6. Module - HTML5 - part 1

-> Helpful event to trigger XSS in HTML5
onhashchange, onformchange, onscroll, onresize ...

```
<body onhashchange="alert(1)">
<a href="#">Click me!</a>
```

Elements

-> element that supports autofocus are useful to trigger a XSS
`<keygen autofocus onfocus="alert(1);">`
-> elements that supports src attribute are usefull to trigger a XSS
`<video>` `<audio>` `<track>` `<source>`
`<embed src="http://hacker.site/evil.swf">`
`<embed src="javascript:alert(1)">`

Web Application offline !!

website can be store on our machine and visit them offline by using feature like Application Cache and Web Storage (alias Client-side storage or Offline storage).

The attack scenarios may vary from **Session Hijacking**, **User Tracking**, **Disclosure of Confidential Data**, as well as a new way to **store attack vectors**.

Session Hijacking

to steal the stored session in the browser storage

security measure implementation like HTTOnly does not help here, so exploiting session hijacking much easier.

With offline web applications, the most critical security
issue is **Cache Poisoning**.

HTML5 features:
-> Geolocation API
-> Fullscreen API (mages, videos, etc to be viewed fullscreen)

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.

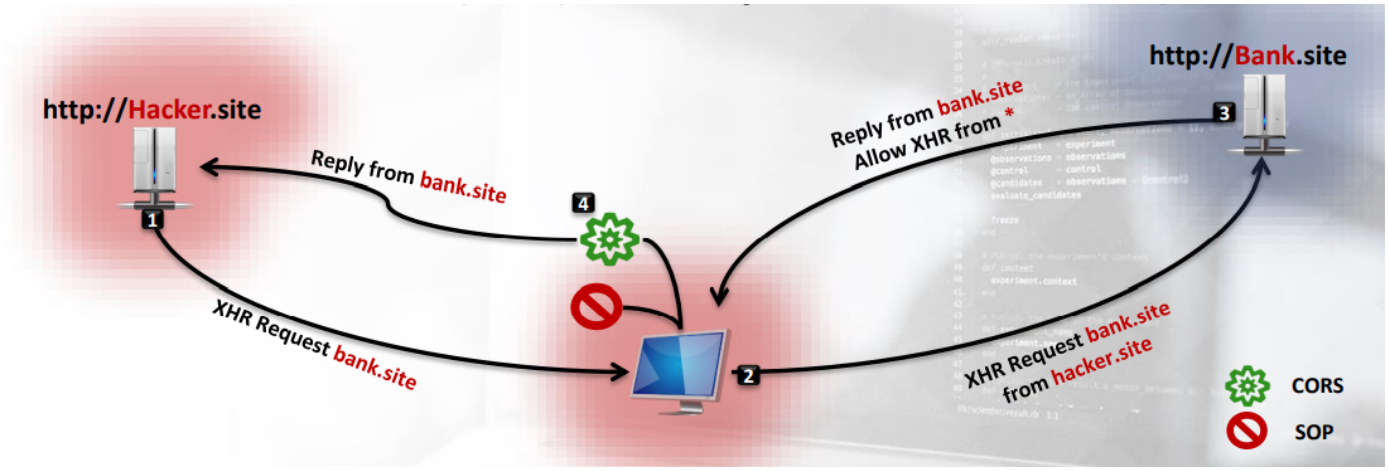
Security View: the most important features introduced are: CSP, **CORS**, **CDM** and the ifram attribute **sandbox**

Exploiting HTML5

websites become increasingly more dynamic, so the **same origin policy** (SOP) become more restrictive and **Cross-origin Resource sharing** (CORS) has been instroduced.

CORS

These allow both server and browser to communicate in relation to which requests are or not allowed.



Access-Control-Allow-Origin

It is a CORS response header and it indicates whether a resource can be shared or not. based on the returning value of the Origin request header (the star * , null or origin-list) in the response

```
Access-Control-Allow-Origin = "Access-Control-Allow-Origin" ":" origin-list-or-null | "*"
```

some website sue the wildcard value * to allow any website to make CORS query to a page. this is one of the most common misconfigurations with CORS headers. because of the laziness of the implementer.

How to abuse this ?

Examples :
-> if XSS found on the vulnerable page to CORS, it can be use to infect or impersonate the user like

making request from user browser automatically like CSRF

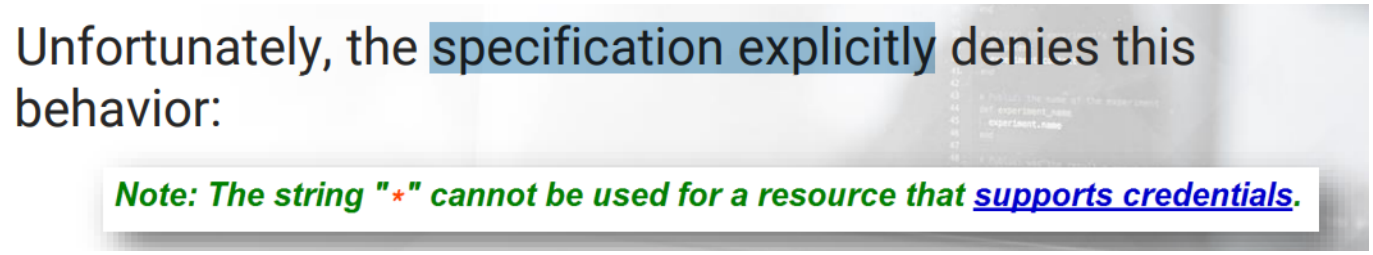
-> It can be used to bypass intranet websites.
example, tricking internal users to access a controlled website and making a COR query to their internal resources in order to read the responses.
use the users as a proxy to exploit
vulnerabilities

