Similarity Examinations of Webpages and Complexity Reduction in Web Application Scanners

Daniel Kreischer
3kreisch@informatik.uni-hamburg.de
MIN-Faculty, Department of Informatics
University of Hamburg

Christopher Schwardt
3schward@informatik.uni-hamburg.de
MIN-Faculty, Department of Informatics
University of Hamburg

March 31st, 2008

Contents

I Introduction **4**

1 The HyperText Transfer Protocol **4**
- 1.1 The Design . . . 5
- 1.2 Protocol Versions . . . 7
- 1.3 Request Methods . . . 8
- 1.4 Transmitting Arguments . . . 9
  - 1.4.1 The `GET` Request . . . 9
  - 1.4.2 The `POST` Request . . . 11
- 1.5 Status Codes . . . 11

2 Introducing Cross-Site-Scripting **11**
- 2.1 Same-Origin Policy . . . 13
- 2.2 XSS Types and Examples . . . 14
- 2.3 Filtering . . . 15

3 Web Application Scanners **17**
- 3.1 False-Positives and Vulnerability Duplicates . . . 18
- 3.2 404-Detection . . . 18
- 3.3 Login and Authentication Detection . . . 18
- 3.4 Infinite web sites . . . 19
- 3.5 Finding all pages and functionality . . . 19
- 3.6 Crawl Guidance . . . 20
- 3.7 What Web Application Scanners Cannot Do . . . 20

II The Project **22**

4 Introduction **22**

5 The Design **23**
- 5.1 Starting Point . . . 23
- 5.2 Circumventing the Same-Origin Policy . . . 24
- 5.3 Coping with Sessions . . . 25
- 5.4 Session Riding Defense . . . 26
- 5.5 Recognizing Success . . . 27

6 The Implementation **28**
- 6.1 The Main Loop . . . 28
- 6.2 Classifications of Requests . . . 29
  - 6.2.1 `FORWARD REQUEST` . . . 29
  - 6.2.2 `NEXT VECTOR` . . . 29
  - 6.2.3 `SENT SOURCE` . . . 30
  - 6.2.4 `STATUS REQUEST` . . . 30
  - 6.2.5 `VECTOR FOUND` . . . 30
  - 6.2.6 `Controlling the Scanner` . . . 31
- 6.3 The `Request Generator` . . . 31

6.4 Control Flow 31

7 Evaluation **32**

III Similarity Examinations of Webpages **33**

8 Introduction **33**

9 The Criteria **33**
9.1 Head Data 33
9.2 Comparison of Strings 34
9.3 Document Structure 37
9.3.1 The Document as a List 37
9.3.2 The Structure as a Tree 38
9.3.3 Extending the Tree by using Placeholders 41

10 Weighting of Correlations **45**

11 Integration into the Scan Process **46**

12 Future Work **46**

13 Conclusion **47**

IV Complexity Reduction **48**

14 Crawl Process **48**
14.1 Crawl First 48
14.2 Parameter Occurrences (Part I) 49
14.3 Content Types 50
14.4 Empty pages 50

15 Vectors **50**
15.1 Browser Tests 50
15.2 Filtering Tests 52
15.3 Parameter Occurrence (Part II) 53
15.4 Caching 53

16 Analysis **55**

17 Conclusion and Perspective **56**

Part I
Introduction

1 The HyperText Transfer Protocol

The Hypertext Transfer Protocol is a protocol designed to transfer data across networks. It is mainly used to retrieve web pages and their resources from a web server inside the World Wide Web in order to display it in a web browser.

A communications protocol in our case can be regarded as a standard defining a set of rules of syntax and semantics between nodes in a network, that allows those nodes to interpret received messages as supposed by the sender.

The hypertext the retrieved web pages consist of, is basically semantic content, containing logical links to related information. That links, called hyperlinks, create a net like structure between nodes of knowledge, allowing to connect semantically corresponding content without redundantly replicating it. Furthermore its associative structure is supposed to serve the human brain much more appropriately that large, not clearly arranged collections of linearly organized pieces of information.

The disadvantage of those hypertext is nowadays identified as selectively finding distinctive pieces of information. This is among other reasons due to the general missing of a reading structure provided by the author, such as a guided tour. As a consequence, a highly interconnected hypertext document may lead to information overload that prevents the user to effectively crawl and interpret the provided information.

HTTP itself is part of the so called application layer of the most mainstream network models. That layer is being used, as its name suggests, by applications. In the case of HTTP, that is generally a web browser. In the Open Systems Interconnection Basic Reference Model (OSI Reference Model or just OSI Model) the application layer matches the layers five to seven.

As a matter of principle, HTTP is a stateless protocol. That means, that, after a successful data transfer, the connection between two communication partners does not have to be sustained. If there are more pieces of data to be transferred, a new connection has to be established. As a consequence, reliable information about interconnected transfers have to be implemented on the application layer, as such a mechanism is not native to the protocol.

So HTTP is unaware of information from earlier requests. To work around that property, there is a header field that contains information about cookies to manage sessions.

Sessions are lasting connections, typically between a client and a server within a network, that allow to transfer multiple related packages. Sessions are usually implemented in a layer of a network protocol. In transport pro-

tocols that do not provide a session layer or only support very short-lived sessions, session management may be implemented on a higher level.

Cookies in our case are unique identification numbers that assign a user (i. e. the client that launched the request) to a set of information held at the server, containing details as real life names to personalize the delivered page, facts about the user's preferences or about authorization levels.

By extending its request methods, header information and status codes, HTTP is not necessarily limited to transferring hypertext, but is increasingly used to exchange arbitrary types of data. It is, though, dependant on a reliable transport protocol, which is why typically TCP is employed.

The protocol has been developed by Tim Berners-Lee at CERN, the European Organization for Nuclear Research, in 1989, together with the Uniform Resource Identifier URI and the Hypertext Markup Language HTML, therefore effectively bringing life to the World Wide Web, the way it is known today.

A URI is a generic form of a resource locator called URL, that identifies a network resource by declaring the protocol the resource is to be received by and its location on the network, which is typically concatenated from the host, the port a web service is connected to, delivering the requested resource when queried according to the conditions of the given protocol, and the remote path to the resource.

The Hypertext Markup Language is a textbased markup language that is utilized to structure mainly online content. Apart from the actual content of a document structured by HTML, the document furthermore contains meta information about the text, as the language it is composed in, its author or a summary. HTML has been advanced by the World Wide Web Consortium (W3C) until version 4.01 [Wor99] and is supposed to be replaced by the Extensible HyperText Markup Language XHTML, which is based on the stricter syntax rules of the Extensible Markup Language XML instead of the Standard Generalized Markup Language SGML.

XML is a subset of SGML. It is a markup language designed to represent hierarchically structured data as text files. It is mainly used to transmit information between technically differing nodes in a network.

1.1 The Design

Considering the communicating units between clients and web servers, HTTP differentiates between two types of messages, which are requests, initiated by the client, and responses, with which a web server reacts to a client's query for a resource.

Each message then is composed of the message header and the message body. The header contains information about the body, as the used encoding and the content type, in order to enable the recipient to interpret the message correctly. The body itself contains the payload that is to be transferred.

```
GET /welcome.html HTTP/1.1
Host: www.example.com
```

Figure 1: A simple first HTTP GET Request

HTTP has been designed to retrieve web pages or in principle any given file from a remote computer via a network. Activating a link on a web page pointing to the URL "`http://www.example.com/welcome.html`", a request addressed to the network node with the hostname "`www.example.com`" is created and deployed, inquiring the resource "`/welcome.html`".

For that purpose, first of all the name "`www.example.com`" is resolved by the Domain Name System DNS into an IP address.

The Domain Name System is a web service that responds to a request giving a domain name with the corresponding IP address. Internet Protocol addresses serve the purpose of uniquely identifying and addressing nodes in networks. Corresponding to the function of a telephone number in the telephone network, the DNS functions as a phone book for a computer network.

After resolving the required IP address, the created HTTP `GET` request is by default sent to the standard port 80. The request might look like shown in figure 1.

Characters that are not allowed in a request are percent-encoded. Percent- or URL encoding is a mechanism to encode reserved characters that are usually syntactically interpreted.

The header of an HTTP request or response may contain additional information, as a specification of the utilized browser or details on the language of the transferred document. As soon as the header is terminated by a line feed the web server replies with an HTTP response. It consists of the server's header information, a blank line and the payload, i.e. the content of the reply. Usually the content consists of text composed in one of the markup languages described above, in order to be interpreted by the inquiring web browser, as well as other resources[1] that are necessary to display the delivered information appropriately. A web server's answer may look as shown in figure 2.

The first line of the response displays the protocol version and the status of the response, which is divided into two parts. The first one is a numeric code indicating the class a status belongs to and the second one is a more human-readable description. We will have a closer look at the classification of HTTP status codes in section 1.5.

The server will reply with an error message if the requested information cannot be sent for any reason. The exact process of requests and responses in HTTP is defined in [Gro99a].

[1] e.g. images, CSS scripts or JavaScripts

```
HTTP/1.1 200 OK
Server: Apache/2
Content-Length: 184
Content-Language: de
Content-Type: text/html
Connection: close

<html><head><title>Welcome to the World of Examples!</title>
</head><body><h1>You are welcome!</h1><p>Enjoy your stay and
the plenty of useful information we provide!</p></body></html>
```

Figure 2: The Response to the Query of www.example.com/welcome.html

Generally files in arbitrary formats may be transmitted. They may even be dynamically generated as in PHP applications and therefore do not have to be actual files on the server delivering them.

1.2 Protocol Versions

There are currently two versions of HTTP in use, which are HTTP/1.0 and HTTP/1.1. HTTP/1.0 starts a new TCP connection for every request and terminates it when the response has been transferred. So if there are five images in a page, it will take six TCP connections to fetch the page and all its resources.

HTTP/1.1 may fetch several request and responses over one TCP connections. As the speed of TCP connections is due to the slow start algorithm relatively low at the beginning, the combination of several transfers increases load times significantly. In addition, interrupted transfers can be resumed with HTTP/1.1.

Apart from receiving data from a server, with HTTP/1.1 it is possible to transfer data to a server. Utilizing the `PUT` method, web designers might publish their pages via the web server, removing them again with `DELETE`. Addionally, HTTP/1.1 offers a `TRACE` method for checking a packet's way to the web server and for verifying that the data being transmitted is received correctly.

To avoid the possibility of an attacker reading the data transferred via the plain text HTTP on any node in the network a packet is routed on, connections may be encrypted employing HTTPS.

The Hypertext Transfer Protocol over Secure Socket Layer is built upon HTTP but it uses 443 as the default port and inserts an additional layer handling authentication and encryption between HTTP and TCP. It offers reasonable protection against eavesdropping and man-in-the-middle attacks.

SSL and its successor TLS (Transport Layer Security) are cryptographic

protocols that offer secure transmissions across unsafe networks.

A man-in-the-middle attack is a form of eavesdropping, in which an attacker pretends to be the desired communications partner for two victims and relays the messages between them, making them believe they are communicating directly with eachother over a private connection, while in fact the attacker is able to read all the data being transmitted.

1.3 Request Methods

`GET` is the most commonly used method. It requests content from a server. It is defined as *safe*, as it is supposed not to change the state of the server, i. e. not to have side effects, but only retrieve information.

`POST` is very similar to `GET`, additionally transferring a data block. It is commonly composed of name-value pairs from html `<form>`s. `POST` is considered an *unsafe* method, as it is supposed to change the state of the web server.

In principal, data may be transferred via `GET` requests as parameters in the URI, but in a `POST` request, they are much less visible to the standard user. That discreetness may be advantageous when transmitting sensitive data. In addition, the allowed number of transfered characters is much higher in a `POST` request's data block.

`HEAD` instructs the server to reply with a standard `GET` or `POST` header, but drop the actual content. It is a way to quickly verify the validity of a page in the browser cache.

`PUT` serves the purpose of uploading files onto the server. It is mainly deprecated nowadays, hardly implemented in webservers anymore and if so, usually deactivated in standard configurations.

`DELETE` corresponds to the previously described `PUT` method and removes files from the server. It is equally obsolete.

`TRACE` returns the request as it was received by the server. It is useful to reproduce if or where the request was manipulated on its way to the server, which is oftentimes sensible when debugging connections.

`OPTIONS` delivers a list of methods and features supported by the queried web server.

`CONNECT` is usually implemented by proxy servers that are capable of offering SSL tunnels.

The methods `HEAD`, `GET`, `OPTIONS` and `TRACE` are considered *safe*, as they are designed to not have side effects, i. e. to not change the state of the server in addition to returning the result page.

`POST`, `PUT` and `DELETE` requests on the other hand are considered *unsafe*, as they have been created to transmit (possibly tainted) data and change the state of the server. Therefore the use of an unsafe method should be made visible to the user by displaying for example an HTML button rather

```
category=web_security&flaw=xss&type=non-persistent
```

Figure 3: A List of Name-Value Pairs

```
GET /breedingGeeks.php?category=web_security&flaw=xss&type=non-persistent HTTP/1.1
Host: www.example.com
```

Figure 4: Requesting Information on XSS with a GET request

than a link, indicating that sending that request will influence data on the server.

