Project Report

CS 649 : Network Security

Cross Site Scripting Prevention

Under Guidance of

Prof. Bernard Menezes

Submitted By

Neelamadhav (09305045)
Raju Chinthala (09305056)
Kiran Akipogu (09305074)
Vijaya Kumar (09305081)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Contents:

Abstract :

Web applications have become the dominant method for implementing and providing access to on-line services. As the use of web applications increases, a greater focus on web security is needed. In recent years, there has been a significant increase in the number of web-based attacks. Cross-site scripting (mostly called as XSS) is one of the most dangerous and most common website vulnerability on the internet. The attack is based on the possibility to insert malicious JavaScript code into web pages . Since Web browsers support the execution of commands embedded in web pages to enable dynamic web pages attackers can make use of this feature to enforce the execution of malicious code in a user's Web browser.

As part of the project we have studied two types (stored and reflected) of XSS attacks and demonstrated those attacks on two websites. We have implemented XSS prevention technique called Blueprint which is proposed by : V.N. Venkatakrishnan and Mike Ter Louw , University of Illinois at Chicago (Appeared in 30th IEEE Symposium on Security and Privacy, Oakland, May 2009.).

Keywords:

XSS, Content Model, Model Interpreter

Introduction :

Cross-site scripting is a type of computer security vulnerability that is found in web-based applications which allows code injection by web users into any webpage that is viewed by other users. XSS is made possible due to the fact that faulty coding causes XSS holes (vulnerabilities on websites that allows attackers to avoid security measures) in the client-side script that allows for insertion of malicious code. During an attack, “everything looks fine” to the end user, but in actuality they are subject to a wide variety of threats. XSS is a potentially dangerous vulnerability that is easy to execute and very long and arduous to repair. XSS holes exist in 7 out of every 10 websites. Many site owners do not consider an XSS hole to be a big threat, which is a commonly made mistake because the consequences of an XSS attack against web applications and its users have been proven to be extremely serious. The most frequent kinds of web applications that are victimized by XSS attacks are search engines, discussion boards, web-based emails, and posts. Even the most well-known websites in today’s world like Google, Yahoo!, MySpace, Facebook, PayPal, and WikiPedia were once victims.

The most commonly used programming languages during XSS attacks are HTML, XHTML, JavaScript, and Adobe’s Flash. However the most popular and potentially the most detrimental language used by malicious attackers is JavaScript.

How Cross-Site Scripting works :

A website is vulnerable if it accepts and subsequently return the same input back to a user. The most common example is when a user does a search and the Web server returns the same data the user typed in. As an example, a user does a search for “XSS” and the browser returns a message of, “Your search for XSS returned the following.”

A cross-site scripting attack can be done rather easily to a Web server that is not properly protected. Web servers generate both text and HTML markup on their web pages. The client’s browser then interprets the web pages. HTML uses special characters to distinguish text from markup. Different characters are special at different points in the document, depending on the grammar. The less-than sign “<” usually indicates the beginning of an HTML tag. An HTML tag can affect the formatting of the page or introduce a program that will be executed by the browser. If the Web server creates pages by inserting dynamic data into a template, it should be checked to ensure that the data to be inserted does not contain any special characters. The user’s Web browser could mistake any special characters as HTML markup. This would result in the browser mistaking some data values as HTML tags or script instead of displaying them as text. An attacker can choose the data that the Web server inserts into the web page, thereby tricking the user’s browser into running a malicious script or program. The program will run in the browser’s security context. The attacker can use this to run the program in an inappropriate security context.

Types of XSS Attacks :

There are two significant types of XSS vulnerabilities that exist and they are

1.Non-persistent XSS

2.Persistent XSS

1. Non-persistent XSS : It is also referred as reflected XSS vulnerability. If a web user provides data to a server- side script to instantly generate a resulting page back to him/herself, a resulting page without html encoding can be intercepted by an invalidated user. The malicious client- side code can then be injected into the dynamic page. The attacker can apply a little social engineering (which is the power to manipulate someone to perform actions) to persuade a user to follow a malicious URL that will inject code into the resulting page.