-> use the users as a proxy to exploit vulnerabilities

Access-Control-Allow-Credentials

reference: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Credentials

-> indicates that the request can include user credentials. (Credentials are cookies, authorization headers, or TLS client certificates)
-> the only value for this CORS header is true



so developers do the implementation on server-side

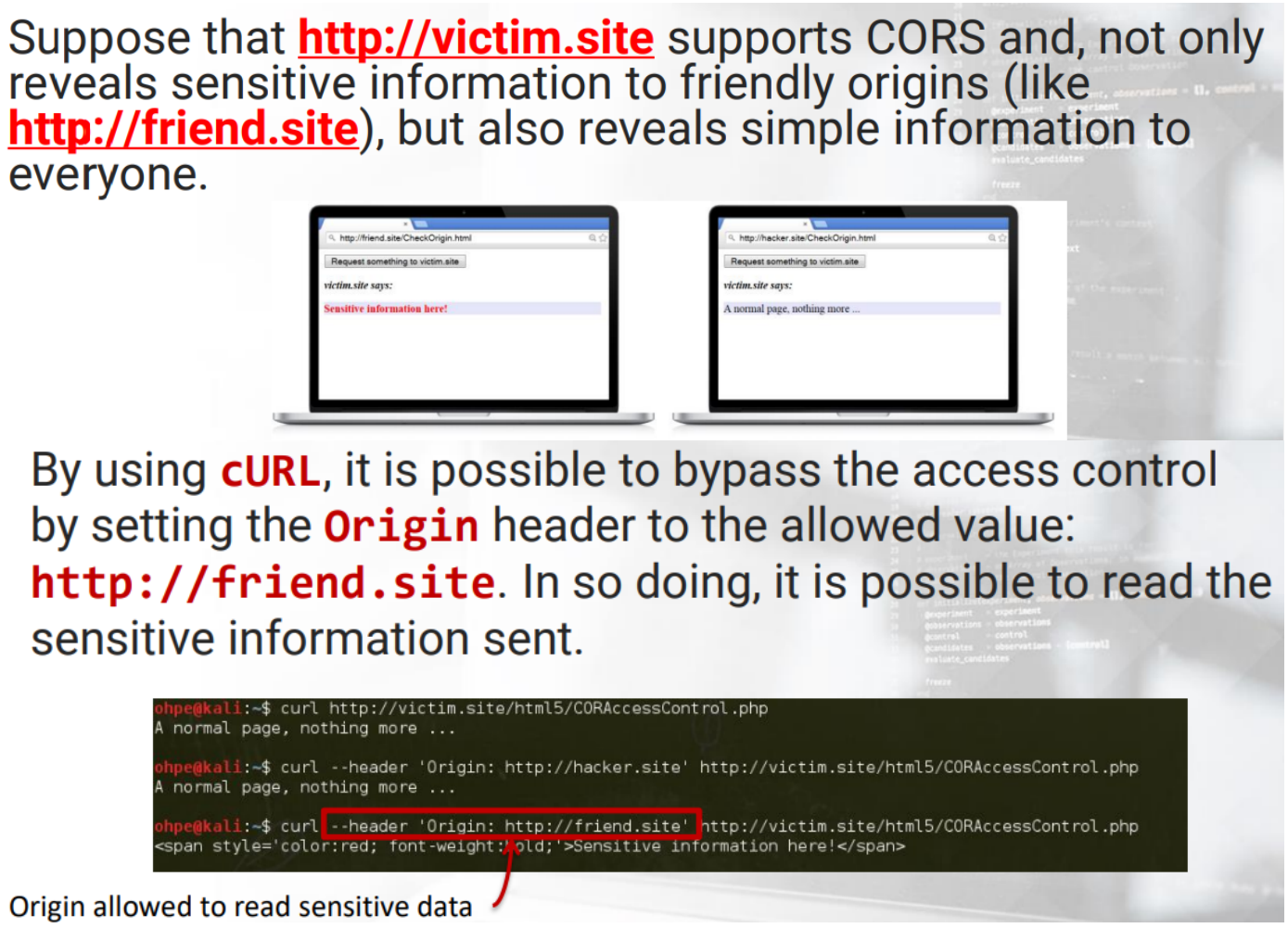
```
<?php
header('Access-Control-Allow-Origin: ' + $_SERVER['HTTP_ORIGIN']);
header('Access-Control-Allow-Credentials: true');
```

above code to make Access-Control-Allow-Credentials: *
with this a CSRF is exploitable, any origin can read the anti-CSRF token from the page

to make it trusting only certain origin, we use the CORS header named **Origin**.

the **Origin** header cannot be spoofed from the browser, the header can be spoofed by creating requests outside of the browsers (using proxy or tool like curl, wget, etc...)