1.4 Transmitting Arguments

Oftentimes the user wants to transmit information to the web server. Generally there are two ways to reach this goal with HTTP.

First, the data may be transfered together with the request for a resource, encoded as parameters in the URI, as done when triggering a `GET` request. When doing so, the information is contained in the URL and saving a link will preserve the carried information. This may be sensible when emailing a link to a product of a web shop, but will probably be highly undesired when containing sensitive data as passwords.

Second, the data may be transfered in a special block of a `POST` request. As the information is in that case not contained in the URL, not only can more information be transmitted, but data affected by data protection laws is also slightly better protected against misuse. Usually, information transmitted in `POST` requests is URL encoded.

In the following parts, we will have some closer looks at those two methods.

1.4.1 The `GET` Request

In case of the `GET` Request, data that is to be transmitted will be encoded in the query part of the URI, starting with a "`?`".

Usually this approach is chosen to transmit parameters that are to be considered when interpreting the request on the server side. It typically consists of a list of name-value pairs divided by "`&`". An example of such a list can be seen in figure 3.

If now a user navigates through our page at "`www.example.com`" and finally clicks the menu button for explanations of non-persistent XSS, the browser will send a request like the one shown in figure 4.

So three name-value pairs are handed over to the server at "`www.example.com`", of which the first seems to indicate the category of the requested topic,

name	value
category	web_security
flaw	xss
type	non-persistent

Figure 5: Parameters when reading on XSS

```
HTTP/1.1 302 Moved Temporarily
Date: Mon, 31 Mar 2008 11:25:14 GMT
Location: http://www.example.com/concentrated_knowledge/non-persistent_xss.html
```

Figure 6: A Redirect Instruction from a Web Server

the second one determines the actual subject and the third one denotes a relevant aspect more specifically (figure 5).

After those pairs have been concatenated (remember figure 3 for an example), they will be appended to the URL after the question mark as seen in figure 4.

That way the server at "`www.example.com`" knows upon which criteria it has to evaluate its database and with which information it has to fill the result page.

Generally the queried server may use the parameters derived from a `GET` request either to fill a template page with dynamically generated content according to the users needs or to refer the client to the location the actual file holding the information is placed.

In the first case, the server would just dynamically retrieve information from an attached database, generate a result page (probably from a template) and return the result in the same manner it did with the *welcome.html* (figure 2).

In the second case, the server would answer with an error code from the 300 range (more details on HTTP error codes in subsection 1.5), indicating a redirection, as can be seen in figure 6.

The browser will follow the instruction and fetch the denoted resource (figure 7).

Receiving that valid request, the server will respond with the queried resource, as he delivered `welcome.html` in figure 2.

```
GET /concentrated_knowledge/non-persistent_xss.html HTTP/1.1
Host: www.example.com
```

Figure 7: Fetching the New Resource after the Redirect

```
POST /breedingGeeks.php HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

category=web_security&flaw=xss&type=non-persistent
```

Figure 8: Requesting Information on XSS with a POST request

1.4.2 The POST Request

If we had used a POST request instead of a GET request in section 1.4.1, the parameters necessary for the server would not have stood in the URI, but in the body of the request (figure 8).

It would not have made much sense in our case to use a POST request, because as explained in section 1.3, POST requests are designed to transfer data that changes the state of the server, whereas we only want to retrieve information. So, our link in the page menu should clearly deploy a GET request. In an HTML `<form>` though, a POST request might well make sense. It could easily be triggered by changing the `method` attribute of the `<form>` tag from the default `method="get"` to `method="post"`.

Nevertheless, the server would have understood the information transferred in the POST request as well and either delivered the requested resource or responded with a redirect instruction (see figure 6).

1.5 Status Codes

Each HTTP response contains a status code, indicating whether the associated request was dealt with successfully. If not, the status code gives a reason for the occurrence of the error and may indicate a way to receive the requested information.

For that purpose, HTTP status codes are classified into categories (figure 9).

In addition to the status code, the header of the response contains a defined description of the error in plain text. As an example, a 404 error will display an error like figure 10.

For a more detailed description of HTTP error codes, see [Gro99c] or [Wik08c].

2 Introducing Cross-Site-Scripting

XSS is at the moment probably the widest spread disease of web applications, maybe on one level with Session Riding. Some extrapolations say about 70

	Status Code	Description
1xx	Informational	The request is still being handled. This feedback may be necessary with long processing times, as some clients otherwise would suspect an error while transmitting or editing the request and fire a timeout.
2xx	Successful Operation	The request has been processed successfully. The answer is being returned.
3xx	Redirection	The requested resource may now be found in an other destination. This may appear if the web page has been restructured. The server will designate the new location in the `Location` header.
4xx	Client Error	An error occurred that is within the responsibility of the client. A `404` code indicates that a resource has been requested that does not exist on the server. A `403` error gives notice that the client is not authorized to access the queried resource. This may be the case with confidential information.
5xx	Server Error	The server caused an error. For example a `501` status code reveals that the server does not possess the functions necessary to handle the request.

Figure 9: Categories of HTTP Status Codes

```
HTTP/1.1 404 Not Found
```

Figure 10: An HTTP Error Code

percent of the current web application are vulnerable to XSS [Rel08]. As so many new web application spring up like mushrooms at this time, it is a good period for security consultants to hawk at companies with an internet appearance. But it is a very tedious job to search for all possible flaws only by hand, as the most wide spread flaws look the same or at least alike.

This is where all of the security scanner developers step in. But as the result of benchmarking tests evince most of them are not doing a very sophisticated job ([WWSS06] and [Sut07]).

So what is XSS all about? XSS is the injection of code, mostly JavaScript, into a foreign web page.

The great danger here is that JavaScript is powerful enough, to change the entire contents of a page, vulnerable for XSS. With the aid of iframes, and the XMLHttpRequest object, famous since the arise of Web 2.0 technologies, even complex operations can be performed on behalf of the user, without him know about it. Moreover, this way cookies may be read and changed. This allows the attacker to change the behavior of the site, take actions on behalf of the victim, or read sensitive data. XSS is one of the few methods to circumvent the Same-Origin Policy.

Ultimately, the possibilities with XSS are almost only restricted by the creativity of the attacker. Slowly the first major worms begin to afflict the momentarily so popular online communities [Sam05].

2.1 Same-Origin Policy

The philosophy of the Same-Origin Policy is simple: it is not safe to trust content loaded from any other website. As semi-trusted scripts are run within the sandbox, they should only be allowed to access resources from the same website, but not resources from other websites, which could be malicious.

The term "origin" is defined using the domain name, protocol and sometimes port. Two pages belong to the same origin if and only if these three values are the same. To illustrate, the following table gives examples of origin comparisons to the URL "`http://www.example.com/dir/page.html`".

URL	Outcome	Reason
`http://www.example.com/dir2/other.html`	Success	Same protocol and host
`http://www.example.com/dir/inner/other.html`	Success	Same protocol and host
`http://www.example.com:81/dir2/other.html`	Failure*	Same protocol and host but 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

The Same-Origin Policy does not apply on all properties of all HTML objects. For example the proportions of an image may be accessed, even if it originates from an alien domain.

For some applications the Same-Origin Policy is too restrictive [Fla02]. In our scanner for example we needed to circumvent the Same-Origin Policy

somewhat too, which gets described further in the architecture section at 5.4.

But also in the usual web application world this problem arises, as for example cooperating web applications that need to read data from each other. There are already some approaches to soften the same origin policy, like in flash the `crossdomain.xml`. This document resides on the side of the web server, that gets requested from an flash application originating from another domain. It tells the flash player whether the requesting domain is allowed to request the data or not, like an access control list. [Ado]

Such feature is also implemented into newer browsers, respectively newer versions of browsers to work with the XMLHttpRequest object [xhr08].

2.2 XSS Types and Examples

XSS vulnerabilities are divided into three different types. Following they get described by example [Wik08a].

Type 0 is called DOM based / local XSS [Kle05]. In contrast to the other two types in this technique the webserver plays a minor role. If a page, for example, contains something like

```
<script>
  var pos=document.URL.indexOf("name=")+5;
  document.write(document.URL.substring(
    pos,document.URL.length));
</script>
```

arbitrary content may be added to the page by placing the desired content after `"name="` in the URL. There is even the possibility, to do that in a very clever way, so that the server is not able to recognize the attack at all.

```
http://some-domain.tld/index.html#name=
  <script>alert('Local XSS');</script>
```

Because everything after a `#` is not interpreted by the browser as a part of the request, this part is excluded from the request to the server. So the meaning of local is that all the steps leading to the vulnerability are processed and kept on the client side. Whereas in the other two methods, the malicious code is included by the webserver into the document.

Type 1 is called non-persistent / reflected XSS. In this example, the search string is echoed on the page of search results. The relevant section of the code results page:

```
<?php
  echo "Search results for "
    .$_GET[search].".";
?>
```

One can therefore inject arbitrary strings into the page, and hence new HTML elements. On the request of

```
http://some-domain.tld/search.php?search=
  <script>alert('Reflected XSS');</script>
```

a alert box with the text "*Reflected XSS*" would appear.

The type 2 is called persistent / stored XSS. Unlike in type 1, in type 2 the malicious content gets stored on the server, so that when requesting the page on which the malicious content resides, it does not have to be contained in the request to the server. The example described here is a messaging service. The messages are saved as follows.

```
$sql = mysql_real_escape_string(
  "INSERT INTO messages
   (`from`, `to`, `msg`, `id`) VALUES
   ('".$from."', '".$to."',
    '".$_POST[msg]."', '".$to."')");
mysql_query($sql);
```

When a message gets stored and is retrieved afterwards, the contents of the message are directly included onto the page.

```
$sql = "SELECT * FROM messages
        WHERE id=".$messageID;
$result = mysql_query($sql);
$row = mysql_fetch_array($result);

 ⋮

echo $row[msg];
```

Would a malicious user of this service send the message

```
<script>alert('Stored XSS');</script>
```

to another user, this would open an alert box with the content „*Stored XSS*“.

2.3 Filtering

The most frequently cause for the occurrence of XSS is the developers lack of understanding what input is used within which context and the possible consequences arbitrary input may have. To solve the XSS issue standard procedure is to filter out or sanitize characters that are used to break out of the current context. Having this in mind it seems to be pretty easy to avoid XSS, but as the applications grows in size it is hard to keep track of all input and where it is going to used. So there are two options for the time to filter.

```
if data.contains("script"):
  rejectRequest()
else:
  process(data)
```

Figure 11: Filter Script Code Snippet

On the one hand input can be filtered as soon as it enters the application. If done conscientiously no input gets into the application without being filtered. But in doing so, there must be clearly defined in which context the input occurs in. Besides the fact that it is hard to know where the data will occur on beforehand. Another problem is that different situations need different treatment. For example data needs to be sanitized different when being passed to an SQL database than being passed to the users browser or a shell script. Data would be needed to go through several sanitizing steps if it perambulates several of those components and these steps must be done in the same order as the data will pass the components, to make sense. Knowing every time the entering data will be passed to other subsystems of the application is hard to accomplish and also needed to be updated every time it changes, which will almost surely be overlooked if the application gets updated, upgraded or extended.

On the other hand input may be filtered every time it passes to a subsystem. This is a more sophisticated approach, because the filtering or sanitizing can be fitted for the exact purpose it is needed. Usually every current language has built-in functions to accomplish most common filtering scenarios. But still too often developers create functions to sanitize data on their own, without understanding completely why and how it has to be done correctly. This is for example frequently the case if the user should be able to add some HTML elements. The developer has to understand the effect of all elements and its attributes he is allowing, to not take the risk of being exploitable.

This leads us to another discrimination of filtering methods called black- and white-listing. Black-listing means having a list of patterns that are forbidden, whereas white-listing means working with a list of allowed content. In general white-listing should be the preferred approach, because when using black-listing the developer has to know every appearance of malicious content, which is hard to achieve, since elements and attributes may change over time. White-listing instead allows only content that is known to be safe an will not raise vulnerabilities in the web application.

A rather simplified example for illustration purpose. The web application rejects every request that contains the string "*script*" by using the code from figure 11. Bypassing this filter is pretty easy. All the attacker has to do is to replace every occurrence of the string "*script*" by some sort of camel cased version, like "*ScRiPt*". The filtering function will not recognize the string

```
<img src="http://no.where/notfound.jpg" onerror="alert('XSS')" />
```

Figure 12: Exploiting the filter script

anymore and will process the data.

But even improving this function by making the string search case insensitive will not prevent from exploitation. The attacker may still insert content like in figure 12 that will result in a pop-up saying "*XSS*".

To solve this flaw the developer has to know the context in which the data occurs in. We assume it is echoed into a text node of a `div`-tag. In order to prevent this vulnerability all content that enables to break out of the text node context needs to be sanitized or filtered out. For all special characters, that have a particular meaning within an XML text node, there are escape sequences to represent the desired character [Fly07]. Implementing such a replacement function will prohibit the injection of JavaScript and other content breaking out of the XML text node context.