After the attacker has accomplished that, he now has full access to that web pages content.

2. Persistent XSS : It is also referred to as stored XSS vulnerability. This vulnerability is susceptible to the most powerful kinds of attacks. First, the data is stored on the server (in a database, file system, or other location) provided by a web application. Then it is later reopened and shown to other users on a webpage without any html encoding. An example of this is an online discussion or message board that allows users to sign in to post messages for other users to read. Persistent XSS is one of the more prestigious types of vulnerabilities because the malicious scripts are capable of being provided and used more than once. This means an attacker can exploit this vulnerability and affect a large magnitude of users. In addition to the huge number of users already at risk, this web application can also be infected by a cross-site scripting virus or worm.

Reasons Why XSS Vulnerabilities are Exploited :

- Account Hijacking for identity theft
- Cookie theft/poisoning to acquire sensitive information
- Conduct phishing attacks
- Gain free access to otherwise paid for content
- False advertising

Steps for XSS Attack :

In order to execute a basic cross-site scripting attack, we must follow these four simple steps:

1. Select a target

The first step is to select a target. This is done by searching for an XSS hole in a web-based application on a website.

2. Testing

The next step is to decide what kind of XSS hole this website contains because all XSS holes are different in how they are exploited. We need to check for different attack vectors so as to pass the server filter.

3. XSS Execution

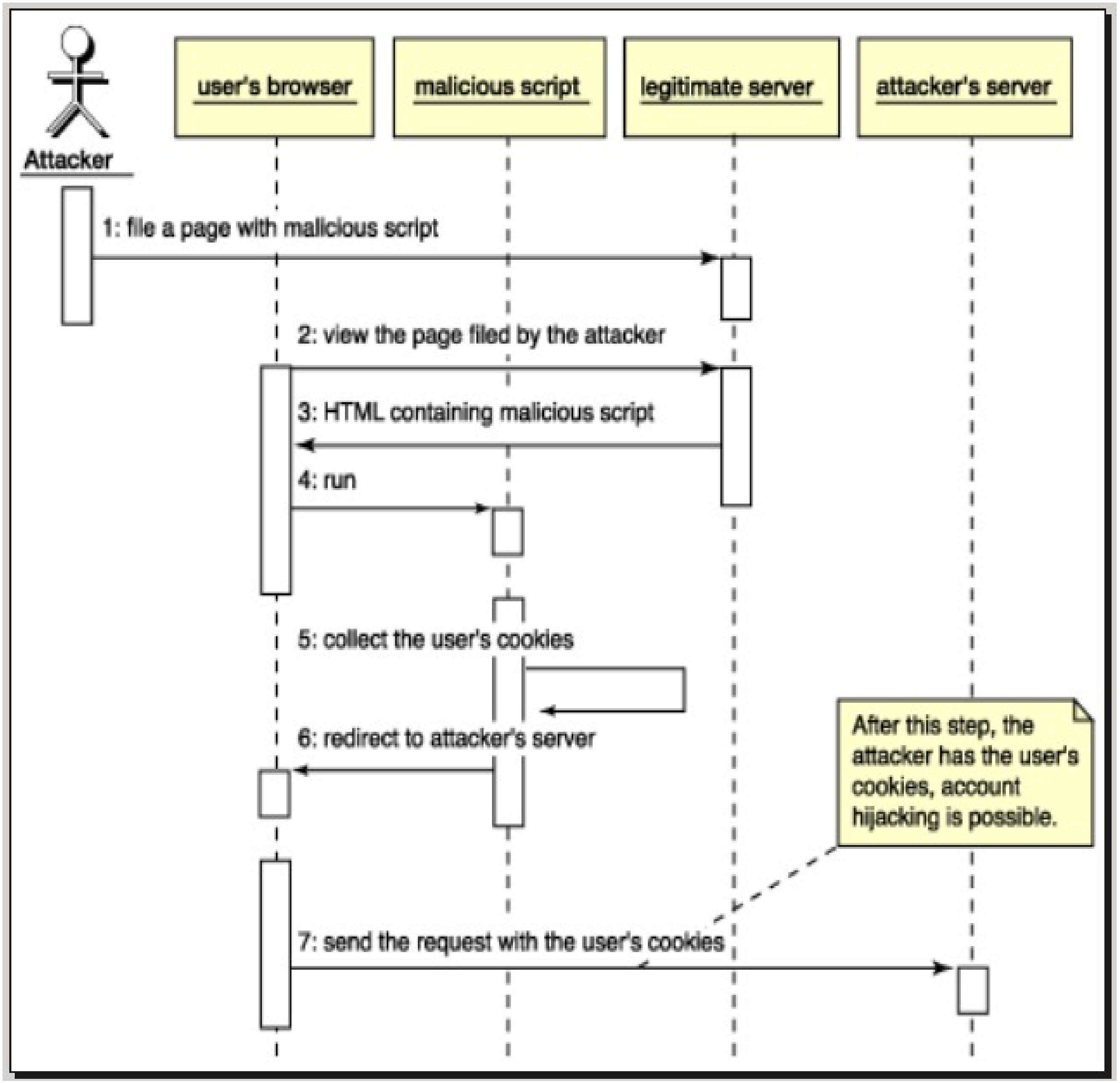
Insert malicious script (ex : cookie stealing script) into the webpage

