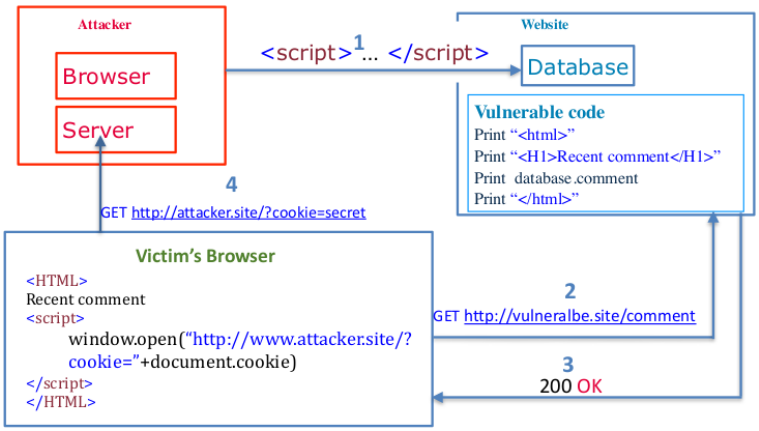
With DOM-based XSS, the advantage is that any kind of filtering on the server side will be avoided, since it never goes to the server.

Stored XSS



- Attacker permanently stores script in target application
  - Using comment/input fields in social sites, blogs, forums, wikis, etc
  - E.g., <script> alert(document.cookie); </script>
- Script then executed on all visitors' browsers
- Script can be crafted to send users' cookies or other sensitive information to the attacker

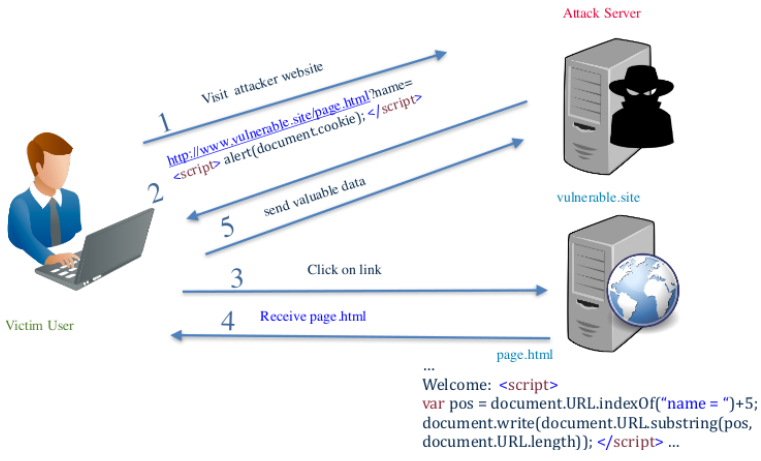
DOM-based XSS



- Advanced fully client side type of XSS attack
  - Attacker code not embedded in response page
  - May not even be sent to the server
  - Difficult to detect using firewalls and server log analysis
- Attack script executed in the victim's browser
  - By modifying the DOM "environment".
- DOM objects that can be used as sources for this attack:
  - document.url, document.hash, document.referrer, ...
- DOM object used as sink includes:
  - document.write, element.innerHTML, element.src, eval,...

DOM Based XSS simply means a Cross-site scripting vulnerability that appears in the DOM (Document Object Model) instead of part of the HTML. In reflective and stored Cross-site scripting attacks you can see the vulnerability payload in the response page but in DOM based cross-site scripting, the HTML source code and response of the attack will be exactly the same, i.e. the payload cannot be found in the response. It can only be observed on runtime or by investigating the DOM of the page.

The Document Object Model is a convention for representing and working with objects in an HTML document (as well as in other document types). Basically all HTML documents have an associated DOM, consisting of objects representing the document properties from the point of view of the browser. Whenever a script is executed client-side, the browser provides the code with the DOM of the HTML page where the script runs, thus, offering access to various properties of the page and their values, populated by the browser from its perspective.



Server does not use the script, but your code on the client side uses the parameters after ? In a wrong way. E.g it looks for the "name" parameter and writes that name on the webpage. But if its not your name and its a script then it writes the script to the webpage.

In this example the server still sees the attack, even though they dont use it!

There are ways to encode a URL such that you don't send certain parts of it for example. (instead of **?** Using **#** gibi)

Or you can manipulate some other resources in the browser that will be used as input

→ There are many ways, some of them won't be sent to the server.

`textContent` changes just the text. **`innerHTML`** can also add different text as well as text.

What is the DOM?

The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page.

A Web page is a document. This document can be either displayed in the browser window or as the HTML source. But it is the same document in both cases. The Document Object Model (DOM) represents that same document so it can be manipulated. The DOM is an **object-oriented representation** of the web page, which can be modified with a scripting language such as JavaScript.

DOM XSS is a type of cross site scripting attack which relies on inappropriate handling, in the HTML page, of the data from its associated DOM. Among the objects in the DOM, there are several which the attacker can manipulate in order to generate the XSS condition, and the most popular, from this perspective, are the document.url, document.location and document.referrer objects.

The **W3C DOM** and **WHATWG DOM** standards are implemented in most modern browsers. Many browsers extend the standard, so care must be exercised when using them on the web where documents may be accessed by various browsers with different DOMs.

