Module 6 - HTML5

▼ Introduction, Recap & More

- HTML4, XHTML, HTML DOM Level2 ***(search)***
- Its not a single, big lang rewrite, its a collection of:
  - multimedia
  - 3d, graphics & effects
  - storage
  - semantics, device access
  - performance & integration

▼ Semantics

- it can be used to conduct new attack vectors such as XSS

Form Elements

- <keygen> → used with autofocus will trigger xss without user interaction

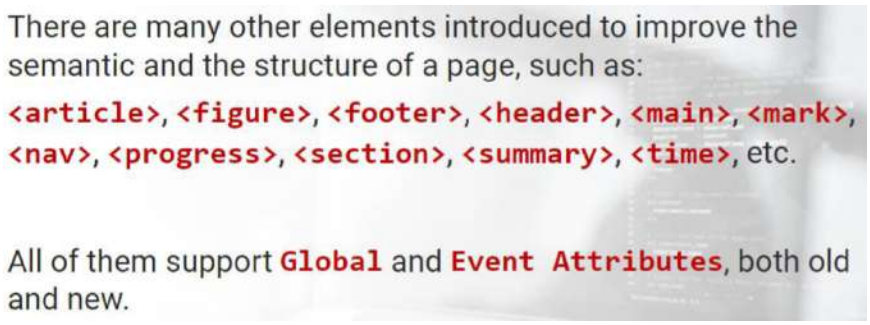
```
<!DOCTYPE HTML>
<form action="#" method="GET">
    Encryption: <keygen name="security" autofocus onfocus="
    <input type="submit">
</form>
```

Media Elements

- <video>, <audio>, <source>, <track> and <embed> are used to evade xss filters
  - cuz they support ***src*** attribute
    - eg: <embed src=”http://hacker.site/evil.swf”>

- eg: <embed src=”javascript:alert(1)”>

Semantic/Structural Elemants

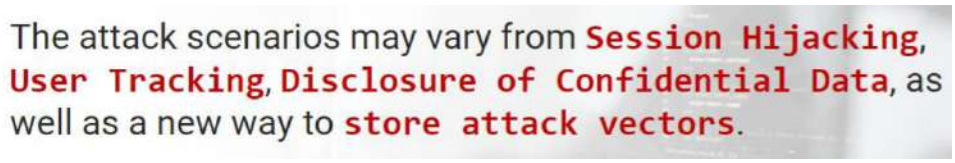


Attributes

▼ Offline & Storage

- TiddlyWiki
- Features
  - application cache and web storage (alias client-side storage or Offline storage)

▼ Web Storage > Attack Scenarios



Session Hijacking

- if dev chooses to store session IDs by using sessionStorage instead of cookie, its still possible to perform session hijacking yb levaraging

an xss flaw

- Web storage solutions **do not** implement security mechanisms to mitigate the risk of malicious access to the stored information (see **HttpOnly**)

Offline Web Application > Attack Scenario

- With offline web apps, `cache poisoning` is a critical security issue

▼ Device Access

▼ Codes

- GeoLocation API feature

```
if (navigator.geolocation) {
    // Geolocation is supported
    // Your code to use the Geolocation API goes here
} else {
    // Geolocation is not supported
    console.log("Geolocation is not supported by this
}
```

```
//The getCurrentPosition method is used to retrieve t
navigator.geolocation.getCurrentPosition(successCallb
```

```
//The successCallback function is called when the pos
function successCallback(position) {
    const latitude = position.coords.latitude;
    const longitude = position.coords.longitude;
    console.log(`Latitude: ${latitude}, Longitude: ${
}
```

```
function errorCallback(error) {
    console.error(`Error getting location: ${error.me
}
```

```
The watchPosition method is used to continuously moni
const watchId = navigator.geolocation.watchPosition(s
```

```
//The watchId returned can be used later to stop watc
navigator.geolocation.clearWatch(watchId);
```

```
//Both getCurrentPosition and watchPosition can take
const options = {
    enableHighAccuracy: true, // Use GPS if available
    timeout: 5000,            // Maximum time to wait
    maximumAge: 0             // Maximum age of cache
};

navigator.geolocation.getCurrentPosition(successCallb
```

```
//Handling User Permissions:
//The Geolocation API requires user permission to acc
navigator.geolocation.getCurrentPosition(
    successCallback,
    errorCallback,
    { enableHighAccuracy: true }
);
```