4. Decide what to do with the data

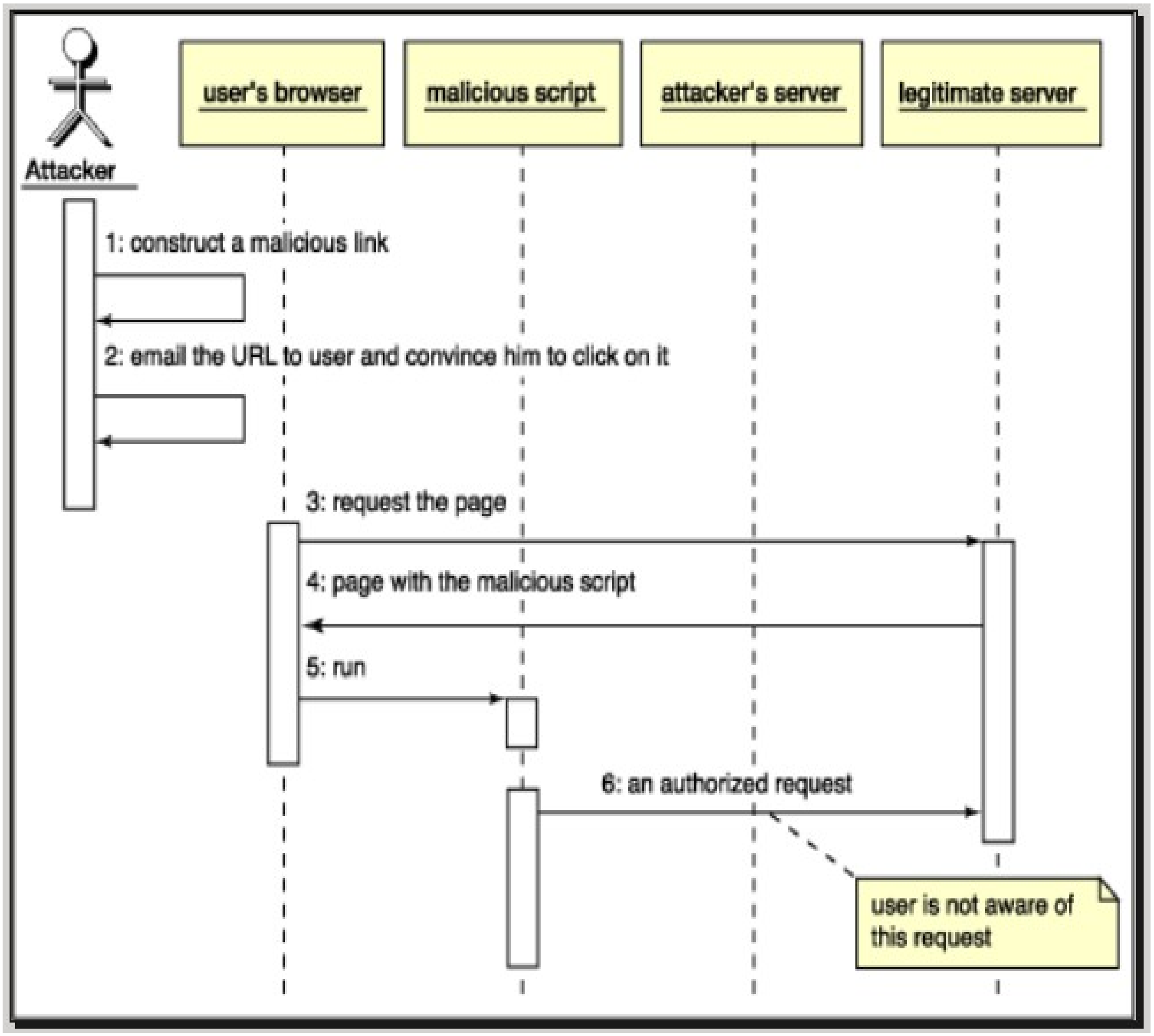
Once you get the user to execute the malicious script, their cookie will be sent to your CGI script. The last thing to do is to see if account hijacking is possible.