For example, the standard DOM specifies that the getElementsByTagName method in the code below must return a list of all the <P> elements in the document:

```
var paragraphs = document.getElementsByTagName("P");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

All of the properties, methods, and events available for manipulating and creating web pages are organized into objects (e.g., the document object that represents the document itself, the table object that implements the special HTMLTableElement DOM interface for accessing HTML tables, and so forth). This documentation provides an object-by-object reference to the DOM implemented in Gecko-based browsers.

DOM and JavaScript

The short example above, like nearly all of the examples in this reference, is JavaScript. That is to say, it's written in JavaScript, but it uses the DOM to access the document and its elements. The DOM is not a programming language, but without it, the JavaScript language wouldn't have any model or notion of web pages, HTML documents, XML documents, and their component parts (e.g. elements). Every element in a document—the document as a whole, the head, tables within the document, table headers, text within the table cells—is part of the document object model for that document, so they can all be accessed and manipulated using the DOM and a scripting language like JavaScript.

In the beginning, JavaScript and the DOM were tightly intertwined, but eventually, they evolved into separate entities. The page content is stored in the DOM and may be accessed and manipulated via JavaScript, so that we may write this approximative equation: API (HTML or XML page) = DOM + JS (scripting language)

The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API. Though we focus exclusively on JavaScript in this reference documentation, **implementations of the DOM can be built for any language**, as this **Python** example demonstrates:

```
# Python DOM example
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # DOM property of document object
p_list = doc.getElementsByTagName("para")
```

How Do I Access the DOM?

You don't have to do anything special to begin using the DOM. Different browsers have different implementations of the DOM, and these implementations exhibit varying degrees of conformance to the actual DOM standard (a subject we try to avoid in this documentation), but every web browser uses some document object model to make web pages accessible via JavaScript.

When you create a script--whether it's inline in a <script> element or included in the web page by means of a script loading instruction--you can immediately begin using the API for the **`document`** or **`window`** elements to manipulate the document itself or to get at the children of that document, which are the various elements in the web page. Your DOM programming may be something as simple as the following, which displays an alert message by using the alert() function from the window object, or it may use more sophisticated DOM methods to actually create new content, as in the longer example below.

This following JavaScript will display an alert when the document is loaded (and when the whole DOM is available for use):

```
<body onload="window.alert('Welcome to my home page!');">
```

Another example. This function creates a new H1 element, adds text to that element, and then adds the H1 to the tree for this document:

```
<html>
  <head>
    <script>
       // run this function when the document is loaded
       window.onload = function() {

         // create a couple of elements in an otherwise empty HTML page
         var heading = document.createElement("h1");
         var heading_text = document.createTextNode("Big Head!");
         heading.appendChild(heading_text);
         document.body.appendChild(heading);
       }
    </script>
  </head>
```

A Typical Example of a DOM XSS Attack

Let's take the basic example of a page which provides users with customized content, depending on their user name which is encoded in the URL, and uses their name on the resulting page: In this case the HTML source of http://www.example.com/userdashboard.html would look like this:

```
<html>
<head>
<title>Custom Dashboard </title>
...
</head>
Main Dashboard for
<script>
        var pos=document.URL.indexOf("context=")+8;
        document.write(document.URL.substring(pos,document.URL.length));
</script>
...
```

The result of **http://www.example.com/userdashboard.html?context=Mary** would be a customized dashboard for Mary, containing the string "**Main Dashboard for Mary**" at the top. The malicious script can be embedded in the URL as follows

```
http://www.example.com/userdashboard.html?context=<script>SomeFunction(somevariable)
http://www.example.com/userdashboard.html#context=<script>SomeFunction(somevariable)
```

- Whereas traditional XSS takes advantage of vulnerable back-end CGI scripts to directly emit the code into served pages, DOM-based XSS takes advantage of vulnerable JavaScript scripts which execute directly in the user's browser. **Writing user input directly into the document content without escaping**.

Furthermore, the victim's browser receives the above URL and sends a HTTP request to http://www.example.com, receiving the static HTML page described above. Then, the browser starts building the DOM of the page, and populates the document.url property, of the document object with the URL containing the malicious script.

When the browser arrives to the script which gets the user name from the URL, referencing the document.url property, it runs it and consequently updates the raw HTML body of the page, resulting in