EXAMPLE:



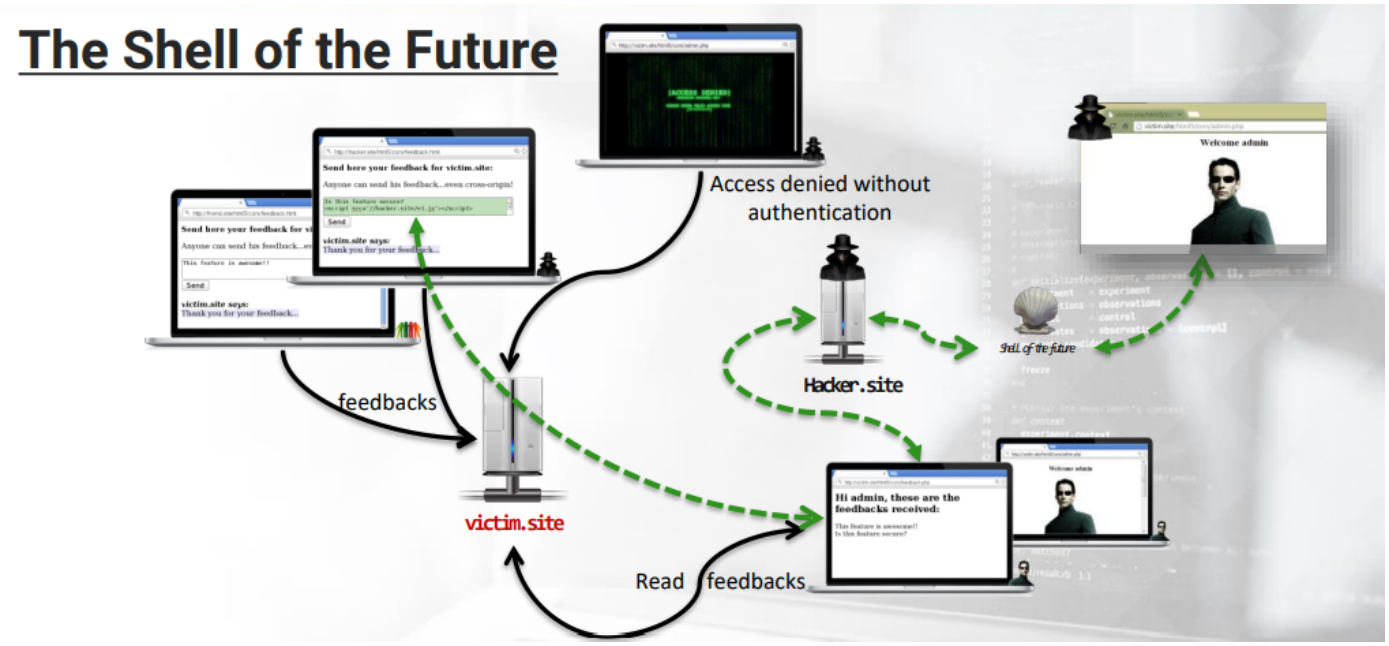
Intranet Scanning

Abusing COR to do time based Intranet Scanning by sending XHR to domains and based on the reponse time, we know port open or close.

tools:
-> **jsrecon**: this is an HTML5 based JS, which leverages features like **COR** and **Web Sockets** to do scanning from the browser
https://web.archive.org/web/20120313125925/http:/www.andlabs.org/tools/jsrecon/jsrecon.html

-> **Remote Web Shells**: If we have XSS and target web supports CORS, an attacker can hijack the victim session, establishing a communication channel between the victim's browser and the attacker. The attacker can do anything the victim can do in that web application context
the Shell of the future

https://web.archive.org/web/20150223205517/http:/www.andlabs.org/tools/sotf/sotf.html