Attack Scenarios :

1. **Cookie Stealing** (Persistent XSS): Following diagram shows various steps in the attack:


Attacker
user's browser
malicious script
legitimate server
attacker's server
1: file a page with malicious script
2: view the page filed by the attacker
3: HTML containing malicious script
4: run
5: collect the user's cookies
6: redirect to attacker's server
After this step, the attacker has the user's cookies, account hijacking is possible.
7: send the request with the user's cookies

2. **Phishing Attack (**Non Persistent XSS)**:**


Attacker
user's browser
malicious script
attacker's server
legitimate server
1: construct a malicious link
2: email the URL to user and convince him to click on it
3: request the page
4: page with the malicious script
5: run
6: an authorized request
user is not aware of this request

Defense approaches:

To disallow script execution in untrusted (User entered data) web content, a web application might possibly take one of the following approaches.

1. Content Filtering.
2. Browser Collaboration.

1. Content Filtering:

"The application may attempt to detect and remove all scripts from untrusted HTML before sending it to the browser."

Content filtering is otherwise known as sanitization. This defense technique uses filter functions to remove potentially malicious data or instructions from user input. Filter functions are applied after user input is read by a web application, but before the input is employed in a operation or output to the web browser.

Removal of scripts from untrusted content is a difficult problem for web applications that permit HTML markup in user input such as blog. To be completely effective in eliminating XSS, a filter function must necessarily model the full range of parsing behaviors pertaining to script execution for several browsers.

Challenges of Content Filtering:

Allowing all benign HTML user input, while simultaneously blocking all potentially harmful scripts in the untrusted output.

Every control character that can be used to introduce attack code also has a legitimate use in some benign, non-script context. For example, the ' < ' character needs to be present in hyperlinks and text formatting, and the ' " ' character needs to be present in generic text content. Both are legitimate and allowed user inputs, but can be abused to mount XSS attacks.

Browser behavior vary from browser to browser, they are complex to model, not entirely understood and not all known (especially for closed-source browsers like Microsoft internet explorer). Therefore, from a web application perspective, the task of implementing correct and complete content filter functions is very difficult, if not impossible.

Ex: HTML purifier, NoScript

2. Browser Collaboration:

"The application may collaborate with the browser by indicating which scripts in the web page are authorized, leaving the browser to ensure the authorization policy is upheld."