```
...
Main Dashboard for <script>SomeFunction(somevariable)</script>
...
```

Next, the browser finds the malicious code in the HTML body and executes it, thus finalizing the DOM XSS attack. In reality, the attacker would hide the contents of the payload in the URL using encoding so that it is not obvious that the URL contains a script.

Note however, that some browsers may encode the < and > characters in the URL, causing the attack to fail. However there are other scenarios which do not require the use of these characters, nor embedding the code into the URL directly, so these browsers are not entirely immune to this type of attack either.

Başka bir örnek

Mesela basit bir websiten var. item name ve price giriyosun. send'e basınca tabelada gözüküyor yazdıkların. Bunlar servera gitmiyor! Senin inputun JS ile alınıp DOM manipule ediliyor ve senin yazdığıın şeyler o şekilde ekranda gözüküyor. Sen oraya bir script tagi koyarsan, eğer HTML'in içindeki JS senin girdiğin şeyleri kontrol etmez ise (escaping/sanitize) o zaman onu olduğu gibi DOMa yazıyor. **Document.write()** dediği zaman HTML fileına yazıyor. Yani senin browserın o scripti excecute ediyor. Cookieler vs çalınabilir o sayede.

- Genel olarak bir HTML içindeki JS, DOMu user inputlar ile manipule ediyorsa (O html sayfasına dynamic olarak değiştiriyor ise) ve o user inputları kontrol etmiyorsa, DOM based XSS vulnerabilitysi olabilir.

How is DOM XSS different?

Using the above example, we can observe that:

- The HTML page is static, and there is no malicious script embedded into the page, as in the case of other types of XSS attacks; T
- he script code never gets to the server, if the "#" character is used; it is seen as fragment and the browser does not forward it further.

Hence server-side attack detection tools will fail to detect this attack; in some cases, depending on the type of the URL, the payload might get to the server and it may be impossible to hide it. Hence, the main characteristics of XSS, as reviewed in the introduction of this article, are not valid in the case of DOM XSS. Instead, DOM XSS exploits inappropriate manipulation of the associated DOM objects and properties in the client-side code.

Defending against DOM XSS attacks

Since the root of the problem still resides in the code of the page, this time the client-side code, the same sanitization and prevention techniques apply, but in this case the code review as well as the implementation of sanitization functionality needs to be performed on the client-side code.

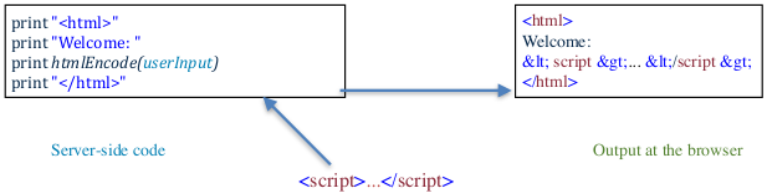
- **InnerHTML** = ... is vulnerable because tags can be written to the page as well as text. If **innerText** would be used it can prevent <script> elements written to the page.

DOM-based XSS

- The attacker inserts malicious script after the file name
  - http://www.vulnerable.site/page.html?name= <script> alert(document.cookie); </script>
  - The script still arrives at the server
  - Can be detected and prevented at the server
- If ? is replaced by # the script is never sent to the server
  - # tells the browser to remove everything after it from the request
  - Browser directly write the fragment after # to the DOM
  - Server based detection method can not prevent it

Encoding

- Escapes the html, javascript, css or URL special characters
  - E.g., characters < and > are encoded into < and >



- The script code interpreted as data after escaping
- Encoding can also be done at the client-side using javascript
- e.g., node.textContent = *userInput*, element[attribute] = *userInput*, etc

Preventing XSS Attacks

- XSS attacks exist due to malicious user inputs
- Handling user inputs against some specifications is required.
  - Form fields, cookies, headers, hidden fields, etc
- Fundamental input handling techniques:
  - Input encoding/escaping
  - Validation
- **Encoding**
  - Escape user input so that browsers interpret it as pure data
  - Input not executed by browser
- **Validation**
  - Removes all malicious parts of the input using some filters

Both methods share three important features

- **Context** Go to
  - Where the input rendered (html element, attribute, CSS value, etc)
  - E.g., in <input name = "user input"> an attacker can enter "><script>...</script><input value=" as input
  - Result: <input name = ""><script>...</script> <input value = "">
  - Filtering quote prevents attack in this context but may not in others
- **Inbound/Outbound**
  - User input can be handled at the entry or exit of the website.
  - Outbound handling is preferable as it considers the context
- **Client/Server**
  - Client-side or sever-side input handling

There are different ways to escape inputs, depending on where you will use it.

Escaping (encoding) rules (OWASP)

- *Html escaping*
  - <body>…escape untrusted data before inserting here…</body>
  - Same for other html elements
- *Attribute escaping*
  - content
- *Javascript escaping*
  - <script>alert('...escape untrusted data...')</script>
  - <div onmouseover="x='...escape untrusted data...'"
- *CSS escaping*
  - text
- *Url escaping*
  - link

Limitation of encoding

- Can not prevent attack using javascript:
  - E.g., document.querySelector('a').href = javascript: alert('xss')
- Inappropriate to define custom html in a page
  - It becomes plaint text when encoded

Validation

- Validation/filtering complements the limitation of encoding
  - By removing active/malicious contents (blacklisting) or
  - By specifying what is allowed(whitelisting)
  - whitelisting is recommended

Blacklisting

- Specifies list of forbidden input parts
  - E.g., forbids/filters javascript:, script, etc
    document.querySelector('a').href = javascript: alert('xss')

Invalid

- Has two limitations
  - ***Complexity***: difficult to specify all malicious contents
    - e.g., <scr<scriptipt scr= "" is filtered out to <script scr = ""
  - ***Staleness***: new feature that allow malicious content may appear

Whitelisting

- Describes allowed input patterns
  - Any input with different patter is invalid
- Two advantages over blacklisting
  - ***Simplicity***: defining allowed pattern is easy
  - ***Longevity***: new features added to browser does not affect it

Limitation: When you url encode < > and html tags then you cannot include custom HTML tags in this case, like forums / mySpace. Adding images to your custom website e.g.

Usually **white listing** is better.

Validation outcome

- Two decisions can be performed when the input is invalid
  - *Rejection*: totally reject the input
  - *Sanitization*: remove invalid part of the input and use the remaining as input

Disadvantage of input handling

- Both encoding and validation can not fully prevent XSS
  - Single fail of security can comprise the website
  - Content Security Policy (CSP) resolves this problem

Sanitization is difficult to get right, rejection is easier

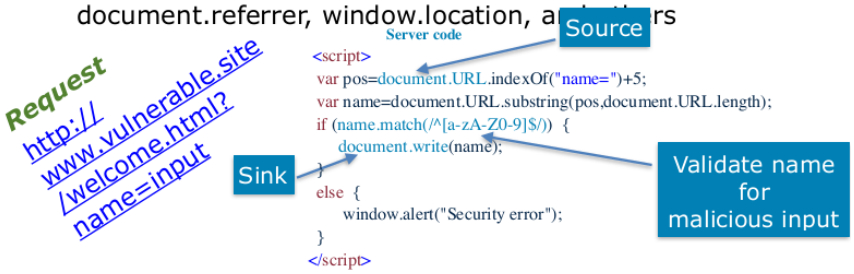
Content Security Policy (CSP)

- Instructs the browser the location/type of resource to load on a page
- Some of the rules that can be enforced by CSP are:
  - *No untrusted source*: load from clearly defined trusted sources
  - *No inline resource*: does not evaluate inlined javascript and CSS
  - *No eval:* does not allow javascript eval function

- Syntax of CSP
  - Content-Security-Policy: directive source-expression, source-expression, …; directive ...; …
- E.g., Content-Security-Policy: default-src: 'self'; script-src: 'self' other.domain
  - Browser loads javascript from the page's origin and other.domain,
  - It loads all other resources only from the page's origin

DOM based XSS defense

- Document rewriting, redirection, or other sensitive action using client side data should be avoided.
- DOM object references that can be affected by an attacker must be analyzed carefully
  - Includes document.URL, document.location, document.referrer, window.location, a[illegible]rs



  - but, can be fooled if attacker sent request: http://www.vulnerable.site/welcome.html?notname=<script> alert(document.cookie); </script> &name=Joe
- Apply strict intrusion prevention system (IPS) policy
  - e.g., restrict the number of parameters
  - In this case only one parameter (*name*)

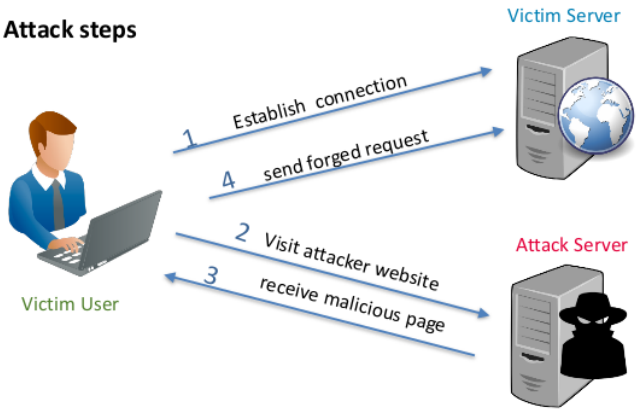
- Use HTTPOnly cookie flag

  Set-Cookie: <name> = <value>…[; HttpOnly]
  - if HttpOnly is included to to response header client-side javascript can not access the cookie

- Implement Content Security Policy

Cross-Site Request Forgery (CSRF)

- Attacker tricks an authenticated user to access website or click URL link that contains malicious request
  - Uses the trust of the website on the user's browser
  - The browser send the user's session cookie
  - The attacker impersonates the user
- Aims at state change request on the server
  - E.g., changing the victim's email address or password, transfer money, or shopping online.



CSRF Examples

Using HTTP GET

- Attacker posts image/link in forum or send by email to victim
  - <img src = "http://bank.com/transfer.php?acct=Attacker&amount=1000">
  - view profile!
- Victim logs into bank and forget to sign out
  - Victim clicks the image or link
  - Browser sends cookie and payment is accomplished

Using HTTP POST

- Attack using img/link not possible with POST method
- The attacker creates hidden form
  - Victim visits attacker's page containing this form