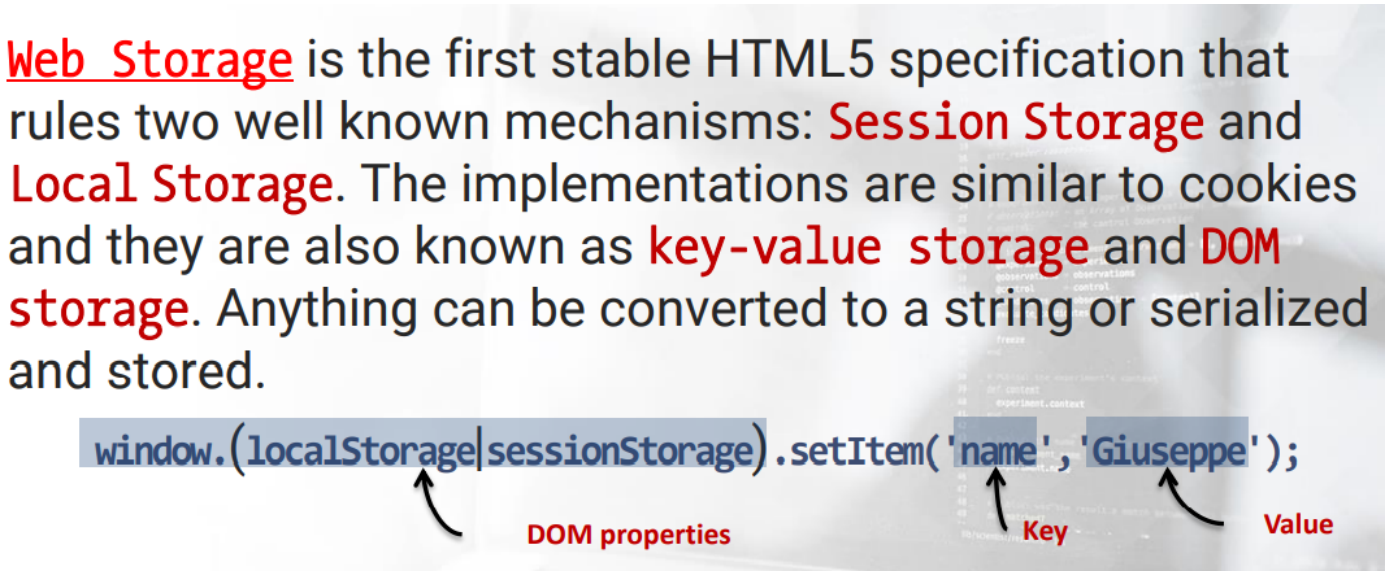


-> cookie limitation (for knowledge):

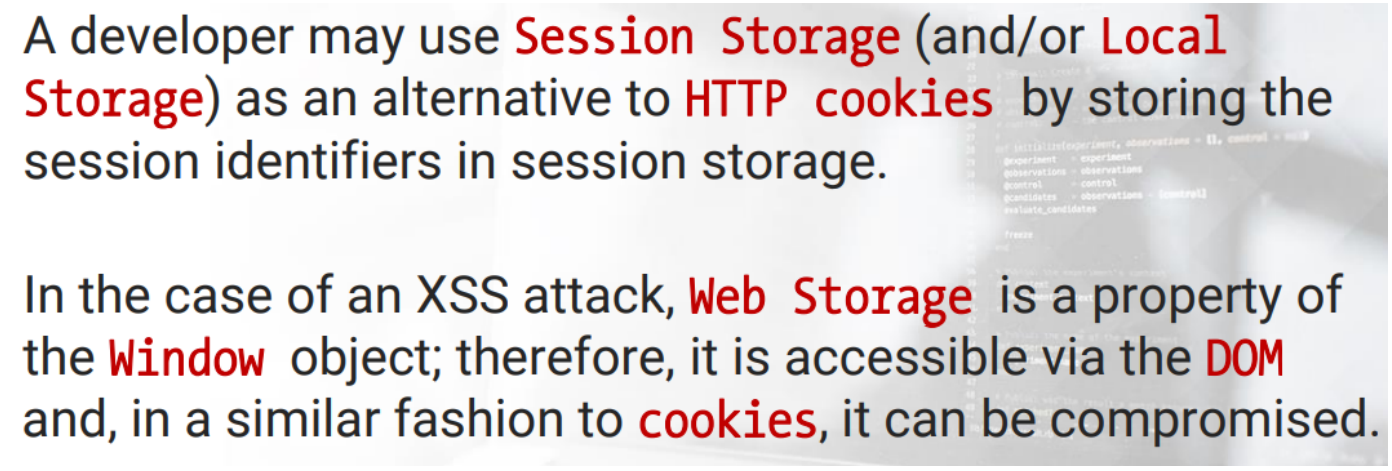
- 4096 bytes per cookie
- 50 cookies per domain
- 3000 cookies total

attack scenarios of Web Storage and IndexedDB technologies.