Robust prevention of XSS attacks can be achieved if web browsers are made capable of distinguishing authorized from unauthorized scripts.

This approach can be implemented by

(a) creating a server–browser collaboration protocol to communicate the set of authorized scripts, then

(b) modifying the browser to understand this protocol and enforce a policy denying unauthorized script execution.

Challenges of Browser Collaboration:

Although this defense strategy is compelling and effective longterm solution, but its implementation will take long time because web applications adopting this approach require their users to employ modified browsers for protection from XSS attacks.

To implement this there must be agreement on some standards for server-browser collaboration, then these new standards must be incorporated in the normal browser implementation. This is a long, complicated process that can take several years.

Ex: BEEP

Defense Requirements:

• Robustly protects against XSS attacks, even in the presence of browser parsing quirks,

• Supports benign, structured HTML derived from untrusted user input, and is

• Compatible with existing browsers currently in use by today's web users.

Objective:

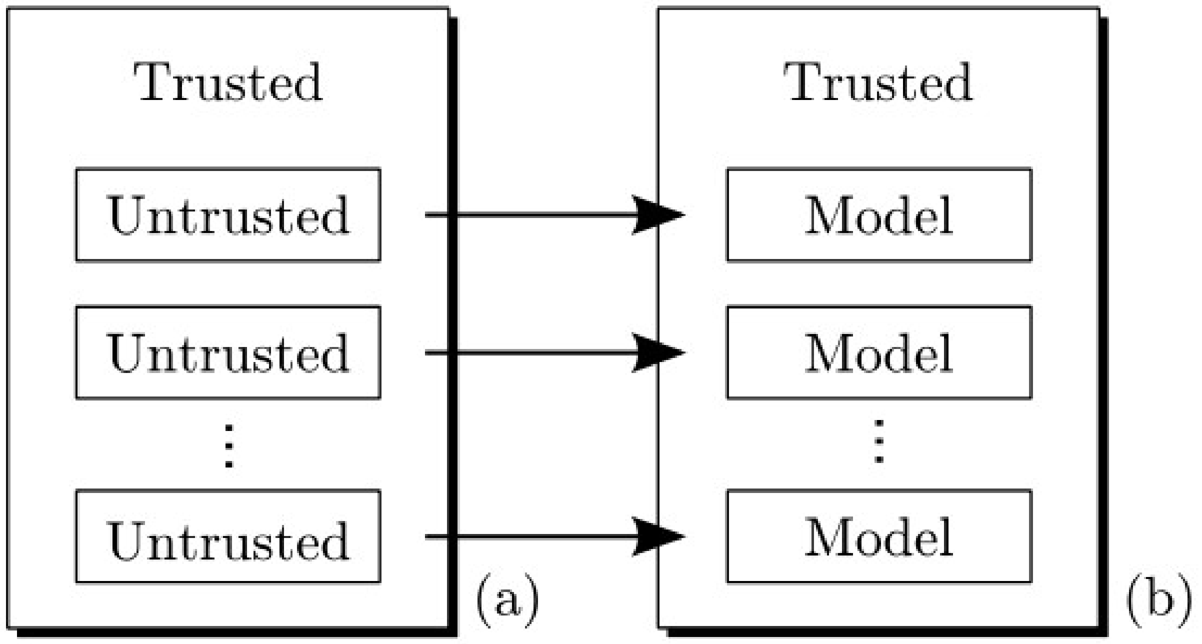
Existing web browsers cannot be trusted to make script identification decisions in untrusted HTML due to their unreliable parsing behavior. So the objective is to enable a web application to effectively take control of parsing decisions and to automatically create a structural representation — a "blueprint" —of untrusted web content that is free of XSS attacks.