```
<form action = "http://bank.com/transfer.php" method = "POST">
<input type = "hidden" name = "acct" value = "attacker"/>
<input type = "hidden" name = "amount" value = "1000"/>
<input type = "submit" name = "View my pictures"/>
</form>
```

- Browser submit the malicious form and cookie of the victim
  - Uses javascript to automatically submit the form.

```
<body onload = "document.forms[0].submit()">
<form ...
```

Basically tricks a user who is already authenticated (to some website) into doing some actions on that website that will benefit the attacker in a way. Generally there will be a malicious link that the user clicks.

When you are authenticated/logged in etc, then your browser will store a cookie (token?) of that website, helping you automatically authenticate so you don't have to enter your password every time in a new page. Your requests each will contain that session cookie, it will be sent to the server so the server knows its still you. (Those cookies have different expiration dates: Facebook's is forever, Sparkasse is 12 mins and so on.)

→ This means **anybody** that can send a Request including your session cookie, can do the exact same things as you do. You have to be logged in meaning the session cookie that you were using needs to be still valid(not expired) in order for it to work. The attacker does not steal your cookie, does not see the cookie at all.

The attacker just assumes/hopes that you are currently logged in to facebook/bank/etc. and when you click the malicious link your browser sends your cookie directly to fb/bank and does that action that the attacker wants. (changing email address to the attackers, transfering money to the attacker, deleting you account etc)

Different ways of doing it:

- **Using GET**: an image or link contains a get request to the site which you are assumed to logged in.
- **Using POST**: You need to visit attackers website. The Html document will contain a **pre-filled invisible form** and when the page is loaded, your browser will automatically send that form without you even knowing about it.

Login CSRF

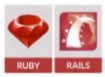
Login CSRF example: attacker can prepare a form on his website, fill it with his own google id and password, set the method to post, action to google.com.login and when you load that page, you will log yourself to the attackers account. From now on the attackers can see every action you are doing on google services.

- Attacker forges login request to an honest site
  - Attacker's username and password are used
  - User logged in as the attacker
  - Server mutate session-cookie using **Set-Cookie** header
- Some effect of login CSRF:
  - *Search history*: User's search queries stored on attacker's search history
    - Attacker can spy the user by retrieving the queries
  - *Paypal*: User's credit card or bank account added to attackers payPal account
    - While purchasing/funding using PayPal account

CSRF Defence

Secret Validation Token

- Add secret and unique validation token in the request