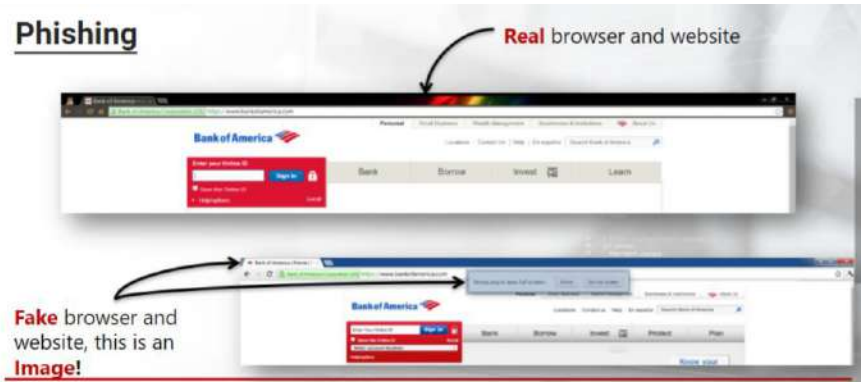
Geolocation > Attack Scenarious

- it can be used for user tracking and breaking anonymity

Another interesting feature introduced to control devices is `Fullscreen`. This is an API that allows a single element (images, videos, etc.) to be viewed in full-screen mode.

▼ Fullscreen mode > attack scenario

- the **Fullscreen API** can be used for **Phishing Attacks**



▼ Performance, Integration & Connectivity

With HTML5, many new features were introduced to both improve **performance** and **user interaction**, such as `Drag and Drop`, `HTML editing` and `Workers` (`Web` and `Shared`).

Improvements were also made on **communications**, with features such as `WebSocket` and `XMLHttpRequest2`.

Attack Scenarios

The new attack scenarios vary from `interactive XSS`, with Drag and Drop, to `Remote shell`, `port scanning`, and `web-based botnets` exploiting the new communication features like `WebSocket`.

It is also possible to manipulate the `history` stack with methods to add and remove locations, thus allowing `history tampering` attacks.

Performance, Integration & Connectivity

- from a security pov, the most important features introduced are
  - CSP
  - CORS
  - Cross-Document Messaging
  - iframes with the Sandboxed attribute

▼ Exploiting HTML5

▼ CORS Attack Scenarios

- CORS is used to relax the SOP restrictions.
- CORS is similar to flash and silverlight's cross-domain policy files
  - but instead of XML config, it uses a set of **HTTP headers**

▼ CORS Attack Scenarios

Universal Allow

- the first CORS response headers is `Access-Control-Allow-Origin`

Allow by Wildcard Value*

Allow by Wildcard Value *

Often, developers use the wildcard value * to allow any website to make a CORs query to a specific page.

Generally this is not a required behavior, but rather a matter of laziness of the implementer. This is one of the most common misconfigurations with CORS headers.

Allow by Wildcard Value *

There are several ways to abuse this implementation.

For example, if XSS is found on the page served with `Access-Control-Allow-Origin *`, it can be used to infect or impersonate users.

Allow by Wildcard Value *

This can also be used to bypass intranet websites. For example, tricking internal users to access a controlled website and making a COR query to their internal resources in order to read the responses.

Another option is to use the users as a proxy to exploit vulnerabilities, therefore leveraging the fact that the `HTTP Referrer` header is often not logged.

In CORS, the `Access-Control-Allow-Credentials` indicates that the request can include **user credentials**. This is pretty interesting if we also have the wildcard * set on the allowed origin header!

Unfortunately, the specification explicitly denies this behavior:

> *Note: The string "*" cannot be used for a resource that supports credentials.*

Allow by Server-Side

Allow by server-side

So, what can developers do? They can simply adjust the implementation server-side, allowing COR from all domains with credentials included!

```
<?php
header('Access-Control-Allow-Origin: ' + $_SERVER['HTTP_ORIGIN']);
header('Access-Control-Allow-Credentials: true');
```

Allow by server-side

By design, this implementation clearly allows CSRF.

Any origin will be able to read the anti-CSRF tokens from the page, therefore consenting any domain on the internet to impersonate the web application users.