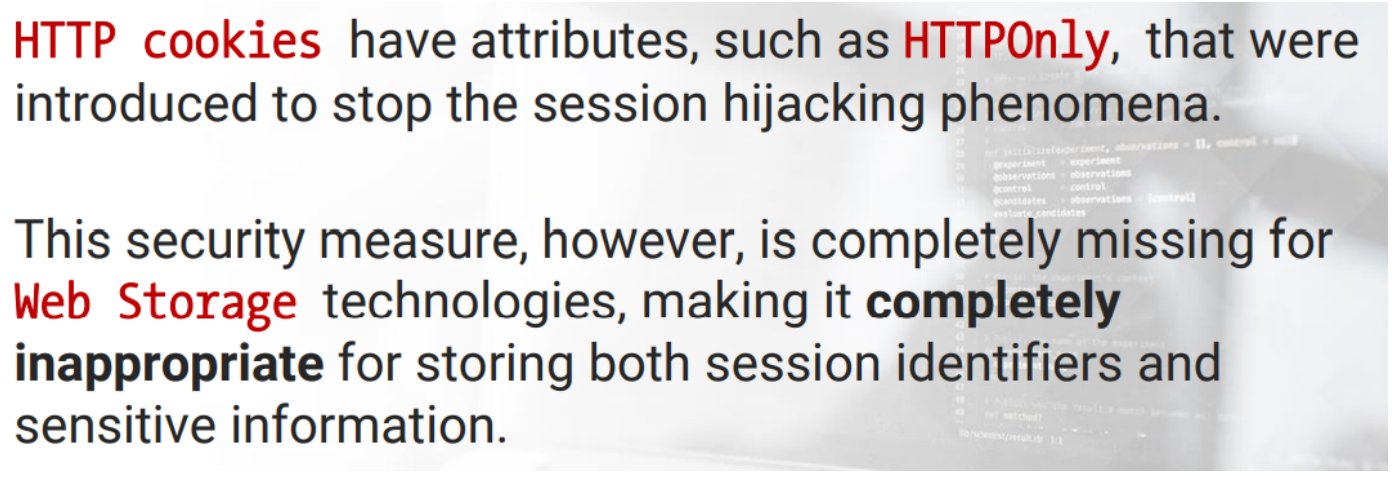
web storage



Session Hijacking



exploitation same as in cookie but in the api



Cross-directory Attacks

In **Web Storage** there is no feature that restrict the access to the pathname like http cookie `cookie:asd; path:"\users"`. This makes the Web Storage content available in the whole origin. which lead to **Cross-directory attacks**
if we have XSS in a specific path `example.com/pathname`, we can read all stored data in all the directories available in the domain

IndexedDB

IndexedDB vs. **WebSQL Database**:
IndexedDB is an alternative to **WebSQL** Database. IndexedDB and WebSQL are solutions for storage.

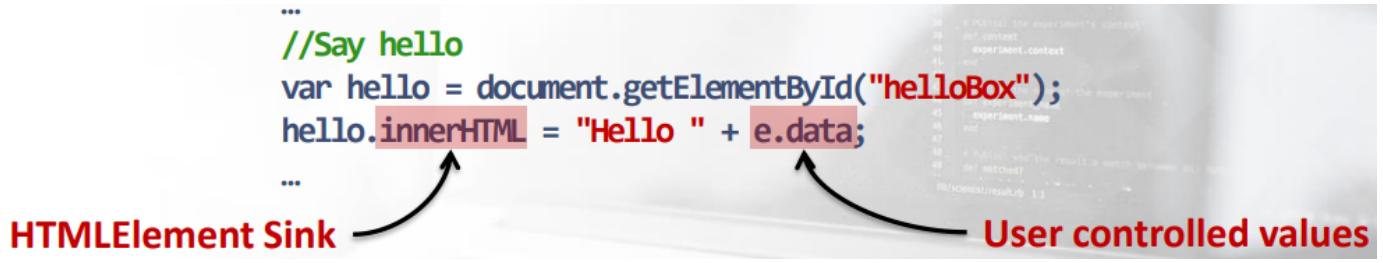
WebSQL Database is a relational database access system, whereas **IndexedDB** is an indexed table system.

Indexed Database API is an HTML5 API for high performance searches on client-side storage.
Note: data search in storage with indexes known as keys.

Attacks here are similar to Web storage attacks, Primary risks are **information leakage** and **information spoofing**.

Web Messaging Attack Scenarios

Web Messaging referred as Cross Document Messaging (CDM) or postMessage.
-> this makes the communication between the iframe and the host possible. the domain can recieve content from other domains.
Vulnerability will be most likely DOM XSS



some protection like checking from where the request is coming from

However, this can be bypassed via spoofing the Origin header outside the browser via proxy or via tools like curl, etc..

WebSocket Protocol

It defines 2 schemes for web socket connections:
`ws` for unencrypted
`wss` for encrypted
so if the implementation uses the unecrypted channel, we can have `MiTM`

If we can run JS code on the victim (xss in a website / trick user to visit my website), we can have a Remote Shell active until the window tab is closed.

Web Workers : feature allows most modern browsers to run JavaScript in the background.