Implementation:

Reducing HTML parser influence:

- On the server side, we produce a parse tree from untrusted HTML using HTML parser.
- The parse tree is transformed into a script-free approximation by pruning all nodes not allowed by a configurable whitelist of known, non-executable content types.
- Then the static content parse tree that results after pruning is converted into an encoded form (a model).
- The model encoding employs a map of content types from the whitelist to numeric values, which is decoded on the client-side using a trusted JavaScript library to reconstruct the parse tree.

The figure below gives an overview of this process. Typical web application output (a) may contain several instances of untrusted HTML. Our implementation automatically generates and embeds a model for each untrusted HTML. Thus the application's output is modified by replacing each instance of untrusted HTML with its corresponding model and leaving trusted content unaltered, as shown in (b).



We embed the model in web application output along with a short, trusted script that invokes the client-side JavaScript library, which in turn decodes and safely reconstructs the parse tree within the browser.

```
<code style="display:none;" id="__bp1">
=Enk/sCkhlcmUgaXMgYSBwYWdlIHlvdSBta...
=SkKICAgICI+dmVyeQ===C/k/QIGhlbHBmd...
=ECg===C/Enk/gCiAgUmVzcGVjdGZ1bGx5L...
</code><script id="__bp1s">
__bp__.cxPCData("__bp1", "__bp1s");
</script>
```

Above code shows an actual encoded model and accompanying script for the malicious comment.

Model Generation:

Three general restrictions on models:

1) The model is embedded as text content in a code element (a narrowly defined HTML grammar context).

2) Characters used by the encoded representation are selected from a syntactically inert alphabet(Base64).

3) Text line lengths are conservatively restricted to a maximum length of 65 characters.

By imposing these constraints we restrict the browser's HTML parser, so that it need not to interpret complex HTML syntax from untrusted data.

Base64:

Base64-encoded strings can use a very limited alphabet that contains no HTML syntax control characters:

{a,..., z, A,..., Z, 0,..., 9, /, +, =}

This encoding step enforces a whitelist of syntactically inert characters that ensure predictable transport of untrusted data through the browser's HTML parser.

Limit on length:

We limited line length to 65 for the following two reasons

- some browsers may be implemented without support for arbitrarily long lines of HTML, and may erroneously truncate or alter the data to suit requirements of the parser.
- As we are using Base64 so we are limiting the length of the line to 64 characters.

Model Interpreter:

- On the client-side, we decode models using trusted JavaScript code that we call the model interpreter.

- Decoding is performed by the model interpreter using a reverse map of numeric values to content types, and parse trees are constructed.

- Model interpreter uses a small set of DOM interfaces that are present and exhibit consistent behavior in existing browsers.

- The DOMAPI calls that were used do not recursively invoke the browser's HTML or JavaScript parser and thus can be carefully used without risk of XSS attacks.

Technical specifications:

The Blueprint implementation consists of a server-side component and a client side script library.

The server-side component is written in PHP.

The client-side library(Model interpreter) is included in every web page by linking to an external JavaScript file.

Example:

Let the user entered data is <h1>hi</h1>

At Server side:

1. First the user entered data will be passed through htmlpurifier.
2. Htmlpurifier will block malicious content from the user entered data.
3. The output of the htmlpurifier will be converted into content model based on following rules.
    - Tags and Attributes are mapped into new form (as shown in the appendix).
    - Data with in the Tags will be converted into Base64.

The output will be in the following form

=EZk/EaGk==C/f=

Here

=:	Start of the content model
E:	to represent Tag
Z:	Map for “h1” Tag
k/:	Indicates start of data
E:	for length (4) of data in base64
aGk= :	is the data with in the Tag (“hi”) in base64
C/:	Indicates end of Tag
f=:	Indicates the end of the content model

above content model in included in a small set of javascript code and sent along with the requested page. The final output will be.

```
<code style="display:none;" id="__bp1">
=EZk/EaGk==C/f=
  </code><script id="__bp1s">
 __bp__.cxPCData("__bp1", "__bp1s");
 </script>
```

At Client side:

Here the model interpreter will call the function cxPCData of the model interpreter (javascript file “bp.js”) which will reconstruct the parse tree and will execute the tags by bypassing the javascript parser of the browser.