Weak Access Control

After we have "secured" the *Universal Allow* issue, we know that we only trust certain origins.

To do this, CORS specifications provide a request header named **`Origin`**. It indicates where the COR or Preflight Request originates from.

Since the **`Origin`** header cannot be spoofed from the browser, one of the most common mistakes is to establish trust only on this header. Usually this is done in order to perform access control for pages that provide sensitive information.

Of course, the header can be spoofed by creating requests outside of the browsers. For example, one can use a proxy or using tool like cURL, Wget, etc.

Check Origin Example

Suppose that **http://victim.site** supports CORS and, not only reveals sensitive information to friendly origins (like **http://friend.site**), but also reveals simple information to everyone.

Check Origin Example

By using **cURL**, it is possible to bypass the access control by setting the **Origin** header to the allowed value: **http://friend.site**. In so doing, it is possible to read the sensitive information sent.

```
:~$ curl http://victim.site/html5/CORAccessControl.php
A normal page, nothing more ...

:~$ curl --header 'Origin: http://hacker.site' http://victim.site/html5/CORAccessControl.php
A normal page, nothing more ...

:~$ curl --header 'Origin: http://friend.site' http://victim.site/html5/CORAccessControl.php
<span style='color:red; font-weight:bold;'>Sensitive information here!</span>
```

Origin allowed to read sensitive data

Intranet Scanning

It is also possible to abuse COR in order to perform time-based Intranet scanning.

This can be done by sending XHR to either an arbitrary IP address or domain names and, depending on the **response time**, establish whether a host is up or a specific port is open.

JS-Recon

JS-Recon

For this purpose, Lavakumar Kuppan has developed **JS-Recon**. This is an HTML5 based JavaScript Network Reconnaissance tool, which leverages features like COR and Web Sockets in order to perform both network and port scanning from the browser. The tool is also useful for guessing user's private IP addresses.

Remote Web Shell

If an XSS flaw is found in an application that supports CORS, an attacker can hijack the victim session, establishing a communication channel between the victim's browser and the attacker.

That allows the attacker to do anything the victim can do in that web application context.

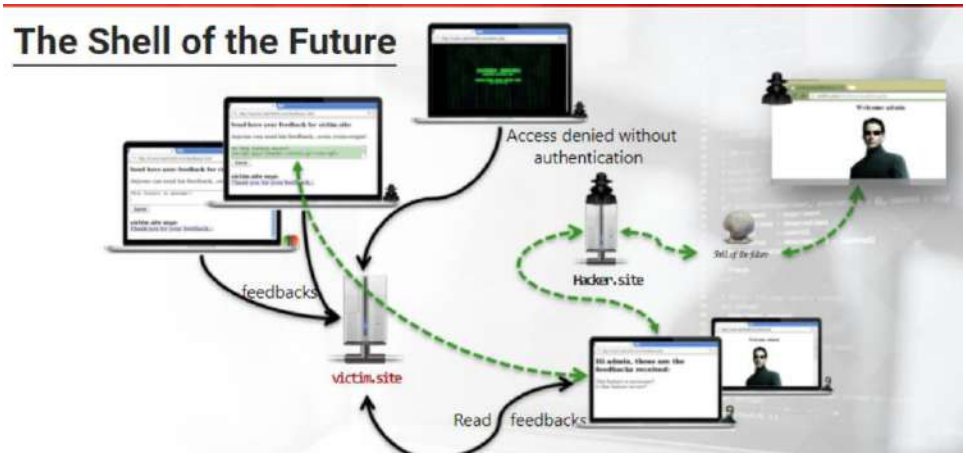
The Shell of the future

- http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html

The Shell of the Future

Shell of the Future is a Reverse Web Shell handler. It can be used to hijack sessions where JavaScript can be injected using Cross-site Scripting or through the browser's address bar. It makes use of HTML5's Cross Origin Requests and can bypass anti-session hijacking measures like Http-Only cookies and IP address-Session ID binding...