2 Type of workers:

- **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

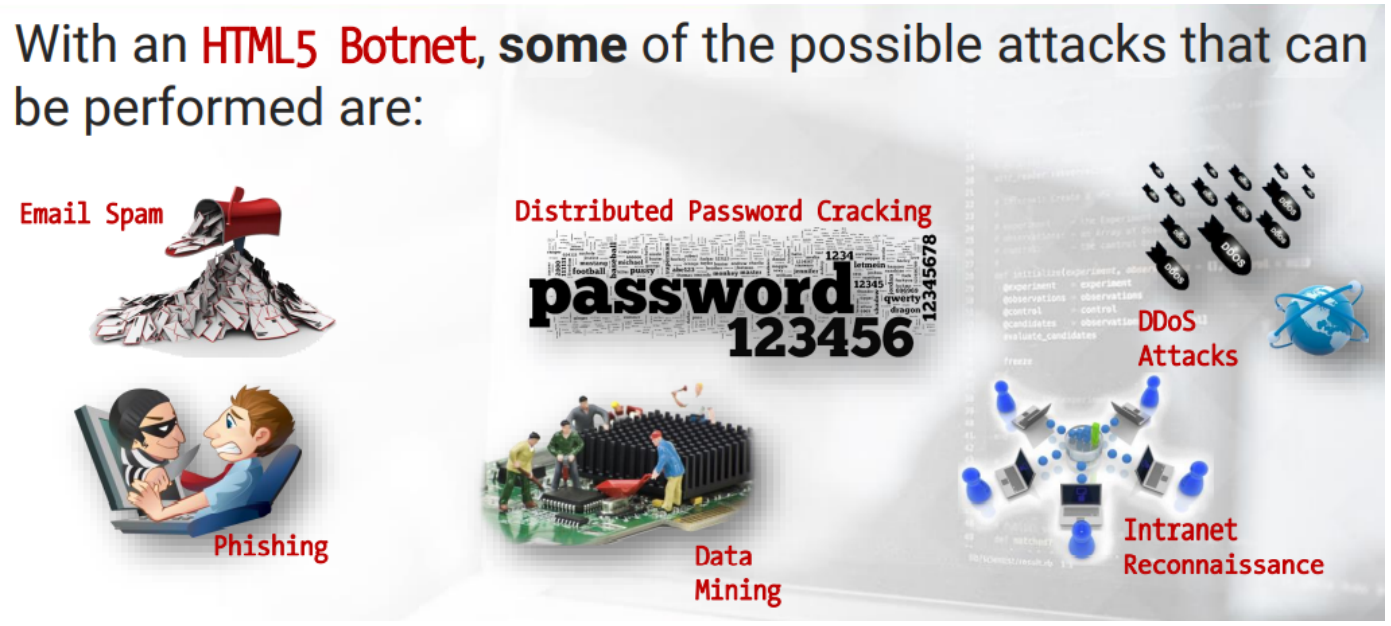
Browser-Based Botnet

-> running a botnet in a browser is limitless. so we can run JS code on any Browser that supports the feature **the bot uses**.

To setup a **Browser-Based Botnet** and run an attack, 2 things to do:

- first infect the victim (via XSS, Spam, etc.)
- Manage Persistence, because the malicious code will work until the tab is closed.

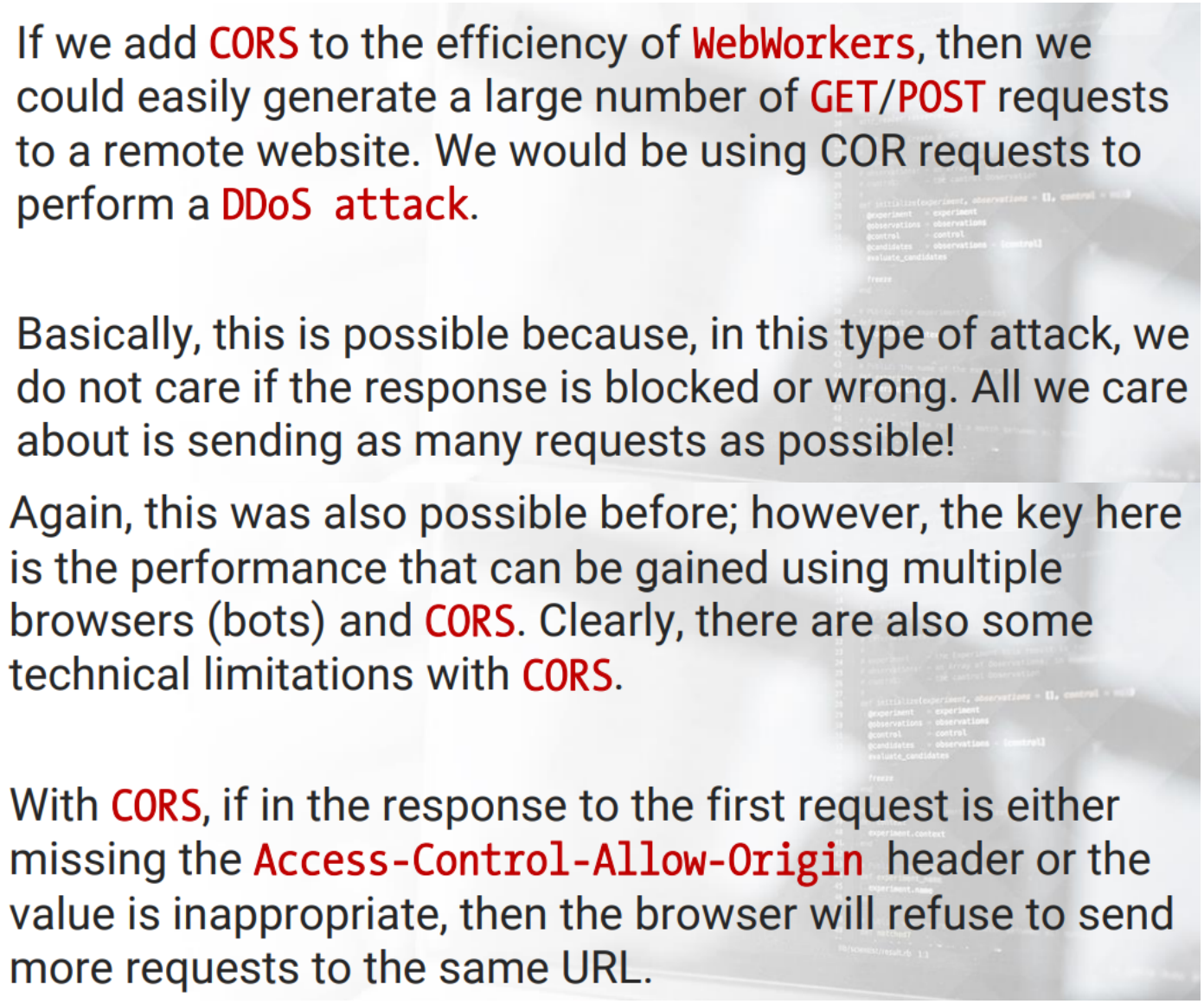
Example: implementing a game can help you keep the victim on the malicious page. Game with score or any wins