```
<input type = "hidden" name = "token"
value = "KbyUmhTLMP1P3qmLlkPt"/>
```

Custom HTTP Headers

- Sites can send custom HTTP headers only to themselves using XMLHttpRequest, but not to others.

X-Requested-By: *XMLHttpRequest*

Referer validation header

- The **Referer** provides the URL of the actual requesting site

facebook **Referer: http://www.facebook.com/home.php**

Origin header

- Browser modified to send request's origin in an origin header
  - Improves the ***Referer*** header by respecting user's privacy

A lot of frameworks use **Secret Validation Token**. Whenever you try to start a request, the server sends you a page with a secret token. In order to make that request you have to copy that data and send it back to the server, meaning you need to send back the same token that they provided. (Can be bypassed)

Referer validation: Where this request is actually coming from? The browser adds that information as an additional header when making a request to somewere. (not a good solution)

Custom headers: You can have your own HTTP headers. You can only send some custom headers only to your own website. This way you can make sure that the request is coming from your own page and not by another page.

Origin header:

Cross-Site Scripting is not necessary for **CSRF** to work. However, any cross-site scripting vulnerability can be used to defeat token, Double-Submit cookie, referer and origin based CSRF defenses. This is because an XSS payload can simply read any page on the site using a XMLHttpRequest and obtain the generated token from the response, and include that token with a forged request. This technique is exactly how the MySpace (Samy) worm defeated MySpace's anti-CSRF defenses in 2005, which enabled the worm to propagate. XSS cannot defeat challenge-response defenses such as Captcha, re-authentication or one-time passwords. It is imperative that no XSS vulnerabilities are present to ensure that CSRF defenses can't be circumvented. Please see the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.

What is a "Cookie"?

A cookie is a small piece of text stored on a user's computer by their browser. Common uses for cookies are authentication, storing of site preferences, shopping cart items, and server session identification.

Each time the users' web browser interacts with a web server it will pass the cookie information to the web server. Only the cookies stored by the browser that relate to the domain in the requested URL will be sent to the server. This means that cookies that relate to www.example.com will not be sent to www.exampledomain.com.

In essence, a cookie is a great way of linking one page to the next for a user's interaction with a web site or web application.

What is a "Session"?

A session can be defined as a server-side storage of information that is desired to persist throughout the user's interaction with the web site or web application.

Instead of storing large and constantly changing information via cookies in the user's browser, **only a unique identifier** is stored on the client side (called a "**session id**"). This session id is passed to the web server every time the browser makes an HTTP request (ie a page link or AJAX request). The web application pairs this session id with it's internal database and retrieves the stored variables for use by the requested page.

Users can disable cookies in some browsers, if this is the case then another way needs to be implemented.

Secret validation token

- Validation token must be secrete and hard to guess
  - Attacker can not forge it
- Server rejects the request if:
  - Token is missing or
  - does not match expected value
- Validation token can defend login CSRF as well
  - Site first creates pre-session
  - Implement token-based protection
  - Transition to real session after authentication

If you set your tokens from 1 to a billion, when you see 2 request after eachother then you can probably guess what the next number will be.

Nonce: just a random number

Token designs

Session Identifier

- User's session identifier as validation token
  - Server checks if token matches the session identifier
- *Disadvantage*: attacker can access user's account
  - Users may expose web pages they view
  - The session identifier can be read from the CSRF token

Session-Independent Nonce

- Server generated random nonce stored in cookie
  - Used by trac issue tracking system
- *Disadvantage*: the cookie (including secure cookie) can be overwritten by active network attackers.

Session-Dependent Nonce

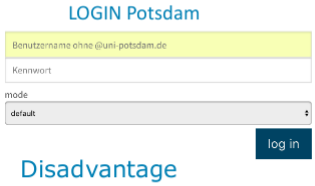
- Server binds CSRF token value to user's session ID
  - Used by CSRFx, CSRFGuard, and NoForge
- *Disadvantage:*
  - Need to maintain a large state table

HMAC of Session Identifier

- Uses cryptography to bind the session ID and CSRF token
  - E.g., Ruby on Rails use HMAC of Session ID as token
  - Attacker can not compute Session ID from the token

The Referer Header

- Differentiates same-site request from a cross-site request.



Referer: https://mailup.uni-potsdam.de/ ✓

Referer: https://atacker.site.com/ ✗

Disadvantage

- May not keep privacy of web users
  - E.g., reveals the content of search query
- Can be suppressed due to bug in browsers

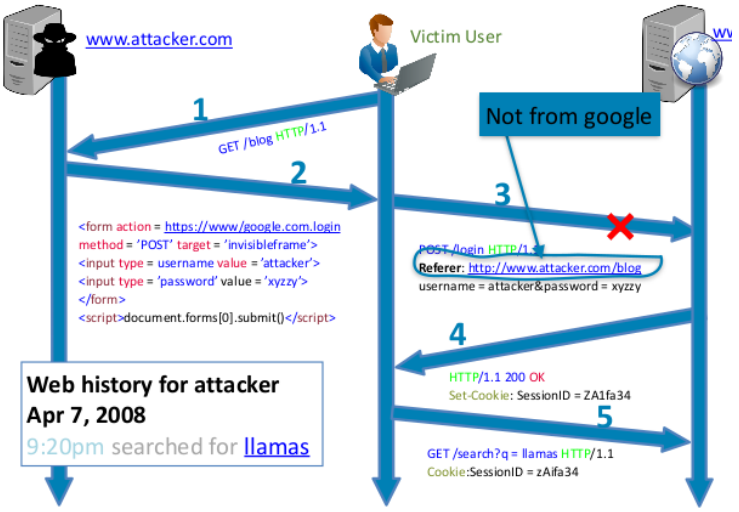
The Referer Header

Strictness

- *Lenient Referer validation (not strict)*
  - Only requests with incorrect **Referer** header are blocked
  - Requests lacking the header are accepted