- Impersonate administrator by exploiting an XSS flaw via COR
  - attacker doesn't have to steal his session cookie



▼ Storage Attack Scenarios

Web Storage

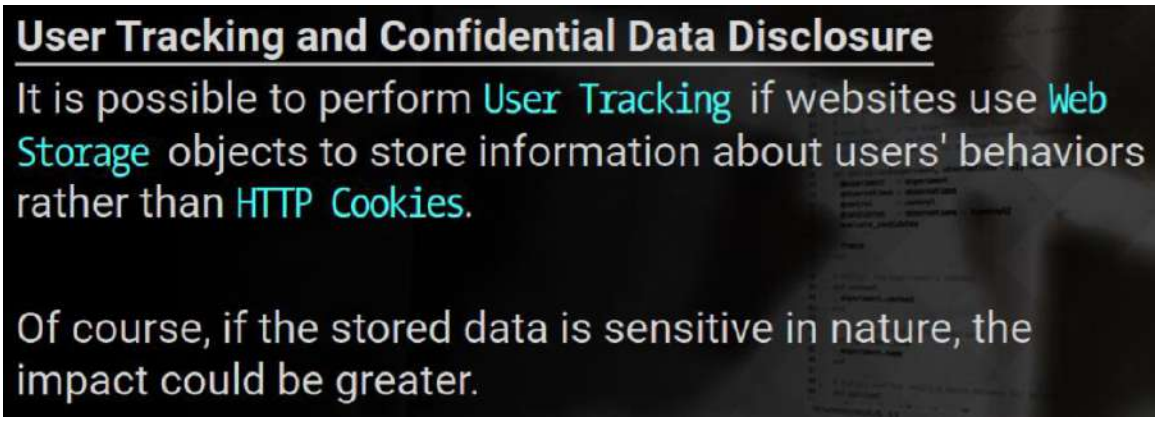
- Web Storage
- IndexedDB

Session Hijacking

- there's no ***HTTPOnly*** in **Web Storage Technologies**

Cross Directory Attacks

Using Tracking and Confidential Data Disclosure



IndexedDB

- **IndexedDB** is an alternative to **Web SQL Database**
- primary risks are related to **information leakage** and **information spoofing**
- it follows **SOP** but litmits the use to HTTP and HTTPS in all browsers except Internet Explorer
- it allows **ms-wwa** and **ms-wwa-web**

▼ Web Messaging Attack Scenarios

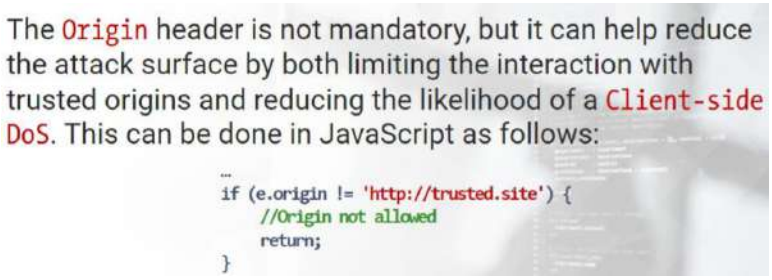
- web messaging also refered as **Cross Document Messaging** → api name: **postMessage**
- with using this, it means that the hosting web page can receive content from other domains without the server being involved, thus bypassing possible server-side security checks.

DOM XSS

- this occurs if the `postMessage` data received is used without validation.
  - eg: using **DOM sinks** such as **innerHTML**, **write**

Origin Issue

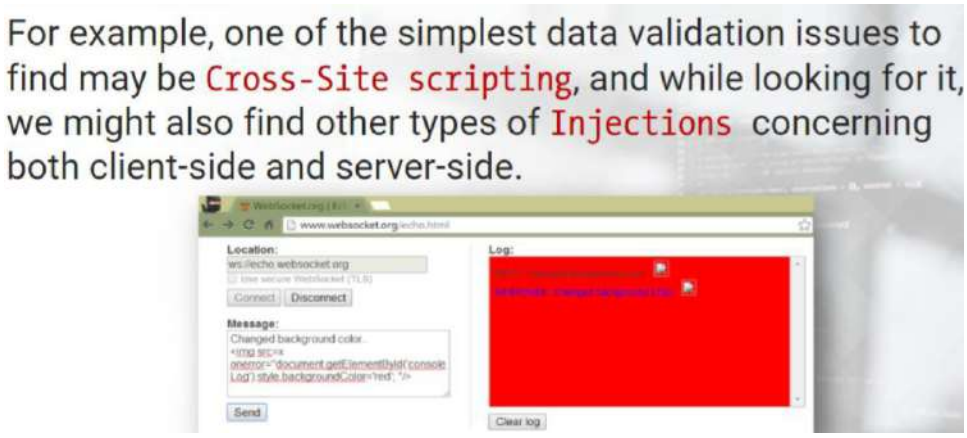
- the **Web Messaging Protocol** allows the presence of the Origin header field
- should be used against cross origin use of a websocket server by scripts that are using the websocket API in a web browser