Distributed Password Cracking

Webworkers is also used to perform bruteforcing using a distributed computing system. This kind of implementation is around 100 times slower in respect to custom solutions. but in the botnet content, this can be reduced.
http://web.archive.org/web/20160315031218/http:/www.andlabs.org/tools/ravan.html

WebWorkers + CORS – DDoS Attacks



there is some bypassing rate limiting technique, like making request with a random parameter and changing the value in each request

HTML5 Security Measures

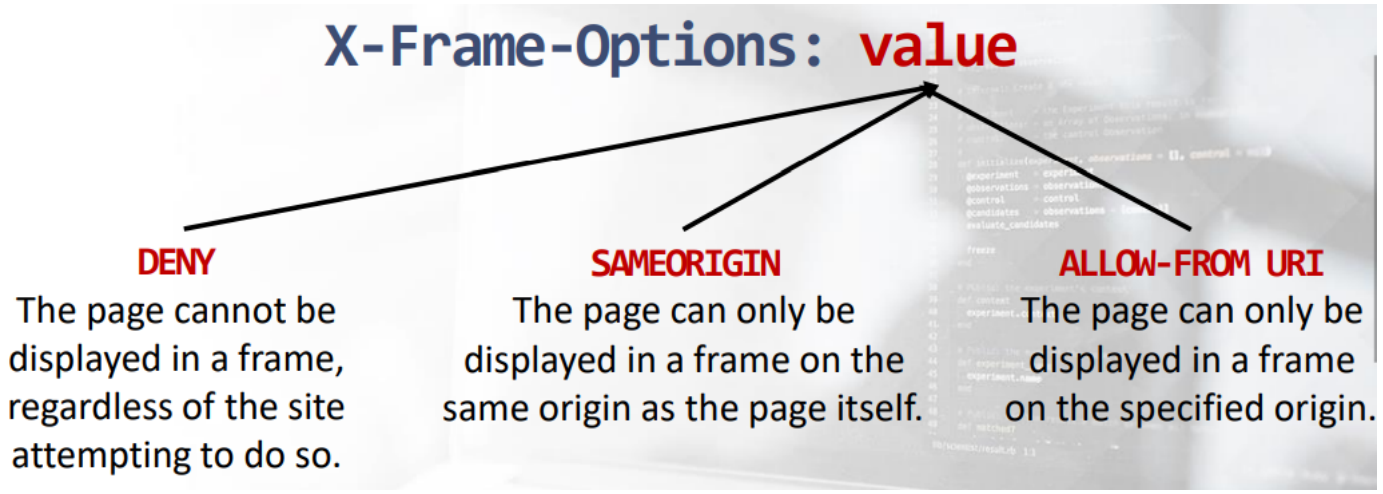
Note: New feature lead to new security risks
so security enhancements, except for iframe sandboxing, are HTTP headers in HTML5

X-XSS-Protection

-> to protect user agents against a subset of Reflected Cross-site Scripting attacks.
-> Not standardized, firefox does not support it, but we can use (CSP + NoScript)