Conclusion:

Blueprint is a robust XSS prevention approach that was demonstrably effective on existing web browsers comprising over 96% market share. It reduces the web application's dependency on unreliable browser parsers, and it provides a strong assurance that browsers will not execute unauthorized scripts in low-integrity application output. XSS attacks can be avoided with less overheads by adding native browser support for enforcing high-level policies dictated by the web application.

It is a good option for vendors to integrate into the browser unlike the browser based techniques which takes long time for prevention.

Appendix:

For Length:

```
{0:"A",1:"B",2:"C",3:"D",4:"E",5:"F",6:"G",7:"H",8:"I",9:"J",
10:"K",11:"L",12:"M",13:"N",14:"O",15:"P",16:"Q",17:"R",18:"S
",19:"T",20:"U",21:"V",22:"W",23:"X",24:"Y",25:"Z",26:"a",27:
"b",28:"c",29:"d",30:"e",31:"f",32:"g",33:"h",34:"i",35:"j",3
6:"k",37:"l",38:"m",39:"n",40:"o",41:"p",42:"q",43:"r",44:"s"
,45:"t",46:"u",47:"v",48:"w",49:"x",50:"y",51:"z",52:"0",53:"
1",54:"2",55:"3",56:"4",57:"5",58:"6",59:"7",60:"8",61:"9",62
:"+",63:"/"}
```

For Tag:

```
{"a":"A","abbr":"B","acronym":"C","address":"D","b":"E","base
font":"F","bdo":"G","big":"H","blockquote":"I","br":"J","capt
ion":"K","center":"L","cite":"M","code":"N","col":"O","colgro
up":"P","dd":"Q","del":"R","dfn":"S","dir":"T","div":"U","dl"
:"V","dt":"W","em":"X","font":"Y","h1":"Z","h2":"a","h3":"b",
"h4":"c","h5":"d","h6":"e","hr":"f","i":"g","img":"h","ins":"
i","kbd":"j","li":"k","menu":"l","ol":"m","p":"n","pre":"o","
q":"p","s":"q","samp":"r","small":"s","span":"t","strike":"u"
,"strong":"v","sub":"w","sup":"x","table":"y","tbody":"z","td
":"0","tfoot":"1","th":"2","thead":"3","tr":"4","tt":"5","u":
"6","ul":"7","var":"8"};
```

For Attributes:

```
{"abbr":"A","align":"B","alt":"C","bgcolor":"D","border":"E",
"cellpadding":"F","cellspacing":"G","charoff":"H","class":"I"
,"clear":"J","color":"K","colspan":"L","dir":"M","face":"N","
frame":"O","height":"P","hspace":"Q","id":"R","lang":"S","rel
":"T","rev":"U","rowspan":"V","rules":"W","size":"X","span":"
Y","start":"Z","style":"a","summary":"b",
"target":"c","title":"d","type":"e","valign":"f","value":"g",
"vspace":"h","width":"i","xml:lang":"j"};
```

References:

[1] S. Kamkar, “I’m popular,” 2005, description and technical explanation of the JS.Spacehero (a.k.a. “Samy”) MySpace worm. [Online].

Available: http://namb.la/popular

[2] R. Hansen, “XSS (cross site scripting) cheat sheet esp: for filter evasion,” 2008. [Online].

Available: http://ha.ckers.org/XSS.html

[3] World Wide Web Consortium, “Document object model
(DOM) level 2 core specification,” Nov. 2000. [Online].

Available: http://www.w3.org/TR/DOM-Level-2-Core/

[4] E. Z. Yang, “HTML Purifier.” [Online].

Available: http: //htmlpurifier.org

[5] S. Josefsson, “The Base16, Base32, and Base64 data encodings,” Jul. 2003, RFC 3548. [Online].

Available: http://tools.ietf.org/html/rfc3548