▼ Web Sockets Attack Scenarios

- the websocket api
- if we are able to execute javascript code inside the victim browser, its possible to start a web socket connection and perform our malicious operations

Data Validation



MiTM

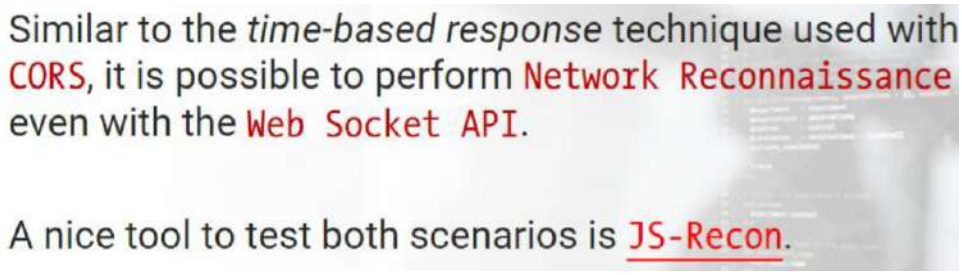
- webSocket Protocol
  - ws → unencrypted

- wss → encrypted

- if the implementation uses the unencyrpted channel, we have a MiTM issue whereby, anybody on the network can see and manipulate the traffic

Remote Shell

Network Reconnaissance



▼ Web Workers Attack Scenarios

- Dedicated Web Workers
  - can only be accessed through the script that created it
- Shared Web Workers

- can be accessed by any script from the same domain

From a security point of view, `Web Workers` did not introduce new threats. However, it provided a new way that was both easier and, at times, quicker for common attack vectors to be exploited. These can also use other HTML5 technologies such as `Web Sockets` or `CORS` in order to increase the performance and feasibility of the attack.

Browser-Based Botnet

To setup a `Browser-Based Botnet` and run an attack, there are two main stages to consider. The **first** is to `Infect` the victims. There are several ways to achieve this, such as: `XSS`, `email spam`, `social engineering`...

Once the victims have been hooked, the **second** stage is to `Manage Persistence`. This is an important stage because of the nature of the botnet. In fact, the malicious code will work until the victim browser (window|tab) is closed, then it stops running.

The real challenge is to both keep alive the connection with the bot and to also be able to **re-infect** the system if required. For this purpose, attack vectors, like `Persistent XSS`, are the best.

Sometimes, implementing a game can help you keep the victim on the malicious page. If the game is both interactive and especially addictive, the user may remain online the entire day. Often, these types of users do this to keep their score active and avoid restarting the game.

With an HTML5 Botnet, **some** of the possible attacks that can be performed are:

Distributed Password Cracking

One of the uses of WebWorkers is using a distributed computing system to perform brute force attacks on password hashes. This kind of implementation is obviously slower in respect to custom solutions. They are around 100 times slower; however, we should not forget that in a botnet context this gap can be significantly reduced!

With this considerations in mind, Lavakumar Kuppan has developed Ravan. A system based on WebWorkers to perform password cracking of MD5, SHA1, SHA256, SHA512 hashes.

If we add CORS to the efficiency of WebWorkers, then we could easily generate a large number of GET/POST requests to a remote website. We would be using COR requests to perform a DDoS attack.

Basically, this is possible because, in this type of attack, we do not care if the response is blocked or wrong. All we care about is sending as many requests as possible!

WebWorkers + CORS - DDoS Attacks

If we add CORS to the efficiency of WebWorkers, then we could easily generate a large number of GET/POST requests to a remote website. We would be using COR requests to perform a DDoS attack.

Basically, this is possible because, in this type of attack, we do not care if the response is blocked or wrong. All we care about is sending as many requests as possible!

Again, this was also possible before; however, the key here is the performance that can be gained using multiple browsers (bots) and CORS. Clearly, there are also some technical limitations with CORS.

With CORS, if in the response to the first request is either missing the Access-Control-Allow-Origin header or the value is inappropriate, then the browser will refuse to send more requests to the same URL.