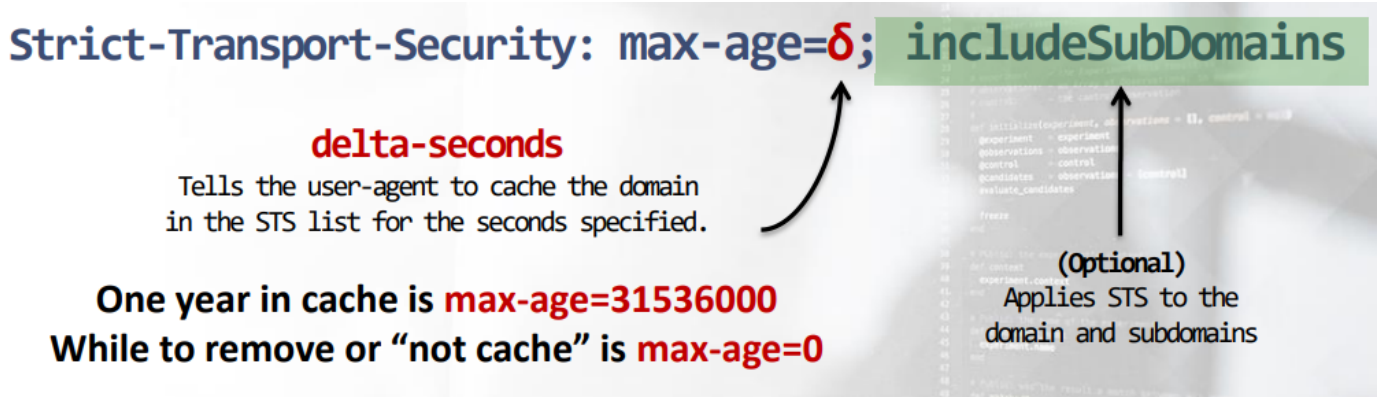
X-Frame-Options

-> to prevent **ClickJacking**
-> the server to instruct the browser on whether to display the transmitted content in frames of other web pages.
-> The syntax is simple:



Strict-Transport-Security (HSTS)

-> Force the website to be accessible only via https
-> The response header syntax is:



X-Content-Type-Options

This header instructs the browser to "not guess" the content type of the resource and trust of the Content-Type header.

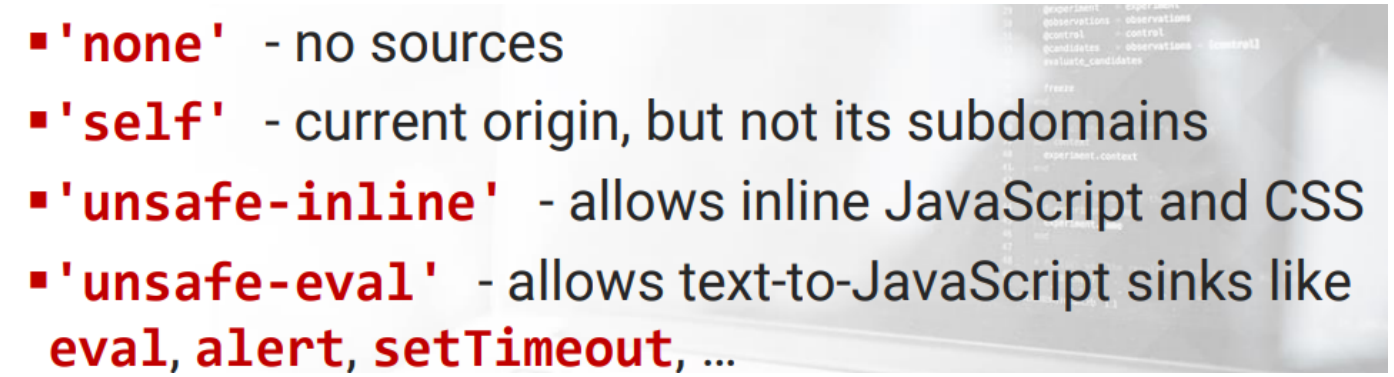
```
X-Content-Type-Options: nosniff
```

-> works on IE and Chrome.

Content Security Policy

It instructs the browser to only execute or render resources from the allowed sources.
-> see my notes about the security headers

-> the keywords below must be used with single-quotes:



-> CSP provide also an option to report violations to a specfied location:

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

So, 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-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"
    }
}
```

MIME sniffing:
https://blog.fox-it.com/2012/05/08/mime-sniffing-feature-or-vulnerability/

UI Redressing: The x-Jacking Art

-> UI redressing techniques are also referred to as ClickJacking, LikeJacking, StrokeJacking, FileJacking and many others.
-> examples of UI Redressing attacks include overlaying a visible button with an invisible one

Basic ClickJacking

A nested iframe from a target website and a little bit of CSS magic (position, opacity and other css attributes).

LikeJacking

https://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/

StrokeJacking

http://blog.andlabs.org/2010/04/stroke-triggered-xss-and-strokejacking_06.html

New Attack Vectors in HTML5

Drag-and-Drop

https://web.archive.org/web/20160413235555/http:/www.contextis.com/documents/5/Context-Clickjacking_white_paper.pdf

Text Field Injection

Content Extraction

Ebrahim Hegazy

SOP and CORS. The idea is that request can by made from other origins

1. attacker site evil.com & target site example.com
2. Attacker try to make request from evil.com to target.com to fetch the cookie or any other infos.
3. attacker put a malicious code in the evil.com like this

```
<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function(){
if(this.readyState == 4 && this.state == 200){
alert(this.responseText);
```

```
}};
xhttp.open("GET",url,true);
xhttp.withCredentials = true;
xhttp.send();
</script>
```

4. once the victim visit evil.com, a request will be made to target.com and cookie/data will be sent to the attacker

SOP will not allow origin to access anything from the origin, but this is not very good, because sometimes we have to give access to some unknown websites so we use with SOP the CORS, which limit the access, specially accessing credentials (xhttp.withCredentials = true | false)

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

this is a misconfiguration because we give access with credentials. so best pratice is to disable the credentials (Access-Control-Allow-Credentials: false)

Content-Security-Policy header specify what origin are allowed to run scripts on the website

How to detect?

1. way: look in the response header for this

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

2. way: set Origin:test.target.com as a header in the request and see if it is get reflected.