3 Web Application Scanners

In the early stages of the internet the world wide web consited of static documents that were simply delivered by the web server to the browser and looked for everyone the same. Dynamic contents were dreams of the future and almost no site authenticated its users. The web has completely changed since those times. Nowadays the web sites are like real applications. Most web applications have a restricted area where only authenticated user may go and have personalized appearance depending on the user. Especially since the keyword web 2.0 came around there was a huge rise of more dynamically designed web sites and the urge to integrate the user as much as possible into the content. This means also to abandon the mindset of having all content pregenerated on the server side, towards having the client side involved more and more.

Today almost everyone has a web presence, particularly companies and organizations turn their attention increasingly towards this appearance. In many cases those web applications get written by developers that are not aware of the variety of vulnerabilities there are these days and did not get educated regarding web application security. The security awareness is not yet as mature as it is in longer established fields as in operating systems or networks. To ease the situation for them and the potential auditors, web application scanners are being developed to automate the search for vulnerabilities as good as possible.

Many CGI scanners are wrongly called web application scanners. In effect they do really detect vulnerabilities that were previously known in

specific pages or files [DC05]. Popular examples for this are Nikto [nik] or Metasploit [met]. Whereas web application scanners are supposed to find new, previously unknown vulnerabilities, even in individual web applications.

Most current web application scanners suffer from large false positive numbers, which implies unneeded work for the penetration tester or developer.

Jeremiah Grossman divides the major problems when developing a web application scanner into 5 topics [Gro06].

3.1 False-Positives and Vulnerability Duplicates

These are two unwanted effects, that can be noticed in all types of vulnerability scanners. The first, false-positives, are the occurrence of positive findings by the scanner, which turn out to be false finding, because the reported vector does not actually work. To have them sorted out by hand is a huge waste of time and gets even enlarged by vulnerability duplicates.

As the words vulnerability duplicates suggest, this expression means finding one and the same vulnerability more than once. It may be the same spot just exploited by another vector or another place, but the identical injection generated by the same vector injected in a parameter shared across different CGI's.

The conclusion report may be hard to subsume into well defined categories, to not end up in a huge series of vectors.

3.2 404-Detection

Most current web application scanners try to guess files and folders that might be left on the webserver, but are not linked at. This might be backup files of existing files like login.php.bak or login.php.old or files and folders that disclose important informations like a left behind admin folder.

So far the only problem here seems to arise is enumerating all possibilities, but in the real world it is not that easy. Usually web servers are supposed to return a 404 status code [Gro99c], however sometimes there are web server handlers that return a 200 status code, which normally means everything went well with the request. That makes it impossible for an automated scanner to clearly state whether the file or folder does exist or not. But the status code is not the only strangeness the scanner might encounter. Other behaviour may contain things like multiple stages of re-directs or dynamic not found page content. All of these circumstances are potential causes for false-positives.

3.3 Login and Authentication Detection

At first glance it sounds pretty simple to recognize and keep track of the user authentication. Just define a username and password, receive the au-

thentication cookie and you are done. But reality proves that wrong. "*Some POST, some GET. Others rely on JavaScript (or even broken JavaScript), Java Applets, a specific web browser because it checks the DOM, multiple layers of cookies and redirects, virtual keyboards. The list and combinations is endless.*" [Gro06] Implementing all these technologies is tedious work and needs to be updated every time new technologies are introduced or web developers got creative again.

The other important concern is to determine if the scanner is still logged in to the application. There are multiple scenarios of logging the scanner out. Like when "*The crawler hits the logout link, session timeouts, IDS systems purposely causing logout on attack detection*" [Gro06]. And being logged out while the scan procedure will pollute the results and may lead to further false-positives and miss parts of the application.

3.4 Infinite web sites

In many cases sites have a huge amount of pages of which many or probably most of them share the same functionality. This problem is also referenced to as the "calendar problem", because this is a very clear illustration of the issue. Other examples are Amazon or eBay, that contain huge quantity of product sites, that refer to the same functionality.

Furthermore there are possibly places in the web applications, where the crawler itself may generate new pages that need to get crawled again or the page generates new dynamic links. Both can lead into further infinite or at least not feasible crawling and scanning effort.

Due to the fact, that most upcoming web sites try to be more and more dynamic, the probability having the scanner trapping itself in an infinite crawl process is increasing. A web application scanner needs to be smart enough to detect that it is trapped and stop crawling this very large or infinite branch of the application.

3.5 Finding all pages and functionality

This problem is rather similar to the infinite web site issue. In order to find all possible vulnerabilities within a web application, the scanner needs to know all fragments of the web site. But in black box testing it is impossible to be sure if the whole application has been discovered or not.

There are several scenarios, so the scanner would not find every part of the web application. Parts may be only disclosed to users by email or other personal messages, that cannot be accessed by the scanner. Specific functionality may also be "*buried several form layers deep, accessible by only specific users, or hidden behind/inside JavaScript/Applets/Flash/ActiveX. Sometimes the only way to find this stuff is manually and even then your guessing you found it all.*" [Gro06]

3.6 Crawl Guidance

The previously five mentioned challenges are not the only issues to be taken care of, but covers a broad, very important portion of it. Another aspect that needs to be defined and coped with before developing a web application scanner is to decide on the type of crawl process. These types are divided into three members.

The first is automated crawling. This is, as the name proposes, a completely automatic process wherein the user does not have to interact with the application. With this type of crawling it is especially important to have an effective crawl engine, to not miss important parts of the web application.

Manual crawling is the opposite to this. Here the crawler only saves the pages the user himself visits and does not do anything on its own. The user has to do all the work, whereas the crawler simply traces him and saves the points of the path. In large web applications this approach has another drawback besides having the user to do a lot of work, but also he needs to plan some kind of route through web site, to on the one hand, not missing anything and, on the other hand, not doing too much redundant work, by visiting pages more than once. If the user decides to not include particular parts of the web application to save effort, flaws may be lost out on.

Semi-guided crawling however mixes those two methods, so that the scanner does crawl the application on itself but is aided by the user. There are many different kinds of semi-guided testing. It may already be having the user logging in at the beginning of the crawl operation, or that the user clicks himself through the web application and the crawler tries to crawl all pages he is able to. The lastly described methodology would probably result in the best crawl outcome but still needs extensive user interaction, maybe even as much as with manual crawling, because the user does not know if the crawler did already crawl the page he is going to visit.

3.7 What Web Application Scanners Cannot Do

Even though web application scanners can simplify and reduce the work to be done by a security auditor, nevertheless there are some points that cannot (yet) be done by those aiding applications. Some examples of these shortcomings are the following.

Access Control

A scanner is not able evaluate the significance of the data or functions it discovers. It is also not able to understand the access control requirements that are relevant to the application. So encountering situations, which enable a user to achieve higher privileges than he is supposed to have or access data from other users, he is not supposed to see, will not be conceived by the scanner.

Intelligent Parameter Modification

Utilizing a parameter that has a definite meaning to the web application to achieve behaviour the application was not assumed to have cannot be obtained with a scanner, because it is unable to comprehend the semantics used by the applications.

A simple example is a shopping web site, where the price is written into a hidden form field and gets this way passed to the shopping cart. Changing it to a lower price is certainly a flaw of the web application, but will not get identified by the scanner.

Other Logic Flaws

Vulnerabilities based upon logical flaws, such as bypassing steps of a password recovery by passing distinct parameters or using a negative value to beat a transaction limit.

Design Flaws

Shortcomings occurring because of poor design, like easily guessable password hints or weak password rules. Those flaws are very hard or even unable to be found by an automatic scanning process, because of its manifoldness. Every developer thinks (a little bit) different, and so the appearance of such flaws, even if it is a flaw of the same category, will be a little bit different in other applications.

Weak Session Algorithms

Being able to predict the web applications session tokens, because the algorithm used to create those tokens is too weak, would enable an attacker impersonate another user. So even if the scanner is able to recognize that a parameter has a predictable value over several logins, it still needs to know what this value is used for and at best reassure that that it really does what it is presumed to do.

Information Leakage

Leakage of sensitive or other information the user is not supposed to have access to. For example username listing or logs, that could contain valuable information. The scanner would need to be able to automatically distinguish between information that it has rights to access and information that it should not be able to access.

Most shortcomings the current web application scanners are suffer from would need full-blown artificial intelligence engines in order to get addressed. That is because it is comparatively easy to develop a scanner based on syntactic content, but very hard to introduce semantics to the contents.

Part II
The Project

4 Introduction

The goal of this project is to develop a vulnerability scanner for web applications with main focus on XSS. It is designed to be a black box scanner. This means, that the source code is not known to our scanner. Only the results of requests are used in the penetration testing process. It should generate at least as good results as comparable black box scanners in the category of XSS.

During the orientation and implementation some innovative approaches have been discovered, that separates this project from commercial black-box scanners. It also contains many thought-provoking impulses from [Sha07], who described a first departure of utilizing the browser to improve the scanning results. Probably the most noteworthy innovations are the subsequent.

In our scanner we use an innovative approach of identifying vulnerabilities. While most scanners only analyze the resulting code of the page or at most use a self-written or lent HTML and JavaScript parser to find their injections, we utilize the browser itself.

The scanner consists of two parts. First the proxy, which holds most of the logic. It is responsible to generate the requests that should be done and does (most of) the crawling activity. The second part is a browser. Its chore is to evaluate the resulting page.

This approach guarantees, that the used injection is working for the web application, at least for this very browser. Browserspecific bugs are getting more and more attention, since many standard vulnerabilities get better and broader known, but quite frequently are not handled the way it should be done, leaving little holes, that may be exploited with one browser or another. Having no false positives is a big step in fully automated scanning, since no user has to revalidate the results. The only thing we now need to concentrate on is not leaving vulnerabilities unnoticed.

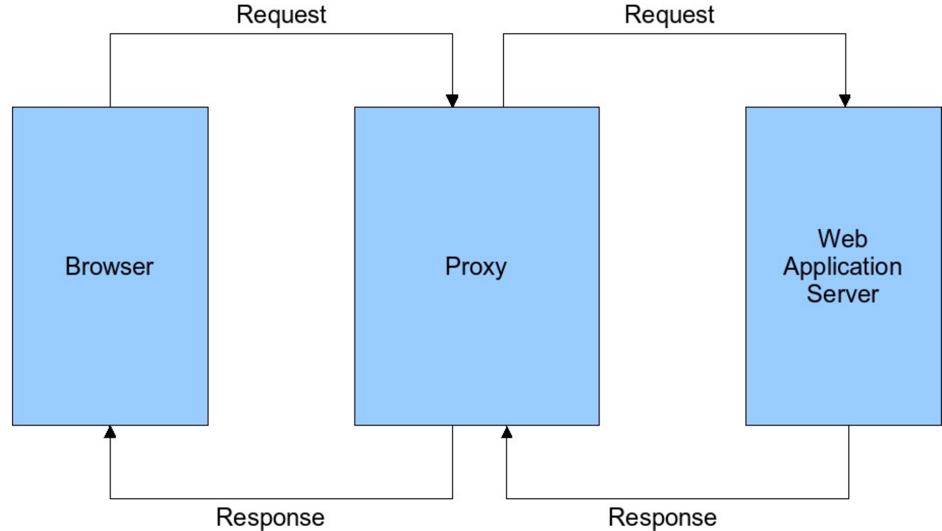
Another notably introduction is the execution of event handlers. Hereby more dynamic pages can be approached, since more and more of the functionality is evaluated dynamically on one page and user interaction is very basically simulated. This is also aiding the discovery and testing of XMLHttpRequests and their results and implications for the content. Also client-side script evaluation can be checked rudimentarily.

5 The Design

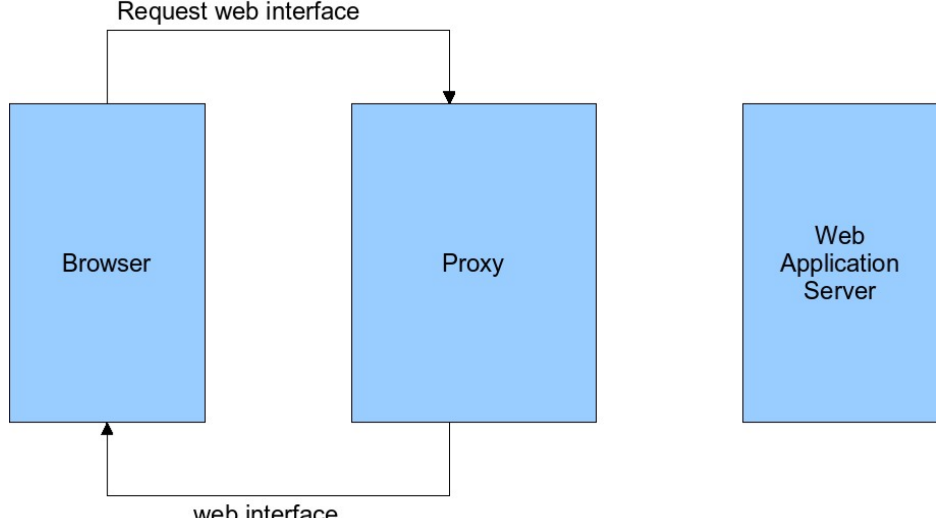
5.1 Starting Point

In order to start a scan a starting point is needed. Since, for the user, the whole application is reachable through a webinterface, there must be a defined URL to reach the scanner itself, but not blocking the residual internet access.