To bypass this limitation, we create dummy requests and add fake parameters in the query-string. In doing so, we force the browser to transform each request, therefore identifying it as **unique**. This obviously makes the browser treat it as a new request each time. For example:

```
http://victim.site/dossable.php?search=x
```

Use random values here

▼ HTML Security Measures

- HTTP Headers ^_^

▼ Security Headers: X-XSS-Protection

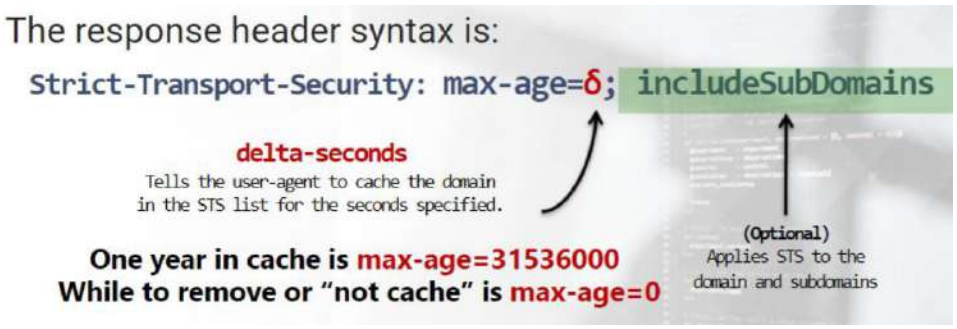
- to protect ***user agents*** against a subset of **reflected xss** attacks
- supported by chrome, IE, webkit based browsers (Safari, Opera, etc..)
- its not standardized,, not supported by browsers like ***Firefox***
  - they seems to by okay with ***Content-Security-Policy*** and ***NoScript***

▼ Security Headers: X-Frame-Options

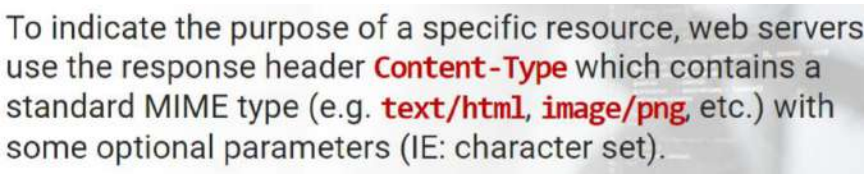
- in order to prevent ***clickJacking***, *the* ***X-Frame-Options*** response header was introduced
- the server instructs the browser on whether to display the transmitted content in frames of other web pages.
- Syntax: **X-Frame-Options: *value***
  - **DENY** → page cannot be displayed in a frame regardless of the site attempting to do so
  - **SAMEORIGIN** → page can only be displayed in a frame on the same origin as the page inself
  - **ALLOW-FROM URI** → can only be displayed in a grame on the specified origin

▼ Security Headers: Strict-Transport-Security

- **HTTP Strict Transport Security → HSTS**
- Allows secure connections only (***HTTPS***)
- Not supported by all vendors
- Syntax: Strict-Transport-Security: max-age=...



▼ Security Headers: X-Content-Type-Options



- Content-sniffing (MIME type sniffing)
  - While content sniffing can be useful in certain situations, it has also been a source of security concerns. For instance, it can be exploited in attacks such as MIME **sniffing-based XSS (Cross-Site Scripting) attacks**, where malicious content is disguised as a different MIME type to bypass security controls.
- To mitigate these risks, there have been efforts to standardize and improve the handling of MIME types, and some security headers, like `X-Content-Type-Options`, can be set by servers to instruct browsers not to perform content sniffing. This header is designed to reduce the risk of certain types of web vulnerabilities associated with MIME type confusion.
- the **X-Content-Type-Options** header instructs the browser to `not guess` the content type of the resource and trust of the **Content-Type** header
- Syntax: **X-Content-Type-Options: nosniff**
- only works in ***IE*** and ***Chrome***

▼ Security Headers: Content Security Policy

- **CSP** → **Content Security Policy**
- reduce the risk of broad class of **content injection** vulnerabilities