  - Web attacker can easily suppress the Referer header
  - E.g., request from *ftp* and *data* do not carry Referer headers
- *Strict Referer validation*
  - Requests without **Referer** header are also blocked
  - Prevent web attackers from suppressing the *Referer*
  - But some browsers and networks suppress *Referer* headers for legitimate requests (compatibility problem)

Some browsers may not send the referer and users could also disable it due to privacy violation issues(?)

The Referer Header (Login CSRF)



Custom HTTP Headers

- Browsers use same-origin policy to send custom http header
  - Sites cannot send custom headers to other sites
- Use XMLHttpRequests to send the headers
  - Prototype.js JS library attaches *X-Requested-By* header
  - Google web Toolkit attaches X-XSRF-cookie header
- To use custom headers as CSRF defense:
  - Sites issue state-modifying request using XMLHttpRequest
  - Attach the custom header (e.g., X-Request-By)
  - Reject state-modifying requests lacking the header
  - Value of headers is not relevant

Origin Header

- Browser sends Origin header with POST requests
  - Sends the null value if not able to determine the origin
- Improves Referer header by respecting user's privacy
  - Contains only information relevant to identify the principal
    - E.g., scheme, host, and port of the document's URL
  - Unlike Referer, not sent for all requests (only POST)
- To use Origin header as CSRF defense:
  - All state-modifying requests need to be sent using POST
  - Undesired header value including null must be rejected

→ **Origin**: Only needs the scheme: https.google.com. Only POST

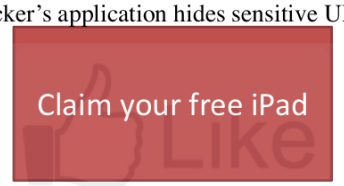
→ **Referer**: Needs the whole Url with all there parameters.

CSRF defense recommendations

- Login CSRF
  - Strict Referer validation
  - Login forms typically submitted over HTTPS and Referer header not suppressed
- HTTPS sites
  - Strict Referer validation
- Third-party Content
  - Framework such as Ruby-on-Rails or other that apply secret token validation

Clickjacking

- Applications sharing a graphical display can be vulnerable to clickjacking
- Caused by tricking the user to act out of context
  - E.g., Attacker's application hides sensitive UI by making it transparent



- User clicks transparent Facebook "Like" button
  - Overlaying on top of the "claim your free ipad" button.
  - If 100% transparent completely invisible

Clickjacking attack methods

- Compromising target display integrity
  - Hiding the target element
  - Partial overlays
  - Cropping
- Compromising pointer integrity
  - Attacker displays fake cursor away from the real pointer
- Compromising temporal integrity
  - Attacker moves target to top after victim visually confirmed, but before clicking it

Browser checks whats the top most element at the clicked point. If there is the hidden like button on the very top but **invisible**, then the user clicked that like button. Someone it is impossible to make it invisible, then people make it oppeague almost invisible.

Compromising target display integrity

Hiding The target element

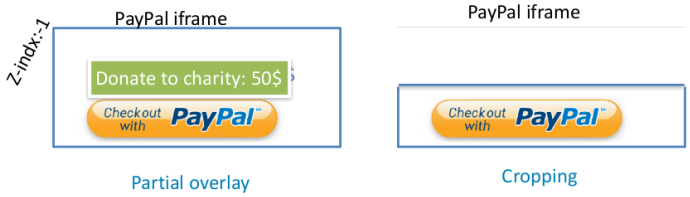
- Two ways
  - Using CSS *opacity* and *z-index* properties
  - Using CSS pointer-events:none property

Partial overlays

- Partially displaying the target to visually confuse the victim
  - Use *z-index* or *flash player* with window mode property set to wmode=direct

Cropping

- Crop target elements to show only some part of information



X, y and z is the 3 dimention: meaning what is on top.

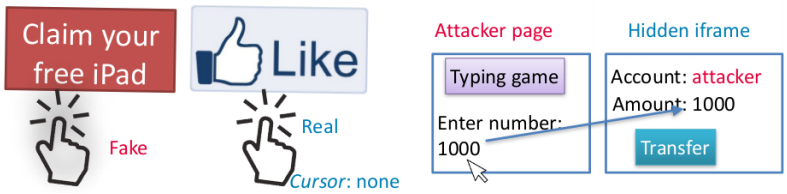
Changing opacity: make it almost invisible.

Change css Z-index: make it on top.

Pointer-event:none: when clicked dont take this element but take one below it.

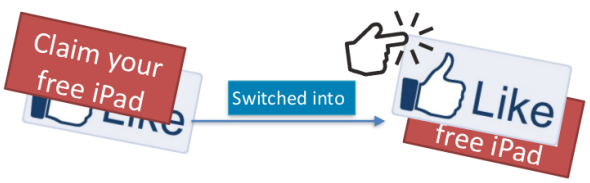
Compromising pointer integrity

- Attacker tricks the victim to click on a fake cursor pointer
  - Hide the default cursor using CSS *cursor:none* property
- Switch keyboard focus by confusing the user with blinking cursor.
  - Prevent the victim from noticing the switch



Compromising temporal integrity

- An attacker uses the time delay between visual confirmation of an element and actual click
  - E.g., attacker bring the target element on top after victim's cursor hovered on the other element
  - Can also ask double click and switch position after first click



  - The victim confirmed "claim your free iPad" is on top , but in between Facebook Like button comes top and that is clicked.

Paitnt a fake cursor so person thinks he clicks smth but in reality clicks smth else

Anti-clickjacking defences

- The same origin policy does not prevent click jacking
- The defense method for clickjacking are categorized into:
  - Protecting visual context
  - Protecting temporal context
  - Access control Gadgets

Protecting visual context

User Confirmation
- Present a confirmation prompt to user when target clicked
- *Disadvantage*: degrade user experience and is vulnerable to double click temporal attack

UI randomization
- Make harder covering the target by other elements
- But the attacker can ask the victim to click repeatedly
  - To guess the position of the target element

Opaque overlay policy
- Force all cross-origin frames to be rendered opaquely
- But break benign sites by removing transparency

Protecting visual context

Framebusting
- Disallow the target element from being rendered on frames
  - Effective defense method
- Can be achieved by:
  - Making target element top-level document or
  - Using *X-Frame-Options* and CSP's *frame-ancestors*
- *Limitation:*
  - Target elements may need to be framed by arbitrary third-party website
    - E.g., Facebook like button
  - Can also be bypassed using popup windows

Visibility detection on click
- Allow framing but disallow mouse clicks on transparent elements that show the cross-origin frame underneath
  - Vulnerable to false positives and to cursor spoofing (pointer integrity)

Protecting temporal context

- Browsers provides enough time to understand any UI changes after a dialog is displayed
  - Users must refrain from interacting until the delay expires
  - E.g., deployed in Flash Player's webcam access dialog
- Limitations
  - The time delay annoys users
  - Can also be defeated by whack-a-mole attack (discussed later)

Access control gadgets (ACG)

- Introduced by operating systems
- Grant application permissions to access user-owned resources
  - E.g., access to camera or GPS
- ACG are privileged UI that are embedded in application that need to access resources.
  - Applications require authentic users for access
  - Maintain both visual and temporal integrity
- Limitations:
  - Does not consider pointer integrity

Advanced attacks

Cursor spoofing

Compromises the pointer integrity to steal the user's webcam
- When the fake cursor move to "skip this ad" the real cursor which is hidden using *cursor*: none property rests on the allow webcam button and that is actually clicked.

Double click attack

A bait-and-switch attack against an OAuth dialog for google account which uses X-Frame-Options to prevent clickjacking attack.
- After the first click and the OAuth popup dialog is switched to top
- The cursor is positioned on the "allow access" button

Whack-a-mole attack

Combines cursor spoofing and bait-and-switch attack approaches
- User asked to play whack-a-mole game by clicking fast
- Fake cursor is used to control user attention
- Finally user switched to a "Facebook Like" button at the real cursor

InContext defense for advance attacks

- Acting out of context is the main cause of all the attacks
  - Attacker provides sensitive UI element from other applications to the user
- Ensuring context integrity is an important isolation mechanism
  - In addition to memory or resource access controls
- The InContext deferences are categorized according to:
  - *Ensuring visual integrity*
    - Guaranteeing target display integrity
    - Guaranteeing pointer integrity
  - *Ensuring temporal integrity*

Guaranteeing target display integrity

- Enforcing display integrity for all UI elements not important
  - Enforce on sensitive elements

CSS Checking

- Browser checks the CSS styles of elements
  - E.g., size, opacity, z-index
  - Makes sure the sensitive elements are not overlaid by cross-origin elements
- Can be bypassed by
  - createPopup() method or
  - Flash Player's window mode
- No need to solely rely on it

Static reference bitmap

- Websites can provide static bitmap of their sensitive element as reference
  - Browser make sure the rendered sensitive element matches the reference
  - But all browsers do not produce same bitmap for the same HTML code element.
- InContext solves this by comparing OS-level screenshot of the area containing the sensitive element when the user see it with its bitmap at the time of user action.
  - Browser should not allow zooming and rotating of sensitive elements for this to work
  - User action canceled if the two bitmaps differ

Attacker violates visual context of the Twitter Follow button by changing its opacity and obstructing it with two DIVs.
InContext detects this during its bitmap comparison

Guaranteeing pointer integrity

- Look for system-provided cursor instead of attacker simulated cursor

No cursor customization

- Disallow cross-origin cursor customization
- Additionally, disallow cursor customization on hostage and its ancestors
  - Users will always see the system cursor in the areas surrounding the sensitive element

Screen freezing around sensitive element

- Attackers can not distract users by using animations

Muting

- Sound must not draw users attention when interacting with sensitive elements
  - Better to mute speaker

Lightbox around sensitive element

- Use randomly generated mask around the sensitive element
  - Keeps the users attention
- Not static implies attacker can not use similar mask

No programatic cross-origin keyboard focus change

- After the sensitive element gets keyboard focus programatic change of focus is disallowed.

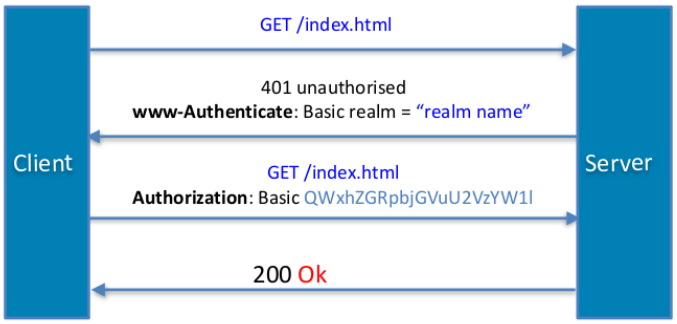
Ensuring temporal integrity

- **UI delay**
  - Visual context must be the same for some minimum time period
  - Still vulnerable to whack-a-mole attack
- **UI delay after pointer entry**
  - Strong variant of UI delay
  - Also impose delay each time the pointer enters the sensitive element
- **Pointer re-entry on a newly visible sensitive element**
  - Invalidate input events until the user explicitly moves the pointer from outside of a newly created sensitive element to inside.
- **Padding area around sensitive element**
  - User can easily identify whether the cursor is on the sensitive element or the embedding page

Sessions

- Sequence of http requests and responses associated to user
  - Keep information about the user for some period of time
  - Save access rights and other information of the user
- Can be stablished before and after authentication