When surfing the ordinary web the scanner behaves like a transparent proxy, which means "A 'transparent proxy' is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification" [Gro99b]. The image below describes the transparent proxy functionality.



In contrast to this the proxy recognizes the calls of the pre interface, that is represented as an arbitrary configurable domain, and the web interface, which is represented as an also arbitrary and configurable filename, are intercepted and the proxy replies the desired pre interface respectively the web interface source code. This behaviour is illustrated in the image below.

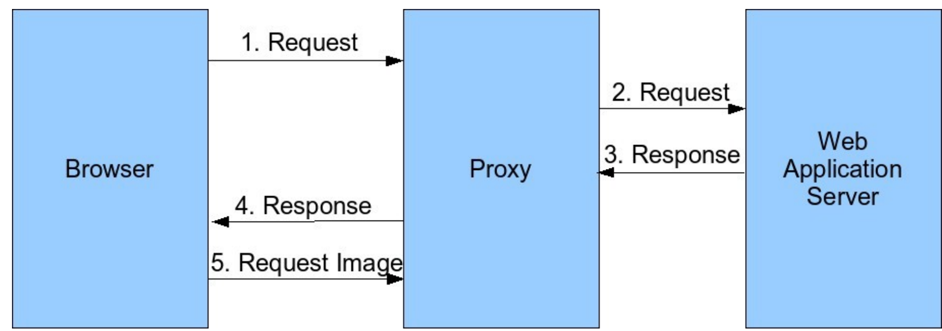


As may be seen in the illustration the server does not receive the requests.

5.2 Circumventing the Same-Origin Policy

The client side of our application needs business logic too. It is implemented in JavaScript, but this raises a problem generated by the same-origin policy. The problem is to access the data displayed within an iframe from the business logic to evaluate it.

A solution to this is to place the business logic into the domain of the application that is to be scanned. That means when a special URL is requested the proxy recognizes it and returns the business logic of the scanner. This way everything within this domain is accessible to the client side application part. This rises only one little drawback. If the web application, that is to be scanned, itself has a page called the way the business logic request string is named it will affect the scan results.



Another approach to this problem could be to use the new XMLHttpRequest object functionality as described before in 2.1. In every requested page could be an access-control header included, like "Access-Control: allow <scannerDomain.dom>", if the business logic should reside on scannerDomain.dom. But this will exclude most current browsers and all older versions.

Similar behaviour is acquired with the flash solution described in the same chapter, when implementing the business logic in flash instead of JavaScript. However this leads to the same problem of having at least a defined flash version installed with the browser, otherwise the scan will not work.

5.3 Coping with Sessions

As in many web applications large areas are only accessible to a user if he is validly logged in, managing sessions is of essential meaning to a web application scanner. Not doing so would in many cases mean to miss out on most parts of the application.

Initially this seems to be a negligible problem. When the scanner's web interface requests the home page of the web application that is to be scanned, the application will usually send a session identifier[2], typically as a cookie or, if cookies are disabled by the browser, as a parameter dynamically added to all links in the page. When logging in, the value to the session identifier's key is set. In any future requests to the domain of the tested web application, the browser will automatically attach the session information which will then identify the session on the server-side.

On second thought it becomes apparent that ensuring that the session is not terminated is much more demanding. When penetration testing the application, the scanner has to provide that the session is not terminated by a request it sets off. To do so, it requires knowledge about which criteria, when matching a request, will elicit a logout event. As there are no such criteria and every web application may handle the logout process individually, there is hardly any possibility to equip the scanner with the necessary knowledge base to perceive a fundamental amount of those session terminating requests.

An obviously severely limited approach is to search all found URLs as well as all `GET` and `POST` parameter's key-values pairs for promising strings like 'logout'.

As the scanner cannot have the knowledge at its disposal that would be required to reliably perceive requests that will cause the termination of the session, it has to gain it. This learning process is initiated by the user before exemplarily logging out. Informing the web interface that the next request will be part of the logout process will enable the interface to defer all variants of this request in the scan process. This learning process would have to be

[2] If the web application makes use of sessions at all.

repeated by the user for all of the web application's logout possibilities.

As the reliability of this learning phase depends on the user's accuracy and expertise and additionally deferring the perceived request does not completely eliminate the problem of being logged out[3], the scanner might try to check whether the session is still valid. It might do so by comparing a given webpage to its information about the source code when being logged out respectively when being logged in and then if necessary deploying a determined request to log in again, attaching the deposited credentials. Of course this proceeding is highly unreliable when examining dynamic pages with changing contents.

5.4 Session Riding Defense

The most widespread solution for dealing with *Session Riding* is to include a *nonce* into every state-changing request. This poses a huge problem towards automated scanning of web applications. Using the usual proceeding with those special requests may result in a decline by the server. The usual proceeding is saving the parameters and replacing certain ones with a penetration vector as described above. In doing so, the nonce either gets overwritten or stays the same as in the request generated by the application before. It depends on the way the Session Riding prevention is implemented on the server-side. There are two common ways this is accomplished. One is to have the nonce generated when a new session is created, which remains unchanged throughout the session. Such implementations do not interfere the scan process, because the once found nonce remains valid as long as the scanning takes place within the same session.

The other one is, that every time a link to a state-changing resource on the server is passed to the client, a new nonce is generated and appended to the link. This nonce is only valid for one request and gets invalidated afterwards. That means all penetration requests generated by this link will not work due to the fact that the server will refuse them as invalid.

To solve this problem it has to be defined what a nonce is and how to detect it. Nonces appear in diverse looks, which makes it hard to automatically determine whether the given parameter is a nonce. It is possible to declare characteristics for a usual nonce, like a minimum length or that it does not contain whitespaces; but the more accurate the definition is, the lower will be the chance to catch all nonces. The other way round, the less accurate the definition is, the more other parameters will be erroneously identified as nonces. The solution to this problem may be incomputable, since not even humans are always able to separate nonces from other parameters. But when having created a sufficiently correct algorithm for identifying nonces, then the proxy could request the page, that contains the desired link and use the

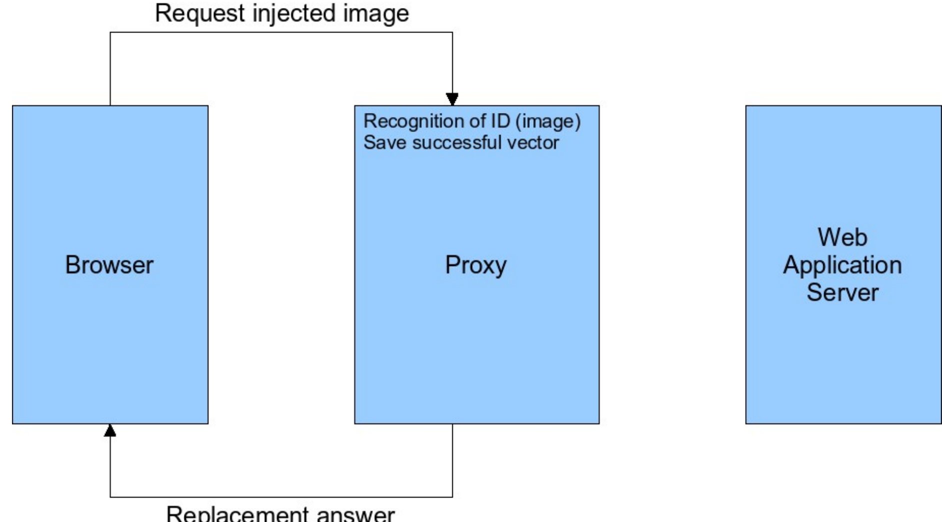
[3] At some point the scanner will have to start to set off the deferred request.

freshly generated nonce.

By developing an algorithm that recognizes nonces, another feature can be gained. Since in a well elaborated web application only `POST` parameters are state-changing, only those have to be augmented with a nonce. So searching for possible Session Riding vulnerabilities within the application would be as easy as searching for requests that contain `POST` parameters, but no nonce.

5.5 Recognizing Success

An axiomatic problem when identifying successful code injections is to decide whether a potential penetration vector would be executed by the web application. Any kind of analysis of the web page's code always depends on the quality of its algorithm. Due to the many different possibilities that lead to XSS vulnerabilities in the produced source code, present proceedings are afflicted with unacceptable false-positives rates that assume a particular vector to be successful when it is not. To advance this problem, the mechanism of purely analyzing the code could be replaced by a method that actually executes the tested injection and then inspects the result, analyzing whether the injection was successful in contrast to the prior approach, that analyzed whether an injection would be successful. This may be achieved by sending a request containing a penetrating vector to the tested web application, route the response back to the browser and make it interpret the returned code. To unerringly determine whether the injection was successful, the scanner has to verify that the result occurred that the injected script was to generate. It is conceivable to inject a script that provokes a request for some kind of resource when being interpreted by the browser. Recognizing that request, the proxy component identifies the vector that provoked this particular request by decoding the necessary information from it. For this purpose, the vectors created by the proxy component may for example hold a unique identification that is encoded in the name of the resource that is requested.



Supplementary, one should consider that vectors that induce invalid HTML code when injected into the tested web application might still be interpreted and run. This is due to the fact that current browsers differ in their tolerance accepting statements that are not conform to the specified HTML document type declaration. Some browsers will for example close opened tags that have not been closed or ignore unexpected special characters. As a consequence, if only the proxy component would analyze the returned web page, it would not notice the successful injection, as the script would not be run. But as browsers will try to patch different levels of non-conform code, some of them will still validate and run the script injected by the *browser specific penetration vector* and thus revealing a vulnerability in the tested web application.

6 The Implementation

6.1 The Main Loop

When starting the scanner via the `Controller` class, several objects are created. A `Web Interface Generator` for creating result, command and status pages, a `Crawler` for parsing retrieved web pages, a `Request Generator` that combines found links with IDs for recognizing successful injections (see section 5.5) and penetration vectors, an `Exception Handler` and a `Proxy` module.

In the main control loop, the `Controller` starts the `Proxy` that listens either on a default port or one specified when starting the scanner. As soon as it receives incoming packages from the browser, it creates a `Request` that, following the command design pattern, classifies itself. The `Proxy`

then returns that Request and the corresponding socket to the Controller, which spawns and starts a Work Thread, handing over references to the received Request and its socket as well as the objects mentioned above. While the Proxy returns to listening for new incoming requests, the Work Thread manages the given Request dependent on its classification.

6.2 Classifications of Requests

Possible classifications of Requests are "FORWARD REQUEST", "NEXT VECTOR", "SENT SOURCE", "STATUS REQUEST", "VECTOR FOUND", "PREINTERFACE", "WEB INTERFACE", "WEB INTERFACE SCRIPT", "FAVICON" and "SHUTDOWN". Future classifications may include "BROWSERTEST FOUND" and "PROHIBITED FILE REQUEST". More on browser tests can be found in section 15.1.

6.2.1 FORWARD REQUEST

If the Work Thread identifies the classification of its Request as "FORWARD REQUEST", it orders the Proxy to transmit it to the web application.

When the Proxy receives the response, it parses it, creates a Response and returns it.

The Work Thread extracts the URL and the GET and POST parameters from the Response and creates a Link, which it then hands over to the Request Generator.

Then the Crawler, which is an implementation of an SGML parser, parses the response (held by the returned Response object) for links and holds them in an internal Link List. That Link List inherits from the standard *Python* List but is optimized for avoiding redundancy.

The Work Thread then gets that list and adds every Link in it to the Request Generator. The inconvenience that the Crawler does not return the resulting Link List immediately is due to the fact that it follows the mentioned SGML parser interface.

Finally the Proxy forwards the Response to the browser.

6.2.2 NEXT VECTOR

In case the browser sends a "NEXT VECTOR" request, the Work Tread tries to fetch a valid response from the web application. For that purpose, it gets a new penetrating Link from the Request Generator and forwards it through the Proxy.

The Response is then checked for HTTP status codes indicating redirections, which are followed directly instead of forwarding them to the browser, making *it* re-request the resource.

Later, further complexity optimizations will check for empty responses, valid content-types and caching at this point. For more information on this subject, see section IV.

As soon as the `Work Thread` received a valid response upon one of its injections, is hands it over to the `Crawler` for parsing contained links, fetches the resulting `Link List` and adds every `Link` in it to the `Request Generator`.

It then replaces JavaScript functions that might prevent the automation of the scan process, such as "`alert()`", that interrupt the program flow and require user interaction.

It then places a JavaScript "`onLoad()`" event in the response's `<body>` that will, when executed, send the interpreted source code back to the proxy for parsing purposes, and finally request a new page.

After the response has been fetched and the `Work Thread` is done with preparing it, it is sent back to the browser.

If there are no more penetrating `Links` left in the `Request Generator` that may be requested from the scanned web application, the `Work Thread` will forward a post analysis of the scan process to the browser (of course containing no `onload()` event that will ask for new vectors).