- It’s not a **first line defense** mechanism against content injection vulns;
  - it’s a **defense-in-depth** solution
- **X-Content-Security-Policy** and **X-WebKit-CSP** (deprecated, was used by webkit based browsers eg:safari)
  - Now its named **Content-Security-Policy**
- The CSP uses a collection of directives

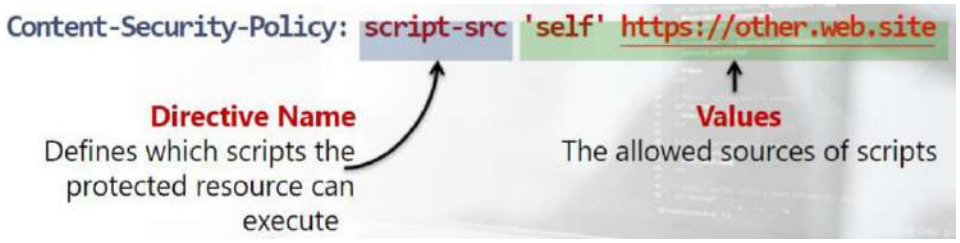
▼ Common Directives

Each directive in the CSP header is associated with a specific type of resource or behavior. Here are some common CSP directives:

1. `default-src`: Specifies the default source for content that doesn't have a specific directive. For example, `default-src 'self';` allows content to be loaded only from the same origin.
2. `script-src`: Defines the allowed sources for JavaScript. For example, `script-src 'self' example.com;` allows scripts from the same origin and from `example.com`.
3. `style-src`: Specifies the allowed sources for stylesheets. For example, `style-src 'self' 'unsafe-inline';` allows styles from the same origin and inline styles.
4. `img-src`: Defines the allowed sources for images. For example, `img-src data: https:;` allows images from `data:` URLs and secure (`https:`) sources.
5. `font-src`: Specifies the allowed sources for fonts.
6. `connect-src`: Defines the allowed sources for network requests such as XHR, Fetch, and WebSockets.
7. `frame-src`: Specifies the allowed sources for embedding content in frames.
8. `media-src`: Defines the allowed sources for media elements such as `<audio>` and `<video>`.
9. `object-src`: Specifies the allowed sources for plugins using `<object>`, `<embed>`, or `<applet>`.

10. `child-src`: Defines the allowed sources for worker, embed, or inline frame content loaded using `<frame>`, `<iframe>`, `<object>`, `<embed>`, or `<applet>`.
11. report-uri: reporting feature
12. sandbox: optional

- it instructs the browser to only execute or render resources from the allowed sources
- Most Important and commonly used directives is **script-src**



Directives work in **default-allow** mode. This simply means that if a specific directive does not have a policy defined, then it is equal to *; thus, every source is a valid source.

For example, if the CSS and style markup directive `style-src` is not set, the browser can then load stylesheets from anywhere without restriction.

To both avoid this type of behavior and define a common rule for all the directives unset, the specification provides the `default-src` directive. Clearly, it will be applied to all the unspecified directives.

For example:

```
Content-Security-Policy: default-src 'self'
```

With CSP, it is also possible to deny resources. For example, if the web application does not need plugins or to be framed, then the policy can be enriched as follow:

```
Content-Security-Policy: default-src https://my.web.site;
object-src 'none'; frame-src 'none'
```

None returns an empty set for the allowed sources.

In addition to the source list values, there are four keywords that are also allowed. It is important to realize that they must be used with single-quotes, otherwise they refer to server named *none*, *self*, etc...

- `'none'` - no sources
- `'self'` - current origin, but not its subdomains
- `'unsafe-inline'` - allows inline JavaScript and CSS
- `'unsafe-eval'` - allows text-to-JavaScript sinks like `eval`, `alert`, `setTimeout`, ...

An interesting mechanism provided by CSP is the option to report violations to a specified location.

```
Content-Security-Policy: default-src 'self'; report-uri /csp_report;
```

Once a violation is detected, the browser will perform a POST request to the path specified, sending a JSON object, similar to the one on the next slide.

CSP Violation Report

```
{
  "csp-report": {
     "document-uri": "http://my.web.site/page.html",
     "referrer": "http://hacker.site/",
     "blocked-uri": "http://hacker.site/xss_test.js",
     "violated-directive": "script-src 'self',
     "original-policy": "script-src 'self'; report-uri
http://my.web.site/csp_report"
  }
}
```

If you want to familiarize yourself with CSP, check out this web site: http://www.cspplayground.com/. It contains some useful examples of CSP Violations and related fixed versions.