  - *Before:* to track anonymous user's request
    - e.g., to maintain language preference.
  - *After:* to guarantee an authorized access to private data and increases usability.

Pre-Session (HTTP authentication)



Browser sends hashed passwords for every http request

Session management

- Tracks a user's interaction with a website during sessions
- Links authentication and access control
  - Users are authenticated first
  - All their requests/actions are tied together

- Associate each user with unique identifier
  - Session ID or token

Session ID (Tokens)

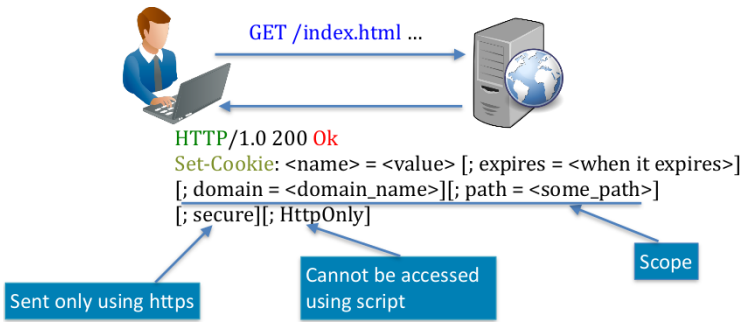
Session ID (Tokens)

Session tokens can be implemented differently

- **Store in browser cookie**
  - Set-Cookie: sessionId ="kujasdfiea"
  - But can be sent with every request including CSRF
- **Send by hidden field of a form**
  - <input type = "hidden" name = "sessionId" value = 'kujasdfiea'>
  - Relevant only for short sessions
- **Embedding on URL**
  - https://site.com/transfer?sessinId=ojldjaf
  - Can be leaked through HTTP Referer header

NB: their combination is recommended for best security

Recap: Setting a cookie



When you visit another website from google, lets say while you logged in in google, then the session id might be included in the referer header that is sent to the website.

When you tag a cookie with the **HttpOnly** flag, it tells the browser that this particular cookie should only be accessed by the server. Any attempt to access the cookie from client script is strictly forbidden. Of course, this presumes you have: A modern web browser. A browser that actually implements **HttpOnly** correctly. Aug 28, 2008

If the HttpOnly flag (optional) is included in the HTTP response header, the cookie cannot be accessed through client side script (again if the browser supports this flag). As a result, even if a cross-site scripting **(XSS)** flaw exists, and a user accidentally accesses a link that exploits this flaw, the browser (primarily Internet Explorer) will not reveal the cookie to a third party.

Token generation vulnerabilities

- Tokens generated by composing some user information
  - E.g., username or e-mail
- Can also be elements of alphanumeric sequence
  - It must be as random as possible
- Vulnerabilities in generation lets attackers predict and use it
- Examples of vulnerable token generation strategies
  - *Hidden sequences*: if normal sequence numbers used
  - *Time dependences*: if tokens are functions of time
  - Weak generation algorithms

Session management vulnerabilities

Attacker can intercept unpredictable tokens

- Exploit insecure communication between client and server
  - Secure flag of cookie is not set
- Exploit XSS vulnerabilities
  - HTTPOnly flag of cookie is not set
- Detect from log files
  - Browser, web server and server proxy logs,
- Read from browser or proxy cache
- Exploit the poor session termination policies
  - Gets enough time to test many values
- Improper use of https
  - Login using https then request using http

Session management attacks

Attacks on session can be classified as:

- Session sniffing
  - Passively intercept session data being transmitted
- Session prediction
  - Attacker guess the session
- Session fixation
  - Attacker set the session of the victim

Session Sniffing

- **HTTP packet sniffing**
  - By intercepting HTTP packet
  - Vulnerabilities: secure flag not set, does not use https
- **Log/cache sniffing**
  - By reading log/cache files
  - Vulnerabilities: token in URL parameters, hidden field, or cookie is logged to file or cached
- **XSS cookie sniffing**
  - Using XSS vulnerabilities
  - Vulnerabilities:HTTPOnly flag is not set

Session prediction

- Attacker guess the token even if:
  - Interception is not possible
  - Token generation algorithm is strong
- Two kinds of attacks
  - Token tampering (predictable)
  - Brute-force (session ID is low in entropy, e.g., less than 32 bits)
- Two enabling vulnerabilities
  - Long idle time (long session expiration time)
  - Weak implementation of session termination

Session Fixation

- Attacker fixes token before victim authenticates
- Attack steps
  - Session setup: setup session on the server and get token
  - Session fixation: introduce the token to the victim
  - Session entrance: attacker waits till user enters the session and enters at the same time
- Two session management mechanisms
  - Permissive systems: attacker chooses a token and uses it
  - Strict systems: attacker establishes a session and keeps it alive

Attack approaches

- URL parameter: sending link to victim
  - http://online.worldbank.dom/?jsessionid=1234
- Hidden field: using XSS vulnerability
  - http://online.worldbank.dom/ <script>document.cookie="sessionid=1234";</script>
  - http://online.worldbank.dom/ <script>document.cookie="sessionid=1234; domain=.worldbank.dom";</script>
    - Attacker's host is hacker.worldbank.dom
- Browser cookie: HTML tag <*META*> with *Set-cookie* attribute
  - http://online.worldbank.dom/<meta http-equiv=Set-Cookie content="sessionid=1234">

Securing session management

- **Generate strong token**
  - Use extremely large set of value
  - Strong source of pseudo randomness
- **Protect token throughout their life time**
  - Transmit it only using HTTPS
  - Do not transmit via URLs
  - Set HTTPOnly flag
  - Implement logout functionality
  - Use short expiration time
  - Prevent concurrent login
  - Use restrictive session cookie domain and path