6.2.3 SENT SOURCE

If the browser receives a response, it interprets it and sends the generated code back to the scanner, so that it is parsed for links, generated from JavaScripts.

If the scanner recognizes such a `SENT SOURCE` request, it will extract the URL of the interpreted page and the code itself from the body of the `POST` request and hand it over to the `Crawler` for parsing.

That way, JavaScript code will be interpreted by the browser and then be sent back to the scanner for analysis instead of being investigated by the scanner immediately with the intention to find links that might be generated by JavaScript.

After the `Crawler` has finished, the `Work Thread` will add newly found `Links` to the `Request Generator` again.

6.2.4 STATUS REQUEST

If the scanner receives one of the "`STATUS REQUEST`"s the browser's web interface spawns regularly, it lets the `Web Interface Generator` create a status report and makes the `Proxy` send it back to the browser.

6.2.5 VECTOR FOUND

As mentioned in section 5.5, successful injections will insert an image into the penetrated page. If the `Proxy` receives the request for such an image and creates a `Request`, that `Request` will classify itself as `VECTOR FOUND`.

If the `Work Thread` then meets such a vector, it will forward a placeholder for the requested resource to the browser, so it will not timeout and continue the program flow.

The scanner will extract the ID specifying the successful vector from the request and add it to the Request Generator, so it is available for the post analysis of the scan.

6.2.6 Controlling the Scanner

If the scanner receives a request for the "PREINTERFACE", the "WEB INTERFACE", the "WEB INTERFACE SCRIPT" the "FAVICON" of an interface or for a SHUTDOWN, it queries the requested resource from the Web Interface Generator and returns it through the Proxy.

In case of a "SHUTDOWN", that means it will return a *good-bye* message and exit the program.

6.3 The Request Generator

The Request Generator runs in an independent thread. Everytime the Work Thread adds a Link, that Link is being appended to a temporary list.

The Request Generator then regularly checks that list for new entries. If there are any, updates its Pages with the new Links.

Pages are URLs without their parameters. So if the new Link points to a known Page, it is only added to the already existing Page; otherwise a new Page will be created. Those Pages are then added to a permanent list.

As soon as the Work Thread requests a penetrating vector from the Request Generator, the generator asks every Page in its permanent list whether it has untested Links. If so, it hands over an ID for recognizing the vector in case it will be successful. The Page then combines the parameters of the untested Link with that ID and penetrating vectors and returns the new vector.

If a Link is marked as potentially ending the session (a *logout link*), it will be deferred and only be returned if there are no other Links left.

6.4 Control Flow

When the scanner has been started, the user may request the resource at "www.wass.xss". The scanner will then return the preinterface, which displays an HTML textfield for entering the domain that is to be scanned.

Sending that information to the scanner will create a "WEB INTERFACE" request. The delivered web interface will additionally request a favicon and a JavaScript that contains a minimum of client-side business logic. During the scan process, it will request the "NEXT VECTOR" and send interpreted source code to the scanner.

The web interface will allow the user to surf through the target domain in a pop-up in order to log in and to semi-guidedly create valid default values for forms, launching requests that will classify itself as "FORWARD REQUEST".

As the browser requests the target and receives the scanner's web interface, as to the browser's Same Origin Policy, the interface is placed on the targeted domain.

If the *Scan* button is clicked, the JavaScript of the web interface will generate a "`NEXT VECTOR`" request. The scanner will test the target domain, crawl the response for new links and return the result after a valid response has been fetched (see section 6.2.2).

The browser will interpret the page's JavaScripts and send the resulting source code home. It will then continue to execute JavaScript event-handlers one by one and everytime send the generated code back, until every handler has been triggered. Due to complexity requirements, combinations of event-handlers are not handled, yet.

Together with the last triggered handler, the web interface requests the "`NEXT VECTOR`".

If a vector was successful, it injected an image containing an ID identifying it. If the request for that injected resource is disengaged, it will classify itself as "`VECTOR FOUND`" (section 6.2.5).

Regularly in between the web interface will launch "`STATUS REQUEST`"s, which are served by the scanner and displayed in the web interface. They contain information about the number of scanned pages and found links as well as other statistical facts.

As soon as there are no more vectors to be tested and the scanner has delivered the post analysis, the user may "`SHUTDOWN`" the scanner via a link in the interface.

7 Evaluation

An evaluation of the current implementation of the scanner made obvious, that despite the enormous precision it works with, there are two major issues with it.

First, the cogging of the crawling and the testing process causes interferences that affect the reliability with which the scanner may differ between links originally generate by the tested application and links that emerged from the testing process. Web applications often filter or sanitize parameter input and therefore it is hard to recognize if it is injected by the scanner.

Second, the duration of the scan process makes it unpractical. Scanning of just initialized content management and contemporary blogging systems might well take more than 24 hours. This behaviour appears, because there is a wide range of vectors, that all have to be tested against all parameters in all parameter combination for every page, resulting in a humongous number of requests, that have to be evaluated.

As a consequence, the following two parts of this work will describe the changes made to the operating mode of the scanner.

Part III
Similarity Examinations of Webpages

8 Introduction

As described above, the approach of injecting penetrating vectors while scanning the affected application implies several disadvantages, including the problem of recognizing vectors that appear on pages of the application that have not been crawled, yet.

To circumvent these problems, the crawling process has to be distinct from the testing phase. The main problem is, that the crawling has to be ended before the testing phase may begin.

The criterion to determine whether the crawling process has finished, is, that there are no more references to ungathered pages on the same domain. To judge whether that criterion has been met, the scanner has to be able to evaluate whether a new page he has crawled is still unknown or corresponds to a known page.

To make up that decision, there has to be a measure of equality. Therefore a bunch of criteria has to be agreed on to define under which circumstances two compared pages will be considered equal.

So for this work, we will consider two documents equal, if they are of the same structure and do contain the same HTML elements that are considered relevant for XSS (section 2), as HTML `<form>` tags for example.

9 The Criteria

To determine whether two pages are similar, different criteria may be applied. The criteria that will be looked at closer in this work are the head data of the examined page and the structure of the document. While the first aspect is dominated by the challenge to recognize similar but not necessarily equal strings, the latter will be divided into bottom-up stages, demanding more and more details of the compared pages to match. The results of the comparisons will then be consolidated again, to valuate them, to finally form a conjoint decision.

9.1 Head Data

While differences in the head data of two pages do not necessarily signify that the pages are unequal as to a web application black box test, compliances hint at the possibility of viewing the same page twice.

Therefore, when classifying a new page, the scanner extracts properties considered characteristic from the head data of the page, creating a list of objects containing information that are assumed to identify the page. It

holds information about the name of the tag that was extracted, as well as its main attributes. The scanner will examine the `<title>` tag and its string content, `<meta>` tags and their attributes as "`name`", "`http-equiv`" and "`content`", `<link>` tags with their "`rel`", "`type`" and "`href`" properties and `<script>`s including information about their "`type`" and their "`src`".

The so gained list is then compared to the previously created lists of every page found so far, deciding which criteria match. We will have a closer look at the procedure implementing the comparison and its evaluation in section 10.

9.2 Comparison of Strings

Especially when examining the `<title>` of a webpage, a just two-valued conclusion stating whether it equals the title of a second webpage that it is being compared to, or immutably does not, appears to be too coarse. To come to a more refined result, the scanner implements an algorithm to determine the Levenshtein Distance of two titles.

The Levenshtein Distance is named after Vladimir Levenshtein, a Russian scientist, who is engaged in research on information theory and error-correcting codes, and who developed the concept in 1965. It can be considered a generalization of the Hamming distance, developed by Richard Hamming, who introduced his idea in a paper about error-detecting and error-correcting codes in 1950.

But whereas the Hamming distance only copes with code words of the same length, and therefore only uses substition, the Levenshtein distance is calculated employing three different transformations and is at least the difference of the lengths of the words that are being compared. By inserting, deleting, or substituting single characters, it calculates the minimum number of operations needed to transform one string into the other.

As an algorithm computing the edit distance of two strings, it is a string metric, indicating a similarity or dissimilarity coefficient for the two strings being compared. For example, the strings "*This is the Homepage of Alice at www.example.com*" and "*This is the Homepage of Bob at www.example.com*" can be considered similar. Acknowledging that only the names in the titles differ, the algorithm transforms "*Alice*" into "*Bob*" by substituting "*A*", "*l*" and "*i*" into "*B*", "*o*" and "*b*" and then deleting "*c*" and "*e*", this example making no use of insertions.

The function implemented by the scanner (figure 13) takes two strings that are to be compared as parameters, calculates their lengths and declares a two dimensional array [4] to compute and hold the result. It is one field wider than the first and one field deeper than the second word.

Taking the words '`kitten`' and '`sitting`' for an example, the (pseudo

[4] that is a list of lists in the employed *Python* programming language

code) array computing the edit distance will be 7 fields (i.e the length of the word 'kitten' plus one) wide and 8 fields deep (`len('sitting') + 1`). So the declaration in pseudo code would look something like "`int[7][8] answerArray;`".

The algorithm then initializes the array by filling the fields of the first row as well as of the first column with its index each (figure 13, lines 09 - 12). The result is as seen in figure 14.

```
01    def levenshteinDistance(self, stringA, stringB):
02        lengthA = len(stringA)
03        lengthB = len(stringB)
04        answerArray = [ ]
05        counter = 0
06        for counter in range(lengthA+1):
07            answerArray.append([0] * (lengthB+1))
08
09        for x in range(lengthA+1):
10            answerArray[x][0] = x
11        for y in range(lengthB+1):
12            answerArray[0][y] = y
13
14        for x in range(1, lengthA+1):
15            for y in range(1, lengthB+1):
16                if stringA[x-1] == stringB[y-1]:
17                    cost = 0
18                else:
19                    cost = 1
20                answerArray[x][y] = min(answerArray[x-1][y] + 1,
22                                        answerArray[x][y-1] + 1,
23                                        anwerArray[x-1][y-1] + cost)
24
25        return answerArray[lengthA][lengthB]
```

Figure 13: Implementation of the Levenshtein Algorithm

To calculate the edit distance of the compared strings, the algorithm computes the values of each field in the initialized array by detecting the minimum of the values of the field left of the current on, the field above the current one and the field left hand above the current one, adding one onto the two first ones each, and adding some cost to the latter one. That cost is being calculated considering whether the characters currently being compared are equal. If so, cost equals zero, one otherwise (figure 13, lines 20 - 23). A visualization of this operation can be found in figure 15.

The existing implementation does so column by column, eventually writing the result of the evaluation into the last field of the answer array (fig-

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1						
i	2						
t	3						
t	4						
i	5						
n	6						
g	7						

Figure 14: The Initialized Answer Array of the Levenshtein Algorithm

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2				
i	2	2	1				
t	3	3	2				
t	4	4					
i	5	5					
n	6	6					
g	7	7					

answerArray[2][4] = min(4+1, 2+1, 3+cost)
cost = “i” == “t” ? 0 : 1

Figure 15: Calculating a Transformation with the Levenshtein Algorithm

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 16: The Calculated Edit Distance of the Levenshtein Algorithm

ure 16).

String metrics, of which the Levenshtein distance is, although compared to its numerous derivates rather rudimentary, the most known one, are currently used in applications as fingerprint analysis, spellchecking, DNA and RNA analysis, image analysis, evidence-based machine learning, database deduplication, data mining and semantic knowledge integration.

9.3 Document Structure

When judging the similarity of two compared webpages, the most evident criterion is the structure of the source code of the two pages. To examine that aspect more precisely, first of all there has to be a defined understanding of what will further on be referred to as the structure of a document.

Looking at modern web applications, the `<div>` element is a universal means for structuring web pages. As described in [Wor99], "*The DIV and SPAN elements, [...] offer a generic mechanism for adding structure to documents. These elements define content to be inline (SPAN) or block-level (DIV) but impose no other presentational idioms on the content. Thus, authors may use these elements in conjunction with style sheets, the lang attribute, etc., to tailor HTML to their own needs and tastes.*"

And as [Rob06] states, "*The* `div` *element is used to identify and label any block-level division of text [...] By marking a section of text as a* `<div>` *and giving it a name by using* `id` *or* `class` *attributes, you are essentially creating a custom HTML element.*"

Therefore, the scanner will apply several means to examine the relations between `<div>` and `<span>` each. The stages of this comparison will build up on one another and apply increasingly strict standards to the document's congruence.

9.3.1 The Document as a List

The most simple way to look at structural elements is just to count them. A little more sophisticated is comparing them to eachother in the sequence in which they appear in the document that is being compared, in addition to accounting their pure number. When comparing them, the most important characteristics are their attributes. The scanner thereby extracts the attributes of all `<div>` and `<span>` elements of an examined document, grouping key-value tuples of related attributes in lists, creating a list of lists representing a ordered data structure holding information about the occurrence of the regarded tags and their attributes.