- http://www.cspplayground.com

▼ UI Redressing: The x-Jacking Art

- Common examples of UI Redressing attacks include overlaying a visible button with an invisible one
- UI redressing techniques are also referred to as **ClickJacking**, **LikeJacking**, **StrokeJacking**, **FileJacking**

▼ ClickJacking

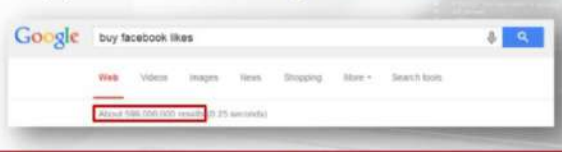
A Basic ClickJacking attack uses a nested iframe from a target website and a little bit of CSS magic. This "magic" often leverages position, opacity and many other CSS attributes.

A nice example can be a simple game where the victim needs to find and click on a character. As shown in the next slide, in the background there is a submission button or whatever the attack chooses to trigger an action.

In this example, when the victim thinks to click on the Willie character, he is actually clicking on the red button that submits the vote against the website survey.

▼ LikeJacking

With the widespread use of Facebook, the amount of "Likes" is often a measure of perceived popularity for a particular product (event, brand, etc.). The more the "likes", the more influential it can be. That is why many organizations who specialize in selling Facebook likes and friends, have appeared over the years.



▼ StrokeJacking

StrokeJacking , a technique to hijack keystrokes, is proof that UI Redressing is not only all about hijacking clicks.

This method can be very useful in the exploiting a code injection flaw. Generally, this is XSS and the payload must be manually typed.

6.4.3.1 StrokeJacking Example – Show Me the Hidden Picture

This type of scenario was discovered by Lavakumar Kuppan while testing.

He took that opportunity to make a **blog post** about it.

▼ New Attack Vectors in HTML5

▼ Drag-and-Drop

The specification allows us to **Drag** data across different origins; therefore, there is no need to care about the SOP because it does not apply to the `DnD API`! Of course, this is pointless to say, but this API introduced several new attacks.

The most interesting "interpretation" of this API has been presented by **Paul Stone**. He has proven powerful attack vectors that mix `Drag-and-Drop` with `ClickJacking` techniques.

Text Field Injection

The first possible technique allows the attacker-controller to drag data into hidden text fields/forms on different origins.

Despite a simple click or typing characters, hijacking drag gestures is much more complex.

We need to be a little more creative and maybe create something like a pilot Drag and Drop game, where the victim will move our malicious pieces of code into the target locations.

The next slide is a single example; however, the possible scenarios are immeasurable, which can be anywhere from ball games to puzzles.

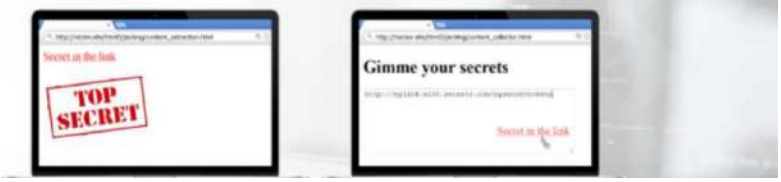
Text Field Injection

Text Field Injection

To explain this kind of exploitation, Krzysztof Kotowicz has developed a vulnerable scenario and a simple game to exploit the vulnerability named **Alphabet Hero**.

Content Extraction

If the **secret** is part of a URL, in an HTML `anchor` element or an `image`, dragging is quite easy. In fact, when the elements are dropped onto a target location, they will be converted into a serialized URL.



Content Extraction

The difficult part is when the secret is not easily draggable because it is part of a textual content of the page. Then, we need to trick the victim into first selecting the section we want and then later dropping the selection on our text area.

Content Extraction

Sometimes, information in clear text is not enough and we need to go deeper into the page.

For example, let's think about anti-CSRF tokens hidden in forms, or whatever is only visible by inspecting the source code.

We could use the `view-source:` pseudo-protocol to load the HTML source code into an iframe. So instead of using:

```
<iframe src="http://victim.site/secretInfoHere/"></iframe>
```

We change the `src` attribute to:

```
<iframe src="view-source:http:// victim.site/secretInfoHere/"></iframe>
```