Comparing these lists, the scanner calculates the level of compliance. This rudimentary and fast examination allows the scanner to assess the approximate concordance upstream. Although even vast differences resulting from this examination do not necessarily indicate that the pages are differ-

ent as to the XSS scanning process, a high accordance is a valid hint that the page might be equal. This knowledge is especially valuable if advanced comparisons fail to recognize this compliance, visible at a lower level.

This can be understood more easily when considering two complexly structured documents which differ in only one level somewhere around the center of the hierarchical DOM tree. Considering the elaborate nesting of the trees representing the documents, any algorithm considering this nesting will find it difficult do match the connected levels of `<div>` and `<span>` elements. On a lower level, abstracting from the interrelations of the tags, on the other hand, the correlations of the two trees might be much more obvious!

It is problematic though, to plainly compare all attributes of the generated lists. The issue with `id` attributes is, that they are designed to be unique: "*The `id` attribute is used to give an element a specific and unique name in the document*" [Rob06]. This might be fine as that clearness is confined to the document and might therefore indicate identical blocks in two documents, but it also may reference semantical units. If considering a web calendar as an example, the `<div>` elements that represent days may carry the date as an ID, making it possible to jump to that part in the code by navigational means. In that scenario, IDs would be unique through all sites of the application, although sites displaying different months would have to be taken as equal.

As "`id`"s do not seem to constitute well as indicators of equality, considering "`class`" attributes may be more rewarding: "*The `class` attribute is used for grouping similar elements. Multiple elements may be assigned the same `class` name, and doing so enables them to be treated similarly*" [Rob06]. Where it is employed, it is a strong indicator that distinguishable sections of the code are semantically equal. Of course that means ignoring information concerning the number of found elements. Just consider a web blog, marking comments to the published entries as `<div class = "comment">` blocks, be it for rendering them with cascading stylesheets. Blog sites displaying different entries are, despite their different number of received comments, equal, of course.

9.3.2 The Structure as a Tree

A more sophisticated way of looking at the structure is to keep the interrelations between structural elements when extracting them. Preserving the interrelations means differentiating between the different levels the tags appear on in the hierarchy of the Document Object Model. If doing so, the resulting tree will be a much more accurate representation of the examined document. Finding equal trees will therefore support the assumption that two documents are equal.

To extract a tree of structural tags from a document, the scanner creates an empty one, and mounts a head element for meeting the XML requirement

that every DOM tree must have exactly one root element. It then marks the head element as the parent for the first match and performs a Depth First Search on the document's DOM.

Depth First Search is an algorithm for passing through tree structures. It pursues the strategy of searching deeper into a tree whenever it is possible. It selects a root element and starts there moving down a branch until it reaches a leaf, that is a node that does not have children. At any given point in time, a vertex is marked as recent. The algorithm explores out of this node, leaving unexamined edges behind. When all edges of the current vertex have been explored, the algorithm starts backtracking, visiting the last node on the current path that had alternative children, opening up new paths. The process continues until all nodes that are reachable from the originally chosen root have been traversed. At that point, a new root element will be chosen and the search process will be activated again, until all vertices have been discovered [CLRS01]. Since the scanner is only going to run across connected graphs, there will be no need to reset the primarily chosen root vertex.

In non-recursive implementations, the algorithm usually adds newly found elements to a stack, which is a data structure that may generally hold an unlimited number of objects, but does only return them in the reverse order of their entry. Stacks usually implement three operations: `void push(element)` to put a new `element` onto the stack, `element pop()` to receive the last added `element` from the stack while removing it, and `element peek()` to return the last `element` without removing it. In current implementations of stacks, there are usually additional methods for reading the length of the stack and other convenient operations.

The scanner, though, does not employ a stack. The information required for backtracking, that is typically gathered from the stack, is already contained in the DOM. The search algorithm therefore does not need to make use of another data structure, but just utilizes the information about its predecessor and its children, that is nodes on the same level, held in each node.

Let us have a short glance at how a Depth-First-Search scans a tree (figure 17).

The search algorithm selects the element "`a`" as the root vertex and starts descending through the tree. Let us agree that the algorithm will always prefer the element on the utmost left, therefore first traveling to "`b`", realizing that there are two children, choosing "`e`" as the unvisited element the farthest on the left. Descending to "`j`", it will find no further children and backtrack to "`e`", descending through "`k`" to "`n`". Arriving at the next leaf, the algorithm will backtrack to "`k`", finding no further unvisited children, backtracking further to "`e`", finding no further unmet paths either, moving up to "`b`" and descending to "`f`". When backtracking through "`b`", finding no unvisited children, the algorithm arrives at "`a`" again and identifies "`c`" as the next

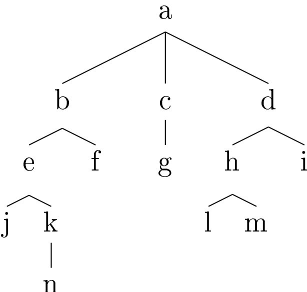


Figure 17: This tree will be scanned by a depth-first-search algorithm

vertex to be examined. After it has descended to "g" and climbed up again, it will traverse the tree underneath "d" in the same manner as it did with the "b" branch, finally, after backtracking through all nodes, arriving back at the root element, having inspected the whole tree.

Back to the scanner. When traversing the tree, everytime the algorithm meets a matching structural element, it adds it as the last child to the marked parent, appends it to a list for tracking purposes (analogue to the stack utilized by a standard non-recursive depth-first-search, although this list operates on the result tree, not on the one being traversed) and resets the current match as the new parent, if it has children (if it has not, it will never be a parent for a following match). It then contines searching down the tree and switches to vertical movement if it reaches a leaf. As soon as there are no deeper knots and there are no further elements on the current level, the depth first algorithm starts backtracking to the next element on the level above. When moving up, the algorithm checks whether is passes a matching element, in order to reset the pointer to the list mentioned above as being maintained 'for tracking purposes'. That way the algorithm keeps track of previously found parents to insert newly found matches into, when moving back up. The scanner will reapply the algorithm for every structural tag type and add the results to a mapping structure, holding the tag type as a key and the extracted tree as the value.

An issue with this approach of extraction is a sort of folding taking place. Imagine a document with a `<body>` consisting of a structural tag, say a `<div id="0">`, and a paragraph `<p>`, the latter one having another `<div id="1">` as its first child, as shown in figure 18. When extracting the `<div>` tags, the scanner will create an empty DOM, insert a root element and start searching through the document. When following down the first branch, having passed the opening `<body>` tag, it will meet the first `<div id="0">`, adding it to the root element, which is at that time referenced as the parent (since no other `<div>` has been found before). Since that first match is a leaf, the search algorithm will continue by moving vertically, then descending when finding that the following paragraph has further children. Then finding the second

```
<html>
   <body>
      <div id="0">
      </div>
      <p>
         <div id="1">
         </div>
      </p>
   </body>
</html>
```

Figure 18: This DOM will be Folded

```
<FOLDED>
   <div id="0">
   </div>
   <div id="1">
   <div>
</FOLDED>
```

Figure 19: The Folded DOM

`<div id="1">`, the algorithm will add it as the last child to the current parent, which still is the root element. Since the first match had no children, it was not set as the next parent (if it had had children, the parent variable would have been reset to the root element when the algorithm ascended). The effect should be obvious by now: Although the two `<div>` tags start on different levels of the hierarchy, the extraction folds the two levels, as the second one is opened by a non-matching `<p>`. The result can be seen in figure 19.

To get over this weakness, the scanner applies a still more restrictive extraction algorithm, replacing non-matching elements with uniform placeholders.

9.3.3 Extending the Tree by using Placeholders

To overcome the habit of folding extracted trees, described in the former section, the scanner employs an again more distinguished algorithm to extract structurally relevant elements from the document's DOM.

This more precise extraction method does not drop non-matching elements that are relevant for the document's structure, but exchanges them with placeholders. This methodology abstracts from irrelevant tags of text formatting but is still sufficiently accurate to capture the aspect of different

hierarchical levels.

To achieve this aim, the scanner performs the known depth-first-search on the document that is to be compared (see figure 20), implementing some changes to meet the new requirements.

The scanner still creates a new, empty DOM at the beginning of the extraction process and populates it with a root element. And when moving down a branch, the algorithm still checks whether the current tag matches the tag type that is presently being searched for. If not, though, the algorithm does not drop it, but converts it to a uniform placeholder, that is, it creates a new tag with a name non-existant in the HTML standard, representing all elements in the examined DOM that were not searched for.

This proceeding would result in a complete copy of the original DOM, plainly substituting non-matches with placeholders, thereby dropping their attributes and content, flattening them (figure 21).

To sharpen this inflated representation, the algorithm evaluates the relevancy of a placeholder when coming across it at its ascension. To do this, the algorithm checks every element in the newly built DOM, when moving up a branch in the original DOM. As long as it only encounters non-matches, it deletes them, therefore stripping loose ends off the represention. As soon as it runs across a match while moving up, it stops deleting placeholders on higher levels and engages in counting the number of levels it crosses, setting the relevant counter to zero (figure 22).

The algorithm will reenable the option of deleting placeholders when descending the next branch, resetting the level counter to zero if it is negative. This secures that no placeholder at levels higher than the root of the new branch will be erased.

The advantage of this more accurate procedure is obvious: It preserves the nesting of the structural elements being extracted. There are some catches, though. If the examined document contains structural elements inside elements serving formatting functionalities, this method will keep representations of these elements as well. As formatting tags are, for example in the context of a web blog, oftentimes under the control of the various users, they may well be handled inconsistently, creating entries that shift `<span>` tags one level down, as they embed them in irrelevant `<b>` or `<i>` tags.

As the scanner at its current developmental state cannot recognize these insignificant shifts yet, it will not consider the documents equal.

Again, this deficiency of a more specific evaluation is a justification for less precise, more abstract examinations. Obviously, the former extraction folding non-matches would not be irritated by any user-dependently inserted tags.

```
<html>
   <head>
      <title>
         6 to 7 million cones provide the eye’s color sensitivity
      </title>
   </head>
   <body>
      <span id="eyecatcher">
         This is
         <i>
            very, very
            <b>
               interesting content
               <blink>
                  !!
               </blink>
            </b>
         </i>
      </span>
      <p>
         But not only is the content of
         <a href="http://www.example.com">
            this page
         </a>
         interesting, it is also
         <b>
            important
            <span class="animatedEmoticons">
               <img src="dancingExclamationMark.gif" alt="!"/>
            </span>
         </b>
      </p>
   </body>
</html>
```

Figure 20: Non-Matching Elements will be Substituted by Placeholders

```
<PLACEHOLDERS>
   <PH>
      <PH>
         <span id="eyecatcher">
            <PH>
               <PH>
                  <PH>
                  </PH>
               </PH>
            </PH>
         </span>
      </PH>
   </PH>
</PLACEHOLDERS>
```

Figure 21: The representation after searching down the second branch, suffering a loose end

```
<PLACEHOLDERS>
   <PH>
      <PH>
         <span id="eyecatcher">
         </span>
         <PH>
            <PH>
               <span class="animatedEmoticons">
               </span>
            </PH>
         </PH>
      </PH>
   </PH>
</PLACEHOLDERS>
```

Figure 22: The Sharpened Representation, only Keeping the Placeholders Necessary to Level the Matches

10 Weighting of Correlations

So far, we have discussed the means available for comparing documents, i.e. for counting correlations or differences.

In order to find a measure that describes similarity, those results have to be weighted and combined. For that purpose, results of the comparison of specific aspects of the documents are multiplied with a factor to influence their impact on the overall equality measure, summed and divided by the number of compared aspects.

In addition to the URLs of two pages, that will always be compared, a judgement upon the equality of two documents will be made, basing on the criteria described above (section 9).

The comparison of the URLs of the two documents being analyzed is very simple, as they are plainly checked for identicalness. As we will see in the description of the comparison of `<title>` tags, there is a much more sophisticated method of matching strings.

Obviously, though, there is no use of a similarity measure for URLs. With the technical specification of a location, there may only be equality or none. Partly resemblances do not carry any meaning with URLs.

When comparing the `<title>` tags of two documents, the algorithm implemented in the scanner will calculate the edit distance, i.e. the number of conversions necessary to transfer one string into the other (section 9.2).

To find a measure for the similarity of two strings, the scanner first of all multiplies the edit distance by 100 to avoid the necessity to deal with chronically imprecise float values.

It then divides the result by the length of the longer word. This makes up for a percental declaration. But as the edit distance indicates the *difference* of two words, the result is being subtracted from 100 to get information about the resemblance of the compared strings.

As described before, the result is then weighted with a factor determining its impact and later added to form an overall similarity measure.

When judging the similarity of the structures of two documents, the scanner firstly checks the lists of extracted HTML tags (section 9.3.1). As there is a list for every structural tag, the scanner compares corresponding lists only.

As lists are ordered sets and the internal comparison of lists in *Python* considers that order, the inspection accounts for the sequence of the occurrence of structural tags. This makes sense as the extracted tags are saved together with their arguments. Otherwise all extracted structural tags within a list would be equal, as they are of the same type.

So after the scanner has counted the number of matching lists, it multiplies it by 100 again to avoid floats and divides it by the number of lists, which is the number of different structural tag types.

The result will be weighted again before flowing into the final calculation.

When comparing the folded trees as well as the trees utilizing placeholders, the comparison is limited to checking for equality, yet. For a planned more sophisticated evaluation, check section 12.

11 Integration into the Scan Process

As described above, before the restructuring, the process of crawling the examined web application was heavily cogged with the penetration testing itself (see section 6).

To divide those two elementary steps, the scanner will first perform a complete crawl through the evaluated web application and will only start injecting potentially penetrating vectors, if that crawling process is done.

When the scan process is initiated by the web interface, the scanner knows about the page marking the starting point and fetches the first resource. In case it receives a valid response, it compares it to all pages known so far, applying the criteria described in section 9. If and only if it is a new page, it is crawled for new links and added to the list of know pages.

The scanner will do so until there are no more unvisited links known and then continue into preparing the actual testing phase.

12 Future Work

To further improve the crawling process, there will be several aspects investigated in greater depth in the future development of the scanner.

First, the extraction of structural tags (section 9.3.1) creates a dict of lists that contain a very flat structure per tag type. For future versions, the extraction process is foreseen to additionally create extracts that combine structural tags of different types.

That will allow for an additional comparison and more detailled information on the document's structure.

The same goes for the current extraction of trees. The present implementation extracts trees of one tag type per time only. In future development, the currently extracted trees will be combined and nested properly, so there will be a very precise definition of the document's structure available to the scanner.

In order to make full use of the gathered information, the algorithm for comparing extracted trees will be heavily modified. So far, it only checks for complete conformity. In future versions, it will have to match corresponding levels, even if they have been shifted against each other by the insertion of an additional level in one tree.

This requirement will demand a highly sophisticated algorithm and will probably hardly match shifts that are more than two or three levels wide.

Nevertheless, being able to match one-level shifts will mean a major improvement to the current state, which purely is highly intolerant to variations.

In addition to the merging of different structural tag types into one tree described further above, there will be an inclusion of non-structural tags that are relevant to XSS testing, such as HTML `<form>` elements, into the extraction process.

This will enable the scanner to recognize documents that differ in elements relevant to XSS injections although their individual structures correspond.

The final and probably most demanding task will be to predict patterns in URL arguments that infinitely lead to equal pages.

Taking the monthly view of a web calendar as an example, every month may contain a link to the next month. If that link takes the number of the next month and the year as parameter values and the application will allow to switch from the current December to the January of the following year, there will be an infinite number of links (as a new value combination of a known link *must* be regarded as a new link, as it may lead into a completely new context).

To avoid the crawling process from not terminating, there has to be a mechanism that recognizes such patterns and stops the tracking of that link.

13 Conclusion

As the results of the described enhancements show, the new approach allows for a basic detachment of the crawling and the testing process.

On the one hand, it enables the scanner to investigate the similarity of selected criteria and to compute an overall measure.

On the other hand, as described in section 12, there are still many options for improving the precision with which the scanner makes those decisions.

And finally, there still is the task of predicting infinite link chains to be solved, to prevent the scanner from losing itself in crawling.

Therefore, the now existing implementation is an ideal stable working base to further refine an independent crawling process, free of side effects and interferences.

Part IV
Complexity Reduction

When scanning a web application for vulnerabilities, Cross-Site-Scripting (XSS) in our case, you come across several problems, which need to get solved, so that the scanner is actually of any use in the real world. One is dealt with in this elaboration. It is the immense complexity that is needed to be coped with scanning a web application. The problem fans out into three sub-units. The first is the crawling process to have a complete overview of the application, without either missing anything nor doing redundant work. Second is the range of vectors that are about to be tested. They need to be chosen in a way that no security flaw is missed, but will not check redundant or unneeded vectors. The third unit is the post analysis, that means that no injected vector is left unnoticed, which would produce false negatives.

14 Crawl Process

14.1 Crawl First

The previously described approach of having the crawling process working during the whole scanning brings up several problems.

A solution that will solve them and make many things much easier is crawling the whole web application first. Thus the links found while the scan progress should be ignored and the previously found pages declared as "whole application". This should be done for a good reason. The injections done while the crawling process may get crippled on the server side when the content gets escaped and sanitized in a way that it is not feasible to do checks if the current link is generated by the injection or is really a new one. Checking if the current parameter constellation contains fragments of the injection can only be accurate if the vector is not altered, because otherwise it may only be speculated if the parameter is only a crippled injection parameter or just similar appearing content originating from the original application.

This behaviour has major impact on the complexity of the scan process, since every new found link has to be tested against every vector with every parameter combination. In most cases not all parameter - vector constellations have to be checked, because supposably only one parameter is affected by the injection, so the link is known on beforehand, with just one differing parameter. Therefor the stained parameter does not have to be tested again with the vectors if duplicate requests are filtered out.

A pretty similar situation may occur in some web application, if the scanner itself generates dynamically new pages, that it needs to crawl thereafter, which may in return generate new ones again and lead to an infinite process. To avoid such endless crawling with no new discoveries, the scan process

should decide to crawl a preferably wide spread amount of pages. Characteristics to divide pages are foremost the path and the file that is accessed in the request, but in detail also the used parameters help to decide either the pages are requesting similar functionality or not.

The most significant drawback of this approach is sometimes referred to as the calendar problem [Gro06]. This means to crawl the web application infinitely (or at least for a very long time) and not starting with the real work, the testing for possible attack vectors. This occurs if an infinite (or very large number) of pages are built dynamically, or if something like a huge news archive is crawled. To deal with that problem the idea of the previous part III was developed.

14.2 Parameter Occurrences (Part I)

After having crawled the whole page the next step is to identify the parameters and categorize them. One part of the category is to search for the appearance of parameter value. There are three possibilities. The first and easiest is, that it does not appear on any page at any time. Second is it is echoed in the resulting page, which lets us conclude that it could be susceptible to reflected or stored XSS. Third is that it appears on a different page. This reveals that the parameter gets stored within the application, which is the first basic condition for reflected XSS.

A pretty straight forward approach to this identification of stored parameters is providing every parameter with a unique identification to the web application. The parameter with the identification as its value needs to be sent with the previously found parameter constellation, since otherwise the context the parameter is embedded in may change. Along with this method of identifying the occurrences of parameters goes the problem of finding those identifications after they have been planted. The most reliable solution is requesting all pages after a parameter has been submitted. But this will most probably result in generating more overhead than the obtained gain in the later scan would be. Requesting all pages after every parameter has been planted will also not lead to the perfect result, because identifiers may get lost while the planting process.

A simple example. The parameters are the fields of a user details page. Now, when filling in an identifier the first time, lets say into the lastname parameter and afterwards filling in the surname parameter, again with the default lastname parameter, the lastname field will get overwritten by the previous default value. That way the identifier would get lost.

This loss of identifiers may already happen while the search for the identifiers, even if the more costly approach is chosen. Because the request of a page that was previously found by the crawler may as well change the content the injected parameter changed, the identifier may get lost during the search procedure.

14.3 Content Types

Another possibility to cut down the number of pages the browser has to evaluate is to forward only pages to the browsers that may contain the desired flaw. In more technical words, the `Content-Type` header has to be set to a value, which represents a content type, the desired flaw may occur in. For XSS the mainly interesting content type is `text/html`. So every page with a content type not within the white list does not need to be sent to the browser.

When looking for vulnerabilities other than XSS one might want to prohibit HTML as a content type, but if the crawling structure is still mixed with the vulnerability testing, this will handicap the crawling of the web application as HTML servers as the linking structure.

14.4 Empty pages

When hitting a page with no content at all, for whatever justification, may it be that the web server did not understand the request or it is just information sent to the browser without the need of a response, which gets used in AJAX applications quite frequently, there is no reason to crawl or forward this page. No additional achievements can be made by forwarding this empty page to the browser for further evaluation, what in return would simply cost more CPU cycles and possibly network traffic.

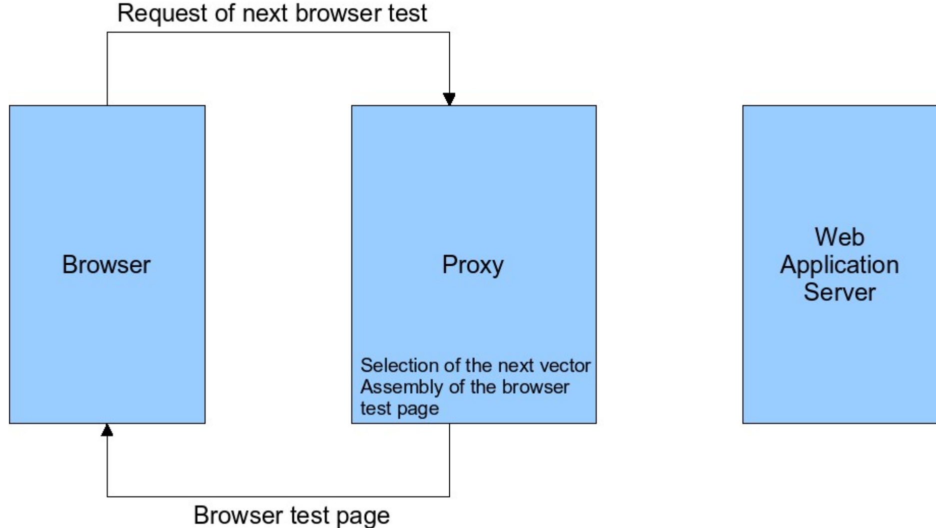
15 Vectors

Vectors are generally the key component for a scanner to be successful finding flaws within a web application. A well elaborated and broad range of vectors is the foundation of good test results. Very often filters and sanitizing functions are used in the real world. But far too frequently those function get developed by people who are not fully aware of possible security threats and why they occur. Or a function is used within the wrong context. In these situations very plain attacks may be prevented, but does not at all mean the web application is secure, just that probably only a trickier vector is needed to circumvent the sanitizing.

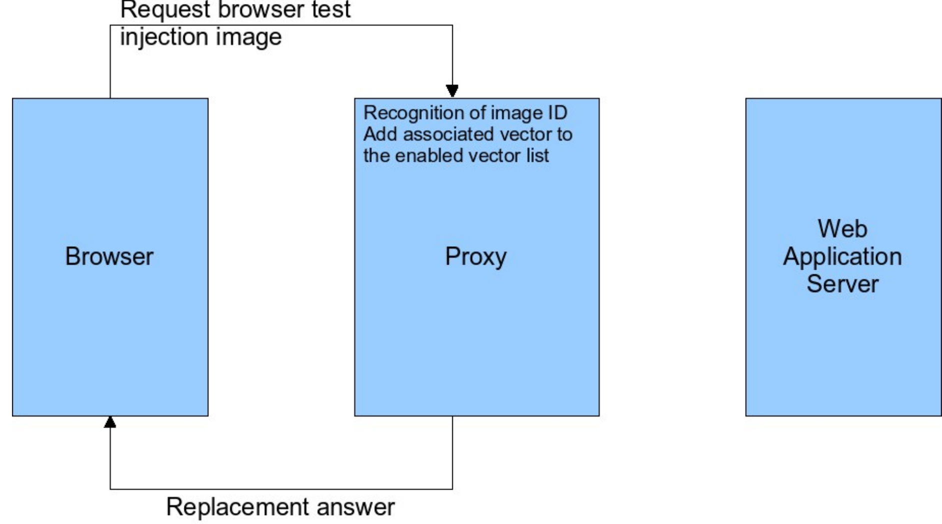
15.1 Browser Tests

First of all there needs to be a huge variety of vectors, which do not for the very same flaw. Many vectors only work for a special browser and probably only within a specific version range. The one way to handle this problem is to have a list of browsers and their version together with which vector is exploitable for it. A second, more persistent solution is doing a pre-test to figure out, which vector actually works for the current, specific browser it

is teamed up with. This saves the injection of each not working vector for every parameter it could have been incorporated with. The procedure is to embed the vector itself into a working environment and send the resulting source code to the browser in order to have him evaluate it.



The vector itself triggers JavaScript that generates a request to an image, just like the procedure described in section 5.5, endued with an ID, to keep apart which vector resulted in the successful execution.



For example the following vector only gets executed as JavaScript in

Firefox and some versions of Netscape, but not Internet Explorer and Opera.

```
<BODY onload!#$%&()*~+-_.,:;?@[/|\]^`=alert("XSS")>
```

This behaviour is explained as:

"*Non-alpha-non-digit part 2 XSS. yawnmoth brought my attention to this vector, based on the same idea as above, however, I expanded on it, using my fuzzer. The Gecko rendering engine allows for any character other than letters, numbers or encapsulation chars (like quotes, angle brackets, etc...) between the event handler and the equals sign, making it easier to bypass cross site scripting blocks. Note that this also applies to the grave accent char as seen here:*" [RSn].

Whereas this example only works in Internet Explorer and some Netscape versions.

```
exp/*<A STYLE='no\xss:noxss("*//*");
xss:&#101;x&#x2F;*XSS*//*/*/pression(alert("XSS"))'>
```

With the following explanation:

"*IMG STYLE with expression (this is really a hybrid of the above XSS vectors, but it really does show how hard STYLE tags can be to parse apart, like above this can send IE into a loop)*" [RSn].

15.2 Filtering Tests

The most wide spread technique to circumvent XSS, filtering and sanitizing, as described in section 2.3. There are many methods of filtering and sanitizing out there with various security implications. The major differences between can be categorized in which character gets dealt with in what way. Examining the circumstances from the vector perspective will come up with the result, that various vectors need different characters to get passed to the page content. Advanced vectors yet inject characters or even character sequences in order to have the web application altering the input into one or more defined characters. If certain characters get filtered or sanitized by the web application specific vector groups are taken out.

This approach reduces, like the browser test, the possible vectors, but in this case on a parameter constellation base. It is definately not clever to do this checking for all characters, however testing for the most frequently by vectors needed characters will reduce the number of injections the scanner has to request. In this step it is important to know which pages the content will occur on.

Furthermore a list of which characters gets filtered on which occurring site can be created while the filter test evaluation. This will ease the search for successfull vulnerability findings futher, because only the pages the current vector is able to occur on need to be tested for successfull exploitation.

15.3 Parameter Occurrence (Part II)

The occurrence of parameters can be additionally interesting, besides the described aspects of section 14.2. Not only the page the parameter value is inserted in is of interest. Also the context the data is residing in can be utilized to limit the number of vectors that have to be tested against a specific parameter.

Vectors usually have one or more determined environments they will work in. Knowing the surrounding areas the parameter data will be placed in reveals that particular vectors do not need to get tested in that parameter constellation.

This is the counterpart to the filtering tests to decrease the number of vectors that have to be tested in a parameter. If the parameter occurrence has already been implemented this check does not need any additional requests or evaluation by the browser. All parts of the logic can be included into parameters occurrence test.

The next example demonstrates this behaviour. A parameter, representing a search string, gets echoed into a text field on the search result page. So trying to break out of the text field content by using `">` does make sense, whereas breaking out of a JavaScript string context by using `";` or something similar does not make sense in that situation and therefore would only produce unwanted overhead in the scanning process.

15.4 Caching

During the crawling process further work can be reduced by implementing a cache. The cache compares the current resulting page with previously cached ones. When hitting a previously crawled page there is no reason to crawl the page again. This methodology increases the performance when crawling also while scanning for vectors immensely, as the architecture was structured in the beginning. But when crawling the application first and scan for flaws afterwards in a separete step, this approach will slow down the process on most web sites.

Caching only makes sense in the vector test process, because web applications often use a default parameter if the given parameter is not within a known range. So the answer to such requests will be always the same, and thus the behaviour of the browser will be the same as well. Such pages need not to be sent back to the browser. Especially if handling a very long list of vectors, this form of caching results in an immense gain of speed, because only pages with differing content as a consequence of an injected parameter are presented to the browser more than once.

Several caching algorithms prevailed over time. The most wide spread ones are the following.

Least Recently Used (LRU)

The least recently used algorithm discards the item that has not been used for longest period of time first. Implementing this means a pretty big overhead. For example having every element attached an identifier when it was last used, all elements need to be checked every time a new item should be added to the cache for the least recently used element to discard it and when hitting a cached item the identifier needs to be updated. Another method would be to have all cache items in an ordered list, where the last used element is at the first position and every time an element is hit it will get placed in the first position of the list. When an element has to be replaced the last element of the list will be discarded. [Tan01]

Most Recently Used (MRU)

The most recently used algorithm is pretty much the opposite of the LRU algorithm. As opposed to LRU in MRU the most recently used items gets replaced first. This method is used when the item, that is to be accessed next, is unpredictable and the determination of the least recently used item is is a high time complexity operation. A common example of this is database memory caches.

Pseudo-LRU/Tree-LRU (PLRU)

The pseudo least recently used algorithm bases upon a binary search tree, which in return entitles the PLRU also as tree least recently used algorithm. The nodes contain a bit that stands for either the pseudo-LRU item is left or it is on the right. The element at the end of this traversal, following the directives of the nodes bits, is the item to be discarded next. Accessing an item X, which is already in the cache, will update the tree. The nodes on the path taken to access the item X have their bits to denote the direction of the next discard changed to the opposite side than the path has to continue to access the item X.

It is used in the CPU cache of the Intel 486 and many processors in the Power Architecture (PowerPC) family.

Least Frequently Used (LFU)

The least frequently used algorithm is rather similar to the least recently used algorithm. Though the parameter that defines which item is going to be discarded first in this case is not the time as in LRU, but it is the frequency. The more often a cache item is hit the less likely this item will be discarded.

Adaptive Replacement Cache (ARC)

The adaptive replacement cache is a mixture of LRU and LFU, to gain better, combined results. This algorithm consists of four ordered lists.

Two of them are implementing the LRU and LFU in its original version. The other two lists are to keep track of the recently discarded items. Now

```
[ LRUdiscard <-][    LRU   <-]![->   LFU    ][-> LFUdiscard ]
[ . . . . . . . [ . . . . . . ^ . . . . . . . ] . . . . . . . . ]
                [     fixed cache size      ]
```

Figure 23: The ARC Algorithm at Default Alignment

```
[ LRUdiscard <-][    LRU   <-]![->   LFU    ][-> LFUdiscard ]
[ . . . . . . . . . . . [ . . . . . . ^ . . . . . . . ] . . . . ]
                        [     fixed cache size      ]
```

Figure 24: The ARC Algoritm Favouring LFU

the real cache is of fixed length, with the size of the original LRU and LFU lists lengths summed up (figure 23).

The clue with this technique is that the real cache is not at a fixed position within these lists, so it may move to the left having items that were actually discarded by the LRU still included within the cache or move to the right, so it would favour LFU and its discarded items (figure 24).

First-In First-Out (FIFO)
FIFO is an easy to implement, low overhead algorithm. It is pretty much what the name suggests. The first element that got added to the ordered list of cached pages is being deleted first when the number of chached elements is reached and another item is to be added. The next one that will be removed is secondly inserted item and so on. [Tan01]

The LFU algorithm is probably the best choice for our assignment, because the most frequent occurrence of this behaviour is when the web application does not understand the request correctly and a error or default page is returned. Now the amount of cache hits of those pages will be significantly larger than these of normal pages, so they will not be replaced by less frequently appearing pages, as their probability of reoccurrence is lower.

16 Analysis

The key element on analyzing the results the test delivers is to not miss anything and not interpret something the wrong way. To catch all successful vectors it is, just as described before in the subsection 14, not sufficient to check on the directly requested page for occurrence, but spider the whole application after the injection process took place. Here again the same problems arise, that previous injections may have gotten overwritten in another

injection step.

Therefore the occurrences of the parameters have been discovered. So now we only need to check the pages the parameter has influence on to check whether the injection was successful or not. This method generates a lot of requests, but checking the occurring pages later on may influence the results to contain more false negatives.

This way the problem of having to find injected content is shifted to the occurrences test. Through this the issue only needs to get addressed once.

17 Conclusion and Perspective

The number of vectors can be wisely reduced. Some of the elaborated methods aim for the reduction of the overall list of vectors, others just for certain parameter combinations. The previous examination of the behaviour of the web application leads to more sophisticated testing and vector injection.

This approach already signalizes the source of the problem, when a vulnerability is encountered. For instance if a vulnerability is encountered an easy solution to this problem might simply be sanitizing the character or characters, if the vector has more than one required to pass successfully, will solve the flaw.

Such behaviour could possibly as well be automated, if the source code is known to the scanner. This course of action is probably not the best approach to cope with security flaws in web applications, because resourceful individuals might still be able to circumvent the filter and sanitization steps taken by the application to prevent this exploitation. If the scanner does not cover all possibilities, which might not only arise as a carelessness of the developer, but also from a new possibility to exploit web applications, it is not able to fix the flaw properly. On the other hand such automation would save time that security people would have to spend fixing this vulnerability and would definitely be an improvement to the security if otherwise no steps would be taken to prevent security issues.

There are two implication that make this approach more uncomely. The first is, that the resulting changes in the representation needs to be sophistacted enough, that it does not change the characters on the representation layer, no user wants to see `<` if he wanted to have a `<`. And it also must not lead to new security implications by introducing new characters that lead to flaws in other subsystems. It might even though break wanted functionality. Imagine a web site like `myspace.com`. MySpace does want the users to be able to change the look of their personal web site as much as they can. They try to only forbid elements which lead to security issues. If the automatically fixing scanner would find a vulnerability and therefore sanitize all characters the found vector depends on. This automated fixing will very likely break most of the functionality.

On the other hand there may be the discovery of code execution, which is intended. Frequently content management systems allow the administrator to introduce JavaScript into pages. But this is wanted behaviour and would disrupt the usage of the application.

So in general before such a fix is implemented the flaw should be examined manually and be decided hereupon.

Before employ this automated fixing it has to be clear to one's mind that this approach does not tackle the source of the problem, but only, maybe not even all, of the consequences.

References

[Ado] External data not accessible outside a macromedia flash movie's domain. http://www.adobe.com/go/tn_14213. [Online; accessed 18-March-2008].

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, September 2001.

[DC05] Nitesh Dhanjani and Justin Clarke. *Network Security Tools*, chapter 8. O'Reilly, 1st edition, April 2005.

[Fla02] David Flanagan. *JavaScript: The Definitve Guide*. O'Reilly, 4th edition, January 2002.

[Fly07] Peter Flynn. The xml faq - what are the special characters in xml? http://xml.silmaril.ie/authors/specials/, August 2007.

[Gro99a] Network Working Group. Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.txt, June 1999.

[Gro99b] Network Working Group. Hypertext transfer protocol – http/1.1: 1 introduction 3 terminology. http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1.3, June 1999.

[Gro99c] Network Working Group. Hypertext transfer protocol – http/1.1: 10 status code definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, June 1999.

[Gro06] Jeremiah Grossman. 5 challenges of web application scanning. http://jeremiahgrossman.blogspot.com/2006/07/5-challenges-of-web-application.html, July 2006. [Online; accessed 20-March-2008].

[GT02] David Gourley and Brian Totty. *HTTP: The Definitive Guide*. O'Reilly, October 2002.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, May 1997.

[Hus04] Sverre H. Huseby. *Innocent code: a security wake-up call for Web programmers*. Wiley Publishing, September 2004.

[Kle05] Amit Klein. Dom based cross site scripting or xss of the third kind: A look at an overlooked flavor of xss. http://

www.webappsec.org/projects/articles/071105.shtml, 2005. [Online; accessed 16-March-2008].

[met] The metasploit project. http://metasploit.com/. [Online; accessed 20-March-2008].

[nik] Nikto 2. http://cirt.net/code/nikto.shtml. [Online; accessed 20-March-2008].

[Rel08] WhiteHat Security Press Release. Whitehat security's latest website security statistics report shows that nine out of ten public websites are still vulnerable to attack. http://www.whitehatsec.com/home/news/08presssarchives/NR_stats032408.html, March 2008. [Online; accessed 27-March-2008].

[Rob06] Jennifer Niederst Robbins. *Web Design in a Nutshell.* O'Reilly, 3rd edition, February 2006.

[RSn] RSnake. Xss (cross site scripting) cheat sheet. http://ha.ckers.org/xss.html. [Online; accessed 19-March-2008].

[Sam05] Samy. Technical explanation of the myspace worm. http://namb.la/popular/tech.html, 2005. [Online; accessed 16-March-2008].

[Sha07] Shreeraj Shah. Crawling ajax-driven web 2.0 applications. http://www.net-security.org/dl/articles/Crawling_Ajax_driven_Web_2.0.pdf, 2007. [Online; accessed 16-March-2008].

[SP08] Dafydd Suttard and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws.* Wiley Publishing, 2008.

[Sut07] Larry Suto. Analyzing the effectiveness and coverage of web application security scanners. http://www.stratdat.com/webscan.pdf, October 2007. [Online; accessed 19-March-2008].

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*, chapter 4.4. Prentice-Hall, 2nd edition, 2001.

[Wik08a] Wikipedia. Cross-site scripting — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cross-site_scripting&oldid=197969424#Types, 2008. [Online; accessed 16-March-2008].

[Wik08b] Wikipedia. Hypertext transfer protocol — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=198896157, 2008. [Online; accessed 18-March-2008].

[Wik08c] Wikipedia. List of http status codes — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=List_of_HTTP_status_codes&oldid=200629031, 2008. [Online; accessed 31-March-2008].

[Wor99] World Wide Web Consortium (W3C). *HTML 4.01 Specification*, December 1999.

[WWSS06] Andreas Wiegenstein, Frederik Weidemann, Dr. Markus Schumacher, and Sebastian Schinzel. Web application vulnerability scanners - a benchmark. http://www.virtualforge.de/whitepapers/web_scanner_benchmark.pdf, October 2006. [Online; accessed 19-March-2008].

[xhr08] Cross-site xmlhttprequest. http://developer.mozilla.org/en/docs/Cross-Site_XMLHttpRequest, February 2008. [Online; accessed 18-March-2008].

[Zel06] Jeffrey Zeldman. *Designing with Web Standards.* New Riders, 2nd edition, August 2006.